クラスとインスタンス

概要 - 「クラス」と「インスタンス」とは何か

- 「インスタンス」とは、プロパティやメソッドを持った実体のこと
- 「インスタンス」は、クラスから生成される
- ひとつの「クラス」から、複数の「インスタンス」を生成できる

「クラス」から話をはじめるとかえって分かりにくい。
最終的に生成して操作したいのは「インスタンス」のほう。

なので、「インスタンスとは何か」というところから話をはじめたほうがおそらく分かりやすい。

「インスタンス」は、「クラス」を元にして生成する。
そして、「インスタンス」は、「プロパティ(インスタンス変数)」と「インスタンスメソッド」の2つを有する。

属性(*1)	性質	説明	例(人間であれば)
プロパティ (インスタンス変数)	静的	インスタンスが持つ基本情報	身長、体重、苗字、名前、 ...
メソッド	動的	インスタンスができる動作	ジュースを買う、コーヒーゼリーを食べる、皿を洗う、コードを書く、叫ぶ、 ...

このようなプロパティ、メソッドを有した「ヒト」のような「インスタンス」を作るとして、その雛形になるものが「クラス」。

雛形になる「クラス」をしっかり作っておけば、その「クラス」から「インスタンス」を作るのは容易になる。
ひとつのクラスから、少しずつ特徴の異なる、それぞれ別個の存在たる複数の「ヒト」を次々に作ることもできる。

ということで、クラスとインスタンスの関係は以下のようになる。

クラス/インスタンス	特徴
クラス	インスタンスを生成するための雛形
インスタンス	クラスから生成された実体

(*1)
エクセルVBAでは、オブジェクトのプロパティ、メソッドをあわせて「メンバー」と呼んだ。
Pythonでは、プロパティ、メソッドをあわせて「属性(attribute)」と呼ぶ。

クラスの定義

クラスは、以下の構文で定義する。

```
class Metal:
    pass
```

上記は、何のプロパティもメソッドの有さないクラス。
なお、 pass は、「文法上なんらかの処理を記述する必要があるが、何もしない場合」に記述する文。

インスタンスは、クラスを元にして生成する。

```
class Metal:
    pass


my_metal = Metal()
```

クラス/インスタンスの属性

クラス/インスタンスには、属性(attribute)を持たせることができる。

属性の種類	説明
インスタンス変数	インスタンスのデータを保持する
インスタンスメソッド	インスタンスの振る舞いを定義する
クラス変数	クラスのデータを保持する
クラスメソッド	クラスの振る舞いを定義する

インスタンスの属性 - インスタンス変数とインスタンスメソッド

プロパティ、メソッドを有するインスタンスを作ってみる。

以下で、 Metal クラスを元にして作られたインスタンスは、所有する貴金属の名前、重さ、価格をプロパティとして持つ。
また、自身の情報を出力するメソッドや、追加で入手した貴金属の情報を組み入れるメソッドを持つ。

```
class Metal:
    type_name = "silver"
    total_price = 10000
    amount = 2000

    def get_per_unit_price(self):
        """ 1グラムあたりの単価を返す """
        return self.total_price / self.amount


my_silver = Metal()

print(my_silver.amount, my_silver.total_price)

result = my_silver.get_per_unit_price()
print(result)
```

メソッドはカラブルで、引数を受け取ることができる。
ただし、上の例のとおり、メソッドの定義では、仮引数のうち、第一引数は `self` 。これは、インスタンス自身を指す。(*2)
呼び出し側から受け取った実引数は、第二引数以降に渡される。

(*2)
「インスタンス」であって、「クラス」ではないので注意。
第一引数の名称は `self` でなくても良いのだが、慣例により `self` とする。

以下では、引数つきのメソッドも定義した。

```
class Metal:
    type_name = "silver"
    total_price = 10000
    amount = 2000

    def get_per_unit_price(self):
        """ 1グラムあたりの単価を返す """
        return self.total_price / self.amount

    def add(self, price, new_amount):
        """ 追加で入手した貴金属の情報を組み入れる """
        self.total_price += price
        self.amount += new_amount


my_silver = Metal()

print(my_silver.amount, my_silver.total_price)

result = my_silver.get_per_unit_price()
print(result)

my_silver.add(price=10000, new_amount=3000)

print(my_silver.amount, my_silver.total_price)

result = my_silver.get_per_unit_price()
print(result)
```

@property デコレータ

メソッドに `@property` デコレータを付与すると、プロパティとして定義できる。
値の設定はできないが、値の取得はできるプロパティを定義したいときに使う。

プロパティなので、実引数を受け取らないメソッドのみをプロパティとできる。

```
class Metal:
    type_name = "silver"
    total_price = 10000
    total_weight = 2000

    @property
    def per_gram_price(self):
        """ 1グラムあたりの単価を返す """
        return self.total_price / self.total_weight

    def add(self, price, weight):
        self.total_price += price
        self.total_weight += weight


my_silver = Metal()

print(my_silver.total_weight, my_silver.total_price)

result = my_silver.per_gram_price
print(result)

my_silver.add(price=10000, weight=3000)

print(my_silver.total_weight, my_silver.total_price)

result = my_silver.per_gram_price
print(result)
```

プロパティはカラブルではないので、その点注意。慣れるまではよく間違える。

`__init__` メソッドと `__call__` メソッド

クラスを元にしてインスタンスを生成する際、 `__init__` メソッドが呼び出される。(*3)

クラスの生成時に指定した実引数は、 `__init__` メソッドに渡される。

`__call__` メソッドは、インスタンスを関数のように呼び出した際に呼び出される。

(*3)
いちおう述べておくと、厳密には、 `__new__` メソッドが呼び出され、その後、 `__init__` メソッドの第一引数に渡される。
ただし、 `__new__` メソッドは、通常は編集することはない。 `__init__` メソッドを編集できるのであれば、スキル的には問題ない。

```
class Metal:
    def __init__(self, type_name, base_price, weight):
        self.type_name = type_name
        self.total_weight = weight
        self.total_price = base_price * weight

    def __call__(self, *args, **kwargs):
        return f'{self.type_name} を {self.total_price:,} 円相当有しています'

    def get_per_unit_price(self):
        """ 1グラムあたりの単価を返す """
        return self.total_price / self.total_weight


silver = Metal('silver', 2000, 100)
platinum = Metal('platinum', 3000, 50)

silver_result = silver()
print(silver_result)

platinum_result = platinum()
print(platinum_result)
```

`__method_name__`

dunder method と呼ばれる特殊なメソッドがある。 magic method とも呼ばれる。

これらは、Pythonのクラス/インスタンスの振る舞いを定義するための既存のメソッド。

`__init__`, `__call__` も、dunder method の一種。

以下、dunder methodの例を紹介する。

メソッド名	説明
`__str__`	`str()` 関数の引数になったときに呼び出される
`__repr__`	`repr()` 関数の引数になったときに呼び出される
`__add__`	`+` 演算子を使った際に呼び出される
`__sub__`	`-` 演算子を使った際に呼び出される
`__mul__`	`*` 演算子を使った際に呼び出される
`__truediv__`	`/` 演算子を使った際に呼び出される
`__floordiv__`	`//` 演算子を使った際に呼び出される
`__mod__`	`%` 演算子を使った際に呼び出される
`__pow__`	`**` 演算子を使った際に呼び出される
`__eq__`	`==` 演算子を使った際に呼び出される
`__ne__`	`!=` 演算子を使った際に呼び出される
`__lt__`	`<` 演算子を使った際に呼び出される
`__le__`	`<=` 演算子を使った際に呼び出される
`__gt__`	`>` 演算子を使った際に呼び出される
`__ge__`	`>=` 演算子を使った際に呼び出される

上記のうちいくつかを実装してみよう。

なお、クラスが有する既存のメソッドを上書きすることを、「オーバーライド」という。

```
class Metal:
    def get_per_unit_price(self):
        """ 1グラムあたりの価格を返す """
        return self.total_price / self.amount

    def add(self, new_amount, new_total_price):
        """ 追加で入手した金属の重さと価格を追加する """
        self.amount += new_amount
        self.total_price += new_total_price

    def __init__(self, type_name, amount, total_price):
        self.type_name = type_name
        self.amount = amount
        self.total_price = total_price

    def __call__(self, ):
        """ インスタンスを関数のように呼び出すときに使う """
        return f'{self.get_per_unit_price()}円相当の{self.type_name}を有しています'

    def __str__(self):
        """ str() 関数の引数になったときに呼び出される """
        return f"{self.type_name} {self.amount}g {self.total_price}円"

    def __repr__(self):
        """ repr() 関数の引数になったときに呼び出される """
        return f"Metal({self.type_name}, {self.amount}, {self.total_price})"

    def __add__(self, other):
        return self.total_price + other.total_price

    def __sub__(self, other):
        return self.total_price - other.total_price

    def __eq__(self, other):
        if self.total_price != other.total_price:
            return False
        if self.type_name != other.type_name:
            return False
        return True


my_metal = Metal('platinum', 100, 35000)
gold = Metal('platinum', 200, 35000)

call_result = my_metal()
print(call_result)

str_result = str(my_metal)
print(str_result)

repr_result = repr(my_metal)
print(repr_result)

str_result_1 = str(my_metal)
str_result_2 = my_metal.__str__()

repr_result_1 = repr(my_metal)
repr_result_2 = my_metal.__repr__()

total_metal_price_1 = my_metal + gold
total_metal_price_2 = my_metal.__add__(gold)

minus_result_1 = my_metal - gold
minus_result_2 = my_metal.__sub__(gold)
print(minus_result_1, minus_result_2)

eq_result_1 = my_metal == gold
eq_result_2 = my_metal.__eq__(gold)
print(eq_result_1, eq_result_2)

print("終了しました")
```

インスタンスの属性へのアクセス

一度生成したインスタンスについて、生成後に属性を取得/追加/変更/削除できる。
以下の構文を使う。

属性の操作	説明	補足(あれば)
`hasattr(インスタンス, 属性名)`	属性の存在確認	属性が存在する場合は、Trueを返す。 属性が存在しない場合は、False を返す。
`getattr(インスタンス, 属性名, [初期値])`	属性の取得	属性が存在しない場合は、初期値を返す。 初期値がない場合はエラーになる。
`setattr(インスタンス, 属性名, 値)`	属性の設定	属性が存在しない場合は、追加される。 すでに属性が存在する場合は、上書きされる
`delattr(インスタンス, 属性名)`	属性の削除	属性が存在する場合は、その属性を削除する。 属性が存在しない場合は、エラーになる。あまり出番はない。
`インスタンス.属性名 = 値`	属性の設定	属性が存在しない場合は、追加される。 すでに属性が存在する場合は、上書きされる
`インスタンス.属性名` (*4)	属性の取得	すでに属性が存在する場合は、その値を取得する。 属性が存在しない場合は、エラーになる

(*4)
`インスタンス.属性名` での属性の取得は、`getattr(インスタンス, 属性名)` と同じ動作をする。
両者の違いは、`getattr(インスタンス, 属性名)` は、属性が存在しない場合は、初期値が設定されていれば戻り値を返すが、`インスタンス.属性名` は、属性が存在しない場合は、エラーになる点。

なお、いちおう書いておくと、上記の構文で取得できるのは属性。
なので、プロパティだけでなくメソッドもハンドルできる。

```
class Metal:
    def __init__(self, type_name, base_price, amount):
        self.type_name = type_name
        self.amount = amount
        self.total_price = base_price * amount

    def get_per_unit_price(self):
        return self.total_price / self.amount


silver = Metal('silver', 6000, 120)

print(hasattr(silver, 'type_name'))
print(hasattr(silver, 'company_name'))

print(getattr(silver, 'type_name'))
# print(getattr(silver, 'company_name'))
print(getattr(silver, 'company_name', '購入元不明'))

setattr(silver, 'company_name', 'ガラパゴス貴金属')
print(getattr(silver, 'company_name'))

delattr(silver, 'company_name')
print(hasattr(silver, 'company_name'))

silver.date = '2020/01/01'
print(silver.date)

silver.amount = 150
print(silver.amount)

# print(silver.origin_country) #存在しないのでエラー

# プロパティだけでなく、メソッドも取得できる。使うことはまずないが、紹介。
method = silver.get_per_unit_price
print(method())
```

参考

`__init__` 等のオーバーライドされるメソッドの仮引数に、`*args`, `**kwargs` 等を置いて、任意の引数を受け取れるようにするケースもまま見うけられる。
setattr を使う好例として紹介。

```
class Metal:
    def __init__(self, type_name, amount, base_price, **kwargs):
        self.type_name = type_name
        self.amount = amount
        self.total_price = base_price * amount

        for key, value in kwargs.items():
            setattr(self, key, value)

    def get_per_unit_price(self):
        return self.total_price / self.amount


silver = Metal('silver', 2000, 100, date='2022-10-14',
               origin_country='Argentina', store='ガラパゴス貴金属',
               memo='担当K氏')

print(silver.get_per_unit_price())

print(silver.type_name)
print(silver.amount)
print(silver.total_price)

print(silver.date)
print(silver.origin_country)
print(silver.store)
print(silver.memo)
```

クラスの属性 - クラス変数とクラスメソッド

クラス変数

クラス変数は、クラスに属するプロパティ(変数)。
クラスから参照できる。
また、インスタンスからも参照できる。
ただし、インスタンスで同名の変数を定義した場合は、インスタンスからは参照できなくなる。

```
class Metal:
    asset_type = '貴金属'

    def __init__(self, type_name, base_price, amount):
        self.type_name = type_name
        self.amount = amount
        self.total_price = base_price * amount


print(Metal.asset_type)

silver = Metal('silver', 2000, 100)
platinum = Metal('platinum', 3000, 50)

print(silver.asset_type)
print(platinum.asset_type)
```

クラス変数は、クラスに直接アクセスすると変更できる。

一方、インスタンスで同名の変数を定義することができる。
ただし、インスタンスで同名の変数を定義した場合、インスタンスからは、同名のクラス変数に簡単にはアクセスできなくなる。(ちょっと凝った方法を使えば不可能ではない)
この件が問題になることは滅多にないが、知識としては知っておくべき。

以下の挙動だと意識しておけば、実運用ではおそらく問題ない。

- インスタンスは、属性を指定すると、まずは、インスタンス変数を探す
- インスタンス変数がなければ、クラス変数を探す
- インスタンスを作ったあと、クラス変数を変更しないようにする

```
class Metal:
    asset_type = '貴金属'

    def __init__(self, type_name, base_price, weight):
        self.type_name = type_name
        self.total_weight = weight
        self.total_price = base_price * weight


silver = Metal('silver', 2000, 100)
platinum = Metal('platinum', 3000, 50)

print(Metal.asset_type)
print(silver.asset_type)
print(platinum.asset_type)

# platinum インスタンスの type 属性を新たに設定(クラスの type 属性ではないので注意！)
platinum.asset_type = 'プラチナ'

print(Metal.asset_type)
print(silver.asset_type)
print(platinum.asset_type)

# クラスの type 属性を変更
Metal.asset_type = '地金'

print(Metal.asset_type)
print(silver.asset_type)
print(platinum.asset_type)  # gold.type_name は gold インスタンスの type 属性 なので 'stock'
print(platinum.__class__.asset_type)  # クラスの type 属性を参照するのに# __class__ を使った
```

クラスメソッド

クラスメソッドは、クラスに属するメソッドである。 クラスメソッドの定義では、仮引数のうち、第一引数は `cls` 。これは、クラス自身を指す。(*5)
呼び出し側から受け取った引数は、第二引数以降に渡される。

(*5)
クラスであって、インスタンスではないので注意。
第一引数の名称は `cls` でなくても良いのだが、慣例により `cls` とする。

```
class Metal:
    asset_type = '貴金属'

    @classmethod
    def get_asset_type(cls):
        return cls.asset_type

    @classmethod
    def set_asset_type(cls, asset_type):
        cls.asset_type = asset_type

    def __init__(self, type_name, base_price, weight):
        self.type_name = type_name
        self.total_weight = weight
        self.total_price = base_price * weight


silver = Metal('silver', 2000, 100)

print(Metal.get_asset_type())
print(silver.get_asset_type())

Metal.set_asset_type('地金')

print(Metal.get_asset_type())
print(silver.get_asset_type())
```

クラスの継承とオーバーライド

既存のクラスを元にして、別のクラスを作ることができる。
これを継承という。

元のクラス(親クラス)を継承したクラス(子クラス)は、親クラスの属性をすべて引き継ぐ。
そのうえで、子クラス(サブクラスとも言う)で新たに属性を追加/変更/削除することができる。
これをオーバーライドという。

以下では、 Asset クラスを継承した Metal クラスを作ってみる。

なお、 super() は、親クラスを呼び出すための関数である。

asset_management/asset.py

```
class Asset:
    """ 資産クラス """
    type_name = 'asset'

    def __init__(self, type_name, total_price, ):
        self.type_name = type_name
        self.total_price = total_price

    def __str__(self):
        return f'{self.type_name} {self.total_price}'

    def __add__(self, other):
        return self.total_price + other.total_price

    def __eq__(self, other):
        return self.total_price == other.total_price

    def get_total_price(self):
        return self.total_price

    def get_info(self):
        return f'{self.type_name}を総額{self.total_price:,}円有しています。'
```

metal.py

```
import requests

from .asset import Asset


class Metal(Asset):
    type_name = 'metal'

    def __init__(self, name, base_price, amount):
        super().__init__(name, base_price * amount)
        self.amount = amount

    def get_info(self):
        return f'{self.name} {self.amount}グラム 総額{self.total_price:,}円有しています。'

    def get_price_per_unit(self):
        """ 購入単価を返す """
        return self.total_price / self.amount

    def get_current_price_per_unit(self):
        """ flask.pc5bai.com から最新の貴金属買取単価を得る """
        url = f'https://flask.pc5bai.com/metal/api/info/{self.name}'
        response = requests.get(url)
        return response.json()['buy']

    def get_current_total_price(self):
        return self.get_current_price_per_unit() * self.amount

    def sell(self, units):
        """ units 相当を売却する """
        url = 'https://flask.pc5bai.com/metal/api/buy/'
        data = {"name": self.name, "amount": units, "email": "foo@bar.com", "user": "山田太郎"
                }
        response = requests.post(url, json=data, )
        if response.status_code != 200:
            raise Exception('売却に失敗しました。')
        self.amount -= units
        self.total_price -= self.get_current_price_per_unit() * units
        return response.json()['price']
```

stock.py

```python
import csv

import requests

from .metal import Metal


class Stock(Metal):
    type_name = 'stock'

    def get_current_price_per_unit(self):
        """ flask.pc5bai.com から最新の株買取単価を得る """
        url = f'http://flask.pc5bai.com/stock/info/csv'
        response = requests.get(url)
        reader = csv.DictReader(response.text.splitlines())
        for row in reader:
            if row['company_name'] == self.name:
                return int(row['buy'])
        raise NotImplementedError

    def sell(self, units):
        """ units 相当を売却する """
        url = 'https://flask.pc5bai.com/stock/api/buy/'
        data = {
            "name": self.name,
            "amount": units,
            "email": "foo@bar.com",
            "user": "山田太郎"
        }
        response = requests.post(url, json=data, )  # headers={'Content-Type': 'application/json'})
        if response.status_code != 200:
            raise Exception('売却に失敗しました。')
        self.amount -= units
        self.total_price -= self.get_current_price_per_unit() * units
        return response.json()['price']
```

cls_51_inheritance.py

```python
from asset_management.asset import Asset
from asset_management.metal import Metal
from asset_management.stock import Stock

if __name__ == '__main__':
    jp_bond = Asset('日本国債', 1000000)
    print(jp_bond)
    print(jp_bond.get_info())
    print(jp_bond.get_total_price())

    gold = Metal('gold', 50000, 1000)
    print(gold)
    print(gold.get_info())
    print(gold.get_total_price())
    print(gold.get_price_per_unit())
    print(gold.get_current_price_per_unit())
    print(gold.get_current_total_price())

    # 一部を売る
    result = gold.sell(100)
    print(result)
    print(gold)
    print(gold.get_info())
    print(gold.get_total_price())

    orange = Stock('orange', 12000, 500)
    print(orange)
    print(orange.get_info())
    print(orange.get_price_per_unit())
    print(orange.get_current_price_per_unit())
    print(orange.get_current_total_price())

    # 一部を売る
    result = orange.sell(150)
    print(result)
    print(orange)
    print(orange.get_info())
    print(orange.get_total_price())

    print('終了しました')
```

なお、 Python では、すべてのクラスは、 object を継承して作られている。
継承元のクラスを省略した場合は、 object が継承元となる。
つまり、以下の2つは、同じ意味。

```python
class Metal:
    pass
```

```python
class Metal(object):
    pass
```

クラスの多重継承

クラスは、複数のクラスを継承することができる。
これを、多重継承という。

mixins.py

```python
import csv

import requests


class CSVDealsMixin:
    """ 株取引のためのメソッドを有する、多重継承用のクラス """

    def get_info(self):
        return f'{self.name} {self.amount}株 総額{self.total_price:,}円有しています。'

    def get_price_per_unit(self):
        """ 購入単価を返す """
        return self.total_price / self.amount

    def get_csv_deals(self):
        url = f'http://flask.pc5bai.com/stock/info/csv'
        response = requests.get(url)
        return response.text.splitlines()

    def get_current_price_per_unit_from_csv(self):
        """
        flask.pc5bai.com から最新の株買取単価を得る
        """
        reader = csv.DictReader(self.get_csv_deals())
        for row in reader:
            if row['company_name'] == self.name:
                return int(row['buy'])
        raise NotImplementedError

    def sell(self, units):
        """ units 相当を売却する """
        url = 'https://flask.pc5bai.com/stock/api/buy/'
        data = {"name": self.name, "amount": units, "email": "foo@bar.com", "user": "山田太郎"}
        response = requests.post(url, json=data, )
        if response.status_code != 200:
            raise Exception('売却に失敗しました。')
        self.amount -= units
        self.total_price -= self.get_current_price_per_unit() * units
        return response.json()['price']
```

cls_52_multiple_inheritance.py

```
from asset_management.asset import Asset
from asset_management.mixins import CSVDealsMixin


class PreciousMetal(CSVDealsMixin, Asset):
    """ CSV.、貴金属クラス """
    type_name = 'metal'

    def __init__(self, type_name, base_price, total_weight):
        super().__init__(type_name, base_price * total_weight)
        self.total_weight = total_weight

    def __str__(self):
        return f'{self.type_name} {self.total_price} {self.total_weight}'

    def get_info(self):
        return f'{self.type_name} {self.total_weight}グラム 総額{self.total_price:,}円有しています。'


gold = PreciousMetal('gold', 8500, 50)
silver = PreciousMetal('silver', 100, 200)

gold_base_price = gold.check_current_base_price()
print(gold_base_price)

silver_base_price = silver.check_current_base_price()
print(silver_base_price)

gold_total_price = gold.get_total_current_price()
print(gold_total_price)

silver_total_price = silver.get_total_current_price()
print(silver_total_price)

gold_current_value = gold.compare_with_current_value()
print(gold_current_value)

silver_current_value = silver.compare_with_current_value()
print(silver_current_value)
```

isinstance 関数、 issubclass 関数

isinstance() 関数は、オブジェクトが指定したクラスのインスタンスかどうかを判定する関数。
issubclass() 関数は、指定したクラスが指定したクラスのサブクラスかどうかを判定する関数。

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


class Metal(Asset):
    def __init__(self, name, price, weight):
        super().__init__(name, price)
        self.weight = weight


jp_fund = Asset('日本国債', 1500)
us_fund = Asset('米国債', 2000)
silver = Metal('silver', 2000, 100)
gold = Metal('gold', 1000, 50)

print(isinstance(jp_fund, Asset))
print(isinstance(silver, Asset))

print(issubclass(silver.__class__, Metal))
print(issubclass(Metal, Asset))
```

クラスとインスタンスに関連する「カラブル」

クラスとインスタンスに関連する「カラブル」としては、以下の4つがある。

1. クラスメソッドの実行
2. インスタンスの生成
3. インスタンスの呼び出し
4. インスタンスのメソッドの実行

すべて part05_callable で紹介済だが、以下に再掲しておく。

```python
class MyClass:
    @classmethod
    def some_class_method(cls):
        print('class method が呼ばれました')

    def __init__(self):
        print('init が呼ばれました')

    def __call__(self):
        print('call が呼ばれました')

    def some_instance_method(self):
        print('instnace method が呼ばれました')


# クラスメソッドの呼び出し
MyClass.some_class_method()

# インスタンスの生成
my_object = MyClass()

# インスタンスの呼び出し
my_object()

# インスタンスメソッドの呼び出し
my_object.some_instance_method()
```

「オブジェクト」なのか「インスタンス」なのか問題

「オブジェクト」と「インスタンス」という言葉がある。
どちらも、同じ意味で使われることが多い。
また、プログラミング言語によって定義が異なることがある。

そこで、「オブジェクト」と「インスタンス」という言葉の、 Python での一般的な使われ方を整理した。

分類	「オブジェクト」と称されるか	「インスタンス」と称されるか
ビルトインオブジェクト	○	△
ビルトイン以外の(クラスを元に生成された)オブジェクト	○	○

△:「インスタンス」ではないのか？といえば、もちろんインスタンス。ただ、そう呼ばれることはあまりなというだけ。

同時に、「オブジェクト型」という、すべてのクラスの継承元となるクラスもあることにも注意。
「オブジェクト」という言葉が出てきたときには、「オブジェクト型」のことを指しているのか、何らかのインスタンスのことを指しているのかを、文脈から判断する必要がある。

Pycharm の重要関連機能

機能名	ショートカット	説明
定義に移動	`[Ctrl] + [B]`	定義に移動するショートカット。 定義から実装で利用しているコードへの移動にも使える。
実装に移動	`[Ctrl] + [Alt] + [B]`	実装に移動するショートカット。 とはいえ、 `[Ctrl] + [B]` でも目的は十分に達成できる。
Structure	`[Ctrl] + [E]` → `[Structure]`	クラスの構造を示す。 継承元で定義されているメソッド等も見いだせる。
メソッドのオーバーライド	`[Ctrl] + [O]`	オーバーライドしたい既存のメソッドを選択して、簡単にオーバーライドメソッドを作ることができる。

flask.pc5bai.com apiリファレンス:

[flask.pc5bai.com apiリファレンスは、このページにあります]