PythonDoc Guide

Software Development Best Practices

March 30, 2025

Contents

1 Introduction to PythonDoc

PythonDoc refers to the practice of documenting Python code using docstrings. Unlike formal tools like JavaDoc, PythonDoc is more of a convention than a specific tool. Documentation in Python revolves around:

- **Docstrings**: String literals that appear immediately after the definition of a module, function, class, or method
- **Comments**: Regular Python comments marked with `#`
- **Tools**: Various tools that extract, process, and render docstrings into formatted documentation

Python's built-in `help()` function and the `__doc__` attribute utilize docstrings to provide interactive documentation.

2 Python Docstrings

2.1 Docstring Basics

A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. It becomes the `__doc__` special attribute of that object.

```
def example_function():
    """This is a docstring.

    It provides documentation for the function.
    """
    pass

# Access the docstring
print(example_function.__doc__)
```

Docstrings can be defined using either triple double quotes `"""` or triple single quotes `'''`. Triple double quotes are generally preferred due to consistency with PEP 257.

2.2 PEP 257 - Docstring Conventions

PEP 257 defines the conventions for docstrings in Python:

- **One-line docstrings**: Brief, to the point, ending with a period
- **Multi-line docstrings**: Summary line followed by a blank line, then more elaborate description
- **Module docstrings**: Should list the classes, exceptions, and functions exported by the module
- **Function docstrings**: Should document the function's behavior, parameters, return values, and exceptions
- **Class docstrings**: Should document the class's behavior and public attributes
- **Method docstrings**: Similar to function docstrings

3 Docstring Styles

There are several common styles for formatting docstrings, each with their own conventions and compatible documentation generators.

3.1 Google Style

Google style docstrings use indentation and sections with headings like Args, Returns, Raises, etc.

```
def function_with_google_style(param1, param2):
    """Summary of function purpose.

    Extended description of function.

    Args:
        param1 (int): Description of param1.
        param2 (str): Description of param2. Longer
```

```
            descriptions can be broken up over multiple lines.

    Returns:
        bool: Description of return value.

    Raises:
        ValueError: If param1 is negative.

    Examples:
        >>> function_with_google_style(1, 'test')
        True
    """
    if param1 < 0:
        raise ValueError("param1 must be non-negative")
    return True
```

3.2 reStructuredText (reST) Style

The reStructuredText style is used by Sphinx and is more formal, using directives with colons:

```
def function_with_rest_style(param1, param2):
    """Summary of function purpose.

    Extended description of function.

    :param param1: Description of param1
    :type param1: int
    :param param2: Description of param2
    :type param2: str
    :returns: Description of return value
    :rtype: bool
    :raises ValueError: If param1 is negative

    .. code-block:: python

        >>> function_with_rest_style(1, 'test')
        True
    """
    if param1 < 0:
        raise ValueError("param1 must be non-negative")
    return True
```

3.3 NumPy Style

NumPy style is a variant of reST that uses sections and is popular in scientific computing:

```
def function_with_numpy_style(param1, param2):
    """Summary of function purpose.

    Extended description of function.

    Parameters
    ----------
    param1 : int
        Description of param1
    param2 : str
        Description of param2

    Returns
    -------
    bool
        Description of return value

    Raises
    ------
    ValueError
        If param1 is negative

    Examples
    --------
```

```
    >>> function_with_numpy_style(1, 'test')
    True
    """
    if param1 < 0:
        raise ValueError("param1 must be non-negative")
    return True
```

4 Common Docstring Sections

4.1 Function and Method Docstrings

Comprehensive function docstrings typically include:

- **Summary**: A one-line summary of the function's purpose
- **Extended description**: More detailed explanation of functionality
- **Parameters**: Description of each parameter (type, constraints, defaults)
- **Returns**: Description of return value(s)
- **Raises**: Exceptions that might be raised and why
- **Examples**: Code snippets showing usage
- **Notes**: Additional information (implementation details, algorithm complexity)
- **See Also**: References to related functions or classes

4.2 Class Docstrings

Class docstrings typically include:

```
class ExampleClass:
    """Summary of class purpose.

    Extended description of class.

    Attributes:
        attr1 (int): Description of attr1.
        attr2 (str): Description of attr2.

    Examples:
        >>> obj = ExampleClass(1)
        >>> obj.attr1
        1
    """

    def __init__(self, attr1):
        """Initialize the class.

        Args:
            attr1 (int): Initial value for attr1.
        """
        self.attr1 = attr1
        self.attr2 = "default"
```

4.3 Module Docstrings

Module docstrings appear at the beginning of a module and describe its purpose and contents:

```
"""Module for handling mathematical operations.

This module provides functions for advanced mathematical
operations beyond what's available in the standard math module.

Functions:
    factorial(n): Calculate factorial of n
```

```
    fibonacci(n): Get nth Fibonacci number

Classes:
    MathSequence: Class for generating mathematical sequences

Exceptions:
    NegativeValueError: Raised for inappropriate negative inputs
"""

# Module code follows...
```

5 Type Hints and Docstrings

Python 3.5+ supports type hints, which can complement docstrings:

```
from typing import List, Dict, Optional, Union

def process_data(data: List[Dict[str, Union[str, int]]],
                 verbose: bool = False) -> Optional[Dict[str, int]]:
    """Process the input data and return summary statistics.

    Args:
        data: List of dictionaries containing the data to process.
              Each dictionary should have str keys and str or int values.
        verbose: Whether to print verbose output during processing.

    Returns:
        A dictionary with statistical summaries, or None if data is empty.

    Raises:
        TypeError: If data contains invalid types.
    """
    # Implementation...
```

Type hints can reduce the need for explicit type documentation in docstrings, but you should still document the meaning and constraints of parameters.

6 Documentation Generation Tools

6.1 Built-in Tools

- **pydoc**: Python's built-in documentation generator
- Usage: `python -m pydoc module_name`
- Creates HTML pages: `python -m pydoc -w module_name`

6.2 Sphinx

Sphinx is the most popular tool for generating Python documentation:

```
# Install Sphinx
pip install sphinx sphinx-rtd-theme

# Set up a new Sphinx project
mkdir docs
cd docs
sphinx-quickstart

# Generate documentation
make html
```

6.2.1 Configuring Sphinx

The `conf.py` file in your Sphinx project needs configuration:

```python
# Add path to your Python modules
import os
import sys
sys.path.insert(0, os.path.abspath('..'))

# Extensions
extensions = [
    'sphinx.ext.autodoc',      # Auto-generate from docstrings
    'sphinx.ext.viewcode',     # Link to source code
    'sphinx.ext.napoleon',     # Support Google/NumPy style docstrings
    'sphinx.ext.intersphinx', # Link to other projects
    'sphinx.ext.coverage',     # Check documentation coverage
]

# Theme
html_theme = 'sphinx_rtd_theme'
```

6.2.2 Using autodoc Directives

Create `.rst` files to structure your documentation:

module.rst

```
Module Documentation
====================

.. automodule:: mypackage.mymodule
   :members:
   :undoc-members:
   :show-inheritance:
```

6.3 MkDocs

MkDocs with the mkdocstrings plugin is a lighter-weight alternative to Sphinx:

```bash
# Install MkDocs
pip install mkdocs mkdocstrings mkdocs-material

# Create a new project
mkdocs new my_project
cd my_project

# Build and serve
mkdocs serve
```

Configure in `mkdocs.yml`:

mkdocs.yml

```yaml
site_name: My Project Documentation
theme:
  name: material

plugins:
  - search
  - mkdocstrings:
      handlers:
        python:
          selection:
            docstring_style: google
```

6.4 pdoc

pdoc is a simple, modern documentation generator:

```bash
# Install pdoc
pip install pdoc

# Generate documentation
pdoc --html --output-dir docs mymodule
```

7 Best Practices

7.1 Documentation Best Practices

- **Be consistent**: Choose one style and stick with it throughout your project
- **Document as you code**: Write docstrings when you write the code, not after
- **Keep it updated**: Update documentation when you change code
- **Focus on the why**: Explain why code exists, not just what it does
- **Use examples**: Provide examples for non-trivial functions
- **Document exceptions**: Explain what exceptions can be raised and why
- **Automate testing**: Use `doctest` to ensure examples are correct

7.2 Using doctest

The `doctest` module tests code examples in docstrings:

```python
def add(a, b):
    """Add two numbers and return the result.

    Args:
        a (int): First number
        b (int): Second number

    Returns:
        int: The sum of a and b

    Examples:
        >>> add(1, 2)
        3
        >>> add(-1, 1)
        0
    """
    return a + b

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Run with:

```bash
python -m doctest -v my_module.py
```

7.3 Documentation Coverage

Use tools to check documentation coverage:

```bash
# Install coverage tools
pip install interrogate

# Check documentation coverage
interrogate -v mypackage
```

8 Advanced Topics

8.1 Cross-referencing

Sphinx allows cross-referencing between documentation elements:

```
def process_data(data):
    """Process the given data.

    See Also:
        :func:`validate_data`: Used to validate data before processing
        :class:`DataProcessor`: Class implementation of this functionality
    """
    pass
```

8.2 Autodoc Configuration

Customize autodoc behavior in Sphinx:

```
# In conf.py
autodoc_default_options = {
    'members': True,
    'member-order': 'bysource',
    'special-members': '__init__',
    'undoc-members': True,
    'exclude-members': '__weakref__'
}
```

8.3 Conditional Documentation

Hide certain documentation from generated docs:

```
def internal_function():
    """This function is for internal use only.

    .. rubric:: Notes

    .. only:: developer

       This is a low-level function that directly manipulates
       internal data structures.
    """
    pass
```

9 Real-World Examples

9.1 Simple Function

```
def calculate_discount(price, discount_percent):
    """Calculate the discounted price.

    Args:
        price (float): The original price.
        discount_percent (float): The discount percentage (0-100).

    Returns:
        float: The price after applying the discount.

    Raises:
        ValueError: If discount_percent is not between 0 and 100.

    Examples:
        >>> calculate_discount(100, 20)
        80.0
        >>> calculate_discount(50, 10)
        45.0
    """
    if not 0 <= discount_percent <= 100:
```

```
        raise ValueError("discount_percent must be between 0 and 100")

    discount_factor = 1 - (discount_percent / 100)
    return price * discount_factor
```

9.2 Class Example

```
class ShoppingCart:
    """A shopping cart for an online store.

    The ShoppingCart class handles items added to a virtual shopping cart,
    managing quantities, calculating totals, and applying discounts.

    Attributes:
        items (dict): A dictionary mapping item IDs to quantities.
        prices (dict): A dictionary mapping item IDs to prices.

    Examples:
        >>> cart = ShoppingCart()
        >>> cart.add_item("apple", 1.2, 3)
        >>> cart.add_item("banana", 0.5, 4)
        >>> cart.calculate_total()
        5.6
    """

    def __init__(self):
        """Initialize an empty shopping cart."""
        self.items = {}
        self.prices = {}

    def add_item(self, item_id, price, quantity=1):
        """Add an item to the cart.

        Args:
            item_id (str): The ID of the item.
            price (float): The price of the item.
            quantity (int, optional): The quantity to add. Defaults to 1.

        Raises:
            ValueError: If quantity is less than 1.
        """
        if quantity < 1:
            raise ValueError("Quantity must be at least 1")

        if item_id in self.items:
            self.items[item_id] += quantity
        else:
            self.items[item_id] = quantity
            self.prices[item_id] = price

    def calculate_total(self):
        """Calculate the total price of all items in the cart.

        Returns:
            float: The total price.
        """
        total = 0
        for item_id, quantity in self.items.items():
            total += self.prices[item_id] * quantity
        return total
```

9.3 Module Example

```
"""
Geometry Module
==============

This module provides functions and classes for working with
geometric shapes and calculations.
```

```
Functions:
    calculate_area(shape, **kwargs): Calculate area of different shapes
    calculate_perimeter(shape, **kwargs): Calculate perimeter of shapes

Classes:
    Circle: Represents a circle with radius
    Rectangle: Represents a rectangle with width and height
    Triangle: Represents a triangle with three sides

Exceptions:
    InvalidShapeError: Raised when an invalid shape is specified
    NegativeDimensionError: Raised when a negative dimension is provided
"""

import math

class InvalidShapeError(Exception):
    """Raised when an invalid shape is specified."""
    pass

class NegativeDimensionError(Exception):
    """Raised when a negative dimension is provided."""
    pass

def calculate_area(shape, **kwargs):
    """Calculate the area of a geometric shape.

    Args:
        shape (str): The type of shape ('circle', 'rectangle', 'triangle').
        **kwargs: Dimensions of the shape.
            For circle: radius
            For rectangle: width, height
            For triangle: base, height

    Returns:
        float: The calculated area.

    Raises:
        InvalidShapeError: If shape is not supported.
        NegativeDimensionError: If any dimension is negative.
        KeyError: If required dimensions are not provided.

    Examples:
        >>> calculate_area('circle', radius=5)
        78.53981633974483
        >>> calculate_area('rectangle', width=4, height=5)
        20
    """
    # Implementation details...
    pass
```

10 Conclusion

Effective documentation is an essential part of professional Python development. By following established conventions and using the right tools, you can create documentation that is helpful, maintainable, and integrated with your codebase. Key takeaways include:

- Choose a consistent docstring style (Google, reST, or NumPy)
- Document all public modules, functions, classes, and methods
- Include examples and expected behavior
- Use appropriate tools to generate formatted documentation
- Keep documentation up-to-date as code changes
- Consider documentation as part of your development process, not an afterthought

Clear, comprehensive documentation not only helps others use your code but also serves as a design tool and helps you think through your code's interface and behavior.