🐰 Pythonメソッドと継承の注意点 完全ガイド

ふわふわ大福店のうさうさ店長で学ぶ、実践的でわかりやすい解説

📚 目次

---

1. メソッドの種類

📖 3つのメソッドタイプ

Pythonには3種類のメソッドがあります：

種類	第一引数	呼び出し方	用途
インスタンスメソッド	`self`	`obj.method()`	個別の処理
クラスメソッド	`cls`	`Class.method()`	クラス全体の処理
スタティックメソッド	なし	`Class.method()`	補助関数

💻 実際のコード

```python
```

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

    # クラス変数
    total_shops = 0
    tax_rate = 0.10

    def __init__(self, owner_name, stock):
        """コンストラクタ"""
        self.owner_name = owner_name
        self.stock = stock
        DaifukuShop.total_shops += 1

    # ========================================
    # 1. インスタンスメソッド（最も一般的）
    # ========================================
    def sell(self, quantity):
        """
        個別の店舗で販売処理

        特徴:
        - 第一引数は self（自分自身）
        - インスタンス変数にアクセスできる
        - クラス変数にもアクセスできる
        """
        if quantity > self.stock:
            print(f"❌ 在庫不足！")
            return False

        self.stock -= quantity  # インスタンス変数を変更
        total = int(quantity * 150 * (1 + DaifukuShop.tax_rate))
        print(f"💰 {self.owner_name}店長: {quantity}個販売（¥{total}）")
        return True

    # ========================================
    # 2. クラスメソッド
    # ========================================
    @classmethod
    def get_total_shops(cls):
        """
        クラス全体の情報を取得

        特徴:
        - @classmethod デコレーター必須
        - 第一引数は cls（クラス自身）
        - クラス変数にアクセスできる
```

```
    - インスタンスを作らずに呼べる
    """
    return f"現在の総店舗数: {cls.total_shops}店"

@classmethod
def from_dict(cls, data):
    """
    辞書からインスタンスを生成（ファクトリーメソッド）

    特徴:
    - 別の方法でインスタンスを作成
    - 代替コンストラクタとして使用
    """
    return cls(
        owner_name=data["owner"],
        stock=data["stock"]
    )

@classmethod
def change_tax_rate(cls, new_rate):
    """
    税率を変更（全店舗に影響）

    特徴:
    - クラス変数を変更
    - 全インスタンスに影響する
    """
    cls.tax_rate = new_rate
    print(f"📢 税率を{new_rate*100}%に変更しました")

# ========================================
# 3. スタティックメソッド
# ========================================
@staticmethod
def calculate_total(quantity, price=150):
    """
    合計金額を計算（補助関数）

    特徴:
    - @staticmethod デコレーター必須
    - self も cls も不要
    - インスタンスやクラスのデータにアクセスしない
    - ただの関数だが、クラスに属する
    """
    return quantity * price

@staticmethod
```

```python
    def validate_stock(stock):
        """
        在庫数のバリデーション

        特徴:
        - 検証ロジックなど、独立した処理
        - クラスに関連するが、データは使わない
        """
        return isinstance(stock, int) and stock >= 0


# ========================================
# 使用例
# ========================================

# インスタンスメソッド
shop = DaifukuShop("うさうさ", 20)
shop.sell(5)  # インスタンスから呼ぶ

# クラスメソッド（インスタンス不要）
print(DaifukuShop.get_total_shops())  # クラスから直接呼べる

# ファクトリーメソッド
data = {"owner": "もちもち", "stock": 15}
shop2 = DaifukuShop.from_dict(data)

# スタティックメソッド（インスタンス不要）
total = DaifukuShop.calculate_total(10, 150)
print(f"合計: ¥{total}")
```

⚠️ メソッドの種類での注意点

注意1: selfを忘れる

```python
class Shop:
    def sell(quantity):  # ❌ selfがない
        pass

# エラー: sell() takes 1 positional argument but 2 were given
```

注意2: デコレーターを忘れる

```python
```

```
class Shop:
    # @classmethod を忘れている
    def get_total(cls):  # ❌ デコレーターなし
        return cls.total

# 呼び出すとエラー
```

注意3: 間違ったメソッドタイプを選ぶ

```
python

class Shop:
    @staticmethod
    def sell(self, quantity):  # ❌ staticなのにselfがある
        pass
```

🎯 使い分けの判断チャート

```
Q: このメソッドは何をする？

├── インスタンスのデータを使う？
│   └── YES → インスタンスメソッド（self）
│
├── クラス全体に関わる処理？
│   ├── インスタンスを作る？
│   │   └── YES → クラスメソッド（@classmethod）
│   └── クラス変数を操作？
│       └── YES → クラスメソッド（@classmethod）
│
└── インスタンスもクラスも使わない？
    └── YES → スタティックメソッド（@staticmethod）
```

---

2. アクセス制御（public/private）

📖 Pythonのアクセス制御

重要: Pythonには厳密なprivate/protectedは存在しません！
命名規則による「紳士協定」です。

🎯 3つのアクセスレベル

命名	意味	アクセス	例
`name`	public（公開）	どこからでも	`self.stock`

命名	意味	アクセス	例
`_name`	protected（保護）	クラス内と継承先	`self._price`
`__name`	private（非公開）	クラス内のみ	`self.__secret`

💻 実際のコード

```python
```

```python
class DaifukuShop:
    """アクセス制御のデモ"""

    def __init__(self, owner_name, stock):
        # ========================================
        # public（公開変数）
        # ========================================
        self.owner_name = owner_name  # 外部からアクセスOK
        self.stock = stock          # 外部からアクセスOK

        # ========================================
        # protected（保護変数）
        # ========================================
        # 慣習: クラス内と継承先でのみ使用すべき
        # 技術的には外部からもアクセス可能
        self._base_price = 150      # 基本価格
        self._cost = 80             # 原価

        # ========================================
        # private（非公開変数）
        # ========================================
        # 慣習: クラス内でのみ使用すべき
        # 名前マングリングで外部からアクセスしにくくなる
        self.__secret_recipe = "特製餡の秘密"
        self.__profit_margin = 0.5    # 利益率

    # ========================================
    # public メソッド
    # ========================================
    def sell(self, quantity):
        """公開メソッド: 誰でも呼べる"""
        if not self._check_stock(quantity):  # protectedメソッドを呼ぶ
            return False

        self.stock -= quantity
        price = self._calculate_price(quantity)  # protectedメソッドを呼ぶ
        print(f"💰 {quantity}個販売: ¥{price}")
        return True

    # ========================================
    # protected メソッド
    # ========================================
    def _check_stock(self, quantity):
        """
        保護メソッド: クラス内と継承先で使用
```

```
        命名規則:
        - 先頭に _ を1つ付ける
        - 外部から呼ばないでほしいという意思表示
        - 技術的には呼べるが、呼ぶべきではない
        """
        if quantity > self.stock:
            print(f"❌ 在庫不足！")
            return False
        return True

    def _calculate_price(self, quantity):
        """
        保護メソッド: 価格計算

        継承先でオーバーライドすることを想定
        """
        return quantity * self._base_price

    # ========================================
    # private メソッド
    # ========================================
    def __calculate_profit(self, quantity):
        """
        非公開メソッド: クラス内でのみ使用

        命名規則:
        - 先頭に __ を2つ付ける
        - 名前マングリング（変換）が行われる
        - 外部から呼ぶのは非常に困難
        """
        revenue = quantity * self._base_price
        cost = quantity * self._cost
        return revenue - cost

    def show_profit(self, quantity):
        """公開メソッド: 利益を表示"""
        profit = self.__calculate_profit(quantity)  # privateメソッドを呼ぶ
        print(f"💵 {quantity}個の利益: ¥{profit}")


# ========================================
# アクセス制御のテスト
# ========================================

shop = DaifukuShop("うさうさ", 20)

# ========================================
```

```
# public: 問題なくアクセスできる
# ========================================
print(f"店長: {shop.owner_name}")  # ✅ OK
print(f"在庫: {shop.stock}")      # ✅ OK
shop.sell(5)                    # ✅ OK

# ========================================
# protected: アクセスできるが、すべきではない
# ========================================
print(f"基本価格: {shop._base_price}")  # ⚠️ 可能だが非推奨
shop._check_stock(10)                 # ⚠️ 可能だが非推奨

# ========================================
# private: アクセスは困難
# ========================================
# print(shop.__secret_recipe)  # ❌ エラー: AttributeError

# 名前マングリングで変換されている
print(shop._DaifukuShop__secret_recipe)  # 😈 技術的には可能だが、やるべきではない

# shop.__calculate_profit(5)  # ❌ エラー: AttributeError
```

⚠️ アクセス制御での注意点

注意1: Pythonにはガチガチのprivateはない

python

```
# 他の言語（Java等）: 完全にアクセス不可
# Python: 慣習的に「触らないでね」というだけ

class Shop:
    def __init__(self):
        self.__secret = "秘密"  # privateのつもり

shop = Shop()
# 技術的にはアクセス可能（名前マングリング）
print(shop._Shop__secret)  # "秘密" ← アクセスできてしまう
```

注意2: _(アンダースコア1つ) は完全に慣習

python

```python
class Shop:
    def _internal_method(self):
        pass

# 何も防げない
shop = Shop()
shop._internal_method()  # 呼べてしまう（でも呼ぶべきではない）
```

注意3: privateは継承先でも使えない

```python
class Parent:
    def __init__(self):
        self.__private = "秘密"

    def __private_method(self):
        return "秘密のメソッド"

class Child(Parent):
    def use_parent_private(self):
        print(self.__private)  # ❌ エラー！継承先でも使えない
        self.__private_method()  # ❌ エラー！
```

📊 アクセス制御のベストプラクティス

用途	使うべき	理由
外部API	public	利用者に使ってほしい
内部実装	protected `_`	変更する可能性がある
継承で使う	protected `_`	子クラスで使える
絶対に隠したい	private `__`	名前の衝突を避ける

💡 推奨パターン

```python
```

```python
class DaifukuShop:
    """推奨されるアクセス制御パターン"""

    def __init__(self, owner_name):
        # public: 外部から使ってほしい
        self.owner_name = owner_name
        self.stock = 20

        # protected: 内部実装（変更する可能性あり）
        self._base_price = 150
        self._cost = 80

        # private: 本当に隠したい（名前衝突を避ける）
        self.__secret_key = "xyz123"

    # public API
    def sell(self, quantity):
        """外部から呼んでほしいメソッド"""
        return self._execute_sale(quantity)

    # protected: 継承先でカスタマイズできる
    def _execute_sale(self, quantity):
        """継承先でオーバーライド可能"""
        self.stock -= quantity
        return self._base_price * quantity

    # private: 絶対に触らせたくない
    def __encrypt_data(self, data):
        """内部でのみ使用する暗号化処理"""
        return data + self.__secret_key
```

---

3. 継承での注意点

📖 継承の基本ルール

継承は強力ですが、誤用すると複雑化します。

⚠️ 10の重要な注意点

注意1: is-a関係が成り立つかチェック

```python
```

```python
# ❌ 悪い例: 「車はエンジンである」は成り立たない
class Car(Engine):  # 間違った継承
    pass

# ✅ 良い例: 「車はエンジンを持つ」（コンポジション）
class Car:
    def __init__(self):
        self.engine = Engine()  # has-a関係
```

注意2: 継承は浅く（2-3階層まで）

python

```python
# ❌ 悪い例: 深すぎる継承
class A:
    pass
class B(A):
    pass
class C(B):
    pass
class D(C):
    pass
class E(D):  # 5階層は深すぎる！
    pass

# ✅ 良い例: 浅い継承
class Shop:
    pass
class PremiumShop(Shop):  # 2階層で十分
    pass
```

注意3: super()を忘れない

python

```python
class Parent:
    def __init__(self, name):
        self.name = name

class Child(Parent):
    def __init__(self, name, age):
        # ❌ 悪い例: 親の初期化を忘れる
        self.age = age  # nameが初期化されない！

    # ✅ 良い例
    def __init__(self, name, age):
        super().__init__(name)  # 親を初期化
        self.age = age
```

注意4: メソッド名の衝突

```python
class Parent:
    def process(self):
        print("親の処理")

class Child(Parent):
    def process(self):  # 親のメソッドを上書き
        print("子の処理")
        # 親の処理を呼びたい場合はsuper()を使う
        super().process()  # 親も呼ぶ
```

注意5: 親のprivateメソッドは使えない

```python
class Parent:
    def __private_method(self):  # private
        return "秘密"

class Child(Parent):
    def use_parent(self):
        return self.__private_method()  # ❌ エラー！使えない
```

注意6: 多重継承のMRO（メソッド解決順序）

```python
```

```python
class A:
    def method(self):
        print("A")

class B(A):
    def method(self):
        print("B")

class C(A):
    def method(self):
        print("C")

class D(B, C):  # 多重継承
    pass

# MRO（解決順序）を確認
print(D.__mro__)
# (<class 'D'>, <class 'B'>, <class 'C'>, <class 'A'>, <class 'object'>)

d = D()
d.method()  # "B" が出力される（MROの順番）
```

注意7: コンストラクタの引数の変更

```python
class Parent:
    def __init__(self, name):
        self.name = name

class Child(Parent):
    # ❌ 悪い例: 引数を勝手に変更
    def __init__(self, name, age, address):  # 増やしすぎ
        super().__init__(name)
        self.age = age
        self.address = address

# ✅ 良い例: 最小限の追加
class Child(Parent):
    def __init__(self, name, age):  # 1つだけ追加
        super().__init__(name)
        self.age = age
```

注意8: Liskovの置換原則

```python
# 親クラスを使える場所では、子クラスも使えるべき

class Bird:
    def fly(self):
        return "飛ぶ"

class Penguin(Bird):
    def fly(self):
        raise Exception("ペンギンは飛べない！") # ❌ 原則違反

# 親クラスの動作を壊している
```

注意9: 属性の追加は慎重に

```python
class Parent:
    def __init__(self):
        self.value = 10

class Child(Parent):
    def __init__(self):
        super().__init__()
        self.value = 20  # 親の属性を上書き
        self.new_value = 30  # ✅ 新しい属性を追加

# 上書きは注意が必要
```

注意10: 抽象メソッドの実装忘れ

```python
```

```
from abc import ABC, abstractmethod

class Shop(ABC):
    @abstractmethod
    def sell(self):
        """子クラスで必ず実装すること"""
        pass

class MyShop(Shop):
    pass  # ❌ sellを実装していない

# インスタンス化しようとするとエラー
# shop = MyShop()  # TypeError
```

💻 継承の良い例・悪い例

python

```python
# ========================================
# 悪い例
# ========================================

class BadExample1(list, dict):  # ❌ 意味のない多重継承
    pass

class BadExample2:
    """❌ 巨大な神クラス"""
    def method1(self): pass
    def method2(self): pass
    # ... 50個のメソッド
    def method50(self): pass

class BadExample3(Parent):
    """❌ 親のメソッドをすべてオーバーライド"""
    # 親の機能をほとんど使っていない
    # → 継承ではなくコンポジションを使うべき
    pass


# ========================================
# 良い例
# ========================================

class DaifukuShop:
    """✅ シンプルな親クラス"""
    def __init__(self, owner_name, stock):
        self.owner_name = owner_name
        self.stock = stock

    def sell(self, quantity):
        """基本的な販売処理"""
        if quantity > self.stock:
            return False
        self.stock -= quantity
        return True

class PremiumDaifukuShop(DaifukuShop):
    """✅ 明確なis-a関係"""
    def __init__(self, owner_name, stock, vip_count=0):
        super().__init__(owner_name, stock)  # 親を初期化
        self.vip_count = vip_count  # 子クラス固有の属性

    def sell(self, quantity, is_vip=False):
        """親のメソッドを拡張"""
```

```
    result = super().sell(quantity)  # 親の処理を呼ぶ
    if result and is_vip:
        print(f"VIP特典: ポイント2倍！")
    return result
```

---

4. メソッドのオーバーライド

📖 オーバーライドとは

親クラスのメソッドを子クラスで上書きすること。

🎯 オーバーライドのルール

1. メソッド名を同じにする
2. 引数の数を勝手に変えない（増やすのはOK）
3. 親の機能を壊さない
4. 必要なら`super()`で親を呼ぶ

💻 正しいオーバーライド

python

```python
class DaifukuShop:
    """親クラス"""

    def __init__(self, owner_name):
        self.owner_name = owner_name
        self.stock = 20

    def sell(self, quantity):
        """販売メソッド（基本）"""
        if quantity > self.stock:
            print("❌ 在庫不足")
            return False

        self.stock -= quantity
        price = quantity * 150
        print(f"💰 {quantity}個販売: ¥{price}")
        return True

    def show_info(self):
        """情報表示"""
        print(f"🏪 {self.owner_name}店長の店")
        print(f"📦 在庫: {self.stock}個")


class PremiumDaifukuShop(DaifukuShop):
    """子クラス"""

    def __init__(self, owner_name, vip_count=0):
        # ✅ 正しい: 親の初期化を呼ぶ
        super().__init__(owner_name)
        self.vip_count = vip_count

    # ========================================
    # パターン1: 完全に置き換える
    # ========================================
    def sell(self, quantity, discount=0):
        """
        販売メソッドを完全にオーバーライド
        親の処理を呼ばずに、独自実装
        """
        if quantity > self.stock:
            print("❌ 在庫不足")
            return False

        self.stock -= quantity
        price = quantity * 150 * (1 - discount)  # 割引機能追加
```

```
        print(f"✨ {quantity}個販売: ¥{int(price)} (割引{discount*100}%)")
        return True

    # ========================================
    # パターン2: 親の処理を拡張（推奨）
    # ========================================
    def show_info(self):
        """
        情報表示をオーバーライド
        親の処理を呼んでから、追加情報を表示
        """
        # 親の処理を実行
        super().show_info()

        # 追加情報を表示
        print(f"👑 VIP会員: {self.vip_count}名")


# 使用例
shop = PremiumDaifukuShop("うさうさ", vip_count=5)
shop.sell(3, discount=0.1)  # オーバーライドされたメソッド
shop.show_info()  # 親+子の処理
```

⚠️ オーバーライドの注意点

注意1: 引数を減らさない

python

```
class Parent:
    def method(self, a, b, c):
        pass

class Child(Parent):
    def method(self, a):  # ❌ 引数を減らしている
        pass

# 親クラスを期待するコードが壊れる
```

注意2: 返り値の型を変えない

python

```python
class Parent:
    def get_value(self):
        return 10  # int を返す

class Child(Parent):
    def get_value(self):
        return "10"  # ❌ str を返している
```

注意3: 例外の種類を変えない

```python
class Parent:
    def process(self):
        raise ValueError("エラー")

class Child(Parent):
    def process(self):
        raise TypeError("エラー")  # ❌ 例外の種類を変えている
```

---

5. super()の正しい使い方

📖 super()とは

親クラスのメソッドを呼び出すための関数。

🎯 super()の3つの用途

用途1: コンストラクタで親を初期化

```python
class Parent:
    def __init__(self, name):
        self.name = name

class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)  # 親を初期化
        self.age = age
```

用途2: オーバーライドしたメソッドで親を呼ぶ

```python
```

```python
class Parent:
    def greet(self):
        print("こんにちは")

class Child(Parent):
    def greet(self):
        super().greet()  # 親のgreetを呼ぶ
        print("私は子です")
```

用途3: 多重継承でMROに従って呼ぶ

python

```python
class A:
    def method(self):
        print("A")

class B(A):
    def method(self):
        print("B")
        super().method()  # MROに従ってAを呼ぶ

class C(A):
    def method(self):
        print("C")
        super().method()

class D(B, C):
    def method(self):
        print("D")
        super().method()  # MRO: D → B → C → A

d = D()
d.method()
# 出力:
# D
# B
# C
# A
```

⚠️ super()の注意点

注意1: super().init()を忘れる

python

```
class Parent:
  def __init__(self, name):
    self.name = name

class Child(Parent):
  def __init__(self, age):
    # super().__init__()を忘れている
    self.age = age

child = Child(10)
print(child.
```