

36 | Pandas & Numpy: 策略与回测系统

2019-07-31 景霄

Python核心技术与实战

[进入课程 >](#)



讲述：冯永吉

时长 16:00 大小 14.66M



大家好，我是景霄。

上节课，我们介绍了交易所的数据抓取，特别是 orderbook 和 tick 数据的抓取。今天这节课，我们考虑的是，怎么在这些历史数据上测试一个交易策略。

首先我们要明确，对于很多策略来说，我们上节课抓取的密集 orderbook 和 tick 数据，并不能简单地直接使用。因为数据量太密集，包含了太多细节；而且长时间连接时，网络随机出现的不稳定，会导致丢失部分 tick 数据。因此，我们还需要进行合适的清洗、聚合等操作。

此外，为了进行回测，我们需要一个交易策略，还需要一个测试框架。目前已存在很多成熟的回测框架，但是为了 Python 学习，我决定带你搭建一个简单的回测框架，并且从中简单

一窥 Pandas 的优势。

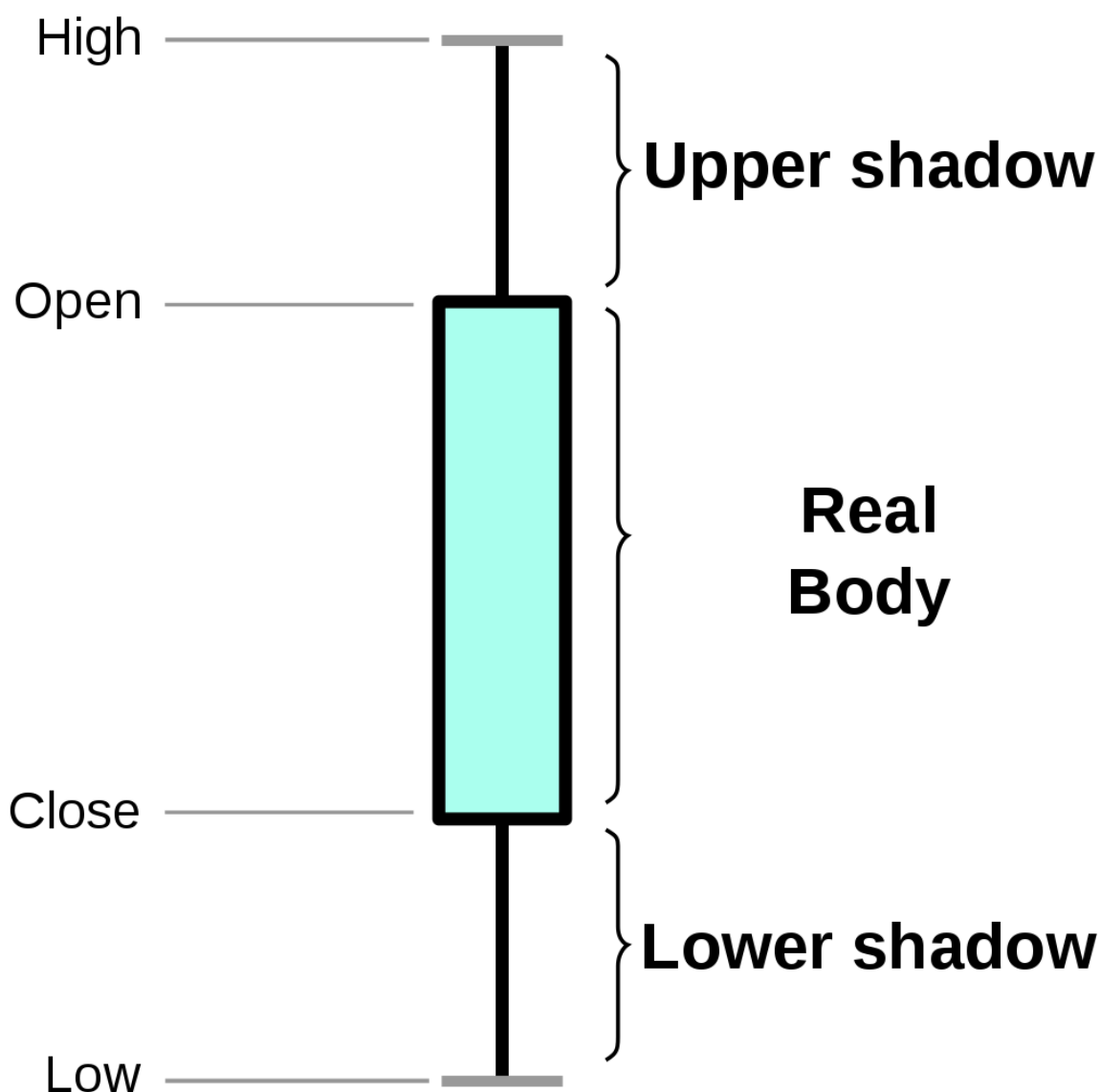
OHLCV 数据

了解过一些股票交易的同学，可能知道 K 线这种东西。K 线又称“蜡烛线”，是一种反映价格走势的图线。它的特色在于，一个线段内记录了多项讯息，相当易读易懂且实用有效，因此被广泛用于股票、期货、贵金属、数字货币等行情的技术分析。下面便是一个 K 线示意图。



K 线示意图

其中，每一个小蜡烛，都代表着当天的开盘价（Open）、最高价（High）、最低价（Low）和收盘价（Close），也就是我画的第二张图表示的这样。



K 线的“小蜡烛” -- OHLC

类似的，除了日 K 线之外，还有周 K 线、小时 K 线、分钟 K 线等等。那么这个 K 线是怎么计算来的呢？


我们以小时 K 线图为例，还记得我们当时抓取的 tick 数据吗？也就是每一笔交易的价格和数量。那么，如果从上午 10:00 开始，我们开始积累 tick 的交易数据，以 10:00 开始的第一个交易作为 Open 数据，11:00 前的最后一笔交易作为 Close 值，并把这一个小时最低和最高的成交价格分别作为 High 和 Low 的值，我们就可以绘制出这一个小时对应的“小蜡烛”形状了。

如果再加上这一个小时总的成交量（Volume），就得到了 OHLCV 数据。

所以，如果我们一直抓取着 tick 底层原始数据，我们就能在上层聚合出 1 分钟 K 线、小时 K 线以及日、周 k 线等等。如果你对这一部分操作有兴趣，可以把此作为今天的课后作业来实践。

接下来，我们将使用 Gemini 从 2015 年到 2019 年 7 月这个时间内，BTC 对 USD 每个小时的 OHLCV 数据，作为策略和回测的输入。你可以在[这里](#)下载数据。

数据下载完成后，我们可以利用 Pandas 读取，比如下面这段代码。

 复制代码

```
1 def assert_msg(condition, msg):
2     if not condition:
3         raise Exception(msg)
4
5 def read_file(filename):
6     # 获得文件绝对路径
7     filepath = path.join(path.dirname(__file__), filename)
8
9     # 判定文件是否存在
10    assert_msg(path.exists(filepath), " 文件不存在 ")
11
12    # 读取 CSV 文件并返回
13    return pd.read_csv(filepath,
14                       index_col=0,
15                       parse_dates=True,
16                       infer_datetime_format=True)
17
18 BTCUSD = read_file('BTCUSD_GEMINI.csv')
19 assert_msg(BTCUSD.__len__() > 0, '读取失败')
20 print(BTCUSD.head())
21
22
23 ##### 输出 #####
24 Time                Symbol    Open    High    Low    Close    Volume
25 Date
26 2019-07-08 00:00:00  BTCUSD  11475.07  11540.33  11469.53  11506.43  10.770731
27 2019-07-07 23:00:00  BTCUSD  11423.00  11482.72  11423.00  11475.07  32.996559
28 2019-07-07 22:00:00  BTCUSD  11526.25  11572.74  11333.59  11423.00  48.937730
29 2019-07-07 21:00:00  BTCUSD  11515.80  11562.65  11478.20  11526.25  25.323908
30 2019-07-07 20:00:00  BTCUSD  11547.98  11624.88  11423.94  11515.80  63.211972
```

这段代码提供了两个工具函数。

一个是 `read_file`，它的作用是，用 `pandas` 读取 `csv` 文件。

另一个是 `assert_msg`，它的作用类似于 `assert`，如果传入的条件（`condition`）为否，就会抛出异常。不过，你需要提供一个参数，用于指定要抛出的异常信息。

回测框架

说完了数据，我们接着来看回测数据。常见的回测框架有两类。一类是向量化回测框架，它通常基于 `Pandas+Numpy` 来自己搭建计算核心；后端则是用 `MySQL` 或者 `MongoDB` 作为源。这种框架通过 `Pandas+Numpy` 对 `OHLC` 数组进行向量运算，可以在较长的历史数据上进行回测。不过，因为这类框架一般只用 `OHLC`，所以模拟会比较粗糙。

另一类则是事件驱动型回测框架。这类框架，本质上是针对每一个 `tick` 的变动或者 `orderbook` 的变动生成事件；然后，再把一个个事件交给策略进行执行。因此，虽然它的拓展性很强，可以允许更加灵活的策略，但回测速度是很慢的。

我们想要学习量化交易，使用大型成熟的回测框架，自然是第一选择。

比如 `Zipline`，就是一个热门的事件驱动型回测框架，背后有大型社区和文档的支持。

`PyAlgoTrade` 也是事件驱动的回测框架，文档相对完整，整合了知名的技术分析（`Technical Analysis`）库 `TA-Lib`。在速度和灵活方面，它比 `Zipline` 强。不过，它的一大硬伤是不支持 `Pandas` 的模块和对象。

显然，对于我们 `Python` 学习者来说，第一类也就是向量型回测框架，才是最适合我们练手的项目了。那么，我们就开始吧。

首先，我先为你梳理下回测流程，也就是下面五步：

1. 读取 `OHLC` 数据；
2. 对 `OHLC` 进行指标运算；
3. 策略根据指标向量决定买卖；
4. 发给模拟的“交易所”进行交易；
5. 最后，统计结果。

对此，使用之前学到的面向对象思维方式，我们可以大致抽取三个类：

交易所类（ExchangeAPI）：负责维护账户的资金和仓位，以及进行模拟的买卖；

策略类（Strategy）：负责根据市场信息生成指标，根据指标决定买卖；

回测类框架（Backtest）：包含一个策略类和一个交易所类，负责迭代地对每个数据点调用策略执行。

接下来，我们先从最外层的大框架开始。这样的好处在于，我们是从上到下、从外往内地思考，虽然还没有开始设计依赖项（Backtest 的依赖项是 ExchangeAPI 和 Strategy），但我们可以推测出它们应有的接口形式。推测接口的本质，其实就是推测程序的输入。

这也是我在一开始提到过的，对于程序这个“黑箱”，你在一开始设计的时候，就要想好输入和输出。

回到最外层 Backtest 类。我们需要知道，输出是最后的收益，那么显然，输入应该是初始输入的资金数量（cash）。

此外，为了模拟得更加真实，我们还要考虑交易所的手续费（commission）。手续费的多少取决于券商（broker）或者交易所，比如我们买卖股票的券商手续费可能是万七，那就是 0.0007。但是在比特币交易领域，手续费通常会稍微高一点，可能是千分之二左右。当然，无论怎么多，一般也不会超过 5 %。否则我们大家交易几次就破产了，也就不会有人去交易了。

这里说一句题外话，不知道你有没有发现，无论数字货币的价格是涨还是跌，总有一方永远不亏，那就是交易所。因为只要有人交易，他们就有白花花的银子进账。

回到正题，至此，我们就确定了 Backtest 的输入和输出。

它的输入是：

OHLC 数据；

初始资金；

手续费率；


交易所类；

策略类。

输出则是：

最后剩余市值。

对此，你可以参考下面这段代码：

 复制代码

```
1 class Backtest:
2     """
3     Backtest 回测类，用于读取历史行情数据、执行策略、模拟交易并估计
4     收益。
5
6     初始化的时候调用 Backtest.run 来时回测
7
8     instance, or `backtesting.backtesting.Backtest.optimize` to
9     optimize it.
10    """
11
12    def __init__(self,
13                  data: pd.DataFrame,
14                  strategy_type: type(Strategy),
15                  broker_type: type(ExchangeAPI),
16                  cash: float = 10000,
17                  commission: float = .0):
18        """
19        构造回测对象。需要的参数包括：历史数据，策略对象，初始资金数量，手续费率等。
20        初始化过程包括检测输入类型，填充数据空值等。
21
22        参数:
23        :param data: pd.DataFrame pandas Dataframe 格式的历史 OHLCV 数
24        :param broker_type: type(ExchangeAPI) 交易所 API 类型，负责执行买卖操作以及账
25        :param strategy_type: type(Strategy) 策略类型
26        :param cash: float 初始资金数量
27        :param commission: float 每次交易手续费率。如 2% 的手续费此处为
28        """
29
30        assert_msg(issubclass(strategy_type, Strategy), 'strategy_type 不是一个 Strategy
31        assert_msg(issubclass(broker_type, ExchangeAPI), 'strategy_type 不是一个 Strateg
32        assert_msg(isinstance(commission, Number), 'commission 不是浮点数值类型')
33
34        data = data.copy(False)
35
36        # 如果没有 Volume 列，填充 NaN
37        if 'Volume' not in data:
38            data['Volume'] = np.nan
```

```

39
40 # 验证 OHLC 数据格式
41 assert_msg(len(data.columns & {'Open', 'High', 'Low', 'Close', 'Volume'}) == 5,
42            (" 输入的`data`格式不正确，至少需要包含这些列： "
43             "'Open', 'High', 'Low', 'Close'"))
44
45 # 检查缺失值
46 assert_msg(not data[['Open', 'High', 'Low', 'Close']].max().isnull().any(),
47            ('部分 OHLC 包含缺失值，请去掉那些行或者通过差值填充。'))
48
49 # 如果行情数据没有按照时间排序，重新排序一下
50 if not data.index.is_monotonic_increasing:
51     data = data.sort_index()
52
53 # 利用数据，初始化交易所对象和策略对象。
54 self._data = data # type: pd.DataFrame
55 self._broker = broker_type(data, cash, commission)
56 self._strategy = strategy_type(self._broker, self._data)
57 self._results = None
58
59 def run(self):
60     """
61     运行回测，迭代历史数据，执行模拟交易并返回回测结果。
62     Run the backtest. Returns `pd.Series` with results and statistics.
63
64     Keyword arguments are interpreted as strategy parameters.
65     """
66     strategy = self._strategy
67     broker = self._broker
68
69     # 策略初始化
70     strategy.init()
71
72     # 设定回测开始和结束位置
73     start = 100
74     end = len(self._data)
75
76     # 回测主循环，更新市场状态，然后执行策略
77     for i in range(start, end):
78         # 注意要先把市场状态移动到第 i 时刻，然后再执行策略。
79         broker.next(i)
80         strategy.next(i)
81
82     # 完成策略执行之后，计算结果并返回
83     self._results = self._compute_result(broker)
84     return self._results
85
86 def _compute_result(self, broker):
87     s = pd.Series()
88     s['初始市值'] = broker.initial_cash
89     s['结束市值'] = broker.market_value
90     s['收益'] = broker.market_value - broker.initial_cash

```




这段代码有点长，但是核心其实就两部分。

初始化函数（**init**）：传入必要参数，对 OHLC 数据进行简单清洗、排序和验证。我们从不同地方下载的数据，可能格式不一样；而排序的方式也可能是从前往后。所以，这里我们把数据统一设置为按照时间从之前往现在的排序。

执行函数（**run**）：这是回测框架的主要循环部分，核心是更新市场还有更新策略的时间。迭代完成所有的历史数据后，它会计算收益并返回。

你应该注意到了，此时，我们还没有定义策略和交易所 API 的结构。不过，通过回测的执行函数，我们可以确定这两个类的接口形式。

策略类（**Strategy**）的接口形式为：

初始化函数 `init()`，根据历史数据进行指标（**Indicator**）计算。

步进函数 `next()`，根据当前时间和指标，决定买卖操作，并发给交易所类执行。

交易所类（**ExchangeAPI**）的接口形式为：

步进函数 `next()`，根据当前时间，更新最新的价格；

买入操作 `buy()`，买入资产；

卖出操作 `sell()`，卖出资产。

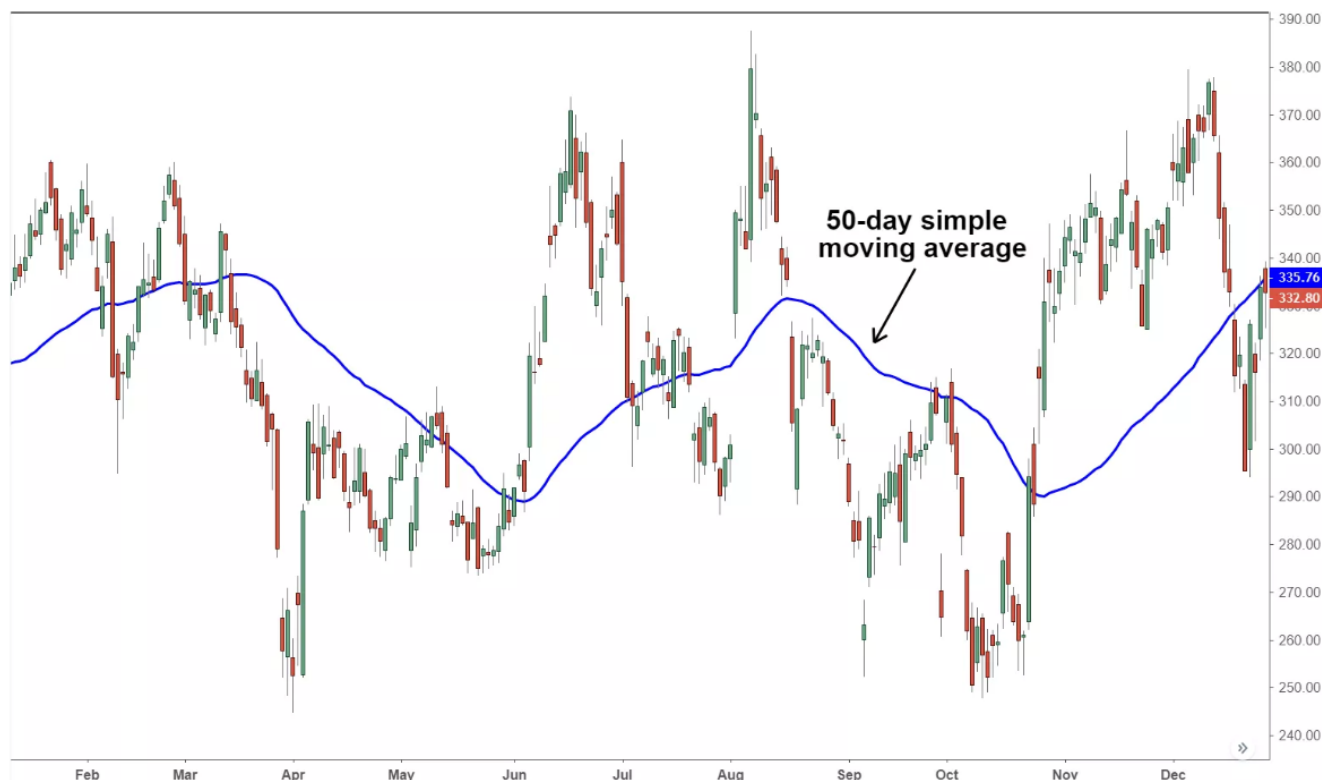
交易策略

接下来我们来看交易策略。交易策略的开发是一个非常复杂的学问。为了达到学习的目的，我们来想一个简单的策略——移动均值交叉策略。

为了了解这个策略，我们先了解一下，什么叫做简单移动均值（**Simple Moving Average**，简称为 **SMA**，以下皆用 **SMA** 表示简单移动均值）。我们知道， N 个数的序列 $x[0]$ 、 $x[1]$ $x[N]$ 的均值，就是这 N 个数的和除以 N 。

现在，我假设一个比较小的数 K ，比 N 小很多。我们用一个 K 大小的滑动窗口，在原始的数组上滑动。通过对每次框住的 K 个元素求均值，我们就可以得到，原始数组的窗口大小为 K 的 SMA 了。

SMA，实质上就是对原始数组进行了一个简单平滑处理。比如，某支股票的价格波动很大，那么，我们用 SMA 平滑之后，就会得到下面这张图的效果。



某个投资品价格的 SMA，窗口大小为 50

你可以看出，如果窗口大小越大，那么 SMA 应该越平滑，变化越慢；反之，如果 SMA 比较小，那么短期的变化也会越快地反映在 SMA 上。

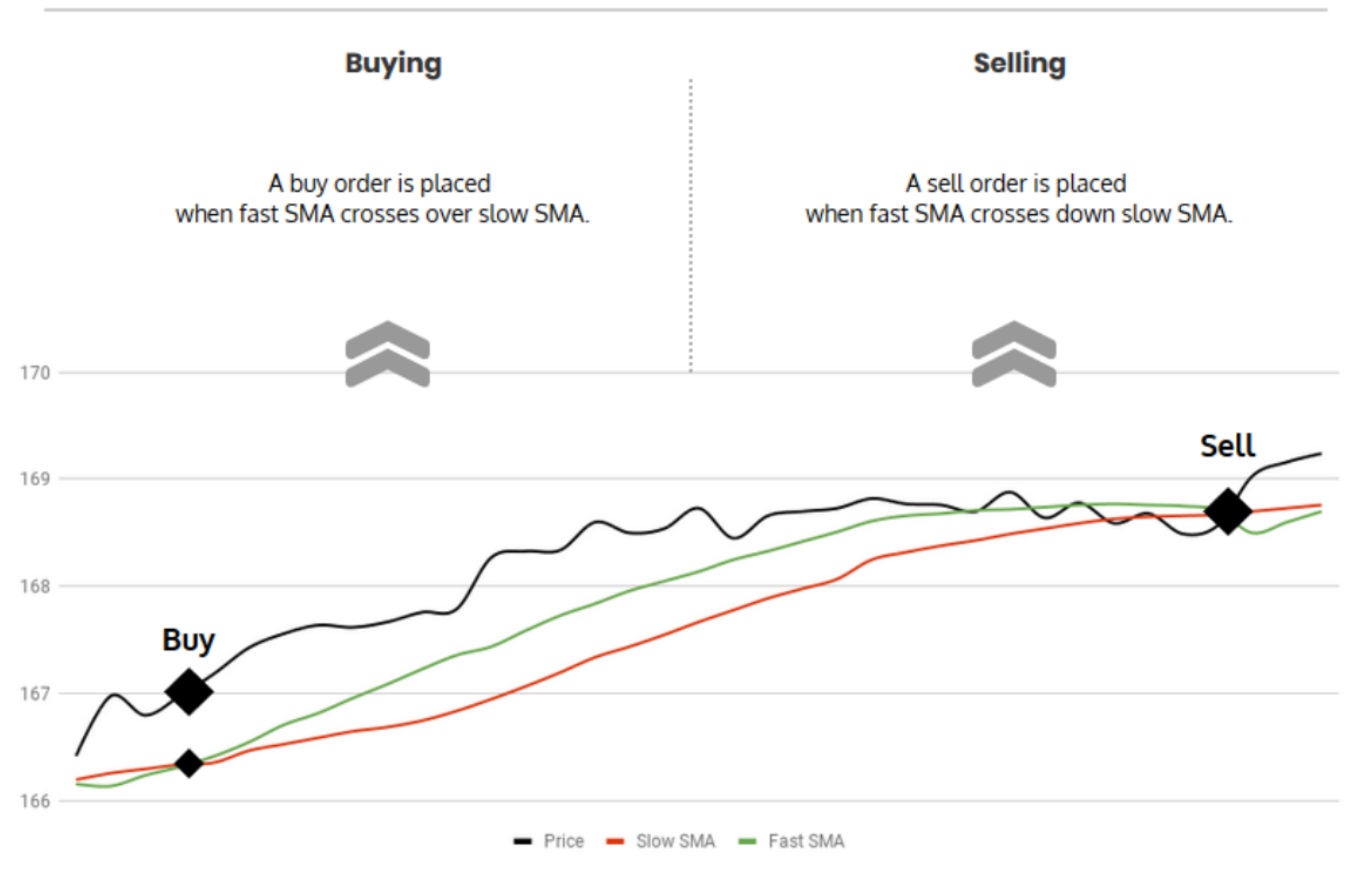
于是，我们想到，能不能对投资品的价格设置两个指标呢？这两指标，一个是小窗口的 SMA，一个是大窗口的 SMA。

如果小窗口的 SMA 曲线从下面刺破或者穿过大窗口 SMA，那么说明，这个投资品的价格在短期内快速上涨，同时这个趋势很强烈，可能是一个买入的信号；

反之，如果大窗口的 SMA 从下方突破小窗口 SMA，那么说明，投资品的价格在短期内快速下跌，我们应该考虑卖出。

下面这幅图，就展示了这两种情况。

SMA cross




明白了这里的概念和原理后，接下来的操作就不难了。利用 Pandas，我们可以非常简单地计算 SMA 和 SMA 交叉。比如，你可以引入下面两个工具函数：

复制代码

```
1 def SMA(values, n):
2     """
3     返回简单滑动平均
4     """
5     return pd.Series(values).rolling(n).mean()
6
7 def crossover(series1, series2) -> bool:
8     """
9     检查两个序列是否在结尾交叉
10    :param series1: 序列 1
11    :param series2: 序列 2
12    :return: 如果交叉返回 True，反之 False
13    """
14    return series1[-2] < series2[-2] and series1[-1] > series2[-1]
```

如代码所示，对于输入的一个数组，Pandas 的 `rolling(k)` 函数，可以方便地计算窗口大小为 K 的 SMA 数组；而想要检查某个时刻两个 SMA 是否交叉，你只需要查看两个数组末尾的两个元素即可。

那么，基于此，我们就可以开发出一个简单的策略了。下面这段代码表示策略的核心思想，我做了详细的注释，你理解起来应该没有问题：

 复制代码


```
1 def next(self, tick):
2     # 如果此时快线刚好越过慢线，买入全部
3     if crossover(self.sma1[:tick], self.sma2[:tick]):
4         self.buy()
5
6     # 如果是慢线刚好越过快线，卖出全部
7     elif crossover(self.sma2[:tick], self.sma1[:tick]):
8         self.sell()
9
10    # 否则，这个时刻不执行任何操作。
11    else:
12        pass
```

说完策略的核心思想，我们开始搭建策略类的框子。

首先，我们要考虑到，策略类 `Strategy` 应该是一个可以被继承的类，同时应该包含一些固定的接口。这样，回测器才能方便地调用。

于是，我们可以定义一个 `Strategy` 抽象类，包含两个接口方法 `init` 和 `next`，分别对应我们前面说的指标计算和步进函数。不过注意，抽象类是不能被实例化的。所以，我们必须定义一个具体的子类，同时实现了 `init` 和 `next` 方法才可以。

这个类的定义，你可以参考下面代码的实现：

 复制代码

```
1 import abc
2 import numpy as np
3 from typing import Callable
4
5 class Strategy(metaclass=abc.ABCMeta):
6     """
```

```

7     抽象策略类，用于定义交易策略。
8
9     如果要定义自己的策略类，需要继承这个基类，并实现两个抽象方法：
10    Strategy.init
11    Strategy.next
12    """
13    def __init__(self, broker, data):
14        """
15        构造策略对象。
16
17        @params broker: ExchangeAPI    交易 API 接口，用于模拟交易
18        @params data: list              行情数据数据
19        """
20        self._indicators = []
21        self._broker = broker # type: _Broker
22        self._data = data # type: _Data
23        self._tick = 0
24
25    def I(self, func: Callable, *args) -> np.ndarray:
26        """
27        计算买卖指标向量。买卖指标向量是一个数组，长度和历史数据对应；
28        用于判定这个时间点上需要进行 " 买 " 还是 " 卖 "。
29
30        例如计算滑动平均：
31        def init():
32            self.sma = self.I(utils.SMA, self.data.Close, N)
33        """
34        value = func(*args)
35        value = np.asarray(value)
36        assert_msg(value.shape[-1] == len(self._data.Close), '指示器长度必须和 data 长度相
37
38        self._indicators.append(value)
39        return value
40
41    @property
42    def tick(self):
43        return self._tick
44
45    @abc.abstractmethod
46    def init(self):
47        """
48        初始化策略。在策略回测 / 执行过程中调用一次，用于初始化策略内部状态。
49        这里也可以预计算策略的辅助参数。比如根据历史行情数据：
50        计算买卖的指示器向量；
51        训练模型 / 初始化模型参数
52        """
53        pass
54
55    @abc.abstractmethod
56    def next(self, tick):
57        """
58        步进函数，执行第 tick 步的策略。tick 代表当前的 " 时间 "。比如 data[tick] 用于访问当

```


```

59         """
60         pass
61
62     def buy(self):
63         self._broker.buy()
64
65     def sell(self):
66         self._broker.sell()
67
68     @property
69     def data(self):
70         return self._data

```

为了方便访问成员，我们还定义了一些 Python property。同时，我们的买卖请求是由策略类发出、由交易所 API 来执行的，所以我们的策略类里依赖于 ExchangeAPI 类。

现在，有了这个框架，我们实现移动均线交叉策略就很简单了。你只需要在 init 函数中，定义计算大小窗口 SMA 的逻辑；同时，在 next 函数中完成交叉检测和买卖调用就行了。具体实现，你可以参考下面这段代码：

 复制代码

```

1  from utils import assert_msg, crossover, SMA
2
3  class SmaCross(Strategy):
4      # 小窗口 SMA 的窗口大小，用于计算 SMA 快线
5      fast = 10
6
7      # 大窗口 SMA 的窗口大小，用于计算 SMA 慢线
8      slow = 20
9
10     def init(self):
11         # 计算历史上每个时刻的快线和慢线
12         self.sma1 = self.I(SMA, self.data.Close, self.fast)
13         self.sma2 = self.I(SMA, self.data.Close, self.slow)
14
15     def next(self, tick):
16         # 如果此时快线刚好越过慢线，买入全部
17         if crossover(self.sma1[:tick], self.sma2[:tick]):
18             self.buy()
19
20         # 如果是慢线刚好越过快线，卖出全部
21         elif crossover(self.sma2[:tick], self.sma1[:tick]):
22             self.sell()
23
24         # 否则，这个时刻不执行任何操作。

```

```
25         else:
26             pass
```

模拟交易

到这里，我们的回测就只差最后一块儿了。胜利就在眼前，我们继续加油。


我们前面提到过，交易所类负责模拟交易，而模拟的基础，就是需要当前市场的价格。这里，我们可以用 OHLC 中的 Close，作为那个时刻的价格。

此外，为了简化设计，我们假设买卖操作都利用的是当前账户的所有资金、仓位，且市场容量足够大。这样，我们的下单请求就能够马上完全执行。

也别忘了手续费这个大头。考虑到有手续费的情况，此时，我们最核心的买卖函数应该怎么来写呢？

我们一起来想这个问题。假设，我们现在有 1000.0 元，此时 BTC 的价格是 100.00 元（当然没有这么好的事情啊，这里只是假设），并且交易手续费为 1%。那么，我们能买到多少 BTC 呢？


我们可以采用这种算法：

 复制代码

```
1 买到的数量 = 投入的资金 * (1.0 - 手续费) / 价格
```


那么此时，你就能收到 9.9 个 BTC。

类似的，卖出的时候结算方式如下，也不难理解：

 复制代码

```
1 卖出的收益 = 持有的数量 * 价格 * (1.0 - 手续费)
```

所以，最终模拟交易所类的实现，你可以参考下面这段代码：

 复制代码

```
1 from utils import read_file, assert_msg, crossover, SMA
2
3 class ExchangeAPI:
4     def __init__(self, data, cash, commission):
5         assert_msg(0 < cash, " 初始现金数量大于 0，输入的现金数量：{}".format(cash))
6         assert_msg(0 <= commission <= 0.05, " 合理的手续费率一般不会超过 5%，输入的费率：{}".format(commission))
7         self._initial_cash = cash
8         self._data = data
9         self._commission = commission
10        self._position = 0
11        self._cash = cash
12        self._i = 0
13
14    @property
15    def cash(self):
16        """
17        :return: 返回当前账户现金数量
18        """
19        return self._cash
20
21    @property
22    def position(self):
23        """
24        :return: 返回当前账户仓位
25        """
26        return self._position
27
28    @property
29    def initial_cash(self):
30        """
31        :return: 返回初始现金数量
32        """
33        return self._initial_cash
34
35    @property
36    def market_value(self):
37        """
38        :return: 返回当前市值
39        """
40        return self._cash + self._position * self.current_price
41
42    @property
43    def current_price(self):
44        """
45        :return: 返回当前市场价格
46        """
47        return self._data.Close[self._i]
```



```


48
49     def buy(self):
50         """
51         用当前账户剩余资金，按照市场价格全部买入
52         """
53         self._position = float(self._cash / (self.current_price * (1 + self._commission
54         self._cash = 0.0
55
56     def sell(self):
57         """
58         卖出当前账户剩余持仓
59         """
60         self._cash += float(self._position * self.current_price * (1 - self._commission
61         self._position = 0.0
62
63     def next(self, tick):
64         self._i = tick

```

其中的 `current_price`（当前价格），可以方便地获得模拟交易所当前时刻的商品价格；而 `market_value`，则可以获得当前总市值。在初始化函数的时候，我们检查手续费率和输入的现金数量，是不是在一个合理的范围。

有了所有的这些部分，我们就可以来模拟回测啦！

首先，我们设置初始资金量为 10000.00 美元，交易所手续费率为 0。这里你可以猜一下，如果我们从 2015 年到现在，都按照 SMA 来买卖，现在应该有多少钱呢？


 复制代码

```

1 def main():
2     BTCUSD = read_file('BTCUSD_GEMINI.csv')
3     ret = Backtest(BTCUSD, SmaCross, ExchangeAPI, 10000.0, 0.00).run()
4     print(ret)
5
6 if __name__ == '__main__':
7     main()

```

铛铛铛，答案揭晓，程序将输出：

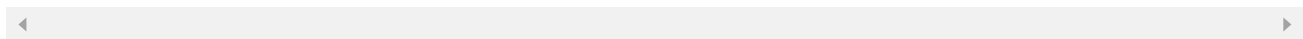
 复制代码

```


1 初始市值      10000.000000

```

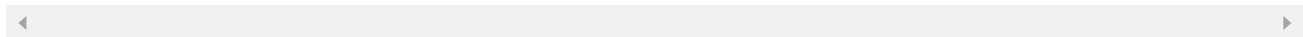
```
2 结束市值      576361.772884
3 收益          566361.772884
```



哇，结束时，我们将有 57 万美元，翻了整整 57 倍啊！简直不要太爽。不过，等等，这个手续费率为 0，实在是有点碍眼，因为根本不可能啊。我们现在来设一个比较真实的值吧，大概千分之三，然后再来试试：

 复制代码

```
1 初始市值      10000.000000
2 结束市值      2036.562001
3 收益          -7963.437999
```



什么鬼？我们变成赔钱了，只剩下 2000 美元了！这是真的吗？

这是真的，也是假的。

我说的“真”是指，如果你真的用 SMA 交叉这种简单的方法去交易，那么手续费摩擦和滑点等因素，确实可能让你的高频策略赔钱。

而我说是“假”是指，这种模拟交易的方式非常粗糙。真实的市场情况，并非这么理想——比如买卖请求永远马上执行；再比如，我们在市场中进行交易的同时不会影响市场价格等，这些理想情况都是不可能的。所以，很多时候，回测永远赚钱，但实盘马上赔钱。

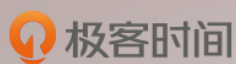
总结

这节课，我们继承上一节，介绍了回测框架的分类、数据的格式，并且带你从头开始写了一个简单的回测系统。你可以把今天的代码片段“拼”起来，这样就会得到一个简化的回测系统样例。同时，我们实现了一个简单的交易策略，并且在真实的历史数据上运行了回测结果。我们观察到，在加入手续费后，策略的收益情况发生了显著的变化。

思考题

最后，给你留一个思考题。之前我们介绍了如何抓取 tick 数据，你可以根据抓取的 tick 数据，生成 5 分钟、每小时和每天的 OHLCV 数据吗？欢迎在留言区写下你的答案和问题，

也欢迎你把这篇文章分享出去。



Python 核心技术与实战

系统提升你的 Python 能力

景霄

Facebook 资深工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 35 | RESTful & Socket: 行情数据对接和抓取

下一篇 37 | Kafka & ZMQ : 自动化交易流水线

精选留言 (8)

写留言



Jingxiao 置顶

2019-08-01

整理后的代码在这里：<https://github.com/Eyelidstl/GeekTimePythonClass>



👍 8



方向

2019-07-31

有没有整理后的源代码，想统一查看

展开 ▾



👍 4



fy

2019-07-31

老师，可以用git管理每次分析的代码么？

展开 ▾



1



小侠龙旋风

2019-08-04

很多时候，回测永远赚钱，但实盘马上赔钱[捂脸]

好几行关于pandas的看不懂，比如：`pd.Series(values).rolling(n).mean()`，再比如：`data[['Open', 'High', 'Low', 'Close']]`

这种写法挺实用的，`__file__`就是当前python文件的绝对路径，取上层目录拼出另一个文...

展开 ▾



瞳梦

2019-08-02

`assert_msg(not data[['Open', 'High', 'Low', 'Close']].max(skipna=False).isnull().any())`

这一行`max()`方法应该要加一个参数: `skipna=False`

展开 ▾



无才不肖生

2019-08-02

而想要检查某个时刻两个 SMA 是否交叉，你只需要查看两个数...

这个我理解的有问题吗，只拿最后两人数作比较不能确定吧，窗口设置10个数时，可能在1到8个数时相等，不是判断不准确？

展开 ▾



Destroy、

2019-07-31

`def buy(self):`

`"""`

用当前账户剩余资金，按照市场价格全部买入

`"""`

`self._position = float(self._cash / (self.current_price * (1 + self._commission)))...`

展开 ▾





许童童

2019-07-31

看了老师的文章，对金融又感兴趣了。

展开 ∨

