

Variables in Python: Usage and Best Practices

by [Leodanis Pozo Ramos](#) Jan 12, 2025 52m 24 Comments basics python

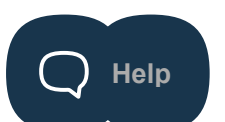
Mark as Completed



Share

Table of Contents

- [Getting to Know Variables in Python](#)
 - [Creating Variables With Assignments](#)
 - [Setting and Changing a Variable's Data Type](#)
- [Working With Variables in Python](#)
 - [Expressions](#)
 - [Object Counters](#)
 - [Accumulators](#)
 - [Temporary Variables](#)
 - [Boolean Flags](#)
 - [Loop Variables](#)
 - [Data Storage Variables](#)
- [Naming Variables in Python](#)
 - [Rules for Naming Variables](#)
 - [Best Practices for Naming Variables](#)
 - [Public and Non-Public Variable Names](#)
 - [Restricted and Discouraged Names](#)
- [Exploring Core Features of Variables](#)
 - [Variables Hold References to Objects](#)
 - [Variables Have Dynamic Types](#)
 - [Variables Can Use Type Hints](#)
- [Using Complementary Ways to Create Variables](#)
 - [Parallel Assignment](#)
 - [Iterable Unpacking](#)
 - [Assignment Expressions](#)
- [Understanding Variable Scopes](#)



- [Global, Local, and Non-Local Variables](#)
- [Class and Instance Variables \(Attributes\)](#)
- [Deleting Variables From Their Scope](#)
- [Conclusion](#)
- [Frequently Asked Questions](#)

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: [Variables in Python](#)

In Python, variables are symbolic names that refer to objects or values stored in your computer’s memory. They allow you to assign descriptive names to data, making it easier to manipulate and reuse values throughout your code. You create a Python variable by assigning a value using the syntax `variable_name = value`.

By the end of this tutorial, you’ll understand that:

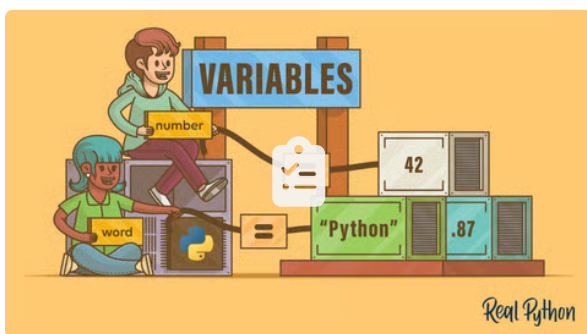
- **Variables** in Python are **symbolic names pointing to objects** or values in memory.
- You define variables by **assigning them a value** using the **assignment operator**.
- Python variables are **dynamically typed**, allowing type changes through reassignment.
- Python variable names can include **letters, digits, and underscores** but can’t start with a digit. You should use **snake case** for multi-word names to improve readability.
- Variables exist in different **scopes** (global, local, non-local, or built-in), which affects how you can access them.
- You can have an **unlimited number** of variables in Python, limited only by computer memory.

To get the most out of this tutorial, you should be familiar with Python’s [basic data types](#) and have a general understanding of programming concepts like [loops](#) and [functions](#).

Don’t worry if you don’t have all this knowledge yet and you’re just getting started. You won’t *need* this knowledge to benefit from working through the early sections of this tutorial.

Get Your Code: [Click here to download the free sample code](#) that shows you how to use variables in Python.

Take the Quiz: Test your knowledge with our interactive “Variables in Python: Usage and Best Practices” quiz. You’ll receive a score upon completion to help you track your learning progress:



Interactive Quiz

[Variables in Python: Usage and Best Practices](#)

In this quiz, you'll test your understanding of variables in Python. Variables are symbolic names that refer to objects or values stored in your computer's memory, and they're essential building blocks for any Python program.

Getting to Know Variables in Python

In Python, [variables](#) are names associated with concrete objects or values stored in your computer’s memory. By associating a variable with a value, you can refer to the value using a descriptive name and reuse it as many times as needed in your code.

Variables behave as if they were the value they refer to. To use variables in your code, you first need to learn how to create them, which is pretty straightforward in Python.

Creating Variables With Assignments

The primary way to create a variable in Python is to **assign** it a value using the assignment operator and the following syntax:

```
variable_name = value
```

In this syntax, you have the variable's name on the left, then the assignment (=) operator, followed by the value you want to assign to the variable at hand. The value in this construct can be any Python object, including [strings](#), [numbers](#), [lists](#), [dictionaries](#), or even custom objects.

Note: To learn more about assignments, check out [Python's Assignment Operator: Write Robust Assignments](#).

Here are a few examples of variables:

Python



```
>>> word = "Python"

>>> number = 42

>>> coefficient = 2.87

>>> fruits = ["apple", "mango", "grape"]

>>> ordinals = {1: "first", 2: "second", 3: "third"}

>>> class SomeCustomClass: pass
>>> instance = SomeCustomClass()
```

In this code, you've defined several variables by assigning values to names. The first five examples include variables that refer to different [built-in types](#). The last example shows that variables can also refer to custom objects like an instance of your `SomeCustomClass` class.

Setting and Changing a Variable's Data Type

Apart from a variable's value, it's also important to consider the **data type** of the value. When you think about a variable's type, you're considering whether the variable refers to a string, integer, floating-point number, list, tuple, dictionary, custom object, or another data type.

Python is a dynamically typed language, which means that variable types are determined and checked at runtime rather than during compilation. Because of this, you don't need to specify a variable's type when you're creating the variable. Python will infer a variable's type from the assigned object.

Note: In Python, variables themselves don't have data types. Instead, the objects that variables reference have types.

For example, consider the following variables:

Python



```
>>> name = "Jane Doe"
>>> age = 19
>>> subjects = ["Math", "English", "Physics", "Chemistry"]

>>> type(name)
<class 'str'>
>>> type(age)
<class 'int'>
>>> type(subjects)
<class 'list'>
```

In this example, `name` refers to the "Jane Doe" value, so the type of `name` is `str`. Similarly, `age` refers to the integer number 19, so its type is `int`. Finally, `subjects` refers to a list, so its type is `list`. Note that you don't have to explicitly tell Python which type each variable is. Python determines and sets the type by checking the type of the assigned value.

Because Python is dynamically typed, you can make variables refer to objects of different data types in different moments just by reassigning the variable:

Python



```
>>> age = "19"
>>> type(age)
<class 'str'>

>>> subjects = {"Math", "English", "Physics", "Chemistry"}
>>> type(subjects)
<class 'set'>
```

Now, `age` refers to a string, and `subjects` refer to a set object. By assigning a value of a different type to an existing variable, you change the variable's type.

Working With Variables in Python

Variables are an essential concept in Python programming. They work as the building blocks for your programs. So far, you've learned the basics of creating variables. In this section, you'll explore different ways to use variables in Python.

You'll start by using variables in expressions. Then, you'll dive into [counters](#) and **accumulators**, which are essential for keeping track of values during iteration. You'll also learn about other common use cases for variables, such as temporary variables, [Boolean](#) flags, loop variables, and data storage variables.

Expressions

In Python, an [expression](#) is a [simple statement](#) that Python can evaluate to produce and return a value. For example, consider the following expressions that compute the circumference of two different circles:

Python



```
>>> 2 * 3.1416 * 10
62.912

>>> 2 * 3.1416 * 20
125.824
```

Each of these expressions represents a specific computation. To build the expressions, you've used values and the multiplication operator (*). Python evaluates each expression and returns the resulting value.

Note: To dive deeper into expressions and operators, check out the [Operators and Expressions in Python](#) tutorial.

The above expressions are sort of rigid. For each expression, you have to repeat the input values, which is an error-prone and repetitive task.

Now consider the following examples:

Python



```
>>> pi = 3.1416
>>> radius = 10

>>> 2 * pi * radius
62.912

>>> radius = 20
>>> 2 * pi * radius
125.824
```

In this example, you first define variables to hold the input values. Then, you use those variables in the expressions. Note that when you build an expression using variables, Python replaces the variable by its value. As shown in the example, you can conveniently reuse the values in different expressions.

Another important point to note is that now you have descriptive names to properly identify the values used in the expressions.

To summarize, variables are great for reusing values in expressions and running computations with data that varies over time. In general, variables let you name or label objects so that you can reference and manipulate them later in the program. In the following sections, you'll see use cases of variables in practice.

Object Counters

A **counter** is an [integer](#) variable that allows you to count objects. Counters typically have an initial value of zero, which increments to reflect the number of times a given object appears. To illustrate, say that you need to count the objects that are strings in a given list of objects. In this situation, you can do something like the following:

Python



```
>>> str_counter = 0

>>> for item in ("Alice", 30, "Programmer", None, True, "Department C"):
...     if isinstance(item, str):
...         str_counter += 1
...

>>> str_counter
3
```

In this example, you create the `str_counter` variable by initializing it to 0. Then, you run a [for](#) loop over a list of objects of different types. Inside the loop, you check whether the current object is a string using the built-in [isinstance\(\)](#) function. If the current object is a string, then you increment the counter by one.

At the end of the loop, `str_counter` has a value of 3, reflecting the number of string objects in the input list.

Note: The highlighted line in the above example uses the expression `str_counter += 1`, which is a shortcut for `str_counter = str_counter + 1`. The `+=` operator is known as the [augmented addition operator](#).

You reuse the expression `str_counter += 1` in each iteration of the loop to increment the value of `str_counter`, making it change over time. This dynamic updating is a key feature of variables. As the name *variables* suggests, they are designed to hold values that can *vary* over time.

Accumulators

Accumulators are another common type of variable used in programming. An accumulator is a variable that you use to add consecutive values to form a total that you can use as an intermediate step in different calculations.

A classic example of an accumulator is when you need to compute the sum of numeric values:

Python



```
>>> numbers = [1, 2, 3, 4]
>>> total = 0

>>> for number in numbers:
...     total += number
...

>>> total
10
```

In this example, the loop iterates over a list of numbers and accumulates each value in `total`. You can also use accumulators as parts of larger computations, for example to calculate the mean of a list of numbers:

Python



```
>>> total / len(numbers)
2.5
```

Then, you take advantage of `total` to compute the average using the built-in `len()` function. You could have used an object counter instead of `len()`. Similarly, Python comes with several [accumulator functions](#) that you can often use instead of explicit accumulators. For example, you can use `sum()` instead of calculating `total` as above.

Temporary Variables

Temporary variables hold intermediate results that you need for a more elaborate computation. A classic use case for a temporary variable is when you need to swap values between variables:

Python

```
>>> a = 5
>>> b = 10

>>> temp = a
>>> a = b
>>> b = temp

>>> a
10
>>> b
5
```

In this example, you use a temporary variable called `temp` to hold the value of `a` so that you can swap values between `a` and `b`. Once you've done the swap, `temp` is no longer needed.

Note: In Python, there's a clean and elegant way to swap values between variables without using temporary variables. You'll learn about this topic in the section on [iterable unpacking](#).

For a more elaborate example of using temporary variables, consider the following function that calculates the [variance](#) of a sample of numeric data:

Python

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

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

The expression that you use as the function's return value is quite involved and challenging to understand. It's also difficult to debug because you're running multiple operations in a single expression.

To make your code easier to understand and debug, you can take advantage of an incremental development approach that uses temporary variables for intermediate calculations:

Python

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

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

In this alternative implementation of `variance()`, you calculate the variance in several steps. Each step is represented by a temporary variable with a meaningful name, making your code more readable.

Boolean Flags

Boolean flags help you manage control flow and decision-making in your programs. As their name suggests, these variables can be either `True` or `False`. You can use them in [conditionals](#), [while](#) loops, and [Boolean expressions](#).

Suppose you need to perform two different actions alternatively in a loop. In this case, you can use a flag variable to toggle actions in every iteration:

Python



```
>>> toggle = True

>>> for _ in range(4):
...     if toggle:
...         print(f"✅ toggle is {toggle}")
...         print("Do something...")
...     else:
...         print(f"❌ toggle is {toggle}")
...         print("Do something else...")
...     toggle = not toggle
...
✅ toggle is True
Do something...
❌ toggle is False
Do something else...
✅ toggle is True
Do something...
❌ toggle is False
Do something else...
```

Every time this loop runs, the conditional checks the value of `toggle` to decide which course of action to take. At the end of the loop, you change the value of `toggle` using the [not](#) operator. The next iteration will run the alternative action.

Flags are also used as function arguments. Consider the following toy example:

Python



```
>>> def greet(name, verbose=False):
...     if verbose:
...         print(f"Hello, {name}! It's great to see you!")
...     else:
...         print(f"Hello, {name}!")
...

>>> greet("Pythonista")
Hello, Pythonista!

>>> greet("Pythonista", verbose=True)
Hello, Pythonista! It's great to see you!
```

In this example, the `verbose` argument is a Boolean variable that lets you decide which greeting message to display.

You'll find that a few Python built-in functions use flag arguments. The [sorted\(\)](#) function is a good example:

Python



```
>>> sorted([4, 2, 7, 5, 1, 6, 3])
[1, 2, 3, 4, 5, 6, 7]

>>> sorted([4, 2, 7, 5, 1, 6, 3], reverse=True)
[7, 6, 5, 4, 3, 2, 1]
```

The `sorted()` function takes an iterable as an argument and returns a list of sorted objects. This function has a `reverse` argument that is a flag and defaults to `False`. If you set this argument to `True`, then you get the objects sorted in reverse order.

In practice, you'll find that Boolean variables are often named using the `is_` or `has_` naming pattern:

Python



```
>>> age = 20

>>> is_adult = age > 18
>>> is_adult
True
```

In this example, the `is_adult` variable is a flag that changes depending on the value of `age`. Note that the `is_` naming pattern allows you to clearly communicate the variable's purpose. However, this naming convention is just a common practice and not a requirement or something that Python reinforces.

Loop Variables

Loop variables help you process data during iteration in `for` loops and sometimes in `while` loops. In a `for` loop, the variable takes the value of the current element in the input iterable each time you go through the loop:


```
Python 

>>> colors = [
...     "red",
...     "orange",
...     "yellow",
...     "green",
...     "blue",
...     "indigo",
...     "violet"
... ]

>>> for color in colors:
...     print(color)
...
red
orange
yellow
green
blue
indigo
violet
```

In this example, you iterate over a list of colors using a `for` loop. The loop variable, `color`, holds the current color in each iteration. This way, you can do something with the current item while you iterate over the data.

Python's `for` loops can have multiple loop variables. For example, say that you want to map each color with its index. In this case, you can do something like the following:

```
Python 

>>> for index, color in enumerate(colors):
...     print(index, color)
...
0 red
1 orange
2 yellow
3 green
4 blue
5 indigo
6 violet
```

In this example, you use the built-in [enumerate\(\)](#) function to generate indices while you iterate over the input data. Note how this loop has two variables. To provide multiple loop variables, you use a comma-separated series of variables.

You can also use variables to control `while` loops. Here's a toy example:

```
Python 
```



```
>>> count = 5

>>> while count:
...     print(count)
...     count -= 1
...
5
4
3
2
1
```

In this example, the loop variable works as a counter that defines the number of iterations. Inside the loop, you use the augmented subtraction operator (`-=`) to update the variable's value.

Data Storage Variables

Data storage variables allow you to work with containers of values, such as lists, tuples, dictionaries, or [sets](#). For example, say that you're writing a [contact book application](#) that uses a list of tuples to store the information of your contacts:

Python



```
>>> contacts = [
...     ("Linda", "111-2222-3333", "linda@example.com"),
...     ("Joe", "111-2222-3333", "joe@example.com"),
...     ("Lara", "111-2222-3333", "lara@example.com"),
...     ("David", "111-2222-3333", "david@example.com"),
...     ("Jane", "111-2222-3333", "jane@example.com"),
... ]
```

This `contacts` variable allows you to manipulate your data using a single and descriptive name. You can use the variables in a loop, for example:

Python



```
>>> for contact in contacts:
...     print(contact)
...
('Linda', '111-2222-3333', 'linda@example.com')
('Joe', '111-2222-3333', 'joe@example.com')
('Lara', '111-2222-3333', 'lara@example.com')
('David', '111-2222-3333', 'david@example.com')
('Jane', '111-2222-3333', 'jane@example.com')

>>> for name, phone, email in contacts:
...     print(phone, name)
...
111-2222-3333 Linda
111-2222-3333 Joe
111-2222-3333 Lara
111-2222-3333 David
111-2222-3333 Jane
```

The first loop iterates over the items in the contact list and prints them as tuples. The second loop uses three loop variables to process every piece of data individually.

Note that inside the loop, you can use the variables as needed. You don't need to use all the variables or use them in the same order. However, when you have an unused variable, then you should name it using an underscore to point out that it's a [throwaway](#) variable.

So, you could rewrite the loop header to look like this: `for name, phone, _ in contacts:`. In this case, the underscore represents the email variable, which you aren't using in the body of the loop.

Naming Variables in Python

The examples you’ve seen so far use short variable names. In practice, variable names should be descriptive to improve the code’s readability, so they can also be longer and include multiple words.

In the following sections, you’ll learn about the rules to follow when creating valid variable names in Python. You’ll also learn best practices for naming variables and other naming-related practices.

Rules for Naming Variables

Variable names in Python can be any length and can consist of uppercase letters (A-Z) and lowercase letters (a-z), digits (0-9), and the underscore character (`_`). The only restriction is that even though a variable name can contain digits, the first character of a variable name can’t be a digit.

Note: Python currently has full [Unicode](#) support, and you can use many unicode characters in variable names. For example, the following variable names are valid:

Python

```
>>> α = 45
>>> β = 90
>>> δ = 180
>>> π = 3.14
```

These variable names may be uncommon in Python code, but they’re completely valid. You can use them in code that performs scientific calculations when you want the code to reflect the notation used in the target discipline.

All of the following are valid variable names:

Python

```
>>> name = "Bob"
>>> year_of_birth = 1970
>>> is_adult = True
```

These variables follow the rules for creating valid variable names in Python. They also follow best naming practices, which you’ll learn about in the next section.

The variable name below doesn’t follow the rules:

Python

```
>>> 1099_filed = False
File "<input>", line 1
    1099_filed = False
      ^
SyntaxError: invalid decimal literal
```

The variable name in this example starts with a number, which isn’t allowed in Python. Therefore, you get a [SyntaxError](#) exception.

It’s important to know that variables are case-sensitive. Lowercase and uppercase letters aren’t treated the same:

Python

```
>>> age = 1
>>> Age = 2
>>> aGe = 3
>>> AGE = 4

>>> age
1
>>> Age
2
>>> aGe
3
>>> AGE
4
```

In this example, Python interprets the names as different and independent variables. So, casing is something to consider when you're creating variable names in Python.

Nothing stops you from creating two different variables in the same program called `age` and `Age`, or, for that matter, `agE`. However, this practice isn't recommended because it can confuse people trying to read your code and even yourself after a while. In general, you should use lowercase letters when creating your variable names.

The use of [underscore](#) characters is also significant. You'll use an underscore to separate multiple words in a variable name:

Python

```
>>> first_name = "John"
>>> pen_color = "red"
>>> next_node = 123
```

In these variable names, you use the underscore character as a separator for multiple words. This is a way to improve the readability of your code by substituting the space character with an underscore. To illustrate this, consider how your variables would look without the underscores:

Python

```
>>> firstname = "John"
>>> pencolor = "red"
>>> nextnode = 123
```

Even though these names are technically valid, they can be challenging to read and understand at a glance. The lack of separation makes it harder to grasp the meaning of each variable quickly, and they require more effort to interpret. Using underscores improves the clarity of your code and makes it more maintainable.

Best Practices for Naming Variables

You should always give a variable a descriptive name that clearly explains the variable's purpose. Sometimes, you can find a single word to name a given variable:

Python

```
>>> temperature = 25
>>> weight = 54.5
>>> message = "Hello, Pythonista!"
```

Variables always refer to concrete objects, so their names should be nouns. You should try to find specific names for your variables that uniquely identify the referred object. Names like `variable`, `data`, or `value` may be too generic. While these names can work for short examples, they're not descriptive enough for production code.

In general, you should avoid single-letter names:

Python

```
>>> t = 25 # Don't do this
>>> temperature = 25 # Do this instead
```

Single-letter names may be hard to decipher, making your code difficult to read, especially when you use them in expressions with other similar names. Of course, there are exceptions. For example, if you're working with nested lists, then you can use single-letter names to identify indices:

Python



```
>>> matrix = [  
...     [9, 3, 8],  
...     [4, 5, 2],  
...     [6, 4, 3],  
... ]  
  
>>> for i in matrix:  
...     for j in i:  
...         print(j)  
...  
9  
3  
8  
4  
5  
2  
6  
4  
3
```

It's common to use letters like `i`, `j`, and `k` to represent indices, so you can use them in the right context. It's also common to use `x`, `y`, and `z` to represent point coordinates, so these are also okay to use.

Using abbreviations to name variables is discouraged, in favor of using the complete name:

Python



```
>>> doc = "Freud" # Don't do this  
>>> doctor = "Freud" # Do this instead
```

It's best practice to use a complete name instead an abbreviated name because it's more readable and clear. However, sometimes abbreviations are okay when they're widely accepted and used:

Python



```
>>> cmd = "python -m pip list"  
>>> msg = "Hello, Pythonista!"
```

In these examples, `cmd` is a commonly used abbreviation for *command* and `msg` is commonly used for *message*. A classic example of a widely used abbreviation in Python is the `c1s` name, which you should use to identify the current class object in a class method.

Sometimes, you need multiple words to build a descriptive variable name. When using multi-word names, you can struggle to read them if there isn't a distinguishable boundary between words:

Python



```
>>> numberofgraduates = 200
```

This variable name is difficult to read. You have to pay close attention to figuring out the words' boundaries so that you can understand what the variable represents.

The most common practices for multi-word variable names are the following:

- **Snake case:** Lowercase words are separated by underscores. For example: `number_of_graduates`.
- **Camel case:** The second and subsequent words are capitalized to make word boundaries easier to see. For example: `numberOfGraduates`.
- **Pascal case:** Similar to camel case, except the first word is also capitalized. For example: `NumberOfGraduates`.

The [Style Guide for Python Code](#), also known as **PEP 8**, contains [naming conventions](#) that list suggested standards for names of different object types. Regarding variables, PEP 8 recommends using the *snake case* style.

When you need multi-word names, it's common to combine an adjective as a qualifier with a noun:

Python

```
>>> initial_temperature = 25
>>> current_file = "marketing_personel.csv"
>>> next_point = (2, 4)
```

In these examples, you create descriptive variable names by combining adjectives and nouns, which can dramatically improve your code's readability. Another point to consider is to avoid multi-word names that start with `my_`, like `my_file`, `my_color`, and so on. The `my_` part doesn't really add anything useful to the name.

Flag variables are another good example of when to use a multi-word variable name:

Python

```
>>> is_authenticated = True
>>> has_permission = False
```

In these examples, you've used an underscore to separate the words, making their boundaries visible and quick to spot.

When it comes to naming lists and dictionaries, you should use plural nouns in most situations:

Python

```
>>> fruits = ["apple", "banana", "cherry"]

>>> colors = {
...     "Red": (255, 0, 0),
...     "Green": (0, 255, 0),
...     "Blue": (0, 0, 255),
...     "Yellow": (255, 255, 0),
...     "Black": (0, 0, 0),
...     "White": (255, 255, 255),
... }
```

Using plural nouns in these examples makes it clear that the variable refers to a container that stores several objects of similar types.

When naming tuples, you should consider that they're commonly used to store objects of different types or meanings. So, it's okay to use singular nouns:

Python

```
>>> color = (255, 0, 0)
>>> row = ("Jane", 25, "Python Dev", "Canada")
```

Although these tuples store multiple objects, they represent a single entity. The first tuple represents an [RGB \(red, green, blue\)](#) color, while the second represents a row in a database table or some other tabular data.

Public and Non-Public Variable Names

A widely used naming convention for variables in Python is to use a leading underscore when you need to communicate that a given variable is what Python defines as [non-public](#). A non-public variable is a variable that shouldn't be used outside its defining module. These variables are for [internal](#) use only:

Python

timeout.py

```
_timeout = 30

def get_timeout():
    return _timeout

def set_timeout(seconds):
    global _timeout
    _timeout = seconds
```

In this module, you have a non-public variable called `_timeout`. Then, you have a couple of functions that work with this variable. However, the variable itself isn't intended to be used outside the containing module.

Restricted and Discouraged Names

Python reserves a small set of words known as [keywords](#) that are part of the language's syntax. To get the list of Python's keywords, go ahead and run the following code:

Python



```
>>> import keyword
>>> keyword.kwlist
[
    'False',
    'None',
    'True',
    'and',
    'as',
    'assert',
    'async',
    ...,
    'yield'
]
```

In most cases, you won't be able to use these words as variable names without getting an error:

Python



```
>>> class = "Business"
File "<input>", line 1
    class = "Business"
      ^
SyntaxError: invalid syntax
```

This behavior is true for most keywords. If you need to use a name that coincides with a keyword, then you can follow PEP 8's recommendation, which is to add a trailing underscore to the name:

Python



```
>>> class_ = "Business"
```

In this example, you've added an underscore character at the end of the keyword, which enables you to use it as a variable name. Even though this convention works, sometimes it's more elegant to do something like the following:

Python



```
>>> passenger_class = "Business"
```

Now, your variable's name is way more descriptive and specific, which improves your code's readability.

There are also [soft keywords](#), which are keywords only in specific contexts. For example, the `match` keyword is only considered a keyword in [structural pattern matching](#).

Because `match` is a soft keyword, you can use it to name variables:

Python



```
>>> import re

>>> text = "Some text containing a number: 123"
>>> match = re.search("123", text)

>>> if match:
...     print("Found a match 😊")
... else:
...     print("No match found 😞")
...
Found a match 😊
```

In this example, you import the `re` module to use [regular expressions](#). This example only searches for a basic expression in a target text. The idea here is to show that `match` can be used as a valid variable name even though it's a keyword.

Another practice that you should avoid is using built-in names to name your variables. To illustrate, say that you're learning about Python lists and run the following code:

```
Python ⌵

>>> list = [1, 2, 3, 4]
>>> list
[1, 2, 3, 4]
```

In this example, you've used `list` as the name for a `list` object containing numeric values. This shadows the original object behind the name, which prevents you from using it in your code:

```
Python ⌵

>>> list(range(10))
Traceback (most recent call last):
...
TypeError: 'list' object is not callable
```

Now, calling `list()` fails because you've overridden the built-in name in your code. So, you're better off avoiding built-in names to define your variables. This practice can make your code fail in different ways.

Note: For a complete list of built-in names, run the following code in an interactive session:

```
Python ⌵

>>> import builtins
>>> dir(builtins)
[
    'ArithmeticError',
    'AssertionError',
    'AttributeError',
    ...
    'tuple',
    'type',
    'vars',
    'zip'
]
```

The list of built-in names is quite long. To make sure that your variable name doesn't shadow one of the names in this list, you can do something like `"name" in dir(builtins)`. If this check returns `True`, then it's best to find a different name.

Note that this recommendation is also valid for names that refer to objects defined in third-party libraries that you use in your code.

Exploring Core Features of Variables

When you start digging deeper into how Python variables work internally, you discover several interesting features worth studying. In the following sections, you'll explore some of the core features of variables so that you can better understand them.

Variables Hold References to Objects

What happens when you create a variable with an assignment? This is an important question in Python because the answer differs from what you'd find in many other programming languages.

Python is an [object-oriented programming](#) language. Every piece of data in a Python program is an object of a specific type or class. Consider this code:

Python

```
>>> 300
300
```

When presented with the statement `300`, Python does the following operations:

1. Creates an integer object
2. Gives it a value of `300`
3. Displays it on the screen

You can see that an integer object is created using the built-in `type()` function:

Python

```
>>> type(300)
<class 'int'>
```

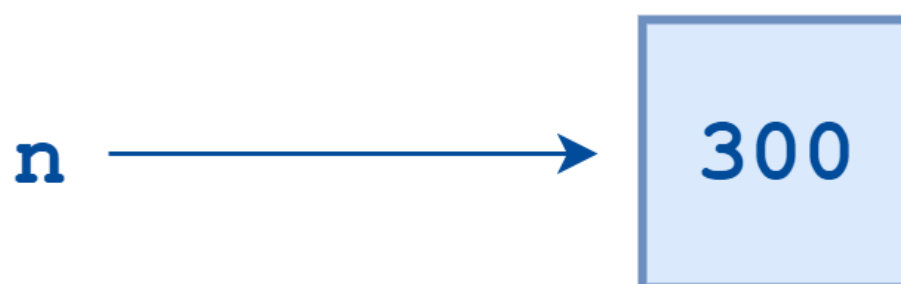
A Python variable is a symbolic name that refers to or [points](#) to an object like `300`. Once an object is assigned to a variable, you can refer to it by the variable's name, but the data itself is still contained within the object.

For example, consider the following variable definition:

Python

```
>>> n = 300
```

This assignment creates an integer object with a value of `300` and makes the variable `n` point to that object. The diagram below shows how this happens:



Variable Assignment

In Python, variables don't store objects. They point or refer to objects. Every time you create an object in Python, it's assigned a unique number, which is then associated with the variable.

The built-in `id()` function returns an object's identifier or identity:

Python

```
>>> n = 300
>>> id(n)
4399012816
```

In [CPython](#), the standard Python distribution, an object's identity coincides with its memory address. Therefore, CPython variables *store memory addresses*. Through these memory addresses, variables access the concrete objects stored in memory.

You can create multiple variables that point to the same object. In other words, variables that hold the same memory address:

Python

```
>>> m = n
>>> id(n) == id(m)
True
```

In this example, Python doesn't create a new object. It creates a new variable name or reference, `m`, which points to the same object that `n` points to:



Multiple References to a Single Object

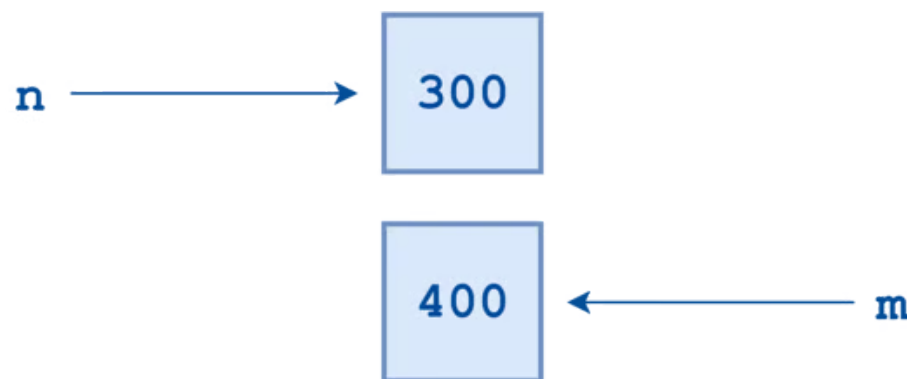
Next, suppose you do something like this:

Python



```
>>> m = 400
```

Now, Python creates a new integer object with the value 400, and `m` becomes a reference to it:



References to Separate Objects

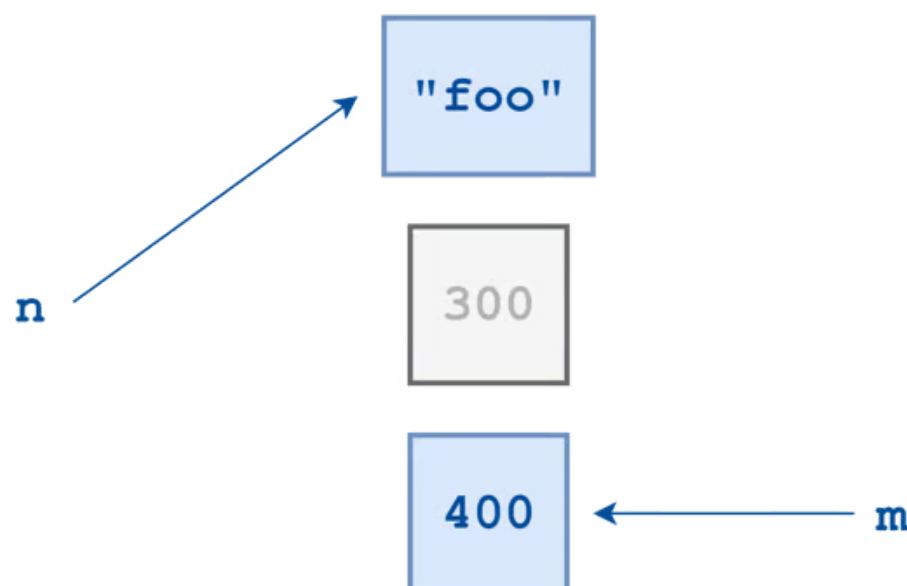
Finally, say that you run the following statement:

Python



```
>>> n = "foo"
```

Now, Python creates a string object with the value "foo" and makes `n` a reference to that:



Orphaned Object

Because of the `n` and `m` reassignments, you no longer have a reference to integer object 300. It's orphaned, and you have no way to access it again.

When the references to an object drop to zero, the object is no longer accessible. At that point, its lifetime is over. Python reclaims the allocated memory so it can be used for something else. In programming terminology, this process is known as [garbage collection](#).

Variables Have Dynamic Types

In many programming languages, variables are **statically typed**, which means they're initially declared to have a specific data type during their lifetime. Any value assigned to that variable during its lifetime must be of the specified data type.

Python variables aren't typed this way. In Python, you can assign values of different data types to a variable at different moments:

Python



```
>>> value = "A string value"
>>> value
'A string value'

>>> # Later
>>> value = 23.5
>>> value
23.5
```

In this example, you're making the `value` variable refer to or point to an object of another type. Because of this feature, Python is a dynamically typed language.

It's important to note that changes in a variable's data type can lead to runtime errors. For example, if a variable's data type changes unexpectedly, you may face type-related bugs or even get an exception:

Python



```
>>> value = "A string value"
>>> value.upper()
'A STRING VALUE'

>>> # Later
>>> value = 23.5

>>> # Try to use .upper() again
>>> value.upper()
Traceback (most recent call last):
...
AttributeError: 'float' object has no attribute 'upper'
```

In this example, the variable type changes during the code's execution. When `value` points to a string, you can use the `.upper()` method to convert the letters into uppercase. However, when the type changes to `float`, the `.upper()` method isn't available, and you get an `AttributeError` exception.

Variables Can Use Type Hints

You can use type hints to add explicit type information to your variables. To do this, you can use the following Python syntax:

Python Syntax

```
variable: data_type [= value]
```

The square brackets aren't part of the syntax. They denote that the enclosed part is optional. Yes, you can declare a Python variable without assigning it a value:

Python



```
>>> number: int

>>> number
Traceback (most recent call last):
...
NameError: name 'number' is not defined
```

The variable declaration on the first line works and is valid Python syntax. However, this declaration doesn't really create a new variable for you. That's why when you try to access the `number` variable, you get a [NameError](#) exception. Even though `number` isn't defined, Python has recorded the type hint:

Python



```
>>> __annotations__  
{'number': <class 'int'>}
```

When it comes to [basic data types](#) such as [numbers](#) and [strings](#), type hints may look superfluous:

Python



```
>>> language: str = "Python"  
>>> number: int = 42  
>>> coefficient: float = 2.87
```

If you're familiar with Python's built-in types, then you won't need to add type hints to these variables because you'll soon know that you have a string, integer, and floating-point value, respectively. So, in this situation, you're okay to skip the type hint. Also, your static type checker or [linter](#) won't complain.

The story changes when you use collection types, such as lists, tuples, dictionaries, and sets. With these types, it makes sense to provide type hints for their contained data.

To illustrate, say that you have the following dictionary of colors:

Python



```
>>> colors = {  
...     "red": "#FF0000",  
...     "green": "#00FF00",  
...     "blue": "#0000FF",  
...     "yellow": "#FFFF00",  
...     "black": "#000000",  
...     "white": "#FFFFFF",  
... }
```

In this case, it'd be great to have information about the data types of keys and values. You can provide that information using the following type hint:

Python



```
>>> colors: dict[str, str] = {  
...     "red": "#FF0000",  
...     "green": "#00FF00",  
...     "blue": "#0000FF",  
...     "yellow": "#FFFF00",  
...     "black": "#000000",  
...     "white": "#FFFFFF",  
... }
```

In this updated code, you're explicitly communicating that your `colors` dictionary will store its keys and values as strings.

Why is this type hint important? Say that in another part of your code, you have another `color` dictionary defined as shown below:

Python



```
>>> colors = {  
...     "Red": (255, 0, 0),  
...     "Green": (0, 255, 0),  
...     "Blue": (0, 0, 255),  
...     "Yellow": (255, 255, 0),  
...     "Black": (0, 0, 0),  
...     "White": (255, 255, 255),  
... }
```

At some point, you might be working with both dictionaries and won't have an unambiguous way to know which dictionary is needed. To work around the issue, you can add a type hint to the second dictionary as well:

Python



```
>>> colors: dict[str, tuple[int, int, int]] = {  
...     "Red": (255, 0, 0),  
...     "Green": (0, 255, 0),  
...     "Blue": (0, 0, 255),  
...     "Yellow": (255, 255, 0),  
...     "Black": (0, 0, 0),  
...     "White": (255, 255, 255),  
... }
```

The type hint in this example is a bit more elaborate, but it clearly states that your dictionary will have string keys and tuples of three integers as values.

It's important to note that type-hinting variables that refer to container data types is especially useful when you initialize the variable with an empty container:

Python



```
>>> fruits: list[str] = []  
>>> rows: list[tuple[str, int, str]] = []
```

In these examples, you have two empty lists. Because they have type hint information, you can quickly figure out the data type of the content. Now you know that the first list will contain strings, while the second list will hold tuples, each consisting of three items: a string, an integer, and another string value.

Note that in this example, your static type checker or linter will explicitly complain about the type hint of your lists if you don't provide any type information.

Later in your code, you can append items to each list using the correct data type:

Python



```
>>> fruits.append("apple")  
>>> fruits.append("orange")  
  
>>> rows.append(("Jane", 25, "Python Dev"))  
>>> rows.append(("John", 30, "Web Dev"))
```

First, you use the `.append()` method to add new fruits as strings to the end of `fruits`, and then you add new rows to the end of `rows`.

Using Complementary Ways to Create Variables

In Python, you'll find a few alternative ways to create new variables. Sometimes, defining several variables simultaneously with the same initial value is convenient or needed. To do this, you can use a [parallel assignment](#).

In other situations, you may need to initialize several variables with values from a sequence data type, like a list or tuple. In this case, you can use a technique called [iterable unpacking](#).

You'll also find situations where you need to retain the value that results from a given expression. In this case, you can use an [assignment expression](#).

In the following sections, you'll learn about all these alternative or complementary ways to create Python variables.

Parallel Assignment

Python also allows you to run multiple assignments in a single line of code. This feature makes it possible to assign the same value to several variables simultaneously:

Python



```
>>> is_authenticated = is_active = is_admin = False

>>> is_authenticated
False
>>> is_active
False
>>> is_admin
False
```

The parallel assignment in this example initializes three different but related variables to `False` simultaneously. This way of creating and initializing variables is more concise and less repetitive than the following:

Python



```
>>> is_authenticated = False
>>> is_active = False
>>> is_admin = False

>>> is_authenticated
False
>>> is_active
False
>>> is_admin
False
```

By using parallel assignments instead of dedicated assignments, you make your code more concise and less repetitive.

Iterable Unpacking

Iterable unpacking is a cool Python feature also known as **tuple unpacking**. It consists of distributing the values in an iterable into a series of variables. In most cases, the number of variables will match the number of items in the iterable. However, you can also use the `*variable` syntax to grab several items in a list.

You can use iterable unpacking to create multiple variables at a time using an iterable of values. For example, say that you have some data about a person and want to create dedicated variables for each piece of data:

Python



```
>>> person = ("Jane", 25, "Python Dev")
```

If you didn't know about iterable unpacking, then your first approach might be to distribute the data into different variables manually, as shown below:

Python



```
>>> name = person[0]
>>> age = person[1]
>>> job = person[2]

>>> name
'Jane'
>>> age
25
>>> job
'Python Dev'
```

This code works. However, using indices to extract the data may lead to an error-prone and hard-to-read result. Instead of using this technique, you can take advantage of iterable unpacking and end up with the following code:

Python



```
>>> name, age, job = person

>>> name
'Jane'
>>> age
25
>>> job
'Python Dev'
```

Now, your code looks cleaner and more readable. So, when you find yourself creating variables from iterables using indices, consider using unpacking instead.

A great use case for unpacking is when you need to swap the values between two variables:

Python

```
>>> a = 5
>>> b = 10

>>> a, b = b, a

>>> a
10
>>> b
5
```

In the highlighted line, you swap the values of `a` and `b` without using a [temporary variable](#), as you saw before. In this example, it's important to note that the iterable to the right of the equal sign is a tuple of variables.

Assignment Expressions

Assignment expressions allow you to assign the result of an expression used in a conditional or a `while` loop to a name in one step. For example, consider the following loop that takes input from the keyboard until you type the word "stop":

Python

```
>>> line = input("Type some text: ")

>>> while line != "stop":
...     print(line)
...     line = input("Type some text: ")
...
Type some text: Python
Python
Type some text: Walrus
Walrus
Type some text: stop
```

This loop works as expected. However, this code has the drawback that it unnecessarily repeats the call to `input()`.

You can rewrite the loop using an assignment expression and end up with the following code:

Python

```
>>> while (line := input("Type some text: ")) != "stop":
...     print(line)
...
Type some text: Python
Python
Type some text: Walrus
Walrus
Type some text: stop
```

In the expression `(line := input("Type some text: "))`, you create a new variable called `line` to hold a reference to the input data. This data is also returned as the expression's result, which is finally compared to the "stop" word to finish the loop. So, assignment expressions are another way to create variables.

To learn more about assignment expressions, check out [The Walrus Operator: Python's Assignment Expressions](#).

Understanding Variable Scopes

The concept of **scope** defines how variables and names are looked up in your code. It determines the visibility of a variable within your code. The scope of a variable depends on the place in your code where you create the variable.

In Python, you may find up to four different scopes, which can be presented using the **LEGB** acronym. The letters in this acronym stand for **local**, **enclosing**, **global**, and **built-in** scopes.

Note: To dive deeper into how scopes work in Python, check out the following tutorials:

- [Python Scope & the LEGB Rule: Resolving Names in Your Code](#)
- [Namespaces and Scope in Python](#)

In the following sections, you'll learn the basics of variable scopes in Python and how this concept can affect the way you use your variables.

Global, Local, and Non-Local Variables

Global variables are those that you create at the module level. These variables are visible within the containing module and in other modules that import them. So, for example, if you're using the Python [REPL](#), then the current global scope is defined in a module called `__main__`.

This module defines your current global scope. All the variables defined in this module are global, so you can use them at anytime during an interactive session run:

Python



```
>>> value = 42
>>> dir()
[
    '__builtins__',
    '__doc__',
    '__loader__',
    '__name__',
    ...
    'value'
]
```

In this code snippet, you define the `value` variable and call the built-in `dir()` function to check the list of names defined in your current global scope. At the end of the list, you'll find the `'value'` entry, which corresponds to your variable.

Now, say that you use this same interactive session to run several pieces of code while you try out some cool features of Python. After all those examples, you need to use the `value` variable again:

Python



```
>>> value
42
```

You'll notice that the variables will still be available for you. That's because this `value` variable is global to your code.

You can also import variables defined outside your current module by using the `import` statement. A typical example of this practice is when you have a module that defines a variable to hold some configuration parameters. You can import this variable into your global scope and use it as you'd use any other global variable.

As a quick example, say that you have the following `config.py` module:

Python

`config.py`

```
settings = {
    "background": "black",
    "foreground": "white",
    "logo": "logo.png",
    "font": "Helvetica",
}
```

This file defines a dictionary that contains several configuration parameters for a hypothetical application. You can import this variable to your app's global scope and use the settings parameters as needed:

Python



```
>>> from config import settings

>>> settings["background"]
'black'
>>> settings["font"]
'Helvetica'
```

With the import in the first line, you've brought the `settings` variable into your current global scope. You can use the variable from any part of your code.

Local variables are those that you define inside a function. These variables can help you store the results of intermediate computations under a descriptive name. They can make your code more readable and explicit.

Consider the following example:

Python



```
>>> def function():
...     integer = 42
...     return integer
...

>>> function()
42

>>> integer
Traceback (most recent call last):
...
NameError: name 'integer' is not defined
```

Local variables are only visible within their defining function. Once the function returns, the variables disappear. That's why you can't access `integer` in the global scope.

Similarly, **non-local variables** are those that you create in a function that define [inner functions](#). The variables local to the outer function are non-local to the inner functions. Non-local variables are useful when you're creating [closure](#) functions and [decorators](#).

Note: To learn more about closures and decorators, you can check out the following tutorials:

- [Python Closures: Common Use Cases and Examples](#)
- [Primer on Python Decorators](#)

Here's a toy example that illustrates how global, local, and non-local variables work and how to identify the different scopes in Python:

Python

`scopes.py`

```
# Global scope
global_variable = "global"

def outer_func():
    # Nonlocal scope
    nonlocal_variable = "nonlocal"
    def inner_func():
        # Local scope
        local_variable = "local"
        print(f"Hi from the '{local_variable}' scope!")
        print(f"Hi from the '{nonlocal_variable}' scope!")
        print(f"Hi from the '{global_variable}' scope!")
    inner_func()
```

In this example, you first create a global variable at the module level. Then, you define a function called `outer_func()`. Inside this function, you have `nonlocal_variable`, which is local to `outer_func()` but non-local to `inner_func()`.

In `inner_func()`, you create another variable called `local_variable`, which is local to the function itself.

Note: You can access global and non-local variables from within an inner function. However, to update a global or non-local variable from within an inner function, you need to explicitly use the `global` and `nonlocal` statements.

The calls to `print()` are intended to show how you can access variables from different scopes inside a function.

Here's how this function works:

Python



```
>>> from scopes import outer_func

>>> outer_func()
Hi from the 'local' scope!
Hi from the 'nonlocal' scope!
Hi from the 'global' scope!
```

In summary, global variables are accessible from every place in your code. Local variables are visible in their defining function. Non-local variables are visible in their defining or enclosing function and in any inner function that lives in the enclosing function.

Class and Instance Variables (Attributes)

You can create variables inside custom Python classes when using **object-oriented programming** tools. These variables are called attributes. In practice, you can have class and instance attributes.

Class attributes are variables that you create at the class level, while **instance attributes** are variables that you attach to instances of a given class. These attributes are visible only from within the class or its instances. So, classes define a namespace, which is similar to the scope.

Note: To dive deeper into object-oriented programming in Python, check out the following tutorials:

- [Object-Oriented Programming \(OOP\) in Python](#)
- [Python Classes: The Power of Object-Oriented Programming](#)

To illustrate how class and instance attributes work, say that you need a class to represent your company's employees. The class should keep information about the employee at hand, which you can store in instance attributes like `.name`, `.position`, and so on. The class should also keep a count of how many employees are currently in the company. To implement this feature, you can use a class attribute.

Here's a possible implementation of your class:

Python

employees.py

```
class Employee:
    count = 0

    def __init__(self, name, position, salary):
        self.name = name
        self.position = position
        self.salary = salary
        Employee.count += 1

    def display_profile(self):
        print(f"Name: {self.name}")
        print(f"Position: {self.position}")
        print(f"Salary: ${self.salary}")
```

In this `Employee` class, you define a class attribute called `count` and initialize it to `0`. You'll use this attribute as a counter to keep track of the number of `Employee` instances. Class attributes like `.count` are common to the class and all its instances.

Inside the [initializer](#), `__init__()`, you define three instance attributes to hold the name, position, and salary of the employee. The last line of code in `__init__()` increments the counter by 1 every time you create a new employee. To do this, you access the class attribute on the class object itself.

Note: You can't change the value of a class attribute using the `self` object. For example, if you do something like `self.count += 1` instead of `Employee.count += 1`, then you create a new instance attribute that shadows the class attribute. You can alternatively access the class attribute with `type(self).count` instead of `Employee.count`.

Finally, you have a method to display the employee's profile according to their current information. This method shows that you need to use the `self` argument to access instance attributes within a class.

Here's how you can use this class in your code:

Python



```
>>> from employees import Employee

>>> jane = Employee("Jane Doe", "Software Engineer", 90000)
>>> john = Employee("John Doe", "Product Manager", 120000)

>>> jane.display_profile()
Name: Jane Doe
Position: Software Engineer
Salary: $90000

>>> john.display_profile()
Name: John Doe
Position: Product Manager
Salary: $120000

>>> f"Total employees: {Employee.count}"
'Total employees: 2'
```

In this code, you first create two instances of `Employee` by calling the class [constructor](#) with appropriate arguments. Then, you call the `display` method on both employees and get the corresponding information. Finally, you access the `.count` attribute on `Employee` to get the current number of employees.

It's important to note that you can access instance attributes like `.name` or `.position` using the dot notation on the target instance:

Python




```
>>> jane.name
'Jane Doe'

>>> john.name
'John Doe'

>>> Employee.name
Traceback (most recent call last):
...
AttributeError: type object 'Employee' has no attribute 'name'
```


Instance attributes are specific to one instance, so you can't access them through the class. In contrast, class attributes are common to the class and all its instances:

```
Python 

>>> jane.count
2
>>> john.count
2

>>> Employee.count
2
```

To access a class attribute, you can either use an instance or the class itself. However, to change a class attribute directly, you need to use the class. Go ahead and give it a try with the following example:

```
Python 

>>> john.count = 100
>>> john.count
100


>>> Employee.count
2

>>> Employee.count = 100
>>> Employee.count
100
```

In this example, you try to update the `count` attribute using an instance, `john`. This action results in you attaching a new instance attribute to `john` instead of updating the value of `Employee.count`. To update the class attribute, you need to use the class itself.

Deleting Variables From Their Scope

In Python, you can explicitly remove variables or, more generically, names from a given scope using the `del` statement:

```
Python 

>>> city = "New York"
>>> city
'New York'

>>> del city
>>> city
Traceback (most recent call last):
...
NameError: name 'city' is not defined
```

In this code snippet, you first create a new variable called `city` in your current global scope. Then, you use the `del` statement to remove the variable from its containing scope. When you try to access the variable again, you get a `NameError` exception.

Note: In practice, you can use the `del` statement in several ways to remove names from scopes and containers. To dive deeper into how this statement works, check out [Python's `del`: Remove References From Scopes and Containers](#).

You should know that while `del` removes the reference to an object, it doesn't necessarily free the memory immediately. Python's garbage collector claims the memory once there are no more references to the object.

Conclusion

You now know the fundamentals of using variables in Python, including how to create and use them in your code. You've learned that Python variables can point to objects of different data types at different moments, which makes Python a dynamically typed language.


You've also learned to use variables in expressions and other common use cases like counters, accumulators, and Boolean flags. Additionally, you've explored best practices for naming variables.

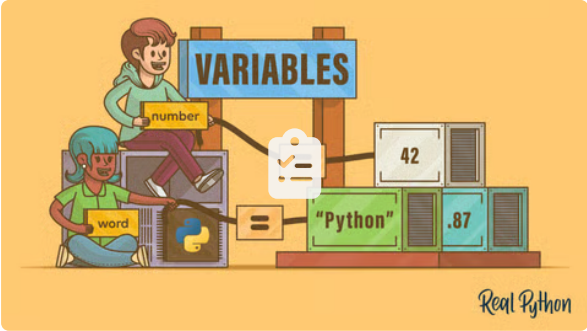
In this tutorial, you've:

- **Created** variables and **assigned** values to them
- **Changed** a variable's **data type** dynamically
- Used variables in **expressions**, **counters**, **accumulators**, and **Boolean flags**
- Followed best practices for **naming** variables
- Created, accessed, and used variables in their specific **scopes**

With these skills, you can now confidently manage data in your Python programs, write readable and maintainable code, and apply best practices to ensure the quality of your code.

Get Your Code: [Click here to download the free sample code](#) that shows you how to use variables in Python.

 **Take the Quiz:** Test your knowledge with our interactive “Variables in Python: Usage and Best Practices” quiz. You'll receive a score upon completion to help you track your learning progress:



Interactive Quiz

[Variables in Python: Usage and Best Practices](#)

In this quiz, you'll test your understanding of variables in Python. Variables are symbolic names that refer to objects or values stored in your computer's memory, and they're essential building blocks for any Python program.

Frequently Asked Questions

Now that you have some experience with Python variables, you can use the questions and answers below to check your understanding and recap what you've learned.

These FAQs are related to the most important concepts you've covered in this tutorial. Click the *Show/Hide* toggle beside each question to reveal the answer.

What are variables in Python?	Show/Hide
How do you set a variable in Python?	Show/Hide
Can you change the data type of a variable in Python?	Show/Hide
Does Python have variable types?	Show/Hide

What are some common use cases for variables in Python?

Show/Hide

What are the four rules for naming Python variables?

Show/Hide

What are best practices for naming variables in Python?

Show/Hide

What is the scope of a variable in Python?

Show/Hide

How many variables can you have in Python?

Show/Hide

Mark as Completed



Share

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: [Variables in Python](#)

About Leodanis Pozo Ramos



Leodanis is a self-taught Python developer, educator, and technical writer with over 10 years of experience.

[» 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](#)



[Brenda](#)



[Dan](#)



[Geir Arne](#)



[Joanna](#)



[John](#)



[Martin](#)

What Do You Think?

Rate this article:



[LinkedIn](#) [Twitter](#) [Bluesky](#) [Facebook](#) [Email](#)

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.

Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. [Get tips for asking good questions](#) and [get answers to common questions in our support portal](#).

Looking for a real-time conversation? Visit the [Real Python Community Chat](#) or join the next [“Office Hours” Live Q&A Session](#). Happy Pythoning!

Keep Learning

Related Topics: [basics](#) [python](#)

Recommended Video Course: [Variables in Python](#)

Related Tutorials:

- [Conditional Statements in Python](#)
- [Basic Data Types in Python: A Quick Exploration](#)
- [Operators and Expressions in Python](#)
- [Python for Loops: The Pythonic Way](#)
- [Defining Your Own Python Function](#)

Learn Python

[Start Here](#)
[Learning Resources](#)
[Code Mentor](#)
[Python Reference](#)
[Support Center](#)

Courses & Paths

[Learning Paths](#)
[Quizzes & Exercises](#)
[Browse Topics](#)
[Workshops](#)
[Books](#)

Community

[Podcast](#)
[Newsletter](#)
[Community Chat](#)
[Office Hours](#)
[Learner Stories](#)

Membership

[Plans & Pricing](#)
[Team Plans](#)
[For Business](#)
[For Schools](#)
[Reviews](#)

Company

[About Us](#)
[Team](#)
[Sponsorships](#)
[Careers](#)
[Press Kit](#)
[Merch](#)



[Privacy Policy](#) · [Terms of Use](#) · [Security](#) · [Contact](#)

♥ Happy Pythoning!

