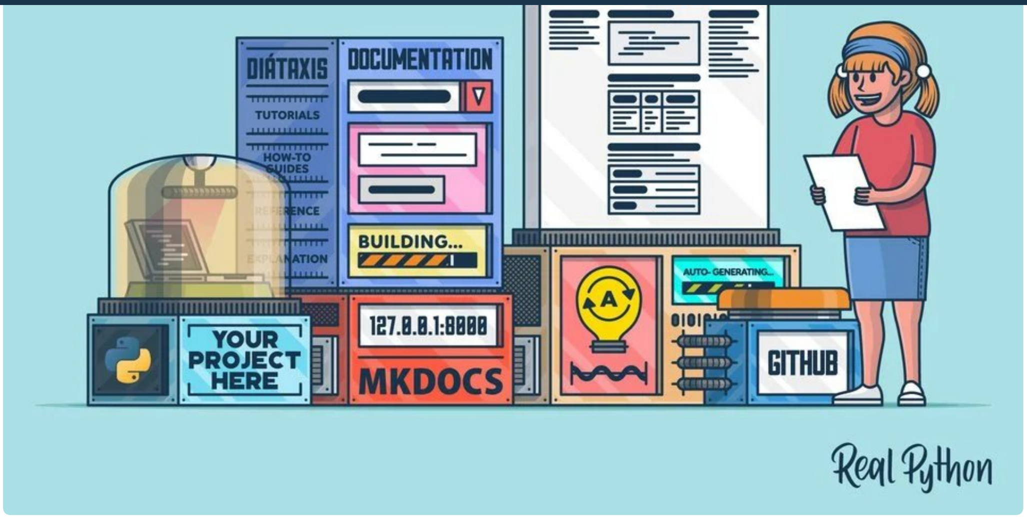


Build Your Python Project Documentation With MkDocs

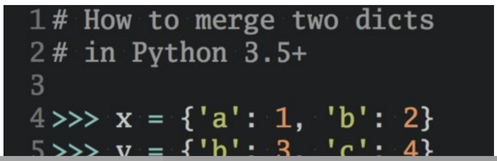
by Martin Breuss Jun 15, 2022 14 Comments basics projects python

Mark as Completed

Tweet Share Email

Table of Contents

Improve Your Python



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

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

Improve Your Python

...with a fresh Python Trick

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: **Building Python Project Documentation With MkDocs**

In this tutorial, you'll learn how to quickly build **documentation** for a Python package using **MkDocs** and **mkdocstrings**. These tools allow you to generate nice-looking and modern documentation from **Markdown** files and your code's **docstrings**.

Maintaining **auto-generated documentation** means less effort because you're linking information between your code and the documentation pages. However, good documentation is *more* than just the technical description pulled from your code! Your project will appeal more to users if you guide them through examples and connect the dots between the docstrings.

The **Material for MkDocs theme** makes your documentation **look good** without any extra effort and is used by popular projects such as Typer CLI and FastAPI.

In this tutorial, you'll:

- Work with **MkDocs** to produce **static pages from Markdown**
- Pull in **code documentation from docstrings** using **mkdocstrings**
- Follow **best practices** for project documentation
- Use the **Material for MkDocs theme** to make your documentation look good
- **Host** your documentation on **GitHub Pages**

If you use the auto-generation features of MkDocs together with mkdocstrings, then you can create good documentation with less effort. Start your documentation with docstrings in your code, then build it into a deployed and user-friendly online resource that documents your Python project.

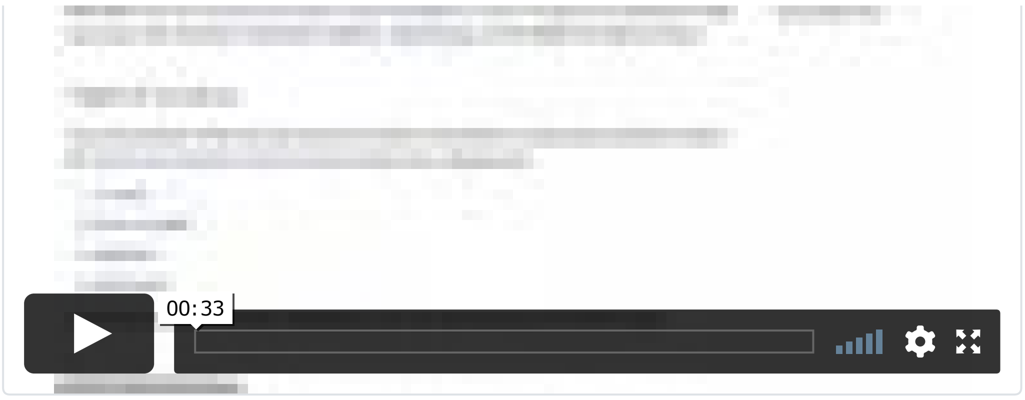
Ready to go? Then click the link below to get the source code for the project:

Get Source Code: Click here to get access to the source code that you'll use to build your documentation.

Demo

In this tutorial, you'll build project documentation that's partly auto-generated from docstrings in your code. The example code package is intentionally simplistic, so you can focus your attention on learning how to use MkDocs and the associated libraries.

After you set up your project documentation locally, you'll learn how to host it on GitHub Pages, so it'll be available for everyone to see:



You can use the example project documentation that you'll build in this tutorial as a blueprint to create documentation for your own Python projects.

Project Overview

You'll build project documentation for a toy package called `calculator` that contains only one module named `calculations.py`, which has a couple of example Python functions.

> **Note:** The provided code doesn't offer any new functionality and is only meant as a basis to learn how to use existing project code to build your documentation.

You'll follow a guideline for project documentation called the Diátaxis documentation framework, which has widespread adoption in the Python community and is used by large projects such as Django and NumPy.

This system suggests splitting up your documentation into four different parts with different orientations:

1. **Tutorials**: Learning-oriented
2. **How-To Guides**: Problem-oriented
3. **Reference**: Information-oriented

4. **Explanation:** Understanding-oriented

Splitting your project documentation into these four different purposes with different orientations will help you create comprehensive documentation for your Python project.

From a technical perspective, you'll build your documentation using three Python packages:

1. MkDocs for building static pages from Markdown
2. mkdocstrings for auto-generating documentation from docstrings in your code
3. Material for MkDocs for styling your documentation

When you want to use MkDocs for auto-generating parts of your documentation from your docstrings, you'll *need* to add the mkdocstrings package.

Note: Sphinx, another popular tool for Python project documentation, can auto-generate text from your docstrings without additional extensions. However, Sphinx primarily uses reStructuredText instead of Markdown and is overall less straightforward to work with than MkDocs.

You don't absolutely need to add the Material for MkDocs theme for building your project documentation, but it'll help to render the documentation in a user-friendly manner.

Prerequisites

To complete this tutorial, you should be comfortable with the following concepts:

- **Virtual environments:** Set up a Python virtual environment and work within it.
- **Package management:** Install Python packages with `pip` and create your own modules and packages.
- **Code documentation:** Understand what docstrings are, what

structure they follow, what information they should contain, and how you should document a Python project in general.

- **GitHub repositories:** Create and update a repo on GitHub to host your documentation, which you can learn about in the intro to Git and GitHub.

If you don’t have all of the prerequisite knowledge before starting this tutorial, that’s okay! You might learn more by going ahead and getting started. You can always stop and review the resources linked here if you get stuck.

Step 1: Set Up Your Environment for Building Documentation

Before you start working on your project, you need a virtual environment into which you can install the packages you need.

At the end of this step, you’ll have created and activated a virtual environment and installed the necessary dependencies.

Start by creating a new folder for your project that you call `mkdocs-documentation/`. Navigate into the folder, and then create a virtual environment and install the necessary packages:

Windows | Linux + macOS

Windows PowerShell

```
PS> mkdir mkdocs-documentation
PS> cd mkdocs-documentation
PS> python -m venv venv
PS> venv\Scripts\activate
(venv) PS> python -m pip install mkdocs
(venv) PS> python -m pip install "mkdocstrings[python]"
(venv) PS> python -m pip install mkdocs-material
```

Wait for all downloads to finish, then confirm that the installation worked by calling `pip list`. The command will give you a list of all installed packages alongside their versions:

Shell

```
(venv) $ python -m pip list
Package                    Version
-------------------------- -------
click                      8.1.3

# ...

mkdocs                     1.3.0
mkdocs-autorefs            0.4.1
mkdocs-material            8.3.6
mkdocs-material-extensions 1.0.3
mkdocstrings               0.19.0
mkdocstrings-python        0.7.1

# ...

zipp                       3.8.0
```

Confirm that the packages highlighted in the output above are installed. You should see four packages that are directly related to your installation command:

1. `mkdocs`
2. `mkdocs-material`
3. `mkdocstrings`
4. `mkdocstrings-python`

The package called `mkdocstrings-python` is the Python handler for mkdocstrings that allows mkdocstrings to parse Python code. You installed it by adding the extension `[python]` when installing the `mkdocstrings` package with `pip`.

You'll see some additional packages in your list, which shows that your virtual environment and the three installed packages come with additional dependencies that `pip` installed for you automatically. You can pin your dependencies in a `requirements.txt` file for reproducibility.

Step 2: Create the Sample Python Package

In this step, you'll write a sample Python package called `calculator`, which is what you'll be building documentation for.

To document a Python project, you first need a Python project. In this tutorial, you use a toy implementation of a calculator package that returns floating-point numbers.

> **Note:** If you already have a Python project for which you want to generate your documentation, then feel free to use your project instead. In this case, you can skip this part and jump directly to Step 3: Write and Format Your Docstrings.

Your example Python project will be small and consist only of one folder with two files in it:

```
calculator/
│
├── __init__.py
└── calculations.py
```

Create this package folder and the two files right inside your project folder, then open `calculations.py` in your favorite code editor or IDE.

Add the example code for some mathematical calculations that you'll use as the basis for your documentation:

Python

```python
# calculator/calculations.py

def add(a, b):
    return float(a + b)

def subtract(a, b):
    return float(a - b)

def multiply(a, b):
    return float(a * b)

def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("division by zero")
    return float(a / b)
```

The code you added to `calculations.py` repackages some fundamental math operations into new functions that return the result as a floating-point number.

Keep the second file, `__init__.py`, empty for now. It's here to help declare `calculator` as a package. Later you'll add package-level docstrings in this file, which you'll also pull into your auto-generated documentation.

In this step, you've created the sample Python project that you'll use as an example project to build your documentation for. In the next step, you'll add docstrings to your functions to set yourself up for success later on, when you go to generate documentation from these docstrings.

Step 3: Write and Format Your Docstrings

The mkdocstrings package can pull valuable information from your codebase to help auto-generate parts of your documentation. As the package name suggests, you'll need docstrings to accomplish this.

You don't yet have any docstrings in your functions, and it's time to change that. You'll write docstrings for your module, the contained functions, and your package in this step. You'll also reformat your function signatures and docstrings to use type hints.

Understand Python Docstrings

Docstrings are your biggest help for documenting your Python code.

They’re built-in strings that you can configure to hold usage instructions and information about your functions, classes, and modules.

A Python docstring consists of text in between a pair of three double quotation marks (`"""`). Most commonly, you’ll read and write function, class, and method docstrings. In these cases, the docstring is located right below the line that defines the class, method, or function:

Python >>>

```python
>>> def greet(name):
...     """Print a greeting.
...
...     Args:
...         name (str): The name of the person to greet.
...     """
...     print(f"Hello {name}!")
...
```

The code snippet above shows an example docstring for a function called `greet()`. The docstring starts with a single-line description of the function’s purpose, followed by more in-depth information:

> The docstring for a function or method should summarize its behavior and document its arguments, return value(s), side effects, exceptions raised, and restrictions on when it can be called (all if applicable). (Source)

Docstrings can help to make the code that you’re working with easier to understand. They provide information about code objects. If you write your docstrings well, then they clarify the context and use of an object.

You can access the information saved in a docstring using the built-in `help()` function:

Python >>>

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

```
greet(name)
    Print a greeting.

    Args:
        name (str): The name of the person to greet.
```

If you call `help()` on any code object, then Python will print the object's docstring to your terminal.

An object's docstring is saved in `.__doc__`, and you can also inspect it there directly:

Python >>>

```python
>>> greet.__doc__
'Print a greeting.\n    \n    Args:\n        name (str):
↳ The name of the person to greet.\n    '
```

This attribute contains your docstring, and you could read any docstring with `.__doc__`. However, you'll usually access it through the more convenient `help()` function. Displaying a docstring with `help()` also improves the formatting.

Note: Python uses the built-in pydoc module to generate the formatted description from `.__doc__` when you call `help()`.

Other types of docstrings, for example module and package docstrings, use the same triple double-quote syntax. You place a module docstring right at the beginning of a file, and you write a package docstring at the beginning of an `__init__.py` file.

These docstrings provide high-level information about the module or package:

> The docstring for a module should generally list the classes, exceptions and functions (and any other objects) that are exported by the module, with a one-line summary of each. (These summaries generally give less detail than the summary line in the object's docstring.) The docstring for a package

> (i.e., the docstring of the package's `__init__.py` module) should also list the modules and subpackages exported by the package. (Source)

The basic syntax for all Python docstrings is the same, although you'll find them in different locations based on what the docstring is documenting.

> **Note:** In this tutorial, you'll create function, module, and package docstrings. If your personal Python project contains classes, then you should document these as well using class docstrings.

Docstrings were formalized in PEP 257, but their structure isn't strictly defined. Subsequently, different projects have developed different standards for Python docstrings.

MkDocs supports three common types of Python docstring formats:

1. Google-Style Docstrings
2. NumPy Docstring Standard
3. Sphinx Docstring Format

The Python handler for MkDocs uses Google-style docstrings by default, which is what you'll stick with for this tutorial.

Add Function Docstrings to Your Python Project

It's time to add Google-style docstrings to your example functions in `calculations.py`. Start by writing your one-line docstring, which should concisely explain the purpose of the function:

Python

```python
def add(a, b):
    """Compute and return the sum of two numbers."""
    return float(a + b)
```

After adding the initial description of your function, you can expand the docstring to describe the function arguments and the function's return value:

Python

```python
def add(a, b):
    """Compute and return the sum of two numbers.

    Args:
        a (float): A number representing the first addend in th
        b (float): A number representing the second addend in t

    Returns:
        float: A number representing the arithmetic sum of `a`
    """
    return float(a + b)
```

You can keep inspecting your functions by using `help()` to peek at the automatic documentation that Python builds from the information that you add to the function docstrings.

By describing the arguments and the return value and their types, you provide helpful usage information for programmers working with your code.

Write docstrings for all functions in `calculations.py`:

`calculations.py` Show/Hide

When you're done, you've successfully added the first line of defense for your project code documentation directly into your codebase.

But Python docstrings can do more than describe and document. You can even use them to include short test cases for your functions, which you can execute using one of Python's built-in modules. You'll add these examples in the upcoming section.

Write Examples and Test Them Using Doctest

You can add examples right in your docstrings. Doing this clarifies how to use the functions, and when you stick to a specific format, you can even test your code examples using Python's `doctest` module.

Google suggests adding examples to your docstring under a headline called `"Examples:"`, which works well for running doctests and building your documentation using MkDocs.

Head back into `calculations.py` and add example use cases to your function docstrings:

Python

```python
def add(a, b):
    """Compute and return the sum of two numbers.

    Examples:
        >>> add(4.0, 2.0)
        6.0
        >>> add(4, 2)
        6.0

    Args:
        a (float): A number representing the first addend in th
        b (float): A number representing the second addend in t

    Returns:
        float: A number representing the arithmetic sum of `a`
    """
    return float(a + b)
```

You added another header called `"Examples:"` and with an extra indentation level, you added example calls to the function you're documenting. You provided the input after the default Python REPL prompt (>>>), and you put the expected output in the next line.

These examples will render well in your auto-generated documentation and add context to your function. You can even test them! Verify that your functions work as expected by executing the file using Python's `doctest` module:

Windows | Linux + macOS

Windows PowerShell

```
(venv) PS> python -m doctest calculator\calculations.py
```

If you don't see any output, then all tests passed. Excellent work, you've successfully added doctests to your function!

Note: Try to change a number in one of the examples and run `doctest` again to explore how a failing doctest displays. Then change it back to make the test pass again.

Revisit all of your functions and add doctests in the same way as you did for `add()` further up:

`calculations.py` Show/Hide

When you finish writing doctests for all your functions, run the tests using `doctest` to confirm that all tests pass.

Well done, you're expanding your docstrings to be more comprehensive and valuable! To improve your codebase, even more, you'll next add type hints to your function definitions. Type annotations allow you to remove the type information from within your docstrings.

Use Type Hints to Provide Automatic Type

Information

As you might have noticed, in the docstrings that you've written so far, you declared that the input variables should be of the type `float`. However, the functions work just as well when you use integers. You even have proof for that through the function calls that you wrote in your doctests!

You should probably update the argument types in your docstrings accordingly. But instead of doing so in your docstrings, you'll use Python type hints to declare the argument and return types of your functions:

Python

```python
from typing import Union

def add(a: Union[float, int], b: Union[float, int]) -> float:
    """Compute and return the sum of two numbers.

    Examples:
        >>> add(4.0, 2.0)
        6.0
        >>> add(4, 2)
        6.0

    Args:
        a (float): A number representing the first addend in th
        b (float): A number representing the second addend in t

    Returns:
        float: A number representing the arithmetic sum of `a`
    """
    return float(a + b)
```

You've imported `Union` from the built-in `typing` module, which allows you to specify multiple types for an argument. Then you changed the first line of your function definition by adding type hints to your parameters and the return value.

Note: Starting with Python 3.10, you can alternatively use the pipe operator (|) as a type union alias:

Python

Python

```python
def add(a: float | int, b: float | int) -> float:
    """Compute and return the sum of two numbers.

    ...
    """
    return float(a + b)
```

This more succinct syntax also allows you to remove the import statement from the top of your file.

However, to keep the type hints more compatible with older versions of type-checking tools, you'll stick with `Union` in this example project.

Adding type hints to your code allows you to use type checkers such as `mypy` to catch type-related errors that might otherwise go unnoticed.

Note: If you want to learn more about writing type hints and type-checking your Python code using third-party libraries, then you can refresh your memory in the Python Type Checking Guide.

But wait a moment, your sixth sense is tingling! Did you notice that you've introduced some repeated information and inconsistencies regarding the types you're mentioning in the docstring?

Fortunately, mkdocstrings understands type hints and can infer typing from them. This means that you don't need to add type information to the docstring. Google-style docstrings don't have to contain type information if you use type hints in your code.

You can therefore remove the duplicated type information from your docstrings:

Python

```python
from typing import Union
```

```
def add(a: Union[float, int], b: Union[float, int]) -> float:
    """Compute and return the sum of two numbers.

    Examples:
        >>> add(4.0, 2.0)
        6.0
        >>> add(4, 2)
        6.0

    Args:
        a: A number representing the first addend in the additi
        b: A number representing the second addend in the addit

    Returns:
        A number representing the arithmetic sum of `a` and `b`
    """
    return float(a + b)
```

This change gives you a clean and descriptive docstring with an accurate representation of the expected types for your arguments and the return value of your function.

Documenting the types using type hints gives you the advantage that you can now use type checker tools to assure correct usage of your functions and hedge yourself against accidental misuse. It also allows you to record type information in only one place, which keeps your codebase DRY.

Revisit `calculations.py`, and add type hints to all your functions:

`calculations.py` Show/Hide

With these updates, you've completed writing a solid suite of docstrings for all the functions in your Python module. But what about the module itself?

Add Module Docstrings

Python docstrings aren't restricted to functions and classes. You can also use them to document your modules and packages, and mkdocstrings will extract these types of docstrings as well.

You'll add a module-level docstring to `calculations.py` and a package-level docstring to `__init__.py` to showcase this functionality. Later, you'll render both as part of your auto-generated documentation.

The docstring of `calculations.py` should give a quick overview of the module, then list all functions that it exports, together with a one-line description of each function:

Python

```python
# calculator/calculations.py

"""Provide several sample math calculations.

This module allows the user to make mathematical calculations.

The module contains the following functions:

- `add(a, b)` - Returns the sum of two numbers.
- `subtract(a, b)` - Returns the difference of two numbers.
- `multiply(a, b)` - Returns the product of two numbers.
- `divide(a, b)` - Returns the quotient of two numbers.
"""

from typing import Union
# ...
```

Add this example module docstring to the very top of `calculations.py`. You'll notice that this docstring contains Markdown formatting. MkDocs will render it to HTML for your documentation pages.

Just like for function docstrings, you can also add usage examples for your module to the docstring:

Python

```python
# calculator/calculations.py

"""Provide several sample math calculations.

This module allows the user to make mathematical calculations.
```

```
Examples:
    >>> from calculator import calculations
    >>> calculations.add(2, 4)
    6.0
    >>> calculations.multiply(2.0, 4.0)
    8.0
    >>> from calculator.calculations import divide
    >>> divide(4.0, 2)
    2.0

The module contains the following functions:

- `add(a, b)` - Returns the sum of two numbers.
- `subtract(a, b)` - Returns the difference of two numbers.
- `multiply(a, b)` - Returns the product of two numbers.
- `divide(a, b)` - Returns the quotient of two numbers.
"""

from typing import Union
# ...
```

You can test these examples as before by running `doctest` on the module. Try swapping one of the return values to see the doctest fail, and then fix it again to ensure your examples represent your module's functionality.

Finally, you'll also add a package-level docstring. You add these docstrings to the top of your package's `__init__.py` file before any exports that you'd define there.

> **Note:** In this example package, you'll export all functions defined in `calculations.py`, so `__init__.py` won't contain any Python code aside from the docstring.
>
> In your Python project, you may want to define which objects your package exports, and you'd add the code below the docstring for your package.

Open your empty `__init__.py` file and add the docstring for your `calculator` package:

Python

```
# calculator/__init__.py

"""Do math with your own functions.

Modules exported by this package:

- `calculations`: Provide several sample math calculations.
"""
```

You've added a short description of your package and the module it contains to the top of your `__init__.py` file. If your package was going to export more modules and subpackages, you'd also list them here.

After writing the docstring for your package, you completed all the docstrings that you wanted to add to your code. Your Python project's source code is well documented using docstrings and type hints, and it even contains examples that you can run as doctests.

You've finished the first line of defense for your project code's documentation, and it'll always stick with your code. You're now ready to raise the bar for your project's documentation by building user-friendly documentation pages using MkDocs.

Step 4: Prepare Your Documentation With MkDocs

At this point, you should have an activated virtual environment with all the necessary packages installed. You should also have set up your toy `calculator` package and added docstrings to your code.

In this step, you'll set up your `mkdocs.yml` file, which holds the instructions for building the documentation with MkDocs. You'll write your additional documentation pages in Markdown, including the syntax that defines where mkdocstrings will insert the auto-generated part of your documentation.

Create Your MkDocs Project Structure

With solid docstrings in your source code, you next want to make your project more accessible to a broad user base. Your project will shine more brightly if you can provide user-friendly documentation that's accessible online.

MkDocs is a Python package that allows you to build static pages using Markdown. The basic structure of an MkDocs project consists of three main components:

1. Your project code
2. All your Markdown documentation pages inside a `docs/` folder
3. A configuration file named `mkdocs.yml`

You already have your project code ready to go. Create the other two puzzle pieces next using the handy `new` command provided by MkDocs:

Shell

```
(venv) $ mkdocs new .
```

This command creates a default project structure for your MkDocs project in the current directory, which the dot (`.`) at the end of the command references.

Feel free to explore the new files and folders that it created:

```
mkdocs-documentation/
│
├── calculator/
│   ├── __init__.py
│   └── calculations.py
│
├── docs/
│   └── index.md
│
├── mkdocs.yml
```

```
└── requirements.txt
```

In just a bit, you'll edit `index.md` and expand your written documentation by adding new Markdown files to the `docs/` directory.

But first, you'll explore the `mkdocs.yml` settings file, which tells MkDocs how to handle your project documentation.

Adapt Your Project Settings File

Mkdocs uses a YAML file for configuration. When you create a new project using `new`, MkDocs creates a bare-bones `mkdocs.yml` file for you:

YAML

```yaml
# mkdocs.yml
site_name: My Docs
```

The default settings file only contains one element, `site_name`, which defines the default name *My Docs* for your documentation.

Of course, you don't want your project to keep that name, so you'll change it to *Calculation Docs* instead. As a connoisseur of Google's Material Design, you also want your documentation to look great right away.

By adding a second element to your YAML settings file, you can replace the default theme with the popular Material for MkDocs theme, which you installed at the beginning of this tutorial:

YAML

```yaml
# mkdocs.yml
site_name: Calculation Docs

theme:
  name: "material"
```

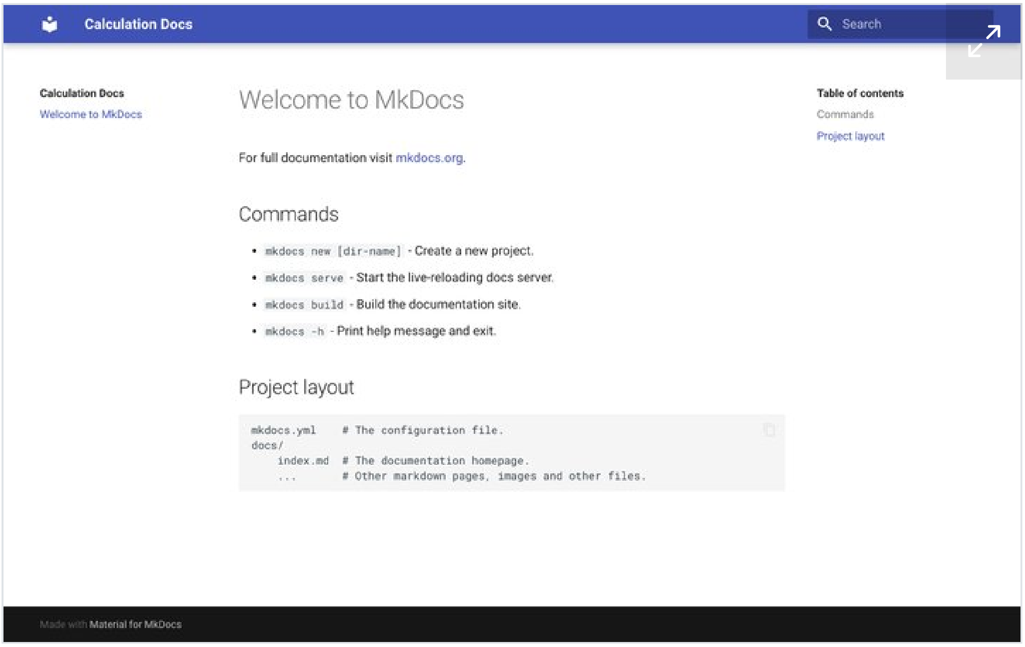
Once you've adapted the settings file like this, you can take a look at the current state of your boilerplate documentation by building the site:

Shell

```
(venv) $ mkdocs serve
INFO     -  Building documentation...
INFO     -  Cleaning site directory
INFO     -  Documentation built in 0.22 seconds
[I 220510 0:0:0 server:335] Serving on http://127.0.0.1:8000
INFO     -  Serving on http://127.0.0.1:8000
```

As soon as your terminal tells you that it's serving the documentation on your localhost, as shown above, you can view it in your browser.

The information printed to your terminal tells you that MkDocs is serving your documentation at `http://127.0.0.1:8000`. Open a new browser tab pointing to that URL. You'll see the MkDocs boilerplate index page with your custom title, styled with the Material for MkDocs theme:



Welcome to MkDocs

For full documentation visit mkdocs.org.

Commands

- `mkdocs new [dir-name]` - Create a new project.
- `mkdocs serve` - Start the live-reloading docs server.
- `mkdocs build` - Build the documentation site.
- `mkdocs -h` - Print help message and exit.

Project layout

```
mkdocs.yml    # The configuration file.
docs/
    index.md  # The documentation homepage.
    ...       # Other markdown pages, images and other files.
```

If you want to know where all the text you see is stored, you can open up `index.md`. You can edit this page and see the changes automatically reflected in your browser.

Note: While you keep working on your documentation, you can keep coming back to your localhost in your browser to view the changes. If you don't see an update, then stop the server and rebuild the site using the `mkdocs serve` command.

You've made two adaptations in `mkdocs.yml` to change the look and feel of your documentation. However, the *content* of your docs is still just pre-built boilerplate text that isn't related to your Python project. It's time to fix that.

Create Static Pages From Markdown

Setting up a new MkDocs project creates a default `index.md` page in `docs/`. The index page is the default entry point for your project documentation, and you can edit the text in this file to fit your project landing page. You can also add more Markdown files to the `docs/` folder, and each of them will render into a new page of your documentation.

As you learned in the project overview, you'll follow the structure proposed in the Diátaxis documentation framework, which suggests splitting your documentation into four distinct parts:

1. Tutorials
2. How-To Guides
3. Reference
4. Explanation

Excellent project documentation doesn't consist only of nicely rendered function docstrings!

Note: To build excellent documentation for your project, you can draw inspiration from the Diátaxis resources. You'll create drafts for all four of the mentioned parts in the course of this tutorial.

To set up a structure for your project documentation, create four additional Markdown files representing the different parts:

1. `docs/tutorials.md`
2. `docs/how-to-guides.md`
3. `docs/reference.md`
4. `docs/explanation.md`

After adding these four files, your `docs/` folder will contain five Markdown files:

```
docs/
├── explanation.md
├── how-to-guides.md
├── index.md
├── reference.md
└── tutorials.md
```

MkDocs builds every Markdown file that it finds in `docs/` as a separate page. The first page that shows up is always `index.md`. All remaining pages show up in the order listed in `docs/`.

Files are listed alphabetically by default, but you'd like to preserve the order proposed by the Diátaxis documentation framework.

To determine a custom order for your documentation pages, you need to add the `nav` element to your settings file and list all files in the order in which you want to show them:

YAML

```yaml
# mkdocs.yml
site_name: Calculation Docs

theme:
  name: "material"

nav:
  - index.md
  - tutorials.md
  - how-to-guides.md
  - reference.md
  - explanation.md
```

You’ve added the filenames for all your documentation pages under the `nav` element with appropriate indentation. You can now click through your documentation in the intended order on your localhost page.

You might have noticed that each page already has a title, which MkDocs inferred from the filenames. If you don’t like a page’s title, you can optionally add another element in front of the filename whose title you want to change:

YAML

```yaml
# mkdocs.yml
site_name: Calculation Docs

theme:
  name: "material"

nav:
  - Calculation Docs: index.md
  - tutorials.md
  - How-To Guides: how-to-guides.md
  - reference.md
  - explanation.md
```

With the order and the titles updated in your settings file, you can now fill your documentation with information about your package.

Feel free to practice writing your own documentation pages, or copy the content of the files below to see an example of how MkDocs does a great job at rendering your Markdown text to a styled web page:

`index.md` Show/Hide

`tutorials.md` Show/Hide

`how-to-guides.md` Show/Hide

`reference.md` Show/Hide

`explanation.md` Show/Hide

You built an excellent start for your Python project documentation! By using MkDocs, you can write your text in Markdown and render it nicely for the Internet.

But so far, you haven't connected the information in your docstrings with the documentation rendered by MkDocs. You'll integrate your docstrings into your front-end documentation as your next task.

Insert Information From Docstrings

Keeping documentation up to date can be challenging, so auto-generating at least parts of your project documentation can save you time and effort.

MkDocs is a static-site generator geared toward writing documentation. However, you can't fetch docstring information from your code using MkDocs alone. You can make it work with an additional package called mkdocstrings.

You already installed mkdocstrings into your virtual environment at the beginning of this tutorial, so you only need to add it as a plugin to your MkDocs configuration file:

YAML

```yaml
# mkdocs.yml
```

```yaml
site_name: Calculation Docs

theme:
  name: "material"

plugins:
  - mkdocstrings

nav:
  - Calculation Docs: index.md
  - tutorials.md
  - How-To Guides: how-to-guides.md
  - reference.md
  - explanation.md
```

By recording mkdocstrings as a list item to the `plugins` element, you activated the plugin for this project.

Mkdocstrings allows you to insert docstring information right into your Markdown pages using a special syntax of three colons (`:::`) followed by the code identifier that you want to document:

Markdown Text

```
::: identifier
```

Because you've already written your code documentation in your docstrings, you now only need to add these identifiers to your Markdown documents.

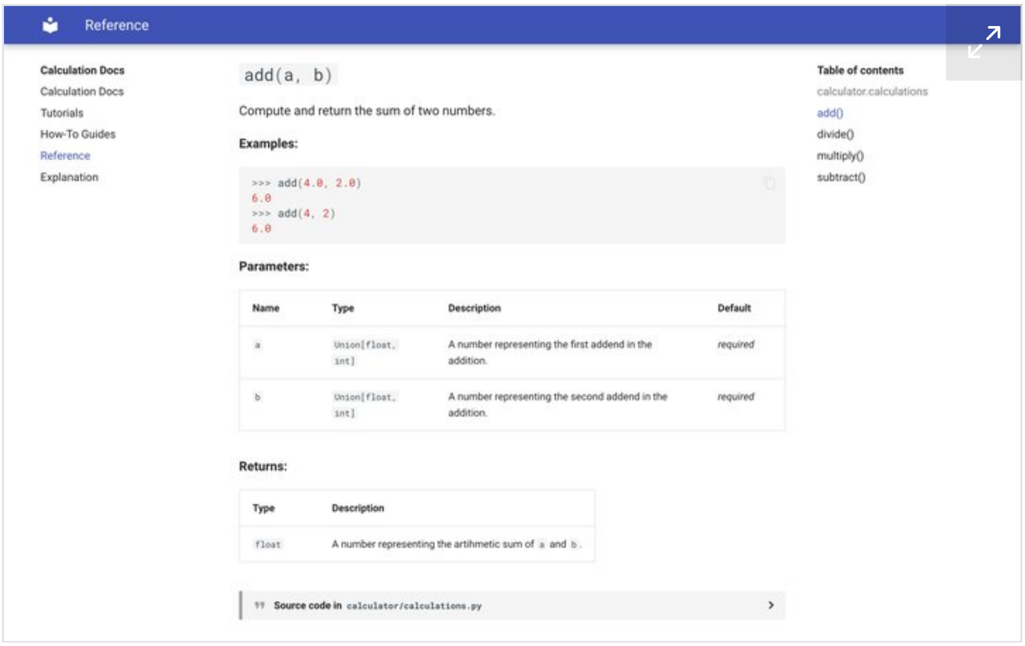
The central part of your code reference goes into `reference.md`, and you'll let mkdocstrings add it for you automatically based on your docstrings:

Markdown Text

```markdown
This part of the project documentation focuses on
an **information-oriented** approach. Use it as a
reference for the technical implementation of the
`calculator` project code.

::: calculator.calculations
```

You've only added a single line to the Markdown file, but if you view the reference page on your localhost, you can see that mkdocstrings gathered all the information from your docstrings in `calculator/calculations.py` and rendered them:



`add(a, b)`

Compute and return the sum of two numbers.

Examples:

```
>>> add(4.0, 2.0)
6.0
>>> add(4, 2)
6.0
```

Parameters:

Name	Type	Description	Default
`a`	`Union[float, int]`	A number representing the first addend in the addition.	*required*
`b`	`Union[float, int]`	A number representing the second addend in the addition.	*required*

Returns:

Type	Description
`float`	A number representing the artihmetic sum of `a` and `b`.

Source code in `calculator/calculations.py`

You may notice that mkdocstrings pulled information from your type hints and the function and module-level docstrings and now presents them to you in a user-friendly manner.

It also created clickable links in the navigation panel on the right to jump to any function definition with a single click. It also generated a collapsible section that contains the source code of the relevant function.

The work you did when writing your docstrings is paying off! The best part is that you'll only need to keep the documentation right inside your codebase up to date. You can continually update the user-facing documentation that you built with MkDocs from your docstrings.

Instead of rendering all your module information on the reference page, you can also render just the package docstring that you recorded in `__init__.py` by noting the name of your package as the identifier:

```
index.md                                    Show/Hide
```

If you need to update your docstrings because you changed your project code, then you just need to rebuild your documentation to propagate the updates to your user-facing documentation.

If you integrate mkdocstrings into your project documentation workflow, then you can avoid repetition and reduce the effort needed to keep your documentation updated.

Step 5: Build Your Documentation With MkDocs

At this point, you should've written all your documentation pages and the project structure file. At the end of this step, you'll have built your documentation and be ready to deploy it online.

You already built your documentation using the `serve` command. This command builds a development version of your documentation and makes it available locally in your browser at `http://127.0.0.1:8000`. Serving your documentation like that is helpful during development because any changes you apply will update automatically.

However, once you've finished developing your documentation, you'll want to build it without starting a server on localhost. After all, MkDocs is a static-site generator that allows you to create documentation that you can host *without* running a server!

To just build your documentation and create the `site/` directory that'll contain all the necessary assets and static files that'll allow you to host your documentation online, you can use the `build` command:

Shell

```
(venv) $ mkdocs build
```

When you build your documentation with this command, MkDocs creates a `site/` directory that contains your documentation converted to HTML pages, as well as all the static assets that are necessary to build the Material for MkDocs theme:

```
site/
│
├── assets/
│   │
│   ├── ...
│   │
│   └── _mkdocstrings.css
│
├── explanation/
│   └── index.html
│
├── how-to-guides/
│   └── index.html
│
├── reference/
│   └── index.html
│
├── search/
│   └── search_index.json
│
├── tutorials/
│   └── index.html
│
├── 404.html
├── index.html
├── objects.inv
├── sitemap.xml
└── sitemap.xml.gz
```

This folder structure is a self-contained static representation of your documentation that'll look just like what you previously saw on your localhost during development. You can now host it on any static-site hosting service to get your documentation in front of your users.

Step 6: Host Your Documentation on GitHub

At this point, you've completed your toy `calculator` project's documentation, which was partly auto-generated from the docstrings in your code.

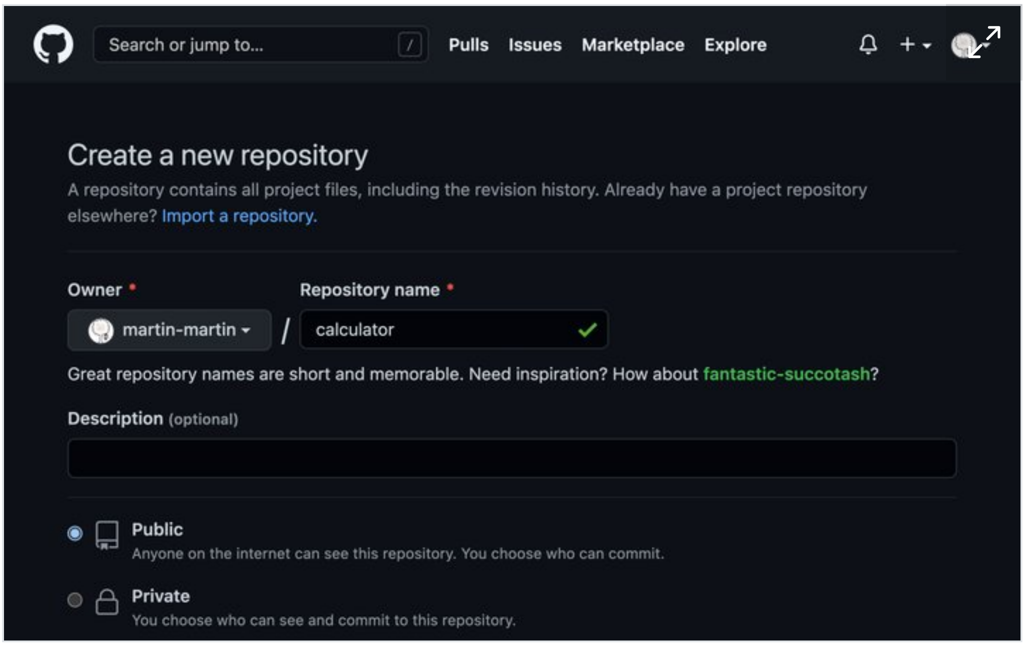
In this step, you'll deploy your documentation to GitHub and add additional files that should be part of a complete Python project documentation.

While you could host the documentation that you built using MkDocs on any static file hosting service, you'll learn to do it using GitHub Pages. As a developer, you probably already have a GitHub account, and the platform also offers some excellent features for adding additional parts to your project documentation from boilerplate.

Create a GitHub Repository

If you're already hosting your Python project code on GitHub, then you can skip this part and continue with deploying your documentation.

If you don't have a GitHub repository for your project yet, then create a new repository through the GitHub web interface:



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner * martin-martin / Repository name * calculator

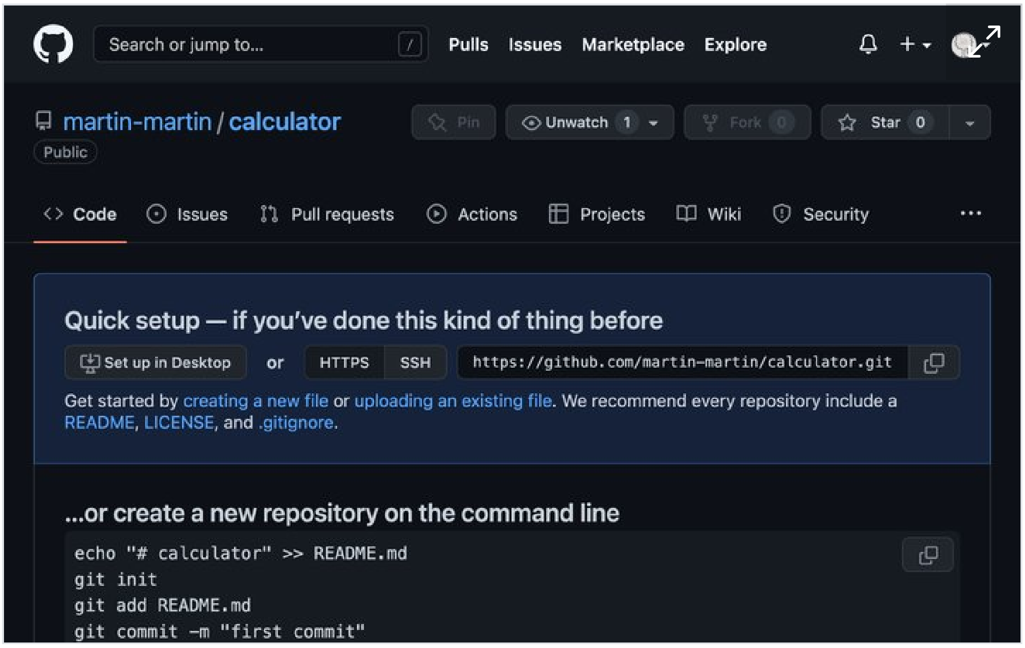
Great repository names are short and memorable. Need inspiration? How about fantastic-succotash?

Description (optional)

[✓ Public] Anyone on the internet can see this repository. You choose who can commit.

Private You choose who can see and commit to this repository.

Initialize it *without* a `README.md` file so that it starts empty, then copy the URL of the repository:



Quick setup — if you've done this kind of thing before

Set up in Desktop or HTTPS SSH https://github.com/martin-martin/calculator.git

Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.

...or create a new repository on the command line

```
echo "# calculator" >> README.md
git init
git add README.md
git commit -m "first commit"
```

Back on your terminal, initialize a local Git repository for your Python project:

Shell

```
(venv) $ git init
```

After successfully initializing an empty Git repository at your project root, you can next add the URL to your GitHub repository as a remote:

Shell

```
(venv) $ git remote add origin https://github.com/your-username
```

After adding your GitHub repository as a remote to your local Git repository, you can add all project files and push everything to your remote:

Shell

```
(venv) $ git add .
(venv) $ git commit -m "Add project code and documentation"
(venv) $ git push origin main
```

These commands add all files in the current folder to Git's staging area, commit the files to version control with a commit message, and push them to your remote GitHub repository.

Note: Your local default branch for Git might be called `master` instead of `main`. If that's the case, then you can rename your local branch:

Shell

```
(venv) $ git branch -m master main
```

Run this command to rename your local Git branch from `master` to `main`. Then, you can run `push` again to send your project files to your remote GitHub repository.

Next, you can push the documentation you built using MkDocs to a particular branch on your GitHub repository, immediately making it available for your users to browse online.

Deploy Your Documentation to GitHub

GitHub repositories automatically serve static content when committed to a branch named `gh-pages`. MkDocs integrates with that and allows you to build and deploy your project documentation in a single step:

Shell

```
(venv) $ mkdocs gh-deploy
```

Running this command rebuilds the documentation from your Markdown files and source code and pushes it to the `gh-pages` branch on your remote GitHub repository.

Because of GitHub's default configuration, that'll make your documentation available at the URL that MkDocs shows you at the end of your terminal output:

Shell

```
INFO - Your documentation should shortly be available at:
       https://user-name.github.io/project-name/
```

To explore your project documentation online, head over to your browser and visit the URL that's shown in your terminal output.

Note: If you get a 404 error when visiting the URL, take it as an opportunity for a quick break. It can take up to ten minutes until your documentation goes live when you create or publish your GitHub Pages site.

Once you've got a remote GitHub repository set up for your project code, this is a quick way to get your documentation live on the Internet.

Nice work! You now have a well-structured base for your project documentation, which is partly auto-generated from your docstrings using mkdocstrings, and built for front-end consumption with MkDocs. You even made it available online through GitHub Pages!

Conclusion

In this tutorial, you learned how to quickly build **documentation** for a Python package based on Markdown files and docstrings using **MkDocs** and **mkdocstrings**.

You created a partly **auto-generated documentation** for your Python project by linking information in your docstrings to the documentation pages. Not content to rest at this, you added additional documentation pages that made your project more appealing to users by guiding them through examples and use cases.

You styled your documentation with the **Material for MkDocs theme** and deployed it to the Internet through **GitHub Pages**.

In this tutorial, you learned how to:

- Write **docstrings** for your code objects
- Work with **MkDocs** to produce **static pages from Markdown**
- Pull in **code documentation from docstrings** using **mkdocstrings**
- Follow **best practices** for project documentation
- Customize your documentation using the **Material for MkDocs** theme
- **Deploy** your documentation on **GitHub Pages**

Building your project documentation using MkDocs and mkdocstrings allows you to write Markdown and auto-generate parts of the documentation directly from your docstrings. This setup means that you can create excellent documentation with less effort.

To review the source code for this project, click the link below:

Get Source Code: Click here to get access to the source code that you'll use to build your documentation.

Next Steps

You can use the same approach outlined in this tutorial to document your own Python project. Follow the tutorial a second time, but instead of using the `calculator` module, write the documentation for your own package. This process will help train your understanding of how to create helpful documentation.

Additionally, here are some ideas to take your project documentation to the next level:

- **Fill All Four Corners:** Train writing all aspects of wholesome project documentation by filling the information for the

Tutorials and the *Explanation* pages. The documentation outlined in this tutorial only creates examples for the *Reference* and *How-to Guides* pages of the Díataxis documentation system.

- **Add Complementary Documentation:** Use GitHub's templates for adding a code of conduct, a license, and contributing guidelines to build out your project documentation even more.
- **Tune Your Configuration:** Explore advanced configuration for MkDocs, such as adding support for search and multiple languages. Or install and include additional plugins. A good option is autorefs, which allows you to add relative links in your docstrings that work in your rendered documentation.
- **Customize the Material for MkDocs Theme:** Add your own CSS stylesheet or JavaScript file to adapt the Material for MkDocs theme with advanced customization
- **Automate Your Deployment:** Use GitHub Actions to automate the deployment of your documentation when you apply any edits.
- **Publish as a Package:** Create a package from your project code and publish it to PyPI. Linking back to your professional project documentation will give it a better standing among end users.

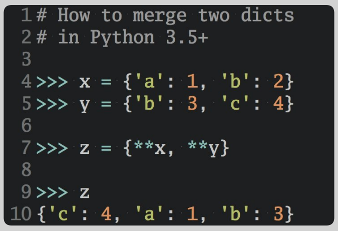
What other ideas can you come up with to improve your project documentation and make it less effort for you to keep it up-to-date? Be creative, have fun, and leave a comment below!

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: **Building Python Project Documentation With MkDocs**

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

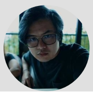
Send Me Python Tricks »

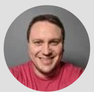
About **Martin Breuss**

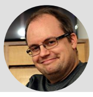
Martin likes automation, goofy jokes, and snakes, all of which fit into the Python community. He enjoys learning and exploring and is up for talking about it, too. He writes and records content for Real Python and CodingNomads.

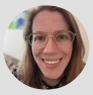
» More about Martin

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

 Aldren

 David

 Geir Arne

 Kate Leodanis

What Do You Think?

Rate this article:

Tweet Share in 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.

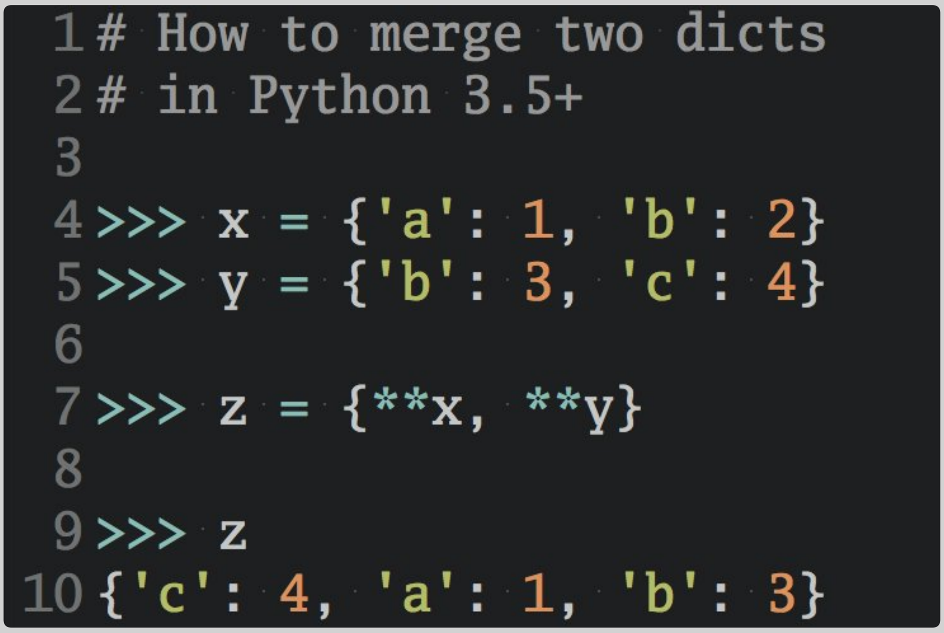
Commenting Tips: The most useful comments are those written with the goal of learning from or helping out other students. Get tips for asking good questions and get answers to common questions in our support portal.

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: basics projects python

Recommended Video Course: Building Python Project Documentation With MkDocs



```
# 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}
```

All Tutorial Topics

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

Table of Contents

Mark as Completed

Tweet Share Email

Recommended Video Course

Building Python Project Documentation With MkDocs