🐰 スタティックメソッド完全ガイド

ふわふわ大福店のうさうさ店長で学ぶ、文法と使う理由7選

📚 目次

---

1. スタティックメソッドとは

📖 一言で言うと

スタティックメソッド = クラスに属する「ただの関数」

- インスタンス（`self`）を使わない
- クラス（`cls`）も使わない
- でもクラスに関連している
- 名前空間の整理に便利

🎯 たい焼き屋での例え

```
たい焼き屋の厨房にある「共用ツール」

🏪 たい焼き屋
├── 各店員（インスタンス）が持つ道具
│   ├── 自分のエプロン（インスタンス変数）
│   └── 自分の作業（インスタンスメソッド）
│
├── 全店員共通の道具（クラス）
│   ├── 共通レシピ（クラス変数）
│   └── 全体ミーティング（クラスメソッド）
│
└── 共用ツール（スタティックメソッド）
    ├── 温度計 ← 誰でも使える
```

```
├── タイマー ← 特定の人に紐づかない
└── 計算機 ← ただのツール
```

---

2. 文法と基本的な書き方

💻 基本構文

python

```python
class クラス名:

    @staticmethod  # ← このデコレーターが必須！
    def メソッド名(引数1, 引数2, ...):  # self も cls も不要
        """処理内容"""
        # インスタンスやクラスのデータは使わない
        # ただの計算や処理を行う
        return 結果
```

🔍 3つのメソッドの文法比較

python

```python
class DaifukuShop:
    """大福店クラス"""

    total_shops = 0  # クラス変数

    def __init__(self, owner_name):
        self.owner_name = owner_name  # インスタンス変数

    # ========================================
    # 1️⃣ インスタンスメソッド（通常）
    # ========================================
    def sell(self, quantity):
        """
        第一引数: self（必須）

        使えるもの:
        - self.xxx（インスタンス変数）
        - self.メソッド()（インスタンスメソッド）
        - ClassName.xxx（クラス変数）
        """
        print(f"{self.owner_name}店長が{quantity}個販売")
        return quantity * 150

    # ========================================
    # 2️⃣ クラスメソッド
    # ========================================
    @classmethod
    def get_total(cls):
        """
        第一引数: cls（必須）
        デコレーター: @classmethod

        使えるもの:
        - cls.xxx（クラス変数）
        - cls.メソッド()（クラスメソッド）
        """
        return f"総店舗数: {cls.total_shops}店"

    # ========================================
    # 3️⃣ スタティックメソッド
    # ========================================
    @staticmethod
    def calculate_tax(price):
        """
        第一引数: なし（自由に決められる）
        デコレーター: @staticmethod
```

```
        使えるもの:
        - 引数として渡されたデータのみ
        - インスタンス変数もクラス変数も使わない
        """
        return int(price * 1.1)


# ============================================
# 呼び出し方の違い
# ============================================

# インスタンスメソッド: インスタンスから呼ぶ
shop = DaifukuShop("うさうさ")
shop.sell(5)  # self は自動で渡される

# クラスメソッド: クラスから呼べる（インスタンス不要）
print(DaifukuShop.get_total())  # cls は自動で渡される

# スタティックメソッド: クラスから呼べる（インスタンス不要）
tax_price = DaifukuShop.calculate_tax(1000)  # 引数だけ渡す
print(f"税込: ¥{tax_price}")

# スタティックメソッドはインスタンスからも呼べる
# （でもクラスから呼ぶのが一般的）
tax_price2 = shop.calculate_tax(1000)
```

📊 文法の完全比較表

項目	インスタンスメソッド	クラスメソッド	スタティックメソッド
デコレーター	なし	`@classmethod`	`@staticmethod`
第一引数	`self`（必須）	`cls`（必須）	なし（自由）
呼び出し	`obj.method()`	`Class.method()`	`Class.method()`
インスタンス変数	✅ 使える	❌ 使えない	❌ 使えない
クラス変数	✅ 使える	✅ 使える	❌ 使えない
インスタンス必要？	✅ 必要	❌ 不要	❌ 不要

---

3. 使う理由7選

理由 1️⃣ : 名前空間の整理（最重要！）

問題: 関連する関数をどこに置くべきか？

python

```python
# ❌ 悪い例: クラスの外に関数を書く
def validate_price(price):
    """価格の検証"""
    return price > 0

def calculate_tax(price):
    """税込価格の計算"""
    return price * 1.1

def format_currency(amount):
    """通貨フォーマット"""
    return f"¥{amount:,}"

class DaifukuShop:
    """大福店クラス"""
    pass

# 問題点:
# - 関数が散らばっている
# - どの関数がDaifukuShopに関連しているか不明
# - 名前の衝突が起きやすい
```

python

```python
# ✅ 良い例: スタティックメソッドで整理
class DaifukuShop:
    """大福店クラス"""

    @staticmethod
    def validate_price(price):
        """価格の検証"""
        return price > 0

    @staticmethod
    def calculate_tax(price):
        """税込価格の計算"""
        return price * 1.1

    @staticmethod
    def format_currency(amount):
        """通貨フォーマット"""
        return f"¥{amount:,}"

# メリット:
# - 関連する関数がクラスにまとまっている
# - DaifukuShop.validate_price() で呼べる
# - 名前空間が整理される
```

理由2️⃣: バリデーション（検証）関数

問題: 入力値のチェックをどこで行うか？

```python
```

```python
class DaifukuShop:
    """大福店クラス"""

    def __init__(self, owner_name, stock):
        # ここでバリデーション
        if not DaifukuShop.validate_name(owner_name):
            raise ValueError("店長名が不正です")

        if not DaifukuShop.validate_stock(stock):
            raise ValueError("在庫数が不正です")

        self.owner_name = owner_name
        self.stock = stock

    @staticmethod
    def validate_name(name):
        """
        店長名の検証

        理由:
        - インスタンスを作る前にチェックしたい
        - self も cls も不要
        - 純粋な検証ロジック
        """
        return isinstance(name, str) and len(name) > 0

    @staticmethod
    def validate_stock(stock):
        """
        在庫数の検証

        理由:
        - 独立した検証ロジック
        - テストしやすい
        """
        return isinstance(stock, int) and stock >= 0

    @staticmethod
    def validate_price(price):
        """価格の検証"""
        return isinstance(price, (int, float)) and price > 0


# 使用例
print(DaifukuShop.validate_name("うさうさ"))  # True
print(DaifukuShop.validate_stock(20))      # True
```

```
print(DaifukuShop.validate_stock(-5))      # False

# メリット:
# 1. インスタンスを作る前に検証できる
# 2. 単体テストが書きやすい
# 3. 他のクラスからも使える
```

理由 3️⃣: ユーティリティ関数（補助関数）

問題: クラスに関連する便利な計算をどこに書くか？

```
python
```

```python
class DaifukuShop:
    """大福店クラス"""

    @staticmethod
    def calculate_tax(price, tax_rate=0.1):
        """
        税込価格を計算

        理由:
        - 単純な計算ロジック
        - インスタンスのデータを使わない
        - いろんな場所から呼びたい
        """
        return int(price * (1 + tax_rate))

    @staticmethod
    def calculate_discount(price, discount_rate):
        """
        割引価格を計算

        理由:
        - 純粋な計算
        - どのインスタンスにも属さない
        """
        return int(price * (1 - discount_rate))

    @staticmethod
    def format_yen(amount):
        """
        円記号付きでフォーマット

        理由:
        - 単純な文字列整形
        - 補助的な機能
        """
        return f"¥{amount:,}"

    @staticmethod
    def parse_date(date_string):
        """
        日付文字列をパース

        理由:
        - 変換処理
        - インスタンスに依存しない
        """
```

```
        from datetime import datetime
        return datetime.strptime(date_string, "%Y-%m-%d")


# 使用例
price = 1000
tax_price = DaifukuShop.calculate_tax(price)
discount_price = DaifukuShop.calculate_discount(price, 0.1)

print(DaifukuShop.format_yen(tax_price))     # ¥1,100
print(DaifukuShop.format_yen(discount_price))  # ¥900

# メリット:
# 1. 計算ロジックがクラスにまとまる
# 2. 再利用しやすい
# 3. テストしやすい
```

理由4️⃣: ファクトリーパターンの補助

問題: いろいろな方法でインスタンスを作りたい

python

```python
class DaifukuShop:
    """大福店クラス"""

    def __init__(self, owner_name, stock):
        # 入力検証（スタティックメソッドを使用）
        if not DaifukuShop._validate_inputs(owner_name, stock):
            raise ValueError("入力値が不正です")

        self.owner_name = owner_name
        self.stock = stock

    @classmethod
    def from_dict(cls, data):
        """
        辞書からインスタンスを作成（クラスメソッド）

        内部でスタティックメソッドを使用
        """
        # スタティックメソッドで前処理
        cleaned_data = cls._clean_dict_data(data)

        return cls(
            owner_name=cleaned_data["owner"],
            stock=cleaned_data["stock"]
        )

    @staticmethod
    def _validate_inputs(owner_name, stock):
        """
        入力値の検証（スタティックメソッド）

        理由:
        - クラスメソッドから呼ばれる
        - 純粋な検証ロジック
        - cls も self も不要
        """
        if not isinstance(owner_name, str) or len(owner_name) == 0:
            return False
        if not isinstance(stock, int) or stock < 0:
            return False
        return True

    @staticmethod
    def _clean_dict_data(data):
        """
        辞書データのクリーニング（スタティックメソッド）
```

```
    理由:
    - データ変換処理
    - クラスメソッドの補助
    - 独立した処理
    """
    return {
      "owner": data.get("owner", "").strip(),
      "stock": int(data.get("stock", 0))
    }


# 使用例
data = {"owner": " うさうさ ", "stock": "20"}
shop = DaifukuShop.from_dict(data)
print(f"{shop.owner_name}店長、在庫{shop.stock}個")

# メリット:
# 1. クラスメソッドとスタティックメソッドの役割分担
# 2. 処理を小さな関数に分解できる
# 3. テストしやすい
```

理由 5 : 定数関連の計算

問題: 定数から派生する値を計算したい

```
python
```

```python
class DaifukuShop:
    """大福店クラス"""

    # 定数（クラス変数）
    BASE_PRICE = 150        # 基本価格
    PREMIUM_MULTIPLIER = 1.5  # プレミアム倍率
    DISCOUNT_RATE = 0.1     # 割引率

    @staticmethod
    def get_premium_price():
        """
        プレミアム価格を計算

        理由:
        - 定数から計算
        - インスタンスに依存しない
        - でもクラスに関連している
        """
        return int(DaifukuShop.BASE_PRICE * DaifukuShop.PREMIUM_MULTIPLIER)

    @staticmethod
    def get_discount_price():
        """
        割引価格を計算

        理由:
        - 定数から計算
        - 設定値の組み合わせ
        """
        return int(DaifukuShop.BASE_PRICE * (1 - DaifukuShop.DISCOUNT_RATE))

    @staticmethod
    def get_price_range():
        """
        価格帯を返す

        理由:
        - 複数の定数を組み合わせる
        - 計算結果のみを返す
        """
        min_price = DaifukuShop.get_discount_price()
        max_price = DaifukuShop.get_premium_price()
        return (min_price, max_price)


# 使用例
```

```python
print(f"基本価格: ¥{DaifukuShop.BASE_PRICE}")
print(f"プレミアム価格: ¥{DaifukuShop.get_premium_price()}")
print(f"割引価格: ¥{DaifukuShop.get_discount_price()}")
print(f"価格帯: ¥{DaifukuShop.get_price_range()[0]} - ¥{DaifukuShop.get_price_range()[1]}")

# メリット:
# 1. 定数とその計算をまとめられる
# 2. マジックナンバーを避けられる
# 3. 変更が容易
```

理由 6️⃣: 型変換・データ変換

問題: いろいろな形式のデータを扱いたい

python

```python
class DaifukuShop:
    """大福店クラス"""

    @staticmethod
    def parse_stock_string(stock_str):
        """
        文字列を在庫数に変換

        理由:
        - 純粋な変換処理
        - エラーハンドリング
        - インスタンス不要

        例:
        "20個" → 20
        "15" → 15
        "在庫: 30" → 30
        """
        import re
        # 数字だけを抽出
        numbers = re.findall(r'\d+', stock_str)
        if numbers:
            return int(numbers[0])
        return 0

    @staticmethod
    def format_shop_data(owner_name, stock):
        """
        店舗データをフォーマット

        理由:
        - データ整形
        - 表示用の変換
        """
        return {
            "店長": owner_name,
            "在庫": f"{stock}個",
            "状態": "営業中" if stock > 0 else "売り切れ"
        }

    @staticmethod
    def csv_to_dict(csv_line):
        """
        CSV行を辞書に変換

        理由:
```

```
    - データインポート
    - 形式変換
    """
    parts = csv_line.strip().split(',')
    if len(parts) >= 2:
        return {
            "owner": parts[0],
            "stock": int(parts[1])
        }
    return None


# 使用例
stock1 = DaifukuShop.parse_stock_string("20個")
stock2 = DaifukuShop.parse_stock_string("在庫: 15")
print(f"在庫1: {stock1}個")  # 20個
print(f"在庫2: {stock2}個")  # 15個

data = DaifukuShop.format_shop_data("うさうさ", 25)
print(data)  # {'店長': 'うさうさ', '在庫': '25個', '状態': '営業中'}

csv = "もちもち,30"
shop_data = DaifukuShop.csv_to_dict(csv)
print(shop_data)  # {'owner': 'もちもち', 'stock': 30}

# メリット:
# 1. データ変換をクラスにまとめられる
# 2. 入力形式が柔軟
# 3. エラー処理を一箇所に
```

理由 7️⃣ : テストの容易さ

問題: ユニットテストを書きやすくしたい

python

```python
class DaifukuShop:
    """大福店クラス"""

    @staticmethod
    def calculate_profit(revenue, cost):
        """
        利益を計算

        理由:
        - テストしやすい（純粋関数）
        - 副作用がない
        - 入力と出力が明確
        """
        return revenue - cost

    @staticmethod
    def is_profitable(revenue, cost):
        """
        黒字かどうか判定

        理由:
        - 純粋なロジック
        - モック不要でテストできる
        """
        return DaifukuShop.calculate_profit(revenue, cost) > 0

    @staticmethod
    def calculate_roi(revenue, cost):
        """
        ROI（投資収益率）を計算

        理由:
        - 計算式が複雑
        - 単体でテストしたい
        """
        if cost == 0:
            return 0
        return (revenue - cost) / cost * 100


# ========================================
# ユニットテストの例（pytest）
# ========================================
def test_calculate_profit():
    """利益計算のテスト"""
    # スタティックメソッドはインスタンス不要でテストできる
```

```
    assert DaifukuShop.calculate_profit(1000, 600) == 400
    assert DaifukuShop.calculate_profit(500, 500) == 0
    assert DaifukuShop.calculate_profit(300, 400) == -100

def test_is_profitable():
    """黒字判定のテスト"""
    assert DaifukuShop.is_profitable(1000, 600) == True
    assert DaifukuShop.is_profitable(500, 500) == False
    assert DaifukuShop.is_profitable(300, 400) == False

def test_calculate_roi():
    """ROI計算のテスト"""
    assert DaifukuShop.calculate_roi(1000, 500) == 100.0
    assert DaifukuShop.calculate_roi(600, 400) == 50.0
    assert DaifukuShop.calculate_roi(100, 0) == 0  # ゼロ除算対策


# メリット:
# 1. インスタンスを作らずにテストできる
# 2. モックが不要
# 3. テストが高速
# 4. テストコードがシンプル
```

---

4. 3つのメソッドの完全比較

📊 使い分け早見表

判断基準	インスタンスメソッド	クラスメソッド	スタティックメソッド
インスタンス変数を使う？	✅ YES	❌ NO	❌ NO
クラス変数を使う？	✅ YES	✅ YES	❌ NO
インスタンスを作成する？	❌ NO	✅ YES	❌ NO
個別の処理？	✅ YES	❌ NO	❌ NO
クラス全体の処理？	❌ NO	✅ YES	❌ NO
補助的な計算？	❌ NO	❌ NO	✅ YES

💻 実践的な比較コード

python

```python
class DaifukuShop:
    """3つのメソッドの使い分けデモ"""

    # クラス変数
    total_shops = 0
    base_price = 150

    def __init__(self, owner_name, stock):
        self.owner_name = owner_name
        self.stock = stock
        DaifukuShop.total_shops += 1

    # ========================================
    # インスタンスメソッド: 個別の店舗の処理
    # ========================================
    def sell(self, quantity):
        """
        使うタイミング:
        - この店舗の在庫を減らす
        - この店舗の店長名を表示
        - 個別のデータを操作
        """
        if quantity > self.stock:
            return False

        self.stock -= quantity
        price = quantity * DaifukuShop.base_price
        print(f"{self.owner_name}店長: {quantity}個販売（¥{price}）")
        return True

    # ========================================
    # クラスメソッド: クラス全体の処理
    # ========================================
    @classmethod
    def get_stats(cls):
        """
        使うタイミング:
        - 全店舗の統計を取得
        - クラス変数を参照
        - インスタンスを作成
        """
        return f"総店舗数: {cls.total_shops}店"

    @classmethod
    def create_franchise(cls, owner_name):
        """
```

```
        使うタイミング:
        - 標準的な店舗を作成
        - ファクトリーメソッド
        """
        return cls(owner_name, stock=20)  # cls()でインスタンス作成

    # ========================================
    # スタティックメソッド: 補助的な計算
    # ========================================
    @staticmethod
    def calculate_tax(price):
        """
        使うタイミング:
        - インスタンスもクラスも使わない
        - 純粋な計算
        - ユーティリティ関数
        """
        return int(price * 1.1)

    @staticmethod
    def validate_stock(stock):
        """
        使うタイミング:
        - 検証ロジック
        - インスタンス作成前にチェック
        """
        return isinstance(stock, int) and stock >= 0


# ========================================
# 使い分けの実例
# ========================================

# インスタンスメソッド: 個別の処理
shop = DaifukuShop("うさうさ", 20)
shop.sell(5)  # この店舗で販売

# クラスメソッド: 全体の処理やインスタンス作成
print(DaifukuShop.get_stats())  # 全店舗の統計
shop2 = DaifukuShop.create_franchise("もちもち")  # 標準店舗を作成

# スタティックメソッド: 補助的な計算
price = 1000
tax_price = DaifukuShop.calculate_tax(price)
print(f"税込: ¥{tax_price}")
```

```
is_valid = DaifukuShop.validate_stock(20)
print(f"在庫数は有効？ {is_valid}")
```

---

5. 実践例：ふわふわ大福店

💻 実戦的な完全実装

python

```
"""
==========================================
🐰 ふわふわ大福店
スタティックメソッドの実践例
==========================================
"""

class DaifukuShop:
    """
    大福店クラス
    スタティックメソッドを効果的に使用
    """

    # クラス変数
    total_shops = 0
    company_name = "ふわふわ大福株式会社"

    # 定数
    BASE_PRICE = 150
    PREMIUM_RATE = 1.5
    DISCOUNT_RATE = 0.1
    TAX_RATE = 0.10

    def __init__(self, owner_name, location, stock):
        # 入力検証（スタティックメソッド使用）
        if not self.validate_owner_name(owner_name):
            raise ValueError("店長名が不正です")

        if not self.validate_stock(stock):
            raise ValueError("在庫数が不正です")

        self.owner_name = owner_name
        self.location = location
        self.stock = stock
        self.sold = 0
        self.revenue = 0

        DaifukuShop.total_shops += 1

    # ==========================================
    # インスタンスメソッド: 個別の店舗の処理
    # ==========================================

    def sell(self, quantity, is_premium=False, discount=0):
        """販売処理"""
        if not self.validate_quantity(quantity):
```

```
    print("❌ 数量が不正です")
    return None

if quantity > self.stock:
    print(f"❌ 在庫不足（在庫: {self.stock}個）")
    return None

# 価格計算（スタティックメソッド使用）
base = self.BASE_PRICE
if is_premium:
    base = self.calculate_premium_price(base)

if discount > 0:
    base = self.calculate_discounted_price(base, discount)

subtotal = quantity * base
total = self.calculate_with_tax(subtotal)

# 在庫更新
self
```