オブジェクトとクラス

Python 入門
鈴木 敬彦

オブジェクト指向プログラミング

「オブジェクト指向プログラミング」に明確な定義はなく、また具体的な技術や方法でもありません。「オブジェクト指向プログラミング」はプログラミングの概念であり、考え方です。

初めて「オブジェクト指向」という言葉を用いた計算機学者 Alan Kay の言葉を借りるなら、

1. Everything Is An Object.
2. Objects communicate by sending and receiving messages (in terms of objects).
3. Objects have their own memory (in terms of objects).
4. Every object is an instance of a class (which must be an object).
5. The class holds the shared behavior for its instances (in the form of objects in a program list).
6. To eval a program list, control is passed to the first object and the remainder is treated as its message.

となります。しかし、現在の「オブジェクト指向」は必ずしもこの通りではなく、多くの研究者、開発者の考え方が反映された「オブジェクト指向」です。例えば、上記1の「Everything Is An Object」は広く受け継がれていますが、2の「Objects communicate by sending and receiving messages」（メッセージ・パッシングと呼ばれます）は、一部の言語では重要な役割を担っていますが、python を含むその他の言語ではあまり重要視されてはいません。

ここでは最初に「オブジェクト指向」に相対するプログラミングの概念として、「手続き型」の説明を行います。「手続き型」のプログラミングは、データ（変数等）とデータを処理するサブルーチン（関数）で構成されます。データとサブルーチンは互いに独立していて、データはサブルーチンを呼び出す側が管理します。サブルーチンは、与えられたデータに対して処理を行い、その結果を呼び出し元に返します。

```
>>> a = 1               # 呼び出し側が a という変数に 1 を代入します。
>>> a = add_two(a)      # add_two()と言うサブルーチンに a を渡し、
                        # その結果を a に代入します。
```

このように、変数 a とサブルーチン add_two() は互いに独立して存在し、関わりあうことは add_two() の引数として a が渡された時のみです。

「オブジェクト指向」では、「手続き型」とは正反対に、プログラムを構成する要素全てをオブジェクト（物）として捉え、オブジェクトを組み合わせてプログラムを構築しようとするプログラミングの概念です。オブジェクトについては、後述します。

オブジェクト指向プログラミングを用いる際の利点として、以下のような事柄が挙げられます。

1. 大規模なプロジェクト、多人数での開発に向いている。
2. カスタマイズしやすい。
3. コードの記述量が少なくて済む。

これらはオブジェクト指向の三大要素ともいわれる「カプセル化」、「継承」、「多態性」の利点とも言えます。その一方で、欠点として「設計の難易度が高い」ということが挙げられます。設計が悪ければ、オブジェクト指向プログラミングの利点を得ることはできません。しかし、それでも尚、多くの人々がオブジェクト指向でプログラムを構築しようとするのかといえば、ただ単に皆「楽をしたい」からです。オブジェクト指向は、一般的に、とても難しいものと捉えられがちですが、理解を妨げる要因の多くは、用語の乱立や適切ではない比喩であったりします。そして、理解すべき概念を理解していない状態で次へと進もうとすれば、より混乱してしまうという罠が待ち構えています。一つ一つの理解を決して疎かにする事なく学習するよう心掛けてください。

オブジェクト

オブジェクトとは「物」・「対象」という意味で、オブジェクトはデータを持ち、またそのデータに対する処理を行う関数を併せ持っています。オブジェクトの持つデータは、メンバー変数、フィールド等と呼ばれ、オブジェクトの持つ関数は、メンバー関数、メソッド等と呼ばれます。python では、これらを総じて属性（アトリビュート）と呼んでいます。

python 自体もオブジェクト指向でプログラミングされています。学習の初期段階で、データ型や変数、演算子というものを学習しました。

```
>>> a = 1
>>> b = a + 2
```

1行目では、a という変数に 1 という整数型のデータを代入しています。2行目では、変数 a に 2 を加算した結果で変数 a を更新しています。ここで a や b は変数、1 や 2 は単なる整数値だと学習しました。これは間違いではありませんが、python での実態は少々異なります。python では、 a も b も 1 も 2 も全てオブジェクトです。dir() という関数を使うと、引数に渡されたオブジェクトの属性を表示してくれます。python シェルで dir(a) を実行してみます。

```
>>> dir(a)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__',
'__class__', '__delattr__', '__dir__', '__divmod__', '__doc__',
'__eq__', '__float__', '__floor__', '__floordiv__', '__format__',
'__ge__', '__getattribute__', '__getnewargs__', '__gt__',
'__hash__', '__index__', '__init__', '__init_subclass__', '__int__',
'__invert__', '__le__', '__lshift__', '__lt__', '__mod__',
'__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__',
'__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__',
'__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__',
'__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'as_integer_ratio', 'bit_length', 'conjugate', 'denominator',
'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

上記のような結果が得られたはずです。結果は、a というオブジェクトが持つ属性名のリストです。前述した通り、属性名はメソッドやフィールドの名前です。オブジェクトの属性を参照するには、変数名と属性名をドット（ . ）で連結します。

```
<変数名>.<属性名>
```

ここでは試しにいくつかのメソッドを呼び出してみます。

```
>>> a.__int__()
1
>>> a.__add__(3)
4
```

1行目の __int__() というメソッドは、自身が保持している整数値を返すメソッドです。2行目の __add__() というメソッドは、自身が持っている整数値に引数で渡された整数値を加算した結果を返すメソッドです。

このように変数 a の実態は、ただ単に整数値を記憶するだけの「入れ物」ではなく、いわゆるオブジェクトと呼ばれる「物」であることが理解できたことと思います。

更には、変数 a に代入されていた数値 1 もまた、python ではオブジェクトです。先程と同様に dir() 関数を使ってみれば、同様の結果が返ってきます。

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__',
'__class__', '__delattr__', '__dir__', '__divmod__', '__doc__',
'__eq__', '__float__', '__floor__', '__floordiv__', '__format__',
'__ge__', '__getattribute__', '__getnewargs__', '__gt__',
'__hash__', '__index__', '__init__', '__init_subclass__', '__int__',
'__invert__', '__le__', '__lshift__', '__lt__', '__mod__',
'__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__',
'__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__',
'__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__',
'__setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'as_integer_ratio', 'bit_length', 'conjugate', 'denominator',
'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

もう少し詳しく何が起こっているのかを調べてみます。今度は、id() という関数を使用します。id() は、そのオブジェクトの ID を返してくれます。ID は一意で、異なるオブジェクトであれば必ず異なる ID を持っています。CPython では、そのオブジェクトのメモリー上での開始アドレスを返します。

```
>>> a = 1
>>> id(a)
4540971680
>>> id(1)
4540971680
```

a の ID と 1 の ID は同一です。つまり、同じオブジェクトであるということです。
更に先程の例を進めます。

```
>>> a = a + 2
>>> a
3
>>> id(a)
4540971744
>>> id(3)
4540971744
```

a に 2 を加えたものを a に代入しました。勿論、その結果は 3 となります。ID を見てみると、a の ID は先程とは異なっています。そして、3 の ID と同一です。

以上のことを踏まえて、先程の例で何が起こっているのかを考えてみましょう。

```
>>> a = 1
>>> a = a + 2
```

先ず、各行は実行されるたびに python のインタープリターで評価されます。

1行目では、1という整数型の値を持つオブジェクトが生成され、そのオブジェクトへの参照が変数 a に代入されます。

2行目では、2という整数型の値を持つオブジェクトが生成され、+ という演算子により呼び出される a.__add__() の引数として渡されます。そして a が参照するオブジェクトは、a.__add__(2) が実行された結果として、3という整数型の値を持つオブジェクトを生成し、そのオブジェクトへの参照を返します。最後にその参照が変数 a に代入されます。

通常、オブジェクトのメソッドは <オブジェクト名>.<メソッド名>() の形式で呼び出されます。演算子は特別な例で、python の設計段階からこのような仕組みが組み込まれています。また、一定の範囲内の整数のオブジェクトは、python 起動時に python によって生成されています。

クラス

クラスとインスタンス

クラスとは、オブジェクトを生成するための設計図、雛形です。定義されたクラスから生成され、メモリー上に展開された実体をインスタンスと呼びます。クラスを宣言するには、class 文を使用して、以下の形式で宣言します。

```
class <クラス名>:
    statement_1
    statement_2
        ⋮
    statement_n
```

斜体の statement_n の部分には、フィールドやメソッドの定義を記述します。

インスタンスを生成するには、クラス名に ( ) を付けて呼び出します。

```
変数 = <クラス名>()
```

例として、とても単純な何もしないクラスを宣言し、インスタンス化を行ってみます。

```
>>> class MyClass:
...     pass
...
>>> my_instance = MyClass()
>>> my_instance
<__main__.MyClass object at 0x10f11f2e0>
>>> dir(my_instance)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__']
```

無事にクラスが定義され、インスタンスを生成できたことがわかります。クラスの属性を表示させてみると、何も定義していないにも関わらず、色々と属性が定義されていることがわかります。これらは、クラスが宣言された際に python が暗黙のうちに定義する属性です。

クラスを他者と共有する場合には、関数等と同様に docstring を記述することを推奨します。また、docstring はクラスのみではなく、後述のメソッドにも docstring を記述することが望ましいです。

メソッド

クラスの中に定義された関数をメソッド、メンバー関数と呼びます。python では、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の3種類のメソッドを定義することができますが、一般にメソッドと言った時にはインスタンスメソッドを指します。ここでは、インスタンスメソッドについて説明します。

インスタンスメソッド

インスタンスメソッドを定義するには、クラス定義の中に、通常の関数を定義するのと同じように定義します。ただし、一つだけ通常の関数定義とは違いがあり、メソッドはその第1引数にインスタンスへの参照を受け取ります。一般的にインスタンスへの参照を受け取る変数の変数名には、「self」を用います。通常の引数は2番目以降に指定します。メソッドを呼び出すには、インスタンス名とメソッド名をドット（ . ）で連結して呼び出します。

```
>>> class MyMultiplicationClass:
...      def do_it(self, a, b=100):
...          print('%g × %g = %g' % (a, b, a*b))
...
>>> my_multiplier = MyMultiplicationClass()
>>> my_multiplier.do_it(3)
3 × 100 = 300
```

特別なメソッド

前のページの例でインスタンスの属性名を表示させました。これらの属性名のうちのいくつかは特殊メソッドと呼ばれるものです。例えば、__init__ は、クラスのインスタンスが生成された後、インスタンスが呼び出し側に返される前に呼び出されるメソッドです。インスタンスの初期化のためによく用いられるメソッドです。

```
>>> class MyMultiplicationClass:
...     def __init__(self, b=100):
...         print('__init__() called.')
...         self.b = b                  # インスタンス変数（後述）
...     def do_it(self, a):
...         print('%g × %g = %g' % (a, self.b, a*self.b))
...
>>>
>>> my_multiplier1 = MyMultiplicationClass()
__init__() called.
>>> my_multiplier1.do_it(4)
4 × 100 = 400
>>> my_multiplier2 = MyMultiplicationClass(1000)
__init__() called.
>>> my_multiplier2.do_it(4)
4 × 1000 = 4000
```

その他の特殊メソッドについては、下記 URL を参照してください。

https://docs.python.org/ja/3/reference/datamodel.html?#special-method-names

フィールド

フィールドとは、クラスの中に定義される変数です。メンバー変数とも呼ばれます。クラス、インスタンスに属する変数にはクラス変数とインスタンス変数の2種類があり、一般にメンバー変数、フィールドと言った時にはインスタンス変数を指します。インスタンス変数を参照するには変数名と属性名をドット（ . ）で連結し、その値はインスタンスごとに独立しています。クラス変数を参照するにはクラス名と属性名をドット（ . ）で連結し、その値はそのクラスとそのクラスの全てのインスタンスの間で共有されます。他の多くのプログラミング言語でクラスの中に変数を定義する際には、メソッドと同じレベルで変数を定義しますが、python では少々異なるので注意してください。python でインスタンス変数を定義するには、一般的に __init__() 内で定義します。

```
>>> class MyClass2:
...     a = 1                               # クラス変数を定義。
...     def __init__(self, b=2):
...         self.b = b                      # インスタンス変数を定義。
...
>>> MyClass2.a                              # クラス変数は、インスタンスが生成
1                                           # されていなくても参照可能です。
>>> MyClass2.b                              # クラス変数は、インスタンスが生成
Traceback (most recent call last):          # されていないと参照できません。
  File "<stdin>", line 1, in <module>
AttributeError: type object 'MyClass2' has no attribute 'b'
>>> c1 = MyClass2()                         # インスタンスを生成。
>>> c2 = MyClass2(3)
>>>
>>> c1.a, c2.a, c1.b, c2.b                  # 各インスタンスの変数を確認。
(1, 1, 2, 3)
>>> MyClass2.a = 999                        # クラス変数の値を変更。
>>> c1.b = -1                               # インスタンス変数の値を変更。
>>> c1.a, c2.a, c1.b, c2.b                  # 再度、変数を確認。
(999, 999, -1, 3)
>>>
>>> c2.a = 333                              # エラーにはなりませんが...
>>> c1.a, c2.a, c1.b, c2.b
(999, 333, -1, 3)
>>> MyClass2.a = 777
>>> c1.a, c2.a, c1.b, c2.b                  # 以後、c2 はクラス変数を参照
(777, 333, -1, 3)                           # することができなくなります。
```

上例の最後の6行について、クラス変数をインスタンス名から参照してその値を更新しようとすると、そのインスタンスのクラス変数自体が新たなインスタンス変数で上書きされてしまいます。（付録の「モンキーパッチ」の項を参照してください。）

余談ですが、うっかりクラス変数を新たなインスタンス変数で書き換えてしまっても、python でクラスの各部がどのように取り扱われているのかさえ知っていれば、復帰は可能です。

```
>>> c1.__dict__, c2.__dict__                # __dict__ にはインスタンスの書込
({'b': -1}, {'b': 3, 'a': 333})             # 可能な属性が保存されています。
>>> del c2.__dict__['a']                    # __dict__ から 'a' を削除。
>>> c1.__dict__, c2.__dict__
({'b': -1}, {'b': 3})
>>> c1.a, c2.a, c1.b, c2.b                  # 再度、クラス変数を参照できるように
(777, 777, -1, 3)                           # なりました。
```

フィールドの隠蔽、ゲッター、セッター

インスタンス変数の値を更新する際に、新しい値が正しい値か否かを確認することはよくあることであり、またその確認を怠るべきではありません。例えば、本来数値であるべきフィールドに文字列等の数値以外の値が代入されてしまっては、処理を正しく実行できません。実際に前出の例の様な方法では、新しい値を確認することができません。そのため、他のプログラミング言語ではインスタンス変数への直接的なアクセスを禁じる（隠蔽）方法が用意されており、そのインスタンス変数へアクセスするためのメソッド（アクセサー）を記述することが可能です。値を設定するためのメソッドをセッター（setter）、値を取得するメソッドをゲッター（getter）と呼びます。また、隠蔽されている変数をプライベート変数と呼びます。python でもインスタンス変数を隠蔽する方法が用意されており、プライベート変数を宣言するにはフィールド名を2つのアンダースコア（ _ ）で始まる名前にします。

```
>>> class MyClass3:
...     def __init__(self, x=3):
...         self.__my_private_var = x          # プライベート変数を定義。
...     def get_my_private_var(self):          # getter を定義。
...         return self.__my_private_var
...     def set_my_private_var(self, x):       # setter を定義。
...         if type(x) == int:
...             self.__my_private_var = x
...         else:
...             raise ValueError('整数以外はお断りします。')
...
>>> mc3 = MyClass3()                    # インスタンスを生成。
>>> mc3.__my_private_var                # プライベート変数を直接参照すると…
Traceback (most recent call last):      # エラーが発生します。
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass3' object has no attribute
'__my_private_var'
>>> mc3.get_my_private_var()            # getter からなら可能。
3
>>> mc3.set_my_private_var(1.23)        # setter からなら値の確認も可能。
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 10, in set_my_private_var
ValueError: 整数以外はお断りします。
>>> mc3.set_my_private_var(-1)
>>> mc3.get_my_private_var()
-1
```

しかし、残念ながら python では完全に隠蔽することはできず、クラス名と変数名とアンダースコアの組み合わせでプライベート変数に直接アクセスすることができてしまいます。

```
>>> mc3._MyClass3__my_private_var
-1
>>> mc3._MyClass3__my_private_var = 999
>>> mc3._MyClass3__my_private_var
999
```

プライベートなインスタンス変数の定義方法には、@property というデコレーター（付録の「デコレーター」を参照）を使用するもう一つ別の方法があります。@property を用いてインスタン

ス変数並びにゲッター、セッターを定義するには、クラス定義内にて以下の形式で宣言し記述します。クラス内で @property を用いて宣言されたインスタンス変数にアクセスするには、変数名の前にアンダースコア 2つを付けて参照します。@マークで始まる行がデコレーターと呼ばれるもので、デコレーターは次に続くメソッドや関数を「修飾」します。

```
@property
def <変数名>(self):
    ...
@<変数名>.getter
def <変数名>(self):
    return self.__<変数名>
@<変数名>.setter
def <変数名>(self, ...):
    ...
```

property も、ゲッター、セッターもメソッド名が変数名と同じであることに注意してください。名前が異なると、正しく機能しません。

```
>>> class MyClass4:
...     def __init__(self, x=7):
...         self.__my_private_var = x
...     @property                                   # プライベート変数を定義。
...     def my_private_var(self):
...         print('プロパティーが呼ばれました。')
...     @my_private_var.getter                      # getter を定義。
...     def my_private_var(self):
...         print('ゲッターが呼ばれました。')
...         return self.__my_private_var
...     @my_private_var.setter                      # setter を定義。
...     def my_private_var(self, x):
...         print('セッターが呼ばれました。')
...         self.__my_private_var = x
...
>>> mc4 = MyClass4()                          # インスタンスを生成。
>>> mc4.my_private_var                        # プライベート変数の値を取得
ゲッターが呼ばれました。
7
>>> mc4.my_private_var = -7                   # プライベート変数に値を設定
セッターが呼ばれました。
>>> mc4.my_private_var
ゲッターが呼ばれました。
-7
```

前出の方法がいけないわけではありませんが、値の設定に代入演算子を使用できたり、値の取得・設定時の挙動を自分で制御できるため、@property を使用した方がより分かり易く、使い易いコードを記述できることと思います。

また、ゲッターが定義されていない場合には、ゲッターの代わりに @property で修飾されたメソッドが呼び出されます。セッターが定義されていない場合は、代入演算子で値を更新することはできません。この他に、必要があれば デリーター（deleter）を定義することも可能です。

```
>>> class MyClass5:
...     def __init__(self, x=7):
...         self.__my_private_var = x
...     @property                                   # プライベート変数を定義。
...     def my_private_var(self):
...         print('プロパティーが呼ばれました。')
...         return self.__my_private_var
...
>>>
>>> mc5 = MyClass5()                              # インスタンスを生成。
>>> mc5.my_private_var                            # プライベート変数の値を取得
プロパティーが呼ばれました。
7
>>> mc5.my_private_var = -7                       # プライベート変数に値を設定...
Traceback (most recent call last):                # すると、setter が無いため、
  File "<stdin>", line 1, in <module>             # エラーとなります。
AttributeError: can't set attribute
```

ただし、@property を用いる方法でも、インスタンス変数を完全に隠蔽することはできません。先程の例と同様に、クラス名と変数名の組み合わせで直接アクセスすることが可能です。

```
>>> mc5._MyClass5__my_private_var
7
>>> mc5._MyClass5__my_private_var = 777
>>> mc5._MyClass5__my_private_var
777
```

この様に、フィールドの完全な隠蔽は python では不可能です。また、フィールドだけではなくメソッドも同様です。（他の多くのプログラミング言語では、プライベートメソッドを定義可能です。）

```
>>> class MyClass6:
...     def __init__(self):
...         pass
...     def bang(self):
...         print('bang() が呼ばれました。')
...     def __bang(self):
...         print('__bangが呼ばれました。')
...
>>> mc6 = MyClass6()
>>> mc6.bang()
bang() が呼ばれました。
>>> mc6.__bang()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyClass6' object has no attribute '__bang'
>>> mc6._MyClass6__bang()
__bangが呼ばれました。
```

オブジェクト指向の三大要素

冒頭でも触れましたが、オブジェクト指向プログラミングの大きな特徴として、カプセル化、継承、多態性が挙げられます。これらはオブジェクト指向の基盤を担う重要な要素です。

カプセル化

カプセルとは、何かを詰めておくための容器のことを表します。プログラミングにおいてのカプセル化も同様で、様々なデータやコードを容器に詰め込みます。その容器のことを、プログラミングでは、クラスと呼びます。前節までのクラスの話は、このカプセル化を実装するための方法です。カプセル化の狙いは、関連するデータとそれらを操作するコードを一つのオブジェクトにまとめて、外部に対しては必要とされるデータとコードのみを公開し、その他のデータとコードを隠蔽する事です。オブジェクトに属するデータはフィールド、メンバー変数等と呼ばれ、オブジェクトに属するコードはメソッド、メンバー関数等と呼ばれます。公開されたデータは、外部から直接もしくはメソッドを介して間接的に参照、変更することが可能です。非公開のデータとメソッドは、外部からアクセスできません。これを情報隠蔽（information hiding）と呼びます。ただし python においては、前述の通り、完全に情報を隠蔽することはできません。
オブジェクト/クラス設計の基本は、単一責任の原則に従って、一つの閉じた機能を構成するデータとそれらを操作するメソッドを定義することです。
単一責任の原則（single-responsibility principle、SRP）とは、Robert C. Martin によって提唱されたソフトウェア設計のための5つの原則（SOLID）の内の一つで、全てのクラスや関数がソフトウェアによって提供される機能の単一の部分に対して責任を持つべきであり、その責任はクラスや関数によって完全にカプセル化されるべきである、というものです。

継承

継承とは、受け継ぐことです。プログラミングにおける継承とは、既存クラスのデータやメソッドの構成を引き継いで、新しくクラスを定義する方法です。受け継がれる側のクラスを、基底（base）クラス、スーパークラス、親クラス等と呼びます。受け継いだ側のクラスを派生（derived）クラス、サブクラス、子クラス等と呼びます。派生クラスは一つ以上の基底クラスを持ち、基底クラスが一つであれば単一継承、複数であれば多重継承と呼びます。派生クラスは、基底クラスのデータとメソッドにアクセスでき、必要に応じて新たなデータやメソッドを追加する事ができます。更に、基底クラスのメソッドを派生クラスで新たに定義し直すことも可能で、これをオーバーライドと呼びます。オブジェクト指向プログラミングの用語でオーバーライドと似た名前のオーバーロード（多重定義）という機構もありますが、python では使えません。（オーバーロードは、引数の数や型が異なる同名のメソッドを複数定義することが可能で、実引数に応じて呼び出されるメソッドが選択されます。）
クラスの設計に際しては、基底クラスを派生クラスで置き換えても同一の結果が得られる事が求められます。これも前出の SOLID の一つで、リスコフの置換原則と呼ばれます。そして、何より大切なのことは、クラス或いは処理の共通点と相違点を体系的に分類して整理することです。

python でクラスを継承するには、派生クラスを宣言する際にクラス名の後に継承する基底クラスの名前を丸括弧で括って付加します。多重継承する場合は、カンマで区切って記述します。また、

派生クラス内から基底クラスのメソッドを参照するには、super() 関数を用います。クラス外部からフィールドやメソッドにアクセスする方法は、通常のクラスと変わりありません。

```
class <クラス名>(<基底クラス名>):
    statement_1
    statement_2
        ⋮
    statement_n
```

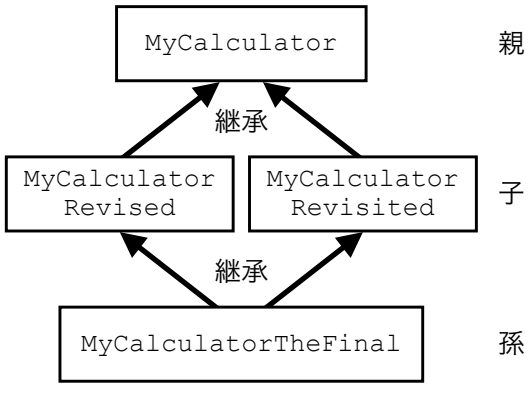
余談ですが、これまでに出てきた例の中でも継承は行われています。5頁の例を思い出してください。定義した覚えのない属性を確認する事ができました。

```
>>> class MyClass:
...     pass
...
>>> my_instance = MyClass()
>>> dir(my_instance)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__']
```

python でクラスを宣言すると、新規クラスは暗黙のうちに object クラスを継承するようになっています。上記の例の MyClass クラスと下記の例の YetAnotherMyClass クラスは同義です。

```
>>> class YetAnotherMyClass(object):
...     pass
...
>>> ya_my_instance = YetAnotherMyClass()
>>> dir(ya_my_instance)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__']
```

次に、継承の例を示します。ここでは単一継承と多重継承を行なっていますが、この継承パターンは特に菱形継承と呼ばれ、同じ基底クラスを持つ2つの派生クラスがあり、更にその2つの派生クラスを多重継承する派生クラスで構成されます。どのように継承されているか、基底クラスのメソッドがどのような順序で呼び出されているか等をよく確認するようにしてください。
最初にクラスの宣言を、その次にシェルでの実行例を例示します。



```
class MyCalculator:                                     # 親（基底）クラスを宣言
    @property                                           # プロパティ x
    def x(self):
        return self.__x
    @x.setter                                           # プロパティ x セッター
    def x(self, x):
        self.__x = x
    @property                                           # プロパティ y
    def y(self):
        return self.__y
    @y.setter                                           # プロパティ y セッター
    def y(self, y):
        self.__y = y
    def __init__(self, x=None, y=None):                          # 初期化
        print('initializing %s...' % __class__.__name__)
        self.__x = x if x else 0
        self.__y = y if y else 1
        print('%s is initialized.' % __class__.__name__)
    def do_add(self):                                            # 加算
        print('%s::%s' % (__class__.__name__, 'do_add()'))
        return self.__x + self.__y
    def do_sub(self):                                            # 減算
        print('%s::%s' % (__class__.__name__, 'do_sub()'))
        return self.__x - self.__y
    def do_something_enigmatic(self):
        print('%s::%s' % (__class__.__name__, 'do_something_enigmatic()'))
        x = self.__x
        y = self.__y
        return x**y/(y**x) if x < y else y**x/(x*y)

class MyCalculatorRevised(MyCalculator):                         # 子クラスを宣言
    def __init__(self, x=None, y=None):                          # 初期化
        print('initializing %s...' % __class__.__name__)
        super().__init__(x, y)
        print('%s is initialized.' % __class__.__name__)
    def do_mul(self):                                            # 乗算
        print('%s::%s' % (__class__.__name__, 'do_mul()'))
        return self.x * self.y

class MyCalculatorRevisited(MyCalculator):                       # 子クラスを宣言
    def __init__(self, x=None, y=None):                          # 初期化
        print('initializing %s...' % __class__.__name__)
        super().__init__(x, y)
        print('%s is initialized.' % __class__.__name__)
    def do_div(self):                                            # 整数除算
        print('%s::%s' % (__class__.__name__, 'do_div()'))
        return self.x // self.y, self.x % self.y

class MyCalculatorTheFinal(MyCalculatorRevised, MyCalculatorRevisited): # 孫ｸﾗｽ
    def __init__(self, x=None, y=None):                          # 初期化
        print('initializing %s...' % __class__.__name__)
        super().__init__(x, y)
        print('%s is initialized.' % __class__.__name__)
    def do_div(self):                                            # 除算
        print('%s::%s' % (__class__.__name__, 'do_div()'))
        return self.x / self.y
    def do_something_enigmatic(self):                            # オーバーライド
        print('%s::%s' % (__class__.__name__, 'do_something_enigmatic()'))
        x = self.x
        y = self.y
        return x**y/(y**x) if x > y else y**x/(x/y)
```

```
>>> mc1 = MyCalculator(1, 2)                # 親クラスをインスタンス化
initializing MyCalculator...
MyCalculator is initialized.
>>>
>>> mc2 = MyCalculatorRevised(3, 4)         # 子クラスをインスタンス化
initializing MyCalculatorRevised...
initializing MyCalculator...
MyCalculator is initialized.
MyCalculatorRevised is initialized.
>>>
>>> mc3 = MyCalculatorRevisited(5, 6)       # 子クラスをインスタンス化
initializing MyCalculatorRevisited...
initializing MyCalculator...
MyCalculator is initialized.
MyCalculatorRevisited is initialized.
>>>
>>> mc4 = MyCalculatorTheFinal(7, 8)        # 孫クラスをインスタンス化
initializing MyCalculatorTheFinal...
initializing MyCalculatorRevised...
initializing MyCalculatorRevisited...
initializing MyCalculator...
MyCalculator is initialized.
MyCalculatorRevisited is initialized.
MyCalculatorRevised is initialized.
MyCalculatorTheFinal is initialized.
>>>
>>> mc1.x, mc1.y                            # 親クラスの値を確認
(1, 2)
>>> mc1.do_add()                            # 親クラスのメソッド
MyCalculator::do_add()
3
>>> mc1.do_sub()                            # 親クラスのメソッド
MyCalculator::do_sub()
-1
>>> mc1.do_something_enigmatic()            # 親クラスのメソッド
MyCalculator::do_something_enigmatic()
0.5
>>>
>>> mc2.x, mc2.y                            # 子クラスの値を確認
(3, 4)
>>> mc2.do_add()                            # 継承した親クラスのメソッド
MyCalculator::do_add()
7
>>> mc2.do_sub()                            # 継承した親クラスのメソッド
MyCalculator::do_sub()
-1
>>> mc2.do_mul()                            # 追加した子クラスのメソッド
MyCalculatorRevised::do_mul()
12
>>> mc2.do_something_enigmatic()            # 継承した親クラスのメソッド
MyCalculator::do_something_enigmatic()
1.265625
>>>
>>> mc3.x, mc3.y                            # 子クラスの値を確認
(5, 6)
>>> mc3.do_add()                            # 継承した親クラスのメソッド
MyCalculator::do_add()
11
>>> mc3.do_sub()                            # 継承した親クラスのメソッド
MyCalculator::do_sub()
```

```
-1
>>> mc3.do_div()                                # 追加した子クラスのメソッド
MyCalculatorRevisited::do_div()
(0, 5)
>>> mc3.do_something_enigmatic()                # 継承した親クラスのメソッド
MyCalculator::do_something_enigmatic()
2.0093878600823047
>>>
>>> mc4.x, mc4.y                                # 孫クラスの値を確認
(7, 8)
>>> mc4.do_add()                                # 継承した親クラスのメソッド
MyCalculator::do_add()
15
>>> mc4.do_sub()                                # 継承した親クラスのメソッド
MyCalculator::do_sub()
-1
>>> mc4.do_mul()                                # 継承した子クラスのメソッド
MyCalculatorRevised::do_mul()
56
>>> mc4.do_div()                                # オーバーライドしたメソッド
MyCalculatorTheFinal::do_div()
0.875
>>> mc4.do_something_enigmatic()                # オーバーライドしたメソッド
MyCalculatorTheFinal::do_something_enigmatic()
2396745.1428571427
```

メソッドの解決順序は、mro() というクラスメソッドからも確認する事ができます。

```
>> MyCalculatorTheFinal.mro()
[<class '__main__.MyCalculatorTheFinal'>, <class
'__main__.MyCalculatorRevised'>, <class
'__main__.MyCalculatorRevisited'>, <class '__main__.MyCalculator'>,
<class 'object'>]
```

多態性

多態性は、ポリモーフィズム、多様性等とも呼ばれ、異なるオブジェクトに共通のインターフェースを持たせる手段です。例えば、前述の通り、python 内の全てのクラスは object クラスを継承しています。object クラスの属性を見てみると、これまでに何度か使用している __init__ メソッドを確認することができます。

```
>>> dir(object)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__']
```

全てのクラスは、この __init__ メソッドを継承し、殆どの場合はオーバーライドしてクラスの初期化を行います。クラスには様々なクラスが存在し、その処理内容もそれぞれ異なりますが、初期化のためのメソッド名は共通です。一般的に基底クラスには、インスタンス化を前提としない抽象基底クラスを継承元とし、共通化させるメソッドは宣言のみを行います。宣言のみで実装を持た

ないメソッドを抽象メソッドと呼びます。ここでは簡単な例として、正多面体の抽象基底クラスを定義して、正多面体の表面積と体積を返す各正多面体毎の派生クラスを定義します。

```
class PlatonicSolid:                          # 抽象基底クラス
    @property
    def faces(self):
        return self.__faces
    def volume(self):                         # 抽象メソッド
        raise NotImplementedError('method is not implemented yet.')
    def surface_area(self):                   # 抽象メソッド
        raise NotImplementedError('method is not implemented yet.')
    def __init__(self, faces):
        if   faces == 4: self.__faces = 4
        elif faces == 6: self.__faces = 6
        elif faces == 8: self.__faces = 8
        elif faces == 12:self.__faces = 12
        elif faces == 20:self.__faces = 20
        else: raise ValueError('正%d面体はこの世に存在しません。' % faces)

class Tetrahedron(PlatonicSolid):
    def __init__(self):
        super().__init__(4)
    def volume(self):          # 抽象メソッド volume() をオーバーライド
        return 2**0.5 / 12
    def surface_area(self):    # 抽象メソッド surface_area() をオーバーライド
        return 3**0.5

class Cube(PlatonicSolid):
    def __init__(self):
        super().__init__(6)
    def volume(self):          # 抽象メソッド volume() をオーバーライド
        return 1
    def surface_area(self):    # 抽象メソッド surface_area() をオーバーライド
        return 6

class Octahedron(PlatonicSolid):
    def __init__(self):
        super().__init__(8)
    def volume(self):          # 抽象メソッド volume() をオーバーライド
        return 2**0.5 / 3
    def surface_area(self):    # 抽象メソッド surface_area() をオーバーライド
        return 2 * 3**0.5

class Dodecahedron(PlatonicSolid):
    def __init__(self):
        super().__init__(12)
    def volume(self):          # 抽象メソッド volume() をオーバーライド
        return (15 + 7 * 5**0.5) * 0.25
    def surface_area(self):    # 抽象メソッド surface_area() をオーバーライド
        return 3 * (25 + 10 * 5**0.5)**0.5

class Icosahedron(PlatonicSolid):
    def __init__(self):
        super().__init__(20)
    def volume(self):          # 抽象メソッド volume() をオーバーライド
        return 5 * (3 + 5**0.5) / 12
    def surface_area(self):    # 抽象メソッド surface_area() をオーバーライド
        return 5 * 3**0.5
```

```
>>> ph1 = Tetrahedron()
>>> ph2 = Cube()
>>> ph3 = Octahedron()
>>> ph4 = Dodecahedron()
>>> ph5 = Icosahedron()
>>> for ph in [ph1, ph2, ph3, ph4, ph5]:
...     print('%12s: Surface Area: %9.6f, Volume: %9.6f' %
(ph.__class__.__name__, ph.surface_area(), ph.volume()))
...
 Tetrahedron: Surface Area:  1.732051, Volume:  0.117851
        Cube: Surface Area:  6.000000, Volume:  1.000000
  Octahedron: Surface Area:  3.464102, Volume:  0.471405
Dodecahedron: Surface Area: 20.645729, Volume:  7.663119
 Icosahedron: Surface Area:  8.660254, Volume:  2.181695
```

インターフェース、この場合はメソッド名、を共通にする事で処理の記述をこのように簡潔にすることができます。python で抽象基底クラスを定義する場合には、abc モジュールを使用するのが一般的です。（付録の「抽象基底クラス」の項を参照してください。）

また、人によってポリモーフィズムに含めるか否かで議論が分かれるところですが、抽象基底クラスや抽象メソッドを用いずに、ただ単に共通のメソッド名を持たせるダックタイピングという方法もあります。ダックタイピングの名前は、ダックテスト「もしもそれがアヒルのように歩き、アヒルのように鳴くのなら、それはアヒルに違いない」に由来しています。
例えば、int、str、list クラス等の属性を見てみると、object クラスの属性は当然全て継承されていますが、それら以外の属性も沢山あり、__add__ や __mul__ はこれら全てのクラスに共通して存在しています。__add__ はメソッドで、int クラスであれば数値同士の加算を、str クラスであれば文字列同士の連結を、list クラスであればリスト同士の連結を行います。共通する基底クラスから継承したものでありませんが、このように共通のインターフェース名を用いるこ方法をダックタイピングと呼びます。先ほどの例では抽象基底クラスの中で抽象メソッドを宣言していますが、もし、抽象メソッドが宣言されていなければ、ダックタイピングとなります。

付録

モンキーパッチ

動的プログラミング言語では、オリジナルのコードを変更することなく、実行時に動的にクラスやオブジェクトを拡張したり、変更する事が可能です。正式な用語ではありませんが、俗にモンキーパッチ、オープンクラス等と呼ばれているようです。テスト等での一時的な変更ややサポートの終わったサードパーティ製のスクリプトの挙動を変更させるといったような用途が多いですが、通常のプログラミングの範疇では非推奨です。

```
>>> class greeting():                    # クラスを定義。
...     def say(self):
...         print('hello!!')
...
>>> def hello(dummy=None):               # 関数を定義。
...     print('bye bye!!')
...
>>>
>>> g1 = greeting()                      # インスタンス化。
>>> g1.x                                 # 存在しないインスタンス変数を参照。
Traceback (most recent call last):       # 当然エラー。
  File "<stdin>", line 1, in <module>
AttributeError: 'greeting' object has no attribute 'x'
>>> g1.x = 999                           # 存在しないインスタンス変数に代入。
>>> g1.x                                 # 追加されました。
999
>>> g1.say()
hello!!
>>> g1.say = hello                       # メソッドを変更。
>>> g1.say()
bye bye!!
>>> greeting.say(g1)
hello!!
>>> greeting.a                           # 存在しないクラス変数を参照。
Traceback (most recent call last):       # これも当然エラー。
  File "<stdin>", line 1, in <module>
AttributeError: type object 'greeting' has no attribute 'a'
>>> greeting.a = 777                     # クラスに対しても変更可。
>>> greeting.say = hello
>>> g2 = greeting()
>>> g2.a
777
>>> g2.say()
bye bye!!
```

モンキーパッチを行えないようにするには、dataclasses モジュールを使用します。

```
>>> @from dataclasses import dataclass
>>> @dataclass(frozen=True)
... class greeting():
        <中略>
>>> g1 = greeting()
>>> g1.x = 999
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 4, in __setattr__
dataclasses.FrozenInstanceError: cannot assign to field 'x'
```

```
>>> g1.say = hello
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 4, in __setattr__
dataclasses.FrozenInstanceError: cannot assign to field 'say'
>>> g1.say()
hello!!
>>> greeting.say(g1)
hello!!
>>> greeting.say = hello
>>> g2 = greeting()
>>> g2.say()
bye bye!!
```

クラスメソッド、スタティックメソッド

インスタンスメソッドの他に、クラスメソッドとスタティックメソッドという種類のメソッドがあります。インスタンスメソッドは、インスタンスから呼び出すことを前提としていますが、クラスメソッドとスタティックメソッドはクラスから呼び出されることを前提としています。つまり、クラスメソッドとスタティックメソッドは、インスタンスが生成されていない状態でも呼び出すことが可能です。ただし、インスタンス変数を参照することはできません。（実際のところ、どのメソッドもクラスからでもインスタンスからでも呼び出すことは可能です。）インスタンスメソッドは、インスタンスへの参照を第一引数 self で受け取ります。クラスメソッドは、クラスへの参照を第一引数で受け取ります。スタティックメソッドは、何も受け取りません。クラスメソッドは @classmethod で、スタティックメソッドは @staticmethod で修飾します。

```
class MyClass():
    var_c = 'my class variable'

    def __init__(self):
        self.var_i = 'my instance variable'

    def method_i(self):
        print('instance method called!!')
        print('class variable:', self.var_c)
        print('instance variable:', self.var_i)
        print(dir(self))

    @classmethod
    def method_c(cls):
        print('class method called!!')
        print('class variable:', cls.var_c)
        #print('instance variable:', cls.var_i)         # 参照不可
        print(dir(cls))

    @staticmethod
    def method_s():
        print('static method called!!')
        print('class variable:', MyClass.var_c)
        #print('instance variable:', MyClass.var_i)     # 参照不可


>>> MyClass.method_c()              # クラスからクラスメソッドを呼び出し。
class method called!!
class variable: my class variable
```

```
['__class__', <中略>, 'method_c', 'method_i', 'method_s', 'var_c']
>>> MyClass.method_i(MyClass())    # クラスからインスタンスメソッドを呼び出し。
instance method called!!           # 何らかのオブジェクトを渡さないと、
class variable: my class variable  # 引数チェックでエラー
instance variable: my instance variable
['__class__', <中略>, 'method_i', 'method_s', 'var_c', 'var_i']
>>> MyClass.method_s()             # クラスからスタティックメソッドを呼び出し。
static method called!!
class variable: my class variable
>>>
>>> m = MyClass()
>>> m.method_c()                   # インスタンスからクラスメソッドを呼び出し。
class method called!!
class variable: my class variable
['__class__', <中略>, 'method_c', 'method_i', 'method_s', 'var_c']
>>> m.method_i()                   # インスタンスからスタティックメソッドを呼び出し。
instance method called!!
class variable: my class variable
instance variable: my instance variable
['__class__', <中略>, 'method_i', 'method_s', 'var_c', 'var_i']
>>> m.method_s()                   # インスタンスからスタティックメソッドを呼び出し。
static method called!!
class variable: my class variable
```

抽象基底クラス w/ abc

抽象基底クラスをより抽象基底クラスらしくするために、abc（Abstract Base Class）モジュールを使って抽象基底クラスを定義してみます。

```
from abc import ABCMeta, abstractmethod
class PlatonicSolid(metaclass=ABCMeta):                    # 抽象基底クラス
    @property
    def faces(self):
        return self.__faces
    @abstractmethod
    def volume(self):                          # 抽象メソッド
        raise NotImplementedError('method is not implemented yet.')
    @abstractmethod
    def surface_area(self):                    # 抽象メソッド
        raise NotImplementedError('method is not implemented yet.')
    def __init__(self, faces):
        if   faces == 4: self.__faces = 4
        elif faces == 6: self.__faces = 6
        elif faces == 8: self.__faces = 8
        elif faces == 12:self.__faces = 12
        elif faces == 20:self.__faces = 20
        else: raise ValueError('正%d面体はこの世に存在しません。' % faces)

class Tetrahedron(PlatonicSolid):
    def __init__(self):
        super().__init__(4)
    def volume(self):          # 抽象メソッド volume() をオーバーライド
        return 2**0.5 / 12
    def surface_area(self):    # 抽象メソッド surface_area() をオーバーライド
        return 3**0.5

class Cube(PlatonicSolid):
```

```
        def __init__(self):
            super().__init__(6)
        def volume(self):          # 抽象メソッド volume() をオーバーライド
            return 1

>>> ph0 = PlatonicSolid()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class PlatonicSolid with
abstract methods surface_area, volume
>>> ph1 = Tetrahedron()
>>> ph2 = Cube()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Cube with abstract
methods surface_area
```

このように abc モジュールを用いると、抽象基底クラスをインスタンス化しようとしたり、抽象メソッドをオーバーライドしていないとエラーを生成してくれます。つまり、抽象基底クラスはインスタンス化を前提としない、抽象メソッドはオーバーライドしないといけないというルールを実装レベルでルール化してくれます。複数人での開発においては、abc モジュールを用いることで開発効率の向上を期待する事ができます。

デコレーター

decorate は、修飾するという意味を持ちます。プログラミングにおいては、ある関数を修飾する関数です。シンタックス・シュガー、糖衣構文とも呼ばれます。

```
>>> def my_decorator(func):
...     def inner():
...         print('デコレートします。')
...         func()
...         print('デコレートしました。')
...     return inner
...
>>> @my_decorator
... def my_func():
...     print('関数が呼ばれました。')
...
>>> my_func()
デコレートします。
関数が呼ばれました。
デコレートしました。
```

イミュータブル・プログラミング

前出のクラスの例として、ゲッター、セッターを紹介しました。しかし、サーバーサイドプログラミング等の分野に於いては、プログラムの堅牢性や悪意あるインスタンスの改変から守るために、インスタンスに対する変更を許容しない考え方があり、これをイミュータブル・プログラミングと呼びます。例えば、python 上での int 同士の加算 1+1 は、1 という整数値を持つオブジェクト同士の加算を行い、そのインスタンスの値を変えるのではなく、新たに 2 という整数値を持つオブジェクトを返します。これもまた、イミュータブル・プログラミングといえます。