量化金融 python 基础 7-Class

1 Class Introduction

Class，类，是一种封装数据和方法的方式，是用于创建对象的模板，其内部定义了对象的属性（变量）和方法（函数）。它是面向对象编程的重要概念。

下面通过创建一个名为 TradeData 的交易数据类，来解释类的各项功能。

[1]:
```
class TradeData:
    """"""
    symbol = None
    price = 0
    direction = None
    volume = 0
    # 以上变量叫 class attributes，类属性。
    def __init__(self, symbol, price, direction, volume):
        self.symbol = symbol
        self.price = price
        self.direction = direction
        self.volume = volume
    # __init__ 函数用于创建类的实例时进行初始化操作。
    # 在 def __init__() 函数这一行里，symbol,price, direction, volume 是在创建实例时，由外部参数传入到类里面的。
    # 然后通过 self.symbol = symbol 这样的方式，将外部参数赋予实例属性。
    # 在实例化时，__init__ 函数内的参数必须传入，否则会报错。

    def output_trade_result(self):
        print(
            f"标的资产\t{self.symbol}\n"
```

```
            f"成交方向\t{self.direction}\n"
            f"成交金额\t{self.price * self.volume}"
        )
    # output_trade_result 叫实例方法。
    # 在实例化之后，可以用过实例去调用该函数。
    # 需要提醒的是，类内部的实例方法，在定义的时候都要传 self 这个参数，
    # 因为 self 包含了属性变量，需要传入该参数，才能在函数中通过 self.xxx 的方式调用属性变量。
```

[2]:
```
trade1 = TradeData("600001", 80, "买入", 1000)
trade2 = TradeData("600002", 90, "卖出", 800)
# trade1, trade2 为两个实例。
# 这个过程就叫实例化。
```

[3]:
```
# 调用实例方法
trade1.output_trade_result()
```

```
标的资产    600001
成交方向    买入
成交金额    80000
```

2 私有方法和私有属性 Private Method and Private Attribute

在定义类的时候，有些类属性和实例方法在设计时不希望被外部实例直接访问。于是有了私有方法和私有属性。

他们是通过在名称前面添加”__“双下划线来实现的。

[4]:
```
class TradeData:
    symbol = None
    price = 0
    direction = None
    volume = 0
    __cashflow = 0 # 私有属性

    def __init__(self, symbol, price, direction, volume):
        self.symbol = symbol
        self.price = price
```

```
        self.direction = direction
        self.volume = volume
        self.__cashflow = 0

    def __trading_amount(self): # 私有方法， 只能在类内部函数调用。
        amount = self.price * self.volume
        return amount

    def cal_cashflow(self):
        if self.direction == "买入":
            self.__cashflow = - self.__trading_amount()
        else:
            self.__cashflow = self.__trading_amount()
        return self.__cashflow

    def output_trade_result(self):
        print(
        f"标的资产\t{self.symbol}\n"
        f"成交方向\t{self.direction}\n"
        f"成交金额\t{self.__trading_amount()}"
        )
```

[5]:
```
trade1 = TradeData("600001", 80, "买入", 1000)
```

[6]:
```
trade1.cal_cashflow()
```

[6]:
```
-80000
```

[7]:
```
# 调用私有方法报错
trade1.__trading_amount()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[7], line 2
      1 # 调用私有方法报错
----> 2 trade1.__trading_amount()
```

```
AttributeError: 'TradeData' object has no attribute '__trading_amount'
```

[8]:
```
# 访问私有属性报错
trade1.__cashflow
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[8], line 2
      1 # 访问私有属性报错
----> 2 trade1.__cashflow

AttributeError: 'TradeData' object has no attribute '__cashflow'
```

3 @classmethod 装饰器，也叫类方法

类在实例化之前是没法直接调用内部方法的。

想要实现通过类本身进行调用，而不进行创建类的实例，那就需要用到 @classmethod 类方法装饰器。

使用 @classmethod 的函数，第一个参数通常被命名为 cls，不再是 self。

提醒一下，添加 @classmethod, 只能访问类属性，不能访问实例属性。

[10]:
```
class TradeData:
    """"""
    symbol = None
    price = 0
    direction = None
    volume = 0

    def __init__(self, symbol, price, direction, volume):
        self.symbol = symbol
        self.price = price
        self.direction = direction
        self.volume = volume
```

```
    @classmethod # 直接在函数上方添加装饰器
    def output_trade_result(cls):
        #trade = cls("Example", 10, "Buy", 5)
        print(
            f"Symbol:\t{cls.symbol}\n"
            f"Direction:\t{cls.direction}\n"
            f"Amount:\t{cls.price * cls.volume}\n"
        )
```

[11]:
```
# 无需实例化，直接输出类属性的值
TradeData.output_trade_result()
```

```
Symbol: None
Direction:      None
Amount: 0
```

4 @staticmethod 装饰器

与 @classmethod 相似的，还有 @staticmethod 装饰器，也叫静态方法。

静态方法是个独立单纯的函数，仅托管于某个类的名称空间中，它不依赖于类实例，也不需要访问实例属性或者调用其他实例方法，

@staticmethod 没有隐含的第一个参数，即没有 self 或者 cls，其更像是寻常普通的函数，只接受函数定义中指定的参数。

[11]:
```
class TradeData:
    """"""
    symbol = None
    price = 0
    direction = None
    volume = 0

    def __init__(self, symbol, price, direction, volume):
        self.symbol = symbol
        self.price = price
        self.direction = direction
```

```
        self.volume = volume

    @staticmethod
    def output_trade_result(symbol, price, direction, volume):
        print(
            f"Symbol:\t{symbol}\n"
            f"Direction:\t{direction}\n"
            f"Amount:\t{price * volume}\n"
        )
```

[12]:
```
TradeData.output_trade_result(symbol = "600001", price = 100, direction = "买入", volume = 80)
```

```
Symbol: 600001
Direction:      买入
Amount: 8000
```

4.1 @staticmethod，@classmethod 的区别

- @classmethod 类方法的第一个参数通常被命名为 cls，它表示类本身。
  类方法通过类名调用，并将类本身作为第一个参数传入，可以使用 cls 参数来访问类的属性和其他类方法。
- @staticmethod 没有隐式的参数，它通过类名直接调用，并接受所有参数作为显式参数。
  静态方法不能直接访问类属性或者实例属性。它们与类的状态无关，不依赖于类的实例。

5 @property 装饰器

python 内置的 @property 装饰器可以把一个方法转变成属性，使用者可以直接通过转变后的属性取值。

[13]:
```
class TradeData:
    """"""
    symbol = None
    price = 0
    direction = None
    volume = 0
```

```
    def __init__(self, symbol, price, direction, volume):
        self.symbol = symbol
        self.price = price
        self.direction = direction
        self.volume = volume

    @property
    def trading_amount(self):
        return self.price * self.volume

    def output_trade_result(self):
        print(
            f"标的资产\t{self.symbol}\n"
            f"成交方向\t{self.direction}\n"
            f"成交金额\t{self.trading_amount}"
        )
```

[14]:
```
# 通过属性调用，而不是方法。
trade1 = TradeData("600001", 80, "买入", 1000)
trade1.trading_amount
```

[14]: 80000

交易额是根据标的物价格乘交易数量决定的。

但有时候要反过来，在知道标的物价格的情况下，想交易大概额度的金额，要交易多少对应的数量呢？

我们可以通过设置 trading_amount，求 volume，但这时候又会遇到问题。

[15]:
```
trade1.trading_amount = 10000
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[15], line 1
----> 1 trade1.trading_amount = 10000
```

```
AttributeError: can't set attribute 'trading_amount'
```

添加了 property 装饰器的 trading_amount，可以被视为属性，但不能赋值。

为了解决这个问题，引入 property setter

[16]:
```
class TradeData:
    """"""
    symbol = None
    price = 0
    direction = None
    volume = 0

    def __init__(self, symbol, price, direction, volume):
        self.symbol = symbol
        self.price = price
        self.direction = direction
        self.volume = volume

    @property
    def trading_amount(self):
        return self.price * self.volume

    @trading_amount.setter # 增加该代码段，根据 trading_amount 计算 volume，使赋值 trading_amount 成为可能。
    def trading_amount(self, assumed_amount):
        self.volume = assumed_amount / self.price

    def output_trade_result(self):
        print(
            f"标的资产\t{self.symbol}\n"
            f"成交方向\t{self.direction}\n"
            f"成交金额\t{self.trading_amount}"
        )
```

[17]:
```
trade1 = TradeData("600001", 80, "买入", 0) # trading_volume 未知，先输入 0 代替。
trade1.trading_amount = 10000
```

```
trade1.volume
```

[17]: 125.0

6 类属性可变数据结构内存共享的问题

在课程基础 4 中提到过，list, dictionary 是可变变量，用 hash() 函数可以判断。

可变变量在类的应用中需要引起注意，特别是放置在类属性中的可变变量，看以下例子。

设计一个 AccountRecord 类用来存储单个账户发生的成交记录 TradeData。

[18]:
```
class AccountRecord:
    records:list[TradeData] = []
    # 注意，records 是类属性中的可变变量。

    def update_record(self, trade_data: TradeData):
        self.records.append(trade_data)
        # 将成交记录以添加到列表方式存储。

    def output_all_record(self):
        for record in self.records:
            record.output_trade_result()
```

[19]:
```
Account1 = AccountRecord()
Account1.update_record(trade1)
Account1.output_all_record()
```

```
标的资产    600001
成交方向    买入
成交金额    10000.0
```

[20]:
```
account2 = AccountRecord()
account2.update_record(trade2)
account2.output_all_record()
```

```
标的资产    600001
成交方向    买入
成交金额    10000.0
```

```
标的资产    600002
成交方向    卖出
成交金额    72000
```

以上不难发现，在初始化第二个账户 account2，并且只添加 trade2 后，account2 出现了两条交易记录。

出现这个问题的原因，就是在设计 AccountRecord 类时，records 被放到了类属性里。造成所有被实例化的对象，属性 records 指向的都是同一块内存区域里的数据。

因此，想要避开这个问题，就要把可变变量，放到类的初始化函数，__init()__ 函数内。

[21]:
```python
class AccountRecord:

    def __init__(self):
        self.records:list[TradeData] = []

    def update_record(self, trade_data: TradeData):
        self.records.append(trade_data)
        # 将成交记录以添加到列表方式存储。

    def output_all_record(self):
        for record in self.records:
            record.output_trade_result()
```

[22]:
```python
Account1 = AccountRecord()
Account1.update_record(trade1)
Account1.output_all_record()
```

```
标的资产    600001
成交方向    买入
成交金额    10000.0
```

[23]:
```python
account2 = AccountRecord()
account2.update_record(trade2)
account2.output_all_record()
```

```
标的资产    600002
成交方向    卖出
成交金额    72000
```

7 Inheritence 继承

类的继承指允许一个类继承另一个类的属性和方法。通过以下方式继承：
class ChildClass(ParentalClass):

通过继承，子类可以直接重用父类的代码，并且可以在不修改父类的情况下添加新的属性和方法。在继承关系中，子类继承了父类的所有公有属性和方法。也就是说，子类可以访问父类的属性和方法。

子类还可以根据需要重写父类的方法或添加自己的方法。

[24]:
```
# 该例子就通过子类 期货交易数据 来继承 普通交易数据类。
# 由于期货交易的合约性质，会多出一个合约乘数 multiplier，在计算交易额时，需要将 multiplier 也乘上。
class FuturesTradeData(TradeData):

    multiplier = 0

    def __init__(self, symbol, price, direction, volume, multiplier):
        super().__init__(symbol, price, direction, volume)
        # 需要在 __init__ 函数内添加代码，又不想重写父类的代码片段，用 super() 函数返回对父类重复代码片段的引用。
        self.multiplier = multiplier

    @property
    def trading_amount(self):
        amount = self.price * self.volume * self.multiplier
        return amount

    def cashflow(self):
        super().cashflow()
```

装饰器的继承规则：

@classmethod 和 @staticmethod 可以被子类继承，子类能直接调用加了这些装饰器的函数

子类无法继承 @property 装饰器（但是能继承 @property 装饰器修饰的属性的 setter 方法和 getter 方法，使用 @ParentalClasss.function_name.setter，getter 本节没有讲述，有兴趣的朋友请自行查阅。）

[25]:
```
trade3 = FuturesTradeData("m2309.DCE", 3000, "买入", 2, 10)
trade3.output_trade_result()
```

```
标的资产    m2309.DCE
成交方向    买入
成交金额    60000
```

私有属性和私有方法也可以在子类中被继承和重写。

子类可以通过定义相同名称的私有属性和私有方法来隐藏或修改父类的实现细节，同时保留接口的一致性。

8 Summary

- 类介绍 Class Introduction
- 私有方法和私有属性 private method and private attribute
- @classmethod
- @staticmethod
- @property
- 类属性可变数据结构内存共享的问题
- 类的继承 Inheritance