【新人エンジニア必見】Pythonドキュメンテーション5大スタイル完全ガイド

はじめに

Pythonでコードを書く時、「この関数、何するんだっけ？」と後で困ったことはありませんか？

そんな時に役立つのが**ドキュメンテーション（docstring）**です。この記事では、Pythonでよく使われる5つのドキュメンテーションスタイルを、実際に動くコード付きで解説します。

なぜドキュメンテーションが必要？

```python
def calc(p, r=0.1):
  return p * r
```

このコード、何をする関数か分かりますか？

```python
def calculate_tax(price: float, tax_rate: float = 0.1) -> float:
  """消費税を計算する

  Args:
    price: 商品の価格
    tax_rate: 消費税率（デフォルト: 0.1）

  Returns:
    消費税額
  """
  return price * tax_rate
```

こちらなら一目瞭然ですね。

スタイル1: Google Style（最もおすすめ）

特徴

- シンプルで読みやすい
- 最も人気があるスタイル
- Google社が推奨

基本フォーマット

```python
def function_name(param1: type, param2: type = default) -> return_type:
    """関数の簡潔な説明（1行）

    必要に応じて詳細な説明をここに書く。
    複数行で書いてもOK。

    Args:
        param1 (type): パラメータ1の説明
        param2 (type, optional): パラメータ2の説明. Defaults to default.

    Returns:
        return_type: 戻り値の説明

    Raises:
        ErrorType: どんな時にエラーが起きるか

    Examples:
        >>> function_name(value1, value2)
        expected_result
    """
    pass
```

実例：ふわふわ大福店

```python
```

```
class DaifukuShop:
    """大福屋さんクラス

    Attributes:
        name (str): お店の名前
        stock (int): 在庫数
        sales (float): 売上金額
    """

    def __init__(self, name: str) -> None:
        """初期化

        Args:
            name (str): お店の名前
        """
        self.name = name
        self.stock = 0
        self.sales = 0.0

    def sell(self, count: int, price: float) -> bool:
        """大福を販売

        Args:
            count (int): 販売個数
            price (float): 単価

        Returns:
            bool: 販売成功ならTrue、在庫不足ならFalse

        Examples:
            >>> shop = DaifukuShop("ふわふわ")
            >>> shop.stock = 10
            >>> shop.sell(3, 300)
            True
        """
        if self.stock >= count:
            self.stock -= count
            self.sales += count * price
            return True
        return False
```

メリット:

- 読みやすい
- 書きやすい

- IDEのサポートが充実

デメリット:

- 特になし（初心者に最適）

スタイル2: NumPy Style（科学計算向け）

特徴

- 詳細な説明に向いている
- NumPy、SciPy、pandas などで使用
- セクションが下線で区切られる

基本フォーマット

```python
def function_name(param1, param2=default):
    """関数の簡潔な説明（1行）

    詳細な説明をここに書く。

    Parameters
    ----------
    param1 : type
        パラメータ1の説明
    param2 : type, optional
        パラメータ2の説明, by default default

    Returns
    -------
    return_type
        戻り値の説明

    Examples
    --------
    >>> function_name(value1, value2)
    expected_result
    """
    pass
```

実例

```python
```

```
def calculate_total(price: float, tax_rate: float = 0.1) -> dict:
    """合計金額を計算

    Parameters
    ----------
    price : float
        商品の価格
    tax_rate : float, optional
        消費税率, by default 0.1

    Returns
    -------
    dict
        価格、税額、合計を含む辞書
        - 'price': 元の価格
        - 'tax': 消費税額
        - 'total': 合計金額

    Examples
    --------
    >>> result = calculate_total(1000)
    >>> result['total']
    1100.0
    """
    tax = price * tax_rate
    return {
        'price': price,
        'tax': tax,
        'total': price + tax
    }
```

メリット:

- 複雑な戻り値の説明がしやすい
- 科学計算コミュニティで標準

デメリット:

- やや冗長
- 書くのに時間がかかる

スタイル3: reStructuredText（Sphinx用）

特徴

- Python公式ドキュメントで使用
- Sphinxで美しいHTMLドキュメント生成
- `:param:` などのフィールドリストを使用

基本フォーマット

```python
def function_name(param1, param2=default):
    """関数の簡潔な説明

    詳細な説明

    :param param1: パラメータ1の説明
    :type param1: type
    :param param2: パラメータ2の説明
    :type param2: type
    :return: 戻り値の説明
    :rtype: return_type
    :raises ErrorType: エラーの説明

    .. note::
       補足情報をここに

    .. code-block:: python

       >>> function_name(value1)
       result
    """
    pass
```

実例

```python
```

```
def calculate_discount(price: float, rate: float) -> float:
    """割引後の価格を計算

    :param price: 商品の価格
    :type price: float
    :param rate: 割引率（0.0〜1.0）
    :type rate: float
    :return: 割引後の価格
    :rtype: float
    :raises ValueError: 割引率が範囲外の場合

    .. note::
       割引率は0.0（割引なし）から1.0（100%割引）

    .. code-block:: python

       >>> calculate_discount(1000, 0.2)
       800.0
    """
    if not 0 <= rate <= 1:
        raise ValueError("割引率は0〜1で指定")
    return price * (1 - rate)
```

メリット:

- Sphinxで自動的にドキュメント生成
- Python公式スタイル

デメリット:

- 記法が複雑
- 書きづらい

スタイル4: Epytext（Epydoc用）

特徴

- `@param`、`@return` などを使用
- Epydocツール用
- 古典的なスタイル

基本フォーマット

```
python
```

```
def function_name(param1, param2=default):
    """関数の簡潔な説明

    詳細な説明

    @param param1: パラメータ1の説明
    @type param1: type
    @param param2: パラメータ2の説明
    @type param2: type
    @return: 戻り値の説明
    @rtype: return_type

    @example:
        >>> function_name(value1)
        result
    """
    pass
```

実例

```
python

def format_price(price: float, currency: str = "円") -> str:
    """価格を整形

    @param price: 価格
    @type price: float
    @param currency: 通貨単位
    @type currency: str
    @return: 整形された価格文字列
    @rtype: str

    @example:
        >>> format_price(1000)
        '1,000円'
    """
    formatted = f"{int(price):,}"
    return f"{formatted}{currency}"
```

メリット:

- シンプル
- Javaライクで分かりやすい

デメリット:

- 現在はあまり使われない
- Epydocの開発が停滞

スタイル5: 型ヒント（Type Hints）

特徴

- Python 3.5以降の機能
- IDEの補完・エラーチェックに最適
- ドキュメントというより、コードの一部

基本フォーマット

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

def function_name(
    param1: int,
    param2: str,
    param3: Optional[float] = None
) -> Dict[str, any]:
    """関数の説明（型情報は不要）"""
    pass
```

実例

```python
```

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

class Shop:
    """型ヒント重視のお店クラス"""

    name: str
    stock: int
    products: List[Dict[str, any]]

    def __init__(self, name: str) -> None:
        self.name = name
        self.stock = 0
        self.products = []

    def add_product(
        self,
        name: str,
        price: float,
        count: int
    ) -> None:
        """商品を追加"""
        product: Dict[str, any] = {
            'name': name,
            'price': price,
            'count': count
        }
        self.products.append(product)
        self.stock += count

    def get_total_value(self) -> float:
        """在庫の総額を計算"""
        total: float = 0.0
        for product in self.products:
            total += product['price'] * product['count']
        return total
```

メリット:

- IDEが自動補完してくれる
- 型エラーを事前に検出
- mypyなどのツールでチェック可能

デメリット:

- 説明文は別途必要

- Python 3.5以降のみ

どれを使えばいい？新人エンジニアへのおすすめ

おすすめの組み合わせ

Google Style + 型ヒント

```python
def calculate_tax(price: float, tax_rate: float = 0.1) -> float:
    """消費税を計算する

    Args:
        price: 商品の価格
        tax_rate: 消費税率（デフォルト: 0.1）

    Returns:
        消費税額

    Examples:
        >>> calculate_tax(1000)
        100.0
    """
    return price * tax_rate
```

選び方のポイント

用途	おすすめスタイル
一般的なプロジェクト	Google Style + 型ヒント
科学計算・データ分析	NumPy Style + 型ヒント
公式ドキュメント生成	reStructuredText
レガシープロジェクト	Epytext（既存コードに合わせる）

実践：うさうさ店長のふわふわ大福店

5つのスタイルを全て使った完全版コードは、こちらのGitHubからダウンロードできます。

VSCodeで実行して、各スタイルの違いを体験してみてください。

まとめ

- **Google Style**: シンプルで読みやすい（初心者におすすめ）
- **NumPy Style**: 詳細な説明が必要な時
- **reStructuredText**: Sphinxでドキュメント生成する時
- **Epytext**: 古いプロジェクトで
- **型ヒント**: 必ず使おう

新人エンジニアは**Google Style + 型ヒント**から始めるのがベストです。

コードは書くより読まれる時間の方が長い。未来の自分や仲間のために、分かりやすいドキュメントを書きましょう。

参考リンク

- PEP 257 - Docstring Conventions
- Google Python Style Guide
- NumPy Documentation Style
- PEP 484 - Type Hints

この記事が役に立ったら、いいねとフォローをお願いします。
質問やコメントもお待ちしています。