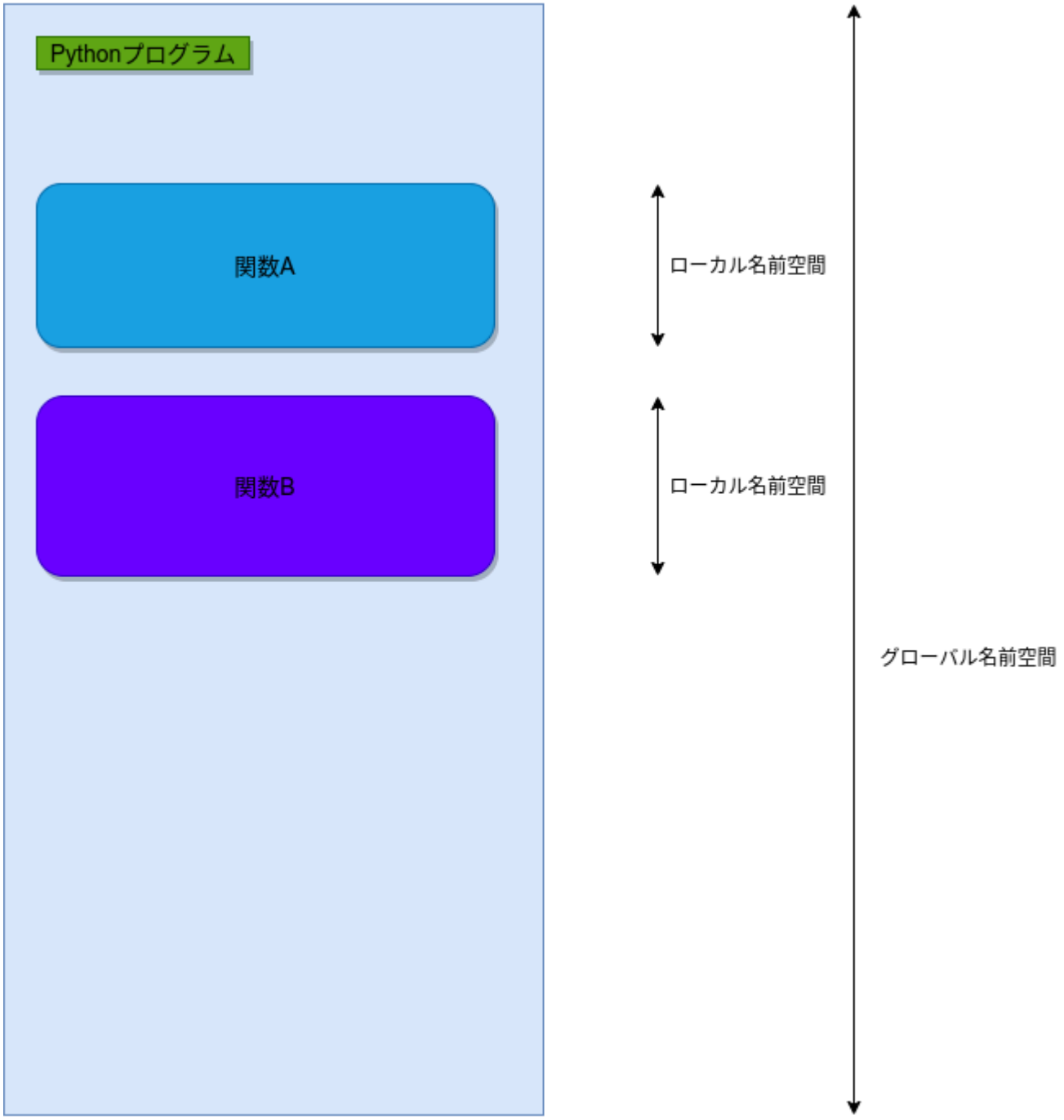
13 Python基礎

クラスの前に

- もう一度、名前空間とスコープの内容を思い出してほしい
    - 09_Python基礎.pdf



- 教科書 P.101 のプログラムを机上トレースもしくはデバッグしてください。
  【9_2_1_scope.py】

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)


scope_test()
print("In global scope:", spam)
```

- 納得できただろうか？

テキストの理解

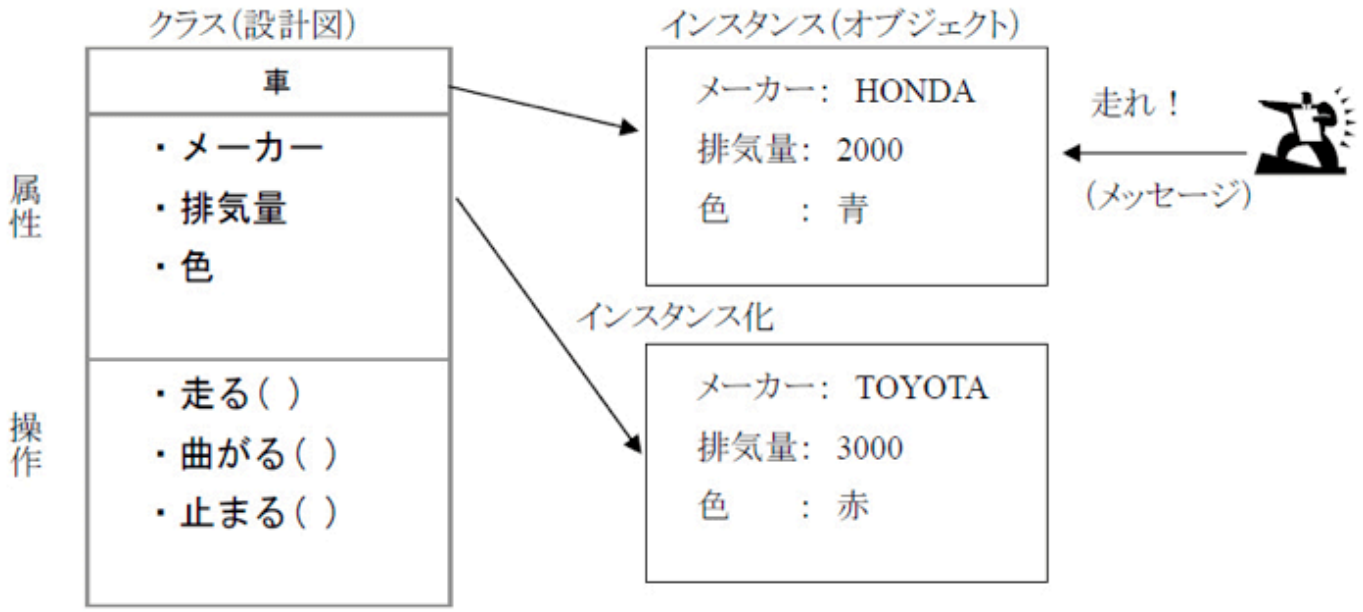
- 教科書 P.109までを、読んでください。

クラスの基本

- クラスとは？
    - データ（属性）と手続き（メソッド）を持つ、集まりの定義
- インスタンスとは？
    - クラス定義に基づいて作られたオブジェクト
- コンストラクタ（初期化メソッド）とは？
    - インスタンスを作るときに実行されるメソッド
        - C# ：クラス名と同一にする
        - Java ：クラス名と同一にする
        - C++ ：クラス名と同一にする
        - PHP ：__construct( )
        - Ruby ：initialize( )
        - Python ：__init__( )
- メソッドとは？
    - クラス内に定義された関数
        - pythonでは、メソッドの第1引数はselfと決まっている
- 属性とは？
    - 値を保持する変数
        - クラス変数
        - インスタンス変数

イメージで理解しよう

> https://eng-entrance.com/what-oop



クラスの定義

> https://docs.python.org/ja/3.9/tutorial/classes.html

- 考え方は、java , C++ , C# などの他のオブジェクト指向言語と同じ
- pythonでは以下のようにclass定義を行う

```
class クラス名:
    変数定義・メソッドなどの内容
```

最も単純なクラス

```
class MyClass:
    pass
```

- 実際の定義を行わず、後から作成する場合に良く使用する
    - とりあえず名前だけを持つクラス

【9_3_2_myclass.py】

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

メソッド定義

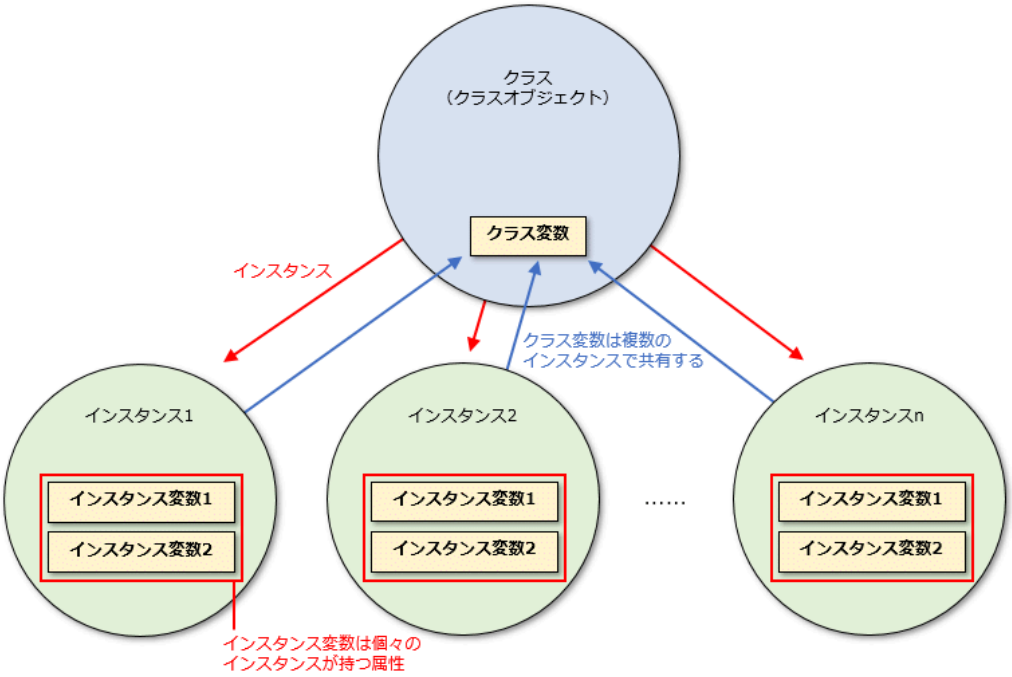
- クラス内に定義された関数
    - 第1引数はselfと決まっている

インスタンスを生成してみよう

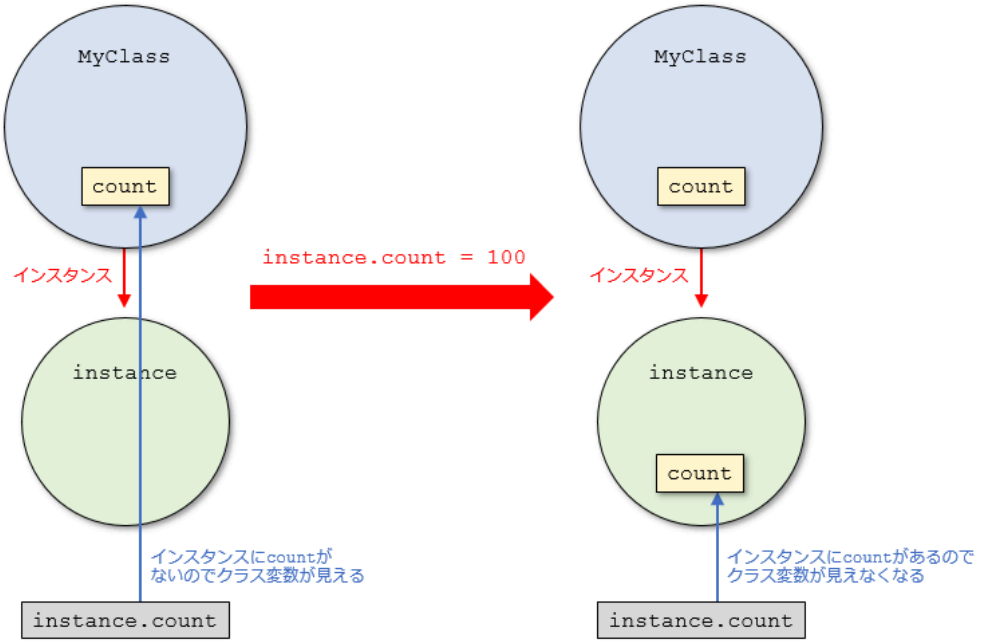
- 上記、9_3_2_myclass.py を修正し、mainを作成しよう
    - インスタンスを作成し、変数 `x` に代入
    - クラス内の `i` を出力する
    - クラス内メソッドを呼び出し、結果を出力する
- コンストラクタを作成する
    - コンストラクタ内で、インスタンス変数リスト `ls` を初期化する（リストは[1,2,3]とする）
    - main側で、lsを出力する

クラス変数とインスタンス変数

> https://www.atmarkit.co.jp/ait/articles/1907/30/news021.html



- Pythonでは、実行時に変数を生成する（できる）ため、同名の変数の場合に、おかしなことが起きやすい。
    - クラス変数のつもりが、インスタンス変数になる場合。



クラス変数のつもり

- クラス変数を使っているつもりだが、実際にはインスタンス変数を操作している

【class_tri.py】

```
class TridentStudent:
    class_name = ''
    num = 0
    name = ''
    gender = ''

# main
p = TridentStudent()
p.class_name = 'NT1'
p.num = 10
p.name = '虎井 太郎'
p.gender = '男'

print(p.__dict__) # Instanceの持つ変数の一覧を確認
```

【実行結果】

```
{'class_name': 'NT1', 'num': 10, 'name': '虎井 太郎', 'gender': '男'}
```

- 実際には、p.class_name = ... 以降（9〜12行目）は、クラス変数と同名のインスタンス変数を定義して代入している

検証1

【class_tri_1.py】

```
from pprint import pprint


class TridentStudent:
    '''Trident学生管理クラス'''
    class_name = ''
    num = 0
    name = ''
    gender = ''


# main
p = TridentStudent()
p.class_name = 'NT1'
p.num = 10
p.name = '虎井 太郎'
p.gender = '男'

pprint(p.__dict__)
pprint(TridentStudent.__dict__)
```

【実行結果】

```
{'class_name': 'NT1', 'gender': '男', 'name': '虎井 太郎', 'num': 10}
mappingproxy({'__dict__': <attribute '__dict__' of 'TridentStudent' objects>,
              '__doc__': 'Trident学生管理クラス',
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'TridentStudent' objects>,
              'class_name': '',
              'gender': '',
              'name': '',
              'num': 0})
```

- 2行目〜を見ると、クラスに定義されているクラス変数は、一切変わっていないのが分かる

検証2

【class_tri_2.py】

```
from pprint import pprint


class TridentStudent:
    '''Trident学生管理クラス'''
    class_name = ''
    num = 0
    name = ''
    gender = ''


# main
p = TridentStudent()
p.class_name = 'NT1'
p.num = 10
p.name = '虎井 太郎'
p.gender = '男'

q = TridentStudent()
q.class_name = 'ST2'
q.num = 30
q.name = '河合 花子'
q.gender = '女'

pprint(q.__dict__)
pprint(p.__dict__)
```

- もし、代入しているのが、クラス変数であれば、上記のコードを実行した場合、最後の出力（print）は同一の結果にならないとおかしい

【実行結果】

```
{'class_name': 'ST2', 'num': 30, 'name': '河合 花子', 'gender': '女'}
{'class_name': 'NT1', 'num': 10, 'name': '虎井 太郎', 'gender': '男'}
```

検証3

【class_tri_3.py】

```
from pprint import pprint


class TridentStudent:
    pass


# main
p = TridentStudent()
p.class_name = 'NT1'
p.num = 10
p.name = '虎井 太郎'
p.gender = '男'

pprint(p.__dict__)
```

- クラス変数を持たないclassを定義し、代入してみる

【実行結果】

```
{'class_name': 'NT1', 'num': 10, 'name': '虎井 太郎', 'gender': '男'}
```

- 最初と同じ結果であることが分かる。
- `p.class_name = ...` は、オブジェクトpの内部に、インスタンス変数を定義し代入していた

正しいクラス変数の使用法を考える

【class_test_1.py】

- クラス変数の動作を確認する

```
class Test:
    c_var = 10

#1
ins = Test()
print(ins.c_var)
print(ins.__dict__)
print(Test.__dict__)
```

【実行結果】

```
10
{}
mappingproxy({'__dict__': <attribute '__dict__' of 'Test' objects>,
              '__doc__': None,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Test' objects>,
              'c_var': 10})
```

- 1行目：`ins.c_var` は10
    - `ins.c_var` はクラス内定義の変数を出力している
- 2行目：`ins` オブジェクトに、変数は存在していない

【class_test_2.py】

- インスタンス変数を追加、クラス変数を変更してみる

```
from pprint import pprint


class Test:
    c_var = 10


# 1
ins = Test()
pprint(ins.__dict__)
pprint(Test.__dict__)
print('-=' * 30)

# 2
ins.c_var = 100
pprint(ins.__dict__)
pprint(Test.__dict__)

Test.c_var = 20
pprint(ins.__dict__)
```

```
pprint(Test.__dict__)
```

【実行結果】

```
{}
mappingproxy({'__dict__': <attribute '__dict__' of 'Test' objects>,
              '__doc__': None,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Test' objects>,
              'c_var': 10})
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
{'c_var': 100}
mappingproxy({'__dict__': <attribute '__dict__' of 'Test' objects>,
              '__doc__': None,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Test' objects>,
              'c_var': 10})
{'c_var': 100}
mappingproxy({'__dict__': <attribute '__dict__' of 'Test' objects>,
              '__doc__': None,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Test' objects>,
              'c_var': 20})
```

【class_test_3.py】

```
from pprint import pprint


class Test:
    c_var = 10

    def func1(self):
        c_var = 30

    def func2(self):
        self.c_var = 40

    def func3(self):
        Test.c_var = 50


# 1
ins = Test()
pprint(ins.__dict__)
pprint(Test.__dict__)
print('-=' * 30)

# 2   以下 c_varのみ出力する
```

```python
ins.c_var = 100
print(ins.__dict__['c_var'])
print(Test.__dict__['c_var'])

Test.c_var = 20
print(ins.__dict__['c_var'])
print(Test.__dict__['c_var'])
print('-=' * 30)

# 3
ins.func1()
print(ins.__dict__['c_var'])
print(Test.__dict__['c_var'])

ins.func2()
print(ins.__dict__['c_var'])
print(Test.__dict__['c_var'])

ins.func3()
print(ins.__dict__['c_var'])
print(Test.__dict__['c_var'])
```

【実行結果】

```
{}
mappingproxy({'__dict__': <attribute '__dict__' of 'Test' objects>,
              '__doc__': None,
              '__module__': '__main__',
              '__weakref__': <attribute '__weakref__' of 'Test' objects>,
              'c_var': 10,
              'func1': <function Test.func1 at 0x7faab3cb4280>,
              'func2': <function Test.func2 at 0x7faab3c409d0>,
              'func3': <function Test.func3 at 0x7faab3c40a60>})
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
100
10
100
20
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
100
20
40
20
40
50
```

クラス変数を正しく使うには

- 上記の結果から、クラス変数とインスタンス変数を正しく使えるように書きたい

【class_sample.py】

```
class ClassName:
    # クラス変数
    class_var = 0

    def __init__(self):
        # インスタンス変数の操作
        self.var = 1

    def func0(self, x):
        '''インスタンス変数へ追加'''
        self.var = self.var + x

    def func1(self):
        '''クラス変数へ追加'''
        # クラス変数の操作
        ClassName.class_var += 10        # できれば避けたい使い方(クラス名の変更時に問題有り)
        type(self).class_var += 100
        self.__class__.class_var += 1000


if __name__ == "__main__":
    obj = ClassName()
    print(obj.var)

    obj.var = 10
    print(obj.var)
    obj.func0(3)
    print(obj.var)

    print('-=' * 30)

    print(obj.class_var)
    obj.func1()
    print(obj.class_var)        # インスタンス変数class_varが存在しないので、クラス変数が参照される
    print(ClassName.class_var)
```

【実行結果】

```
1
10
13
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
0
1110
1110
```

インスタンス変数とクラス変数の違い

> https://www.atmarkit.co.jp/ait/articles/1907/30/news021.html

変数の種類	説明	クラス外部からのアクセス方法	クラス内でのアクセス方法
インスタンス変数	インスタンスが持つデータを保存	インスタンス.インスタンス変数	self.インスタンス変数
クラス変数	クラスが持つデータや複数のインスタンスで共有するデータを保存	クラス.クラス変数 インスタンス.クラス変数（参照）	self.クラス変数（参照） クラス.クラス変数 self.__class__.クラス変数 type(self).クラス変数

カプセル化を行なうために

変数・メソッドの隠蔽

- 変数の先頭にアンダースコア `_` をつける
  - 紳士協定のため、無視して操作できてしまう
- 変数の先頭にアンダースコアを2個 `__` つける
  - ネームマングリング（Name Mangling）といい、名前修飾を行なう
    - これも修飾後の名前を使用すると、アクセスできてしまうが、より安全

> https://qiita.com/mounntainn/items/e3fb1a5757c9cf7ded63 修正版

```
class HogeHoge:
    _dummy_private = "hoge"
    __almost_private = "hoge"


hogehoge = HogeHoge()

# 丸見え
a = hogehoge._dummy_private
print(a)

# エラー（確認後コメントアウト）
# b = hogehoge.__almost_private
# print(b)
```

```
# こうすると見えちゃう
c = hogehoge._HogeHoge__almost_private
print(c)
```

ゲッター・セッターとプロパティ

- 変数名や関数が隠蔽されただけでは、カプセル化とは言えない。
  - ゲッター・セッターを利用できるようにする
  - ゲッター・セッターをプロパティにする

【class_getset.py】

```
class TestGetSet:
    def __init__(self, val):
        self.__val = val

    def get_val(self):
        return self.__val

    def set_val(self, val):
        self.__val = val

class TestProperty:
    def __init__(self, val):
        self.__val = val

    @property
    def val(self):
        return self.__val

    @val.setter
    def val(self, val):
        self.__val = val

# main
obj1 = TestGetSet(100)
print(obj1.get_val())
obj1.set_val(20)
print(obj1.get_val())

obj2 = TestProperty(100)
print(obj2.val)
obj2.val = 20
print(obj2.val)
```

【実行結果】

```
> python class_getset.py
100
20
100
20
```

- ゲッター、セッターは、目的とする変数へのアクセス方法を用意したメソッド
- プロパティは、変数への代入・読み出しと同じ使い方を実現する方法

プロパティ	**役割**
@property	getter
@属性名.setter	setter
@属性名.deleter	deleter

練習1

- 以下のプログラムを作成してください。【class_book_getset.py】
    - クラスBookを作成する。
        - getter / setter を作成し使用する
        - クラス変数として、税率（%）を持っている。
        - 本ごと（インスタンスごと）に本体価格を持っている。
        - 値引率（0-100%）を本ごと（インスタンスごと）に持つが、初期値は0とする
        - 販売価格を計算するメソッドprice( )を持つ
            - 販売価格は、本体価格 - 値引き額 で求める。その後税率を適用する。
        - 異常値は全て、例外Errorを発生させ終了する（raise を使用する）
            - 本体価格がマイナスの場合
            - 値引率が0〜100の範囲を超える場合

> raise で例外エラーを発生させでも良い
>
> ```
> raise ValueError("入力した値が誤っています")
> ```

【実行結果1】

```
Traceback (most recent call last):
  File "/home/yoshimura/src/13/class_book_getset.py", line 26, in <module>
    b0 = Book(-100)  # 動作テスト用コード
  File "/home/yoshimura/src/13/class_book_getset.py", line 7, in __init__
    raise ValueError("価格は0円以上で設定してください")
ValueError: 価格は0円以上で設定してください
```

【実行結果2】

```
現在の値引率：0%
現在の値引率：10%
販売価格：990
現在の値引率：0%
Traceback (most recent call last):
  File "/home/yoshimura/src/13/class_book_getset.py", line 36, in <module>
    b2.set_discounts(-10)   # 動作テスト用コード
  File "/home/yoshimura/src/13/class_book_getset.py", line 18, in set_discounts
    raise ValueError("値引率は0-100で設定してください")
ValueError: 値引率は0-100で設定してください
```

【class_book_getset.py】

```
class Book:
    '''書籍の価格を管理するクラス'''









# main
b0 = Book(-100)  # 動作テスト用

b1 = Book(1000)
print(f'現在の値引率：{b1.get_discounts()}%')
b1.set_discounts(10)
print(f'現在の値引率：{b1.get_discounts()}%')
print(f'販売価格：{b1.price()}')

b2 = Book(2000)
print(f'現在の値引率：{b2.get_discounts()}%')
b2.set_discounts(-10)   # 動作テスト用
print(f'現在の値引率：{b2.get_discounts()}%')
print(f'販売価格：{b2.price()}')
```

練習2

- 上記と同様な動作をするプログラム、【class_book_property.py】 を作成する
    - getter / setter ではなく、プロパティとして作成する。
    - 先の実行結果と同様であることを確認する

【class_book_property.py】

```
class Book:
```

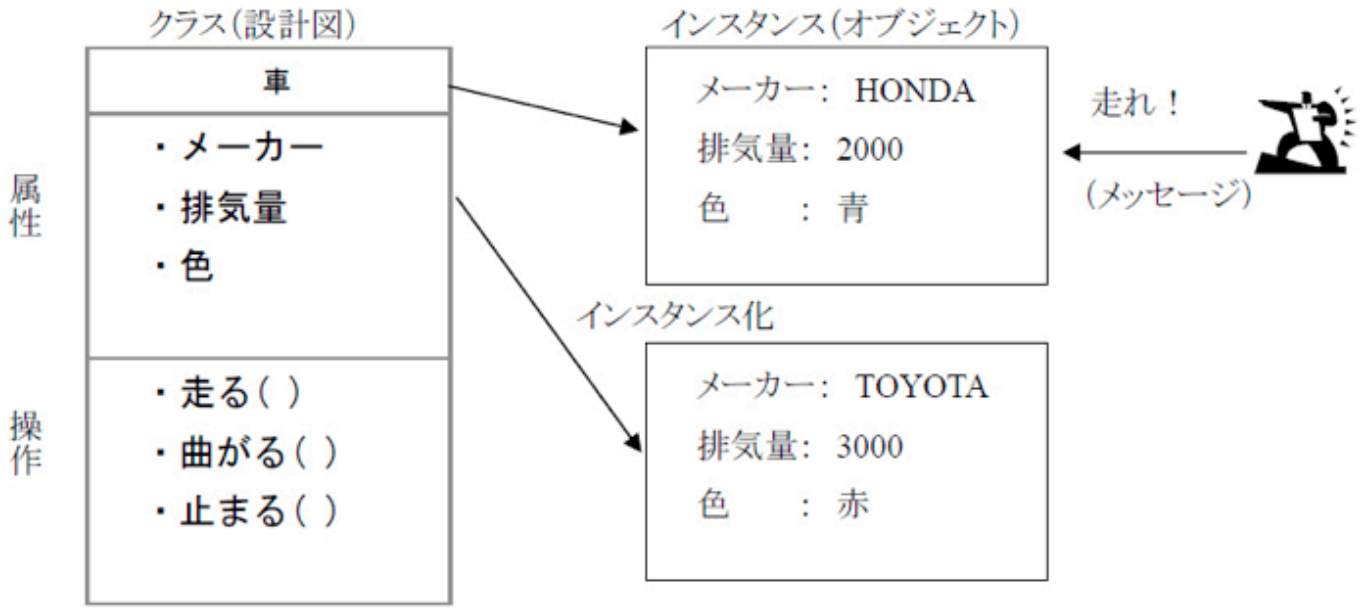
```
#main
b1 = Book(1000)
print(f'現在の値引率：{b1.discounts}%')
b1.discounts = 10
print(f'現在の値引率：{b1.discounts}%')
print(f'販売価格：{b1.price()}')

b2 = Book(2000)
print(f'現在の値引率：{b1.discounts}%')
b1.discounts = -10
print(f'現在の値引率：{b1.discounts}%')
print(f'販売価格：{b1.price()}')
```

【実行結果】

```
現在の値引率：0%
現在の値引率：10%
販売価格：990
現在の値引率：0%
Traceback (most recent call last):
  File "/home/yoshimura/src/13/class_book_property.py", line 37, in <module>
    b2.discounts = -10
  File "/home/yoshimura/src/13/class_book_property.py", line 19, in discounts
    raise ValueError("値引率は0-100で設定してください")
ValueError: 値引率は0-100で設定してください
```

練習3



- 図を元に、プログラムを作成してください。【class_car.py】
  - クラス名Car
    - 変数：メーカー、排気量、色
    - メソッド：走る、曲がる、止まる
      - 走る→引数：走行km
        - 自分の走行距離の合計を覚えている。
        - 走行した距離を表示する。
      - 曲がる→引数：右、左
        - 曲がった回数を覚えている。
        - 曲がった方向を表示する
      - 止まる→引数：なし
        - 止まったことを表示する。
      - その他必要なメソッドを定義して利用しても良い
  - getter / setter もしくはプロパティを用意し使用しても良い
  - メインにて
    - 車を2台作る。
      - car1：HONDA、2000cc、青

    - car2：TOYOTA、3000cc、赤
- 車をそれぞれ、適当な距離を走らせる。何回か曲がる。最後に止まる
    - 止まった後、全走行距離、曲がった回数を車ごとに表示する。

実行例】

```
=== car1を走らせる ===
40km走りました
右に曲がりました
30km走りました
左に曲がりました
20km走りました
止まりました。
=== car2を走らせる ===
30km走りました
10km走りました
右に曲がりました
左に曲がりました
40km走りました
右に曲がりました
止まりました。
=== 走行結果 ===
【車種情報】
メーカー名:HONDA
排気量:2000
色:Blue
全部で90km走りました。
全部で2回曲がりました
【車種情報】
メーカー名:TOYOTA
排気量:3000
色:Red
全部で80km走りました。
全部で3回曲がりました
```