Python Docstrings

What Are Docstrings?

A **docstring** is a special string used to describe what a Python module, function, class, or method does.

Docstrings help other developers understand your code without having to read every line and figure out its purpose.

For example:

```
def calculate_total(price, tax):
    """Calculate the final price including tax."""
    return price + tax
```

The string:

```
"""Calculate the final price including tax."""
```

is the function's **docstring**.

Why Do Docstrings Matter?

Imagine you create a function today and return to the project six months later.

You might see:

```
def process_data(data):
    ...
```

You may wonder:

- What data does this function expect?
- What does it do?
- What does it return?
- Are there any special rules?

A good docstring answers these questions.

Instead of:

```
def process_data(data):
    ...
```

you can write:

```
def process_data(data):
    """Clean user data and return only valid records."""
    ...
```

Now the purpose is immediately clear.

Docstrings vs Comments

Docstrings and comments are both used to make code easier to understand, but they have different purposes.

Comment

A comment explains something about the code.

```
# Add tax to the price
total = price * 1.10
```

Docstring

A docstring documents what a function, class, or module does.

```
def calculate_total(price):
    """Return the price after adding 10% tax."""
    return price * 1.10
```

A simple rule:

> **Comments explain the code. Docstrings document the code's interface and purpose.**

How to Write a Docstring

Docstrings are normally written using triple quotes:

```
"""This is a docstring."""
```

or:

```
'''This is also a docstring.'''
```

Triple double quotes are the common convention:

```
"""This is a docstring."""
```

Function Docstrings

The most common use of docstrings is documenting functions.

```
def greet(name):
    """Return a greeting message for the given name."""
    return f"Hello, {name}!"
```

The docstring should be placed immediately after the function definition:

```
def function_name():
    """Function documentation."""
```

Not here:

```
def function_name():
    print("Hello")

    """This is not the function docstring."""
```

For Python to recognize it as the function's docstring, it must be the first statement inside the function.

Accessing a Docstring

Python stores a function's docstring in its `__doc__` attribute.

```
def greet(name):
    """Return a greeting for a person."""
    return f"Hello, {name}!"

print(greet.__doc__)
```

Output:

`Return a greeting for a person.`

This means the documentation is actually available to Python.

The `help()` Function

Python also provides the built-in `help()` function.

```
def greet(name):
    """Return a greeting for a person."""
    return f"Hello, {name}!"

help(greet)
```

Python displays information about the function, including its docstring.

This is one reason documentation is useful while working in the Python interpreter.

One Line Docstrings

If a function is simple, you can use a one-line docstring.

```
def square(number):
    """Return the square of a number."""
    return number * number
```

Good one-line docstrings are:

- short

- clear
- descriptive
- written as a sentence or phrase

Avoid unnecessary detail.

Bad:

```
def square(number):
    """This function is used to calculate something and it takes a number and then multiplies it by itself."""
```

Better:

```
def square(number):
    """Return the square of a number."""
```

Multi Line Docstrings

For more complicated functions, you can use a multi-line docstring.

```
def calculate_discount(price, percentage):
    """
    Calculate a discounted price.

    The percentage represents the amount
    to subtract from the original price.
    """
    discount = price * percentage / 100
    return price - discount
```

The first line should provide a short summary.

Additional lines can explain details when necessary.

Documenting Parameters

A function may have multiple parameters.

```
def calculate_area(length, width):
    """
    Calculate the area of a rectangle.

    Args:
        length: The length of the rectangle.
        width: The width of the rectangle.

    Returns:
        The area of the rectangle.
    """
    return length * width
```

Here:

`Args:`

describes the function's parameters.

Documenting Return Values

You can explain what the function returns.

```
def calculate_area(length, width):
    """
    Calculate the area of a rectangle.

    Args:
        length: Rectangle length.
        width: Rectangle width.

    Returns:
        The area of the rectangle.
    """
    return length * width
```

The:

```
Returns:
```

section explains the result produced by the function.

Documenting Errors

Some functions can raise exceptions.

You can document them using a `Raises` section.

```
def divide(a, b):
    """
    Divide two numbers.

    Args:
        a: The numerator.
```

```
    b: The denominator.

  Returns:
    The result of the division.

  Raises:
    ZeroDivisionError: If b is zero.
  """
  return a / b
```

This tells another developer that passing `0` as `b` can cause a `ZeroDivisionError`.

A Complete Function Docstring

A more detailed function might look like this:

```
def calculate_average(numbers):
  """
  Calculate the average of a list of numbers.

  Args:
    numbers: A list containing numeric values.

  Returns:
    The average value.

  Raises:
    ValueError: If the list is empty.
  """
  if not numbers:
    raise ValueError("The list cannot be empty")
```

```
    return sum(numbers) / len(numbers)
```

The structure is easy to scan:

```
Summary

Args:
    parameter information

Returns:
    return information

Raises:
    possible errors
```

Docstrings for Classes

Docstrings are not limited to functions.

You can document classes as well.

```
class User:
    """Represent a user in the application."""
```

A more detailed example:

```
class User:
    """
    Represent a registered application user.

    Stores the user's name and email address.
    """
```

```
    def __init__(self, name, email):
        self.name = name
        self.email = email
```

The class docstring explains what the class represents.

Docstrings for Methods

Methods can have their own docstrings.

```
class User:
    """Represent a registered user."""

    def __init__(self, name):
        self.name = name

    def greet(self):
        """Return a greeting for the user."""
        return f"Hello, {self.name}!"
```

Here there are two levels of documentation:

```
User
 □□□ greet()
```

The class has a docstring, and the method has a docstring.

Module Docstrings

A Python file can also have a docstring.

For example:

```
"""
Utility functions for processing user data.
"""

def clean_name(name):
    """Return a cleaned version of a name."""
    return name.strip().title()
```

The module docstring describes the purpose of the entire Python file.

It should normally appear near the top of the file.

Docstring Structure

A useful documentation structure is:

```
def function_name(parameter):
    """
    Short description.

    Args:
        parameter: Description of the parameter.

    Returns:
        Description of the returned value.

    Raises:
```

```
    ErrorType: When the error occurs.
  """
```

Not every function needs every section.

For example, a simple function may only need:

```
def square(number):
  """Return the square of a number."""
```

Do not add unnecessary documentation just to make the docstring longer.

Docstrings and Type Hints

Docstrings work well together with type hints.

```
def add(a: int, b: int) -> int:
  """Return the sum of two integers."""
  return a + b
```

The type hints tell us:

```
a □ int
b □ int
return □ int
```

The docstring tells us:

```
what the function does
```

Together, they make the function easier to understand.

Docstrings for Lists and Data

Suppose you are working with haas.dev resources.

```
def get_free_resources(resources):
    """
    Return resources that are available for free.

    Args:
        resources: A list of resource dictionaries.

    Returns:
        A list containing only free resources.
    """
    return [
        resource
        for resource in resources
        if resource["free"]
    ]
```

A developer can understand the function without studying its implementation first.

Good Docstrings

A good docstring answers the important question:

> **What does this piece of code provide to the developer using it?**

Example:

```
def calculate_total(price, tax):
    """Return the final price after applying tax."""
```

This is useful because it immediately communicates the function's purpose.

Bad Docstrings

Avoid docstrings that simply repeat the code.

```
def add(a, b):
    """Add a and b."""
    return a + b
```

This is not necessarily wrong, but if the function is obvious, the docstring may add little value.

Another weak example:

```
def calculate_total(price, tax):
    """This function calculates total."""
```

Better:

```
def calculate_total(price, tax):
    """Return the final price after applying tax."""
```

Document Meaningful Behavior

Docstrings become especially valuable when behavior is not obvious.

```
def get_active_users(users):
    """
    Return users whose account is currently active.

    Inactive users are excluded from the result.
    """
    return [
        user
        for user in users
        if user["active"]
    ]
```

The docstring explains an important rule of the function.

Docstrings Should Not Explain Every Line

Avoid this:

```
def calculate_total(price, tax):
    """
    Store price in the price variable.
    Store tax in the tax variable.
    Multiply price by tax.
    Add the result to price.
    Return total.
    """
    total = price + (price * tax)
    return total
```

The implementation already explains how the calculation works.

A better docstring focuses on the function's purpose:

```
def calculate_total(price, tax):
    """Return the final price after applying tax."""
    total = price + (price * tax)
    return total
```

Docstrings and IDEs

Modern code editors can display docstrings when you use a function.

For example:

```
def calculate_area(length, width):
    """
    Calculate the area of a rectangle.

    Args:
        length: Rectangle length.
        width: Rectangle width.

    Returns:
        The rectangle's area.
    """
    return length * width
```

When another developer uses:

```
calculate_area(...)
```

their IDE may show the documentation automatically.

This makes well-documented functions easier to use.

Docstrings and `help()`

Built-in Python functions and libraries also use documentation.

For example:

```
help(len)
```

Python can display information about `len()`.

Similarly:

```
help(str.upper)
```

can show documentation for the string method.

This is one reason learning to read documentation is an important programming skill.

Docstrings Are Part of the Code

A docstring is not simply a note written somewhere outside the project.

It is stored inside Python's objects.

Example:

```
def greet():
    """Return a greeting."""
    return "Hello!"
```

You can inspect it:

```
print(greet.__doc__)
```

Output:

```
Return a greeting.
```

This makes documentation accessible programmatically.

Docstrings and Documentation Tools

Docstrings can also be used by documentation tools to generate developer documentation.

A project may contain:

```
def create_user(name, email):
    """
    Create a new user.

    Args:
        name: User's name.
        email: User's email address.

    Returns:
        A dictionary containing the user information.
    """
```

Documentation tools can use information from these docstrings to create readable API or developer documentation.

This becomes especially useful in larger projects and libraries.

Common Docstring Conventions

Different Python projects may use slightly different documentation styles.

Common styles include:

- Google style
- NumPy style
- Sphinx/reStructuredText style

For example, Google style:

```
def add(a, b):
    """
    Add two numbers.

    Args:
        a: First number.
        b: Second number.

    Returns:
        The sum of the two numbers.
    """
    return a + b
```

For a beginner project, consistency is more important than choosing a complicated format.

Common Mistakes

1. Putting the docstring in the wrong place

Incorrect:

```
def greet():
    print("Hello")
    """Return a greeting."""
```

Correct:

```
def greet():
    """Return a greeting."""
    print("Hello")
```

---

2. Writing useless documentation

Avoid:

```
def add(a, b):
    """This function adds."""
```

Prefer:

```
def add(a, b):
    """Return the sum of two numbers."""
```

---

3. Documenting implementation instead of behavior

Avoid explaining every line.

Document what users of the function need to know.

---

4. Letting documentation become outdated

Suppose you originally write:

```
def get_users():
    """Return all users."""
```

Later, the function changes:

```
def get_users(active_only=True):
    ...
```

The docstring should also be updated.

Outdated documentation can be more confusing than no documentation.

---

5. Making Every Docstring Huge

Not every function needs a long explanation.

Simple function:

```
def square(number):
    """Return the square of a number."""
```

Complex function:

```
def process_payment(...):
    """
    Process a customer payment.

    Args:
        ...

    Returns:
        ...

    Raises:
        ...
    """
```

Use the amount of documentation appropriate for the complexity.

Docstrings vs Comments vs Type Hints

Tool	Main Purpose
Comments	Explain implementation details
Docstrings	Document functions, classes, and modules
Type hints	Describe expected data types

Example:

```
def calculate_total(price: float, tax: float) -> float:
    """Return the final price after applying tax."""

    # Calculate the tax amount.
    tax_amount = price * tax

    return price + tax_amount
```

Each has a different job.

A Simple Mental Model

Think of a function as a machine.

```
Input
  ↓
[ FUNCTION ]
  ↓
Output
```

A docstring tells another developer:

```
What does this machine do?
What input does it expect?
What does it produce?
What can go wrong?
```

The implementation tells Python **how** the machine works.

The docstring tells humans **how to use** the machine.

A Practical Documentation Workflow

When writing a function:

Step 1: Write the function

```
def calculate_average(numbers):
    ...
```

Step 2: Ask what the function does

It calculates the average.

Step 3: Document important inputs

numbers list of numbers

Step 4: Document the result

Returns average value

Step 5: Document important errors

Raises ValueError if the list is empty

Final version:

```
def calculate_average(numbers):
    """
    Calculate the average of a list of numbers.
```

```
    Args:
        numbers: A list of numeric values.

    Returns:
        The average value.

    Raises:
        ValueError: If the list is empty.
    """
    if not numbers:
        raise ValueError("The list cannot be empty")

    return sum(numbers) / len(numbers)
```

Real World Example: haas.dev Resources

Suppose haas.dev has a function that searches resources.

```
def search_resources(resources, keyword):
    """
    Find resources whose title contains the given keyword.

    Args:
        resources: A list of resource dictionaries.
        keyword: The word or phrase to search for.

    Returns:
        A list of matching resources.
    """
    keyword = keyword.lower()

    return [
```

```
        resource
        for resource in resources
        if keyword in resource["title"].lower()
    ]
```

A developer reading this function immediately knows:

- what it searches
- what data it expects
- what it returns
- how the search works at a high level

They do not need to inspect every line first.

When Should You Write Docstrings?

Docstrings are especially useful for:

- public functions
- reusable functions
- classes
- modules
- APIs
- libraries
- complex business logic
- code used by multiple developers
- code you expect to maintain for a long time

For tiny private functions, a long docstring may not be necessary.

When You Should Avoid Over Documentation

Do not write documentation that simply repeats obvious code.

For example:

```
def add(a, b):
    """Add a and b."""
    return a + b
```

For a tiny private helper, this may not provide much additional value.

Instead, focus documentation effort on behavior that needs clarification.

Mini Project: Resource Search System

Create a small resource search system.

```
resources = [
    {"title": "Python Basics", "category": "Python"},
    {"title": "JavaScript Functions", "category": "JavaScript"},
    {"title": "Python Lists", "category": "Python"},
]
```

Create a documented function:

```
def search_resources(resources, keyword):
    """
    Find resources containing a keyword in their title.
```

```
    Args:
        resources: A list of resource dictionaries.
        keyword: The keyword to search for.

    Returns:
        A list of matching resources.
    """
    keyword = keyword.lower()

    return [
        resource
        for resource in resources
        if keyword in resource["title"].lower()
    ]
```

Test it:

```
results = search_resources(resources, "python")

print(results)
```

Expected result:

```
[
    {"title": "Python Basics", "category": "Python"},
    {"title": "Python Lists", "category": "Python"}
]
```

The function works, but its docstring makes its intended behavior clear to anyone using it.

---

5 Minute Challenge

Write a function called:

`calculate_discount(price, discount)`

Requirements:

- calculate the discounted price
- add a useful docstring
- document the parameters
- document the return value

Example structure:

```
def calculate_discount(price, discount):
    """
    Calculate the final price after applying a discount.

    Args:
        price: Original price.
        discount: Discount percentage.

    Returns:
        Final price after the discount.
    """
    ...
```

Then test:

```
print(calculate_discount(1000, 20))
```

Expected:

```
800
```

Exercises

Exercise 1

Create a function:

```
def is_even(number):
    ...
```

Add a one-line docstring.

Exercise 2

Create:

```
def find_max(numbers):
    ...
```

Write a docstring containing:

- summary
- argument description
- return description

Exercise 3

Create a class:

```
class Student:
```

`...`

Add a class docstring explaining what the class represents.

Exercise 4

Create a function that raises an exception and document the exception using `Raises`.

Exercise 5

Create a small module containing three functions.

Add:

- a module docstring
- function docstrings
- parameter descriptions where needed
- return descriptions where useful

Mini Quiz

1. What is a docstring?

A. A special string used to document Python code
B. A replacement for variables
C. A type of loop
D. A Python package

Answer: A

---

2. Where should a function's docstring normally appear?

A. At the end of the function
B. Before the function
C. Immediately after the function definition
D. Inside a comment

Answer: C

---

3. Which attribute contains an object's docstring?

A. `__help__`
B. `__doc__`
C. `__info__`
D. `__text__`

Answer: B

---

4. Which built-in function can display documentation?

A. `show()`
B. `docs()`
C. `help()`
D. `info()`

Answer: C

---

5. What should a good docstring primarily explain?

A. Every line of implementation
B. What the code does and how it should be used
C. The programmer's name
D. The computer's hardware

Answer: B

Interview Questions

1. What is a docstring in Python?

A docstring is a string used to document a module, function, class, or method.

---

2. How is a docstring different from a comment?

A comment generally explains implementation details, while a docstring documents the purpose and usage of Python objects such as functions and classes.

---

3. How can you access a function's docstring?

Using:

`function_name.__doc__`

---

4. How can you view documentation interactively?

Using:

```
help(function_name)
```

---

5. Where should a function's docstring be placed?

Immediately after the function definition.

---

6. Can classes have docstrings?

Yes.

```
class User:
    """Represent an application user."""
```

---

7. Can Python modules have docstrings?

Yes. A module docstring describes the purpose of the Python file.

---

8. What are `Args`, `Returns`, and `Raises`?

They are common documentation sections used to describe:

```
Args     — parameters
Returns  — returned result
Raises   — possible exceptions
```

Docstring Cheat Sheet

Simple function

```
def greet(name):
    """Return a greeting for the given name."""
    return f"Hello, {name}!"
```

Detailed function

```
def calculate_area(length, width):
    """
    Calculate the area of a rectangle.

    Args:
        length: Rectangle length.
        width: Rectangle width.

    Returns:
        The rectangle's area.
    """
    return length * width
```

Class

```
class User:
    """Represent an application user."""
```

Module

```
"""
Utility functions for processing users.
"""
```

Access documentation

```
print(greet.__doc__)
```

Use built-in help

```
help(greet)
```

Key Takeaways

- A **docstring** documents Python code.
- Docstrings commonly document modules, functions, classes, and methods.
- They are usually written using triple quotes.
- A function's docstring must be the first statement inside the function.
- You can access a docstring using `__doc__`.
- You can inspect documentation using `help()`.
- `Args` documents parameters.
- `Returns` documents the returned value.
- `Raises` documents possible exceptions.
- Good docstrings explain **what the code does and how to use it**, not every implementation detail.
- Simple functions usually need short docstrings.
- Complex or reusable code benefits from more detailed documentation.
- Keep documentation updated when code behavior changes.
- Docstrings make code easier to understand, maintain, reuse, and document automatically.

Visit haas.dev for more resources and guides.

haas.dev

https://dev-roast-app.vercel.app