



Python Constants: Improve Your Code's Maintainability

by Leodanis Pozo Ramos ⌚ Aug 03, 2022 💬 8 Comments 🔖

intermediate python

Mark as Completed



 Tweet

 Share

 Email

Table of Contents

- [Understanding Constants and Variables](#)
 - [What Variables Are](#)
 - [What Constants Are](#)
 - [Why Use Constants](#)
 - [When Use Constants](#)
- [Defining Your Own Constants in Python](#)
 - [User-Defined Constants](#)
 - [Module-Level Dunder Constants](#)
- [Putting Constants Into Action](#)
 - [Replacing Magic Numbers for Readability](#)
 - [Reusing Objects for Maintainability](#)

- Providing Default Argument Values
- Handling Your Constants in a Real-World Project

Improve Your Python

- Creating a Dedicated Module for Constants
- Storing Constants in Configuration Files

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
```

Improve Your Python

...with a fresh  **Python Trick** 

- Useful String and Math Constants
- Type-Annotating Constants
- Defining Strict Constants in Python
 - The `__slots__` Attribute
 - The `@property` Decorator
 - The `namedtuple()` Factory Function
 - The `@dataclass` Decorator
 - The `__setattr__()` Special Method
- Conclusion



 [The Real Python Podcast »](#)

 Remove ads

In programming, the term **constant** refers to names representing values that don't change during a program's execution. Constants are a fundamental concept in programming, and Python developers use them in many cases. However, Python doesn't have a dedicated syntax for defining constants. In practice, Python constants are just **variables** that *never change*.

To prevent programmers from reassigning a name that's supposed to hold a constant, the Python community has adopted a naming convention: *use uppercase letters*. For every Pythonista, it's essential to know what constants are, as well as why and when to use them.

In this tutorial, you'll learn how to:

- Properly **define constants** in Python
- Identify some **built-in constants**
- Use constants to improve your code's **readability**, **reusability**, and **maintainability**
- Apply different approaches to **organize** and **manage** constants in a project
- Use several techniques to make constants **strictly constant** in Python

By learning to define and use constants, you'll dramatically improve your code's readability, maintainability, and reusability.

To learn the most from this tutorial, you'll need basic knowledge of Python [variables](#), [functions](#), [modules](#), [packages](#), and [namespaces](#). You'll also need to know the basics of [object-oriented programming](#) in Python.

Sample Code: [Click here to download sample code](#) that shows you how to use constants in Python.

Understanding Constants and Variables

Variables and **constants** are two historical and fundamental concepts in computer programming. Most programming languages use these concepts to manipulate data and work in an effective and logical fashion.

Variables and constants will probably be present in each project, app, library, or other piece of code that you'll ever write. The question is: what are variables and constants in practice?

Python Tricks The Book

A Buffet of Awesome Python Features

[Get Your Free Sample Chapter](#)



 [Remove ads](#)

What Variables Are

In math, a variable is defined as a symbol that refers to a value or quantity that *can* change over time. In programming, a variable is also a symbol or name typically associated with a memory address containing a value, object, or piece of data. Like in math, the content of a programming variable can change during the execution of the code that defines it.

Variables typically have a descriptive name that's somehow associated with a target value or object. This target value can be of any data type. So, you can use variables to represent [numbers](#), [strings](#), sequences, custom objects, and more.

You can perform two main operations on a variable:

1. **Access** its value
2. **Assign** it a new value

In most programming languages, you can access the value associated with a variable by citing the variable's name in your code. To assign a new value to a given variable, you'll use an [assignment](#) statement, which often consists of the variable's name, an assignment operator, and the desired value.

In practice, you'll find many examples of magnitudes, data, and objects that you can represent as variables. A few examples include temperature, speed, time, and length. Other examples of data that you can treat as variables include the number of registered users in a [web app](#), the number of active characters in a [video game](#), and the number of miles covered by a runner.

What Constants Are

Math also has the concept of **constants**. The term refers to a value or quantity that *never* changes. In programming, constants refer to names associated with values that never change during a program's execution.

Just like variables, programming constants consist of two things: a

name and an associated value. The name will clearly describe what the constant is all about. The value is the concrete expression of the constant itself.

Like with variables, the value associated with a given constant can be of any of data type. So, you can define integer constants, floating-point constants, character constants, string constants, and more.

After you've defined a constant, it'll only allow you to perform a single operation on it. You can only *access* the constant's value but not change it over time. This is different from a variable, which allows you to access its value, but also reassign it.

You'll use constants to represent values that won't change. You'll find lots of these values in your day-to-day programming. A few examples include the speed of light, the number of minutes in an hour, and the name of a project's root folder.

Why Use Constants

In most programming languages, constants protect you from accidentally changing their values somewhere in the code when you're coding at two in the morning, causing unexpected and hard-to-debug errors. Constants also help you make your code more readable and maintainable.

Some advantages of *using constants* instead of *using their values directly* in your code include:

Advantage	Description
Improved readability	A descriptive name representing a given value throughout a program is always more readable and explicit than the bare-bones value itself. For example, it's easier to read and understand a constant named <code>MAX_SPEED</code> than the concrete speed value itself.
Clear	Most people will assume that <code>3.14</code> may refer

Advantage	Description
communication of intent	to the <code>Pi</code> constant. However, using the <code>Pi</code> , <code>pi</code> , or <code>PI</code> name will communicate your intent more clearly than using the value directly. This practice will allow other developers to understand your code quickly and accurately.
Better maintainability	Constants enable you to use the same name to identify the same value throughout your code. If you need to update the constant's value, then you don't have to change every instance of the value. You just have to change the value in a single place: the constant definition. This improves your code's maintainability.
Lower risk of errors	A constant representing a given value throughout a program is less error-prone than several explicit instances of the value. Say that you use different precision levels for <code>Pi</code> depending on your target calculations. You've explicitly used the values with the required precision for every calculation. If you need to change the precision in a set of calculations, then replacing the values can be error-prone because you can end up changing the wrong values. It's safer to create different constants for different precision levels and change the code in a single place.
Reduced debugging needs	Constants will remain unchanged during the program's lifetime. Because they'll always have the same value, they shouldn't cause errors and bugs. This feature may not be necessary in small projects, but it may be crucial in large projects with multiple developers. Developers won't have to invest

Advantage	Description
	time debugging the current value of any constant.
Thread-safe data storage	Constants can only be accessed, not written. This feature makes them thread-safe objects, which means that several threads can simultaneously use a constant without the risk of corrupting or losing the underlying data.

As you've learned in this table, constants are an important concept in programming for good reason. They can make your life more pleasant and your code more reliable, maintainable, and readable. Now, when should you use constants?

When Use Constants

Life, and particularly science, is full of examples of constant values, or values that never change. A few examples include:

- **3.141592653589793**: A constant denoted by π , spelled as *Pi* in English, which represents the ratio of a circle's [circumference](#) to its diameter
- **2.718281828459045**: A constant denoted by *e* and known as [Euler's number](#), which is closely related to the [natural logarithm](#) and [compound interest](#)
- **3,600**: The number of seconds in one hour, which is considered constant in most applications, even though [leap seconds](#) sometimes are added to account for variability in the Earth's rotation speed
- **-273.15**: A constant representing [absolute zero](#) in degrees Celsius, which is equal to 0 kelvins on the [Kelvin](#) temperature scale

All the above examples are constant values that people commonly

use in life and science. In programming, you'll often find yourself dealing with these and many other similar values that you can consider and treat as constants.

In summary, use a constant to represent a quantity, magnitude, object, parameter, or any other piece of data that's supposed to remain unchanged during its lifetime.

Write Cleaner & More Pythonic Code

realpython.com



 Remove ads

Defining Your Own Constants in Python

Up to this point, you've learned about constants as a general concept in life, science, and programming. Now it's time to learn how Python deals with constants. First, you should know that Python doesn't have a dedicated syntax for defining constants.

In other words, Python doesn't have constants in the strict sense of the word. It only has variables, primarily because of its dynamic nature. Therefore, to have a constant in Python, you need to define a *variable that never changes* and stick to that behavior by avoiding assignment operations on the variable itself.

Note: In this section, you'll focus on *defining* your own constants. However, there are a few constants that are built into Python. You'll learn about them [later on](#).

Then, how would Python developers know that a given variable represents a constant? The Python community has decided to use a strong **naming convention** to distinguish between variables and constants. Keep reading to learn more!

User-Defined Constants

To tell other programmers that a given value should be *treated as a constant*, you must use a widely accepted naming convention for the constant's identifier or name. You should write the name in capital letters with underscores separating words, as stated in the [Constants](#) section of [PEP 8](#).

Here are a few example of user-defined Python constants:

Python

```
PI = 3.14
MAX_SPEED = 300
DEFAULT_COLOR = "\033[1;34m"
WIDTH = 20
API_TOKEN = "593086396372"
BASE_URL = "https://api.example.com"
DEFAULT_TIMEOUT = 5
ALLOWED_BUILTINS = ("sum", "max", "min", "abs")
INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    ...
]
```

Note that you've created these constants exactly as you'd create variables. You've used a descriptive name, the assignment [operator](#) (`=`), and the constant's specific value.

By using capital letters only, you're communicating that the current name is intended to be treated as a constant—or more precisely, as a variable that never changes. So, other Python developers will know that and hopefully won't perform any assignment operation on the variable at hand.

Note: Again, Python doesn't support constants or non-reassignable names. Using uppercase letters is just a convention, and it doesn't prevent developers from assigning new values to your constant. So, any programmer working on your code needs to be careful and never write code that changes the values of constants. Remember this rule because you also need to follow it.

Because Python constants are just variables, both follow similar naming rules, with the only distinction being that constants use uppercase letters only. Following this idea, constants' names can:

- Be of any length
- Consist of uppercase letters (A–Z)
- Include digits (0–9) but not as their first character
- Use underscore characters (`_`) to separate words or as their first character

Using uppercase letters makes your constants stand out from your variables. This way, other developers will unambiguously recognize their purpose.

As a general naming recommendation, avoid abbreviated names when defining constants. The purpose of a constant's name is to *clarify the meaning* of the constant's value so that you can reuse it later. This goal demands descriptive names. Avoid using single-letter names, uncommon abbreviations, and generic names like `NUMBER` or `MAGNITUDE`.

The recommended practice is to define constants at the top of any `.py` file right after any `import` statements. This way, people reading your code will immediately know the constants' purpose and expected treatment.

Module-Level Dunder Constants

Module-level dunder names are special names that start and end with a double underscore. Some examples include names such as `__all__`, `__author__`, and `__version__`. These names are typically treated as constants in Python projects.

Note: In Python, a **dunder name** is a name with special meaning. It starts and ends in double underscores, and the word *dunder* is a **portmanteau** of **double** **underscore**.

According to Python's coding style guide, [PEP 8](#), module-level dunder names should appear after the module's [docstring](#) and before any `import` statement except for `__future__` imports.

Here's a sample module that includes a bunch of dunder names:

Python

```
# greeting.py

"""This module defines some module-level dunder names."""

from __future__ import barry_as_FLUFL

__all__ = ["greet"]
__author__ = "Real Python"
__version__ = "0.1.0"

import sys

def greet(name="World"):
    print(f"Hello, {name}!")
    print(f"Greetings from version: {__version__}!")
    print(f"Yours, {__author__}!")
```

In this example, `__all__` defines up front the list of names that Python will import when you use the `from module import *` import construct in your code. In this case, someone importing `greeting` with a wildcard import will just get the `greet()` function back. They won't have access to `__author__`, `__version__`, and other names not listed on `__all__`.

Note: The `from module import *` construct allows you to import all the names defined in a given module in one go. The `__all__` attribute restricts the imported names to only those in the underlying list.

The Python community strongly [discourages](#) this `import` construct, commonly known as **wildcard imports**, because it tends to clutter your current [namespace](#) with names that you probably won't use in your code.

In contrast, `__author__` and `__version__` have meaning only for the code's authors and users rather than for the code's logic itself. These names should be treated as constants since no code should be allowed to change the author or version during the program's execution.

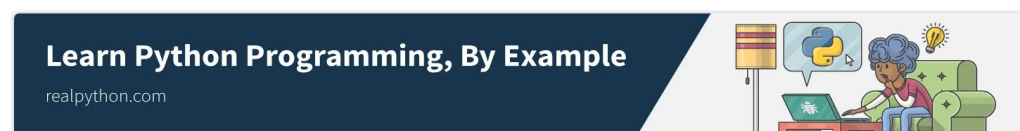
Note that the `greet()` function does access the dunder names but doesn't change them. Here's how `greet()` works in practice:

```
Python >>>

>>> from greeting import *

>>> greet()
Hello, World!
Greetings from version: 0.1.0!
Yours, Real Python!
```

In general, there are no hard rules that prevent you from defining your own module-level dunder names. However, the Python documentation strongly [warns](#) against using dunder names other than those generally accepted and used by the community. The core developers may introduce new dunder names to the language in the future without any warning.



 Remove ads

Putting Constants Into Action

So far, you've learned about constants and their role and importance in programming. You've also learned that Python doesn't support strict constants. That's why you can think of constants as variables that never change.

In the following sections, you'll code examples of how valuable constants can be in your day-to-day coding adventure.

Replacing Magic Numbers for Readability

In programming, the term [magic number](#) refers to any number that appears directly in your code without any explanation. It's a value that comes out of the blue, making your code enigmatic and difficult to understand. Magic numbers also makes programs less readable and more difficult to maintain and update.

For example, say you have the following function:

Python

```
def compute_net_salary(hours):  
    return hours * 35 * (1 - (0.04 + 0.1))
```

Can you tell up front what the meaning of each number in this computation is? Probably not. The different numbers in this function are magic numbers because you can't reliably infer their meanings from the numbers themselves.

Check out the following [refactored](#) version of this function:

Python

```
HOURLY_SALARY = 35  
SOCIAL_SECURITY_TAX_RATE = 0.04  
FEDERAL_TAX_RATE = 0.10  
  
def compute_net_salary(hours):  
    return (  
        hours  
        * HOURLY_SALARY  
        * (1 - (SOCIAL_SECURITY_TAX_RATE + FEDERAL_TAX_RATE))  
    )
```

With these minor updates, your function now reads like a charm. You and any other developers reading your code can surely tell what this function does because you've replaced the original magic numbers with appropriately named constants. The name of each constant clearly explains its corresponding meaning.

Every time you find yourself using a magic number, take the time to

replace it with a constant. This constant's name must be descriptive and unambiguously explain the meaning of the target magic number. This practice will automatically improve the readability of your code.

Reusing Objects for Maintainability

Another everyday use case of constants is when you have a given value repeatedly appearing in different parts of your code. If you insert the concrete value into the code at every required place, then you'll be in trouble if you ever need to change the value for any reason. In this situation, you'll need to change the value in every place.

Changing the target value in multiple places at a time is error-prone. Even if you rely on your editor's *Find and Replace* feature, you can leave some unchanged instances of the value, which can lead to unexpected bugs and weird behaviors later.

To prevent these annoying issues, you can replace the value with a properly named constant. This will allow you to set the value once and repeat it in as many locations as needed. If you ever need to change the constant's value, then you just have to change it in a single place: the constant definition.

For example, say you're writing a `Circle` class, and you need methods to compute the circle's area, perimeter, and so on. After a few coding minutes, you end up with the following class:

Python

```
# circle.py

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius**2

    def perimeter(self):
        return 2 * 3.14 * self.radius
```

```
def projected_volume(self):  
    return 4/3 * 3.14 * self.radius**3  
  
def __repr__(self):  
    return f'{self.__class__.__name__}(radius={self.radius})'
```

This example uncovers how the approximate value of Pi (3.14) has been written as a magic number in several methods of your `Circle` class. Why is this practice a problem? Say you need to increase the precision of Pi. Then you'll have to manually change the value in at least three different places, which is tedious and error-prone, making your code difficult to maintain.

Note: Generally, you don't need to define Pi yourself. Python ships with some built-in constants, including Pi. You'll see how to take advantage of it [later](#).

Using a named constant to store the value of Pi is an excellent approach to solving these issues. Here's an enhanced version of the above code:

Python

```
# circle.py  
  
PI = 3.14  
  
class Circle:  
    def __init__(self, radius):  
        self.radius = radius  
  
    def area(self):  
        return PI * self.radius**2  
  
    def perimeter(self):  
        return 2 * PI * self.radius  
  
    def projected_volume(self):  
        return 4/3 * PI * self.radius**3  
  
    def __repr__(self):  
        return f'{self.__class__.__name__}(radius={self.radius})'
```

This version of `Circle` uses the global constant `PI` to replace the magic number. This code has several advantages compared to the original code. If you need to increase the precision of `Pi`, then you just have to update the `PI` constant's value at the beginning of the file. This update will immediately reflect on the rest of the code without requiring any additional action on your side.

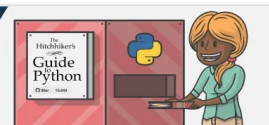
Note: Constants shouldn't change during your code's execution. However, during development, you can change and tweak your constants according to your needs. Updating the precision of `Pi` in your `Circle` class is a good example of why you may need to change the value of a constant during the development of your code.

Another advantage is that now your code is more readable and easier to understand. The constant's name is self-explanatory and reflects the accepted math terminology.

Declaring a constant once and then reusing it several times, as you did in the above example, represents a significant maintainability improvement. If you ever have to update the constant's value, then you'll update it a single place rather than in multiple places, which implies way less effort and error risk.

A Python Best Practices Handbook

python-guide.org



 Remove ads

Providing Default Argument Values

Using named constants to provide default argument values to functions, methods, and classes is another common practice in Python. There are lots of examples of this practice in the Python [standard library](#).

For example, the `zipfile` module provides tools to create, read, write, append, and list ZIP files. The most relevant class in this

module is `ZipFile`. With `ZipFile`, you can manipulate your ZIP files efficiently and quickly.

The `class constructor` of `ZipFile` takes an argument called `compression`, which allows you to select among a few available data compression methods. This argument is `optional` and has `ZIP_STORED` as its default value, meaning that `ZipFile` doesn't compress the input data by default.

In this example, `ZIP_STORED` is a constant defined in `zipfile`. The constant holds a numeric value for uncompressed data. You'll also find other compression methods represented by named constants like `ZIP_DEFLATED` for the `Deflate` compression algorithm, for example.

The `compression` argument in the `ZipFile` class constructor is a good example of using constants to provide default argument values when you have an argument that can take only a limited number of valid values.

Another example of when constants come in handy as default argument values is when you have several functions with a recurrent argument. Say that you're developing an application that connects to a local `SQLite` database. Your app uses the following set of functions to manage the database:

Python

```
import sqlite3
from sqlite3 import Error

def create_database(db_path):
    # Code to create the initial database goes here...

def create_connection(db_path):
    # Code to create a database connection goes here...

def backup_database(db_path):
    # Code to back up the database goes here...
```

These functions perform different actions on your `SQLite` database. Note that all your functions share the `db_path` argument.

While you're developing the application, you decide to provide a default database path to your functions so that you can quickly test them. In this case, you can directly use the path as a default value to the `db_path` argument.

However, it's better to use a named constant to provide the default database path:

Python

```
import sqlite3
from sqlite3 import Error

DEFAULT_DB_PATH = "/path/to/database.sqlite"

def create_database(db_path=DEFAULT_DB_PATH):
    # Code to create the initial database goes here...

def create_connection(db_path=DEFAULT_DB_PATH):
    # Code to create a database connection goes here...

def backup_database(db_path=DEFAULT_DB_PATH):
    # Code to back up the database goes here...
```

This small update enables you to quickly test your app by targeting a sample database during development. It also improves the maintainability of your code because you can reuse this constant in other database-related functions that appear in future versions of your app.

Finally, you'll find situations in which you want to pass an object with certain behavior to a class, method, or function. This practice is typically known as [duck typing](#) and is a fundamental principle in Python. Now say that your code will take care of providing a standard implementation of the required object. If your users want a custom object, then they should provide it themselves.

In this situation, you can use a constant to define the default object and then pass this constant as a default argument value to the target class, method, or function. Check out the following example of a hypothetical `FileReader` class:

Python

```
# file_handler.py

from readers import DEFAULT_READER

class FileHandler:
    def __init__(self, file, reader=DEFAULT_READER):
        self._file = file
        self._reader = reader

    def read(self):
        self._reader.read(self._file)

    # FileHandler implementation goes here...
```

This class provides a way to manipulate different types of files. The `.read()` method uses the injected reader object to read the input file according to its specific format.

Here's a toy implementation of a reader class:

Python

```
# readers.py

class _DefaultReader:
    def read(self, file):
        with open(file, mode="r", encoding="utf-8") as file_obj:
            for line in file_obj:
                print(line)

DEFAULT_READER = _DefaultReader()
```

The `.read()` method in this example takes the path to a file, opens it, and prints its content to the screen line by line. This class will play the role of your default reader. The final step is to create a constant, `DEFAULT_READER`, to store an instance of your default reader. That's it! You have a class that processes the input files and also a helper class that provides the default reader.

Your users can also code custom readers. For example, they can code readers for [CSV](#) and [JSON](#) files. Once they've written a given

reader, they can pass it into the `FileHandler` class constructor and use the resulting instance to handle files that use the reader's target file format.



 Remove ads

Handling Your Constants in a Real-World Project

Now that you know how to create constants in Python, it's time to learn how to handle and organize them in a real-world project. You can use several approaches or strategies to this end. For example, you can put your constants in:

- The **same file as the code** that uses them
- A **dedicated module** for project-wide constants
- A **configuration file**
- Some **environment variables**

In the following sections, you'll write some practical examples that demonstrate the above strategies for managing constants appropriately.

Putting Constants Together With Related Code

The first and maybe most natural strategy to organize and manage your constants is to define them together with the code that uses them. With this approach, you'll be defining the constants at the top of the module that contains the related code.

For example, say that you're creating a custom module to perform calculations, and you need to use math constants like Pi, Euler's number, and a few others. In this case, you can do something like this:

Python

```
# calculations.py

"""This module implements custom calculations."""

# Imports go here...
import numpy as np

# Constants go here...
PI = 3.141592653589793
EULER_NUMBER = 2.718281828459045
TAU = 6.283185307179586

# Your custom calculations start here...
def circular_land_area(radius):
    return PI * radius**2

def future_value(present_value, interest_rate, years):
    return present_value * EULER_NUMBER ** (interest_rate * years)

# ...
```

In this example, you define your constants in the same module where the code using them lives.

Note: If you want to explicitly communicate that a constant should be used in its containing module only, then you can add a leading underscore (`_`) to its name. For example, you can do something like `_PI = 3.141592653589793`. This leading underscore labels the name as **non-public**, which means that the user's code shouldn't use this name directly.

Putting your constants together with the code that uses them is a quick and appropriate strategy for narrow-scope constants that are only relevant to a single module in a given project. In this case, you probably won't be using the constants outside the containing module itself.

Creating a Dedicated Module for Constants

Another common strategy for organizing and managing your

constants is creating a **dedicated module** in which to put them all. This strategy is appropriate for constants that are used in many modules and even packages across a given project.

The central idea of this strategy is to create an intuitive and unique namespace for constants. To apply this strategy to your calculations example, you can create a Python package containing the following files:

```
calc/  
├─ __init__.py  
├─ calculations.py  
└─ constants.py
```

The `__init__.py` file will turn the `calc/` directory into a Python package. Then you can add the following content to your `constants.py` file:

Python

```
# constants.py  
  
"""This module defines project-level constants."""  
  
PI = 3.141592653589793  
EULER_NUMBER = 2.718281828459045  
TAU = 6.283185307179586
```

Once you've added this code to `constants.py`, then you can import the module whenever you need to use any of your constants:

Python

```
# calculations.py  
  
"""This module implements custom calculations."""  
  
# Imports go here...  
import numpy as np  
  
from . import constants  
  
# Your custom calculations start here...
```

```
def circular_land_area(radius):
    return constants.PI * radius**2

def future_value(present_value, interest_rate, years):
    return present_value * constants.EULER_NUMBER ** (interest_

# ...
```

Note that you import the constants module directly from the calc package using a [relative import](#). Then you use fully qualified names to access any required constants in your calculations. This practice improves your communication of intent. Now it's completely clear that PI and EULER_NUMBER are constants in your project because of the constants prefix.

To use your calculations module, you can do something like this:

```
Python >>>

>>> from calc import calculations
>>> calculations.circular_land_area(100)
31415.926535897932

>>> from calc.calculations import circular_land_area
>>> circular_land_area(100)
31415.926535897932
```

Now your calculations module lives inside the calc package. This means that if you want to use the functions in calculations, then you need to import calculations from calc. You can also import the functions directly by referencing the package and the module like you did in the second example above.



Your **Practical Introduction to Python 3** »

Remove ads

Storing Constants in Configuration Files

Now say that you want to go further when it comes to externalizing the constants of a given project. You may need to keep all your

constants out of your project's source code. To do this, you can use an external **configuration file**.

Here's an example of how to move your constants to a configuration file:

Config File

```
; constants.ini

[CONSTANTS]
PI=3.141592653589793
EULER_NUMBER=2.718281828459045
TAU=6.283185307179586
```

This file uses the [INI file](#) format. You can read this type of file using the configparser module from the standard library.

Now get back to calculations.py and update it to look something like the following:

Python

```
# calculations.py

"""This module implements custom calculations."""

# Imports go here...
from configparser import ConfigParser

import numpy as np

constants = ConfigParser()
constants.read("path/to/constants.ini")

# Your custom calculations start here...
def circular_land_area(radius):
    return float(constants.get("CONSTANTS", "PI")) * radius**2

def future_value(present_value, interest_rate, years):
    return (
        present_value * float(constants.get(
            "CONSTANTS",
            "EULER_NUMBER"
        ))) ** (interest_rate * years)
```



```
# ...
```

In this example, your code first reads the configuration file and stores the resulting `ConfigParser` object in a global variable, `constants`. You can also name this variable `CONSTANTS` and use it globally as a constant. Then you update your calculations to read the constants from the configuration object itself.

Note that `ConfigParser` objects store the configuration parameters as strings, so you need to use the built-in `float()` function to convert the values into numbers.

This strategy may be beneficial when you're creating a [graphical user interface \(GUI\) app](#) and need to set some parameters to define the shape and size of the app's windows when loading and showing the GUI, for example.

Handling Constants as Environment Variables

Another helpful strategy to handle your constants is to define them as **system variables** if you're on Windows or **environment variables** if you're on macOS or Linux.

This approach is commonly used to configure [deployment](#) in different environments. You can also use environment variables for constants that imply security risks and shouldn't be directly committed to the source code. Examples of these types of constants include authentication credentials, API access tokens, and so on.


Note: You should [be careful](#) when using environment variables for sensitive information because they may be accidentally exposed in logs or to child processes. All cloud providers offer some kind of [secrets management](#) that's more secure.


To use this strategy, you first must export your constants as environment or system variables in your operating system. There

are at least two ways to do this:

1. Manually export the constants in your current [shell](#) session
2. Add your constants to the shell's configuration file

The first technique is pretty quick and practical. You can use it to run some fast tests on your code. For example, say that you need to export an API token as a system or environment variable. In that case, you just need to run the following command:

 Windows

 Linux + macOS


Windows Command Prompt


```
C:\> set API_TOKEN="593086396372"
```

The main drawback of this technique is that your constants will be accessible only from the command-line session in which you defined them. A much better approach is to make your operating system load the constants whenever you fire up a command-line window.

If you're on Windows, then check out the [Configuring Environment Variables](#) section in [Your Python Coding Environment on Windows: Setup Guide](#) to learn how to create system variables. Follow the instructions in this guide and add an `API_TOKEN` system variable with a value of `593086396372`.

If you're on Linux or macOS, then you can go to your home folder and open your shell's configuration file. Once you've opened that file, add the following line at the end of it:

 Linux

 macOS

Configuration File

```
# .bashrc  
  
export API_TOKEN="593086396372"
```

Linux and macOS automatically load the corresponding shell

configuration file whenever you start a terminal or command-line window. This way, you ensure that the `API_TOKEN` variable is always available on your system.

Once you've defined the required environment variables for your Python constant, then you need to load them into your code. To do this, you can use the `environ` dictionary from Python's `os` module. The keys and values of `environ` are strings representing the environment variables and their values, respectively.

Your `API_TOKEN` constant is now present in the `environ` dictionary. Therefore, you can read it from there with just two lines of code:

```
Python >>>
>>> import os
>>> os.environ["API_TOKEN"]
'593086396372'
```

Using environment variables to store constants, and the `os.environ` dictionary to read them into your code, is an effective way of configuring constants that depend on the environment your application is deployed in. It's particularly useful when working with the cloud, so keep this technique in your Python tool kit.

Find Your Dream Python Job

pythonjobshq.com



 Remove ads

Exploring Other Constants in Python

Apart from user-defined constants, Python also defines several internal names that can be considered as constants. Some of these names are strict constants, meaning that you can't change them once the interpreter is running. This is the case for the `__debug__` constant, for example.

In the following sections, you'll learn about some internal Python names that you can consider and should treat as constants in your code. To kick things off, you'll review some built-in constants and constant values.

Built-in Constants

According to the Python documentation, “A small number of constants live in the built-in namespace” ([Source](#)). The first two constants listed in the docs are `True` and `False`, which are the Python [Boolean](#) values. These two values are also instances of `int`. `True` has a value of 1, and `False` has a value of 0:

```
Python >>>

>>> True
True
>>> False
False

>>> isinstance(True, int)
True
>>> isinstance(False, int)
True

>>> int(True)
1
>>> int(False)
0

>>> True = 42
...
SyntaxError: cannot assign to True

>>> True is True
True
>>> False is False
True
```

Note that the `True` and `False` names are strict constants. In other words, they can't be reassigned. If you try to reassign them, then you get a [SyntaxError](#). These two values are also singleton objects in Python, meaning that there's only one instance of each. That's why

the [identity operator](#) (`is`) returns `True` in the final examples above.

Another important and commonplace constant value is `None`, which is the null value in Python. This constant value comes in handy when you want to express the idea of [nullability](#). Just like `True` and `False`, `None` is also a singleton and strict constant object that can't be reassigned:

```
Python >>>

>>> None is None
True

>>> None = 42
...
SyntaxError: cannot assign to None
```

`None` is quite useful as a default argument value in functions, methods, and class constructors. It's typically used to communicate that a variable is empty. Internally, Python uses `None` as the implicit return value of functions that don't have an [explicit return statement](#).

The ellipsis literal (`...`) is another constant value in Python. This special value is the same as [Ellipsis](#) and is the only instance of the [types.EllipsisType](#) type:

```
Python >>>

>>> Ellipsis
Ellipsis

>>> ...
Ellipsis

>>> ... is Ellipsis
True
```

You can use `Ellipsis` as a placeholder for unwritten code. You can also use it to replace the [pass](#) statement. In type hints, the `...` literal communicates the idea of an [unknown-length collection](#) of data with a uniform type:

Python

>>>

```
>>> def do_something():
...     ... # TODO: Implement this function later
...

>>> class CustomException(Exception): ...
...
>>> raise CustomException("some error message")
Traceback (most recent call last):
...
CustomException: some error message

>>> # A tuple of integer values
>>> numbers: tuple[int, ...]
```

The `Ellipsis` constant value can come in handy in many situations and help you make your code more readable because of its semantic equivalence to the English ellipsis punctuation sign (...).

Another interesting and potentially useful built-in constant is `__debug__`, as you already learned at the beginning of this section. Python's `__debug__` is a Boolean constant that defaults to `True`. It's a strict constant because you can't change its value once your interpreter is running:

Python

>>>

```
>>> __debug__
True

>>> __debug__ = False
...
SyntaxError: cannot assign to __debug__
```

The `__debug__` constant is closely related to the `assert` statement. In short, if `__debug__` is `True`, then all your `assert` statements will run. If `__debug__` is `False`, then your `assert` statements will be disabled and won't run at all. This feature can slightly improve the performance of your production code.

Note: Even though `__debug__` also has a dunder name, it's a

strict constant because you can't change its value once the interpreter is running. In contrast, the internal dunder names in the section below should be treated as constants but aren't strict constants. You can change their values during your code's execution. However, this practice can be tricky and would require advanced knowledge.

To change the value of `__debug__` to `False`, you must run Python in **optimized mode** by using the `-O` or `-OO` command-line options, which provide two levels of [bytecode](#) optimization. Both levels generate an optimized Python bytecode that doesn't contain assertions.



[Real Python for Teams](#) »

 Remove ads

Internal Dunder Names

Python also has a broad set of internal dunder names that you can consider as constants. Because there are several of these special names, you'll just learn about `__name__` and `__file__` in this tutorial.

Note: To dive deeper into other dunder names in Python and what they mean to the language, check out the official documentation about Python's [data model](#).

The `__name__` attribute is closely related to how you run a given piece of code. When importing a module, Python internally sets `__name__` to a string containing the name of the module that you're importing.

Fire up your [code editor](#) and create the following sample module:

Python

```
# sample_name.py

print(f"The type of __name__ is: {type(__name__)}")
```

```
print(f"The value of __name__ is: {__name__}")
```

Once you have this file in place, get back to your command-line window and run the following command:

Shell

```
$ python -c "import sample_name"
The type of __name__ is: <class 'str'>
The value of __name__ is: sample_name
```

With the `-c` switch, you can execute a small piece of Python code at the command line. In this example, you import the `sample_name` module, which [prints](#) some messages to the screen. The first message tells you that `__name__` is of type [str](#), or string. The second message shows that `__name__` was set to `sample_name`, which is the name of the module you just imported.

Alternatively, if you take `sample_name.py` and [run it as a script](#), then Python will set `__name__` to the `"__main__"` string. To confirm this fact, go ahead and run the following command:

Shell

```
$ python sample_name.py
The type of __name__ is: <class 'str'>
The value of __name__ is: __main__
```

Note that now `__name__` holds the `"__main__"` string. This behavior indicates that you've run the file directly as an executable Python program.

The `__file__` attribute will contain the path to the file that Python is currently importing or executing. You can use `__file__` from inside a given module when you need to get the path to the module itself.

As an example of how `__file__` works, go ahead and create the following module:

Python

```
# sample_file.py
```



```
print(f"The type of __file__ is: {type(__file__)}")
print(f"The value of __file__ is: {__file__}")
```

If you import the `sample_file` module in your Python code, then `__file__` will store the path to its containing module on your file system. Check this out by running the following command:

Shell

```
$ python -c "import sample_file"
The type of __file__ is: <class 'str'>
The value of __file__ is: /path/to/sample_file.py
```

Likewise, if you run `sample_file.py` as a Python executable program, then you get the same output as before:

Shell

```
$ python sample_file.py
The type of __file__ is: <class 'str'>
The value of __file__ is: /path/to/sample_file.py
```

In short, Python sets `__file__` to contain the path to the module from which you're using or accessing this attribute.



[Online Python Training for Teams »](#)

 Remove ads

Useful String and Math Constants

You'll find many useful constants in the standard library. Some of them are tightly connected to some specific modules, functions, and classes. Others are more generic, and you can use them in various scenarios. That's the case with some math and string-related constants that you can find in the `math` and `string` modules, respectively.

The `math` module provides the following constants:

Python

>>>

```
>>> import math

>>> # Euler's number (e)
>>> math.e
2.718281828459045

>>> # Pi ( $\pi$ )
>>> math.pi
3.141592653589793

>>> # Infinite ( $\infty$ )
>>> math.inf
inf

>>> # Not a number (NaN)
>>> math.nan
nan

>>> # Tau ( $\tau$ )
>>> math.tau
6.283185307179586
```

These constants will come in handy whenever you're writing math-related code or even code that just uses them to perform specific computations, like your `Circle` class back in the [Reusing Objects for Maintainability](#) section.

Here's an updated implementation of `Circle` using `math.pi` instead of your custom PI constant:

Python

```
# circle.py

import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius**2

    def perimeter(self):
```

```

        return 2 * math.pi * self.radius

    def projected_volume(self):
        return 4/3 * math.pi * self.radius**3

    def __repr__(self):
        return f'{self.__class__.__name__}(radius={self.radius})'

```

This updated version of `Circle` is more readable than your original version because it provides more context on where the `Pi` constant comes from, making it clear that it's a math-related constant.

The `math.pi` constant also has the advantage that if you're using an older version of Python, then you'll get a 32-bit version of `Pi`. In contrast, if you use `Circle` in a modern version of Python, then you'll get a 64-bit version of `Pi`. So, your program will self-adapt to its concrete execution environment.

The `string` module also defines several useful [string constants](#). The table below shows the name and value of each constant:

Name	Value
<code>ascii_lowercase</code>	<code>abcdefghijklmnopqrstuvwxyz</code>
<code>ascii_uppercase</code>	<code>ABCDEFGHIJKLMNOPQRSTUVWXYZ</code>
<code>ascii_letters</code>	<code>ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz</code>
<code>digits</code>	<code>0123456789</code>
<code>hexdigits</code>	<code>0123456789abcdefABCDEF</code>
<code>octdigits</code>	<code>01234567</code>
<code>punctuation</code>	<code>!"#\$%&'()*+,-./:;<=>?@[\]^_`{ }~</code>
<code>whitespace</code>	The combination of the space character, horiz tab , linefeed , carriage return , and form feed

Name	Value
printable	The combination of digits, ascii_letters, punctuation, and whitespace

These string-related constants come in handy in many situations. You can use them when you're doing a lot of string processing, working with [regular expressions](#), processing [natural language](#), and more.

Type-Annotating Constants

Since Python 3.8, the `typing` module includes a `Final` class that allows you to type-annotate constants. If you use this class when defining your constants, then you'll tell static type checkers like `mypy` that your constants shouldn't be reassigned. This way, the type checker can help you detect unauthorized assignments on your constants.

Here are some examples of using `Final` to define your constants:

Python

```
from typing import Final

MAX_SPEED: Final[int] = 300
DEFAULT_COLOR: Final[str] = "\033[1;34m"
ALLOWED_BUILTINS: Final[tuple[str, ...]] = ("sum", "max", "min")

# Later in your code...
MAX_SPEED = 450 # Cannot assign to final name "MAX_SPEED" mypy
```

The `Final` class represents a special typing construct that indicates type checkers to report an error if the name at hand is reassigned at some point in your code. Note that even though you get a type checker's error report, Python does change the value of `MAX_SPEED`. So, `Final` doesn't prevent accidental constant reassignments at runtime.

Defining Strict Constants in Python

Up to this point, you've learned a lot about programming and Python constants. You now know that Python doesn't support strict constants. It just has variables. Therefore, the Python community has adopted the naming convention of using uppercase letters to communicate that a given variable is really a constant.

So, in Python, you don't have constants. Rather, you have variables that never change. This can be an issue if you're working on a large Python project with many programmers at different levels. In this situation, it'd be nice to have a mechanism that guarantees **strict constants**— constants that no one can change after the program has started.

Because Python is a pretty flexible programming language, you'll find several ways to achieve the goal of making your constant unchangeable. In the following few sections, you'll learn about some of these ways. They all imply creating a custom class and using it as a namespace for constants.

Why should you use a class as the namespace for your constants? In Python, any name can be rebound at will. At the module level, you don't have the appropriate tools to prevent this from happening. So, you need to use a class because classes provide way more customization tools than modules.

In the following sections, you'll learn about several different ways to use a class as your namespace for strict constants.



 Remove ads

The `__slots__` Attribute

Python classes allow you to define a special class attribute called `__slots__`. This attribute will hold a sequence of names that'll work as instance attributes.

You won't be able to add new instance attribute to a class with a `__slots__` attribute, because `__slots__` prevents the creation of an instance `__dict__` attribute. Additionally, not having a `__dict__` attribute implies an optimization in terms of memory consumption.

Using `__slots__`, you can create a class that works as a namespace for read-only constants:

```
Python >>>
>>> class ConstantsNamespace:
...     __slots__ = ()
...     PI = 3.141592653589793
...     EULER_NUMBER = 2.718281828459045
...
>>> constants = ConstantsNamespace()
>>> constants.PI
3.141592653589793
>>> constants.EULER_NUMBER
2.718281828459045
>>> constants.PI = 3.14
Traceback (most recent call last):
...
AttributeError: 'ConstantsNamespace' object attribute 'PI' is r
```

In this example, you define `ConstantsNamespace`. The class's `__slots__` attribute holds an empty `tuple`, meaning that instances of this class will have no attributes. Then you define your constants as class attributes.

The next step is to instantiate the class to create a variable holding the namespace with all your constants. Note that you can quickly access any constant in your special namespace, but you can't assign it a new value. If you try to do it, then you get an `AttributeError`.

With this technique, you're guaranteeing that no one else on your team can change the value of your constants. You've achieved the expected behavior of a strict constant.

The @property Decorator

You can also take advantage of the [@property decorator](#) to create a class that works as a namespace for your constants. To do this, you just need to define your constants as properties without providing them with a setter method:

```
Python >>>

>>> class ConstantsNamespace:
...     @property
...     def PI(self):
...         return 3.141592653589793
...     @property
...     def EULER_NUMBER(self):
...         return 2.718281828459045
...

>>> constants = ConstantsNamespace()

>>> constants.PI
3.141592653589793
>>> constants.EULER_NUMBER
2.718281828459045

>>> constants.PI = 3.14
Traceback (most recent call last):
...
AttributeError: can't set attribute 'PI'
```

Because you don't provide setter methods for the `PI` and `EULER_NUMBER` properties, they're [read-only properties](#). This means that you can only *access* their values. It's impossible to assign a new value to either one. If you try to do it, then you get an `AttributeError`.

The `namedtuple()` Factory Function

Python's [collections](#) module provides a [factory function](#) called [namedtuple\(\)](#). This function lets you create **tuple subclasses** that allow the use of **named fields** and the **dot notation** to access their items, like in `tuple_obj.attribute`.

Like regular tuples, named tuple instances are [immutable](#), which implies that you can't modify an existing named tuple object [in place](#). Being immutable sounds appropriate for creating a class that works as a namespace of strict constants.

Here's how to do it:

```
Python >>>

>>> from collections import namedtuple

>>> ConstantsNamespace = namedtuple(
...     "ConstantsNamespace", ["PI", "EULER_NUMBER"]
... )
>>> constants = ConstantsNamespace(3.141592653589793, 2.718281828459045)

>>> constants.PI
3.141592653589793
>>> constants.EULER_NUMBER
2.718281828459045

>>> constants.PI = 3.14
Traceback (most recent call last):
...
AttributeError: can't set attribute
```

In this example, your constants play the role of fields in the underlying named tuple, `ConstantsNamespace`. Once you've created the named tuples instance, `constants`, you can access your constants by using the dot notation, like in `constants.PI`.

Because tuples are immutable, there's no way for you to modify the value of any field. So, your constants named tuple object is a full-fledged namespace of strict constants.

The `@dataclass` Decorator

[Data classes](#) are classes that contain mainly data, as their name indicates. They can also have methods, but that's not their primary goal. To create a data class, you need to use the `@dataclass` decorator from the `dataclasses` module.

How can you use this type of class to create a namespace of strict

constants? The `@dataclass` decorator accepts a `frozen` argument that allows you to mark your data class as immutable. If it's immutable, then once you've created an instance of a given data class, you have no way to modify its instance attributes.

Here's how you can use a data class to create a namespace containing your constants:

```
Python >>>

>>> from dataclasses import dataclass

>>> @dataclass(frozen=True)
... class ConstantsNamespace:
...     PI = 3.141592653589793
...     EULER_NUMBER = 2.718281828459045
...

>>> constants = ConstantsNamespace()

>>> constants.PI
3.141592653589793
>>> constants.EULER_NUMBER
2.718281828459045

>>> constants.PI = 3.14
Traceback (most recent call last):
...
dataclasses.FrozenInstanceError: cannot assign to field 'PI'
```

In this example, you first import the `@dataclass` decorator. Then you use this decorator to turn `ConstantsNamespace` into a data class. To make the data class immutable, you set the `frozen` argument to `True`. Finally, you define `ConstantsNamespace` with your constants as class attributes.

You can create an instance of this class and use it as your constants namespace. Again, you can access all the constants, but you can't modify their values, because the data class is frozen.

The `.__setattr__()` Special Method

Python classes let you define a special method called

`.__setattr__()`. This method allows you to customize the attribute assignment process because Python automatically calls the method on every attribute assignment.

In practice, you can override `.__setattr__()` to prevent all attribute reassignments and make your attributes immutable. Here's how you can override this method to create a class that works as a namespace for your constants:

```
Python >>>

>>> class ConstantsNamespace:
...     PI = 3.141592653589793
...     EULER_NUMBER = 2.718281828459045
...     def __setattr__(self, name, value):
...         raise AttributeError(f"can't reassign constant '{name}'")
...

>>> constants = ConstantsNamespace()

>>> constants.PI
3.141592653589793
>>> constants.EULER_NUMBER
2.718281828459045

>>> constants.PI = 3.14
Traceback (most recent call last):
...
AttributeError: can't reassign constant 'PI'
```

Your custom implementation of `.__setattr__()` doesn't perform any assignment operation on the class's attributes. It just raises an `AttributeError` when you try to set any attribute. This implementation makes the attributes immutable. Again, your `ConstantsNamespace` behaves as a namespace for constants.

Conclusion

Now you know what **constants** are, as well as why and when to use them in your code. You also know that Python doesn't have strict constants. The Python community uses *uppercase letters* as a naming convention to communicate that a variable should be used

as a constant. This naming convention helps to prevent other developers from changing variables that are meant to be constant.

Constants are everywhere in programming, and Python developers also use them. So, learning to define and use constants in Python is an important skill for you to master.

In this tutorial, you learned how to:

- Define **Python constants** in your code
- Identify and understand some **built-in constants**
- Improve your code's **readability, reusability, and maintainability** with constants
- Use different strategies to **organize and manage constants** in a real-world project
- Apply various techniques to make your Python constants **strictly constant**

With this knowledge about what constants are, why they're important, and when to use them, you're ready to start improving your code's readability, maintainability, and reusability immediately. Go ahead and give it a try!

Sample Code: [Click here to download sample code](#) that shows you how to use constants in Python.

Mark as Completed



Python Tricks



Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

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



Geir
Arne

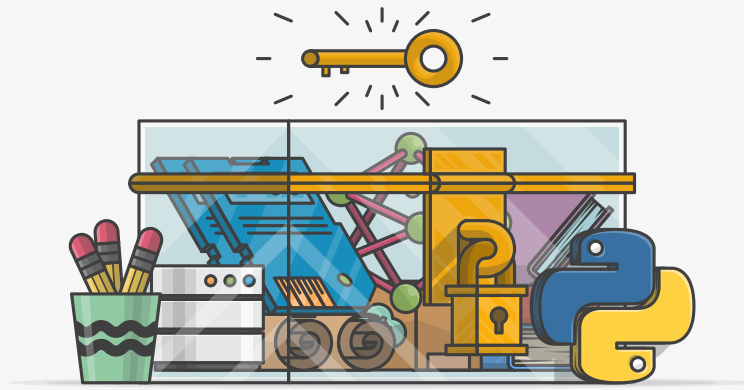


Kate



Martin

Master Real-World Python Skills With
Unlimited Access to Real Python



Join us and get access to thousands of tutorials,
hands-on video courses, and a community of
expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Rate this article:



Tweet

Share

Share

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.

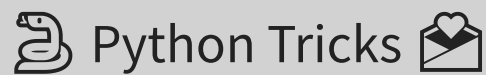
Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. [Get tips for asking good questions](#) and [get answers to common questions in our support portal](#).

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 Tutorial Categories: [intermediate](#) [python](#)

— FREE Email Series —



```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

[Get Python Tricks »](#)

 No spam. Unsubscribe any time.

All Tutorial Topics

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



The Real Python Podcast



Listen Now →

Table of Contents

- Understanding Constants and Variables
- Defining Your Own Constants in Python
- Putting Constants Into Action
- Handling Your Constants in a Real-World Project
- Exploring Other Constants in Python
- Type-Annotating Constants
- Defining Strict Constants in Python
- Conclusion

Mark as Completed



 **Tweet**

 **Share**

 **Email**

```
1 # How to merge two dicts
2 # in Python 3.5+
3
```

Improve Your Python with 🐍 Python Tricks ❤️

```
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Get a short & sweet Python code snippet delivered to your inbox every couple of days:

» [Click here to see examples](#)

Python Dependency Management Pitfalls

A free email class

realpython.com



 [Remove ads](#)

© 2012–2022 Real Python · [Newsletter](#) ·

[Podcast](#) · [YouTube](#) · [Twitter](#) · [Facebook](#) ·

[Instagram](#) · [Python Tutorials](#) · [Search](#) · [Privacy](#)

[Policy](#) · [Energy Policy](#) · [Advertise](#) · [Contact](#)

♥ Happy Pythoning!