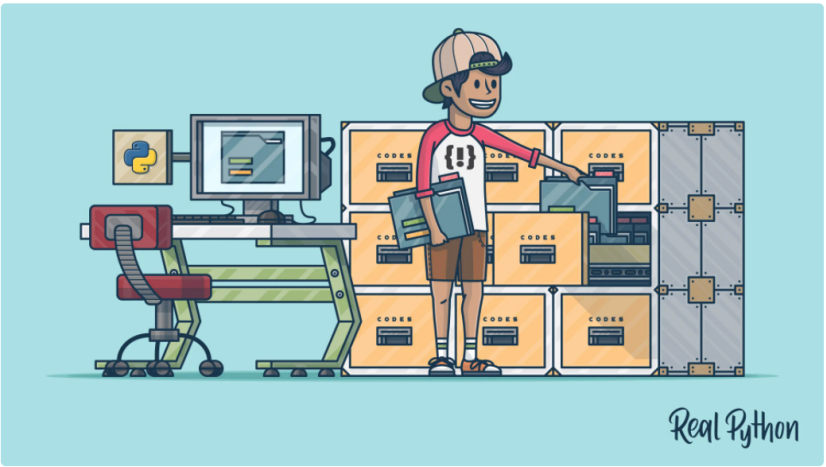

Documenting Python Code: A Complete Guide

by James Mertz Jul 25, 2018 31 Comments best-practices intermediate python

Mark as Completed

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding:
Documenting Python Code: A Complete Guide

Welcome to your complete guide to documenting Python code. Whether you're documenting a small script or a large project, whether you're a beginner or a seasoned Pythonista, this guide will cover everything you need to know.

We've broken up this tutorial into four major sections:

1. **Why Documenting Your Code Is So Important:** An introduction to documentation and its importance
2. **Commenting vs Documenting Code:** An overview of the major differences between commenting and documenting, as well as the appropriate times and ways to use

Table of Contents

- Why Documenting Your Code Is So Important
- Commenting vs Documenting Code
- Documenting Your Python Code Base Using Docstrings
- Documenting Your Python Projects
- Where Do I Start?

Mark as Completed

Tweet Share Email

Recommended Video Course
Documenting Python Code: A Complete Guide

commenting

3. **Documenting Your Python Code Base Using Docstrings:** A deep dive into docstrings for classes, class methods, functions, modules, packages, and scripts, as well as what should be found within each one
4. **Documenting Your Python Projects:** The necessary elements and what they should contain for your Python projects

Feel free to read through this tutorial from beginning to end or jump to a section you're interested in. It was designed to work both ways.

Why Documenting Your Code Is So Important

Hopefully, if you're reading this tutorial, you already know the importance of documenting your code. But if not, then let me quote something Guido mentioned to me at a recent PyCon:

> "Code is more often read than written."
>
> — *Guido van Rossum*

When you write code, you write it for two primary audiences: your users and your developers (including yourself). Both audiences are equally important. If you're like me, you've probably opened up old codebases and wondered to yourself, "What in the world was I thinking?" If you're having a problem reading your own code, imagine what your users or other developers are experiencing when they're trying to use or contribute to your code.

Conversely, I'm sure you've run into a situation where you wanted to do something in Python and found what looks like a great library that can get the job done. However, when you start using the library, you look for examples, write-ups, or even official documentation on how to do something specific and can't immediately find the solution.

After searching, you come to realize that the documentation is lacking or even worse, missing entirely. This is a frustrating feeling that deters you from using the library, no matter how great or efficient the code is. Daniele Procida summarized this situation best:

> "It doesn't matter how good your software is, because **if the documentation is not good enough, people will not use it.**"
>
> — *Daniele Procida*

In this guide, you'll learn from the ground up how to properly document your Python code from the smallest of scripts to the largest of Python projects to help prevent your users from ever feeling too frustrated to use or contribute to your project.

Commenting vs Documenting Code

Before we can go into how to document your Python code, we need to distinguish documenting from commenting.

In general, commenting is describing your code to/for developers. The intended main audience is the maintainers and developers of the Python code. In conjunction with well-written code, comments help to guide the reader to better understand your code and its purpose and design:

> "Code tells you how; Comments tell you why."
>
> — *Jeff Atwood (aka Coding Horror)*

Documenting code is describing its use and functionality to your users. While it may be

helpful in the development process, the main intended audience is the users. The following section describes how and when to comment your code.

Basics of Commenting Code

Comments are created in Python using the pound sign (#) and should be brief statements no longer than a few sentences. Here's a simple example:

```Python
def hello_world():
    # A simple comment preceding a simple print statement
    print("Hello World")
```

According to PEP 8, comments should have a maximum length of 72 characters. This is true even if your project changes the max line length to be greater than the recommended 80 characters. If a comment is going to be greater than the comment char limit, using multiple lines for the comment is appropriate:

```Python
def hello_long_world():
    # A very long statement that just goes on and on and on and on and
    # never ends until after it's reached the 80 char limit
    print("Hellooooooooooooooooooooooooooooooooooooooooooooooooooooooooo World")
```

Commenting your code serves multiple purposes, including:

- **Planning and Reviewing:** When you are developing new portions of your code, it may be appropriate to first use comments as a way of planning or outlining that section of code. Remember to remove these comments once the actual coding has been implemented and reviewed/tested:

  ```Python
  # First step
  # Second step
  # Third step
  ```

- **Code Description:** Comments can be used to explain the intent of specific sections of code:

  ```Python
  # Attempt a connection based on previous settings. If unsuccessful,
  # prompt user for new settings.
  ```

- **Algorithmic Description:** When algorithms are used, especially complicated ones, it can be useful to explain how the algorithm works or how it's implemented within your code. It may also be appropriate to describe why a specific algorithm was selected over another.

  ```Python
  # Using quick sort for performance gains
  ```

- **Tagging:** The use of tagging can be used to label specific sections of code where known issues or areas of improvement are located. Some examples are: `BUG`, `FIXME`, and `TODO`.

  ```Python
  # TODO: Add condition for when val is None
  ```

Comments to your code should be kept brief and focused. Avoid using long comments when possible. Additionally, you should use the following four essential rules as suggested by Jeff Atwood:

1. Keep comments as close to the code being described as possible. Comments that

aren't near their describing code are frustrating to the reader and easily missed when updates are made.

2. Don't use complex formatting (such as tables or ASCII figures). Complex formatting leads to distracting content and can be difficult to maintain over time.

3. Don't include redundant information. Assume the reader of the code has a basic understanding of programming principles and language syntax.

4. Design your code to comment itself. The easiest way to understand code is by reading it. When you design your code using clear, easy-to-understand concepts, the reader will be able to quickly conceptualize your intent.

Remember that comments are designed for the reader, including yourself, to help guide them in understanding the purpose and design of the software.

Commenting Code via Type Hinting (Python 3.5+)

Type hinting was added to Python 3.5 and is an additional form to help the readers of your code. In fact, it takes Jeff's fourth suggestion from above to the next level. It allows the developer to design and explain portions of their code without commenting. Here's a quick example:

Python

```
def hello_name(name: str) -> str:
    return(f"Hello {name}")
```

From examining the type hinting, you can immediately tell that the function expects the input `name` to be of a type `str`, or string. You can also tell that the expected output of the function will be of a type `str`, or string, as well. While type hinting helps reduce comments, take into consideration that doing so may also make extra work when you are creating or updating your project documentation.

You can learn more about type hinting and type checking from this video created by Dan Bader.

Documenting Your Python Code Base Using Docstrings

Now that we've learned about commenting, let's take a deep dive into documenting a Python code base. In this section, you'll learn about docstrings and how to use them for documentation. This section is further divided into the following sub-sections:

1. **Docstrings Background:** A background on how docstrings work internally within Python
2. **Docstring Types:** The various docstring "types" (function, class, class method, module, package, and script)
3. **Docstring Formats:** The different docstring "formats" (Google, NumPy/SciPy, reStructured Text, and Epytext)

Docstrings Background

Documenting your Python code is all centered on docstrings. These are built-in strings that, when configured correctly, can help your users and yourself with your project's documentation. Along with docstrings, Python also has the built-in function `help()` that prints out the objects docstring to the console. Here's a quick example:

Python >>>

```
>>> help(str)
```

```
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors are specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 # Truncated for readability
```

How is this output generated? Since everything in Python is an object, you can examine the directory of the object using the `dir()` command. Let's do that and see what find:

Python >>>

```
>>> dir(str)
['__add__', ..., '__doc__', ..., 'zfill'] # Truncated for readability
```

Within that directory output, there's an interesting property, `__doc__`. If you examine that property, you'll discover this:

Python >>>

```
>>> print(str.__doc__)
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

Create a new string object from the given object. If encoding or
errors are specified, then the object must expose a data buffer
that will be decoded using the given encoding and error handler.
Otherwise, returns the result of object.__str__() (if defined)
or repr(object).
encoding defaults to sys.getdefaultencoding().
errors defaults to 'strict'.
```

Voilà! You've found where docstrings are stored within the object. This means that you can directly manipulate that property. However, there are restrictions for builtins:

Python >>>

```
>>> str.__doc__ = "I'm a little string doc! Short and stout; here is my input and pri
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't set attributes of built-in/extension type 'str'
```

Any other custom object can be manipulated:

Python

```
def say_hello(name):
    print(f"Hello {name}, is it me you're looking for?")

say_hello.__doc__ = "A simple function that says hello... Richie style"
```

Python >>>

```
>>> help(say_hello)
Help on function say_hello in module __main__:

say_hello(name)
    A simple function that says hello... Richie style
```

Python has one more feature that simplifies docstring creation. Instead of directly manipulating the `__doc__` property, the strategic placement of the string literal directly below the object will automatically set the `__doc__` value. Here's what happens with the same example as above:

Python

```python
def say_hello(name):
    """A simple function that says hello... Richie style"""
    print(f"Hello {name}, is it me you're looking for?")
```

Python >>>

```
>>> help(say_hello)
Help on function say_hello in module __main__:

say_hello(name)
    A simple function that says hello... Richie style
```

There you go! Now you understand the background of docstrings. Now it's time to learn about the different types of docstrings and what information they should contain.

Docstring Types

Docstring conventions are described within PEP 257. Their purpose is to provide your users with a brief overview of the object. They should be kept concise enough to be easy to maintain but still be elaborate enough for new users to understand their purpose and how to use the documented object.

In all cases, the docstrings should use the triple-double quote (`"""`) string format. This should be done whether the docstring is multi-lined or not. At a bare minimum, a docstring should be a quick summary of whatever is it you're describing and should be contained within a single line:

Python

```python
"""This is a quick summary line used as a description of the object."""
```

Multi-lined docstrings are used to further elaborate on the object beyond the summary. All multi-lined docstrings have the following parts:

- A one-line summary line
- A blank line proceeding the summary
- Any further elaboration for the docstring
- Another blank line

Python

```python
"""This is the summary line

This is the further elaboration of the docstring. Within this section,
you can elaborate further on details as appropriate for the situation.
Notice that the summary and the elaboration is separated by a blank new
line.
"""

# Notice the blank line above. Code should continue on this line.
```

All docstrings should have the same max character length as comments (72 characters). Docstrings can be further broken up into three major categories:

- **Class Docstrings:** Class and class methods
- **Package and Module Docstrings:** Package, modules, and functions
- **Script Docstrings:** Script and functions

Class Docstrings

Class Docstrings are created for the class itself, as well as any class methods. The docstrings are placed immediately following the class or class method indented by one level:

Python

```python
class SimpleClass:
    """Class docstrings go here."""

    def say_hello(self, name: str):
        """Class method docstrings go here."""
```

```
        print(f'Hello {name}')
```

Class docstrings should contain the following information:

- A brief summary of its purpose and behavior
- Any public methods, along with a brief description
- Any class properties (attributes)
- Anything related to the interface for subclassers, if the class is intended to be subclassed

The class constructor parameters should be documented within the `__init__` class method docstring. Individual methods should be documented using their individual docstrings. Class method docstrings should contain the following:

- A brief description of what the method is and what it's used for
- Any arguments (both required and optional) that are passed including keyword arguments
- Label any arguments that are considered optional or have a default value
- Any side effects that occur when executing the method
- Any exceptions that are raised
- Any restrictions on when the method can be called

Let's take a simple example of a data class that represents an Animal. This class will contain a few class properties, instance properties, a `__init__`, and a single instance method:

Python

```python
class Animal:
    """
    A class used to represent an Animal

    ...

    Attributes
    ----------
    says_str : str
        a formatted string to print out what the animal says
    name : str
        the name of the animal
    sound : str
        the sound that the animal makes
    num_legs : int
        the number of legs the animal has (default 4)

    Methods
    -------
    says(sound=None)
        Prints the animals name and what sound it makes
    """

    says_str = "A {name} says {sound}"

    def __init__(self, name, sound, num_legs=4):
        """
        Parameters
        ----------
        name : str
            The name of the animal
        sound : str
            The sound the animal makes
        num_legs : int, optional
            The number of legs the animal (default is 4)
        """

        self.name = name
        self.sound = sound
        self.num_legs = num_legs

    def says(self, sound=None):
        """Prints what the animals name is and what sound it makes.
```

```
        If the argument `sound` isn't passed in, the default Animal
        sound is used.

        Parameters
        ----------
        sound : str, optional
            The sound the animal makes (default is None)

        Raises
        ------
        NotImplementedError
            If no sound is set for the animal or passed in as a
            parameter.
        """

        if self.sound is None and sound is None:
            raise NotImplementedError("Silent Animals are not supported!")

        out_sound = self.sound if sound is None else sound
        print(self.says_str.format(name=self.name, sound=out_sound))
```

Package and Module Docstrings

Package docstrings should be placed at the top of the package's `__init__.py` file. This docstring should list the modules and sub-packages that are exported by the package.

Module docstrings are similar to class docstrings. Instead of classes and class methods being documented, it's now the module and any functions found within. Module docstrings are placed at the top of the file even before any imports. Module docstrings should include the following:

- A brief description of the module and its purpose
- A list of any classes, exception, functions, and any other objects exported by the module

The docstring for a module function should include the same items as a class method:

- A brief description of what the function is and what it's used for
- Any arguments (both required and optional) that are passed including keyword arguments
- Label any arguments that are considered optional
- Any side effects that occur when executing the function
- Any exceptions that are raised
- Any restrictions on when the function can be called

Script Docstrings

Scripts are considered to be single file executables run from the console. Docstrings for scripts are placed at the top of the file and should be documented well enough for users to be able to have a sufficient understanding of how to use the script. It should be usable for its "usage" message, when the user incorrectly passes in a parameter or uses the `-h` option.

If you use `argparse`, then you can omit parameter-specific documentation, assuming it's correctly been documented within the `help` parameter of the `argparser.parser.add_argument` function. It is recommended to use the `__doc__` for the `description` parameter within `argparse.ArgumentParser`'s constructor. Check out our tutorial on Command-Line Parsing Libraries for more details on how to use `argparse` and other common command line parsers.

Finally, any custom or third-party imports should be listed within the docstrings to allow users to know which packages may be required for running the script. Here's an example of a script that is used to simply print out the column headers of a spreadsheet:

Python

```
"""Spreadsheet Column Printer

This script allows the user to print to the console all columns in the
```

```
spreadsheet. It is assumed that the first row of the spreadsheet is the
location of the columns.

This tool accepts comma separated value files (.csv) as well as excel
(.xls, .xlsx) files.

This script requires that `pandas` be installed within the Python
environment you are running this script in.

This file can also be imported as a module and contains the following
functions:

    * get_spreadsheet_cols - returns the column headers of the file
    * main - the main function of the script
"""

import argparse

import pandas as pd


def get_spreadsheet_cols(file_loc, print_cols=False):
    """Gets and prints the spreadsheet's header columns

    Parameters
    ----------
    file_loc : str
        The file location of the spreadsheet
    print_cols : bool, optional
        A flag used to print the columns to the console (default is
        False)

    Returns
    -------
    list
        a list of strings used that are the header columns
    """

    file_data = pd.read_excel(file_loc)
    col_headers = list(file_data.columns.values)

    if print_cols:
        print("\n".join(col_headers))

    return col_headers


def main():
    parser = argparse.ArgumentParser(description=__doc__)
    parser.add_argument(
        'input_file',
        type=str,
        help="The spreadsheet file to pring the columns of"
    )
    args = parser.parse_args()
    get_spreadsheet_cols(args.input_file, print_cols=True)


if __name__ == "__main__":
    main()
```


Docstring Formats

You may have noticed that, throughout the examples given in this tutorial, there has been specific formatting with common elements: `Arguments`, `Returns`, and `Attributes`. There are specific docstrings formats that can be used to help docstring parsers and users have a familiar and known format. The formatting used within the examples in this tutorial are NumPy/SciPy-style docstrings. Some of the most common formats are the following:

Formatting Type	Description	Supported by Sphynx	Formal Specification
Google docstrings	Google's recommended form of documentation	Yes	No
reStructured Text	Official Python documentation standard; Not beginner friendly but feature rich	Yes	Yes
NumPy/SciPy docstrings	NumPy's combination of reStructured and Google Docstrings	Yes	Yes
Epytext	A Python adaptation of Epydoc; Great for Java developers	Not officially	Yes

The selection of the docstring format is up to you, but you should stick with the same format throughout your document/project. The following are examples of each type to give you an idea of how each documentation format looks.

Google Docstrings Example

Python

```
"""Gets and prints the spreadsheet's header columns

Args:
    file_loc (str): The file location of the spreadsheet
    print_cols (bool): A flag used to print the columns to the console
        (default is False)

Returns:
    list: a list of strings representing the header columns
"""
```

reStructured Text Example

Python

```
"""Gets and prints the spreadsheet's header columns

:param file_loc: The file location of the spreadsheet
:type file_loc: str
:param print_cols: A flag used to print the columns to the console
    (default is False)
:type print_cols: bool
:returns: a list of strings representing the header columns
:rtype: list
"""
```

NumPy/SciPy Docstrings Example

Python

```
"""Gets and prints the spreadsheet's header columns

Parameters
----------
file_loc : str
    The file location of the spreadsheet
print_cols : bool, optional
    A flag used to print the columns to the console (default is False)

Returns
-------
list
    a list of strings representing the header columns
"""
```

Epytext Example

```python
"""Gets and prints the spreadsheet's header columns

@type file_loc: str
@param file_loc: The file location of the spreadsheet
@type print_cols: bool
@param print_cols: A flag used to print the columns to the console
    (default is False)
@rtype: list
@returns: a list of strings representing the header columns
"""
```

Documenting Your Python Projects

Python projects come in all sorts of shapes, sizes, and purposes. The way you document your project should suit your specific situation. Keep in mind who the users of your project are going to be and adapt to their needs. Depending on the project type, certain aspects of documentation are recommended. The general layout of the project and its documentation should be as follows:

```
project_root/
|
├── project/  # Project source code
├── docs/
├── README
├── HOW_TO_CONTRIBUTE
├── CODE_OF_CONDUCT
├── examples.py
```

Projects can be generally subdivided into three major types: Private, Shared, and Public/Open Source.

Private Projects

Private projects are projects intended for personal use only and generally aren't shared with other users or developers. Documentation can be pretty light on these types of projects. There are some recommended parts to add as needed:

- **Readme:** A brief summary of the project and its purpose. Include any special requirements for installation or operating the project.
- **`examples.py`:** A Python script file that gives simple examples of how to use the project.

Remember, even though private projects are intended for you personally, you are also considered a user. Think about anything that may be confusing to you down the road and make sure to capture those in either comments, docstrings, or the readme.

Shared Projects

Shared projects are projects in which you collaborate with a few other people in the development and/or use of the project. The "customer" or user of the project continues to be yourself and those limited few that use the project as well.

Documentation should be a little more rigorous than it needs to be for a private project, mainly to help onboard new members to the project or alert contributors/users of new changes to the project. Some of the recommended parts to add to the project are the following:

- **Readme:** A brief summary of the project and its purpose. Include any special requirements for installing or operating the project. Additionally, add any major changes since the previous version.
- **`examples.py`:** A Python script file that gives simple examples of how to use the projects.
- **How to Contribute:** This should include how new contributors to the project can start contributing.

Public and Open Source Projects

Public and Open Source projects are projects that are intended to be shared with a large group of users and can involve large development teams. These projects should place as high of a priority on project documentation as the actual development of the project itself. Some of the recommended parts to add to the project are the following:

- **Readme:** A brief summary of the project and its purpose. Include any special requirements for installing or operating the projects. Additionally, add any major changes since the previous version. Finally, add links to further documentation, bug reporting, and any other important information for the project. Dan Bader has put together a great tutorial on what all should be included in your readme.
- **How to Contribute:** This should include how new contributors to the project can help. This includes developing new features, fixing known issues, adding documentation, adding new tests, or reporting issues.
- **Code of Conduct:** Defines how other contributors should treat each other when developing or using your software. This also states what will happen if this code is broken. If you're using Github, a Code of Conduct template can be generated with recommended wording. For Open Source projects especially, consider adding this.
- **License:** A plaintext file that describes the license your project is using. For Open Source projects especially, consider adding this.
- **docs:** A folder that contains further documentation. The next section describes more fully what should be included and how to organize the contents of this folder.

The Four Main Sections of the `docs` Folder

Daniele Procida gave a wonderful PyCon 2017 talk and subsequent blog post about documenting Python projects. He mentions that all projects should have the following four major sections to help you focus your work:

- **Tutorials**: Lessons that take the reader by the hand through a series of steps to complete a project (or meaningful exercise). Geared towards the user's learning.
- **How-To Guides**: Guides that take the reader through the steps required to solve a common problem (problem-oriented recipes).
- **References**: Explanations that clarify and illuminate a particular topic. Geared towards understanding.
- **Explanations**: Technical descriptions of the machinery and how to operate it (key classes, functions, APIs, and so forth). Think Encyclopedia article.

The following table shows how all of these sections relates to each other as well as their overall purpose:

	Most Useful When We're Studying	**Most Useful When We're Coding**
Practical Step	*Tutorials*	*How-To Guides*
Theoretical Knowledge	*Explanation*	*Reference*

In the end, you want to make sure that your users have access to the answers to any questions they may have. By organizing your project in this manner, you'll be able to answer those questions easily and in a format they'll be able to navigate quickly.

those questions easily and in a format they'll be able to navigate quickly.

Documentation Tools and Resources

Documenting your code, especially large projects, can be daunting. Thankfully there are some tools out and references to get you started:

Tool	Description
Sphinx	A collection of tools to auto-generate documentation in multiple formats
Epydoc	A tool for generating API documentation for Python modules based on their docstrings
Read The Docs	Automatic building, versioning, and hosting of your docs for you
Doxygen	A tool for generating documentation that supports Python as well as multiple other languages
MkDocs	A static site generator to help build project documentation using the Markdown language
pycco	A "quick and dirty" documentation generator that displays code and documentation side by side. Check out our tutorial on how to use it for more info.

Along with these tools, there are some additional tutorials, videos, and articles that can be useful when you are documenting your project:

1. Carol Willing - Practical Sphinx - PyCon 2018
2. Daniele Procida - Documentation-driven development - Lessons from the Django Project - PyCon 2016
3. Eric Holscher - Documenting your project with Sphinx & Read the Docs - PyCon 2016
4. Titus Brown, Luiz Irber - Creating, building, testing, and documenting a Python project: a hands-on HOWTO - PyCon 2016
5. Restructured Text Official Documentation
6. Sphinx's reStructured Text Primer

Sometimes, the best way to learn is to mimic others. Here are some great examples of projects that use documentation well:

- **Django:** Docs (Source)
- **Requests:** Docs (Source)
- **Click:** Docs (Source)
- **Pandas:** Docs (Source)

Where Do I Start?

The documentation of projects have a simple progression:

1. No Documentation
2. Some Documentation
3. Complete Documentation
4. Good Documentation
5. Great Documentation

If you're at a loss about where to go next with your documentation, look at where your

project is now in relation to the progression above. Do you have any documentation? If not, then start there. If you have some documentation but are missing some of the key project files, get started by adding those.

In the end, don't get discouraged or overwhelmed by the amount of work required for documenting code. Once you get started documenting your code, it becomes easier to keep going. Feel free to comment if you have questions or reach out to the Real Python Team on social media, and we'll help.

Mark as Completed

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding:
Documenting Python Code: A Complete Guide

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.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About **James Mertz**

James is a passionate Python developer at NASA's Jet Propulsion Lab who also writes on the side for Real Python.

» More about James

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

 Dan

 Joanna

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

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

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

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

Keep Learning

Keep Learning

Related Tutorial Categories: best-practices intermediate python

Recommended Video Course: Documenting Python Code: A Complete Guide