08-Python面向对象编程

大纲

- 创建和使用类
- 继承与组合
- 重写(override)类的方法
- Python数据模型
- Python类的特殊方法

创建和使用类

- 使用class关键字创建类
- 类名使用驼峰命名法，每个单词的首字母大写，例如：MyClass、BattleShip
- 但文件名使用小写: dog.py, model.py

In [1]:

```
class Dog:
    """Emulate a dog"""

    # 构造函数
    def __init__(self, name, age):
        """Initialize name and age attributes"""
        self.name = name
        self.age = age

    def sit(self):
        """Simulate a dog sitting in response to a command"""
        print(f"{self.name} is now sitting.")

    def roll_over(self):
        """Simulate a dog rolling over in response to a command"""
        print(f"{self.name} rolled over!")

my_dog = Dog('Willie', 4)  # 调用了__init__方法

# Python的属性和方法都不是私有的
print(my_dog.name, my_dog.age)
my_dog.sit()
my_dog.roll_over()
```

```
Willie 4
Willie is now sitting.
Willie rolled over!
```

类的定义

- 类中的定义函数被称为方法(method)，类中定义的变量被称为属性（attribute）
- __init__这样以下划线开始和结束的方法或者属性都是特殊方法或者属性，特殊方法和属性通常被隐式地使用-
- 这里的方法的第一个参数都是 `self` ， `self` 表示自己这个对象，通过 `self` 可以使用该对象的属性和方法

Car类的定义

Car类的定义

- 属性的默认值
- 汽车里程应该只能增加，不能减少

In [2]:
```
class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0  # 设置Car对象的里程属性默认为0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    def read_odometer(self):
        """Print a statement showing the car's mileage."""
        print(f"This car has {self.odometer_reading} miles on it.")

    def update_odometer(self, mileage):
        """
        Set the odometer reading to the given value.
        Reject the change if it attempts to roll the odometer back.
        """
        if mileage >= self.odometer_reading:
            self.odometer_reading = mileage
        else:
            print("You can't roll back an odometer!")

    def increment_odometer(self, miles):
        """Add the given amount to the odometer reading."""
        self.odometer_reading += miles

my_used_car = Car('subaru', 'outback', 2015)
print(my_used_car.get_descriptive_name())

my_used_car.update_odometer(23_500)
my_used_car.read_odometer()

my_used_car.increment_odometer(100)
my_used_car.read_odometer()

my_used_car.update_odometer(2000)
my_used_car.read_odometer()
```

```
2015 Subaru Outback
This car has 23500 miles on it.
This car has 23600 miles on it.
You can't roll back an odometer!
This car has 23600 miles on it.
```

In [3]:
```
# 无法阻止直接修改属性值
my_used_car.odometer_reading = 0
my_used_car.read_odometer()
```

```
This car has 0 miles on it.
```

继承

子类将自动获得另一个类（父类）的所有属性和方法，子类还可以定义自己的属性和方法。

In [4]:

```
class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """

        # 调用父类的构造函数
        super().__init__(make, model, year)

        # 电动汽车子类增加的属性，默认值是75
        self.battery_size = 75

    # 子类增加的新的方法
    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

my_tesla = ElectricCar('tesla', 'model s', 2019)

# 调用父类继承来的方法
print(my_tesla.get_descriptive_name())

# 调用子类定义的方法
my_tesla.describe_battery()
```

```
2019 Tesla Model S
This car has a 75-kWh battery.
```

组合类

- 组合类，类的属性可以是自定义类的对象
- 组合相比继承更常用

In [5]:

```
class Battery:
    """A simple attempt to model a battery for an electric car."""

    def __init__(self, battery_size=75):
        """Initialize the battery's attributes."""
        self.battery_size = battery_size

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

    def get_range(self):
        """Print a statement about the range this battery provides."""
        if self.battery_size == 75:
            range = 260
        elif self.battery_size == 100:
            range = 315

        print(f"This car can go about {range} miles on a full charge.")


class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
```

```
        """
        super().__init__(make, model, year)

        # battery属性是一个组合的对象
        self.battery = Battery()

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.battery.describe_battery()
my_tesla.battery.get_range()
```

```
2019 Tesla Model S
This car has a 75-kWh battery.
This car can go about 260 miles on a full charge.
```

重写（override）父类的方法

子类和父类完全一样的方法（相同的函数名，相同的参数列表），但是会表现出不同的行为，这就是重写（override）。

In [28]:

```
class Car:
    """A simple attempt to represent a car."""

    def __init__(self, make, model, year):
        """Initialize attributes to describe a car."""
        self.make = make
        self.model = model
        self.year = year
        self.odometer_reading = 0

    def get_descriptive_name(self):
        """Return a neatly formatted descriptive name."""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name.title()

    # 这个方法会被电动汽车子类继承
    def fill_gas_tank(self):
        """Fill gas tank"""
        print("Fill gas tank")

class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, make, model, year):
        """
        Initialize attributes of the parent class.
        Then initialize attributes specific to an electric car.
        """
        super().__init__(make, model, year)
        self.battery = Battery()

    def describe_battery(self):
        """Print a statement describing the battery size."""
        print(f"This car has a {self.battery_size}-kWh battery.")

    # 改写（override）父类继承来的方法
    def fill_gas_tank(self):
        """Electric car doesn't need a gas tank"""
        print("This car doesn't need a gas tank!")

my_honda = Car('Honda', 'Accord', 2016)
my_honda.fill_gas_tank()
```

```
my_tesla = ElectricCar('tesla', 'model s', 2019)
my_tesla.fill_gas_tank()

# 给对象加上属性
my_tesla.price = 20000
print(my_tesla.price)
```

```
Fill gas tank
This car doesn't need a gas tank!
20000
```

关于面向对象编程

- 面向对象编程很容易导致糟糕的设计。
- "a function is all you need" 函数式编程是现代编程语言的趋势。例如：Go语言没有class关键字。
- 应该尽可能避免使用继承，尽量使用组合。

练习：面向对象的海盗

难度： 8kyu

啊哈，伙计!

你是一个小海盗团的首领。而且你有一个计划。在OOP的帮助下，你希望建立一个相当有效的系统来识别船上有大量战利品的船只。 对你来说，不幸的是，现在的人很重，那么你怎么知道一艘船上装的是黄金而不是人呢?

你首先要写一个通用的船舶类。

```
class Ship:
    def __init__(self, draft, crew):
        self.draft = draft
        self.crew = crew
```

每当你的间谍看到一艘新船进入码头，他们将根据观察结果创建一个新的船舶对象。

- `draft` 吃水 - 根据船在水中的高度来估计它的重量
- `crew` 船员 - 船上船员的数量

```
Titanic = Ship(15, 10)
```

任务

你可以访问船舶的 "draft(吃水) "和 "crew(船员)"。"draft(吃水) "是船的总重量，"船员 "是船上的人数。 每个船员都会给船的吃水增加1.5个单位。如果除去船员的重量后，吃水仍然超过20，那么这艘船就值得掠夺。任何有这么重的船一定有很多战利品! 添加方法 `is_worth_it` 来决定这艘船是否值得掠夺。

例如：

```
Titanic.is_worth_it()
False
```

祝你好运，愿你能找到金子!

代码提交地址： https://www.codewars.com/kata/54fe05c4762e2e3047000add

Random类

- Random类产生的伪随机数，当种子确定时，产生的随机数是确定的。
- randint方法: 产生随机整数
- choice:随机选择一个
- sample:随机选择若干个样本
- shuffle: 洗牌

In [20]:
```
import random

random.seed(0)  # 确定了种子以后，产生的随机序列是确定的
print(random.randint(1, 6))
print(random.choice(['apple', 'pear', 'banana']))
print(random.sample(range(100), 10))
```

```
4
pear
[5, 33, 65, 62, 51, 38, 61, 45, 74, 27]
```

In [19]:
```
lst1 =  [1, 2, 3, 4, 5, 6]
random.shuffle(lst1)
lst1
```

Out[19]: [2, 5, 4, 1, 3, 6]

Python数据模型

> Python的最好的特性之一就是：一致性。使用Python一段时间之后，便可以根据自己掌握的知识，正确地猜出新功能的作用。
>
> ---《流畅的Python》

Python风格的代码（Pythonic Codes）：

- 使用 len 函数获取集合对象的长度
- 使用 [] 获取集合对象的元素
- 使用切片语法获取集合对象的子集
- ...

Python风格的扑克牌

In [1]:
```
import collections

# 有名字的元组
Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        """
```

```
        特殊方法:当对象作为参数放入len()函数时，这个方法会被调用
        """
        return len(self._cards)

    def __getitem__(self, position):
        """
        特殊方法:当对象使用[]操作符时，这个方法会被调用
        """
        return self._cards[position]
```

In [3]:
```
beer_card = Card('7', 'diamonds')
beer_card
```

Out[3]:
```
Card(rank='7', suit='diamonds')
```

> In trick-taking card games such as bridge, the beer card is a name informally given to the seven of diamonds (7♦). Players may agree that if a player wins the last trick of a hand with the 7♦, their partner must buy them a beer. This is not considered as part of the rules of these games, but is an optional and informal side-bet between players. This practice likely originates from Danish Tarok or Skat in the middle of the 20th century.[1] In most decks, the 7♦ is the only diamond number card that lacks rotational symmetry. --Wikipedia

In [4]:
```
deck = FrenchDeck()
len(deck)  # 扑克牌对象的__len__方法自动被调用
```

Out[4]:
```
52
```

In [7]:
```
# 使用索引访问扑克牌对象
deck[0]
```

Out[7]:
```
Card(rank='2', suit='spades')
```

In [6]:
```
deck[-1]
```

Out[6]:
```
Card(rank='A', suit='hearts')
```

In [8]:
```
# 使用切片
deck[:3]
```

Out[8]:
```
[Card(rank='2', suit='spades'),
 Card(rank='3', suit='spades'),
 Card(rank='4', suit='spades')]
```

In [9]:
```
deck[12::13]
```

Out[9]:
```
[Card(rank='A', suit='spades'),
 Card(rank='A', suit='diamonds'),
 Card(rank='A', suit='clubs'),
 Card(rank='A', suit='hearts')]
```

In [14]:
```
# 随机选择一张牌
from random import choice

choice(deck)
```

Out[14]:
```
Card(rank='10', suit='hearts')
```

In [ ]:
```
# 迭代访问
for card in deck:
    print(card)
```

In [ ]:
```
# 反向迭代
for card in reversed(deck):
    print(card)
```

如何洗牌?

- 按照目前的设计， FrenchDeck 类的对象是不可变的，因此不能直接洗牌。
- 只需要添加一个 __setitem__ 方法，就可以洗牌了。

In [18]:
```
import collections

# 有名字的元组
Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        """
        特殊方法:当对象作为参数放入len()函数时，这个方法会被调用
        """
        return len(self._cards)

    def __getitem__(self, position):
        """
        特殊方法:当对象使用[]操作符时，这个方法会被调用
        """
        return self._cards[position]

    def __setitem__(self, position, card):
        """
        特殊方法:当对象使用[]操作符赋值时，这个方法会被调用
        """
        self._cards[position] = card
```

In [24]:
```
from random import shuffle

deck = FrenchDeck()
shuffle(deck)
deck[:5]
```

Out[24]:
```
[Card(rank='7', suit='diamonds'),
 Card(rank='9', suit='diamonds'),
 Card(rank='3', suit='hearts'),
 Card(rank='Q', suit='clubs'),
 Card(rank='Q', suit='spades')]
```

可以动态地给对象添加属性和方法，开始时， FrenchDeck 类的对象是不可变的，不能洗牌。

In [7]:

In [7]:
```
import collections

# 有名字的元组
Card = collections.namedtuple('Card', ['rank', 'suit'])

class FrenchDeck:
    ranks = [str(n) for n in range(2, 11)] + list('JQKA')
    suits = 'spades diamonds clubs hearts'.split()

    def __init__(self):
        self._cards = [Card(rank, suit) for suit in self.suits
                                        for rank in self.ranks]

    def __len__(self):
        """
        特殊方法:当对象作为参数放入len()函数时，这个方法会被调用
        """
        return len(self._cards)

    def __getitem__(self, position):
        """
        特殊方法:当对象使用[]操作符时，这个方法会被调用
        """
        return self._cards[position]
```

In [10]:
```
def set_card(deck, position, card):
    deck._cards[position] = card

# 给FrenchDeck类动态地添加一个__setitem__方法
FrenchDeck.__setitem__ = set_card

deck = FrenchDeck()

from random import shuffle
shuffle(deck)
deck[:5]
```

Out[10]:
```
[Card(rank='J', suit='diamonds'),
 Card(rank='3', suit='clubs'),
 Card(rank='9', suit='clubs'),
 Card(rank='8', suit='spades'),
 Card(rank='6', suit='diamonds')]
```

Python类的一些特殊方法

特殊方法供 Python 解释器调用，而不是显示调用

- `__str__`
- `__repr__`
- `__eq__`
- `__iter__`
- `__len__`
- `__getitem__`
- `__abs__`

In [21]:
```
class Vector:

    # _components是特殊属性，存放表示向量的列表
    def __init__(self, components):
        self._components = tuple(components)

    # 遍历向量包含的元组
    def __iter__(self):
```

```
        return iter(self._components)

    # 把向量包含的元组转换为字符串打印
    def __str__(self):
        return str(self._components)

    # 把向量包含的元组表示出来
    def __repr__(self):
        return f'Vector({self._components})'

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

    def __hash__(self):
        return hash(self._components)

    def __len__(self):
        return len(self._components)

    def __getitem__(self, index):
        return self._components[index]

    def __abs__(self):
        return (sum(x**2 for x in self))**0.5

    def __add__ (self, other):
        return Vector(x + y for x, y in zip(self, other))

    # 当对象出现在+号右边时，调用__radd__方法
    def __radd__(self, other):
        return self + other

v1 = Vector([1, 2, 3])
v3 = Vector([3, 4])
```

In [22]:
```
print(v1)  #  __str__ called，如果没有编写该方法，会打印该对象的id
```

```
(1, 2, 3)
```

In [23]:
```
v1 # __repr__ called
```

Out[23]:
```
Vector((1, 2, 3))
```

In [24]:
```
v2 = Vector([1, 2, 3])
v1 == v2 # __eq__ called，默认是比较两个对象的id
```

Out[24]:
```
True
```

In [66]:
```
print(hash(v1)) # __hash__ called
print({v1, v2})
```

```
529344067295497451
{Vector((1, 2, 3))}
```

In [25]:
```
# 向量可以被遍历
for value in v1:  # __iter__ called
    print(value)
```

```
1
2
3
```

In [26]:
```
len(v1)  # __len__ called
```

Out[26]: 3

In [27]:
```
print(v1[0]) # __getitem__ called
print(v1[1])
print(v1[2])
print(v1[1:]) # slice is supported
```

```
1
2
3
(2, 3)
```

In [28]:
```
abs(v3) # __abs__ called
```

Out[28]: 5.0

In [29]:
```
print(v1, v2)
v1 + v2 # __add__ called
```

```
(1, 2, 3) (1, 2, 3)
```

Out[29]: Vector((2, 4, 6))

操作符重载

Common Syntax	Special Method Form	
a + b	a.__add__(b);	alternatively b.__radd__(a)
a − b	a.__sub__(b);	alternatively b.__rsub__(a)
a * b	a.__mul__(b);	alternatively b.__rmul__(a)
a / b	a.__truediv__(b);	alternatively b.__rtruediv__(a)
a // b	a.__floordiv__(b);	alternatively b.__rfloordiv__(a)
a % b	a.__mod__(b);	alternatively b.__rmod__(a)
a ** b	a.__pow__(b);	alternatively b.__rpow__(a)
a << b	a.__lshift__(b);	alternatively b.__rlshift__(a)
a >> b	a.__rshift__(b);	alternatively b.__rrshift__(a)
a & b	a.__and__(b);	alternatively b.__rand__(a)
a ^ b	a.__xor__(b);	alternatively b.__rxor__(a)
a | b	a.__or__(b);	alternatively b.__ror__(a)
a += b a −= b a *= b ...	a.__iadd__(b) a.__isub__(b) a.__imul__(b) ...	
+a	a.__pos__()	
−a	a.__neg__()	
~a	a.__invert__()	
abs(a)	a.__abs__()	
a < b	a.__lt__(b)	
a <= b	a.__le__(b)	
a > b	a.__gt__(b)	
a >= b	a.__ge__(b)	
a == b	a.__eq__(b)	
a != b	a.__ne__(b)	

In [ ]:
```
v1 + v2 # __add__ called
```

In [ ]:
```
v1 += v2 # __add__ called
v1
```

In [ ]:
```
Vector([1, 2]) + Vector([1, 2, 3]) # TypeError: dimensions must agree
```

namedtuple

- 类似于元组（tuple）对象，也是不可变的
- 但是它的数据域是由像字典一样有名字的。
- 自动具备了 `__str__`，`__repr__`，`__eq__`，`__hash__` 等方法

In [89]:
```
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])
```

In [90]:
```
p1 = Point(4, 6)
p2 = Point(4, 6)
print(p1)  #  __str__ called
p1   # __repr__ called
```

```
Point(x=4, y=6)
```

Out[90]:
```
Point(x=4, y=6)
```

In [91]:
```
p1 == p2 # __eq__ called
```

Out[91]:
```
True
```

In [93]:
```
points = { p1, p2, Point(3, 4) } # __hash__ called
points
```

Out[93]:
```
{Point(x=3, y=4), Point(x=4, y=6)}
```

In [95]:
```
print(p1.x, p1.y)
p1.x = 10 # AttributeError: can't set attribute
```

```
4 6
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
c:\Users\zhouj\workspace\python_course\src\08-classes\08-classes copy.ipynb
Cell 36 line 2
      <a href='vscode-notebook-cell:/c%3A/Users/zhouj/workspace/python_course/src/08-cl
asses/08-classes%20copy.ipynb#X66sZmlsZQ%3D%3D?line=0'>1</a> print(p1.x, p1.y)
----> <a href='vscode-notebook-cell:/c%3A/Users/zhouj/workspace/python_cours
e/src/08-classes/08-classes%20copy.ipynb#X66sZmlsZQ%3D%3D?line=1'>2</a> p1.x
= 10 # AttributeError: can't set attribute

AttributeError: can't set attribute
```

DataClass

- Python 3.7 新增的特性
- 在类的定义前面使用 @dataclass

- 在类的定义前面使用 @dataclass
- 自动具备了 __str__，__repr__，__eq__，__hash__ 等方法
- 相比 namedtuple，dataclass 的数据域可以写，可以添加新的方法

In [101…

```
from dataclasses import dataclass
from math import asin, cos, radians, sin, sqrt

@dataclass
class Position:
    name: str
    lon: float
    lat: float

    def distance_to(self, other):
        r = 6371  # Earth radius in kilometers
        lam_1, lam_2 = radians(self.lon), radians(other.lon)
        phi_1, phi_2 = radians(self.lat), radians(other.lat)
        h = (sin((phi_2 - phi_1) / 2)**2
             + cos(phi_1) * cos(phi_2) * sin((lam_2 - lam_1) / 2)**2)
        return 2 * r * asin(sqrt(h))
```

In [102…

```
pos = Position('Oslo', 10.8, 59.9)
print(pos)
vancouver = Position('Vancouver', -123.1, 49.3)
print(pos.distance_to(vancouver))
```

```
Position(name='Oslo', lon=10.8, lat=59.9)
7181.784122942117
```

In [103…

```
print(f'{pos.name} is at {pos.lat}° N, {pos.lon}° E')
```

```
Oslo is at 59.9° N, 10.8° E
```