Table of Content

1 声明

档是基于backtrader官方文档的翻译
文档地址 https://www.backtrader.com/docu/
回测数据文件 https://github.com/jackvip/backtrader/blob/master/orcl-1995-2014.txt

介绍

1. backtrader的2个目标

使用简单
参考1

2. backtrader的运行流程

制定策略1.1 确定潜在的可调参数1.2 实例化您在策略中需要的指标1.3 写下进入/退出市场的逻辑
创建Cerebro引擎（西班牙语大脑的意思） 2.1 注入策略 2.2 使用cerebro.adddata加载回测数据 2.3 执行cerebro.run 2.4 使用cerebro.plot绘制可视化图表

trader是高度可配置的，希望大家发现其中的乐趣。

3. 安装

除了绘图以外，backtrader不需要任何外部依赖包

3.1. 版本要求

基本要求是： Python 2.7 Python 3.2 / 3.3 / 3.4 / 3.5 如有绘图需求，则要求 Matplotlib> = 1.4.1

3.2. 兼容性

backtrader兼容Python2.x/3.x。 backtrader同时在Python2.7和Python3.4下进行过开发和测试。并在Travis下通过持续集成来检查与3.2/3.3/3.5的兼容性测试。

3.3. 从pypi安装

pip install backtrader 如果希望绘图功能，请使用[plotting]选项,这将会安装matplotlib和相关依赖包。 pip install backtrader[plotting]

3.4. 从源码安装

首先从github网站下载一个版本或最新的压缩包： https://github.com/mementum/backtrader

解压并将源代码拷贝到项目目录 命令如下：

```
tar xzf backgrader.tgz
cd backtrader
```

```
cp -r backtrader project_directory
python setup.py install
```

绘图需求，请手动安装matplotlib。

快速开始

们从零开始来跑一些完整的例子，在此之前，先来了解两个基本概念。

1. 折线（Line）

数据（Data Feeds）、技术指标（Indicators）和策略（Strategies）都是折线（Line）。折Line）是由一系列的点组成的。

交易数据（Data Feeds）包含以下几个组成部分： 开盘价（Open）、最高价（High）、最低Low）、收盘价（Close）、成交量（Volume）、持仓量（OpenInterest）等。比如：所有的开盘价（Open）按时间组成一条折线（Line），那么一组交易数据（Data Feeds）就应该包含了6线（Line）。

上时间（DateTime）一共有7条折线（Line）。时间，一般用作一组交易数据的主键。

2. 索引从0开始

问一条折线（Line）的数据时，会默认从下标为0的位置开始，最后一个数据通过下标-1来获这样的设计和Python的迭代器是一致的，所以折线（Line）是可以迭代遍历的。

：创建一个简单移动平均值的策略（均值策略）：

`f.sma = SimpleMovingAverage(.....)` 访问此移动平均线的当前值的最简单方法：`av = self.sma[0]` 所以在回测过程中，无需知道已经处理了多少条/分钟/天/月，“0”一直指向当前值。

按照Python遍历数组的方式，用下标-1来访问最后一个值： previous_value = self.sma[-1] 同理：-2、-3下标也是可以照常使用。

4.3. 从0到100

先跑一个基本框架

```
# 导入backtrader框架
import backtrader as bt
if __name__ == '__main__':
    # 创建Cerebro引擎
    cerebro = bt.Cerebro()
    #Cerebro引擎在后台创建broker(经纪人)，系统默认资金量为10000
    # 引擎运行前打印期出资金
    print('组合期初资金：%.2f' % cerebro.broker.getvalue())
    cerebro.run()
    # 引擎运行后打期末资金
    print('组合期末资金：%.2f' % cerebro.broker.getvalue())
```

运行后，结果输出：

```
组合期初资金：10000.00
组合期末资金：10000.0
```

4.4. 设定现金

在金融世界中，可以肯定的是，只有Losser才以1万开始。让我们更改现金并再次运行该示例。

```
#导入backtrader框架
import backtrader as bt
if __name__ == '__main__':
    # 创建Cerebro引擎
    cerebro = bt.Cerebro()
    # Cerebro引擎在后台创建broker(经纪人)，系统默认资金量为10000

    # 设置投资金额100000.0
    cerebro.broker.setcash(100000.0)
    # 引擎运行前打印期出资金
    print('组合期初资金：%.2f' % cerebro.broker.getvalue())
    cerebro.run()
    # 引擎运行后打期末资金
    print('组合期末资金：%.2f' % cerebro.broker.getvalue())
```

后，结果输出：

```
合期初资金：100000.00
合期末资金：100000.0
```

完成。Let's move to tempestuous waters.（意译：让我们再整点牛逼点的）

5. 加入交易数据

现金是一件快乐的事，这一切背后最终的目的就是: 不动一根手指，就能让自动化策略通过操产的交易数据来增加现金。 无数据，不快乐，让我们继续加入交易数据。

```
ort datetime  #
ort os.path  # 路径管理
ort sys  # 获取当前运行脚本的路径 (in argv[0])

入backtrader框架
ort backtrader as bt
__name__ == '__main__':
 # 创建Cerebro引擎
 cerebro = bt.Cerebro()
 # Cerebro引擎在后台创建broker(经纪人)，系统默认资金量为10000

 # 获取当前运行脚本所在目录
 modpath = os.path.dirname(os.path.abspath(sys.argv[0]))
 # 拼接加载路径
 datapath = os.path.join(modpath, '../../datas/orcl-1995-2014.txt')

 # 创建交易数据集
 data = bt.feeds.YahooFinanceCSVData(
     dataname=datapath,
     # 数据必须大于fromdate
     fromdate=datetime.datetime(2000, 1, 1),
     # 数据必须小于todate
     todate=datetime.datetime(2000, 12, 31),
     reverse=False)

 # 加载交易数据
 cerebro.adddata(data)


 # 设置投资金额100000.0
 cerebro.broker.setcash(100000.0)
 # 引擎运行前打印期出资金
 print('组合期初资金：%.2f' % cerebro.broker.getvalue())
 cerebro.run()
 # 引擎运行后打期末资金
 print('组合期末资金：%.2f' % cerebro.broker.getvalue())
```

运行后，结果输出：

```
组合期初资金：100000.00
组合期末资金：100000.0
```

代码量略有增加，因为我们添加了：找出数据文件所在的路径通过datetime对象过滤我们想要操作的数据

> 注意: Yahoo的价格数据有点非主流，它是以时间倒序排列的。datetime.datetime()中的reversed=True参数是将顺序反转一次，这样就得到了我们想要的正序数据。

4 6. 第一个策略

和交易数据都有了，探险即将开始。让我们给策略加点代码，打印出每天的”收盘价”。DataSeries（交易数据的基类）对象，可以直接访问到 OHLC (开盘价、最高价、最低价、收盘价)据。这使我们打印数据很方便。

```
ort datetime  #
ort os.path  # 路径管理
ort sys  # 获取当前运行脚本的路径 (in argv[0])

入backtrader框架
ort backtrader as bt

l建策略继承bt.Strategy
ss TestStrategy(bt.Strategy):

 def log(self, txt, dt=None):
     # 记录策略的执行日志
     dt = dt or self.datas[0].datetime.date(0)
     print('%s, %s' % (dt.isoformat(), txt))

 def __init__(self):
     # 保存收盘价的引用
     self.dataclose = self.datas[0].close

 def next(self):
     # 记录收盘价
     self.log('Close, %.2f' % self.dataclose[0])


__name__ == '__main__':
 # 创建Cerebro引擎
 cerebro = bt.Cerebro()
 # Cerebro引擎在后台创建broker(经纪人)，系统默认资金量为10000

 # 为Cerebro引擎添加策略
 cerebro.addstrategy(TestStrategy)

 # 获取当前运行脚本所在目录
 modpath = os.path.dirname(os.path.abspath(sys.argv[0]))
 # 拼接加载路径
 datapath = os.path.join(modpath, '../../datas/orcl-1995-2014.txt')

 # 创建交易数据集
 data = bt.feeds.YahooFinanceCSVData(
     dataname=datapath,
     # 数据必须大于fromdate
     fromdate=datetime.datetime(2000, 1, 1),
     # 数据必须小于todate
     todate=datetime.datetime(2000, 12, 31),
     reverse=False)

 # 加载交易数据
 cerebro.adddata(data)


 # 设置投资金额100000.0
 cerebro.broker.setcash(100000.0)
 # 引擎运行前打印期出资金
 print('组合期初资金: %.2f' % cerebro.broker.getvalue())
 cerebro.run()
 # 引擎运行后打期末资金
 print('组合期末资金: %.2f' % cerebro.broker.getvalue())
```

运行后的输出结果为：

```
组合期初资金: 100000.00
2000-01-03T00:00:00, Close, 27.85
2000-01-04T00:00:00, Close, 25.39
2000-01-05T00:00:00, Close, 24.05
```

```
...
...
...
0-12-26T00:00:00, Close, 29.17
0-12-27T00:00:00, Close, 28.94
2000-12-28T00:00:00, Close, 29.29
0-12-29T00:00:00, Close, 27.41
期末资金：100000.0
```

有人说股市有风险，看起来不像啊。 一些解释说明：

在调用init时，该策略已经具有一个数据列表datas，这是标准的Python列表，可以按插入顺序数据。 列表中的第一个数据self.datas[0]是用于交易操作，并且策略中的所有元素都是由框架统时钟进行同步的。 由于只需访问收盘价数据，于是使用 self.dataclose = self.datas[0].close 一条价格数据的收盘价赋值给新变量。 系统时钟当经过一个K线柱的时候，策略的next()方法被调用一次。这一过程将一直循环，直到其他指标信号出现为止。此时，便会输出最终结果。这些，后继内容会讲到。

7. 给策略加点逻辑

K线收盘价出现三连跌，则买入。

```
ort datetime  #
ort os.path  # 路径管理
ort sys  # 获取当前运行脚本的路径 (in argv[0])

入backtrader框架
ort backtrader as bt

创建策略继承bt.Strategy
ss TestStrategy(bt.Strategy):

 def log(self, txt, dt=None):
     # 记录策略的执行日志
     dt = dt or self.datas[0].datetime.date(0)
     print('%s, %s' % (dt.isoformat(), txt))

 def __init__(self):
     # 保存收盘价的引用
     self.dataclose = self.datas[0].close

 def next(self):
     # 记录收盘价
     self.log('Close, %.2f' % self.dataclose[0])
     # 今天的收盘价 < 昨天收盘价
     if self.dataclose[0] < self.dataclose[-1]:
         # 昨天收盘价 < 前天的收盘价
         if self.dataclose[-1] < self.dataclose[-2]:
             # 买入
             self.log('买入, %.2f' % self.dataclose[0])
             self.buy()


if __name__ == '__main__':
    # 创建Cerebro引擎
    cerebro = bt.Cerebro()
    # Cerebro引擎在后台创建broker(经纪人)，系统默认资金量为10000

    # 为Cerebro引擎添加策略
    cerebro.addstrategy(TestStrategy)

    # 获取当前运行脚本所在目录
    modpath = os.path.dirname(os.path.abspath(sys.argv[0]))
    # 拼接加载路径
    datapath = os.path.join(modpath, '../../datas/orcl-1995-2014.txt')

    # 创建交易数据集
    data = bt.feeds.YahooFinanceCSVData(
        dataname=datapath,
        # 数据必须大于fromdate
        fromdate=datetime.datetime(2000, 1, 1),
        # 数据必须小于todate
        todate=datetime.datetime(2000, 12, 31),
```

```
    reverse=False)

# 加载交易数据
cerebro.adddata(data)



# 设置投资金额100000.0
cerebro.broker.setcash(100000.0)
# 引擎运行前打印期出资金
print('组合期初资金：%.2f' % cerebro.broker.getvalue())
cerebro.run()
# 引擎运行后打期末资金
print('组合期末资金：%.2f' % cerebro.broker.getvalue())
```

后的输出结果为：

```
期初资金：100000.00
0-01-03, Close, 27.85
0-01-04, Close, 25.39
0-01-05, Close, 24.05
0-01-05, 买入, 24.05
0-01-06, Close, 22.63
2000-01-06, 买入, 22.63
0-01-07, Close, 24.37



0-12-20, 买入, 26.88
0-12-21, Close, 27.82
0-12-22, Close, 30.06
0-12-26, Close, 29.17
0-12-27, Close, 28.94
0-12-27, 买入, 28.94
0-12-28, Close, 29.29
0-12-29, Close, 27.41
期末资金：99725.08
```

个买入操作被执行，所以余额在减少。 细心的朋友可能会问，买了多少？买的什么？订单怎么被执行的？BackTrader框架替我们做了这些事： 如果没有指定的话，self.datas[0]即是标的物当前交易数据。 交易数量=仓位数量，默认值等于1，后面例子我们会修改此参数。

订单被以”市价”成交了。 Broker（经纪人，之前提到过）使用了下一个交易日的开盘价，因为是broker在当前的交日易收盘后天提交的订单，下一个交易日开盘价是他接触到的第一个价格。 这里没有为订单设置佣金费，后边会加上。

4.8. 不光有买入，还得有卖出

在知道如何买入（做多）之后，需要知道如何卖出，并且还需要了解该策略是否在市场中。 Strategy类有一个变量position保存当前持有的资产数量（可以理解为金融术语中的头寸）， buy()和sell()会返回被创建的订单(尚未执行的)， 订单状态的更改将通过notify方法通知给策略Strategy。

卖出逻辑也很简单： 5个K线柱后（第6个K线柱）不管涨跌都卖。 请注意，这里没有指定具体时间，而是指定的柱的数量。一个柱可能代表1分钟、1小时、1天、1星期等等，这取决于你价格数据文件里一条数据代表的周期。 虽然我们知道每个柱代表一天，但策略并不知道。

因为买入的时候，用的是交易日，所以这里应翻译为：5个交易日（第6个交易日）不管涨跌都卖。

另外，当还有头寸的时候，就不再买入了。

> 注意： 没有将柱的下标传给next()方法，怎么知道已经经过了5个柱了呢？ 这里用了Python的len()方法获取它Line数据的长度。 交易发生时记下它的长度，后边比较大小，看是否经过了5个柱。

```
import datetime  #
import os.path  # 路径管理
import sys  # 获取当前运行脚本的路径 (in argv[0])

#导入backtrader框架
```

Table of Content


```
import backtrader as bt

# 创建策略继承bt.Strategy
class TestStrategy(bt.Strategy):

    def log(self, txt, dt=None):
        # 记录策略的执行日志
        dt = dt or self.datas[0].datetime.date(0)
        print('%s, %s' % (dt.isoformat(), txt))

    def __init__(self):
        # 保存收盘价的引用
        self.dataclose = self.datas[0].close
        # 跟踪挂单
        self.order = None


    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            # broker 提交/接受了，买/卖订单则什么都不做
            return

        # 检查一个订单是否完成
        # 注意：当资金不足时，broker会拒绝订单
        if order.status in [order.Completed]:
            if order.isbuy():
                self.log('已买入, %.2f' % order.executed.price)
            elif order.issell():
                self.log('已卖出, %.2f' % order.executed.price)

            # 记录当前交易数量
            self.bar_executed = len(self)

        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log('订单取消/保证金不足/拒绝')

        # 其他状态记录为：无挂起订单
        self.order = None

    def next(self):
        # 记录收盘价
        self.log('Close, %.2f' % self.dataclose[0])

        # 如果有订单正在挂起，不操作
        if self.order:
            return

        # 如果没有持仓则买入
        if not self.position:
            # 今天的收盘价 < 昨天收盘价
            if self.dataclose[0] < self.dataclose[-1]:
                # 昨天收盘价 < 前天的收盘价
                if self.dataclose[-1] < self.dataclose[-2]:
                    # 买入
                    self.log('买入, %.2f' % self.dataclose[0])
                     # 跟踪订单避免重复
                    self.order = self.buy()
        else:
            # 如果已经持仓，且当前交易数据量在买入后5个单位后
            if len(self) >= (self.bar_executed + 5):
                # 全部卖出
                self.log('卖出, %.2f' % self.dataclose[0])
                # 跟踪订单避免重复
                self.order = self.sell()



if __name__ == '__main__':
    # 创建Cerebro引擎
    cerebro = bt.Cerebro()
    # Cerebro引擎在后台创建broker(经纪人)，系统默认资金量为10000

    # 为Cerebro引擎添加策略
    cerebro.addstrategy(TestStrategy)

    # 获取当前运行脚本所在目录
    modpath = os.path.dirname(os.path.abspath(sys.argv[0]))
    # 拼接加载路径
```

```
datapath = os.path.join(modpath, '../../datas/orcl-1995-2014.txt')

# 创建交易数据集
data = bt.feeds.YahooFinanceCSVData(
    dataname=datapath,
    # 数据必须大于fromdate
    fromdate=datetime.datetime(2000, 1, 1),
    # 数据必须小于todate
    todate=datetime.datetime(2000, 12, 31),
    reverse=False)

# 加载交易数据
cerebro.adddata(data)



# 设置投资金额100000.0
cerebro.broker.setcash(100000.0)
# 引擎运行前打印期出资金
print('组合期初资金: %.2f' % cerebro.broker.getvalue())
cerebro.run()
# 引擎运行后打期末资金
print('组合期末资金: %.2f' % cerebro.broker.getvalue())
```

运行后的输出结果为：

```
组合期初资金: 100000.00
2000-01-03T00:00:00, Close, 27.85
2000-01-04T00:00:00, Close, 25.39
2000-01-05T00:00:00, Close, 24.05
2000-01-05T00:00:00, 买入, 24.05
2000-01-06T00:00:00, 已买入, 23.61
2000-01-06T00:00:00, Close, 22.63
2000-01-07T00:00:00, Close, 24.37
2000-01-10T00:00:00, Close, 27.29
2000-01-11T00:00:00, Close, 26.49
2000-01-12T00:00:00, Close, 24.90
2000-01-13T00:00:00, Close, 24.77
2000-01-13T00:00:00, 卖出, 24.77
2000-01-14T00:00:00, 已卖出, 25.70
2000-01-14T00:00:00, Close, 25.18
...
...
...
2000-12-15T00:00:00, 卖出, 26.93
2000-12-18T00:00:00, 已卖出, 28.29
2000-12-18T00:00:00, Close, 30.18
2000-12-19T00:00:00, Close, 28.88
2000-12-20T00:00:00, Close, 26.88
2000-12-20T00:00:00, 买入, 26.88
2000-12-21T00:00:00, 已买入, 26.23
2000-12-21T00:00:00, Close, 27.82
2000-12-22T00:00:00, Close, 30.06
2000-12-26T00:00:00, Close, 29.17
2000-12-27T00:00:00, Close, 28.94
2000-12-28T00:00:00, Close, 29.29
2000-12-29T00:00:00, Close, 27.41
2000-12-29T00:00:00, 卖出, 27.41
组合期末资金: 100018.53
```

竟然盈利了，是不是哪里出错了?

4.9. 经纪人说：手续费呢?

这比费用叫做佣金。让我们设定一个常见的费率0.1%，买卖都要收（经纪人就是这么贪）。一行代码就能搞定:

```
cerebro.broker.setcommission(commission=0.001) # 0.001即是0.1%
```

让我看看加不加手续费，结果到底有什么区别。

```
import datetime  #
import os.path  # 路径管理
import sys  # 获取当前运行脚本的路径 (in argv[0])
```

```
#导入backtrader框架
import backtrader as bt

    创建策略继承bt.Strategy
class TestStrategy(bt.Strategy):

    def log(self, txt, dt=None):
        # 记录策略的执行日志
        dt = dt or self.datas[0].datetime.date(0)
        print('%s, %s' % (dt.isoformat(), txt))

    def __init__(self):
        # 保存收盘价的引用
        self.dataclose = self.datas[0].close
        # 跟踪挂单
        self.order = None
        # 买入价格和手续费
        self.buyprice = None
        self.buycomm = None

    # 订单状态通知，买入卖出都是下单
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            # broker 提交/接受了，买/卖订单则什么都不做
            return

        # 检查一个订单是否完成
        # 注意：当资金不足时，broker会拒绝订单
        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(
                    '已买入，价格：%.2f，费用：%.2f，佣金 %.2f' %
                    (order.executed.price,
                     order.executed.value,
                     order.executed.comm))

                self.buyprice = order.executed.price
                self.buycomm = order.executed.comm
            elif order.issell():
                self.log('已卖出，价格：%.2f，费用：%.2f，佣金 %.2f' %
                         (order.executed.price,
                          order.executed.value,
                          order.executed.comm))
            # 记录当前交易数量
            self.bar_executed = len(self)

        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log('订单取消/保证金不足/拒绝')

        # 其他状态记录为：无挂起订单
        self.order = None

    # 交易状态通知，一买一卖算交易
    def notify_trade(self, trade):
        if not trade.isclosed:
            return
        self.log('交易利润，毛利润 %.2f，净利润 %.2f' %
                 (trade.pnl, trade.pnlcomm))

    def next(self):
        # 记录收盘价
        self.log('Close, %.2f' % self.dataclose[0])

        # 如果有订单正在挂起，不操作
        if self.order:
            return

        # 如果没有持仓则买入
        if not self.position:
            # 今天的收盘价 < 昨天收盘价
            if self.dataclose[0] < self.dataclose[-1]:
                # 昨天收盘价 < 前天的收盘价
                if self.dataclose[-1] < self.dataclose[-2]:
                    # 买入
                    self.log('买入单，%.2f' % self.dataclose[0])
                     # 跟踪订单避免重复
                    self.order = self.buy()
```

```
        else:
            # 如果已经持仓，且当前交易数据量在买入后5个单位后
            if len(self) >= (self.bar_executed + 5):
                # 全部卖出
                self.log('卖出单, %.2f' % self.dataclose[0])
                # 跟踪订单避免重复
                self.order = self.sell()



if __name__ == '__main__':
    # 创建Cerebro引擎
    cerebro = bt.Cerebro()
    # Cerebro引擎在后台创建broker(经纪人)，系统默认资金量为10000

    # 为Cerebro引擎添加策略
    cerebro.addstrategy(TestStrategy)

    # 获取当前运行脚本所在目录
    modpath = os.path.dirname(os.path.abspath(sys.argv[0]))
    # 拼接加载路径
    datapath = os.path.join(modpath, '../../datas/orcl-1995-2014.txt')

    # 创建交易数据集
    data = bt.feeds.YahooFinanceCSVData(
        dataname=datapath,
        # 数据必须大于fromdate
        fromdate=datetime.datetime(2000, 1, 1),
        # 数据必须小于todate
        todate=datetime.datetime(2000, 12, 31),
        reverse=False)

    # 加载交易数据
    cerebro.adddata(data)


    # 设置投资金额100000.0
    cerebro.broker.setcash(100000.0)
    # 设置佣金为0.001,除以100去掉%号
    cerebro.broker.setcommission(commission=0.001)

    # 引擎运行前打印期出资金
    print('组合期初资金: %.2f' % cerebro.broker.getvalue())
    cerebro.run()
    # 引擎运行后打期末资金
    print('组合期末资金: %.2f' % cerebro.broker.getvalue())
```

运行后输出

```
组合期初资金: 100000.00
2000-01-03T00:00:00, Close, 27.85
2000-01-04T00:00:00, Close, 25.39
2000-01-05T00:00:00, Close, 24.05
2000-01-05T00:00:00, 买入单, 24.05
2000-01-06T00:00:00, 已买入, 价格: 23.61, 费用: 23.61, 佣金 0.02
2000-01-06T00:00:00, Close, 22.63
2000-01-07T00:00:00, Close, 24.37
2000-01-10T00:00:00, Close, 27.29
2000-01-11T00:00:00, Close, 26.49
2000-01-12T00:00:00, Close, 24.90
2000-01-13T00:00:00, Close, 24.77
2000-01-13T00:00:00, 卖出单, 24.77
2000-01-14T00:00:00, 已卖出, 价格: 25.70, 费用: 25.70, 佣金 0.03
2000-01-14T00:00:00, 操作利润, 毛利润 2.09, 净利润 2.04
2000-01-14T00:00:00, Close, 25.18
...
...
...
2000-12-15T00:00:00, 卖出单, 26.93
2000-12-18T00:00:00, 已卖出, 价格: 28.29, 费用: 28.29, 佣金 0.03
2000-12-18T00:00:00, 操作利润, 毛利润 -0.06, 净利润 -0.12
2000-12-18T00:00:00, Close, 30.18
2000-12-19T00:00:00, Close, 28.88
2000-12-20T00:00:00, Close, 26.88
2000-12-20T00:00:00, 买入单, 26.88
2000-12-21T00:00:00, 已买入, 价格: 26.23, 费用: 26.23, 佣金 0.03
2000-12-21T00:00:00, Close, 27.82
```

```
2000-12-22T00:00:00, Close, 30.06
2000-12-26T00:00:00, Close, 29.17
2000-12-27T00:00:00, Close, 28.94
  0-12-28T00:00:00, Close, 29.29
  0-12-29T00:00:00, Close, 27.41
2000-12-29T00:00:00, 卖出单, 27.41
期末资金: 100016.98
```

！竟然还是盈利。 在继续之前，让我们看看这些带有盈亏的操作。

```
  0-01-14T00:00:00, 操作利润, 毛利润 2.09, 净利润 2.04
  0-02-07T00:00:00, 操作利润, 毛利润 3.68, 净利润 3.63
  0-02-28T00:00:00, 操作利润, 毛利润 4.48, 净利润 4.42
  0-03-13T00:00:00, 操作利润, 毛利润 3.48, 净利润 3.41
  0-03-22T00:00:00, 操作利润, 毛利润 -0.41, 净利润 -0.49
  0-04-07T00:00:00, 操作利润, 毛利润 2.45, 净利润 2.37
  0-04-20T00:00:00, 操作利润, 毛利润 -1.95, 净利润 -2.02
  0-05-02T00:00:00, 操作利润, 毛利润 5.46, 净利润 5.39
  0-05-11T00:00:00, 操作利润, 毛利润 -3.74, 净利润 -3.81
  0-05-30T00:00:00, 操作利润, 毛利润 -1.46, 净利润 -1.53
  0-07-05T00:00:00, 操作利润, 毛利润 -1.62, 净利润 -1.69
  0-07-14T00:00:00, 操作利润, 毛利润 2.08, 净利润 2.01
2000-07-28T00:00:00, 操作利润, 毛利润 0.14, 净利润 0.07
  0-08-08T00:00:00, 操作利润, 毛利润 4.36, 净利润 4.29
  0-08-21T00:00:00, 操作利润, 毛利润 1.03, 净利润 0.95
  0-09-15T00:00:00, 操作利润, 毛利润 -4.26, 净利润 -4.34
  0-09-27T00:00:00, 操作利润, 毛利润 1.29, 净利润 1.22
  0-10-13T00:00:00, 操作利润, 毛利润 -2.98, 净利润 -3.04
  0-10-26T00:00:00, 操作利润, 毛利润 3.01, 净利润 2.95
  0-11-06T00:00:00, 操作利润, 毛利润 -3.59, 净利润 -3.65
  0-11-16T00:00:00, 操作利润, 毛利润 1.28, 净利润 1.23
  0-12-01T00:00:00, 操作利润, 毛利润 2.59, 净利润 2.54
  0-12-18T00:00:00, 操作利润, 毛利润 -0.06, 净利润 -0.12
```

益加起来是:15.83 但系统最后的余额是:100016.98 很明显 15.83 不等于 16.98。其实没错，益指的是已经落到口袋里的钱。

差异的原因是，最后一天还持有头寸。其实卖单已经发出去了，但还没来得及执行。Broker净收益率是按照2000-12-29收盘价算的。实际应该按下一个交易日2001-01-02价格算。

```
2001-01-02T00:00:00, 已卖出, 价格: 27.87, 费用: 27.87, 佣金 0.03
2001-01-02T00:00:00, 操作利润, 毛利润 1.64, 净利润 1.59
2001-01-02T00:00:00, Close, 24.87
2001-01-02T00:00:00, 买入单, 24.87
组合期末资金: 100017.41
```

加起来之前的净收益:5.83 + 1.59 = 17.42，这个净收益率17.42和最后的余额100017.41就对上了（忽略小数点误差）。

4.10. 自定义策略:技术指标参数

在实战中，一般不将参数写死到策略中。Parameters（参数）就是用来处理这个的。 参数的定义像这样:

```
# 参数的定义像这样:
params = (('myparam', 27), ('exitbars', 5),)
```

这个 tuple 嵌套看着不方便，格式化一下:

```
params = (
    ('myparam', 27),
    ('exitbars', 5),
)
```

将策略添加到引擎的时候，可以指定刚才定义的参数:

```
# 指定参数
cerebro.addstrategy(TestStrategy, myparam=20, exitbars=7)
```

> 注意： 下面的 setsizing 方法已经被弃用。这里还保留是因为还有一些老示例在用。方法已改为下面这种: cerebro.addsizer(bt.sizers.FixedSize, stake=10) 请参考 sizers 章节。

略类中使用买卖数量参数很容易，它们被保存在 “params” 参数里。例如，参数已经传入，在策略类里的 **init** 方法中这样调用就可以了:

```
据传入的参加设置买卖数量
f.sizer.setsizing(self.params.stake)
```

以直接将买卖数量传入buy和sell方法。 卖出的逻辑改为:

```
经持有，可以卖出了
len(self) >= (self.bar_executed + self.params.exitbars)
```

修改为:

```
ort datetime  #
ort os.path  # 路径管理
ort sys  # 获取当前运行脚本的路径 (in argv[0])

#导入backtrader框架
ort backtrader as bt

建策略继承bt.Strategy
ss TestStrategy(bt.Strategy):
 params = (
     # 持仓够5个单位就卖出
     ('exitbars', 5),
 )

 def log(self, txt, dt=None):
     # 记录策略的执行日志
     dt = dt or self.datas[0].datetime.date(0)
     print('%s, %s' % (dt.isoformat(), txt))

 def __init__(self):
     # 保存收盘价的引用
     self.dataclose = self.datas[0].close
     # 跟踪挂单
     self.order = None
     # 买入价格和手续费
     self.buyprice = None
     self.buycomm = None

 # 订单状态通知，买入卖出都是下单
 def notify_order(self, order):
     if order.status in [order.Submitted, order.Accepted]:
         # broker 提交/接受了，买/卖订单则什么都不做
         return

     # 检查一个订单是否完成
     # 注意：当资金不足时，broker会拒绝订单
     if order.status in [order.Completed]:
         if order.isbuy():
             self.log(
                 '已买入，价格：%.2f，费用：%.2f，佣金 %.2f' %
                 (order.executed.price,
                  order.executed.value,
                  order.executed.comm))

             self.buyprice = order.executed.price
             self.buycomm = order.executed.comm
         elif order.issell():
             self.log('已卖出，价格：%.2f，费用：%.2f，佣金 %.2f' %
                      (order.executed.price,
                       order.executed.value,
                       order.executed.comm))
         # 记录当前交易数量
         self.bar_executed = len(self)

     elif order.status in [order.Canceled, order.Margin, order.Rejected]:
         self.log('订单取消/保证金不足/拒绝')

     # 其他状态记录为：无挂起订单
     self.order = None
```

```
# 交易状态通知，一买一卖算交易
def notify_trade(self, trade):
    if not trade.isclosed:
        return
    self.log('交易利润, 毛利润 %.2f, 净利润 %.2f' %
             (trade.pnl, trade.pnlcomm))

def next(self):
    # 记录收盘价
    self.log('Close, %.2f' % self.dataclose[0])

    # 如果有订单正在挂起，不操作
    if self.order:
        return

    # 如果没有持仓则买入
    if not self.position:
        # 今天的收盘价 < 昨天收盘价
        if self.dataclose[0] < self.dataclose[-1]:
            # 昨天收盘价 < 前天的收盘价
            if self.dataclose[-1] < self.dataclose[-2]:
                # 买入
                self.log('买入单, %.2f' % self.dataclose[0])
                 # 跟踪订单避免重复
                self.order = self.buy()
    else:
        # 如果已经持仓，且当前交易数据量在买入后5个单位后
        # 此处做了更新将5替换为参数
        if len(self) >= (self.bar_executed + self.params.exitbars):
            # 全部卖出
            self.log('卖出单, %.2f' % self.dataclose[0])
            # 跟踪订单避免重复
            self.order = self.sell()



__name__ == '__main__':
 # 创建Cerebro引擎
 cerebro = bt.Cerebro()
 # Cerebro引擎在后台创建broker(经纪人)，系统默认资金量为10000

 # 为Cerebro引擎添加策略
 cerebro.addstrategy(TestStrategy)

 # 获取当前运行脚本所在目录
 modpath = os.path.dirname(os.path.abspath(sys.argv[0]))
 # 拼接加载路径
 datapath = os.path.join(modpath, '../../datas/orcl-1995-2014.txt')

 # 创建交易数据集
 data = bt.feeds.YahooFinanceCSVData(
     dataname=datapath,
     # 数据必须大于fromdate
     fromdate=datetime.datetime(2000, 1, 1),
     # 数据必须小于todate
     todate=datetime.datetime(2000, 12, 31),
     reverse=False)

 # 加载交易数据
 cerebro.adddata(data)


 # 设置投资金额1000000.0
 cerebro.broker.setcash(1000000.0)

 # 每笔交易使用固定交易量
 cerebro.addsizer(bt.sizers.FixedSize, stake=10)
 # 设置佣金为0.001,除以100去掉%号
 cerebro.broker.setcommission(commission=0.001)

 # 引擎运行前打印期出资金
 print('组合期初资金: %.2f' % cerebro.broker.getvalue())
 cerebro.run()
 # 引擎运行后打期末资金
 print('组合期末资金: %.2f' % cerebro.broker.getvalue())
```

运行后的输出为:

```
组合期初资金: 1000000.00

2000-01-03T00:00:00, Close, 27.85
2000-01-04T00:00:00, Close, 25.39
2000-01-05T00:00:00, Close, 24.05
2000-01-05T00:00:00, 买单, 24.05
2000-01-06T00:00:00, 已买入, 交易量 10, 价格: 23.61, 费用: 236.10, 佣金 0.24
2000-01-06T00:00:00, Close, 22.63



2000-12-20T00:00:00, 买入单, 26.88
2000-12-21T00:00:00, 已买入, 交易量 10, 价格: 26.23, 费用: 262.30, 佣金 0.26
2000-12-21T00:00:00, Close, 27.82
2000-12-22T00:00:00, Close, 30.06
2000-12-26T00:00:00, Close, 29.17
2000-12-27T00:00:00, Close, 28.94
2000-12-28T00:00:00, Close, 29.29
2000-12-29T00:00:00, Close, 27.41
2000-12-29T00:00:00, 卖出单, 27.41
组合期末资金: 100169.80
```

> 显示改变已生效，输出中显示了买卖数量。 买卖数量改为了原来的10倍，盈亏也变为了原来倍，变为了169.80 。

I1. 添加技术指标

我提到过indicators（技术指标），下一步就该添加他们了，要做的肯定比“三连跌”这种复杂借用PyAlgoTrade这个框架的一个使用移动平均线的例子：

收盘价高于平均价的时候，以市价买入
持有仓位的时候，如果收盘价低于平均价，卖出
只有一个待执行的订单

大多数代码不用改变，在 **init** 方法中加入移动平均线的实例化:

```
# 加入移动平均线
self.sma = bt.indicators.MovingAverageSimple(self.datas[0], period=self.params.maperiod)
```

当然买入卖出的逻辑依赖平均价，具体代码如下。

> 注意： 起始金额为1000元，无手续费，这和 PyAlgoTrade 保持一致。

```
import datetime  #
import os.path  # 路径管理
import sys  # 获取当前运行脚本的路径 (in argv[0])

#导入backtrader框架
import backtrader as bt

# 创建策略继承bt.Strategy
class TestStrategy(bt.Strategy):
    params = (
        # 均线参数设置15天，15日均线
        ('maperiod', 15),
    )

    def log(self, txt, dt=None):
        # 记录策略的执行日志
        dt = dt or self.datas[0].datetime.date(0)
        print('%s, %s' % (dt.isoformat(), txt))

    def __init__(self):
        # 保存收盘价的引用
        self.dataclose = self.datas[0].close
        # 跟踪挂单
        self.order = None
        # 买入价格和手续费
        self.buyprice = None
        self.buycomm = None
```

```
        # 加入均线指标
        self.sma = bt.indicators.SimpleMovingAverage(self.datas[0], period=self.params.m


    # 订单状态通知，买入卖出都是下单
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            # broker 提交/接受了，买/卖订单则什么都不做
            return

        # 检查一个订单是否完成
        # 注意：当资金不足时，broker会拒绝订单
        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(
                    '已买入，价格：%.2f，费用：%.2f，佣金 %.2f' %
                    (order.executed.price,
                     order.executed.value,
                     order.executed.comm))

                self.buyprice = order.executed.price
                self.buycomm = order.executed.comm
            elif order.issell():
                self.log('已卖出，价格：%.2f，费用：%.2f，佣金 %.2f' %
                         (order.executed.price,
                          order.executed.value,
                          order.executed.comm))
            # 记录当前交易数量
            self.bar_executed = len(self)

        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log('订单取消/保证金不足/拒绝')

        # 其他状态记录为：无挂起订单
        self.order = None

    # 交易状态通知，一买一卖算交易
    def notify_trade(self, trade):
        if not trade.isclosed:
            return
        self.log('交易利润，毛利润 %.2f，净利润 %.2f' %
                 (trade.pnl, trade.pnlcomm))

    def next(self):
        # 记录收盘价
        self.log('Close，%.2f' % self.dataclose[0])

        # 如果有订单正在挂起，不操作
        if self.order:
            return

        # 如果没有持仓则买入
        if not self.position:
            # 今天的收盘价在均线价格之上
            if self.dataclose[0] > self.sma[0]:
                # 买入
                self.log('买入单，%.2f' % self.dataclose[0])
                    # 跟踪订单避免重复
                self.order = self.buy()
        else:
            # 如果已经持仓，收盘价在均线价格之下
            if self.dataclose[0] < self.sma[0]:
                # 全部卖出
                self.log('卖出单，%.2f' % self.dataclose[0])
                # 跟踪订单避免重复
                self.order = self.sell()



if __name__ == '__main__':
    # 创建Cerebro引擎
    cerebro = bt.Cerebro()
    # Cerebro引擎在后台创建broker(经纪人)，系统默认资金量为10000

    # 为Cerebro引擎添加策略
    cerebro.addstrategy(TestStrategy)

    # 获取当前运行脚本所在目录
```

```
modpath = os.path.dirname(os.path.abspath(sys.argv[0]))
# 拼接加载路径
datapath = os.path.join(modpath, '../../datas/orcl-1995-2014.txt')

# 创建交易数据集
data = bt.feeds.YahooFinanceCSVData(
    dataname=datapath,
    # 数据必须大于fromdate
    fromdate=datetime.datetime(2000, 1, 1),
    # 数据必须小于todate
    todate=datetime.datetime(2000, 12, 31),
    reverse=False)

# 加载交易数据
cerebro.adddata(data)



# 设置投资金额1000.0
cerebro.broker.setcash(1000.0)

# 每笔交易使用固定交易量
cerebro.addsizer(bt.sizers.FixedSize, stake=10)
# 设置佣金为0.0
cerebro.broker.setcommission(commission=0.0)

# 引擎运行前打印期出资金
print('组合期初资金: %.2f' % cerebro.broker.getvalue())
cerebro.run()
# 引擎运行后打期末资金
print('组合期末资金: %.2f' % cerebro.broker.getvalue())
```

们仔细地看一下出现在下面日志中的第一条记录： 不再是新千年的第一个交易日2000-01-03 变成了2000-01-24，怎么回事呢？ 是因为，框架根据新代码做出了改变： 在策略中我们加入动平均技术指标。 移动平均需要有个均线周期参数，程序根据这个参数回看计算前边的X条价据然后进行开仓判断，例子中周期是15。 2000-01-24就是第15天 backtrader 框架假定策略加个技术指标是有正当理由的，比如做开平仓的决策。 框架不会在数据没到位的时候就进行下 。 在技术指标产生第一条数据之后，next方法第一个被调用。 在示例中只有一个技术指标，其买策略支持添加多个技术指标。

运行后的输出为:

```
组合期初资金: 1000.00
2000-01-24T00:00:00, Close, 25.55
2000-01-25T00:00:00, Close, 26.61
2000-01-25T00:00:00, 买入单, 26.61
2000-01-26T00:00:00, 已买入, 数量 10, 价格: 26.76, 费用: 267.60, 佣金 0.00
2000-01-26T00:00:00, Close, 25.96
2000-01-27T00:00:00, Close, 24.43
2000-01-27T00:00:00, 卖出单, 24.43
2000-01-28T00:00:00, 已卖出, 数量 10, 价格: 24.28, 费用: 242.80, 佣金 0.00
2000-01-28T00:00:00, 操作利润, 毛利润 -24.80, 净利润 -24.80
2000-01-28T00:00:00, Close, 22.34
2000-01-31T00:00:00, Close, 23.55
2000-02-01T00:00:00, Close, 25.46
2000-02-02T00:00:00, Close, 25.61
2000-02-02T00:00:00, 买入单, 25.61
2000-02-03T00:00:00, 已买入, 数量 10, 价格: 26.11, 费用: 261.10, 佣金 0.00
...
...
...
2000-12-20T00:00:00, 卖出单, 26.88
2000-12-21T00:00:00, 已卖出, 数量 10, 价格: 26.23, 费用: 262.30, 佣金 0.00
2000-12-21T00:00:00, 操作利润, 毛利润 -20.60, 净利润 -20.60
2000-12-21T00:00:00, Close, 27.82
2000-12-21T00:00:00, 买入单, 27.82
2000-12-22T00:00:00, 已买入, 数量 10, 价格: 28.65, 费用: 286.50, 佣金 0.00
2000-12-22T00:00:00, Close, 30.06
2000-12-26T00:00:00, Close, 29.17
2000-12-27T00:00:00, Close, 28.94
2000-12-28T00:00:00, Close, 29.29
2000-12-29T00:00:00, Close, 27.41
2000-12-29T00:00:00, 卖出单, 27.41
组合期末资金: 973.90
```

一个盈利系统被改变之后开始亏损了，还是在手续费率设置为0的情况下。看来简单添加一个技术指标并不是万能的。

> 意： 同样的交易逻辑和数据，和PyAlgoTrade输出的结果并不完全一致，当然只是轻微不一致。最可疑的原因是因为：小数点 处理”调整后价格”（分红、拆股后调整）时，yAlgoTrade并不对小数点进行四舍五入。 在对价格进行调整后，backtrader的数据引擎将ahoo价格数据的价格小数点缩减到2位。虽然输出看起来差不多，但积少成多结果就不同。 将价格小数点缩减到2位是合理的，一般交易所只允许价格保留小数点后面2位。
>
> 1.8.11.99 版本开始，backtrader的Yahoo数据引擎可以设置是否做小数点位数保留，还可设置保留多少位。

12. 可视化:绘图

日志虽然能看到细节，但人们还是喜欢看可视化的东西，所以有必要将结果绘制成图表。 绘容易使用，只需添加一行代码:

```
# 绘制图像
ebro.plot()
```

代码要放在 cerebro.run() 之后。 为方便使用，框架做了下面这些自动化的事情： 将添加第二数移动平均线，默认将使用数据进行绘制（就像第1条）。 将添加第三条加权移动平均线，在区域绘制（也许看起来不合理）。 将添加一条Stochastic（慢），使用默认参数。 将添加一ACD，使用默认参数。 将添加一条RSI指标，使用默认参数。 将添加一条RSI指标的简单移动线，使用默认参数（将和RSI一起被绘制）。 将添加一条ATR指标，修改了默认参数以避免被。

添加的这些指标，等于在策略类的 **init** 方法中添加了以下语句:

```
要绘制的指标
indicators.ExponentialMovingAverage(self.datas[0], period=25)
bt.indicators.WeightedMovingAverage(self.datas[0], period=25).subplot = True
bt.indicators.StochasticSlow(self.datas[0])
bt.indicators.MACDHisto(self.datas[0])
rsi = bt.indicators.RSI(self.datas[0])
bt.indicators.SmoothedMovingAverage(rsi, period=10)
bt.indicators.ATR(self.datas[0]).plot = False
```

> 注意： 即使指标没有被显式地声明为成员变量（如 self.sma = MovingAverageSimple…），它们还是会被自动注册到策略类中，并影响开始执行 next 的最小周期，而且会被绘制。 在例子中，只有RSI的指标被赋予了一个rsi的变量，供后边为它创建移动平均线使用。

现在程序变成了这样:

```
import datetime  #
import os.path  # 路径管理
import sys  # 获取当前运行脚本的路径 (in argv[0])

#导入backtrader框架
import backtrader as bt

# 创建策略继承bt.Strategy
class TestStrategy(bt.Strategy):
    params = (
        # 均线参数设置15天，15日均线
        ('maperiod', 15),
    )

    def log(self, txt, dt=None):
        # 记录策略的执行日志
        dt = dt or self.datas[0].datetime.date(0)
        print('%s, %s' % (dt.isoformat(), txt))

    def __init__(self):
        # 保存收盘价的引用
        self.dataclose = self.datas[0].close
        # 跟踪挂单
        self.order = None
```

```
    # 买入价格和手续费
    self.buyprice = None
    self.buycomm = None
    # 加入均线指标
    self.sma = bt.indicators.SimpleMovingAverage(self.datas[0], period=self.params.m

    # 绘制图形时候用到的指标
    bt.indicators.ExponentialMovingAverage(self.datas[0], period=25)
    bt.indicators.WeightedMovingAverage(self.datas[0], period=25,subplot=True)
    bt.indicators.StochasticSlow(self.datas[0])
    bt.indicators.MACDHisto(self.datas[0])
    rsi = bt.indicators.RSI(self.datas[0])
    bt.indicators.SmoothedMovingAverage(rsi, period=10)
    bt.indicators.ATR(self.datas[0], plot=False)


# 订单状态通知，买入卖出都是下单
def notify_order(self, order):
    if order.status in [order.Submitted, order.Accepted]:
        # broker 提交/接受了，买/卖订单则什么都不做
        return

    # 检查一个订单是否完成
    # 注意：当资金不足时，broker会拒绝订单
    if order.status in [order.Completed]:
        if order.isbuy():
            self.log(
                '已买入，价格：%.2f，费用：%.2f，佣金 %.2f' %
                (order.executed.price,
                 order.executed.value,
                 order.executed.comm))

            self.buyprice = order.executed.price
            self.buycomm = order.executed.comm
        elif order.issell():
            self.log('已卖出，价格：%.2f，费用：%.2f，佣金 %.2f' %
                     (order.executed.price,
                      order.executed.value,
                      order.executed.comm))
        # 记录当前交易数量
        self.bar_executed = len(self)

    elif order.status in [order.Canceled, order.Margin, order.Rejected]:
        self.log('订单取消/保证金不足/拒绝')

    # 其他状态记录为：无挂起订单
    self.order = None

# 交易状态通知，一买一卖算交易
def notify_trade(self, trade):
    if not trade.isclosed:
        return
    self.log('交易利润，毛利润 %.2f，净利润 %.2f' %
             (trade.pnl, trade.pnlcomm))

def next(self):
    # 记录收盘价
    self.log('Close, %.2f' % self.dataclose[0])

    # 如果有订单正在挂起，不操作
    if self.order:
        return

    # 如果没有持仓则买入
    if not self.position:
        # 今天的收盘价在均线价格之上
        if self.dataclose[0] > self.sma[0]:
            # 买入
            self.log('买入单，%.2f' % self.dataclose[0])
                # 跟踪订单避免重复
            self.order = self.buy()
    else:
        # 如果已经持仓，收盘价在均线价格之下
        if self.dataclose[0] < self.sma[0]:
            # 全部卖出
            self.log('卖出单，%.2f' % self.dataclose[0])
            # 跟踪订单避免重复
            self.order = self.sell()
```

```
__name__ == '__main__':
# 创建Cerebro引擎
cerebro = bt.Cerebro()
# Cerebro引擎在后台创建broker(经纪人)，系统默认资金量为10000

# 为Cerebro引擎添加策略
cerebro.addstrategy(TestStrategy)

# 获取当前运行脚本所在目录
modpath = os.path.dirname(os.path.abspath(sys.argv[0]))
# 拼接加载路径
datapath = os.path.join(modpath, '../../datas/orcl-1995-2014.txt')

# 创建交易数据集
data = bt.feeds.YahooFinanceCSVData(
    dataname=datapath,
    # 数据必须大于fromdate
    fromdate=datetime.datetime(2000, 1, 1),
    # 数据必须小于todate
    todate=datetime.datetime(2000, 12, 31),
    reverse=False)

# 加载交易数据
cerebro.adddata(data)


# 设置投资金额1000.0
cerebro.broker.setcash(1000.0)

# 每笔交易使用固定交易量
cerebro.addsizer(bt.sizers.FixedSize, stake=10)
# 设置佣金为0.0
cerebro.broker.setcommission(commission=0.0)

# 引擎运行前打印期出资金
print('组合期初资金：%.2f' % cerebro.broker.getvalue())
cerebro.run()
# 引擎运行后打期末资金
print('组合期末资金：%.2f' % cerebro.broker.getvalue())

# 绘制图像
cerebro.plot()
```

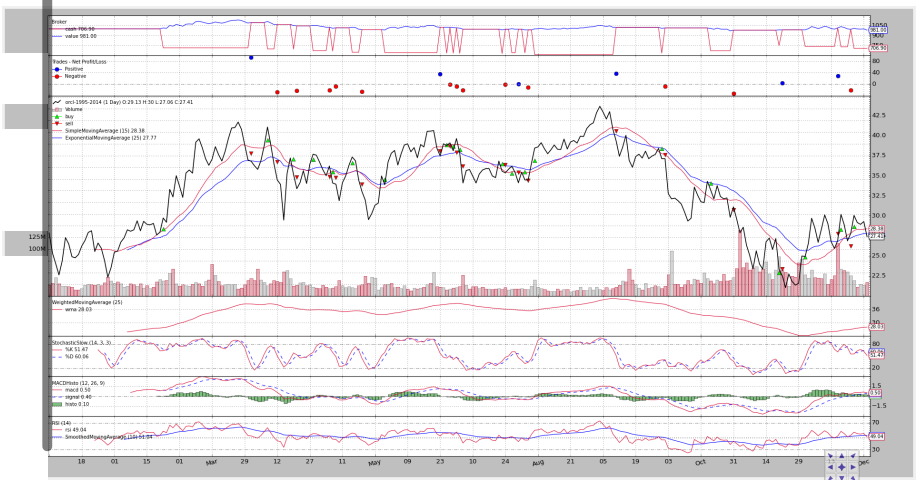
执行后的输出结果为:

```
组合期初资金：1000.00
2000-02-18T00:00:00, Close, 27.61
2000-02-22T00:00:00, Close, 27.97
2000-02-22T00:00:00, 买入单, 27.97
2000-02-23T00:00:00, 已买入, 数量 10, 价格：28.38, 费用：283.80, 佣金 0.00
2000-02-23T00:00:00, Close, 29.73
...
...
...
2000-12-21T00:00:00, 买入单, 27.82
2000-12-22T00:00:00, 已买入, 数量 10, 价格：28.65, 费用：286.50, 佣金 0.00
2000-12-22T00:00:00, Close, 30.06
2000-12-26T00:00:00, Close, 29.17
2000-12-27T00:00:00, Close, 28.94
2000-12-28T00:00:00, Close, 29.29
2000-12-29T00:00:00, Close, 27.41
2000-12-29T00:00:00, SELL CREATE, 27.41
组合期末资金：981.00
```

虽然策略逻辑没有变，但回测结果却变了。这是由于bar的数量发生了变化。

> 注意： 前面提到过，框架会等待所有指标数据到位之后，才会运行next函数。 在上例中，MACD是最后一个数据到位的指标（它的3条线都完成了输出）。 所以第一笔下单已经不是2000年1月份了，而是2000年2月份末。

图表如下:



13. 参数调优

交易书籍都会说每个市场、每只股票（或期货等等）都有不同的节奏，也就是说没有一个参数应所有。在之前的例子里，策略里使用的默认参数是15。这个参数可以被更换并进行测试，估什么值更适合于市场。

:意： 大量文献讨论了关于优化的优缺点。一般建议都会指向同一方向：不要过度优化。如:策略不理想，而在拟合上下功夫，则可能产生一个在回测数据上非常优秀的参数，但这个:数在将来表现可能并不好。

了代码，以测试移动平均线的最优周期参数。为保持清新，删除了所有买入、卖出的输出。后的例子：

```
   ort datetime  #
   ort os.path  # 路径管理
import sys  # 获取当前运行脚本的路径 (in argv[0])

#导入backtrader框架
import backtrader as bt

# 创建策略继承bt.Strategy
class TestStrategy(bt.Strategy):
    params = (
        # 均线参数设置15天，15日均线
        ('maperiod', 15),
        ('printlog', False),
    )

    def log(self, txt, dt=None):
        # 记录策略的执行日志
        dt = dt or self.datas[0].datetime.date(0)
        print('%s, %s' % (dt.isoformat(), txt))

    def __init__(self):
        # 保存收盘价的引用
        self.dataclose = self.datas[0].close
        # 跟踪挂单
        self.order = None
        # 买入价格和手续费
        self.buyprice = None
        self.buycomm = None
        # 加入均线指标
        self.sma = bt.indicators.SimpleMovingAverage(self.datas[0], period=self.params.m


    # 订单状态通知，买入卖出都是下单
    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            # broker 提交/接受了，买/卖订单则什么都不做
            return

        # 检查一个订单是否完成
        # 注意：当资金不足时，broker会拒绝订单
```

```
        if order.status in [order.Completed]:
            if order.isbuy():
                self.log(
                    '已买入, 价格: %.2f, 费用: %.2f, 佣金 %.2f' %
                    (order.executed.price,
                     order.executed.value,
                     order.executed.comm))

                self.buyprice = order.executed.price
                self.buycomm = order.executed.comm
            elif order.issell():
                self.log('已卖出, 价格: %.2f, 费用: %.2f, 佣金 %.2f' %
                         (order.executed.price,
                          order.executed.value,
                          order.executed.comm))
            # 记录当前交易数量
            self.bar_executed = len(self)

        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
            self.log('订单取消/保证金不足/拒绝')

        # 其他状态记录为：无挂起订单
        self.order = None

    # 交易状态通知，一买一卖算交易
    def notify_trade(self, trade):
        if not trade.isclosed:
            return
        self.log('交易利润, 毛利润 %.2f, 净利润 %.2f' %
                 (trade.pnl, trade.pnlcomm))

    def next(self):
        # 记录收盘价
        self.log('Close, %.2f' % self.dataclose[0])

        # 如果有订单正在挂起，不操作
        if self.order:
            return

        # 如果没有持仓则买入
        if not self.position:
            # 今天的收盘价在均线价格之上
            if self.dataclose[0] > self.sma[0]:
                # 买入
                self.log('买入单, %.2f' % self.dataclose[0])
                    # 跟踪订单避免重复
                self.order = self.buy()
        else:
            # 如果已经持仓，收盘价在均线价格之下
            if self.dataclose[0] < self.sma[0]:
                # 全部卖出
                self.log('卖出单, %.2f' % self.dataclose[0])
                # 跟踪订单避免重复
                self.order = self.sell()

    # 测略结束时，多用于参数调优
    def stop(self):
        self.log('(均线周期 %2d)期末资金 %.2f' %
                 (self.params.maperiod, self.broker.getvalue()), doprint=True)


if __name__ == '__main__':
    # 创建Cerebro引擎
    cerebro = bt.Cerebro()
    # Cerebro引擎在后台创建broker(经纪人)，系统默认资金量为10000

    # 为Cerebro引擎添加策略
    # cerebro.addstrategy(TestStrategy)

    # 为Cerebro引擎添加策略，优化策略
    # 使用参数来设定10到31天的均线,看看均线参数下那个收益最好
    strats = cerebro.optstrategy(
        TestStrategy,
        maperiod=range(10, 31))

    # 获取当前运行脚本所在目录
    modpath = os.path.dirname(os.path.abspath(sys.argv[0]))
    # 拼接加载路径
```

```
datapath = os.path.join(modpath, '../../datas/orcl-1995-2014.txt')

# 创建交易数据集
data = bt.feeds.YahooFinanceCSVData(
    dataname=datapath,
    # 数据必须大于fromdate
    fromdate=datetime.datetime(2000, 1, 1),
    # 数据必须小于todate
    todate=datetime.datetime(2000, 12, 31),
    reverse=False)

# 加载交易数据
cerebro.adddata(data)


# 设置投资金额1000.0
cerebro.broker.setcash(1000.0)

# 每笔交易使用固定交易量
cerebro.addsizer(bt.sizers.FixedSize, stake=10)
# 设置佣金为0.0
cerebro.broker.setcommission(commission=0.0)

cerebro.run()
```

没有调用addstrategy，而是用optstrategy函数将策略添加到Cerebro。传入的是要测试的一系，而不是单个值。 在策略类中添加了stop 方法，它将在每轮回测之后被调用，我们用它来打测结束之后的资产余额。 框架将为策略测试每个参数值，下面是输出结果:

```
0-12-29, (均线周期10) 期末资金 880.30
0-12-29, (均线周期11) 期末资金 880.00
0-12-29, (均线周期12) 期末资金 830.30
0-12-29, (均线周期13) 期末资金 893.90
0-12-29, (均线周期14) 期末资金 896.90
0-12-29, (均线周期15) 期末资金 973.90
0-12-29, (均线周期16) 期末资金 959.40
0-12-29, (均线周期17) 期末资金 949.80
0-12-29, (均线周期18) 期末资金 1011.90
2000-12-29, (均线周期19) 期末资金 1041.90
2000-12-29, (均线周期20) 期末资金 1078.00
2000-12-29, (均线周期21) 期末资金 1058.80
2000-12-29, (均线周期22) 期末资金 1061.50
2000-12-29, (均线周期23) 期末资金 1023.00
2000-12-29, (均线周期24) 期末资金 1020.10
2000-12-29, (均线周期25) 期末资金 1013.30
2000-12-29, (均线周期26) 期末资金 998.30
2000-12-29, (均线周期27) 期末资金 982.20
2000-12-29, (均线周期28) 期末资金 975.70
2000-12-29, (均线周期29) 期末资金 983.30
2000-12-29, (均线周期30) 期末资金 979.80
```

结果显示： 周期参数在18以下的亏损（在没有手续费的情况下）; 周期参数在18至26之间的盈利; 周期参数大于26的又会亏损;

对这个策略来说，最优的参数是：回看周期20，本金1000，盈利78元，收益率7.8%。

> 注意 在上面的例子中，移除了多余的用来绘图的指标，数据开始回测的时间仅取决于我们添加的简单移动平均线。所以周期为15的回测结果和之前的略有不同。

总结

上面的教程，我们从一个头开始，一步步搭建了一个能运行的回测系统，并且具备绘制结果和优化参数功能。 除此之外，还能做一些提高胜率的事情： 自定义指标：创建自定义指标很容易，绘制它们同样简单； 下单数量：资金管理是交易成功的关键之一； 委托单类型（限价单、止损单、限价止损单）等等。

阅读后续章节，获取相关功能的介绍。 运气很重要，祝好运～

5. 一些概念

这是backtrader一些概念的集合。了解这些概念对，对于使用平台很有帮助。

1. 开始之前

的代码示例，需要导入以下库才能使用：

```
ort backtrader as bt
import backtrader.indicators as btind
ort backtrader.feeds as btfeeds
```

意： 访问子模块的另一种语法： 以bt形式导入backtrader 然后： thefeed = .feeds.OneOfTheFeeds（...） theind = bt.indicators.SimpleMovingAverage（...）

2. 传递交易数据

台的基础工作都是由“策略”完成。 将交易数据传递给策略，用户无需关心怎么接收收据。 交易数据以数组的形式传递给“策略”，并作为“策略”的成员变量，可以通过数组下标的方式快捷访问

快速预览一下策略派生类的声明和框架的运行

```
ss MyStrategy(bt.Strategy):
 params = dict(period=20)

 def __init__(self):
     sma = btind.SimpleMovingAverage(self.datas[0], period=self.params.period)
 ...

ebro = bt.Cerebro()

a = btfeeds.MyFeed(...)
ebro.adddata(data)

ebro.addstrategy(MyStrategy, period=30)
...
```

请注意以下几点：策略的构造方法**init**中并未接收到 *args或* kwargs任何参数（但是它们仍可以使用），成员变量self.datas，该成员变量为数组/列表/可迭代，至少要要有一条记录（否则将引发异常）。就是这样，交易数据已添加到框架中，并且将按照添加到系统中的顺序显示在策略中。

> 注意： 这种方式，同样适用于框架源码中的现有指标，或者用户开发的自定义指标。

5.2.1. 交易数据快捷访问

可以使用其他自动成员变量直接访问self.datas数组项：

- 通过self.data访问self.datas[0]
- 通过self.dataX访问self.datas[X] 例如：

```
class MyStrategy(bt.Strategy):
    params = dict(period=20)

    def __init__(self):
        sma = btind.SimpleMovingAverage(self.data, period=self.params.period)
    ...
```

5.2.2. 省略交易数据

上面的示例可以进一步简化为：

```
class MyStrategy(bt.Strategy):
    params = dict(period=20)

    def __init__(self):
        sma = btind.SimpleMovingAverage(period=self.params.period)
    ...
```

self.data已从SimpleMovingAverage的调用中完全删除，SimpleMovingAverage默认首个参数就是self.data也即self.data0（self.data[0]）。

5.2.3. 一切皆是数据源

仅交易数据是数据源，可以传递给测策略， 指标和操作结果同样也是数据源。

面的示例中，SimpleMovingAverage接收self.datas[0]作为要进行操作的输入。下面看一个操作结果和额外指标的示例：

```
ss MyStrategy(bt.Strategy):
 params = dict(period1=20, period2=25, period3=10, period4)

 def __init__(self):
     sma1 = btind.SimpleMovingAverage(self.datas[0], period=self.p.period1)

     # 第二移动平均线使用sma1作为参数，均线的均线
     sma2 = btind.SimpleMovingAverage(sma1, period=self.p.period2)

     # 通过算术运算创建的新数据
     something = sma2 - sma1 + self.data.close

     # 第三移动平均线使用something作为参数
     sma3 = btind.SimpleMovingAverage(something, period=self.p.period3)

     # 比较sma3和sma1...
     greater = sma3 > sma1

     # 并没有实际意义的均值
     # 第四移动平均线使用greater做为参数
     sma3 = btind.SimpleMovingAverage(greater, period=self.p.period4)
 ...
```

上，所有内容都会转换为一个对象，一旦对其进行操作，就可以用作数据源。

5.3. 参数

通常，平台中的所有其他类都支持参数的概念。 参数和默认值一起声明为类的属性（元组或类似字典的对象）。 扫描关键字args（**kwargs）以查找匹配的参数，如果找到则将它们从** kwargs中删除，并将值分配给相应的参数。 通过访问成员变量self.params（简写为self.p），最终可以在类的实例中使用参数。 先前的简洁策略已经包含一个参数示例，这里我们再次聚焦参数的使用。

通过元组方式：

```
class MyStrategy(bt.Strategy):
    params = (('period', 20),)

    def __init__(self):
        sma = btind.SimpleMovingAverage(self.data, period=self.p.period)
```

通过字典方式：

```
class MyStrategy(bt.Strategy):
    params = dict(period=20)

    def __init__(self):
        sma = btind.SimpleMovingAverage(self.data, period=self.p.period)
```

5.4. Lines（线群）

同样，平台中几乎所有对象都是使用了Lines（线群）对象。 从用户的角度来看，这意味着：

- Lines（线群）对象可以容纳一个或多个线，线是由一组数据组成的数组，这组值在图表中放在一起就可以形成一条线。 线的一个很好的例子是由股票的收盘价形成的线，这实际上就是我们众所周知的收盘价曲线。

框架中对线的使用通常就是读取操作， 前面的小例子少加改造如下：

```
class MyStrategy(bt.Strategy):
    params = dict(period=20)

    def __init__(self):

        self.movav = btind.SimpleMovingAverage(self.data, period=self.p.period)

    def next(self):
        if self.movav.lines.sma[0] > self.data.lines.close[0]:
            print('移动平均线大于收盘价')
```

了拥有线的两个对象：

self.data具有一个lines属性，该属性包含一个close属性
self.movav是一个SimpleMovingAverage指标，也具有一个lines属性，该属性包含一个sma属性 close和sma通过访问(索引0)来比较值的大小

意： 由此可见，线只是一个名字而已。可以按照声明时的顺序访问它，但这仅适用于指标开发的过程中。

以通过快捷方式访问线：

xxx.lines可以缩短为xxx.l
xxx.lines.name可以缩写为xxx.lines_name
诸如策略和指标之类的复杂对象可以快速访问线数据

- self.data_name是对self.data.lines.name的直接访问
- 同样适用于编号的data变量：self.data1_name -> self.data1.lines.name

，也可以通过以下方式直接访问线的属性：

self.data.close

- self.movav.sma 但是，实际开发中，这种快捷访问的含义不如之前的清晰。

> 注意： 不支持使用后面的两种标记方式来给lines赋值

5.4.1. Lines（线群）声明

如果开发一个指标，则该指标所拥有的Lines必须要声明。 就像params一样，但是只能作为元组类型的成员属性，不支持字典，因为Lines不按照插入顺序存储内容。 对于简单移动平均线，可以这样进行声明：

```
class SimpleMovingAverage(Indicator):
    lines = ('sma',)

    ...
```

> 注意： 如果您将单个字符串传递给元组，则在元组中声明后的添加逗号。否则，字符串中的每个字母都将被解释为要添加到元组的项。 这可能是Python语法中少数不合理的几个地方之一。

如上例所示，该声明在指标中创建了一条sma线，以后可以在该策略进行访问（并可能传递给其他指标来创建更复杂的指标）。

对于开发而言，通过非命名的方式访问lines会很有用，这也是使用数字索引访问的方便之处：

- self.lines[0] 指向 self.lines.sma

如果定义了更多的线，可以将使用索引1、2或者更高的索引进行访问。当然，也确实存在快捷访问版本：

- self.line 指向 to self.lines[0]
- self.lineX 指向 self.lines[X]
- self.line_X 指向 self.lines[X]

交易数据对象也可以通过数字快速访问对象内部的lines：

- self.dataY 指向 self.data.lines[Y]
  self.dataX_Y 指向 self.dataX.lines[X] 这是 self.datas[X].lines[Y] 的缩写版本。

.2. 访问交易数据中的lines（线群）

易数据内部，也可以省略的方式来访问lines，比如可以用比较自然的方式访问收盘价：

```
a = btfeeds.BacktraderCSVData(dataname='mydata.csv')

ss MyStrategy(bt.Strategy):
 ...
 def next(self):
     if self.data.close[0] > 30.0:
         ...
```

方式肯定比 self.data.lines.close[0] > 30.0: 更加自然些。但是这不适用于以下指标：

指标只具有一个属性close，该属性包含一个中间计算，该中间计算随后的结果赋值给lines中的close 对于交易数据，不会进行任何计算，因为它只是一个数据源。

.3. 线的长度

s是一组点的集合，并且在执行过程中会动态增长，因此可以通过调用Python中的len函数随时长度。 这适用于：

交易数据
策略
指标

数据预加载后，data属性也可使用buflen方法，返回交易数据的柱线数。

...buflen之间的区别：

- len报告已处理了多少条
- buflen报告已为交易数据加载的柱线总数

如果两个都返回相同的值，则要么没有数据被预加载，要么当前处理已消耗了所有预加载的数据（除非系统连接到实时交易数据，否则将意味着处理结束）。

5.4.4. 线和参数的继承

框架提供了一种元语言来支持参数和线的声明。我们尽量使其与Python的继承规则兼容。

参数继承

- 支持多重继承
- 基类的参数被继承
- 如果多个基类定义相同的参数，则使用继承列表中最后一个类的默认值
- 如果在子类中重新定义了相同的参数，则新的默认值将覆盖基类的默认值。

Lines继承

- 支持多重继承
- 所有基类的Lines均被继承。被命名为Lines的情况下，如果在基类中多次使用相同的名称，则Lines中只有一个该名称的属性。

5.5. 索引0和-1

如前所述，Lines是线群，线是一组点的集合，这些点在绘制在一起形成一条线（例如，沿着时间轴将所有收盘价连在一起就形成收盘价曲线）

要在常规代码中访问这些点，一般通过0索引的方式对当前点进行get/set操作。 策略只能读取数据，指标既可以读取也可以写入数据。

前面简单的示例，策略中的next方法：

```
next(self):
if self.movav.lines.sma[0] > self.data.lines.close[0]:
    print('简单移动平均线大于收盘价')
```

索引0获得移动平均线的当前值和当前收盘价，并比较它们的大小。

意： 实际上对于索引0，可直接进行逻辑/算术运算操作，如下所示：

```
self.movav.lines.sma > self.data.lines.close:
...
```

相关说明请参阅文档后面的《操作章节》。

标开发的应用中，会出现赋值操作。 例如SimpleMovingAverage的当前值可以通过如下方式读写：

```
 next(self):
elf.line[0] = math.fsum(self.data.get(0, size=self.p.period)) / self.p.period
```

前一个点集合可以按照Python访问数组索引为-1的方式：

它指向数组的最后一项

认为最后一项为（读写当前点的前一个点）索引值为-1。 因此，在策略中比较当前收盘价与个收盘价是通过 0 vs -1的方式。例如：

```
next(self):
if self.data.close[0] > self.data.close[-1]:
    print('今天收盘价更高')
```

同理，使用-1，-2，-3，...便访问-1之前项的价格。

5.6. 切片

backtrader不支持对线对象进行切片，这是遵循[0]和[-1]索引方案的设计决策。 使用常规的可索引Python对象，可以执行以下操作：

```
# 从开始到结尾的切片
myslice = self.my_sma[0:]
```

但是请记住，选择0...实际上是当前开始传递的值，之后也没有任何值。

```
# 从开始到结尾的切片
myslice = self.my_sma[0:-1]
```

同样，...0是当前值，而-1是先前交付的值。 这就是为什么从0->-1进行的切片反向操作就毫无意义的原因。
如果可以反向操作，那么切片可能应该这样：

```
# 从当前点向前的切片
myslice = self.my_sma[:0]
# 从最后的值到当前值
myslice = self.my_sma[-1:0]
# 从最后的值到倒数第3个值
myslice = self.my_sma[-3:-1]
```

5.6.1. 获取切片

可以获得具有最新值的数组，语法：

```
# 显示默认值
myslice = self.my_sma.get(ago=0, size=1)
```

一个数组，该数组的大小为1，当前时刻为0，向后获取。
要从当前时间点获取10个值（即：最后10个值）：

```
go的默认值为0
lice = self.my_sma.get(size=10)
```

数组具有你所期望的顺序。最左边的值是最旧的值，最右边的值是最新的值（这是常规的
n数组，而不是lines对象）。

```
跳过当前点获取最后10个值
lice = self.my_sma.get(ago=-1, size=10)
```

7. 线的延迟索引

算符可用于在next逻辑阶段提取单个值。lines对象支持附加的符号，以便在__init__阶段通过延迟的线对象寻址取值。
一条逻辑是将先前的收盘价与简单移动平均线的实际值进行比较。无需在每次next迭代中进行操作，而是可以生成预定义的lines对象：

```
ss MyStrategy(bt.Strategy):
 params = dict(period=20)

 def __init__(self):

     self.movav = btind.SimpleMovingAverage(self.data, period=self.p.period)
     self.cmpval = self.data.close(-1) > self.sma

 def next(self):
     if self.cmpval[0]:
         print('上一个收盘价高于当前移动平均值')
```

这里使用"()"延迟符号：

- 这提供了收盘价的副本，但延迟了-1。
  比较self.data.close（-1）> self.sma 会生成另一个line对象，如果条件为True，则返回1，否则为0

5.8. 线(群)的耦合

运算符()可以与延迟的数值一起使用，以提供延迟的line对象。
如果使用中不提供延迟数值，则返回LinesCoupler对象。这是为了在操作具有不同时间范围的数据指标之间建立耦合。
不同时间范围的交易数据具有不同的长度，并且指标在操作这些数据时会复制数据的长度。例如：

- 日交易数据每年约有250条
- 周交易数据每年有52条

尝试创建一个比较两个简单移动平均线的操作，每次操作在引用数据时都有可能被中断。因为系统不知道如何在250条的日交易数据和52条的周交易数据之间进行匹配。

读者可以通过找出一天和一周的对应关系进行比较，但是：

- 指标只是数学公式，没有日期时间信息
  他们对上下文环境一无所知，只要提供了足够的数据，就可以进行计算。

于是()表示法（空调用）可用于解决这个问题：

```
class MyStrategy(bt.Strategy):
    params = dict(period=20)

    def __init__(self):
```

```
    # data0 是日交易数据
    sma0 = btind.SMA(self.data0, period=15)  # 15天sma
    # data1 是周要以数据
    sma1 = btind.SMA(self.data1, period=5)  # 5周sma

    self.buysig = sma0 > sma1()

def next(self):
    if self.buysig[0]:
        print('每日sma大于每周sma1')
```

里，较大的时间范围指标sma1通过sma1()与每日时间范围耦合。这将返回与更大数量的sma0的对象，并复制sma1产生的值，从而有效地将52个周数据分散为250个日数据。

9. 通过操作符构造对象

实现“简单易用”的目标，backtrader允许（在Python的语法范围内）使用操作符。为了进一步化，操作符的使用有两种情景。

5.9.1. 情景1-操作符创建对象

之前已经看到了一个例子。在指标和策略类的对象初始化阶段（__init__方法）中，操作符创保存可以持续使用的对象，供策略逻辑在评估阶段使用。
npleMovingAverage的潜在实现方式进一步细分为多个步骤。
leMovingAverage指标__init__内的代码可能如下：

```
__init__(self):
# N个周期值的总和，数据总和是一个Lines对象
# 在与运算符[]和索引0查询时
# 返回当前总和
datasum = btind.SumN(self.data, period=self.params.period)
# datasum（虽然是单行，但仍是一个Lines对象）
# 在这种情况下它可以除以int/float类型的数据。
# 但实际上它被除以后得到另外一个Lines对象。
# 该操作返回分配给av对象
# 当查询为[0]，则返回当前时间点的平均值
av = datasum / self.params.period
# av是对新的Lines对象的命名
# 其他对象使用这个指标可以直接访问计算
self.line.sma = av
```

策略初始化期间显示了更加完整的用法：

```
class MyStrategy(bt.Strategy):

    def __init__(self):

        sma = btind.SimpleMovinAverage(self.data, period=20)

        close_over_sma = self.data.close > sma

        sma_dist_to_high = self.data.high - sma

        sma_dist_small = sma_dist_to_high < 3.5

        # 不幸的是，"and"不能在Python中被重载
        # 在python中and不属于运算符，所以backtrader提供一个函数模拟这个功能
        sell_sig = bt.And(close_over_sma, sma_dist_small)
```

完成上述操作后，sell_sig是一个Lines对象，当指示满足条件时，可以直接在策略中使用。

5.9.2. 情景2-逻辑操作符

首先，策略的next方法，系统要处理每个柱时都要调用该方法，这就是操作符处于情景2地方。以前面的示例为基础：

```
class MyStrategy(bt.Strategy):
    def __init__(self):
```

```
    sma = btind.SimpleMovinAverage(self.data, period=20)

    close_over_sma = self.data.close > sma

    sma_dist_to_high = self.data.high - sma

    sma_dist_small = sma_dist_to_high < 3.5

    # 不幸的是，"and"不能在Python中被重载
    # 在python中and不属于运算符，所以backtrader提供一个函数模拟这个功能
    sell_sig = bt.And(close_over_sma, sma_dist_small)

def next(self):
    # 尽管这看起来不像是“操作号”，但确实返回的是正在测试对象的True/False
    if self.sma > 30.0:
        print('sma大于30.0')

    if self.sma > self.data.close:
        print('sma高于收盘价')

    if self.sell_sig:  # if sell_sig == True: would also be valid
        print('卖出标志为True')
    else:
        print('卖出标志为False')

    if self.sma_dist_to_high > 5.0:
        print('sma到high的距离大于5.0')
```

是一个非常有用的策略，只是一个例子。在情景2中，操作符返回期望值（如果测试值为，则返回布尔值；如果是浮点数进行比较，则返回浮点数），并且算术运算也返回期望值。

:意:
了进一步简化，比较实际上没有使用操做符。
self.sma > 30.0: …比较 self.sma[0] 和 30.0
self.sma > self.data.close: … 比较 self.sma[0] 和 self.data.close[0]

.3. 一些不可重载的运算符/函数

Python不允许重载所有内容，因此提供了一些功能函数来应对这种情况。

> 注意：仅适用于情景1，以创建对象供后面使用。

操作符:

- and -> And
- or -> Or

逻辑控制:

- if -> If

函数:

- any -> Any
- all -> All
- cmp -> Cmp
- max -> Max
- min -> Min
- sum -> Sum
- reduce -> Reduce

Sum实际上使用math.fsum作为底层操作，因为backtrader使用浮点数计算，如果用常规sum可能会影响精度。

这些实用的操作符/函数可迭代使用。可迭代的元素可以是常规的Python数字类型（int，float等），也可以是带有Lines的对象。 例如一个非常原始的买入信号：

```
class MyStrategy(bt.Strategy):
    def __init__(self):
        sma1 = btind.SMA(self.data.close, period=15)
```

```
        self.buysig = bt.And(sma1 > self.data.close, sma1 > self.data.high)

    def next(self):
        if self.buysig[0]:
            pass  # do something here
```

sma1高于最高价，则必高于收盘价，这里重点是说明bt.And的用法。

用法：

```
class MyStrategy(bt.Strategy):

    def __init__(self):
        # 在period=15的data.close上生成SMA
        sma1 = btind.SMA(self.data.close, period=15)
        # 如果sma的值大于close，则返回low，否则返回high
        high_or_low = bt.If(sma1 > self.data.close, self.data.low, self.data.high)
        sma2 = btind.SMA(high_or_low, period=15)
```

说明：

在period=15的data.close上生成sma1

- 如果sma1的值大于close，则返回low，否则返回high
  调用bt.If时不会返回任何实际值。它返回一个Lines对象，就像SimpleMovingAverage一样，这些值将在稍后计算中会用到
  然后将bt.If生成的Lines对象赋值给sma2该,sma2有时会使用最低价，有时会使用高价进行计算

函数也可以使用数值，修改后得到示例：

```
ss MyStrategy(bt.Strategy):
    def __init__(self):
        sma1 = btind.SMA(self.data.close, period=15)
        high_or_30 = bt.If(sma1 > self.data.close, 30.0, self.data.high)
        sma2 = btind.SMA(high_or_30, period=15)
```

现在，sma2使用30.0或最高价进行计算，具体取决于sma1和close的比较结果。

> 注意：数值30在内部转换为伪迭代，始终返回30

6. Cerebro(核心引擎)

6.1. Cerebro

Cerebro类是backtrader的引擎，是以下几个方面的核心：

- 收集所有输入（Data Feeds），执行（Stratgegies），监控（Observers），评测（Analyzers）和记录（Writers），以确保系统随时运行。
- 执行回测或实盘交易
- 返回结果
- 绘图

6.1.1. 收集输入数据

1.以创建cerebro开始：

```
# **kwargs参数支持某些控制执行，请参阅后面文档（相同的参数也应用于run方法）
cerebro = bt.Cerebro(**kwargs)
```

2.添加交易数据
最常见的模式是 cerebro.adddata（data），其中data是已实例化的数据源。例：

```
data = bt.BacktraderCSVData(dataname='mypath.days', timeframe=bt.TimeFrame.Days)
cerebro.adddata(data)
```

重新采样或数据重播遵循相同的模式：

```
data = bt.BacktraderCSVData(dataname='mypath.min', timeframe=bt.TimeFrame.Minutes)
cerebro.resampledata(data, timeframe=bt.TimeFrame.Days)
# 或者
data = bt.BacktraderCSVData(dataname='mypath.min', timeframe=bt.TimeFrame.Minutes)
cerebro.replaydatadata(data, timeframe=bt.TimeFrame.Days)
```

可以接受任何数量的交易数据，包括将常规数据与重采样和/或重播的数据混合。当然，这些组合中的某些组合肯定会毫无意义，并且为了组合数据有意义增加了限制条件：根据时间条件限请参阅数据章节的多个时间范围、数据重采样和数据重播部分。

加策略
易数据不同的是交易数据已经是类的实例，而cerebro是直接使用策略类和参数。因为：在策化方案中，该类将被实例化多次并传递不同的参数。
没有运行策略优化方案，该模式仍然适用：

```
cerebro.addstrategy(MyStrategy, myparam1=value1, myparam2=value2)
优化时，必须将参数作为可迭代对象添加。有关详细说明，请参见“优化”部分。
cerebro.optstrategy(MyStrategy, myparam1=range(10, 20))
```

使用myparam1值从10到19的值运行MyStrategy10次（记住Python中的范围是半开的，不会20）

他要素
添加其他一些元素来增强回测体验，请参见相应的部分。方法是：

addwriter
addanalyzer
addobserver (or addobservermulti)

换经纪人
bro将在backtrader中使用默认经纪人，但是可以被重写：

```
broker = MyBroker()
cerebro.broker = broker  # property using getbroker/setbroker methods
```

6.接收通知
如果交易数据和/或代理发送通知（被store provider创建的通知），则它们将通过Cerebro.notify_store方法接收。有三种处理通知的方法：

- 通过addnotifycallback(callback)方法将回调添加到cerebro实例中。 回调方法如下：
  callback(msg, *args, **kwargs)*
  实际收到的msg， args和** kwarg 由（data/broker/store）来决定，但通常它们是可打印的，以便进行接收和实验。
- 在Strategy子类中的覆盖notify_store方法
  notify_store（self，msg， *args，* *kwargs）
- 通过子类继承Cerebro并覆盖notify_store（与策略中的方法相同）
  最不推荐使用该方法

6.1.2. 执行回测

运行回测比较简单，但是它支持多个选项来决定如何运行（可以在实例化时指定运行模式）：

```
result = cerebro.run(**kwargs)
```

请参阅下文中相关部分，以了解可用的参数选项。

6.1.2.1. 标准观察者

cerebro（除非另有说明）自动实例化三个标准观察者

- 经纪人观察者用来跟踪现金和投资组合的价值
- 交易观察者，会显示每笔交易所产生的影响
- 买/卖观察者主要对订单操作进行记录

如果希望使用更干净的绘图，可通过stdstats = False 来禁用这些观察者。

.3. 返回结果

bro返回在回测期间创建的策略实例。由于策略中的所有元素都可访问，因此可以分析他们的

```
ult = cerebro.run(**kwargs)
```

结果的格式将根据是否使用优化选项而有所不同（策略可以通过optstrategy方式添加到bro中）：

所有策略通过addstrategy添加
esult返回的是一个list
1个或多个策略通过optstrategy添加
esult返回的是list的list， 每个列表项是对应策略返回的list。

意

为使得内核传递的消息更轻便，现在optimization默认返回分析器analyzers。

希望将整套策略作为返回值，请将参数optreturn设置为False。

.4. 便捷绘图

安装了matplotlib，则可以绘制策略结果。 通常的模式是：

```
ebro.plot ()
```

读下面的参考中“绘图”部分。

.5. 回测逻辑

回测流程概述：

1.传递store中的所有通知；

2.令数交易数据以柱集合方式传递给next方法；
在版本1.9.0.99中已更改为：
交易数据根据datetime同步检索出来，然后提供给next方法。当没有新时间段的交易数据时，就使用旧的数据，有则使用新的（指标中的计算也是如此）。
如果过使用旧版本的默认行为，就在使用Cerebro时，将oldsync=True。

插入到系统中的第一个数据是主数据，其他交易数据为从属数据，系统将等待第一次响应时钟，并且：

- 如果下一时钟周期比datamaster已传递的数据时间要新，则该数据将不会被传递。
- 由于多种原因，可能会出现在没有提供新报价的情况下提前返回

该逻辑目的在于同步多个具有不同时间点的数据。

3.将有关订单，交易和现金/价值的消息队列，通过经纪通知给策略；

4.告知经纪人接受订单队列并使用新数据执行挂单操作；

5.调用该策略的next方法，以使该策略评估新数据（以及可能发布在经纪人中排队的订单）
根据阶段的不同，在满足策略/指标的最短期限要求之前（可能在prenext或者nextstart方法中），这些策略内部还将对观察者，指标，分析器和其他活跃要素产生影响；

6.告诉任何记录器将数据写入目标。

重要的考虑因素：

> 注意:
> 在上面的步骤1中，当交易数据传递新的一组柱线时，旧柱线被关闭，这意味着这些被关闭的数据已经发生过。

因此，无法使用步骤1中的数据执行步骤4中策略发出的订单。
这意味着定单将以x + 1的概念执行，其中x是定单执行的柱线，x + 1是下一个定单，这是可能执行[illegible]的最早时间。

2.1.6. 相关介绍

s backtrader.Cerebro() 参数：

preload (default: True) 是否为策略预先加载传递给cerebro的不同的交易数据
runonce (default: True) 在矢量模式下运行指标，以加快整个系统的运行速度，策略和观察者将始终基于事件运行
live (default: False) 如果没有实时数据（通过数据的islive方法，但用户仍希望以实时模式运行，可以将此参数设置为true），这将同时停用preload和runonce,它不会对内存节省方案产生影响。

:完待续...

7 交易数据

trader带有一组交易数据解析器（解析所有基于CSV格式的文件），可让你从不同来源加载数

Yahoo（在线或已保存到文件）
VisualChart（请参见www.visualchart.com）
Backtrader CSV（backtrader用于测试的csv文件）
通用CSV支持

速入门指南中可以清楚地看到，你已将交易数据添加到Cerebro中。 交易数据将用于下不同的：

- 数组self.datas（按插入顺序）
- 数组对象的别名：
  - self.data和self.data0指向第一个元素
  - self.dataX指向数组中索引为X的元素

8. 策略

8.1. 简介

Cerebro是backtrader的心核心，策略是用户的核心。

研究一下策略的生命周期:

> 注意：
> 一个策略可能在出生时被来自backtrader.errors模块中的StrategySkipError异常所中断，这样可以避免在回测期间继续执行该策略，详情请参阅《异常》部分章节。

1. 初始化：__init__ 这是在实例初始化期间调用的：指示和其他所需的属性将在此处创建。例如：

```
def __init__ () :
 self.sma = btind.SimpleMovingAverage (period = 15)
```

2. 启动：start cerebro通知strategy开始运行，默认值是一个空方法。

3. 预循环：prenext
初始化时，声明一个指标，指标限制了策略可使用的最小期限。例如：在**init**上方，创建了一

个Period = 15的SimpleMovingAverage。
只要系统看到的柱线数量少于15条，就会调用prenext（默认实现是空操作）

循环：next
一旦系统看到交易数据中有超过15条柱线，并且SimpleMovingAverage具有足够大的缓冲区，该策略可以真正执行。
有一个nextstart方法，该方法仅被调用一次，以标记从prenext到next的切换。 nextstart的默认实现是简单地调用next方法。

重复：none
主要在进行参数优化时（使用不同的参数），系统会根据不同的参数重复实例化多次。

停止：stop
进行重置的时间到了，系统会告之该策略进行重置操作， 默认方法为空。

多数情况下，常规使用模式，如下：

```
ss MyStrategy(bt.Strategy):

 def __init__(self):
     self.sma = btind.SimpleMovingAverage(period=15)

 def next(self):
     if self.sma > self.data.close:
         # Do something
         pass

     elif self.sma < self.data.close:
         # Do something else
         pass
```

代码段中：

在__init__期间，为属性分配了一个指标
默认的空start方法不会被覆盖

- prenext和nexstart不会被覆盖
- 在next方法中，将指标的值与收盘价进行比较以执行某项操作
- 默认的空stop方法不会被覆盖

策略就像现实世界中的交易者一样，当有事件发生时会得到通知。实际上，回测过程中的每个周期next方法被调用一次。该策略将执行以下动作：

- 通过notify_order（order）通知订单中的任何状态更改
- 通过notify_trade（trade）通知任何开仓/更新/平仓交易
- 通过notify_cashvalue（cash, value）通知经纪人当前的现金和投资组合
- 通过notify_fund（cash, value, fundvalue, shares）通知经纪人当前的现金和投资组合以及基金价值和股票的交易
- 通过notify_store（msg， *args*，*kwargs）实现特定的事件

请参阅Cerebro对有关store通知的说明，即使store通知传递给了Cerebro，也一样会传递给策略（使用重写的notify_store方法或通过回调的方式）。

策略也希望交易者扑捉到市场中的机会，并在next方法中，通过以下操作来获利：

- buy方法可以做多或减少/关闭空头头寸
- sell方法可以做空或减少/关闭多头头寸
- close方法可以平仓
- cancel方法取消尚未执行的订单

9. 指标

9.1. 指标用法

指标可以在backtrader的两个地方使用：

- 策略内部
  其他指标内部

.1. 指标使用逻辑

_标始终在策略的__init__期间实例化
. 指标的值（或者指标派生出来的值）在next方法中被使用

个重要的公理要考虑：
init__期间声明的任何指标（或其派生值）将在调用next之前进行预先计算。
来探讨一下操作模式的差异：

1.1. __init__ vs next

在__init__期间涉及线对象的任何操作都会生成另一个线对象
在next方法中任何涉及line对象的操作都会产生常规的Python类型，例如float和bool。

9.1.1.2. __init__ 初始化期间

```
o_diff = self.data.high - self.data.low
量hilo_diff拥有对lines对象的引用，该引用在调用next之前已预先计算，可以使用标准数组符号[]进行访问，
```

合简单的线（如self.data交易数据的线）和复杂的线（如指标）时，同样适用：

```
 = bt.SimpleMovingAverage(self.data.close)
se_sma_diff = self.data.close - sma
lose_sma_diff 包含一个线对象
```

逻辑运算符：

```
se_over_sma = self.data.close > sma
# 此时生成的lines对象将包含一个布尔数组
```

9.1.1.3. next方法中

```
# 操作示例（逻辑运算符）
close_over_sma = self.data.close> self.sma
```

```
# 使用等效数组（基于索引0的表示法）
close_over_sma = self.data.close [0]> self.sma [0]
```

在这种情况下，close_over_sma会产生一个布尔值，该值是比较两个浮点值的结果，这两个浮点值是由[0]运算符返回的值应用于self.data.close和self.sma的

9.1.1.4. 为什是__init__而不是next的原因

逻辑简化易用是关键。可以在__init__期间进行相关的声明和计算，以便在next方法中将实际操作逻辑降至最低。
还有额外的好处：速度（因为一开始就预先计算好了）。
一个在__init__期间生成购买信号的完整示例：

```
class MyStrategy(bt.Strategy):

    def __init__(self):

        sma1 = btind.SimpleMovingAverage(self.data)
        ema1 = btind.ExponentialMovingAverage()

        close_over_sma = self.data.close > sma1
        close_over_ema = self.data.close > ema1
        sma_ema_diff = sma1 - ema1

        buy_sig = bt.And(close_over_sma, close_over_ema, sma_ema_diff > 0)

    def next(self):
```

```
    if buy_sig:
        self.buy()
```

> 注意
>
> 无法覆盖Python的and运算符，所以使用backtrader自定义的And，类似的函数还有Or和If。

易见，在__init__期间使用“声明式”方法可以将next方法中（实际策略触发的地方）的cpu使用降至最低。

> 注意
>
> 逻辑变得非常复杂并涉及多个操作时，通常最好将其封装在一个指标类中。

.2. 一些注意事项

面的示例中，与其他平台相比，backtrader简化了两件事：

声明的指标既没有获取父参数（例如创建它们的策略）也没有调用任何类型的“注册”方法/函数。
尽管该策略没有启动指标计算，但是由于sma-ema的操作，线对象依然会生成。
在没有self.data的情况下实例化了ExponentialMovingAverage
如果未传递任何数据，则父级（策略）的第1个数据将被自动传递。

.3. 指标绘图

，最重要的是：

声明的指标会自动被绘制（如果调用cerebro.plot）
不会绘制操作的线条对象（例如close_over_sma = self.data.close> self.sma）
如果需要，可以使用LinePlotterIndicator方法来辅助绘制：

```
close_over_sma = self.data.close> self.sma
LinePlotterIndicator（close_over_sma,name ='Close_over_SMA'）
# name参数将名称指定给此指标
```

9.1.3.1. 绘图控制

> 未完待续...

9.2. 指标开发

如果必须定制任何内容（一个或多个获胜策略除外），这一定是自定义指标。
backtrader下自定义指标非常容易。
必要流程如下：

- 从指标（直接或从已经存在的子类）派生的类；
- 定义将需要保存或使用的线对象；
- 指标必须至少有1条线，如果继承了一个指标，则线对象已经被定义过了；
- （可选）定义可以更改行为的参数；
- （可选）提供/定制一些元素，以实现对指标的合理绘制；
- 在__init__中提供一个完整定义的操作，并绑定（分配）到指标的线，或者提供next 和（可选）once方法使用；
  如果在初始化期间使用逻辑/算术运算完全定义一个指标，并且结果分配给该线。
  如果不是如此，至少必须提下一个指标，该指标必须为索引为0的线分配一个值，可以通过once方法来实现Runonce模式（批处理操作）的计算优化。

9.2.1. 重要说明：幂等性

指标为收到的每个柱线产生输出。 不必假设同一柱将被发送多少次，操作必须是幂等的。
其基本原理：

- 相同的柱线（以索引为单位）可以通过更改值（即，更改值是收盘价）多次发送。

例如，“重播”每日会话，可以使用由5分钟柱组成的日内数据。

backtrader允许使用实时交易数据。

1.1. 一个虚拟（但功能正常）指标

```
class DummyInd(bt.Indicator):
    lines = ('dummyline',)

    params = (('value', 5),)

    def __init__(self):
        self.lines.dummyline = bt.Max(0.0, self.params.value)
```

将始终输出相同的值：0.0或self.params.value（如果恰好大于0.0）。
的指标，但使用next方法：

```
ss DummyInd(bt.Indicator):
    lines = ('dummyline',)

    params = (('value', 5),)

    def next(self):
        self.lines.dummyline[0] = max(0.0, self.params.value)
```

意 在__init__版本中，使用bt.Max分配Line对象给self.lines.dummyline。

ax返回一个line对象，该对象会针对传递到指标的每个柱线自动进行迭代。
使用max代替，则分配将是毫无意义的，因为指标得到一个固定值，而不是一个线对象。 在方法中，将直接使用浮点值完成工作，并且可以使用内置标准的max函数。

们回顾一下，self.lines.dummyline是长符号，可以将其缩短为：

self.l.dummyline

甚至

- self.dummyline

后者只有在代码未使用member属性才有能使用。 第三个也是最后一个版本提供了一种额外的once方法来优化计算：

```
class DummyInd(bt.Indicator):
    lines = ('dummyline',)

    params = (('value', 5),)

    def next(self):
        self.lines.dummyline[0] = max(0.0, self.params.value)

    def once(self, start, end):
       dummy_array = self.lines.dummyline.array

       for i in xrange(start, end):
           dummy_array[i] = max(0.0, self.params.value)
```

无论如何__init__版本是最好的：

- 一切都限于初始化
- next和once（均已优化，因为bt.Max已经拥有它们），自动提供而无需使用索引和/或公式

如果需要开发，该指标还可以覆盖与next和once相关的方法：

- prenext和nexstart
- reonce和oncestart

9.2.1.2. 手动/自动最短时间

如果可能，backtrader将对其进行计算，但可能需要手动操作。 这是简单移动平均线的潜在实现：

```
class SimpleMovingAverage1(Indicator):
    lines = ('sma',)
    params = (('period', 20),)

    def next(self):
        datasum = math.fsum(self.data.get(size=self.p.period))
        self.lines.sma[0] = datasum / self.p.period
```

听起来不错，但backtrader不知道最小周期是多少，即使该参数被命名为“period”（名称可能会引起误解，并且某些指标会收到几种用法不同的“period”）

种情况下，将已经为第一个数据柱调用了next方法，因为get方法无法返回所需的
.period，将会导致一系列的异常。在解决问题之前，必须考虑以下事项：

传递给指标的交易数据可能已经存在最短时间 样例SimpleMovingAverage可以在以下位置进行：
常规交易数据：默认最小周期为1（只需等待进入系统的第一个柱）
另一个移动平均线...这已经有一个周期　如果这是20，而我们的样本移动平均线也为20，那么我们的最小周期为40条　实际上，内部计算结果为39...因为第一个移动平均线一旦产生一个柱线，便会计入下一个移动平均线，这会产生重叠的柱线，因此需要39。
其他带有期间的指标/目标

方法如下：

```
ss SimpleMovingAverage1(Indicator):
    lines = ('sma',)
    params = (('period', 20),)

    def __init__(self):
        self.addminperiod(self.params.period)

    def next(self):
        datasum = math.fsum(self.data.get(size=self.p.period))
        self.lines.sma[0] = datasum / self.p.period
```

addminperiod方法是让系统考虑该指标所需的额外周期条，以考虑可能存在的任何最小周期。 如果所有计算都是使用已经将其周期需求传达给系统的对象完成的，则有时绝对不需要。

快速实现带有直方图的MACD指标：

```
from backtrader.indicators import EMA

class MACD(Indicator):
    lines = ('macd', 'signal', 'histo',)
    params = (('period_me1', 12), ('period_me2', 26), ('period_signal', 9),)

    def __init__(self):
        me1 = EMA(self.data, period=self.p.period_me1)
        me2 = EMA(self.data, period=self.p.period_me2)
        self.l.macd = me1 - me2
        self.l.signal = EMA(self.l.macd, period=self.p.period_signal)
        self.l.histo = self.l.macd - self.l.signal
```

完成，无需考虑最小周期问题：

- EMA代表指数移动平均线（系统内置别名）
- 指标“macd”和“signal”的中被命名为lines的对象已经分配了periods周期对象
  - macd取自运算“me1-me2”的periods，而运算“me1-me2”又取me1和me2的periods的最大值（它们是不同周期的指数移动平均值）
  - signal直接采用macd上的指数移动平均线的periods。该EMA还考虑了已经存在的macd周期和所需的样本量（period_signal）来进行自身计算
  - histo取两个操作数“signal-macd”中的最大值。一旦两者都准备好，histo便有了具体的值

9.2.2. 一个完整的自定义指标

让我们开发一个简单的自定义指标，该指标的移动平均值可以用参数修改：

```
import backtrader as bt
import backtrader.indicators as btind

ss OverUnderMovAv(bt.Indicator):
 lines = ('overunder',)
 params = dict(period=20, movav=btind.MovAv.Simple)

 def __init__(self):
     movav = self.p.movav(self.data, period=self.p.period)
     self.l.overunder = bt.Cmp(movav, self.data)
```

，如果平均值高于数据，则指标的值为“1”，如果低于数据，则指标的值为“-1”。
数据是常规数据，则与收盘价相比会产生1和-1。
让指标可以优雅的绘图（更多内容可以参考绘图部分），我们可以添加以下几点：

```
ort backtrader as bt
ort backtrader.indicators as btind

ss OverUnderMovAv(bt.Indicator):
 lines = ('overunder',)
 params = dict(period=20, movav=bt.ind.MovAv.Simple)

 plotinfo = dict(
     # 1s和-1s的上方和下方添加额外的边距
     plotymargin=0.15,

     # 在1.0和-1.0处绘制参考水平线
     plothlines=[1.0, -1.0],

     # 将y的范围设置为1.0和-1.0之间
     plotyticks=[1.0, -1.0])

 # 用破折号样式绘制“overunder”线，ls代表线的样式，并直接传递给matplotlib
 plotlines = dict(overunder=dict(ls='--'))

 def _plotlabel(self):
     # 此方法返回将显示的标签列表，指标名称跟在绘图点后面
     # period 必须要有
     plabels = [self.p.period]

     # 如果不是默认值，则仅放置移动平均线
     plabels += [self.p.movav] * self.p.notdefault('movav')

     return plabels

 def __init__(self):
     movav = self.p.movav(self.data, period=self.p.period)
     self.l.overunder = bt.Cmp(movav, self.data)
```

未完待续...

10. 分析器

10.1. 分析器

无论是回测还是交易，能够分析交易系统的性能是非常重要的。因为不仅了解获得利润，而且还要了解风险。如果风险太大我们就要是否值得为这样的资产付出努力。参考资产（或无风险资产）。

这就是Analyzer对象家族的用武之地：提供对发生的事件甚至实际发生的事件的分析。

10.1.1. 分析器的性质

该接口是按照Lines对象的接口建模的，特征有点像next方法，但有一个主要区别：

- Analyzer不持有Lines对象
  这意味着它们在内存方面开销不大，因为即使在分析了数千个价格柱之后，它们只需将单个结果保存在内存中。

10.1.2. 在生态系统中的位置

cerebro实例将Analyzer对象（与策略，观察者和数据方式雷同）添加到系统中：

- addanalyzer(ancls, *args,* *kwargs)

cerebro.run中运行时，系统中存在的每种策略都会发生以下情况：

- 在cerebro中运行时，ancls会用 *args和* kwargs进行实例化。
  ancls实例将添加给该策略

味着：

如果回测运行包含例如3个策略，则将创建3个ancls实例，并将每个实例附加到不同的策略上。
底线：analyzer分析的是单个策略的性能，而不是整个系统的性能。

.2.1. 附加位置

分析器对象实际上可能会使用其他分析器来完成其工作。 例如：SharpeRatio使用TimeReturn出进行计算。
子分析器或从属分析器也将被插入到创建它们的策略中，但是它们对于用户是完全不可见的。

1.3. 属性

执行预期的工作，Analyzer对象提供了一些默认属性，这些属性会自动传递并在实例中进行设以便于使用：

self.strategy：策略子类的访问引用，策略可以访问的分析器的任何内容也可以被分析器访问。
self.datas[x]：该策略中存在的交易数据，尽管可以在策略中进行访问，但是该快捷方式访问可以更加舒适。

- self.data：self.datas[0]的快捷方式，带来更多舒适感。
- self.dataX：不同self.datas[x]的快捷访问方式。

可以使用其他一些别名，尽管它们可能有点过头了：

- `self.dataX_Y`，指向：`self.datas[X].lines[Y]`

如果该行具有名称，则以下内容也可用：

- `self.dataX_Name` 解析为 `self.datas[X].Name` 返回按名称而不是按索引的line

对于第一个数据，没有初始X的情况下，最后的快捷方式可用的。例如：

- `self.data_2` 指的是 `self.datas[0].lines[2]`
- `self.data_close` 指的是 `self.datas[0] .close`

10.1.3.1. 返回分析

Analyzer基类创建一个self.rets（类型为collections.OrderedDict）成员属性以返回分析结果。这是在create_analysis方法中完成的，如果创建自定义Analyzer，则子类可以覆盖该方法。

10.1.4. 操作方式

尽管Analyzer对象不是“线”对象，因此不会在线上进行迭代，但它们被设计为遵循相同的操作模式。

- 在系统投入运行之前实例化（因此调用__init__）；
- 从start操作中发出开始信号；
- prenext/nextstart/next将在指标所执行的策略最短周期之内被调用，
  prenext和nextstart的默认行为是调用next，因为分析器可能从系统运行的第一刻开始就进行分析；

- 通常可能会在Lines对象中调用len(self)来检查柱的实际数量，同样，在Analyzers中通过返回self.strategy来访问Lines对象的len()方法；
订单和交易将通过notify_order和notify_trade收到与策略一样的通知；
现金和数值通过notify_cashvalue方法通知；
现金，数值，基金价值和基金份额也将通过notify_fund得到通知；
stop将被调用标志着操作结束。

完成正常的操作周期，Analyzers便可使用用于提取/输出信息的方法：

get_analysis：理想情况下（不强制执行）返回包含分析结果的类似dict的对象；
print使用标准的backtrader.WriterFile（除非overriden）来写入get_analysis的分析结果；
pprint（漂亮打印）使用Python pprint模块打印get_analysis的结果；

get_analysis创建一个成员属性self.ret（类型为collections.OrderedDict），Analyzers用来保存分析结果。
Analyzer的子类可以重写此方法来更改默认行为。

1.5. Analyzer模式

cktrader平台中Analyzer对象的开发揭示了两种不同模式来产生成分析结果：

在执行期间，通过notify_xxx和next方法收集信息，并在next中生成分析数据
例如，TradeAnalyzer仅使用notify_trade方法来生成统计信息。

收集（或不收集）上述信息，在stop方法一次性生成分析数据
SQN（系统质量编号）在notify_trade期间收集交易信息，在stop方法中生成统计信息

1.6. 一个尽可能简单的例子

```
m __future__ import (absolute_import, division, print_function,
                     unicode_literals)

import datetime

import backtrader as bt
import backtrader.analyzers as btanalyzers
import backtrader.feeds as btfeeds
import backtrader.strategies as btstrats

cerebro = bt.Cerebro()

# data
dataname = '../datas/sample/2005-2006-day-001.txt'
data = btfeeds.BacktraderCSVData(dataname=dataname)

cerebro.adddata(data)

# strategy
cerebro.addstrategy(btstrats.SMA_CrossOver)

# Analyzer
cerebro.addanalyzer(btanalyzers.SharpeRatio, _name='mysharpe')

thestrats = cerebro.run()
thestrat = thestrats[0]

print('夏普比率:', thestrat.analyzers.mysharpe.get_analysis())
```

执行它（已将其存储在analyzer-test.py中）：、

```
$ ./analyzer-test.py
Sharpe Ratio: {'sharperatio': 11.647332609673256}
```

这里没有绘图，因为SharpeRatio在计算结束时是单个值。
未完待续...

10.2. PyFolio概述

:意
从（至少）2017年7月25日起，pyfolio API已更改，并且create_full_tear_sheet不再有
ross_lev作为命名参数。因此，集成示例不起作用。

引用来自pyfolio主页(http://quantopian.github.io/pyfolio/)

yfolio是一个Python库，用于投资组合的财务绩效和风险分析，由Quantopian公司开发。可
很好的配合Zipline框架，进行回测。

，也可以与backtrader一起使用。需要支持库：

yfolio
及其依赖库（pandas, seaborn等）

:意
与版本0.5.1集成期间，需要更新依赖库的最新软件包。例如将seaborn从先前安装的0.7.0-
dev升级到0.7.1，这主要是因为缺少方法swarmplot造成的。

2.1. 用法

添加PyFolio分析器到cerebro中

```
cerebro.addanalyzer(bt.analyzers.PyFolio)
```

运行并获得第一个策略的返回：

```
strats = cerebro.run()
strat0 = strats[0]
```

使用您提供的名称或默认名称（pyfolio）检索分析器。例如：

```
pyfolio = strats.analyzers.getbyname('pyfolio')
```

- 使用analyzers的get_pf_items方法检索pyfolio以后需要用到的4个数据：

```
returns, positions, transactions, gross_lev = pyfoliozer.get_pf_items()
```

- 与pyfolio一起使用（这已经在backtrader系统之外）

一些与backtrader没有直接关系的使用说明

- pyfolio自动绘图可在Jupyter Notebook外部使用，但在内部使用效果最佳。
- pyfolio数据表的输出似乎无法在Jupyter Notebook之外运行，只能在Notebook内部工作。

如果希望使用pyfolio，最好还是在Jupyter Notebook中运行。

10.2.2. 简单代码

```
...
cerebro.addanalyzer(bt.analyzers.PyFolio, _name='pyfolio')
...
results = cerebro.run()
strat = results[0]
pyfoliozer = strat.analyzers.getbyname('pyfolio')
returns, positions, transactions, gross_lev = pyfoliozer.get_pf_items()
...
...
# pyfolio showtime
import pyfolio as pf
pf.create_full_tear_sheet(
    returns,
    positions=positions,
    transactions=transactions,
    gross_lev=gross_lev,
    live_start_date='2005-05-01',  # 指定日期
```

```
    round_trips=True)

# 当前节点的图表将展现
```

10.2.2.1. 相关内容

PyFolio analyzer的内部使用的相关参考内容。

10.3. Pyfolio集成

t #08中提出了对投资组合工具pyfolio的集成。

zipline和pyfolio之间的紧密集成，该教程的初学者认为这很难，但是其实pyfolio可以用于其他

分用例已经集成到backtrader中：

Analyzer基础结构
- 子Analyzer

TimeReturn Analyzer

一个主PyFolio Analyzer和3个简易的子Analyzer，加上一个pyfolio方法，以及它的依赖库as。

更新所有依赖项：

更新pandas
更新numpy
更新scikit-lean
更新seaborn

Unix环境下，时间完全取决于C编译器。 在Windows下，甚至安装了特定的Microsoft编译器本例中为Python 2.7）可能会失败。 但是，一个知名站点收集了Windows的最新软件包。 如果需要，请访问它： http://www.lfd.uci.edu/~gohlke/pythonlibs/

如果测试不通过，集成将无法完成，这就是为什么通常样例都选择最新的。

10.3.1. 不使用PyFolio

该示例使用random.randint决定何时买入/卖出，这只是一个简单能跑起来的例子：

```
$ ./pyfoliotest.py --printout --no-pyfolio --plot
```

输出：

```
Len,Datetime,Open,High,Low,Close,Volume,OpenInterest
0001,2005-01-03T23:59:59,38.36,38.90,37.65,38.18,25482800.00,0.00
买入  1000 @%23.58
0002,2005-01-04T23:59:59,38.45,38.54,36.46,36.58,26625300.00,0.00
买入  1000 @%36.58
卖出 500 @%22.47
0003,2005-01-05T23:59:59,36.69,36.98,36.06,36.13,18469100.00,0.00
...
卖出 500 @%37.51
0502,2006-12-28T23:59:59,25.62,25.72,25.30,25.36,11908400.00,0.00
0503,2006-12-29T23:59:59,25.42,25.82,25.33,25.54,16297800.00,0.00
卖出 250 @%17.14
卖出 250 @%37.01
```

E的默认时间周期内，随机选择了3个数据和几个买入和卖出操作。

3.2. 使用PyFolio

当在Jupyter Notebook内运行（包括内联绘图）时，pyfolio效果很好。

:意

nstrat作为参数与此处[]一起使用，则跳过传递的参数，使用默认参数运行。

```
tplotlib inline
```

```
m __future__ import (absolute_import, division, print_function,
                     unicode_literals)


ort argparse
ort datetime
ort random

import backtrader as bt


class St(bt.Strategy):
    params = (
        ('printout', False),
        ('stake', 1000),
    )

    def __init__(self):
        pass

    def start(self):
        if self.p.printout:
            txtfields = list()
            txtfields.append('Len')
            txtfields.append('Datetime')
            txtfields.append('Open')
            txtfields.append('High')
            txtfields.append('Low')
            txtfields.append('Close')
            txtfields.append('Volume')
            txtfields.append('OpenInterest')
            print(','.join(txtfields))

    def next(self):
        if self.p.printout:
            # Print only 1st data ... is just a check that things are running
            txtfields = list()
            txtfields.append('%04d' % len(self))
            txtfields.append(self.data.datetime.datetime(0).isoformat())
            txtfields.append('%.2f' % self.data0.open[0])
            txtfields.append('%.2f' % self.data0.high[0])
            txtfields.append('%.2f' % self.data0.low[0])
            txtfields.append('%.2f' % self.data0.close[0])
            txtfields.append('%.2f' % self.data0.volume[0])
            txtfields.append('%.2f' % self.data0.openinterest[0])
            print(','.join(txtfields))
```

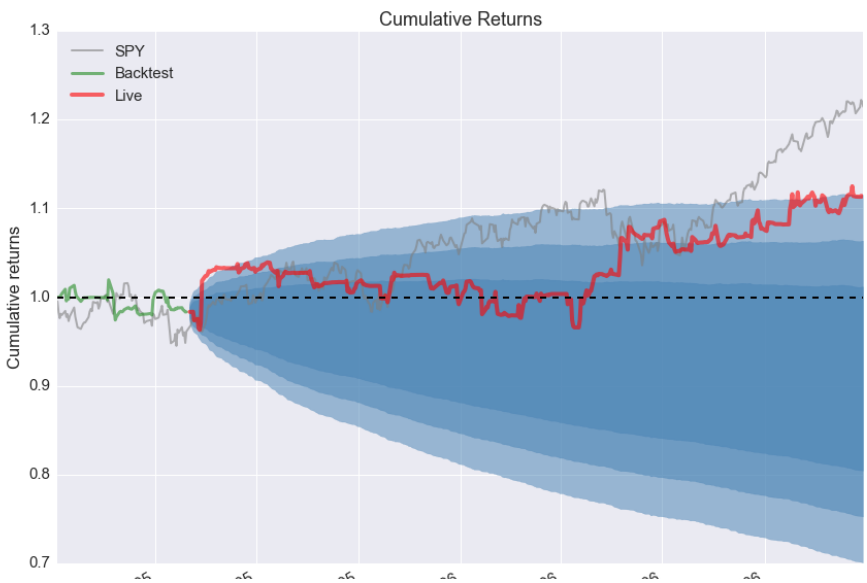
```
        # Data 0
        for data in self.datas:
            toss = random.randint(1, 10)
            curpos = self.getposition(data)
            if curpos.size:
                if toss > 5:
                    size = curpos.size // 2
                    self.sell(data=data, size=size)
                    if self.p.printout:
                        print('SELL {} @%{}'.format(size, data.close[0]))

            elif toss < 5:
                self.buy(data=data, size=self.p.stake)
                if self.p.printout:
                    print('BUY  {} @%{}'.format(self.p.stake, data.close[0]))


    runstrat(args=None):
    args = parse_args(args)

    cerebro = bt.Cerebro()
    cerebro.broker.set_cash(args.cash)

    dkwargs = dict()
    if args.fromdate:
        fromdate = datetime.datetime.strptime(args.fromdate, '%Y-%m-%d')
        dkwargs['fromdate'] = fromdate

    if args.todate:
        todate = datetime.datetime.strptime(args.todate, '%Y-%m-%d')
        dkwargs['todate'] = todate

    data0 = bt.feeds.BacktraderCSVData(dataname=args.data0, **dkwargs)
    cerebro.adddata(data0, name='Data0')

    data1 = bt.feeds.BacktraderCSVData(dataname=args.data1, **dkwargs)
    cerebro.adddata(data1, name='Data1')

    data2 = bt.feeds.BacktraderCSVData(dataname=args.data2, **dkwargs)
    cerebro.adddata(data2, name='Data2')

    cerebro.addstrategy(St, printout=args.printout)
    if not args.no_pyfolio:
        cerebro.addanalyzer(bt.analyzers.PyFolio, _name='pyfolio')

    results = cerebro.run()
    if not args.no_pyfolio:
        strat = results[0]
        pyfoliozer = strat.analyzers.getbyname('pyfolio')

        returns, positions, transactions, gross_lev = pyfoliozer.get_pf_items()
        if args.printout:
            print('-- RETURNS')
            print(returns)
            print('-- POSITIONS')
            print(positions)
            print('-- TRANSACTIONS')
            print(transactions)
            print('-- GROSS LEVERAGE')
            print(gross_lev)

        import pyfolio as pf
        pf.create_full_tear_sheet(
            returns,
            positions=positions,
            transactions=transactions,
            gross_lev=gross_lev,
            live_start_date='2005-05-01',
            round_trips=True)

    if args.plot:
        cerebro.plot(style=args.plot_style)


def parse_args(args=None):

    parser = argparse.ArgumentParser(
```

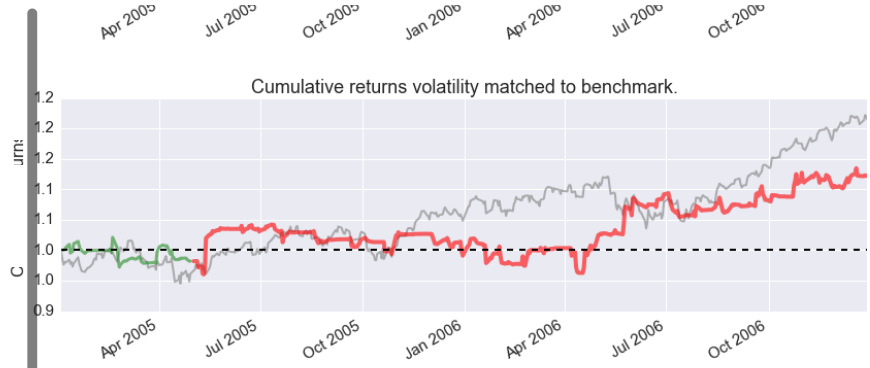
```
    formatter_class=argparse.ArgumentDefaultsHelpFormatter,
    description='Sample for pivot point and cross plotting')

parser.add_argument('--data0', required=False,
                    default='../../datas/yhoo-1996-2015.txt',
                    help='Data to be read in')

parser.add_argument('--data1', required=False,
                    default='../../datas/orcl-1995-2014.txt',
                    help='Data to be read in')

parser.add_argument('--data2', required=False,
                    default='../../datas/nvda-1999-2014.txt',
                    help='Data to be read in')

parser.add_argument('--fromdate', required=False,
                    default='2005-01-01',
                    help='Starting date in YYYY-MM-DD format')

parser.add_argument('--todate', required=False,
                    default='2006-12-31',
                    help='Ending date in YYYY-MM-DD format')

parser.add_argument('--printout', required=False, action='store_true',
                    help=('Print data lines'))

parser.add_argument('--cash', required=False, action='store',
                    type=float, default=50000,
                    help=('Cash to start with'))

parser.add_argument('--plot', required=False, action='store_true',
                    help=('Plot the result'))

parser.add_argument('--plot-style', required=False, action='store',
                    default='bar', choices=['bar', 'candle', 'line'],
                    help=('Plot style'))

parser.add_argument('--no-pyfolio', required=False, action='store_true',
                    help=('Do not do pyfolio things'))

import sys
aargs = args if args is not None else sys.argv[1:]
return parser.parse_args(aargs)
```

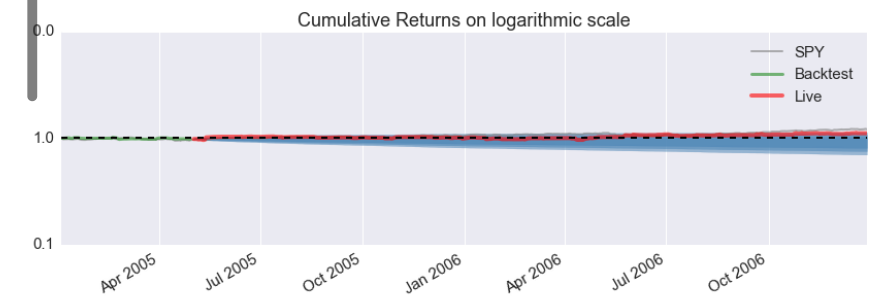
```
runstrat([])
```

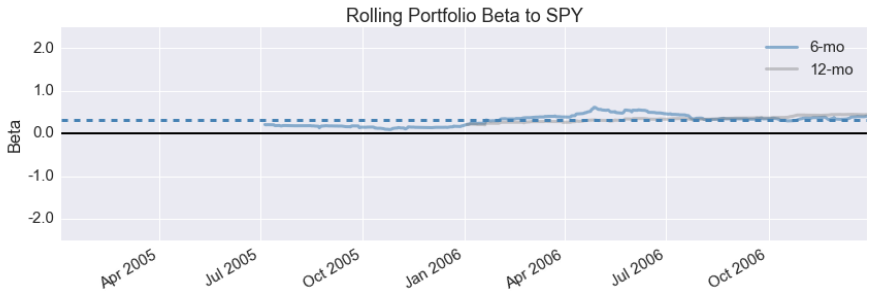
```
Entire data start date: 2005-01-03
Entire data end date: 2006-12-29


Out-of-Sample Months: 20
Backtest Months: 3
```

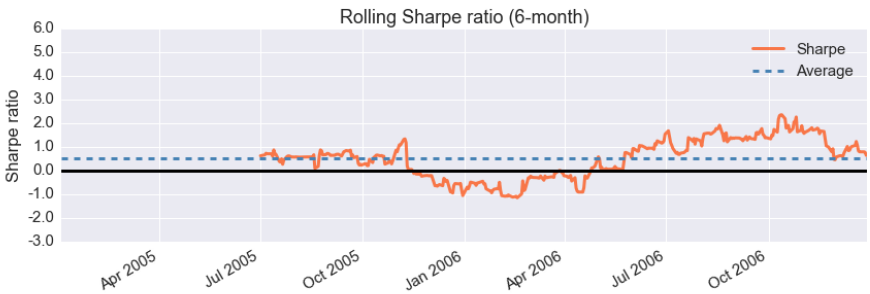
```
[-0.012 -0.025]
```

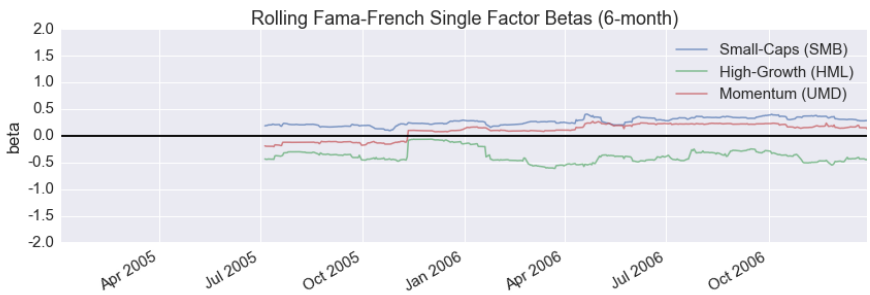












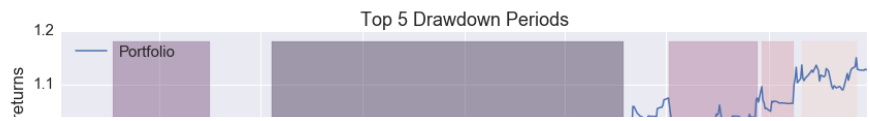
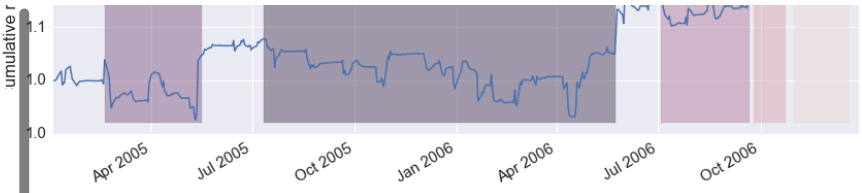
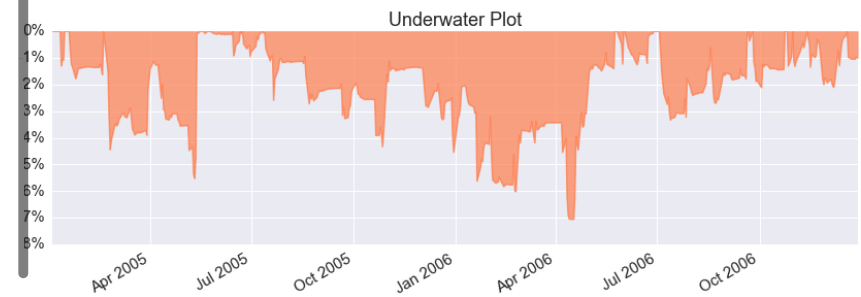
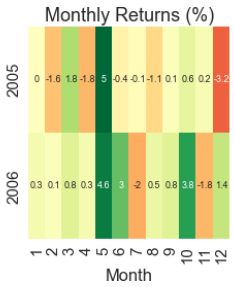
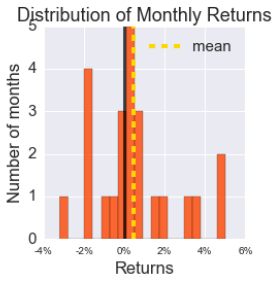


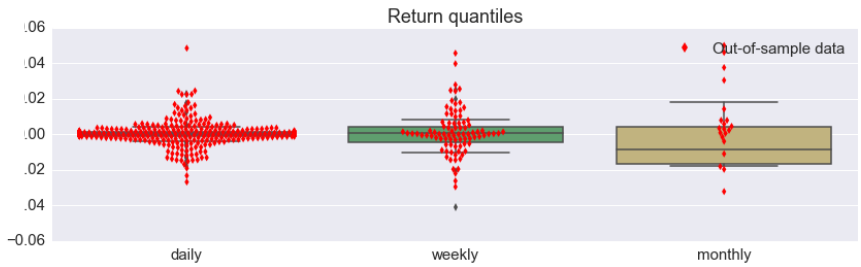
Table of Content

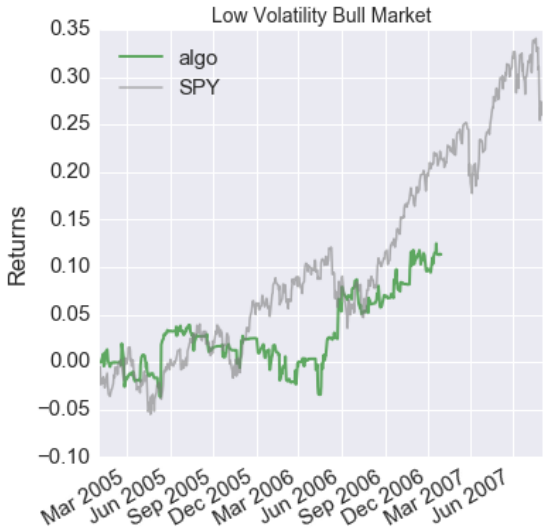


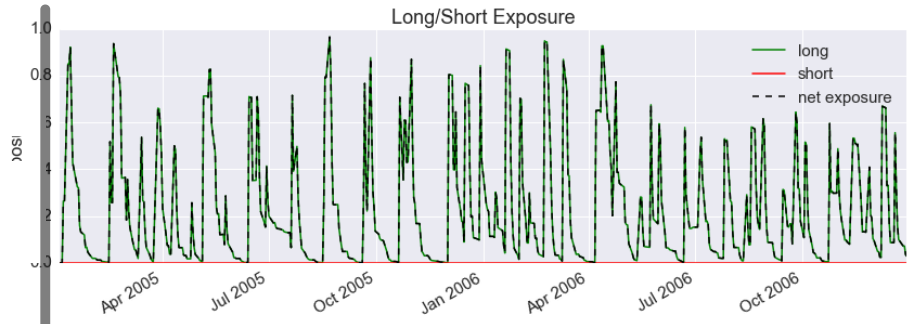


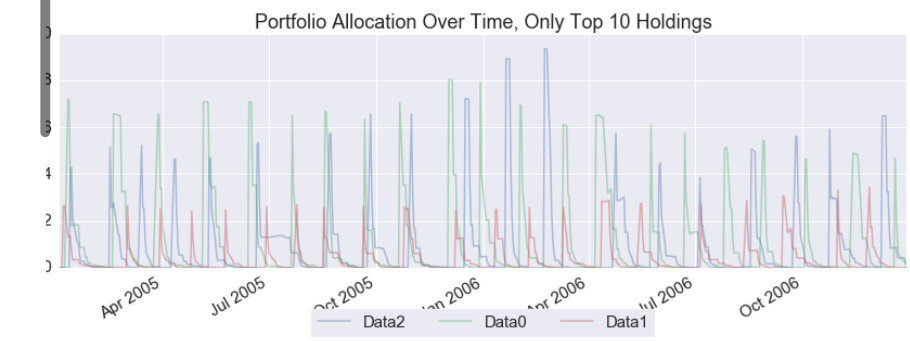


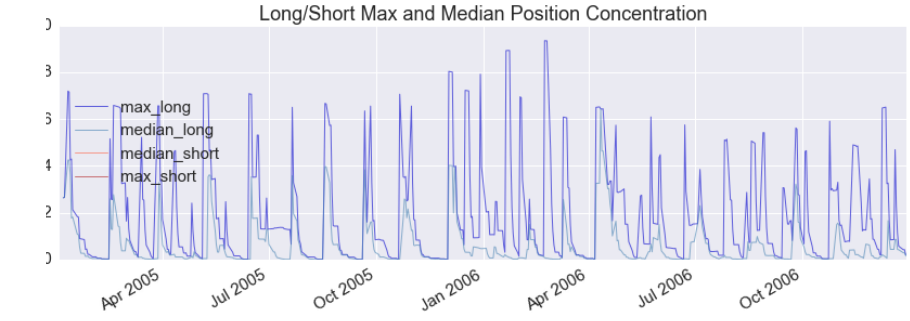


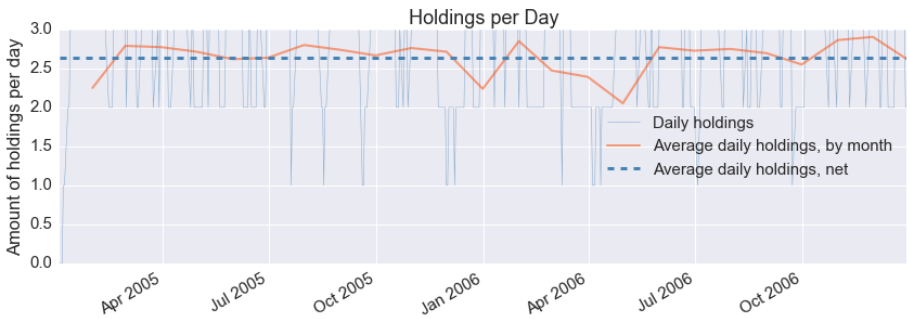


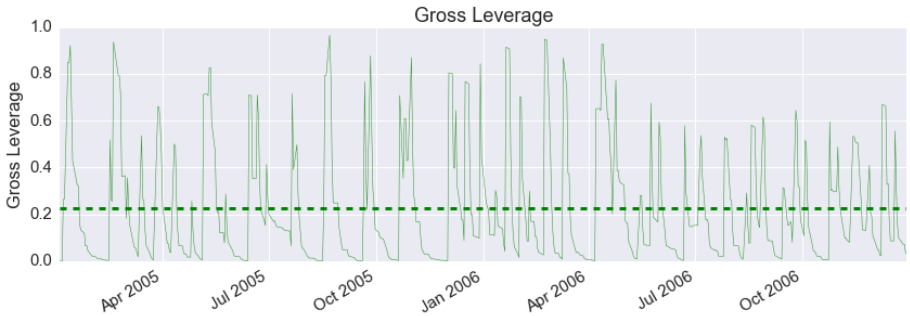




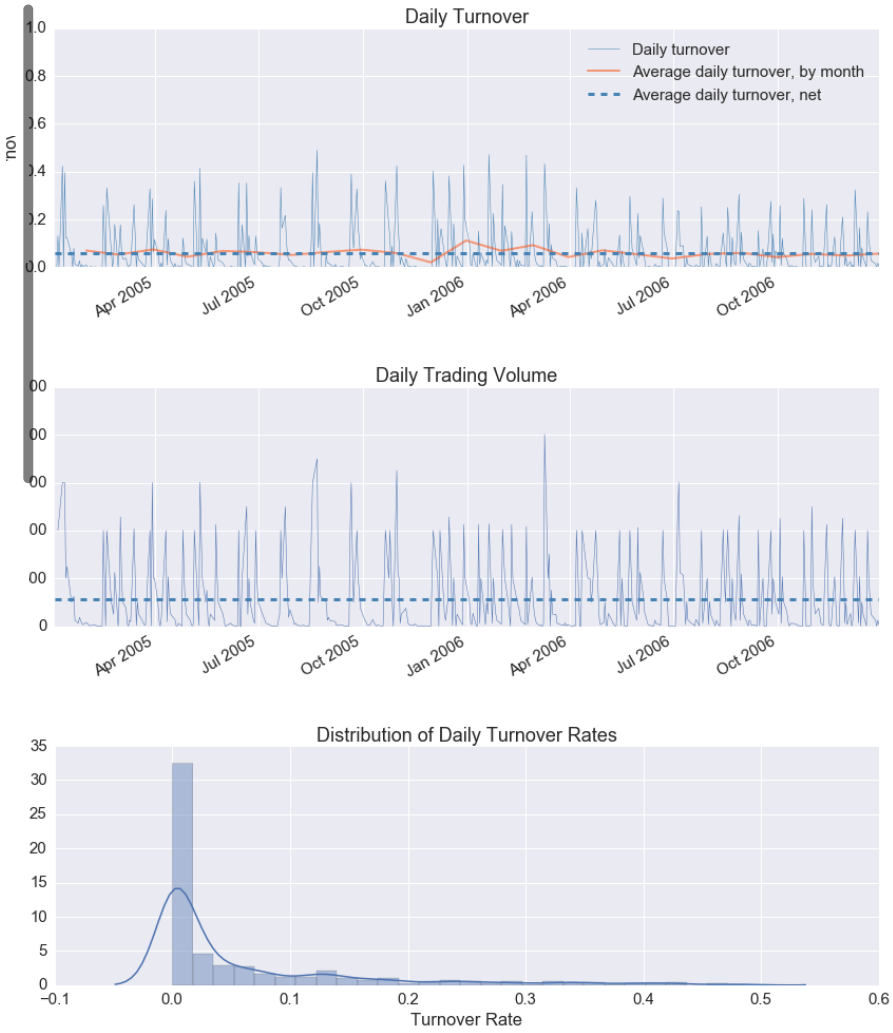




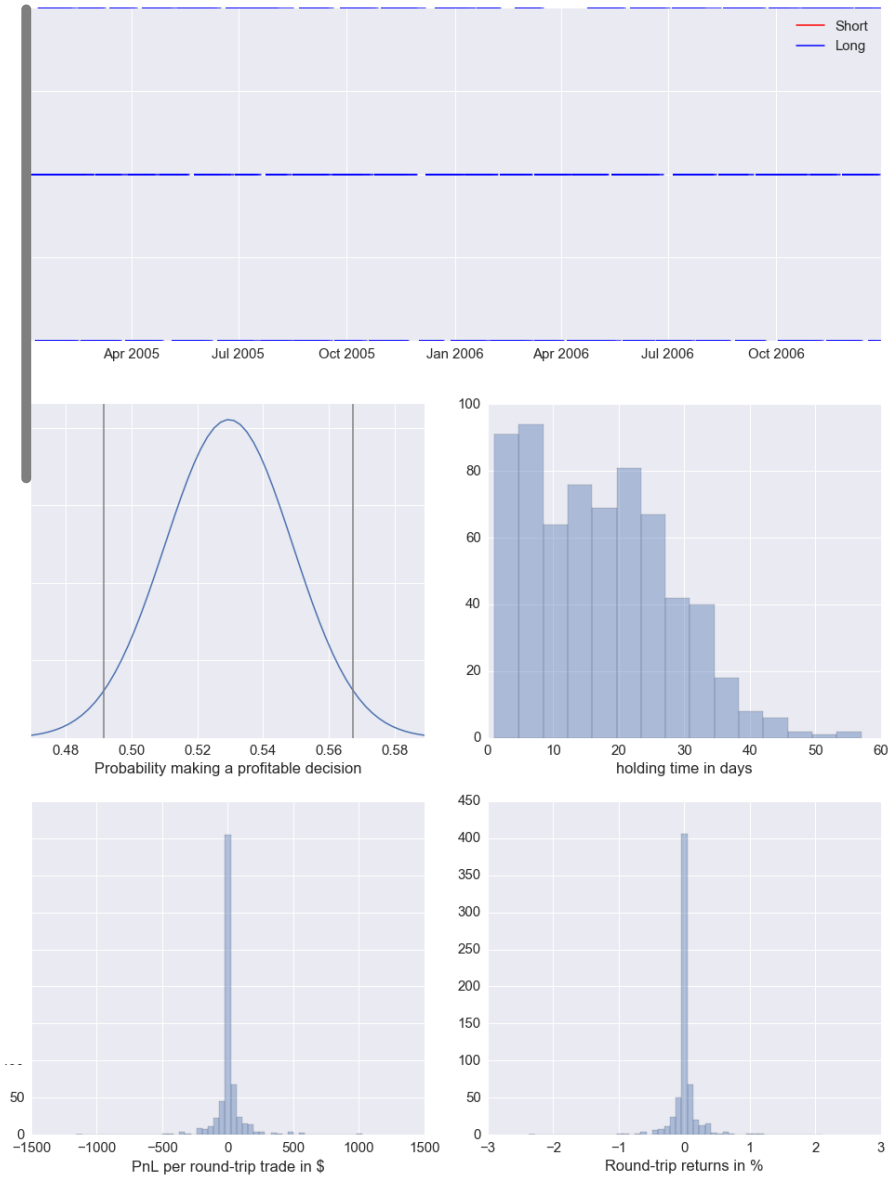





```
pyfolioplotting.py:1210: FutureWarning: .resample() is now a deferred operation
use .resample(...).mean() instead of .resample(...)
  **kwargs)
```




```
<matplotlib.figure.Figure at 0x23982b70>
```



使用例子：

```
$ ./pyfoliotest.py --help
usage: pyfoliotest.py [-h] [--data0 DATA0] [--data1 DATA1] [--data2 DATA2]
                      [--fromdate FROMDATE] [--todate TODATE] [--printout]
                      [--cash CASH] [--plot] [--plot-style {bar,candle,line}]
                      [--no-pyfolio]

Sample for pivot point and cross plotting

optional arguments:
  -h, --help            show this help message and exit
  --data0 DATA0         Data to be read in (default:
                        ../../datas/yhoo-1996-2015.txt)
  --data1 DATA1         Data to be read in (default:
                        ../../datas/orcl-1995-2014.txt)
  --data2 DATA2         Data to be read in (default:
                        ../../datas/nvda-1999-2014.txt)
  --fromdate FROMDATE   Starting date in YYYY-MM-DD format (default:
                        2005-01-01)
  --todate TODATE       Ending date in YYYY-MM-DD format (default: 2006-12-31)
  --printout            Print data lines (default: False)
  --cash CASH           Cash to start with (default: 50000)
  --plot                Plot the result (default: False)
  --plot-style {bar,candle,line}
                        Plot style (default: bar)
  --no-pyfolio          Do not do pyfolio things (default: False)
```

11. 观察者（Observers）

11.1. 观察者与统计

trader内部执行策略，主要处理交易数据和指标。 交易数据被添加到Cerebro实例中，并最终成为策略输入的一部分（被解析并做为实力的属性使用），而指标则由策略本身声明和管理。到目止，所有backtrader样本图都绘制了3样东西：

现金和价值（经纪人的钱变动情况）

- 交易（又名操作）
买卖下单

者存在于子模块backtrader.observers中。 Cerebro有参数支持自动添加（或不添加）到策略

stdstats（默认值：True） 如果遵守默认设置，则Cerebro将执行以下等效的代码：

```
import backtrader as bt
...
cerebro = bt.Cerebro()  #默认参数: stdstats=True
cerebro.addobserver(bt.observers.Broker)
cerebro.addobserver(bt.observers.Trades)
cerebro.addobserver(bt.observers.BuySell)
```

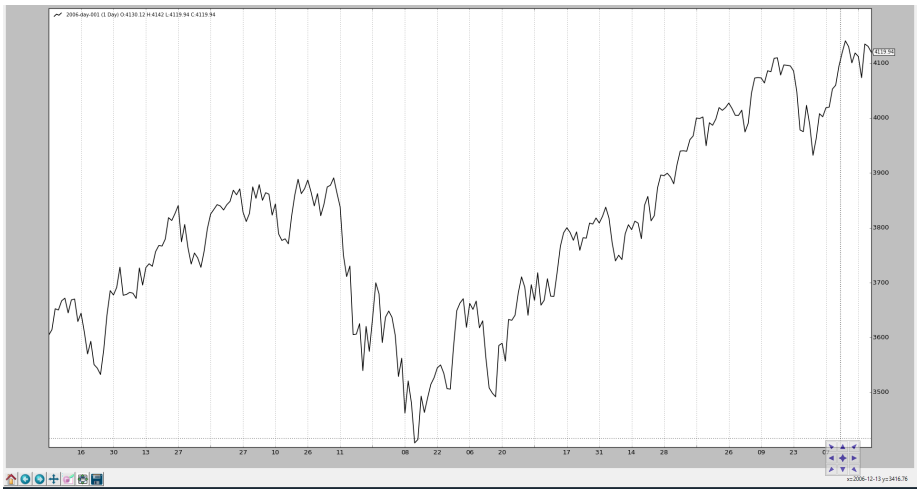
我们来看看带有这三个默认观察者的常规图表（即使没有发出任何订单，因此也没有交易发生，现金和投资组合价值也没有变化） ```python import backtrader as bt import backtrader.feeds as btfeeds

me == 'main': cerebro = bt.Cerebro(stdstats=False) cerebro.addstrategy(bt.Strategy) data = eds.BacktraderCSVData(dataname='../../datas/2006-day-001.txt') cerebro.adddata(data)

```
ebro.run()
ebro.plot()
```

(http://stoi.jusiu.com/bt/observers-default.png)
现在，让我们在创建Cerebro实例时将stdstats的值更改为False（也可以在调用run时完成）：
```python
cerebro = bt.Cerebro(stdstats=False)
```

图表现在有所不同。



未完待续...