

银河启睿策略中心—回测SDK使用说明

概述

DeepQuant 是由银河证券开发的量化交易策略框架（引擎），旨在降低投资者和机构参与量化投资的门槛。用户可以使用简单易懂的代码脚本编写金融模型和投资理念，并在历史行情下进行快速回测，以验证模型的有效性。DeepQuant 还支持构建复杂交易系统，通过实时行情驱动的模式交易，甚至可直接应用于实盘交易。

DeepQuant SDK

DeepQuant SDK 是银河证券策略中心提供的一系列量化工具，包含以下四个主要产品：

产品	资源
gid - 负责提供和管理金融数据的 API	使用说明
quest - 策略回测框架和引擎	使用教程 API 手册
factor - 因子投研工具：因子的编写及检验	使用说明 API 手册
oplib - 金融函数库	使用说明 API 手册

这一套便捷开发环境在网页平台中提供的所有功能均基于 DeepQuant SDK。现在，您可以轻松在本地通过 gid 调用金融数据，通过 quest 进行回测，通过 factor 挖掘因子，以及通过 oplib 进行金融计算。同时，您还可以访问本地数据，并使用熟悉的本地 IDE 进行断点调试。四个组件既各司其职又紧密配合，共同提供了流畅、完整的本地量化开发体验。

DeepQuant 回测框架quest

本文档着重介绍 DeepQuant 中 quest 回测框架的功能和细节。

如何阅读

本文档是 quest 的入门指南，为帮助您快速入门，请阅读以下部分：

- [快速上手](#)：构建对 quest 的整体认识并编写第一个可运行的策略。
- [进阶教程](#)：了解 quest 的进阶用法及编写策略的内部实现逻辑。
- [常见问题](#)：解答在使用 quest 中遇到的疑问。
- [示例策略](#)：获取编写策略的灵感。
- [API 手册](#)：了解全部 API 的详细用法。

特点与优势

1. **易用性**：用户几乎无需具备专业的计算机和开发知识即可编写专业的回测脚本和交易系统，DeepQuant 的初衷和目标是大幅度降低投资者和投资机构参与量化投研的门槛。
2. **高度封装**：DeepQuant 对量化交易的全流程进行了高度封装，将量化交易抽象成了一套事件驱动模型——策略编写者仅需要实现几段响应市场行情变动的逻辑、基于 DeepQuant 提供的丰富的数据接口做出逻辑判断，并发送目标交易信号，随后 DeepQuant 便会自动完成风控、撮合、账户管理等一系列复杂的后续工作，最后呈现出策略运行的结果及一系列直观的数据和图表供用户参考。

3. **Python 支持**：基于 Python 开发，您只需要具备 Python 的入门知识即可编写策略。如果您是有一定经验的 Python 开发者，您亦可以在策略中使用 Python 提供的几乎任何功能，诸如调用第三方库、访问网络和数据库或是使用更高性能的语言或分布式任务队列加速计算等等。DeepQuant 兼具易用性和扩展性，是量化交易投资者的神兵利器。

支持的品种

DeepQuant 支持中国市场几乎所有场内金融工具，具体各品种回测功能概览可参考下表：

品种	回测频率	功能说明	数据更新时间
股票	日级别（基础版）	由真实历史行情驱动的日级别股票回测： _ 将量化交易流程抽象，仅需编写一两个函数即可实现交易策略 _ 高度抽象化的下单 API，一行代码满仓、清仓、下单至目标仓位 _ 支持调用 Datac 数据 API，获取财务因子、行业分类等丰富数据 _ 支持以日线收盘价、开盘价撮合，支持月度、周度、日度定期调仓 _ 撮合引擎通过成交量限制、滑点等模拟真实市场冲击 _ 自动维护账户和持仓，自动处理 T+1、分红拆分等逻辑 _ 回测后输出丰富的数据，包括交割单、持仓历史、收益和风险指标等 _ 回测中支持引入第三方 python 包 * 框架高度模块化、插件化，可自由开发插件对接包括实盘在内的外部系统	每日收盘后更新
股票	分钟	由分钟级别行情驱动的股票回测： _ 支持上述日级别回测所有功能 _ 支持以分钟级别触发信号、分钟级别调仓 * 支持调取原始或动态复权的分钟线、五分钟线、小时线，最大限度模拟历史真实情况	每日收盘后更新
股票	tick	由 tick（快照数据）驱动的股票回测： _ 支持上述日级别回测所有功能 _ 回测逻辑由 level-1 tick 数据驱动 * 支持基于 tick 的拟真撮合模型	每日收盘后更新

品种	回测频率	功能说明	数据更新时间
商品、股指、 国债期货	日级别（基础版）	<p>由真实行情驱动的日级别期货回测：</p> <ul style="list-style-type: none"> _ 将量化交易流程抽象， 仅需编写一两个函数即可实现交易策略 _ 既可交易真实合约模拟真实交易， 亦可使用多种主力连续和平滑主力连续合约快速验证模型 _ 支持调用 Datac API， 获取主力合约数据 _ 支持以日线收盘价、开盘价撮合 _ 撮合引擎通过成交量限制、滑点等模拟真实市场冲击 _ 自动维护账户和持仓， 采用逐日盯市与真实市场保持统一 _ 回测后输出丰富的数据， 包括交割单、持仓历史、收益和风险指标等 _ 回测中支持引入第三方 python 包 <p>* 框架高度模块化、插件化， 可自由开发插件对接包括实盘在内的外部系统</p>	每日收盘后更新
商品、股指、 股债期货	分钟和 tick	<p>由分钟线或 tick（快照数据）驱动的期货回测：</p> <ul style="list-style-type: none"> _ 支持上述日级别回测的所有功能 _ 由分钟线或 tick 数据驱动回测 <p>* 支持多种撮合模型， 模拟不同市场表现</p>	每日收盘后更新
期权	日、分钟、 Tick	<p>由日线、分钟线或 tick 驱动的期权回测：</p> <ul style="list-style-type: none"> _ 支持股票、期货、期权等不同合约混合回测 _ 支持行权操作， 可主动发起行权（美式）或在到期日自动判定行权 <p>* 支持“行权滑点”， 针对快速变动的市场行情引入更为严苛的行权判定</p>	每日收盘后更新
可转债	日、分钟、 Tick	<p>由日线、分钟线或 tick 驱动的可转债回测：</p> <ul style="list-style-type: none"> _ 支持股票、可转债等不同合约的混合回测 _ 自动维护账户和持仓， 自动处理还本付息 <p>* 支持发起回售和转股， 转股后可正常进行股票交易</p>	每日收盘后更新
指数	日级别（基础版）	<p>由日线驱动指数回测：</p> <p>* 支持直接交易指数， 亦可与股票等品种混合回测</p>	每日收盘后更新
场外公募基金回测	日级别	<p>由真实行情驱动的日级别场外基金回测：</p> <ul style="list-style-type: none"> _ 将量化交易流程抽象， 仅需编写一两个函数即可实现交易策略 _ 支持申购、赎回基金， 并可以判断基金的申购赎回状态决定是否可交易 _ 支持以基金当日单位净值撮合 _ 支持设置前端费率， 自动处理基金申购赎回费用 _ 自动维护账户和持仓， 自动处理分红拆分等逻辑 _ 支持参数配置申购赎回到账时间， 自动处理 T+N 逻辑 	每日收盘后更新

快速上手

在编写策略之前，先介绍如何准备样例数据、生成样例策略和运行策略的功能，方便您快速了解和使用回测。如果您之前已经按照文档进行了相关操作，可忽略以下提示。

准备进行一次回测

DeepQuant SDK 的回测功能由`quest`提供，而回测所需的数据——除了您自己的数据之外——则由`gid`提供。

DeepQuant 回测依赖于历史行情数据，而这些数据需要预先被缓存到本地（股票、期货回测及各类品种的 tick 回测）。因此在进行回测之前，首先需要准备好历史行情数据。

取决于您期望进行研究的标的数量和回测频次，历史行情数据量可能会非常庞大。如果直接通过网络下载可能会耗时非常久，并且很可能会突破您账号的每日流量限制。如果您已经正式开通 DeepQuant SDK 权限，可以联系营业部为您提供具体行情数据流控数值。

我们先用一个简单的例子（大约会消耗 600MB 的流量）来演示一下使用 DeepQuant SDK 做回测的简便性。

准备回测所需数据

尊敬的试用客户：由于每日有一定的流量配额，按照前文描述的 600MB 流量消耗是可以满足数据缓存初始化的。同时，我们专门为试用账户准备了[不消耗计费流量的初始化方式](#)。

在开始之前，请确保已通过 `conda activate` 环境名 命令切换到已经装好 DeepQuant SDK 的虚拟环境。关于 DeepQuant 和 conda 虚拟环境的安装请参看阅读[环境安装](#)

通过网络的方式初始化数据缓存只需要运行如下命令：

```
deepquant update-data --base --minbar 000001.SZ
```

该命令在运行中将产生如下图所示的动态输出：

```
(venv) E:\Work\Testing\tgw\deepquant_py38_0.5.64>deepquant update-data --base --minbar 000001.SZ
开始更新日线及基础数据
[#####-----] 18%
```

命令运行完毕后如下，注意会有两行进度条：

```
(venv) E:\Work\Testing\tgw\deepquant_py38_0.5.64>deepquant update-data --base --minbar 000001.SZ
开始更新日线及基础数据
[#####] 100%
开始更新 1 只标的的分钟线数据：{'000001.SZ'}
[#####] 100%
```

该命令的目的是更新所有日线及基础数据，并且更新代号为 000001.SZ（平安银行）的分钟线行情数据。这样您就可以针对所有的股票和期货做日线回测，而仅对 平安银行 做分钟线回测。

注意：

- 如果您对需要做分钟线、tick 回测的合约有更复杂的需求，请运行 `deepquant update-data` 获得该命令的完整用法。
- 更新数据过程中，请不要运行回测，避免出现同时读写数据的情况。

试用客户初始化缓存的方式

我们为试用客户专门准备了不消耗计费流量的数据初始化方式，避免因为要下载 600MB 的数据而耗尽每日流量配额。在完成前面的安装过程后，您只需要运行下列命令：

```
deepquant download-data --sample
```

该命令将下载为您准备的样例数据，不消耗 datac 的每日流量配额，它会为您准备好真实的日级别历史行情数据、合约列表数据、分红拆分数据等回测所需的基础数据。之后如果您再运行 `deepquant update-data` 或该命令的其他参数形式（如前文提到的 `deepquant update-data --minbar 000001.SZ --base`，则对于已下载的部分只会进行增量更新，并不会消耗大量流量。

注意：在调取本节所描述的数据相关命令时，因为只有 DeepQuant 的回测功能依赖本地数据包，因此系统会自动检查是否已经安装了该产品，如无安装则会提示安装。

准备回测所需策略代码

在上述命令执行完毕后，将会在 `<用户目录>\.quest\bundle` 目录下创建历史行情数据的缓存文件。

这是 DeepQuant SDK 管理缓存文件的默认目录，您可以通过参数 `-d <完整路径>` 进行定制化。在回测时同样可以指定 `-d` 参数来更改 DeepQuant 读取回测历史文件的位置。

接下来您需要准备一个策略来进行回测。策略是交易决策逻辑的载体，用 Python 代码来体现。您只需要在策略中实现由 DeepQuant 指定的回调函数即可（详细用法请参考[策略引擎 API 文档](#)）。

但是现在并不需要急着去研究策略引擎的细节，我们已经为您准备好了几个直接能运行的策略。请您用 `cd` 命令切换到您希望放置样例策略的目录，然后运行下面的命令：

```
deepquant examples
```

该命令会在当前目录下创建一个名为 `exmaples` 的目录，其内容如下图所示：

```
C:\Users\guo\examples>dir
驱动器 C 中的卷没有标签。
卷的序列号是 B679-9F72

C:\Users\guo\examples 的目录
2024/11/14 17:54 <DIR>      .
2024/11/14 17:54 <DIR>      ..
2025/01/08 15:55      1,005 buy_and_hold.py
2024/11/14 17:54 <DIR>      data_source
2024/11/14 17:54 <DIR>      extend_api
2025/01/08 15:55      2,151 golden_cross.py
2025/01/08 15:55     22,343 IF1706_20161108.csv
2025/01/08 15:55      2,000 IF_macd.py
2025/01/08 15:55      2,110 macd.py
2025/01/08 15:55      5,532 pair_trading.py
2025/01/08 15:55      2,014 rsi.py
2025/01/08 15:55      822 run_code_demo.py
2025/01/08 15:55      436 run_file_demo.py
2025/01/08 15:55      1,031 run_func_demo.py
2025/01/08 15:55      1,623 subscribe_event.py
2025/01/08 15:55      1,280 test_pt.py
2025/01/08 15:55      4,619 turtle.py
                13 个文件      46,966 字节
                4 个目录      2,046,791,680 可用字节
```

您可以用惯用的 IDE 或者编辑器打开这些 Python 源码文件来查看代码，这里列举了一些常用的策略写法，以便您快速上手策略编写的 API。

我们现在以 `buy_and_hold.py` 策略为例，简单讲解一下策略的几个构成部分。当然您也完全可以跳过下面的代码讲解，直接到下一个环节，先看一下回测引擎能对策略做哪些分析，输出了哪些数据。

```
# 顾名思义，这是一个“买入并持有”的简单策略。在这个策略里，我们将在策略开始时买入平安银行，并持有到策略结束。
```

```

# 首先策略引入了quest的框架依赖，这是所有策略必须具备的。
from deepquant.quest.apis import *

# 在这个方法中编写任何的初始化逻辑。context对象是由引擎构建并传入的，这个对象内涵了关于整个策略
# 的信息，
# 这个对象也会出现在其他回调中，使用同一个实例。
def init(context):
    # 在这里定义了一个类似“全局变量”的变量。因为这个context对象实例会出现在别的回调中，因此在别的
    # 函数中
    # 也可以引用context.s1这个变量
    context.s1 = "000001.SZ"

    # 告诉引擎该策略的股票池包含了什么股票，在这里股票池只有“平安银行”一个，您还可以传入一个列表
    # 或一个
    # 指数代码
    update_universe(context.s1)

    # 创建一个变量，用来判断是否已经执行过买入操作。因为行情会不断触发回调，因此需要策略自行判断
    # 是否
    # 已经买入过，而不是在每一次行情触发时都执行买入
    context.fired = False

    # 日志会直接打印在命令行（标准输出）中，您可以通过将输出流转发到文件的方式将日志保存下来。
    logger.info("RunInfo: {}".format(context.run_info))

# 这个回调模拟的是每个交易日开盘前希望执行的一些操作，例如对昨天收盘后的情况做一些处理来指导今天的
# 交易，
# 但是在我们这个很简单的策略中并不需要这一回调，可以略过。
def before_trading(context):
    pass

# 这是前面提到的行情处理回调，也是整个策略的核心部分。行情是以K线的方式传入的，每当策略收到一个新
# 的行情
# （在回测的情况下，就是下一个时间单位的K线准备好）时，这个函数就会被触发一次。
# 除了context变量之外，bar_dict就是含有行情信息的一个字典结构，它的key是合约代码，值是引擎内定
# 义的
# Bar结构，包含了常见的开盘价、收盘价、最高最低价等信息。
def handle_bar(context, bar_dict):

    # 这里就是策略逻辑的主体了。
    # 我们先判断买入的逻辑是否已经触发过，如果没有触发过，说明是第一次收到行情，那么就进行买入
    # 如果已经触发过，则什么也不做。
    if not context.fired:
        # order_percent并且传入1代表买入该股票并且使其占有投资组合的100%
        logger.info("order_percent: {}".format(order_percent(context.s1, 1)))

        # 注意将代表是否买入过的变量设为True，确保只执行一次买入操作
        context.fired = True

# 类似前面的before_trading，这个回调函数模拟了每个交易日收盘后需要进行的一些处理。
# 在这个策略中并不需要做任何处理，因此直接略过了。
def after_trading(context):
    pass

```


运行回测！

到这里，您已经准备好了运行回测所需的所有条件，可以准备运行回测了。我们仍然以前面提到的 `buy_and_hold.py` —— 一个简单的买入并持有策略为例。

运行回测之前，您需要想好几个参数：

- 策略从那一天开始：起始时间
- 策略运行到那一天为止：结束时间
- 策略所使用的最大资金量是多少：账户资金
- 策略以什么频率进行回测：日线、分钟线或 tick

现在我们假设以**分钟线**频率，从**2018 年 1 月 1 日**开始到**2018 年 12 月 31 日**为止（引擎会自动按各个交易所公示的交易日期针对不同交易品种跳过非交易日，您只需要按您所想的日期指定即可，不必考虑节假日或周末的情况），账户资金**10 万元**。那么执行下列命令即可：

```
deepquant run -f examples/buy_and_hold.py -s 2018-01-01 -e 2018-12-31 -fq 1m --plot --account stock 100000
```

由于在上述策略运行命令中输入了 `--plot` 命令，因此在策略执行完毕后会弹出策略执行结果的图片，展示策略运行情况。如果您希望获得更多的策略运行结果以备分析，可以使用 `--report` 和 `-o` 命令来分别输出 `csv` 格式的报告和 `pickle` 格式的 Python 内存序列化文件。更多的配置参数可以通过下列命令查看：

```
deepquant run --help
```

您将会看到非常多的可配置项。随着使用的逐渐深入，这些配置项都会成为您的得力工具。

上手总结

至此您已经完成了所有 deepquant 的配置流程，并运行了一个策略。这里总结一下几个关键的命令，假设您已经配置了满足条件的 Python 环境，其实安装 DeepQuant SDK 是非常简单的。

从网络安装 DeepQuant SDK：

```
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple deepquant
```

下载试用数据：

```
deepquant download-data --sample
```

下载生产数据：

```
deepquant update-data --minbar 000001.SZ
```

以分钟线频率运行回测样例：

```
deepquant run -f examples/buy_and_hold.py -s 2018-01-01 -e 2018-12-31 -fq 1m --plot --account stock 100000
```

接下来我们会更详细地带您了解 DeepQuant SDK 的每一个细节。会对策略编写、运行策略和获取回测结果等模块进行详细介绍。我们建议您按照前面的内容先行配置好环境，后面的讲解将不再赘述环境的配置、数据的更新等内容。

第一个策略

如下展示的是一个简单的策略，该策略的基本逻辑是在策略运行的第一天半仓买入平安银行（000001.SZ）并持有至策略运行结束。

```
def init(context):
    context.fired = False

def handle_bar(context, bar_dict):
    if not context.fired:
        order_target_percent("000001.sz", 0.5)
        context.fired = True
```

将上述策略运行于 2019 年全年，运行结束后 DeepQuant 会展示出策略运行期间的收益曲线及部分收益和风险控制指标。



如下展示的是一个稍稍复杂一些的策略，该策略关注个股每日 MACD（指数平滑移动平均线）的情况，捕捉 MACD 和 SIGNAL（信号线）的交叉点作为买卖点：

```
import talib

def init(context):
    context.stock = "000001.sz"

    context.SHORTPERIOD = 12
    context.LONGPERIOD = 26
    context.SMOOTHPERIOD = 9
    context.OBSERVATION = 100

def handle_bar(context, bar_dict):
```



```

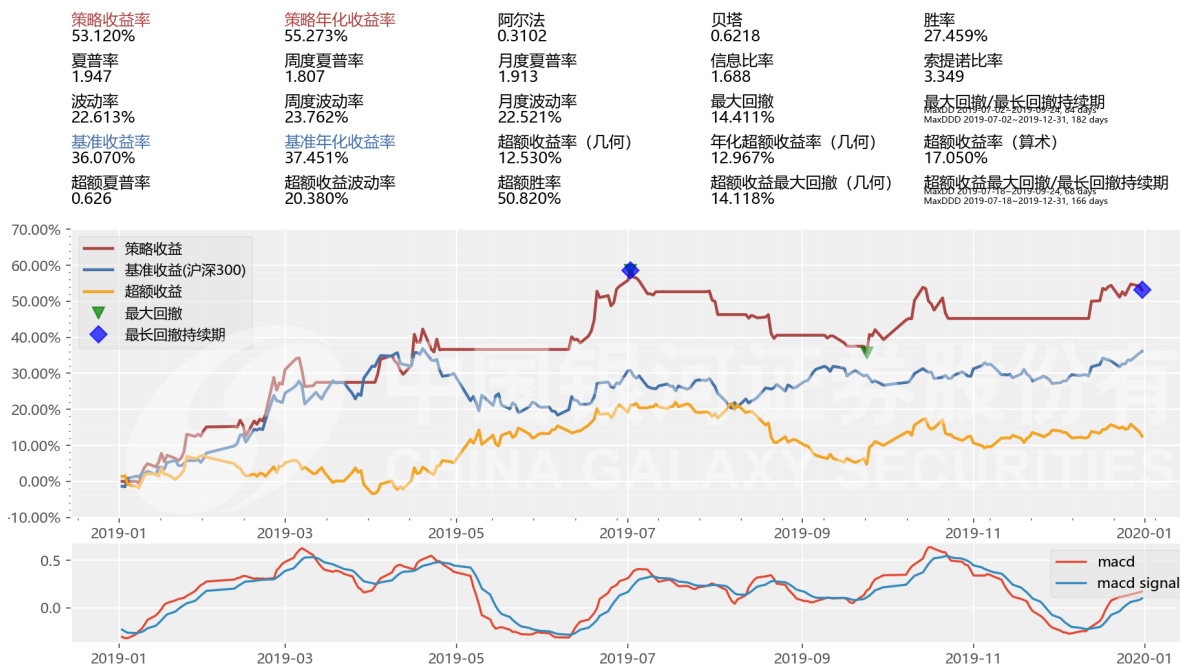
prices = history_bars(context.stock, context.OBSERVATION, '1d',
'close_price')
macd, macd_signal, _ = talib.MACD(
    prices, context.SHORTPERIOD, context.LONGPERIOD, context.SMOOTHPERIOD
)

if macd[-1] > macd_signal[-1] and macd[-2] < macd_signal[-2]:
    order_target_percent(context.stock, 1)

if macd[-1] < macd_signal[-1] and macd[-2] > macd_signal[-2]:
    if get_position(context.stock).quantity > 0:
        order_target_percent(context.stock, 0)

```

上述 MACD 策略在 2019 年全年的运行情况如下：



本章后文将以编写该 MACD 策略为目标引导读者了解和熟悉 DeepQuant 的基本用法。

基本概念

为了使用 DeepQuant，您需要了解几个常用名词，后文中将直接使用这些名词代指相应的概念。

策略

用户编写的代码逻辑的集合，这些代码的呈现方式可以是 .py 文件，可以是几个函数，亦可以是 python 中的字符串。策略内通常包含 init、handle_bar、before_trading 等[约定函数](#)，策略会被 DeepQuant 执行。

约定函数

策略中实现的有固定名字及参数的函数，如 `init(context)`、`before_trading(context)` 和 `handle_bar(context, bar_dict)` 等，这些函数会被 DeepQuant 在诸如策略初始化、每日盘前、k 线行情发生更新等时机调用。用户可以在策略中根据需求选择性地实现约定函数，并在这些函数中实现计算、发单等逻辑。

下文中会统称 `init`、`before_trading` 和 `after_trading` 三个约定函数为“盘外约定函数”，统称 `handle_bar` 和 `handle_tick` 为“盘中约定函数”。

完整的约定函数列表可查阅[约定函数 API 手册](#)

数据包

为了加速策略运行，DeepQuant 需要将部分策略运行所必须的数据存储于用户计算机本地，这些数据以文件形式存储，包括标的基础数据、交易日历数据和行情数据等，上述数据文件统称为“数据包”。数据包不完整可能导致策略运行出现报错或行为异常，数据包的更新方法详见[下载数据包](#)。

接口 (API)

DeepQuant 提供了很多可以在策略中调用的函数，其功能包括数据查询、账户和仓位查询和下撤单等等，这些函数及其返回的数据类型统称为 DeepQuant 的接口或 API。如果您也在使用 Databac，您可能会注意到部分 DeepQuant 提供的接口与 Databac 提供的接口功能和名称形似，但使用方法略有差异，请务必注意区分使用，上述差异形成的原因详见[常见问题](#)。

下载数据包

如上文介绍，为了加速策略运行，DeepQuant 需要将部分策略运行所必须的数据存储于用户计算机本地，所以在编写和运行策略前，您需要先下载或生成数据包至本地磁盘。安装 DeepQuant 后，您便可以在命令行（Windows）或终端（macOS/Linux）中执行命令以下载或更新数据包。

下载样例数据

首次使用 DeepQuant 时，您可以通过执行如下命令下载样例数据以快速体验回测功能。

```
deepquant download-data --sample
```

样例数据包包含完整的日线和基础数据，可供运行（DeepQuant 支持的）任意合约几乎全时间段的日级别回测。样例数据另外包含有限的分钟和 tick 数据，可供运行所提供的标的的分钟和 tick 回测，数据目录如下：

order_book_id	品种	时间段	频率
000001.SZ	股票	2018 年全年	分钟/tick
002891.SZ	股票	2018 年全年	分钟/tick
600185.SH	股票	2018 年全年	分钟/tick
600000.SH	股票	2018 年全年	分钟/tick
000300.SH	股票	2018 年全年	分钟/tick
IF1606	期货	该期货上市交易时间段内	分钟/tick
IF2002	期货	该期货上市交易时间段内	分钟/tick
NR2003	期货	该期货上市交易时间段内	分钟/tick
AG1612	期货	该期货上市交易时间段内	分钟
AU1612	期货	该期货上市交易时间段内	分钟
IO2002C3900	期权	该期权上市交易时间段内	分钟/tick
IO2002P3900	期权	该期权上市交易时间段内	分钟/tick
113010.SH	可转债	2018 年全年	分钟/tick

order_book_id	品种	时间段	频率
113011.SH	可转债	2018 年全年	分钟/tick

更新数据

您可以通过如下命令增量更新数据包。增量更新时数据来自于 Datac，更新数据包会占用您 Datac 许可中的连接数和流量。

```
deepquant update-data
```

上述命令可以通过传入参数以控制更新的数据品种，不传入参数时默认更新日线数据，详细的参数说明可通过运行 `deepquant update-data --help` 查看。

示例，运行如下命令以更新日线、平安银行的分钟线数据、所有螺纹钢期货的分钟线数据和 IO2002C3900 的 tick 数据：

```
deepquant update-data --minbar 000001.SZ --minbar RB --tick IO2002C3900
```

自定义数据包存储目录

默认情况下，数据包存储于用户目录下的 `.quest/bundle` 目录下，您在下载样例数据包和更新数据包时可通过 `-d` 参数指定自定义的数据包目录。需要注意，若您指定了非默认的数据包目录，需要在运行回测时指定同样的数据包目录。

例如，更新位于 D 盘下的 `user_bundle_path` 文件夹下的数据包

```
deepquant update-data -d D:\\user_bundle_path --minbar 000001.SZ --minbar RB --tick IO2002C3900
```

编写策略

完成数据包的更新后，就可以开始策略的编写了，本节以文档开头出现的 MACD 策略为例演示简单的策略如何设计和编写。

首先需要确定策略主要逻辑，单股票 MACD 策略逻辑如下：

- 明确要交易的目标证券，并在每个交易日计算 [MACD 线](#) 和 SIGNAL 线（MACD 线的均线），若 MACD 线突破 SIGNAL 线，则全仓买入目标证券，若 MACD 线跌穿 SIGNAL 线，则清仓。

上文提到，DeepQuant 将交易的整个过程抽象为几段不同的“市场时机”，策略开发者则需要将策略逻辑拆分为对不同“市场时机”的响应，这些时机包括：

- 初始化：一般用于进行策略全局的初始化工作，该阶段不能执行交易逻辑
- 盘前：一般用于执行每日交易前的准备工作，该阶段不能执行交易逻辑
- 行情更新：一般用于执行行情发生变动时的判断及交易逻辑，不同频率级别的策略触发的“时机”有所差异：
 - 日 k 线更新：在日级别的策略中触发，每个交易日触发一次，该阶段可以获取到所有标的当日及之前的日 k 线
 - 分钟 k 线更新：在分钟级别的策略中触发，每分钟触发一次，该阶段可以获取到当前分钟及之前的分钟 k 线

- tick 更新: tick 级别的策略中触发, 需预先“订阅”标的, 当订阅的标的 tick 发生更新时触发, 若订阅了多个标的则每个合约会分别触发
- 盘后: 用于执行每日交易后的逻辑, 如清理、计算、记录等, 该阶段不能执行交易逻辑

将上文确定的 MACD 策略逻辑进行拆分如下:

- 初始化: 明确要交易的目标证券, 本例使用平安银行 (000001.SZ)
- 日 k 线更新:
 - 获取过去一段时间日 k 线中的收盘价数据
 - 计算 MACD 和 SIGNAL 线
 - 判断两条均线是否发生了突破或跌穿
 - 若发生了突破或跌穿则执行开仓或清仓逻辑

逻辑已经明确, 接下来开始正式编码。

首先是初始化阶段。初始化阶段的逻辑需要写在名为 `init` 的函数中, 函数需要接受唯一的参数

`context`:

```
def init(context):  
    pass
```

`context` 变量顾名思义存储的是策略的上下文信息, 策略需要在各个“时机”之间传递的变量都可以存储在 `context` 中, 另外 `context` 中也提供一些内置的上下文相关的属性, 如访问 `context.now` 可以获取到当前“时机”运行的时间。具体到本例, 我们将目标证券的代码定义成变量存储在 `context` 中:

```
def init(context):  
    context.stock = "000001.sz"
```

接下来是日 k 线更新阶段。该阶段的逻辑需要写在名为 `handle_bar` 的函数中, 函数除了接受

`context` 参数外还接受第二个参数 `bar_dict`:

```
def handle_bar(context, bar_dict):  
    pass
```

`bar_dict` 顾名思义就是“dict of bar”, 存储 k 线对象的字典。例如访问平安银行当前 k 线的“收盘价”:

```
bar_dict["000001.sz"].close
```

在本例中, 单个收盘价是不够的, 为了计算均线, 我们需要获取近一段时间以来的收盘价序列。最常用的获取历史行情序列的接口是 `history_bars`, 该函数接受标的代码、序列长度、k 线频率和价格字段四个参数, 例如获取本例中标的证券过去 100 天日 k 线的收盘价序列:

```
# 四个参数分别为标的代码 context.stock, 100 天, 日线 '1d', 收盘价 'close_price'  
prices = history_bars(context.stock, 100, '1d', 'close_price')
```

接下来使用获取到的收盘价序列计算均线, MACD 作为常用的技术指标, 其计算逻辑不需要我们自己实现, 可以直接使用第三方库 [TA-Lib](#)。TA-Lib 中的 [MACD 函数](#) 接受四个参数, 分别为价格序列、短周期均线天数、长周期均线天数、SIGNAL 均线天数, 返回值有三个, 分别为 MACD 线、SIGNAL 线、MACD 和 SIGNAL 线的差值, 类型均为 [numpy.array](#)。本例中需要 MACD 和 SIGNAL 线就够了:

```
import talib

macd, macd_signal, _ = talib.MACD(prices, 12, 26, 9)
```

接下来判断两条均线间的突破和跌穿。所谓 MACD 突破 SIGNAL，即 MACD 的最后一个值大于 SIGNAL 的最后一个值，且 MACD 的倒数第二个值小于 SIGNAL 的倒数第二个值；相反，所谓 MACD 跌穿 SIGNAL，即 MACD 的最后一个值小于 SIGNAL 的最后一个值，且 MACD 的倒数第二个值大于 SIGNAL 的倒数第二个值：

```
if macd[-1] > macd_signal[-1] and macd[-2] < macd_signal[-2]:
    # MACD 突破 SIGNAL，此时应开仓
    pass

if macd[-1] < macd_signal[-1] and macd[-2] > macd_signal[-2]:
    # MACD 跌穿 SIGNAL，此时应平仓
    pass
```

最后，还剩下开仓和平仓逻辑，对于股票，DeepQuant 提供了六个下单接口，均可以用于开仓或平仓：

- order_shares: 用于按股数下单
- order_lots: 用于按手数下单
- order_value: 用于按价值下单
- order_percent: 用于按价值占当前账户总权益的比例下单
- order_target_value: 用于按目标仓位价值下单
- order_target_percent: 用于按目标仓位价值占账户总权益的比例下单

本例中全仓买入和清仓使用 `order_target_percent` 最为方便，该接口接受两个参数，分别为标的代码和目标仓位比例；另有第三个可选参数，为现价单价格，该参数不传表示市价下单。全仓买入和清仓即目标仓位比例为 1 或 0：

```
# 全仓买入
order_target_percent(context.stock, 1)

# 清仓
order_target_percent(context.stock, 0)
```

另外，为了提升效率及减少因下单失败而出现的 WARNING 日志，可以在清仓前进行判断，仅在当前有仓位时执行清仓。获取当前持仓的接口是 `get_position`，该函数接受标的代码为参数（对于期货等具有空头仓位的标的品种，该函数还接受第二个参数——持仓方向，用于控制查询哪个方向的持仓），返回对应标的品种的[持仓对象](#)，持仓对象具有 `.quantity` 属性，其值为持仓股数；

```
if get_position(context.stock).quantity > 0:
    # 仅当持仓股数大于零时执行清仓操作
    order_target_percent(context.stock, 0)
```

将以上代码集成到一起：

```
import talib
```

```
def init(context):
    context.stock = "000001.SZ"

def handle_bar(context, bar_dict):
    prices = history_bars(context.stock, 100, '1d', 'close_price')
    macd, macd_signal, _ = talib.MACD(prices, 12, 26, 9)

    if macd[-1] > macd_signal[-1] and macd[-2] < macd_signal[-2]:
        order_target_percent(context.stock, 1)

    if macd[-1] < macd_signal[-1] and macd[-2] > macd_signal[-2]:
        if get_position(context.stock).quantity > 0:
            order_target_percent(context.stock, 0)
```

注意，将代码中使用到的一些常量定义为全局变量或 `context` 的属性而不是埋没于代码中是一个好习惯，如上述代码中 `talib.MACD` 的后三个参数。可以将上述代码稍作修改：

```
import talib

def init(context):
    context.stock = "000001.SZ"

    context.SHORTPERIOD = 12
    context.LONGPERIOD = 26
    context.SMOOTHPERIOD = 9
    context.OBSERVATION = 100

def handle_bar(context, bar_dict):
    prices = history_bars(context.stock, context.OBSERVATION, '1d',
                          'close_price')
    macd, macd_signal, _ = talib.MACD(
        prices, context.SHORTPERIOD, context.LONGPERIOD, context.SMOOTHPERIOD
    )

    if macd[-1] > macd_signal[-1] and macd[-2] < macd_signal[-2]:
        order_target_percent(context.stock, 1)

    if macd[-1] < macd_signal[-1] and macd[-2] > macd_signal[-2]:
        if get_position(context.stock).quantity > 0:
            order_target_percent(context.stock, 0)
```

运行策略

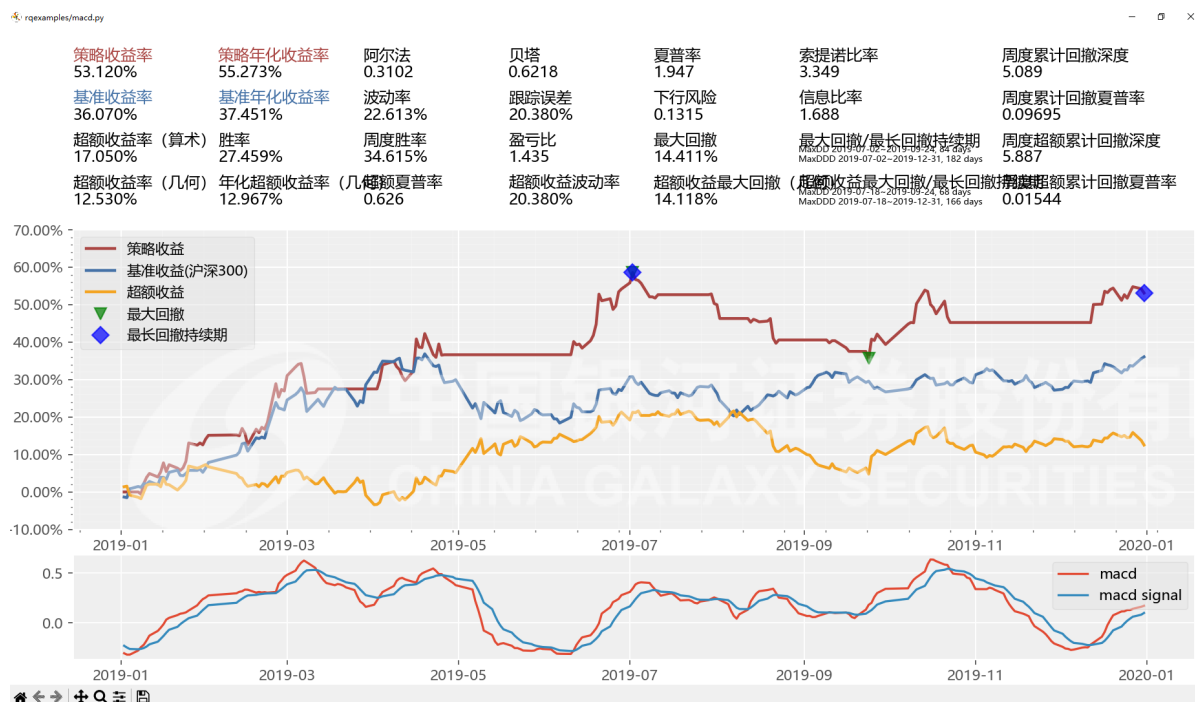
DeepQuant 提供了多种入口以供运行策略，本节介绍其中最常用的两种。

使用终端命令运行策略

将上一节编写好的策略写入扩展名为 `.py` 的文件中，例如 `macd.py`，并在命令行执行如下命令：

```
deepquant run -f macd.py -a stock 100000 -s 20190101 -e 20191231 -bm 000300.SH -p
```


敲击 Enter 键之后，DeepQuant 便开始执行，完成后会弹出类似下图的窗口，窗口内包括一些收益风险指标及收益率曲线图：



上边的运行命令由几部分组成：

- `deepquant`：DeepQuant 所有命令行工具的总入口，执行 `deepquant --help` 以查看所有可用的功能
- `run`：用于运行策略的子命令，
- 参数：策略运行的各种选项，顺序不限，部分参数需要传入参数值
 - `-f macd.py`：指定运行的策略文件，支持绝对路径或相对路径
 - `-a stock 100000`：指定股票账户的初始资金为十万元，DeepQuant 支持股票（`stock`）、期货（`future`）两种账户，策略交易不同品种的标的需要配置对应账户的初始资金
 - `-s 20190101`：回测运行的初始时间为 2019 年 1 月 1 日，回测实际上会从不早于该日期的第一个交易日开始运行
 - `-e 20191231`：回测运行的终止时间为 2019 年 12 月 31 日，回测实际上会运行至不晚于改日期的最后一个交易日
 - `-bm 000300.SH` 使用沪深三百指数（000300.SH）作为回测运行的基准，该基准用于计算 Alpha、Beta 等基于超额收益的指标，基准也会和策略收益一起展示在收益曲线图上
 - `-p`：策略运行结束后展示收益曲线图

除这些参数外，您还可以执行 `deepquant run --help` 以查看运行策略支持传入的更多参数。

使用函数入口运行策略

除命令行入口外，DeepQuant 也提供了函数入口以供在其他脚本中调用运行。最常用的是 `run_func` 函数，该函数接受几个关键字参数，分别为存储设置项的字典以及策略实现的约定函数：

```
config = {
    "base": {
        "accounts": {
            "STOCK": 100000,
```

```

    },
    "start_date": "20190101",
    "end_date": "20191231",
  },
  "mod": {
    "sys_analyser": {
      "plot": True,
      "benchmark": "000300.SH"
    }
  }
}

if __name__ == "__main__":

    from deepquant.quest.alpha import run_func

    run_func(config=config, init=init, handle_bar=handle_bar)

```

这里传入的 config 与上文中命令行运行所传入的参数功能是相同的。完整的配置列表可查阅 [API 手册](#)。

除了上述三个参数外，run_func 还可以接受其他约定函数作为参数，DeepQuant 也另外提供了具有不同功能的其他函数入口，详细信息可查阅 API 手册中[入口函数](#)部分。

您可以对策略回测进行以下参数的设置：

基础参数：

- 开始日期：回测期间的开始日期
- 结束日期：回测期间的结束日期 - 如果回测结束日期在今天之后，将会自动使用最后一天的历史数据
- 起始资金：回测的起始资金 - 您将使用多少钱去投资策略
- 回测频率：可以选择日回测、分钟回测以及 tick 级别回测

高阶设置：

- [基准合约](#)：设置策略表现的对照基准
- [佣金倍率](#)：设置策略佣金在默认佣金基础上的倍数
- [保证金倍率](#)：设置策略保证金在默认保证金率基础上的倍数
- [撮合方式](#)：日频回测可以选择'当前 bar 收盘'；分钟回测时，可以选择'当前 bar 收盘'或'下一 bar 开盘'；tick 级别回测时，目前支持'己方最优'、'对手方最优'和'最新价成交'
- [滑点](#)：交易中理想成交价和实际成交价的差异。在进行策略回测时，用户可以设置一个滑点参数，来提高买入价或降低卖出价，模拟实际交易中出现的滑点。目前支持的滑点类型有百分比和跳/手，百分比：按照成交价的一定比例进行恶化，取值范围 [0, 1)，例如，当股票市价为 10 元，参数设为 0.1，则回测时买入价为 $10 + 10 * 0.1 = 11$ 元；跳/手：按照最小价格变动单位对成交价进行恶化，例如，上例中设置值为 5，最终成交价将变成 10.05
- [分红再投资](#)：基金分红将按照当日净值折算成基金份额；股票分红将按照分红当日收盘价折算为仓位
- 允许买空：回测中可以允许股票买空
- 限制成交量：策略发单成交量将不能超过 bar 成交量的一定比例
- 成交百分比：与限制成交量联动，用户可以设置成交的比例

获取结果

使用 `run_func` 运行策略时，该函数会返回一个字典，这个字典包含了众多策略运行时产生的数据，您可以查看这些数据以了解策略的运行情况，或对策略运行的结果加以分析。

```
result = run_func(config=config, init=init, handle_bar=handle_bar)
```

如获取策略的指标汇总

```
result['sys_analyser']['summary']

# Out:
# {'strategy_name': 'strategy',
#  'start_date': '2019-01-02',
#  'end_date': '2019-12-31',
#  'strategy_file': 'strategy.py',
#  'run_type': 'BACKTEST',
#  'STOCK': 100000.0,
#  'alpha': 0.22,
#  'beta': 0.605,
#  'sharpe': 1.813,
#  'information_ratio': 0.468,
#  'downside_risk': 0.135,
#  'tracking_error': 0.206,
#  'sortino': 0.191,
#  'volatility': 0.226,
#  'max_drawdown': 0.148,
#  'total_value': 148613.493,
#  'cash': 563.493,
#  'total_returns': 0.486,
#  'annualized_returns': 0.506,
#  'unit_net_value': 1.486,
#  'units': 100000.0,
#  'benchmark_total_returns': 0.361,
#  'benchmark_annualized_returns': 0.375}
```

`result['sys_analyser']` 字典中另外有交易流水、每日的账户、持仓等信息：

key	value 格式	说明
summary	dict	回测的收益和风险指标汇总
trades	DataFrame	交易流水
portfolio	DataFrame	投资组合中每日现金、权益、市值、净值等数据
stock_account	DataFrame	股票账户每日现金、权益、市值等数据
stock_positions	DataFrame	股票账户下的每日持仓情况

使用命令行运行策略的情况下，因为无法直接获取到上述返回值，您可以通过传入参数的方式要求 DeepQuant 将回测结果写入文件。

保存回测结果

在 DeepQuant 回测中，加入以下参数至启动命令中，可以将回测结果写入文件。

- `-o result.pkl`：用于将回测结果以 pickle 形式存储至 pickle 文件，该 pickle 文件与 `run_func` 返回的字典内容相同
- `--report report`：用于将回测结果以 csv 报告的形式输出至 report 目录，这些文件内容与 `run_func` 返回的结果相同，只是使用了更便于直接查看和分析的格式呈现：

文件名	说明
report.xlsx	所有以下文件的汇总 excel 表
summary.csv	回测的收益和风险指标
portfolio.csv	投资组合中每日现金、权益、市值、净值等数据
stock_account.csv	股票账户每日现金、权益、市值等数据
stock_positions.csv	股票账户下的每日持仓情况
trades.csv	交易流水

```
deepquant run -f macd.py -a stock 100000 -s 20190101 -e 20191231 -p -bm 000300.SH -o result.pkl --report report
```

使用 pandas 读取回测报告为 DataFrame 对象的示例：

```
import pandas as pd
import os

portfolio_df = pd.read_csv(os.path.join("report", "portfolio.csv"),
encoding="GBK")
stock_account_df = pd.read_csv(os.path.join("report", "stock_account.csv"),
encoding="GBK")
stock_positions_df = pd.read_csv(os.path.join("report", "stock_positions.csv"),
encoding="GBK")
summary_df = pd.read_csv(os.path.join("report", "summary.csv"), encoding="GBK")
trades_df = pd.read_csv(os.path.join("report", "trades.csv"), encoding="GBK")
print(trades_df)

#           datetime  commission  ...  trading_datetime  transaction_cost
# 0  2018-01-02 09:31:00    798.5184  ...  2018-01-02 09:31:00            798.5184
```

使用 pickle，读取回测结果文件 result.pkl 的示例：

```
import pickle
with open("result.pkl", "rb") as f:
    result = pickle.load(f)

result.keys()
# Out[ ]: dict_keys(['trades', 'summary', 'benchmark_portfolio', 'portfolio',
# 'stock_positions', 'stock_account'])

result['trades']
# Out[ ]:
#               commission    ...    transaction_cost
# datetime                ...
# 2019-01-02 15:00:00      79.4016    ...            79.4016
```

收益指标

回测收益率: 策略在期限内的收益率。

$$R_{\text{回测}} = \frac{\text{期末投资组合总权益} - \text{期初投资组合总权益}}{\text{期初投资组合总权益}}$$

年化收益率: 采用了复利累积的[年化方式](#)计算得到的年化收益。

$$R_{\text{年化}} = (1 + R)^{\frac{1}{t}} - 1$$

$$t = \frac{\text{策略运行的交易日}}{252}$$

$$R = \text{累计收益率}$$

基准收益率: 相同条件下，一个简单的买入并持有基准合约策略的收益率（默认基准合约为沪深 300 指数，这里假设指数可交易，最小交易单位为 1）。

$$R_{\text{基准}} = \frac{\text{买入并持有至期末投资组合总权益} - \text{期初投资组合总权益}}{\text{期初投资组合总权益}}$$

每日收益率: 通过投资组合权益计算出的日收益率。

$$R_{\text{每日}} = \frac{\text{当前交易日总权益} - \text{前一交易日总权益}}{\text{前一交易日总权益}}$$

阿尔法(alpha, α): [CAPM](#)模型表达式中的残余项。表示策略所持有投资组合的收益中和市场整体收益无关的部分，是策略选股能力的度量。当策略所选股票的总体表现优于市场基准组合成分股时，阿尔法取正值；反之取负值。

$$\alpha = E[r_p - (r_f + \beta(r_b - r_f))]$$

其中 r_p 为策略所持有投资组合收益； r_f 为无风险组合收益； β 为 CAPM 模型中的贝塔系数； $E[\cdot]$ 表示随机变量的期望。

风险指标

年化波动率(volatility, σ_t): 策略收益率的标准差，最常用的风险度量。波动率越大，策略承担的风险越高。这里假设一年有 252 个交易日。

$$\sigma = \sqrt{\frac{1}{252} \sum_{i=1}^n (r_p(i) - \bar{r}_p)^2}$$

其中， n 为回测期内交易日数目； $r_t(i)$ 表示第 i 个交易日策略所持有投资组合的日收益率； \bar{r}_p 为回测期内策略日收益率的均值。

年化跟踪误差(tracking error,\$\sigma_t\$): 纯多头主动交易策略（阿尔法策略和基准择时策略）收益和市场基准组合收益之间差异的度量。跟踪误差越大，意味着策略所持有投资组合偏离基准组合的程度越大。需要注意，跟踪误差不适用于多-空结合的对冲策略的风险评估。

$$\sigma_t = \sqrt{\frac{1}{n-1} \sum_{i=1}^n [r_{pa}(i) - \bar{r}_{pa}]^2}$$
$$r_{pa}(i) = r_p(i) - r_b(i)$$

其中，\$n\$为回测期内交易日数量；\$r_{pa}(i)\$,\$r_p(i)\$,\$r_b(i)\$分别表示第 \$i\$ 个交易日策略所持有投资组合的日主动收益、日收益率和基准组合的日收益率。

年化下行波动率(downsiderisk,\$\sigma_d\$): 相比波动率，下行波动率对收益向下波动和向上波动两种情况做出了区分，并认为只有收益向下波动才意味着风险。在实际计算中，我们统一使用无风险组合收益为目标收益，作为向上波动和向下波动的判断标准。

$$\sigma_d = \sqrt{\frac{1}{n-1} \sum_{i=1}^n [r_p(i) - r_f(i)]^2 \cdot I(i)}$$
$$I(i) = \begin{cases} 1, & \text{if } r_p(i) < r_f(i) \\ 0, & \text{if } r_p(i) \geq r_f(i) \end{cases}$$

其中，\$n\$为回测期内交易日数量；\$r_p(i)\$,\$r_f(i)\$分别表示第 \$i\$ 个交易日策略所持有投资组合的日收益率、无风险组合的日收益率；\$I(i)\$为指示函数(indicator function)，如果第 \$i\$ 个交易日策略所持有投资组合收益低于无风险组合收益，则标记为 1（向下波动），否则标记为 0（向上波动）。

贝塔(beta,\$\beta\$): CAPM 模型中市场基准组合项的系数，表示资产收益对市场整体收益波动的敏感程度。

$$\beta = \frac{\text{Cov}(r_{p,e}, r_{b,e})}{\text{Var}(r_{b,e})}$$

其中\$r_{p,e}\$为策略收益率；\$r_{b,e}\$为市场基准组合收益率；\$\text{Cov}(\cdot)\$表示协方差；\$\text{Var}(\cdot)\$表示方差

Beta 值	解释	举例
$\beta < 0$	投资组合和指数基准的走向通常反方向	反向指数 ETF 或空头头寸
$\beta = 0$	投资组合和指数基准的走向没有相关性	固定收益产品，他们的走向通常和股市不相关
$0 < \beta < 1$	投资组合和指数基准的走向相同，但是比指数基准的移动幅度更小	稳定的股票，比如制作肥皂的公司的股票，通常和市场的走势相同，但是受到每日的波动影响更小
$\beta = 1$	投资组合和指数基准的走向相同，并且和指数基准的移动幅度贴近	蓝筹股，指数中占比重大的股票
$\beta > 1$	投资组合和指数基准的走向相同，但是比指数基准的移动幅度更大	受每日市场消息或是受经济情况影响很大的股票

最大回撤(max drawdown): 在回测期内，在任一交易日往后推，策略总权益走到最低点时收益率回撤幅度的最大值。最大回撤是评估策略极端风险管理能力的重要指标。其计算方式如下：

$$Drawdown_t = \begin{cases} 0 & \text{if } NET_t = \min_{j \geq t} NET_j \\ \frac{NET_t - \min_{j \geq t} NET_j}{NET_t} & \text{else} \end{cases}$$
$$NET \text{ 为某期净值}$$

$$MaxDrawdown = \max(Drawdown_t)$$

风险调整后收益指标

夏普率(sharpe ratio): 衡量策略相对于无风险组合的表现，是策略所获得风险溢价的度量——即如果策略额外承担一单位的风险，可以获得多少单位的收益作为补偿。

$$Daily \space Sharpe \space Ratio = \frac{\bar{r_e}}{\sigma_e}$$

$$\bar{r_e} = \frac{1}{n} \sum_{i=1}^n [r_p(i) - r_f(i)]$$

$$\sigma_e = \sqrt{\frac{1}{n-1} \sum_{i=1}^n [r_p(i) - r_f(i) - \bar{r_e}]^2}$$

$$Sharpe \space Ratio = \sqrt{252} \cdot Daily \space Sharpe \space Ratio$$

其中 $\bar{r_e}$ 为回测期内策略日超额收益率均值； n 为回测期内交易日数目； $r_p(i)$, $r_f(i)$ 分别为第 i 个交易日策略所持有投资组合的日收益率以及无风险组合日收益率； σ_e 为策略超额收益率的波动率。

索提诺比率(sortino ratio): 衡量策略相对于目标收益的表现。其使用下行波动率作为风险度量，因此区别于夏普率。在目前的计算中，我们使用无风险组合收益作为目标收益，以此作为区分向上波动和向下波动的标准。

$$Daily \space Sortino \space Ratio = \frac{\sqrt{252} \cdot \bar{r_e}}{\sigma_d}$$

$$\bar{r_e} = \frac{1}{n} \sum_{i=1}^n [r_p(i) - r_f(i)]$$

$$Sortino \space Ratio = \sqrt{252} \cdot Daily \space Sortino \space Ratio$$

其中 $\bar{r_e}$ 为回测期内策略日超额收益率均值； n 为回测期内交易日数目； $r_p(i)$, $r_f(i)$ 分别为第 i 个交易日策略所持有投资组合的日收益率以及无风险组合日收益率； σ_d 为策略年化下行波动率。

信息比率(information ratio): 衡量策略相对于市场基准组合的表现。一般用于评估纯多头的主动交易策略（包括阿尔法策略和基准择时策略）。需要注意的是，信息率不适用于多-空结合的对冲策略的表现评估。

$$Daily \space Information \space Ratio = \frac{\sqrt{252} \cdot \bar{r_{pa}}}{\sigma_t}$$

$$\bar{r_{pa}} = \frac{1}{n} \sum_{i=1}^n [r_p(i) - r_b(i)]$$

$$Information \space Ratio = \sqrt{252} \cdot Daily \space Information \space Ratio$$

其中 $\bar{r_{pa}}$ 为回测期主动日收益率均值； n 为回测期内交易日数目； $r_p(i)$, $r_b(i)$ 分别为第 i 个交易日策略所持有投资组合的日收益率以及基准组合日收益率； σ_t 为策略跟踪误差。

在我们提供的三个风险调整后收益指标中，信息率用于评估投资组合相对于市场基准组合的表现，一般适用于纯多头的主动交易策略（包括阿尔法策略和基准择时策略）；夏普率用于评估投资组合相对于无风险组合的表现，一般适用于多-空结合的交易策略（例如市场中性策略或配对交易策略），或没有公认市场基准组合的投资品种的交易策略（例如期货 CTA 策略）。

索提诺比率使用下行波动率作为风险度量，因而有别于信息率和夏普率。下行波动率区分了收益向上波动和向下波动两种情况，并认为收益向下波动才代表风险。因此，索提诺比率的优点，在于其使用的风险度量更为切合我们实际投资中面对的风险；而其缺点则是不如信息率和夏普率常用，认知度较低，且其目标收益（区分收益波动是向上还是向下的标准）的设定是任意的，并不依赖于任何基准组合（不同于信息比率）。因此，在横向对比不同策略或基金业绩时，我们需要使用统一的目标收益来区分向上波动和向下波动。在实际计算中，我们以无风险组合收益作为索提诺比率的目标收益。

复权机制

复权是用来消除由于除权除息造成的价格、指标的走势畸变。

举例来说, 平安银行最近两年的分红派息时间为:

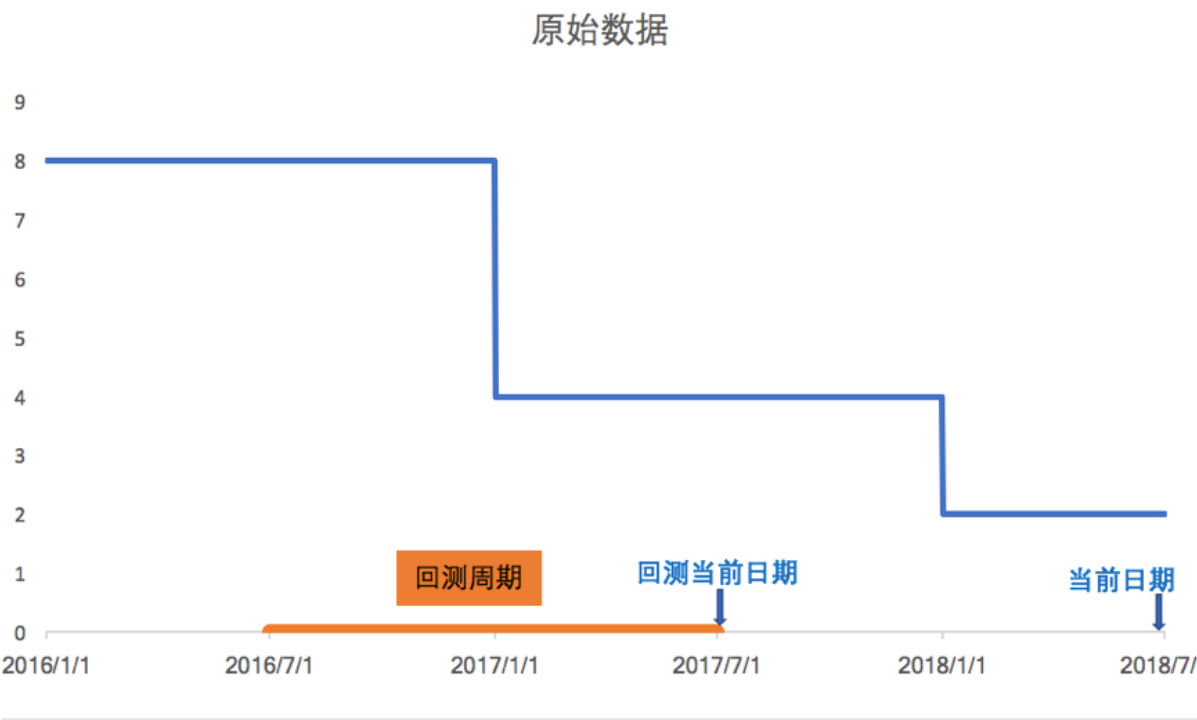
除权除息日	复权因子
2015-04-13	1.210683
2016-06-16	1.217847

平安银行的价格在 2015-04-10 为 19.80, 在除权除息以后的 2015-04-13 价格变为 16.54, 如果什么也不做, 就会导致 K 线上的价格在 2015-04-13 突然降低, 复权的作用就是为了让股价连续, 消除价格和指标的走势畸变。

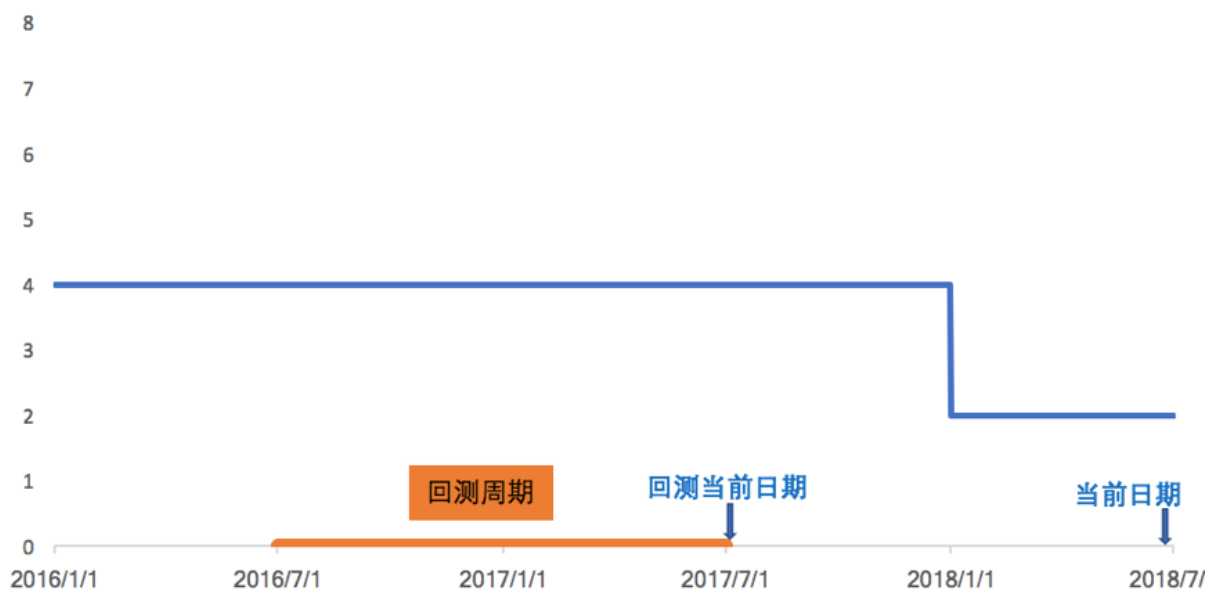
常见的复权机制有前复权和后复权。前复权就是保持现有的价格不变, 将以前的价格缩减, 在 K 线上的表现就是将除权除息日之前的 K 线向下移动; 后复权就是保持先前的价格不变, 将以后的价格提高, 在 K 线上的表现就是将除权除息日之后的 K 线向上移动。

值得注意的是: 无论是前复权还是后复权, 它们都是以当前时间点复权因子进行数据的转变。这就会导致一个问题, 假设我们需要回测 2016 年-2017 年的策略的表现, 若使用前复权, 数据会使用当前 (2018 年) 的复权因子, 这样的数据会混杂未来数据 (即 2018 年的复权因子), 未来数据对于回测来说是致命的。

为了解决这一问题, 回测使用的数据采用了“动态复权”方式, 以策略回测当前日期为基准进行前复权。用同一个例子来说明: 回测 2016 年-2017 年, 数据会使用策略回测当前日期 (即 2017 年) 的复权因子进行前复权, 这样就可以排除未来数据的干扰。更加直观的图像如下图所示:



动态复权



前复权



目前DeepQuant API 提供两种不同的获得历史数据的方式：`history_bars()`；`get_price()`。其中 `history_bars()` 返回的是动态复权的数据；`get_price()` 返回的是前复权的数据，您可以根据自己的需求选择不同的方法。

基准合约

通过引入基准合约，您可以将策略的表现与基准进行对比，以衡量一定时间内策略的超额表现以及相关风险调整收益指标。

您可以在策略编辑页面"更多"选项下进行基准合约的设置，基准合约可以设置为空。在初次创建策略的时候，包含股票的策略默认基准合约都是**沪深 300 指数 000300.SH**；单独期货策略默认没有基准合约。

保证金倍率

在进行期货交易时，您可以通过设置保证金倍率来进行保证金的调整，这一倍率需大于 0。

调整依据基准是交易所规定的最低保证金比例，可以通过 instruments 这一方法查询到。您可以在策略编辑页面"更多"选项下进行保证金倍率设置。该设置在单一股票类型策略中不会出现。

交易费用

在新版策略框架中，原有通过 context.commission 设置佣金费率的方式已经被废弃，不再生效。您需要在策略参数高级设置界面中进行佣金倍率的设置，1 即代表为默认佣金费率的一倍。该倍率不会影响最低佣金以及印花税的收取标准。

- 中国 A 股交易费用

A 股市场交易费用主要由佣金和印花税两部分组成，佣金目前为双边收费，每笔委托最小收取标准为 5 元。默认佣金费率是成交额的万分之 8，即 0.0008。如果设置倍率为 10，则实际影响策略的佣金费率就变成了 0.008。该倍率并不会影响 5 元的最小佣金水平。印花税只对卖出方单边征收，对买入方不征税，目前按照成交金额的 0.1%收取。**由于是强制征收，印花税已经默认加入到我们的收益计算中。**
- 中国期货交易费用

期货交易费用只包括佣金这一个组成部分。佣金收取方式比较复杂，收取方式有按照名义价值的一定比例收取和按照成交合约张数收取两种方式，同时个别合约区分平今仓费率。

我们也支持自定义期货佣金费用的设置，您可以从设置 config 设置项来指定每一个品种的交易费率、平今费率等。初始默认费率表如下请参考：

品种	交易所	佣金类型	回测佣金费率	回测平今费率
铝 AL	上期所	按成交量	3	0
锡 SN	上期所	按成交量	3	0
橡胶 RU	上期所	按成交额	0.00045	0.00045
线材 WR	上期所	按成交额	0.00004	0.00004
螺纹钢 RB	上期所	按成交额	0.000045	0
燃油 FU	上期所	按成交额	0.00002	0.00002
金 AU	上期所	按成交量	10	0
铜 CU	上期所	按成交额	0.000025	0

品种	交易所	佣金类型	回测佣金费率	回测平今费率
银 AG	上期所	按成交额	0.00005	0
铅 PB	上期所	按成交额	0.00004	0
镍 NI	上期所	按成交量	6	0
热轧卷板 HC	上期所	按成交额	0.00004	0
锌 ZN	上期所	按成交量	3	0
沥青 BU	上期所	按成交额	0.00005	0
纸浆 SP	上期所	按成交额	0.00005	0
不锈钢 SS	上期所	按成交额	0.00001	0
原油 SC	上海国际能源交易中心	按成交量	20	0
20 号胶 NR	上海国际能源交易中心	按成交量	10	10
棕榈油 P	大商所	按成交量	2.5	0
细木工板 BB	大商所	按成交额	0.0001	0.00005
鸡蛋 JD	大商所	按成交额	0.00015	0.00015
焦炭 J	大商所	按成交额	0.00006	0.00003
聚乙烯 L	大商所	按成交量	2	0
聚丙烯 PP	大商所	按成交额	0.00005	0.00025
铁矿石 I	大商所	按成交额	0.00006	0.00003
豆粕 M	大商所	按成交量	1.5	0

品种	交易所	佣金类型	回测佣金费率	回测平今费率
玉米 C	大商所	按成交量	1.2	0
焦煤 JM	大商所	按成交额	0.00006	0.00003
中密度纤维板 FB	大商所	按成交额	0.0001	0.00005
玉米淀粉 CS	大商所	按成交量	1.5	0
豆一 A	大商所	按成交量	2	0
豆二 B	大商所	按成交量	2	2
聚氯乙烯 V	大商所	按成交量	2	0
豆油 Y	大商所	按成交量	2.5	0
乙二醇 EG	大商所	按成交量	4	0
粳米 RR	大商所	按成交量	4	0
苯乙烯 EB	大商所	按成交量	6	0
锰硅 SM	郑商所	按成交量	3	0
白糖 SR	郑商所	按成交量	3	0
菜籽粕 RM	郑商所	按成交量	1.5	0
油菜籽 RS	郑商所	按成交量	2	0
早籼稻 RI(ER)	郑商所	按成交量	2.5	2.5
TA	郑商所	按成交量	3	3
动力煤 ZC(TC)	郑商所	按成交量	4	0

品种	交易所	佣金类型	回测佣金费率	回测平今费率
晚籼稻 LR	郑商所	按成交量	3	0
甲醇 MA(ME)	郑商所	按成交量	1.4	0
粳稻谷 JR	郑商所	按成交量	3	3
硅铁 SF	郑商所	按成交量	3	0
菜籽油 OI(RO)	郑商所	按成交量	2.5	0
棉花 CF	郑商所	按成交量	4.3	0
玻璃 FG	郑商所	按成交量	3	0
普麦 PM	郑商所	按成交量	5	5
强麦 WH(WS)	郑商所	按成交量	2.5	0
棉纱 CY	郑商所	按成交量	4	0
苹果 AP	郑商所	按成交量	1.5	0
红枣 CJ	郑商所	按成交量	3	0
尿素 UR	郑商所	按成交量	5	5
纯碱 SA	郑商所	按成交量	3.5	3.5
沪深 300IF	中金所	按成交额	0.000023	0.0023
中证 500IC	中金所	按成交额	0.000023	0.0023
上证 50IH	中金所	按成交额	0.000023	0.0023
5 年期国债 TF	中金所	按成交量	3	0

品种	交易所	佣金类型	回测佣金费率	回测平今费率
10 年期国债 T	中金所	按成交量	3	0
2 年期国债 TS	中金所	按成交量	8	0
生猪 LH	大商所	按成交额	0.0002	0.0004
花生 PK	郑商所	按成交量	4	4

备注

系统支持在策略代码中按照期货合约品种自定义佣金费率和平今费率（此设置在模拟交易中同样生效）：

```
__config__ = {
    "base": {
        "future_info": {
            # 期货品种
            "SC": {
                # 平仓费率
                "close_commission_ratio": 0.0001,
                # 开仓费率
                "open_commission_ratio": 0.0002,
                # 平今费率
                "close_commission_today_ratio": 0,
                # BY_MONEY 为按照名义价值收取，BY_VOLUME 为根据成交合约张数收取
                "commission_type": "BY_MONEY",
            }
        }
    }
}
```

- 中国期权交易费用

期权交易费用收取方式和期货类似，初始默认费率表如下请参考：

品种	交易所	佣金类型	回测佣金费率	回测平今费率
白砂糖 SR	郑商所	按成交量	3	0
棉花 CF	郑商所	按成交量	3	0
菜籽粕 RM	郑商所	按成交量	0.8	0
甲醇 MA	郑商所	按成交量	0.8	0
精对苯二甲酸 TA	郑商所	按成交量	1	0
铜 CU	上期所	按成交量	10	0
橡胶 RU	上期所	按成交量	6	0

品种	交易所	佣金类型	回测佣金费率	回测平今费率
黄金 AU	上期所	按成交量	4	0
豆粕 M	大商所	按成交量	2	1
黄玉米 C	大商所	按成交量	1.2	1.2
铁矿石 I	大商所	按成交量	4	4
液化石油气 PG	大商所	按成交量	1	1
上证 50ETF	上交所	按成交量	12	12
沪深 300ETF	上交所	按成交量	12	12
沪深 300ETF	深交所	按成交量	12	12
300 股指 IO	中金所	按成交量	18	18

- 上金所现货交易费用

交易费用参考上金所交易条款，初始默认费率，如下请参考：

合约代码	佣金类型	佣金费率
PT9995.SGEX	按成交额	0.04%
IAU995.SGEX	按成交额	0.035%
AU995.SGEX	按成交额	0.035%
AU9995.SGEX	按成交额	0.035%
IAU100G.SGEX	按成交额	0.035%
IAU9999.SGEX	按成交额	0.035%
AU100G.SGEX	按成交额	0.035%
AU9999.SGEX	按成交额	0.035%
PGC30G.SGEX	按成交额	0.035%
AU50G.SGEX	按成交额	0.035%
AG9999.SGEX	按成交额	0.02%
AG999.SGEX	按成交额	0.02%
AGTD.SGEX	按成交额	0.02%
AUTN1.SGEX	按成交额	0.02%
AUTD.SGEX	按成交额	0.02%
AUTN2.SGEX	按成交额	0.02%
MAUTD.SGEX	按成交额	0.02%

撮合机制

在最新的版本中，我们加入了允许用户自定义撮合机制的功能。您可以在策略编辑页面"更多"选项下选择不同的撮合机制。目前提供的撮合方式有以下三种：

- 1.当前收盘价。即当前 bar 发单，以当前 bar 收盘价作为参考价撮合。
- 2.下一开盘价。即当前 bar 发单，以下一 bar 开盘价作为参考价撮合
- 3.己方最优。tick 回测专用[仅限期货]，市价单将以己方最优报盘价格成交(无己方报盘时以最新价成交)。
- 4.对手方最优。tick 回测专用[仅限期货]，以发单时下一个 tick 对手方最优报盘价格作为参考价撮合成交(无对手盘时以最新价作为参考价)。
- 5.最新价。tick 回测专用，市价单将以最新价成交

一个完整的撮合机制需要包括两个方面：参考价与投资者在下单时的委托方式。投资者可以选择的委托方式包括有限价单和市价单。

限价单 (LimitOrder)如果买单价格 \geq 参考价，或卖单价格 \leq 参考价，以参考价加入滑点影响成交。市价单 (MarketOrder) 直接以参考价加入滑点影响成交。

对于分钟回测与日回测来说，默认**成交数量都不超过当前 bar 成交量的 25%**。一旦超过，市价单会在部分成交之后被自动撤单；限价单会一直在订单队列中等待下一个 bar 数据撮合成交，直到当日收盘。当日收盘后，所有未成交限价单都将被系统自动撤单。您可以在策略编辑页面"更多"选项下进行是否开启限制成交量，以及开启之后成交百分比的设置。对于 tick 回测来说则**没有成交量的限制**。

- 当前收盘价：在一个 `handle_bar` 内下单，下单时立刻撮合成交，成交价取决于撮合机制以及滑点设置。
- 下一开盘价：在一个 `handle_bar` 内下单，在该 `handle_bar` 结束时统一撮合成交，成交价取决于撮合机制以及滑点设置（分钟回测支持，日回测不支持）。
- 对手方最优/己方最优/最新价：在一个 `handle_tick` 内下单，在该 `handle_tick` 结束时统一撮合成交（成交价取决于撮合机制以及滑点设置）。

所以在“下一开盘价”的撮合方式下，`handle_bar` 内发单之后立刻通过 `cancel_order` 对该订单进行撤单操作，是一定会撤单成功的。但在“当前收盘价”的撮合方式下则很可能撤单失败，因为“当前收盘价”中下单之后立刻撮合成交。

需要注意

- 如果需要当前 bar 开盘成交，可以使用[open_auction](#) API 在盘前集合竞价时发单，以当日开盘价撮合。
- 在当前的分钟回测撮合模式下，用户在回测中无法通过在[scheduler](#)调用的函数中一次性实现 卖出 -> 资金释放 -> 买入 这种先卖后买的逻辑的。因为在分钟回测中，卖出并不能立刻成交。

举例来说，策略 A 设置每周一开盘进行调仓操作，先卖后买。那么，以下这种方式在分钟回测中**无法实现卖出资金立刻释放的**（在开启验资的风控情况下，可能导致后面的买入操作因资金不足而拒单）：

```
#scheduler调用的函数需要包括context, bar_dict两个参数
def rebalance(context, bar_dict):
    order_shares('000001.SZ', -100)
    order_shares('601998.SH', 100)

def init(context):
    scheduler.run_weekly(rebalance, weekday=1)
```

有如下几种情况无法完成下单：

- portfolio 内可用资金不足
- 下单数量不足一手（股票为 100 股）
- 下单价格超过当日涨跌停板限制
- 当前可卖（可平）仓位不足
- 股票当日停牌
- 合约已经退市（到期）或尚未上市

另外需要注意的是，如果当时市场处于涨停或跌停这种单边市情况，买单（对应涨停），卖单（对应跌停）是无法成交的。尽管 bar 数据中可能成交量不为 0。判断单边市的标准我们采用的是：当前 bar 数据的收盘价等于涨停价，则当前市场处于涨停状态。跌停也是类似处理。

滑点

为了更好地模拟实际交易中订单对市场的冲击，我们引入滑点的设置。您可以在策略编辑页面"更多"选项下进行滑点设置，该设置将在一定程度上使最后的成交价"恶化"，也就是买得更贵，卖得更便宜。目前支持的滑点类型有百分比和跳/手，百分比：按照成交价的一定比例进行恶化，取值范围 [0, 1)，例如，当股票市价为 10 元，参数设为 0.1，则回测时买入价为 $10 + 10 * 0.1 = 11$ 元；跳/手：按照最小价格变动单位对成交价进行恶化，例如，上例中设置值为 5，最终成交价将变成 10.05

注意，滑点默认为 0，原有的默认 0.246% 的滑点值以及通过 context.slippage 设置的方式被废弃。

分红派息

- 拆分（送股、增股）

在回测中您无须担心拆分对股票价格带来的影响因为我们已经在数据的预处理中准确地帮您做了这个工作。

- 股息

在股息事件中有四个关键的日期：

- 1.**方案实施公告日**：公司公布股息分配方案的日期。
- 2.**股权登记日**：在股权登记日这一天收盘时仍持有或买进该公司的股票的投资者可以享有此次股息分红。
- 3.**除权除息日**：股权登记日的下一个交易日即是除权除息日，该日证券交易所会计算出股票的除权除息价，以作为投资者在除权除息日开盘的参考。
- 4.**股息到帐日**：现金股息划拨到投资者资金账户的日期。

当您的投资策略在**股权登记日**时仍持有分股息的股票，那么您的投资策略将有资格参与此次股息分红。在**除权除息日**结束的时候您投资组合中的 DividendRecivable 会增加对应持有股票的股息分红数目。然后在**股息到帐日**那天 DividendRecivable 将会被搬入投资组合中的 AvailableCash - 您最终拿到了应收股息分红的金额，并且可以用这笔钱进行再投资了。

分红再投资

红利再投资俗称利滚利，是指基金进行现金分红时，基金持有人将分红所得的现金以当天基金价格直接用于购买该基金，增加原先持有基金的份额。

开启该功能以后，基金分红将按照当日净值折算成基金份额。

仓位管理

在交易中国 A 股市场证券时引入了 T+1 机制，当日买入的股票需要在下一日才能够卖出。另外，如果持有仓位的股票已经退市，那么系统会自动将仓位清零。此时，投资组合价值将会出现"跳水"；如果持有期货直到到期，那么会按照到期日的结算价进行现金交割。

另外，所有仓位均会在当天收盘之后进行统一清理，被平掉的仓位不会出现在下一交易日初始的持仓中。但在日内，存在持仓为 0 的仓位记录（它还记录了该仓位的建仓价格、累计盈亏等信息，具体请参考[仓位信息](#)）。

标的品种

支持股票、期货、期权、可转债、场内基金合约的日/分钟/tick 级别的回测，上金所现货日级别回测。不同品种的标的在发单接口、费用计算、账户和仓位计算等方面有所差异。

股票和场内基金

支持 A 股和 ETF、LOF 等场内基金回测

- 发单接口：股票和场内基金支持 [order shares](#)、[order lots](#)、[order value](#)、[order percent](#)、[order target value](#)、[order target percent](#) 六个下单接口。
- 账户设置：股票和场内基金的持仓归属于股票（STOCK）账户
- T+1：股票交易默认开启 T+1 限制，详细介绍见[仓位管理](#)

期货

支持期货回测

- 发单接口：不同于股票，期货可使用[buy open](#)、[sell close](#)、[sell open](#) 和 [buy close](#)四个接口发单。
- 账户设置：期货持仓归属于期货（FUTURE）账户
- 保证金交易：期货采用保证金交易，持有期货仓位会占用保证金，这部分资金会被冻结，不能再用于发单，保证金会在平仓时解冻。
 - 使用的保证金率可以通过 [Instrument](#) 接口查看，该接口接收合约代码参数，返回 [Instrument](#) 对象，例如使用如下代码查询 RB2010 的保证金率：

```
instruments("RB2010").margin_rate
```
 - 可以通过设置保证金倍率来调整的保证金率，实际使用的保证金率为默认的保证金率乘以保证金倍率
- 逐日盯市：期货采用“逐日盯市”制度，每日盘后会进行结算，将浮盈浮亏计入现金。

期权

支持商品、股指、ETF 期权回测（需开通量化企业版）。

- 发单接口：交易期权使用与期货相同的发单接口[buy_open](#)、[sell_close](#)、[sell_open](#) 和 [buy_close](#)。
- 账户设置：根据实际市场中所在交易所不同，期权持仓分属股票（STOCK）和期货（FUTURE）账户，其中 ETF 期权属于股票账户，商品期权和股指期货期权属于期货账户。
- 行权
 - 行权采用现金交割，即将行权产生的盈利或亏损直接计入现金中。
 - 主动行权：期权可通过 [exercise](#) 接口主动行权，该函数接收合约代码和行权数量两个参数，例如：

```
exercise("M1905C2350", 2)
```

- 被动行权：期权持有至到期日将会触发自动行权。对于权利方（多头）持仓，若回测引擎判定行权可以盈利，则触发自动行权，否则仓位作废；而义务方（空头）持仓会在判定对手方可以盈利时触发行权
- 权利金和保证金
 - 权利方（多头）：开仓需要缴纳权利金，该过程与股票的开仓类似
 - 义务方（空头）：开仓会收取权利金并付出保证金，保证金会被冻结（类似期货开仓）；同时义务方也采取逐日盯市制度，每日盘后结算，浮盈浮亏将被计入现金。

可转债

支持可转换债券、场内公开交易的可交换债券、分离交易可转债（债券等）的回测（需开通量化企业版）。

- 发单接口：可转债使用 [order_shares](#)、[order_value](#)、[order_percent](#)、[order_target_value](#)、[order_target_percent](#) 五个接口下单，用法与股票相同
- 账户设置：可转债持仓归属于股票（STOCK）账户
- 回售和转股：可转债支持主动发起回售或转股，使用 [exercise](#) 接口，相比于期权行权，除了合约代码和数量两个参数，还加入了第三个参数用于区分本次行权是转股还是回售，如：

```
exercise("132003.SH", 100, convert=False) # 回售
exercise("132003.SH", 100, convert=True)  # 转股
```

- 本息偿付：可转债发生付息时，利息将进入对应账户的现金；发生强制赎回时，仓位将被清空，对应账户的现金会按照强赎时实际的现金流变动。

上金所现货

支持上海黄金交易所交易的黄金、白银、铂金等现货合约的回测（需开通量化企业版）。

- 发单接口：与期货交易相同，上金所现合约使用 [buy_open](#)、[sell_close](#)、[sell_open](#) 和 [buy_close](#) 四个接口下单。
 - 账户设置：上金所现合约持仓归属于股票（STOCK）账户。
 - 保证金交易：与期货类似，上金所现合约采用保证金交易，同样可以配置保证金倍率，同样采用“逐日盯市”制度。
-

股票与期货混合策略注意事项

鉴于不同合约交易时间的不同（例如股票没有夜间交易，期货一些品种有夜盘交易），您在编写策略的时候需要注意策略的有效运行时间。比如在 2015 年 12 月之前，中金所股指期货的交易时间段是 09:15~11:30, 13:00~15:15，比 A 股市场多出了 30 分钟。在这个时候进行混合回测的时候就需要通过[订阅](#)的方式让策略引擎'知道'handle_bar 是要在每天 09:16 产生第一个 bar 数据，而不是股票的 09:31。

如果您创建的是单一的期货策略，则必须在策略初始化的时候订阅(subscribe)有效期货合约。由于期货有到期日，所以您需要保证在回测期间，始终都有正在交易的合约被订阅。

混合策略的股票、期货子账户信息可以分别通过[context.stock_account](#)以及[context.future_account](#)获取到。

安装 Anaconda 虚拟环境（强烈建议）

完整版的 Anaconda 对于大多数人来说都是没有必要的，因此 Anaconda 官方提供了精简版的 Miniconda，只安装最核心的工具包。本文档将以 Miniconda 的安装为例。如果您希望安装 Anaconda 本体，请移步至[Anaconda 官网](#)获得安装指引。下面简要列出 Miniconda 的安装步骤（**注意：本章所述的功能只适用于大多数用户。如果您已对 Anaconda 非常熟悉，或者已有惯用的环境，完全可以不按本章节所述方法配置环境**）：

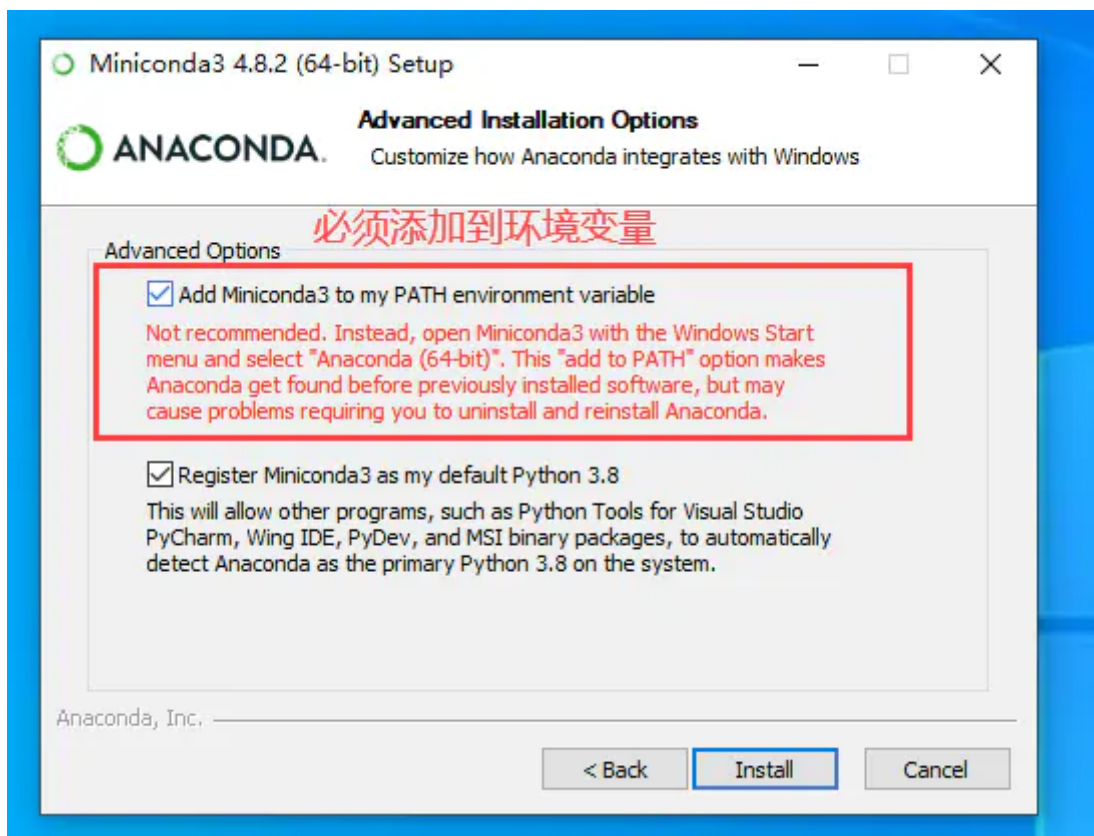
下载 Miniconda

- 如果您身处国内，请在[清华大学的 Miniconda 镜像仓库](#)中寻找适用于您系统的安装包，或直接
 - Windows 系统：[点击下载](#)
 - MacOS 系统
 - X86 版：[点击下载](#)
 - ARM 版：[点击下载](#)
 - Linux 系统：[点击下载](#)
- 如果您身处海外，可以直接去[官方网站](#)下载对应您系统的安装包

安装 Miniconda

在各个系统中安装 Miniconda 的体验与安装其他软件无异，但是在安装过程中有一个需要留意的配置点：

添加命令到环境变量



如果在安装过程中错过了环境变量的选项，可以按如下步骤手工添加

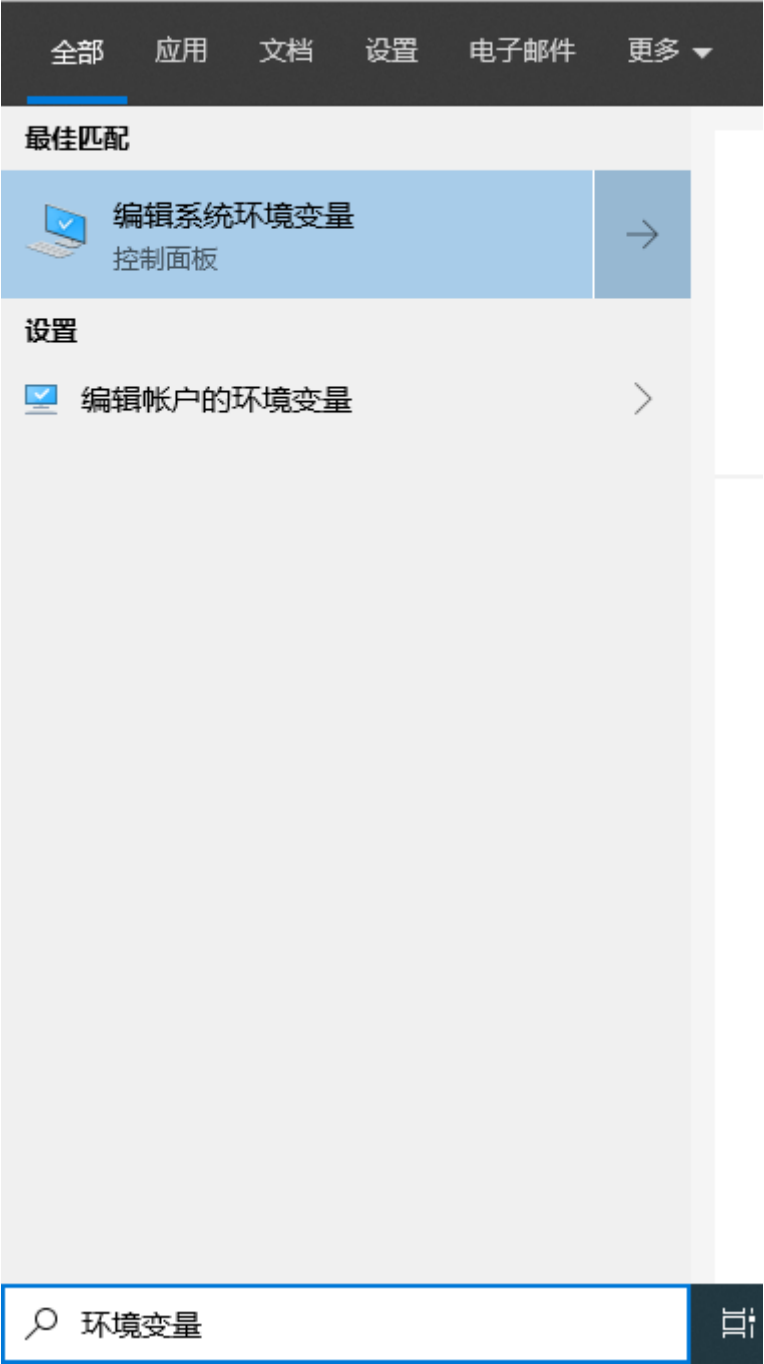
假设您的 Miniconda 安装位置为 `D:\ProgramData\Miniconda3`（默认安装位置是 `C:\Users\<用户目录>\miniconda3`，或 `C:\ProgramData\Miniconda3`，可在安装过程中手工更改）。

将下列值逐个添加到 `Path` 环境变量中（注意在实际操作的时候依据您的实际安装情况来更改目录）：

```
D:\ProgramData\Miniconda3\Scripts
D:\ProgramData\Miniconda3\Library\bin
D:\ProgramData\Miniconda3\Library\usr\bin
D:\ProgramData\Miniconda3\Library\mingw-w64\bin
D:\ProgramData\Miniconda3
```

添加方法如下：

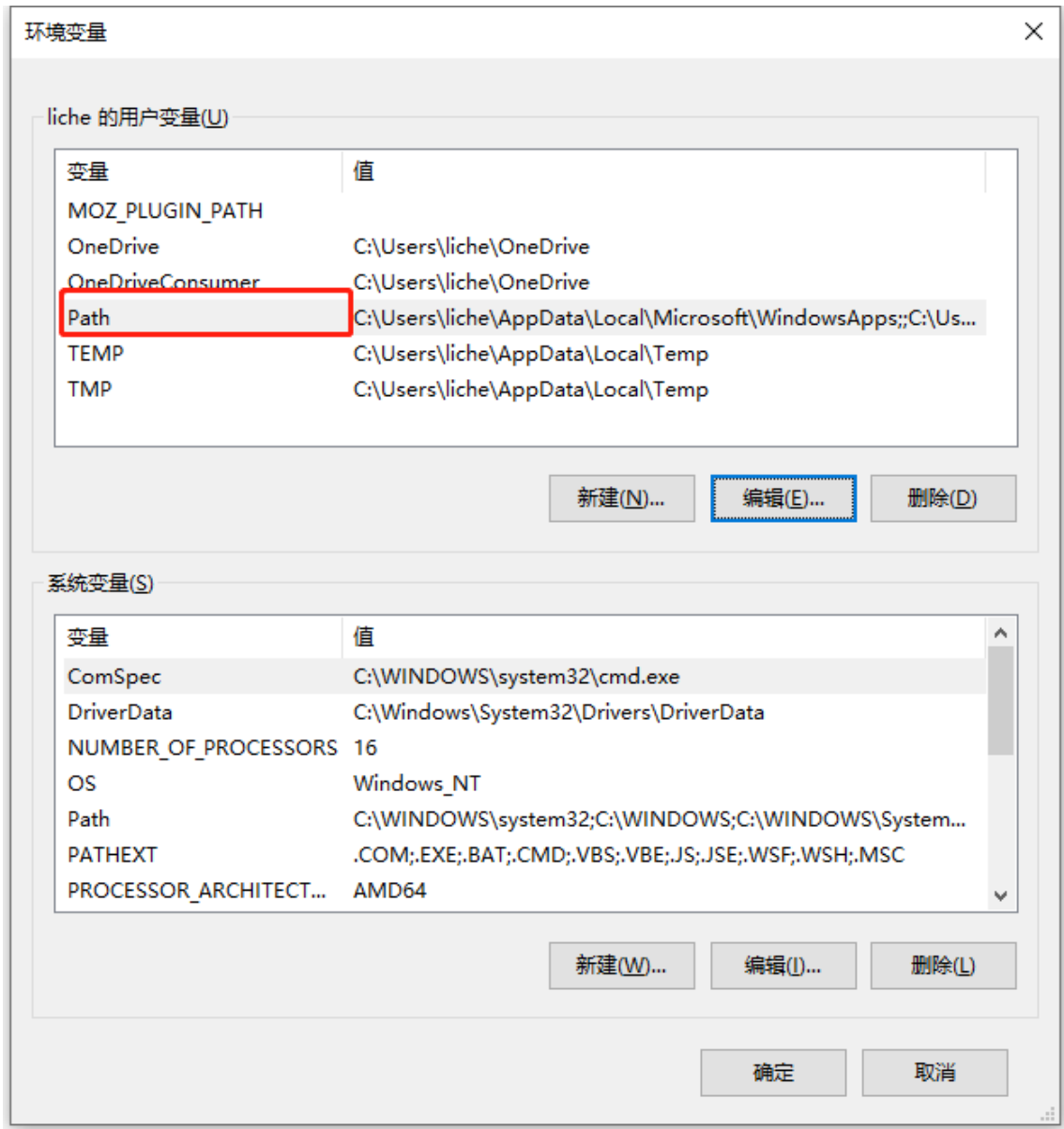
- 在 Windows 搜索框中搜索 环境变量 或 env，系统会给出最佳匹配项



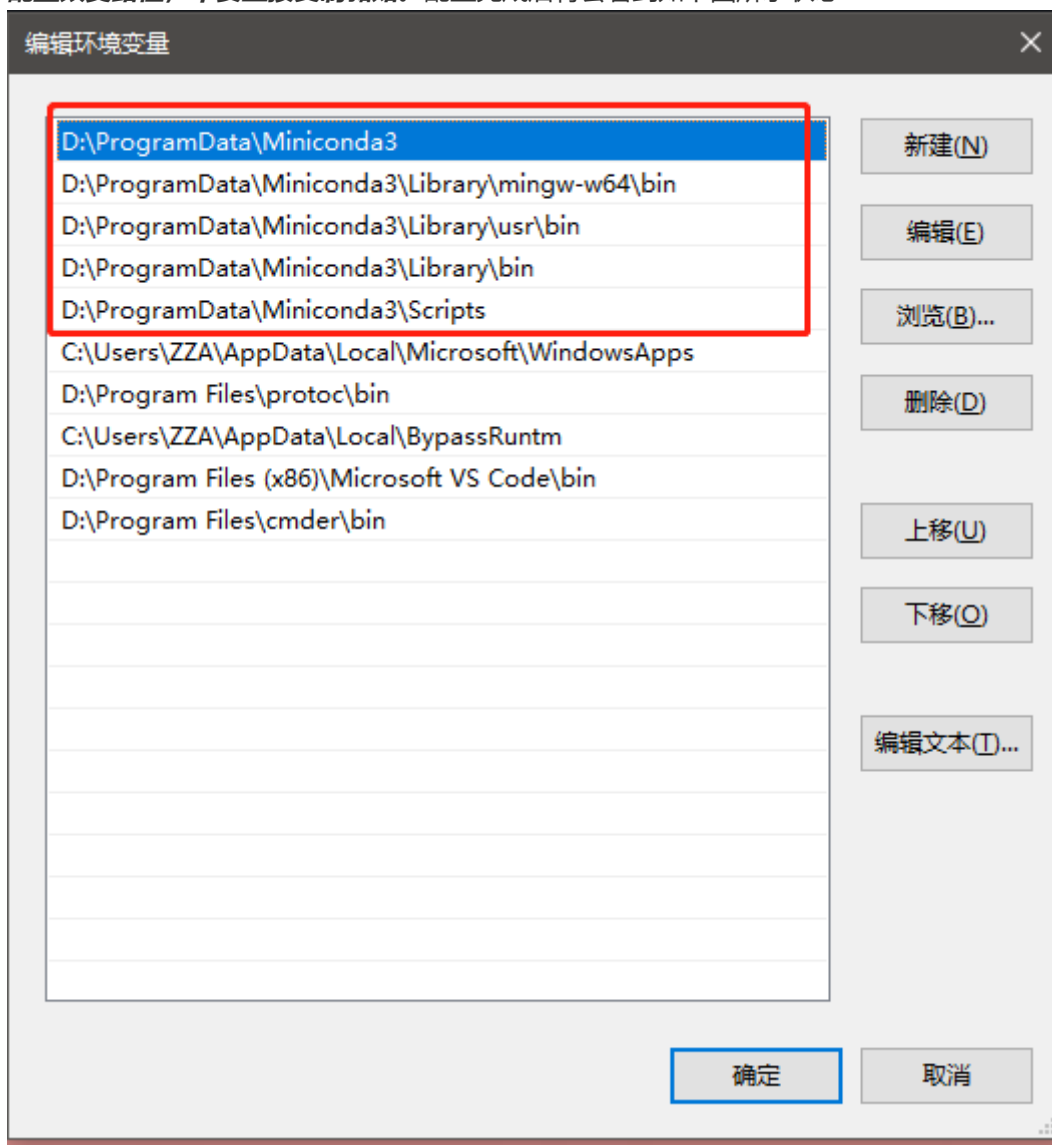
- 打开配置面板后点击 环境变量 按钮



- 在打开的对话框中找到 Path 列，双击或选中后点击 编辑 按钮



- 在打开的对话框中点击 新建 按钮，依次将本节开头的五个路径添加进去，**注意根据自己实际安装的配置改变路径，不要直接复制黏贴**。配置完成后将会看到如下图所示状态：



- 配置完成后打开一个命令行窗口，输入 conda 回车，如果您看到类似下图的结果，说明配置生效。后续的使用中如果碰到类似 No such file or directory 之类关于文件、路径未找到的错误时，可以再次检查一下是否本节描述的五个值都设置正确。

```
命令提示符
Microsoft Windows [Version 10.0.18363.778]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\liche>conda
usage: conda-script.py [-h] [-V] command ...

conda is a tool for managing and deploying applications, environments and packages.

Options:
positional arguments:
  command
  clean                Remove unused packages and caches.
  config              Modify configuration values in .condarc. This is modeled after the git config command. Writes to the
                      user .condarc file (C:\Users\liche\condarc) by default.
  create              Create a new conda environment from a list of specified packages.
  help                Displays a list of available conda commands and their help strings.
  info                Display information about current conda install.
  init                Initialize conda for shell interaction. [Experimental]
  install              Installs a list of packages into a specified conda environment.
  list                List linked packages in a conda environment.
  package             Low-level conda package utility. (EXPERIMENTAL)
  remove              Remove a list of packages from a specified conda environment.
  uninstall            Alias for conda remove.
  run                 Run an executable in a conda environment. [Experimental]
  search              Search for packages and display associated information. The input is a MatchSpec, a query language
                      for conda packages. See examples below.
  update              Updates conda packages to the latest compatible version.
  upgrade             Alias for conda update.

optional arguments:
  -h, --help          Show this help message and exit.
  -V, --version        Show the conda version number and exit.

conda commands available from other packages:
env
```


创建及切换 Python 虚拟环境

安装好 Miniconda 并配置好环境变量之后，您就可以使用 `conda` 命令方便地进行虚拟环境的配置和管理了。

Python 虚拟环境是 Python 提供了一种依赖管理方式，它允许您在同一台电脑上使用不同版本的 Python、不同版本的依赖来开发不同的程序。环境之间互相独立，互不干涉，还可以随意切换。

用下列命令创建一个名为 `deepquant` 的 Python 3.9 环境：

```
conda create -n deepquant python=3.9
```

创建完毕后还需要用 `conda activate deepquant` 来激活刚才创建的环境，这样之后的操作（例如调用 `pip install` 等）就都只会影响这个虚拟环境了。

如果要退出虚拟环境，可以直接在环境激活的状态下运行 `conda deactivate` 命令。

本节仅以 Windows 10 系统为例，更多关于如何使用 `conda` 命令管理虚拟环境的内容可以参考[conda 官方文档——环境管理](#)章节。

升级 Python 环境

如果您已有的环境是低于 Python 3.6 的，请根据您的实际情况升级到 Python 3.6 或以上（推荐升级到最新的 Python 3.9）。我们要求 64-bit 的 Python 环境，如果您的环境是 32-bit 的，请重新安装 64-bit 的环境。

如果您已有环境是 Anaconda 或 Miniconda，可以先用 `conda update conda` 命令更新 `conda` 工具本身，然后再通过 `conda activate <env name>` 激活您的 Python 虚拟环境，然后运行 `conda install python=3.9` 来安装最新的 Python 到该虚拟环境。

中国境内安装加速方案

`pip` 默认下载源服务器在国外，从中国境内访问速度会比较慢。推荐在全球范围内更改默认 `pip` 下载源到国内的清华镜像，将会对安装配置速度有非常明显的提升，感谢清华为中国的 Python 发展做出的贡献。

在命令行中运行如下命令可以轻松配置：

```
conda activate base
pip config set global.index-url https://pypi.tuna.tsinghua.edu.cn/simple
```

conda 基本操作

- 查看版本信息 `conda --version`
- 更新 `conda` `conda update conda`
- 创建一个虚拟环境 `conda create -n deepquant python=3.9`
- 激活新的虚拟环境 `conda activate deepquant`
- 列出环境信息 `conda env list`
- 退出当前环境 `conda deactivate`
- 删除虚拟环境 `conda remove --name deepquant --all`

在 Powershell 中使用 conda 命令

如果您习惯于 Windows 的 Powershell 环境，则在默认的安装状态下，conda 并不能支持，还需要进行以下步骤。

查看您的 conda 版本

通过 `conda --version` 查看您当前的 conda 版本，分两种情况：

- 版本号大于等于 4.6
 - 用管理员权限运行 Powershell
 - 执行命令 `conda init powershell`
- 版本号小于 4.6
 - 用管理员权限运行 Powershell
 - 输入命令 `conda install -n root -c pscondaenvs pscondaenvs` 安装 [PSCondaEnvs](#) 包。
这里注意正确的命令中确实包含了两个 pscondaenvs，并不是文档写错了。
 - 输入命令 `Set-ExecutionPolicy RemoteSigned`，随后输入 Y 并回车，以更改 Powershell 安全策略
 - 在 Powershell 中激活和退出环境的命令分别为 `activate deepquant` 和 `deactivate`，而不是 `conda activate deepquant` 和 `conda deactivate`。

如果对 Conda 环境没有特殊要求的话，建议直接通过 `conda update conda` 命令升级到最新版本

PyCharm 及 vsCode 快速配置

PyCharm

为什么要用 PyCharm?

- PyCharm 作为 IDE（集成开发环境），自带 python 解释器和虚拟环境管理功能，开箱即用。
- PyCharm 默认的内置功能极为丰富（Git、数据库支持、框架支持等），无需手动配置插件便可直接使用。
- PyCharm 内置了在业界无出其右的静态代码审查（code inspect）功能。

PyCharm 下载

[PyCharm 官网](#)提供了专业版和社区版下载。

- 专业版用于科学计算和 Web 开发。同时具有 HTML、JS 和 SQL 等支持。专业版 PyCharm 支持试用 30 天。
- 社区版用于通常的 Python 开发。免费且开源。

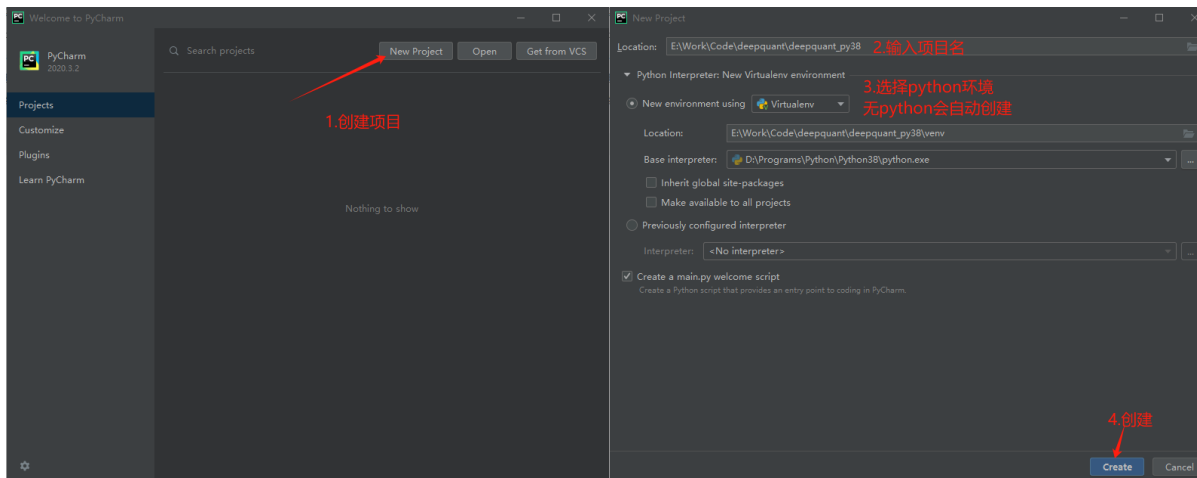
创建 PyCharm 工程（Project）

下载且安装完成 PyCharm 后，便可打开 Pycharm 后建立一个项目（Project）。

建立项目时，可以设置项目使用的 Python 解释器/虚拟环境。后续开发中的代码提示、调试等功能都依赖项目配置的虚拟环境

1. 点击 Create New Project 按钮
2. 展开 Project interpreter
3. 选择虚拟环境（若没有已存在环境，则 PyCharm 会自动创建）

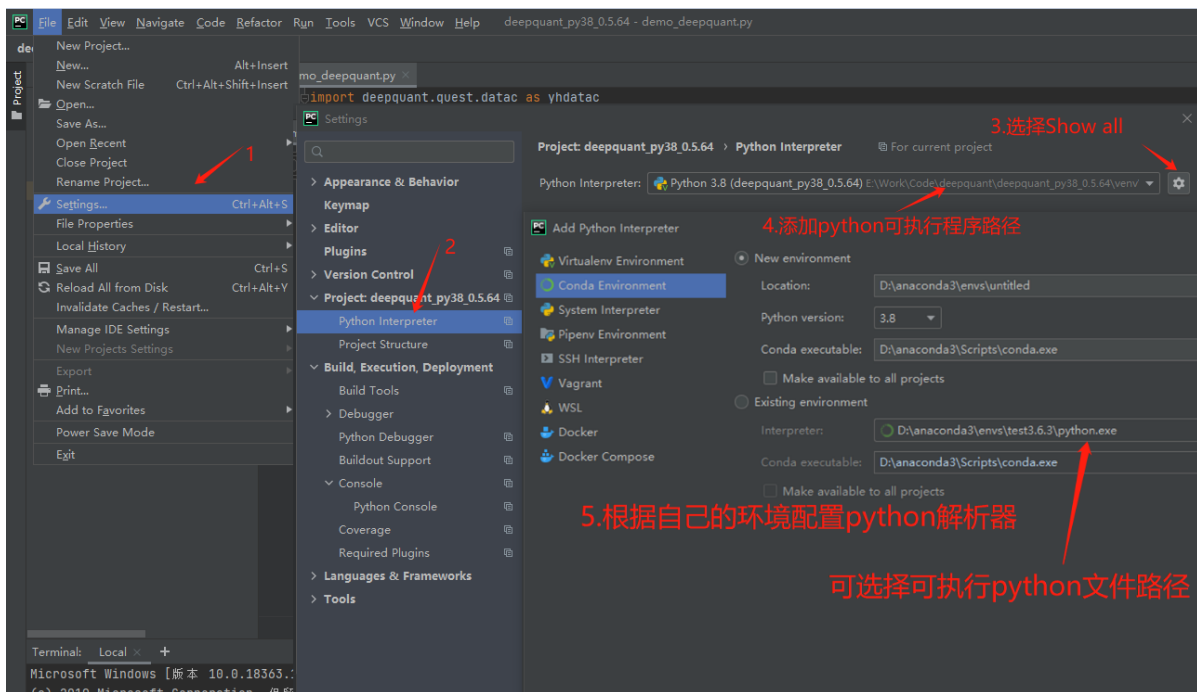
4. 点击 Create 按钮，创建项目



如果没有 python 环境，编辑器右下角会有进度条提示 python 的安装进度。

工程创建完成后，亦可在设置中修改当前工程使用的虚拟环境：

1. 点击左上角菜单栏 File -> Settings (macOS 中为 PyCharm -> Preference)
2. 点击 Project: **** -> Project Interpreter
3. 点击右边小齿轮 -> Show All
4. 点击加号 (+) -> 选择虚拟环境 (Virtualenv Environment) 或者 Conda 环境 (Conda Environment)



在 PyCharm 中安装 deepquant

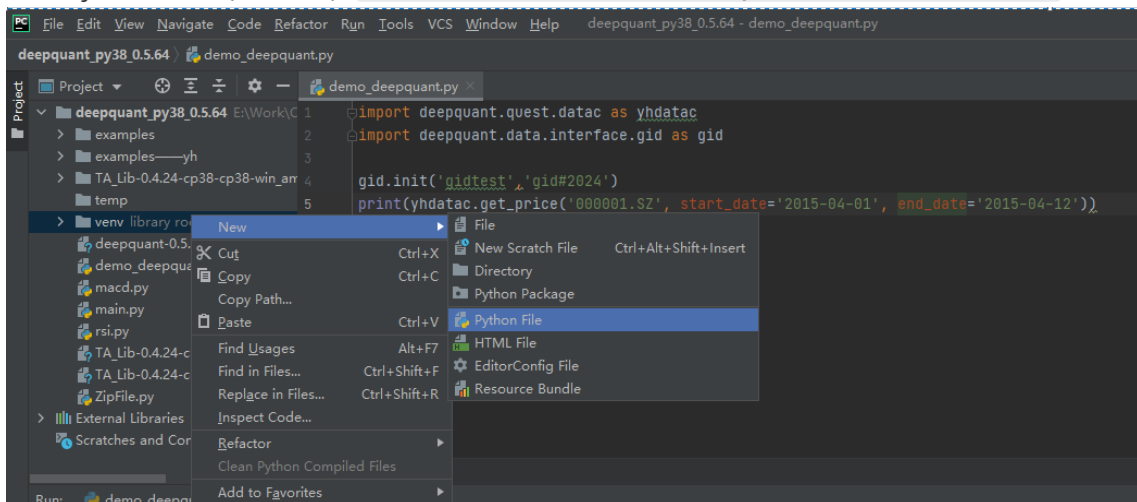
若当前工程配置的虚拟环境中还未安装 deepquant，可以直接在 PyCharm 中调用终端 (terminal) 安装，PyCharm 会自动在改终端激活先前配置好的虚拟环境。

若点击左下角 Terminal 以激活终端，输入以下代码以安装 deepquant

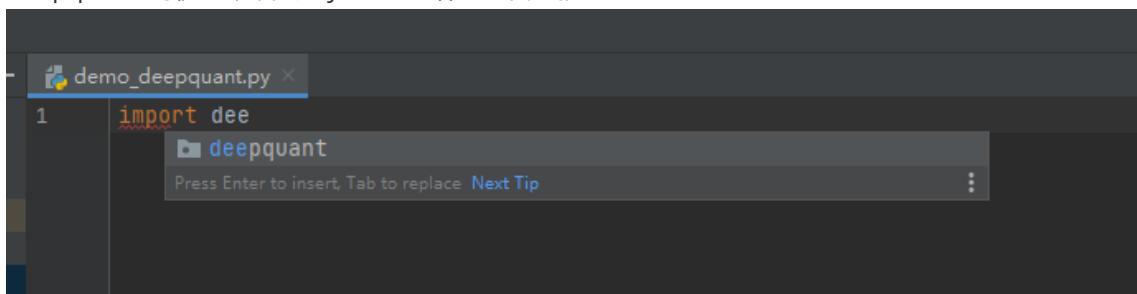
```
pip install -i https://pypi.tuna.tsinghua.edu.cn/simple deepquant
```

使用 PyCharm 编写代码

- 创建 Python 模块 (module) 鼠标右键项目文件夹 -> New -> Python File -> 输入文件名



- 若当前工程正确配置了虚拟环境，且虚拟环境中安装了 deepquant，在 py 文件中输入“import deepquant”时便可以看到 PyCharm 给出的代码提示

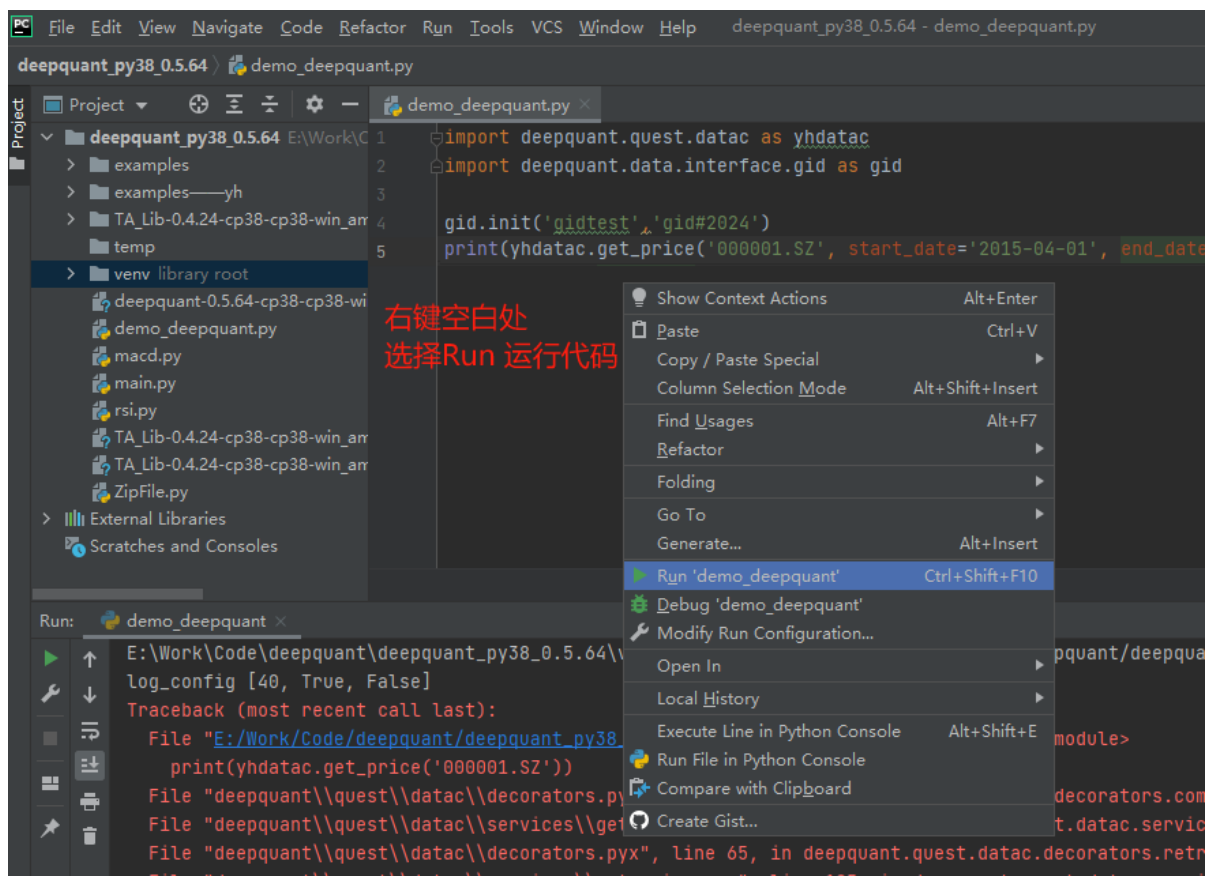


- 在 py 文件中输入下列代码，使用 dataac 调取日线数据：

```
import deepquant.quest.dataac as yhdatac
import deepquant.data.interface.gid as gid

gid.init('gidtest', 'gid#2024')
print(yhdatac.get_price('000001.SZ', start_date='2015-04-01', end_date='2015-04-12'))
```

- 在编辑区域点击右键执行 Run... 便可以运行当前代码。
注意，上述代码的运行要求事先使用 [deepquant license](#) 命令配置好 license。



使用 PyCharm 运行回测

回测在终端中需要通过 `deepquant` 命令而非 `python` 命令运行，故在 PyCharm 中运行回测需要进行一些额外的配置，以简单的 `buy-and-hold` 回测策略为例。

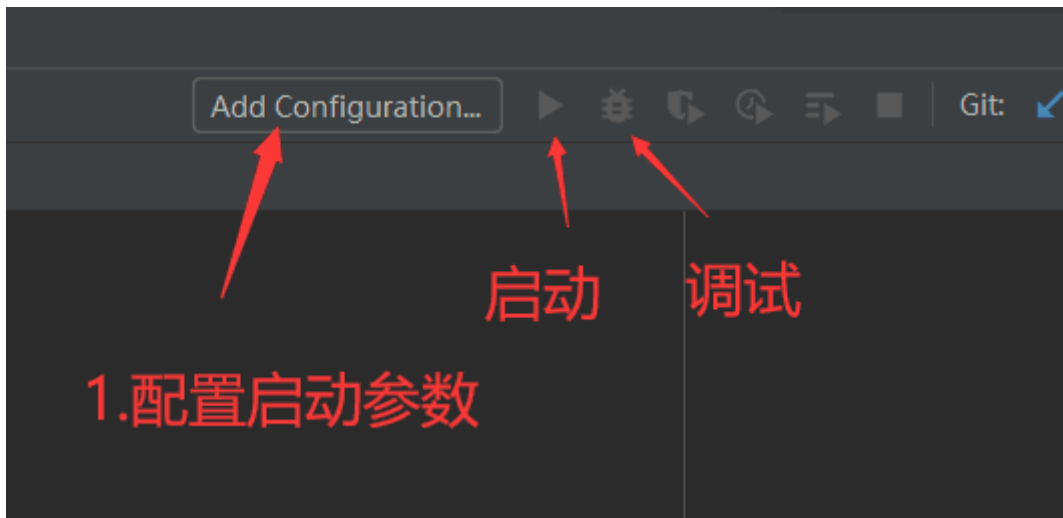
1. 创建名为 `buy_and_hold.py` 的 python 文件并键入以下代码：

```
# buy_and_hold.py

def init(context):
    context.s = "000001.SZ"
    context.fired = False

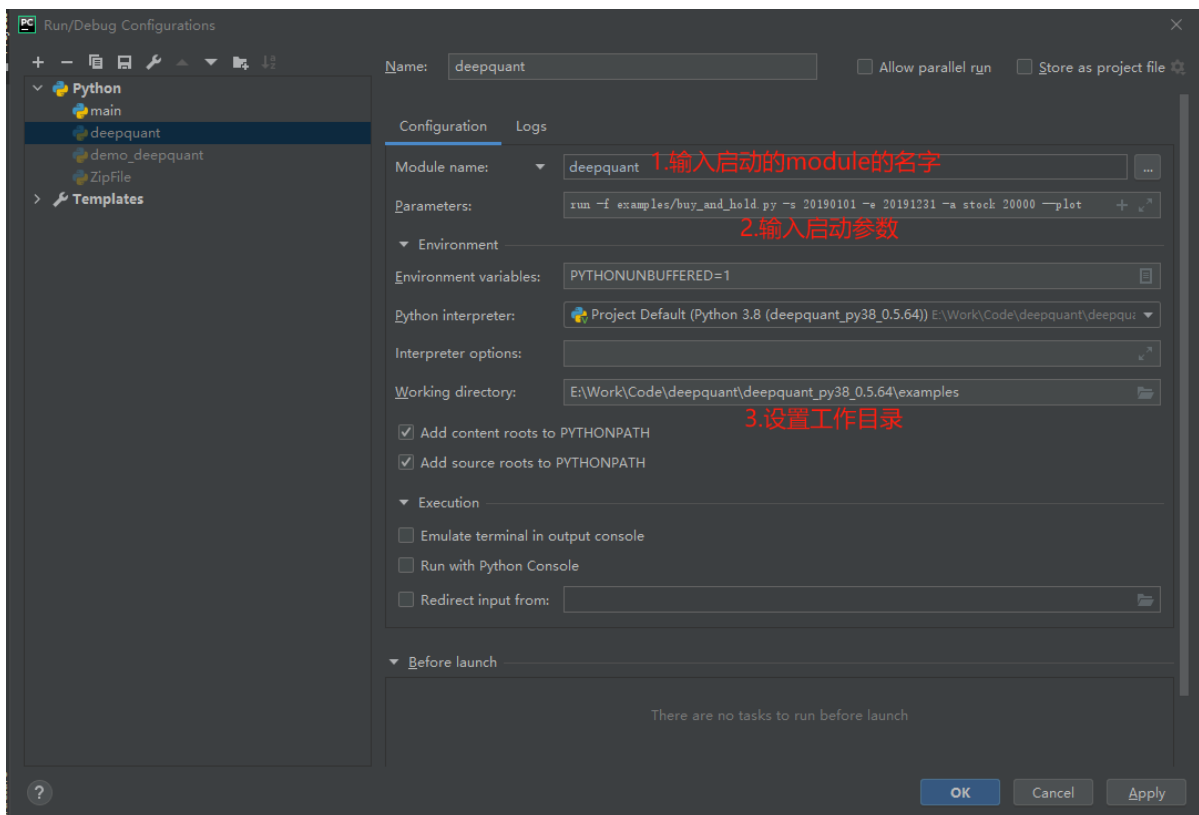
def handle_bar(context, bar_dict):
    if not context.fired:
        order_shares(context.s, 1000)
        context.fired = True
```

2. 点击右上角的 `Add Configuration`

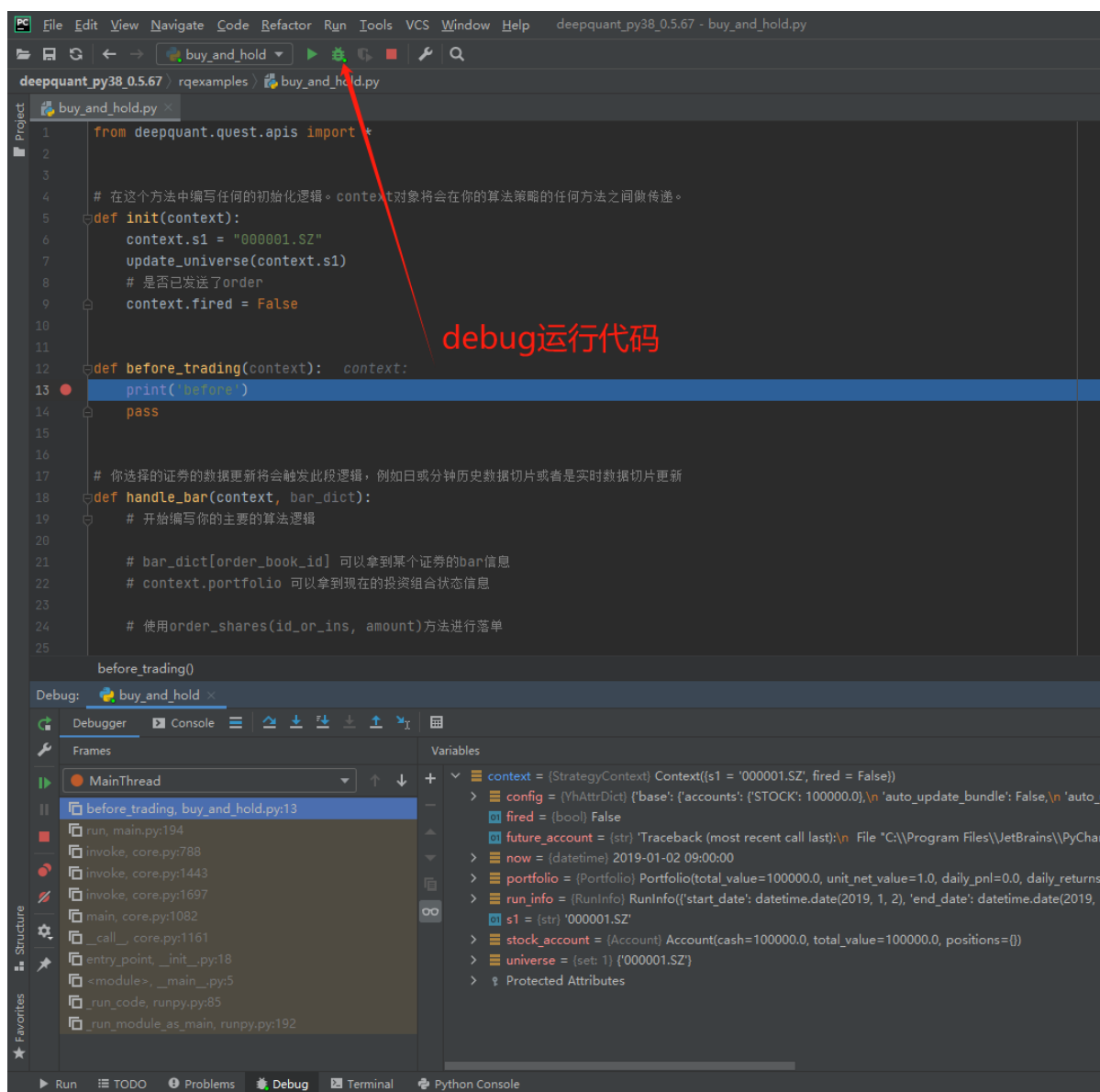


3. 在打开的窗口中将第一项左侧默认的运行方式由 `Script path` 修改为 `Module name`
4. 设置 `Module name` 为 `deepquant`，设置 `Parameters` 为回测运行的子命令 `run` 及其参数，如：

```
run -f buy_and_hold.py -s 20190101 -e 20191231 -a stock 20000 --plot
```



5. 点击 OK 按钮以完成配置
6. 点击右上角的三角形按钮以运行回测，或点击虫子按钮以调试（debug）代码



Visual Studio Code (VS Code)

为什么要用 VS Code?

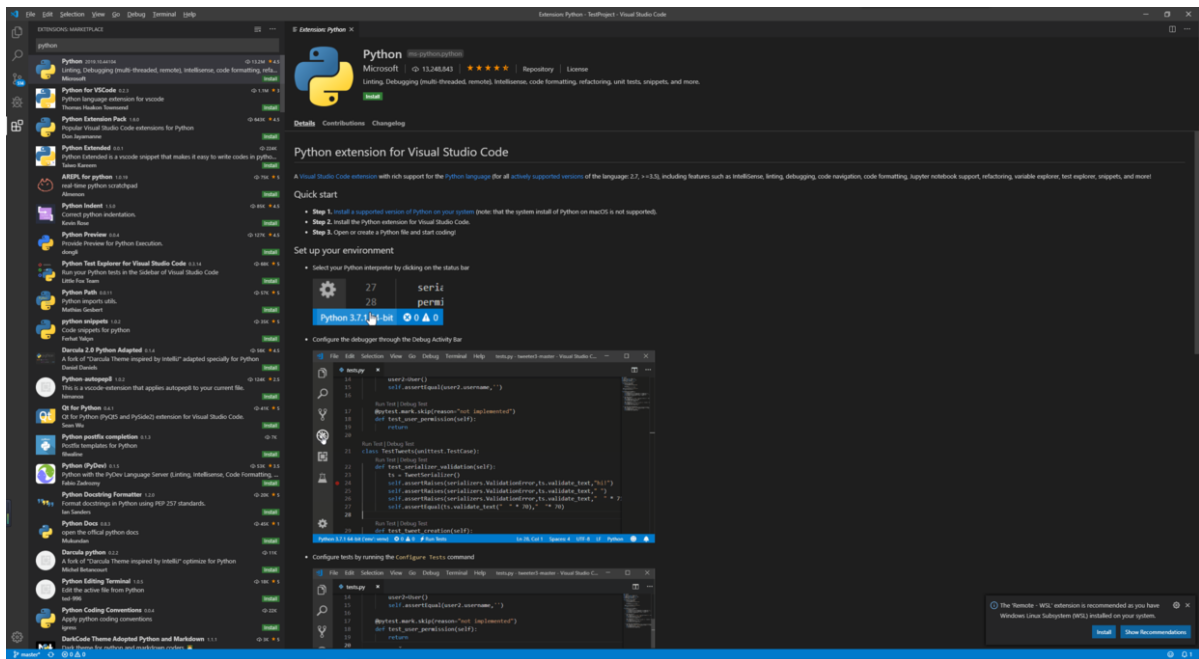
1. 轻量化，下载安装更快；资源占用低，对配置相对不足的计算机更友好。
2. 启动快速，首次创建工程时没有漫长等待创建索引的过程。
3. 生态健全，有着丰富的第三方主题和插件。

在[Visual Studio Code](#)官网可以下载标准版。

安装 Python 插件 (Extensions)

VScode 不是 python 专用的编辑器，故使用其开发 python 需要安装专门的插件支持才能获得代码提示、审查、调试等功能：

- 在左侧栏点击 Extensions 后，搜索 python，选择搜索到的第一项并点击 `install` 按钮安装。

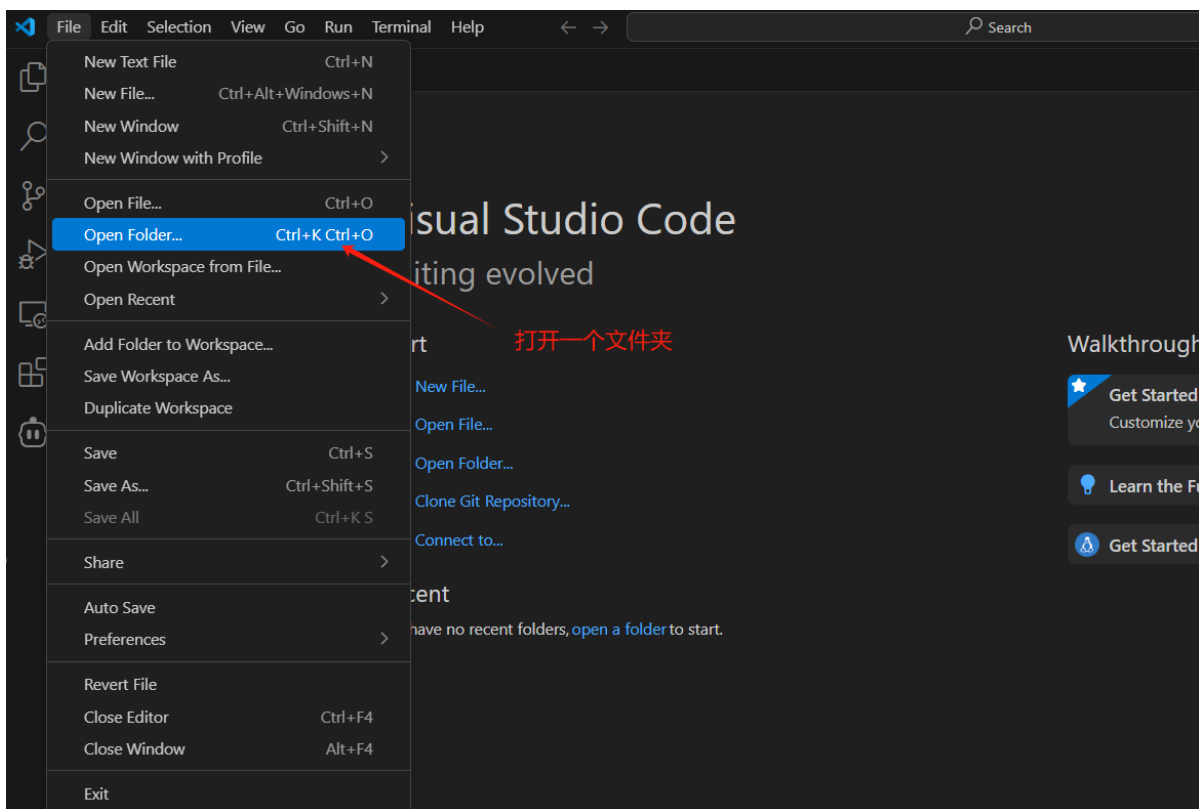


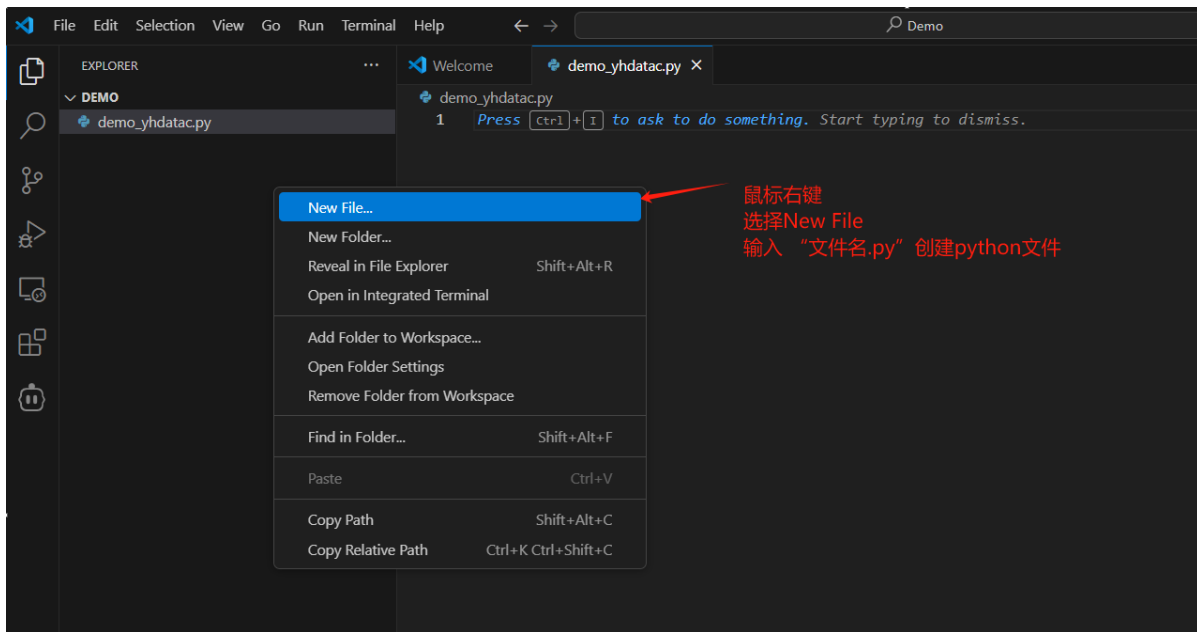
设置虚拟环境/Python 解释器

- 使用 `Ctrl+Shift+P` 快捷键 (macOS 为 `Command+Shift+P`) 打开 command palette 窗口
- 输入关键字 `python select` 并找到 `Python: select Interpreter` 一项, 点击该项并在随后弹出的 Python 解释器列表中选择目标虚拟环境中的解释器 (若目标虚拟环境未列出, 则需要手工输入解释器的路径)

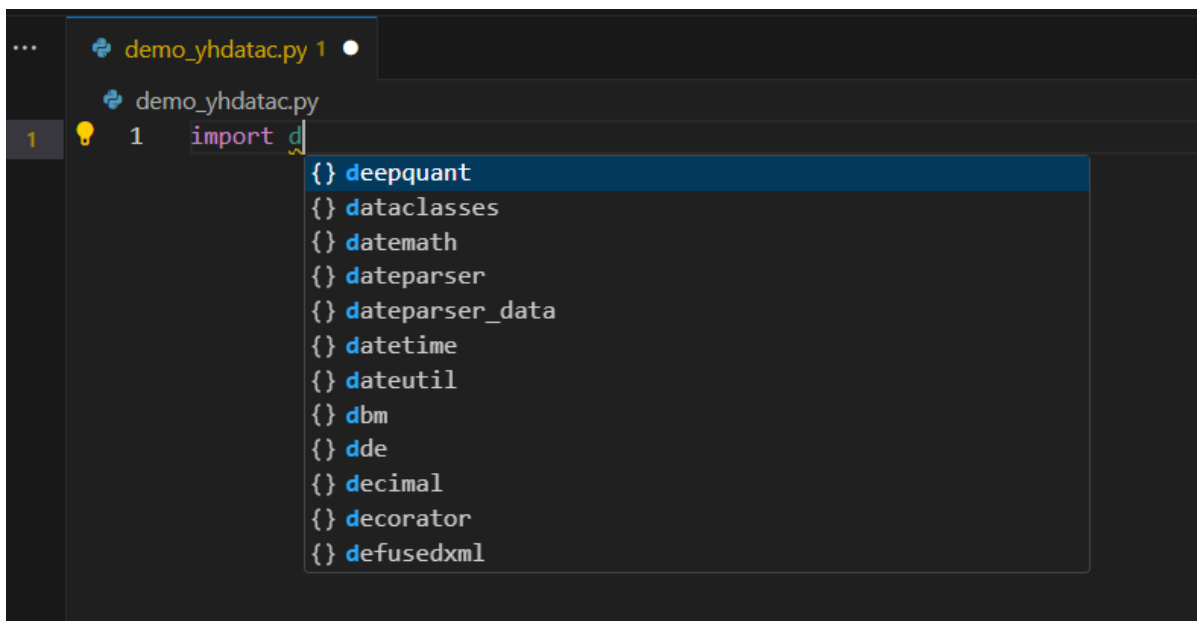
使用 VS Code 编写代码

- 在 VS Code 中点击打开一个系统文件夹, 使用 vs code 会在此文件夹中生成配置文件。
- 在打开的文件夹中创建新的 python 文件, 文件名 `demo_datac.py`





)

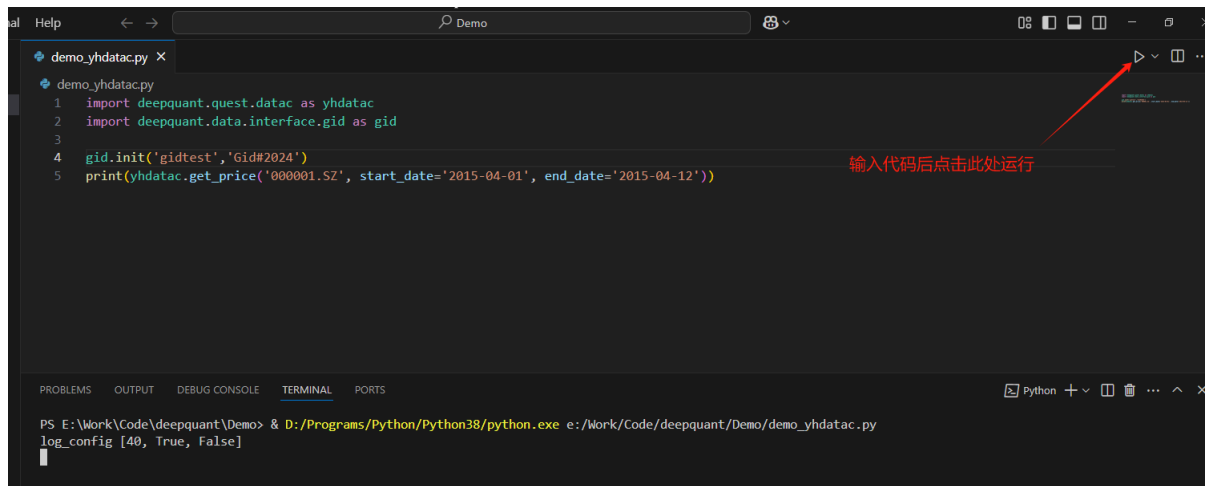


- 使用 datac 查看平安银行日线数据

文件中输入以下代码：

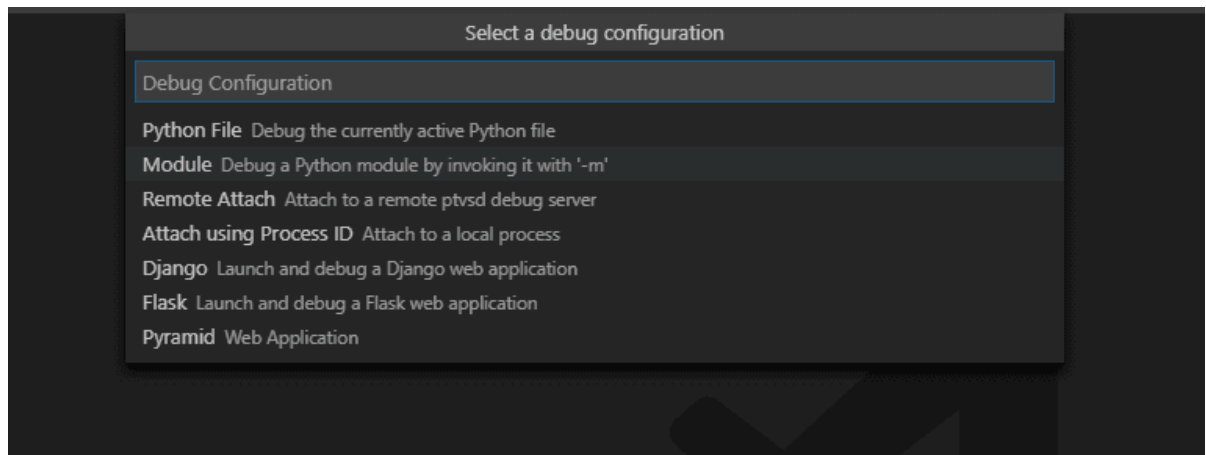
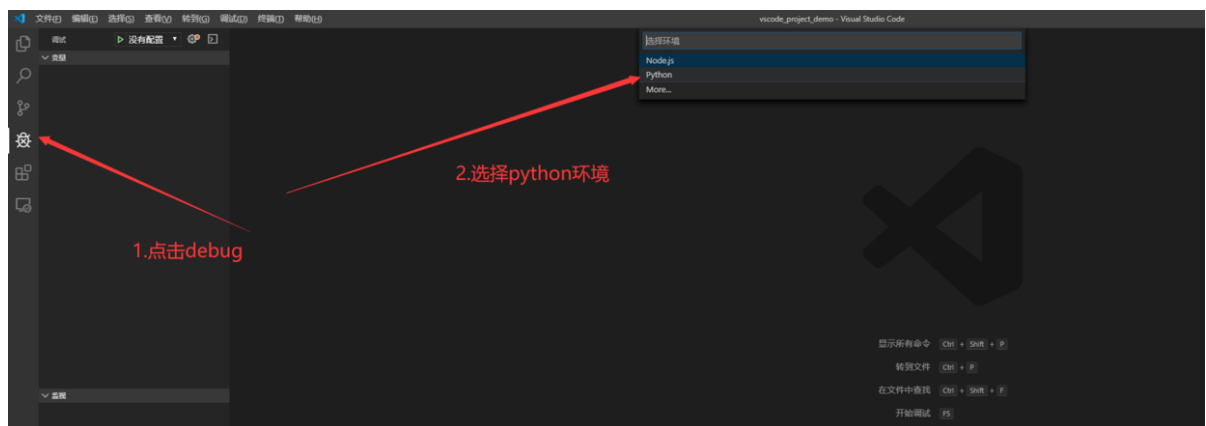
```
import deepquant.quest.datac as yhdatac
import deepquant.data.interface.gid as gid

gid.init('gidtest','gid#2024')
print(yhdatac.get_price('000001.SZ', start_date='2015-04-01', end_date='2015-04-12'))
```



用 debug 方式运行回测

首先需配置 python 解释器。



启动 deepquant debug 模式 需要在 vs code 的配置文件中配置 debug 参数。

debug 配置文件在 .vscode 文件夹下 launch.json 文件中。

需要加入如下代码:

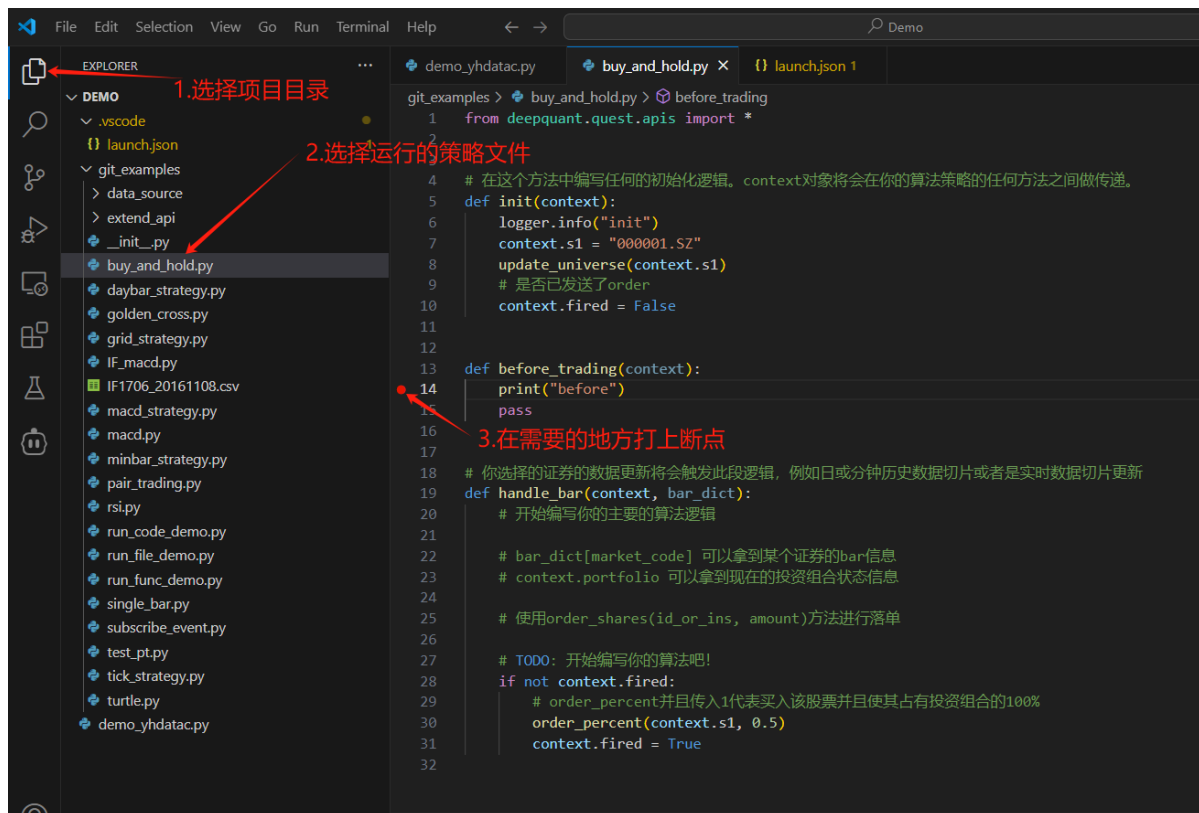
```
{
  // 使用 IntelliSense 了解相关属性。
  // 悬停以查看现有属性的描述。
  // 欲了解更多信息，请访问: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "name": "python: 模块",
      "type": "python",
      "request": "launch",
```

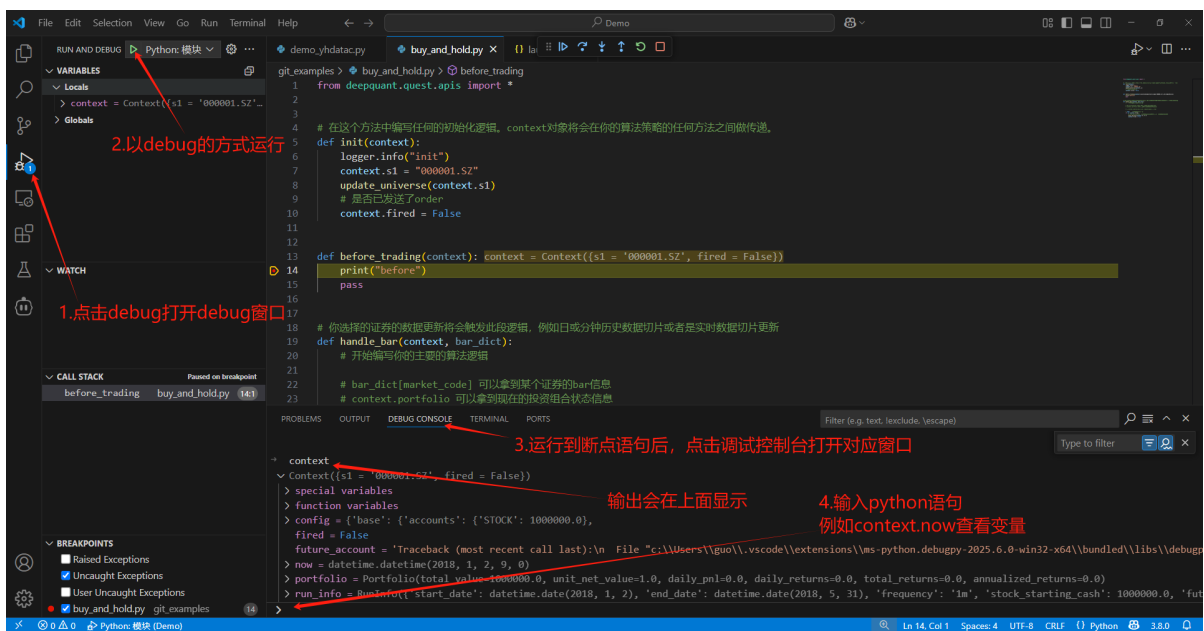
```

"module": "deepquant",
"args": [
    "run",
    "-f",
    "examples\\buy_and_hold.py",
    "-s",
    "2018-01-01",
    "-e",
    "2018-05-31",
    "-fq",
    "1m",
    "--plot",
    "--account",
    "stock",
    "1000000"
]
}
]
}

```

配置完成后，即可在需要的文件上打上断点，然后 debug 运行。





进阶教程

账户和持仓

DeepQuant 内部维护了多层级的账户和持仓结构，可以简化成如下的树形结构：

```
Portfolio()                                # 投资组合
├── Account("STOCK")                       # 股票账户
│   ├── Position("000001.SZ")              # 平安银行持仓
│   ├── Position("900000003", "SHORT")     # 300ETF购1月3900义务方持仓
│   ├── Position("128032.SZ")              # 双环转债持仓
│   ├── Position("AU9999.SGEX", "LONG")    # 上金所Au99.99黄金现货合约多头持仓
│   └── Position("004241")                 # 中欧时代先锋股票C持仓
└── Account("FUTURE")                     # 期货账户
    ├── Position("RB2010", "LONG")         # 螺纹钢2010多头持仓
    ├── Position("RB2010", "SHORT")        # 螺纹钢2010空头持仓
    └── Position("IO2004C4150", "LONG")    # 300INDEX2004购4150权利方持仓
```

- portfolio：投资组合，对应上图中最顶层的结构，表示当前策略中所有投资标的和剩余现金的总和。
 - 通过 `context.portfolio` 可以访问当前策略的 [Portfolio 对象](#)。
 - [Portfolio 对象](#) 具有 `portfolio_value`（总权益）、`unit_net_value`（净值）、`daily_pnl`（当日盈亏）、`daily_returns`（日收益率）等属性，如：

```
# 获取当前策略总权益
```

```
context.portfolio.portfolio_value
```

- account: 账户，对应上图中的第二层结构，DeepQuant 最多支持股票（STOCK）、期货（FUTURE）两种账户。
 - 运行策略需要为每个账户配置初始资金：
 - 命令行运行时，通过 `-a stock 100000 -a future 100000` 配置出初始资金
 - 函数入口运行时，通过如下配置设置初始资金：
- position: 持仓，对应上图的底层结构，表示策略所持有的每一只金融标的的仓位。
 - 持有的每个标的都有自己的 [Position 对象](#)，具有多空头的标的（期货、期权等）有多空两方向两个 [Position 对象](#)。
 - 可以通过 `get_position` 和 `get_positions` 两个接口获取持仓对象
 - `get_position` 接收标的代码、方向（可选）两个参数，方向参数默认为多头，返回对应的 [Position 对象](#)，如：

```
get_position("000001.SZ")
get_position("004241")
get_position("RB2010", "SHORT")
```

- `get_positions` 无参数，返回包含所有 [Position 对象](#) 的列表，如：

```
get_positions()
```

- 通过 `context.portfolio.positions` 或 `account.positions` 访问持仓的方式在未来的版本中或被弃用，如无必要 **请勿** 使用。

回测频率

DeepQuant 支持日、分钟、tick 三种频率级别的回测。

三种频率的回测会在相同的时机触发盘外约定函数 `init`、`before_trading`、`after_trading` 和集合竞价约定函数 `open_auction`，而盘中约定函数 `handle_bar` 和 `handle_tick` 在不同频率的回测中的触发情况则有所不同。

日回测

日回测适用于相对长周期的策略，日回测会忽略掉盘中所有市场变动细节，将每个标的每日的行情变动情况汇总成一根 k 线。

在日回测中，盘中约定函数 `handle_bar` 会在每个交易日**收盘**时被触发一次，在该函数中访问 `bar_dict` 参数可以获取到当前交易日的日 k 线，在该函数中发出的订单都会被以当日的**收盘价**撮合。

分钟回测

分钟回测适用于关注日内行情变动情况的策略，分钟回测会把交易时间按分钟切片，每个标的每分钟内的所有行情变动情况会被合成为一根具有高开低收等信息的分钟 k 线。

分钟回测中盘中约定函数 `handle_bar` 会在每分钟**结束**时触发一次，在该函数中访问 `bar_dict` 参数可以获取到刚刚结束的一分钟的分钟 k 线。例如，股票策略在每个交易日的 9:31 会首次触发 `handle_bar`，此次触发的 `handle_bar` 中可以访问到的分钟线为 9:30-9:31 的分钟线。分钟回测中亦可以通过 `history_bars` 接口获取历史日 k 线。

分钟回测可以[设置撮合方式](#)为**立即使用当前分钟线的收盘价撮合**或在**下一分钟以下一个分钟线的开盘价撮合**。

需要注意，因为不同品种的交易时间段不同，故需要策略告知 DeepQuant 该策略关注的标的品种，以便 DeepQuant 在正确的时间触发对应的 `handle_bar`：

- 若用户配置了股票账户的资金账号，则 DeepQuant 会在股票交易时间内触发 `handle_bar`，即 9:31 - 11:30 和 13:01 - 15:00。
- 若策略交易期货、期权合约，则需要预先（在 `init` 或 `handle_bar` 中）使用 [subscribe 接口](#)订阅所关注的合约，DeepQuant 将会触发对应合约交易时间的 `handle_bar`，[subscribe](#) 的使用方法：

```
subscribe('RB2010')
```

- 若订阅了多种交易时间不同的合约，或同时交易期货和股票，`handle_bar` 触发的时间段将是这些交易时间段的并集。
- 若在 `handle_bar` 中从 `bar_dict` 获取当前**未在交易**的标的的 k 线，策略将会获取到“无效”的 [Bar 对象](#)，该对象所有字段的值均为 NaN。

tick 回测

tick 回测为 DeepQuant 提供的最细粒度的回测。此处的 tick 实际上指的是 A 股市场的快照（snapshot）行情，通常情况下期货合约每 500ms 一个快照，股票每 3s 一个快照（DeepQuant 提供的快照行情直接来源于交易所，故以交易所发出的行情为准）。

tick 回测中盘中约定函数 `handle_tick` 接受两个参数 `context` 和 `tick`。参数 `tick` 的类型为 [TickObject](#)，不同于 `handle_bar` 中的 `bar_dict`，此处的 `tick` 仅包含单个标的的快照行情，也就是说，**每个标的的快照行情更新都会分别触发 `handle_tick` 的运行**。

运行 tick 回测时，策略所关注的所有标的都需要使用 [subscribe 接口](#) 订阅，以便 DeepQuant 触发对应标的的 `handle_tick` 的运行。

标的品种

DeepQuant 支持股票、期货、期权、可转债、场内基金和上金所现货合约等多种金融标的的回测。不同品种的标的在发单接口、费用计算、账户和仓位计算等方面有所差异。

股票和场内基金

DeepQuant 支持 A 股和 ETF、LOF 等场内基金回测

- 发单接口：股票和场内基金的六个发单函数在[上一章](#)已介绍过，此处不再赘述
- 账户设置：股票和场内基金的持仓归属于股票（STOCK）账户
- 分红拆分和复权：DeepQuant 中撮合、计算收益等适用的价格均为未复权价格，发生分红拆分等行为时 DeepQuant 会按照实际情况为策略账户补充现金和持仓。使用 `history_bars` 接口可以获取到在策略运行过程中动态复权的价格。
 - 分红再投资：开启分红再投资后 DeepQuant 会自动使用分红得到的现金买入相同的股票或场内基金持仓
 - 命令行运行时，使用 `--dividend-reinvestment` 参数开启分红再投资
 - 函数入口运行时，使用如下配置开启分红再投资：

```
{"mod": {"sys_accounts": {"dividend_reinvestment": True}}}
```

- 佣金和印花税：股票交易会产佣佣金和印花税。佣金费率默认万八，单笔订单最小佣金为 5 元；印花税对卖方单边征收，税率为 0.1%
 - 可以通过配置佣金倍率控制费率，如佣金倍率设置为 1.1，则 DeepQuant 使用的费率为 0.00088
 - 命令行运行时，使用 `--commission-multiplier 1.1` 配置佣金倍率
 - 函数入口运行时，使用如下配置设置佣金赔率：

```
{"mod": {"sys_transaction_cost": {"commission_multiplier": 1.1}}}
```

- T+1：股票交易默认开启 T+1 限制，即当日买入的股票需要等到下个交易日才能卖出
 - 命令行运行时，使用 `--no-stock-t1` 关闭 T+1 限制
 - 函数入口运行时，使用如下配置关闭 T+1 限制

```
{"mod": {"sys_accounts": {"stock_t1": False}}}
```

期货

DeepQuant 支持期货回测

- 发单接口：不同于股票，期货可使用如下四个接口发单，详细用法可查阅[API手册](#)。
 - buy_open：多头开仓，接受合约代码、交易数量、限价单价格（可选）为参数，例如：

```
buy_open("RB2010", 2)
```

- sell_close：多头平仓，接受合约代码、交易数量、限价单价格（可选）、是否平今（可选）为参数，例如：

```
sell_close("RB2010", 1, price=3100, close_today=True)
```

- sell_open: 空头开仓, 参数与 `buy_open` 相同
- buy_close: 空头平仓, 参数与 `sell_close` 相同
- 账户设置: 期货持仓归属于期货 (FUTURE) 账户
- 保证金交易: 期货采用保证金交易, 持有期货仓位会占用保证金, 这部分资金会被冻结, 不能再用于发单, 保证金会在平仓时解冻。
 - DeepQuant 使用的保证金率可以通过 `Instrument` 接口查看, 该接口接收合约代码参数, 返回 `Instrument` 对象, 例如使用如下代码查询 RB2010 的保证金率:

```
instruments("RB2010").margin_rate
```

- 可以通过设置保证金倍率来调整 DeepQuant 的保证金率, 实际使用的保证金率为默认的保证金率乘以保证金倍率
 - 命令行运行时, 使用 `-mm 1.1` 或 `--margin-multiplier 1.1` 设置保证金倍率
 - 函数入口运行时, 使用如下配置设置保证金倍率

```
{"base": {"margin_multiplier": 1.1}}
```

- 逐日盯市: 期货采用“逐日盯市”制度, 每日盘后会进行结算, 将浮盈浮亏计入现金。

期权

DeepQuant 支持商品、股指、ETF 期权回测。

- 发单接口: 交易期权使用与期货相同的四个发单接口。
- 账户设置: 根据实际市场中所在交易所不同, 期权持仓分属股票 (STOCK) 和期货 (FUTURE) 账户, 其中 ETF 期权属于股票账户, 商品期权和股指期货期权属于期货账户。
- 行权
 - 行权采用现金交割, 即将行权产生的盈利或亏损直接计入现金中。
 - 主动行权: 期权可通过 `exercise` 接口主动行权, 该函数接收合约代码和行权数量两个参数, 例如:

```
exercise("M1905C2350", 2)
```

- 被动行权: 期权持有至到期日将会触发自动行权。对于权利方 (多头) 持仓, 若 DeepQuant 判定行权可以盈利, 则触发自动行权, 否则仓位作废; 而义务方 (空头) 持仓会在 DeepQuant 判定对手方可以盈利时触发行权
- 行权滑点: 为了模拟真实市场中行权委托与到账间这段时间段内底层标的价格发生波动带来的风险, DeepQuant 提供了行权滑点功能, 通过配置行权滑点, 可以使得行权盈利的判定更为严苛。对于认购期权, 0.1 的滑点代表即使在交割日标的价格降低 10%, 本次行权仍然能盈利; 而对于认沽期权, 代表在交割日即使标的价格上涨 10%, 仍然能盈利。默认行权滑点为 0。行权滑点只会影响自动行权的判定, 而不影响行权交割的金额。
 - 命令行运行时, 使用如下参数设置行权滑点:

```
-mc option.exercise_slippage 0.1
```

- 函数入口运行时，使用如下配置设置行权滑点：

```
{"mod": {"option": {"exercise_slippage": 0.1}}}
```

- 权利金和保证金
 - 权利方（多头）：开仓需要缴纳权利金，该过程与股票的开仓类似
 - 义务方（空头）：开仓会收取权利金并付出保证金，保证金会被冻结（类似期货开仓）；同时义务方也采取逐日盯市制度，每日盘后结算，浮盈浮亏将被计入现金。

可转债

DeepQuant 支持可转换债券、场内公开交易的可交换债券、分离交易可转债（债券等）的回测。

- 发单接口：可转债使用 `order_shares`、`order_value`、`order_percent`、`order_target_value`、`order_target_percent` 五个接口下单，用法与股票相同
- 账户设置：可转债持仓归属于股票（STOCK）账户
- 回售和转股：可转债支持主动发起回售或转股，使用 [exercise 接口](#)，相比于期权行权，除了合约代码和数量两个参数，还加入了第三个参数用于区分本次行权是转股还是回售，如：

```
exercise("132003.SH", 100, convert=False) # 回售
exercise("132003.SH", 100, convert=True)  # 转股
```

- 本息偿付：可转债发生付息时，利息将进入对应账户的现金；发生强制赎回时，仓位将被清空，对应账户的现金会按照强赎时实际的现金流变动。

场外基金

- 发单接口：场外基金可使用如下六个接口发单，详细用法可查阅 [API 手册](#)
 - `subscribe_value`：按申购金额申购基金，接受合约代码、交易金额为参数，例如：

```
subscribe_value("004241", 1000)
```

- `subscribe_shares`：按份额申购基金，接受合约代码、交易数量为参数，例如：

```
subscribe_shares("004241", 500)
```

- `subscribe_percent`：按可用资金权重申购基金，接受合约代码、占现有可用资金的百分比为参数，例如：

```
subscribe_percent("004241", 0.1)
```

- `redeem_shares`：按份额赎回基金，接受合约代码、交易数量为参数，例如：

```
redeem_shares("004241", 500)
```

- `redeem_value`：按金额赎回基金，接受合约代码、交易金额为参数，例如：

```
redeem_value("004241", 1000)
```

- `redeem_percent`: 按剩余份额权重赎回基金, 接受合约代码、占剩余份额的百分比为参数, 例如:

```
redeem_percent("004241", 0.1)
```

- 账户设置: 场外基金持仓归属于股票 (STOCK) 账户
- 费用: 所有基金前端收费, 支持通过参数配置前端费率, 默认前端费率 1.5%
 - 命令行运行时, 使用 `--fee-ratio 0.015` 设置基金前端费率
 - 函数入口运行时, 使用如下配置设置前端费率:

```
{"fund": {"fee_ratio": 0.015}}
```

- 赎回不收取费用
- 申购赎回状态限制: 可通过参数配置是否根据状态限制申赎, 默认开启申购赎回状态限制
 - 命令行运行时, 使用 `--status-limit` 参数开启申购赎回状态限制
 - 函数入口运行时, 使用如下配置开启申购赎回状态限制:

```
{"fund": {"status_limit": True}}
```

- 申购金额限制: 是否限制申购金额的上下限, 默认开启。
 - 命令行运行时, 使用 `--subscription-limit` 参数开启申购金额限制, 若开启申购上下限限制, 则超过上限时部分成交, 低于下限时拒单, 若不开启则以申购金额成交。
 - 函数入口运行时, 使用如下配置开启申购金额限制:

```
{"fund": {"subscription_limit": True}}
```

- 申购赎回到账时间: 可通过参数设置所有基金的申购赎回到账时间。
 - 函数入口运行时, 使用如下配置设置基金申购赎回到账时间:

```
{"fund": {  
  # 基金申购份额到账时间  
  "subscription_receiving_days": 1,  
  # 基金赎回金额到账时间  
  "redemption_receiving_days": 3,  
}  
}
```

- 分红拆分: 基金发生分红拆分时, DeepQuant 自动处理为策略账户补充现金或持仓,
 - 分红再投资: 前文介绍的股票分红再投资参数同样适用于场外公募基金, 开启分红再投资后 DeepQuant 会自动使用分红得到的现金买入基金份额, 分红再投资份额到账时间和基金申购设置到账时间一致。
 - 命令行运行时, 使用 `--dividend-reinvestment` 参数开启分红再投资
 - 函数入口运行时, 使用如下配置开启分红再投资:

```
{"mod": {"sys_accounts": {"dividend_reinvestment": True}}}
```

对于货币基金一律采用分红再投资，不受 `--dividend-reinvestment` 参数影响

上金所现货

DeepQuant 支持上海黄金交易所交易的黄金、白银、铂金等现货合约的回测。

- 发单接口：与期货交易相同，上金所现合约使用 `buy_open`、`sell_close`、`sell_open` 和 `sell_close` 四个接口下单。
- 账户设置：上金所现货合约持仓归属于股票（STOCK）账户。
- 保证金交易：与期货类似，上金所现货合约采用保证金交易，同样可以配置保证金倍率，同样采用“逐日盯市”制度。

事前风控

DeepQuant 中发出的订单在撮合前会经过多项事前风控，某项风控不通过会导致下单失败，部分事前风控可以自定义配置。

- 验资风控：检验当前可用资金是否足够下单，默认开启。关闭该风控项可能导致剩余资金为负数
 - 命令行运行策略时，使用 `--no-cash-validation` 以关闭验资风控
 - 函数入口运行策略时，使用如下配置以关闭验资风控：

```
{"mod": {"sys_risk": {"validate_cash": False}}}
```

- 验券风控：针对卖单（平仓单、行权单）检验当前可平仓位是否足够平仓
- 自成交风控：针对新发订单，检验当前是否有方向相反的、存在和新发订单相互成交风险的挂单
 - 自成交风控默认关闭，通过命令行运行策略时，可以使用如下参数开启：

```
-mc sys_risk.validate_self_trade true
```

- 通过函数入口运行时，可以使用如下配置开启：

```
{"mod": {"sys_risk": {"validate_self_trade": True}}}
```

- 债券发行总额风控：针对可转债订单，检验新发订单和已有持仓票面价值总和是否超过债券发行总额
- 行权日期风控：检验行权日期是否合法，如欧式期权仅可在到期日行权，可转债仅可在转股期内转股、仅可在回售登记日期范围内回售

模拟撮合

DeepQuant 在回测中会模拟交易所的行为撮合策略发出的订单。DeepQuant 内置多种撮合和滑点模型，可按需呈现出对真实市场不同程度对模拟。

撮合方式

DeepQuant 支持五种撮合模型，不同撮合模型之前的区别在于撮合的时机以及如何决定撮合使用的参考价格。

使用命令行运行时，使用 `-mt` 或 `--matching-type` 参数设置撮合类型，如：

```
# 设置撮合类型为当前 bar 收盘价撮合
--matching-type current_bar
```

使用函数入口运行时，使用如下的配置设置撮合类型：

```
# 设置撮合类型为当前 bar 收盘价撮合
{"mod": {"sys_simulation": {"matching_type": "current_bar"}}}
```

所有可用的撮合方式如下：

- `current_bar`：立即使用当前 k 线的收盘价作为参考价撮合，可在日回测和分钟回测中使用，该回测方式是 DeepQuant 默认的撮合方式
- `next_bar`：在下一个 `handle_bar` 触发前使用下一跟 k 线的开盘价撮合，可在分钟回测中使用
- `last`：在下一个 `handle_tick` 触发前使用该 tick 的最新价撮合，可在 tick 回测中使用
- `best_own`：在下一个 `handle_tick` 触发前使用该 tick 的己方最优报盘价格撮合，可在 tick 回测中使用
- `best_counterparty`：在下一个 `handle_tick` 触发前使用该 tick 的对手方最优报盘价格撮合，可在 tick 回测中使用
- `vwap`：成交量加权平均价撮合，可在日回测和分钟回测中使用

需要注意：

- `next_bar` 撮合方式在日回测中已不适用，如果需要当前开盘成交撮合，可以使用[open_auction](#)函数在盘前集合竞价时发单，以当日开盘价撮合。
- 对于场外基金全部采用当日单位净值成交。

滑点

DeepQuant 支持两种滑点模型，以模拟真实交易中实际成交价与挂单价格存在差异的情况

使用命令行运行时，使用 `--slippage-model` 参数设置滑点模型，使用 `-sp` 或 `--slippage` 参数设置“滑点值”，如：

```
# 成交价会产生千分之一的恶化
--slippage-model PriceRatioSlippage --slippage 0.001
```

使用命令行运行时，使用如下的配置设置滑点：

```
# 成交价会产生千分之一的恶化
{"mod": {"sys_simulation": {
    "slippage_model": "PriceRatioSlippage",
    "slippage": 0.001
}}}
```

可选的滑点模型如下：

- `PriceRatioSlippage`：成交价格按照一定比例进行恶化，“滑点值”即为价格恶化的比例
- `TickSizeSlippage`：成交价按照最小价格变动单位进行恶化，价格恶化的值为“滑点值”乘以标的的最小价格变动单位

场外基金不支持滑点设置。

成交量限制

在日和分钟回测中，DeepQuant 会对订单的成交量进行限制，每个 `handle_bar` 中发出的订单总成交量不能超过当前 k 线所覆盖时间段内市场上该标的总成交量的一定比例，订单在该比例内的部分会被撮合，超出部分会被拒单。该比例默认为 0.25。

使用命令行运行时，通过 `-mc sys_simulation.volume_limit` 和 `-mc sys_simulation.volume_percent` 参数设置成交量限制情况，如：

```
# 开启成交量限制并把订单成交量限制在市场上总成交量的 10%
-mc sys_simulation.volume_limit true -mc sys_simulation.volume_percent 0.1

# 关闭成交量限制
-mc sys_simulation.volume_limit false
```

使用函数入口运行策略时，使用如下配置设置成交量限制情况：

```
# 开启成交量限制并把订单成交量限制到市场上总成交量的 10%
{"mod": {"sys_simulation": {
    "volume_limit": True,
    "volume_percent": 0.1
}}}

# 关闭成交量限制
{"mod": {"sys_simulation": {
    "volume_limit": False,
}}}
```

场外基金不支持成交量限制设置。

自定义基准

DeepQuant中Quest回测模块支持设置单个合约作为基准外，还对支持 `order_book_id` 加权作为回测的基准。

使用命令行运行时，使用如下的配置设置指数加权基准：

```
--benchmark 000300.SZ:0.7,000905.SH:0.3
```

使用函数入口运行策略时，使用如下配置设置指数加权基准：

```
"mod": {"sys_analyser": {
    "benchmark": {
        "000300.SZ": 0.7,
        "000905.SH": 0.3
    }
}}
```


出入金

DeepQuant中Quest回测模块 支持回测过程中增加或减少资金，以满足回测过程中账户资金调整的需求，支持在 `handle_bar` 中调用。

- 出金：通过 `withdraw(account_type, amount)` 接口对账户减少资金，举例如下：

```
#对期货账户减少10w资金
withdraw("FUTURE", 100000)
```

- 入金：通过 `deposit(account_type, amount)` 接口对账户增加资金，举例如下：

```
#对股票账户增加10w资金
deposit("STOCK", 100000)
```

管理费用

DeepQuant中Quest回测模块 支持通过参数配置管理费，每日计提管理费。

使用命令行运行时，使用如下的配置对股票账户收取 0.02%的管理费（每个交易日收取，也可以对 future 账户收取管理费）：

```
--management-fee stock 0.02%

#管理费 = total_value * 管理费率,每日计提
```

使用函数入口运行时，使用如下配置设置管理费率：

```
"mod": {"sys_simulation": {
    "enabled": True,
    "management_fee": [("stock", 0.02%)],
  }
}
```

增量回测

DeepQuant中Quest回测模块 支持日级别增量回测的功能，即将当前策略回测的结果数据保存到本地，后续对相同策略运行回测时在该策略本地回测结果的基础上继续运行。

- mod 操作
可以使用如下命令开启增量回测的 mod，默认增量回测的 mod 是关闭的。

```
deepquant mod enable incremental
```

使用如下命令关闭增量回测的 mod

```
deepquant mod disable incremental
```

使用如下命令查看目前开启了哪些 mod

```
deepquant mod list
```

- 参数设置

开启增量回测的 mod 后，使用命令行运行时，使用 `--persist-folder` 指定存储文件路径（启动 mod 不设置路径增量无效），使用 `--strategy-id` 指定策略运行 id（若不指定，默认为 1），举例如下：

```
# 在当前目录的/persist下查看是否有文件名为2的文件夹，若有则读取文件内容，在本地保存回测结果的基础上运行增量回测，若没有则在当前目录/persist下生成一个命名为2的文件夹保存本次回测的结果
--persist-folder . --strategy-id 2
```

使用函数入口运行时，配置如下：

```
{
  "mod": {
    "incremental": {
      'enabled': True,
      "persist_folder": '.',
      "strategy_id": 2,
    }
  }
}
```

策略内参数配置

使用命令行运行策略时，可以使用与函数入口运行策略时传入的 `config` 相同的格式编写配置，并把配置写在策略文件内。策略内参数配置的优先级低于命令行参数的优先级。

策略内配置需要赋值给策略文件内的全局变量 `__config__`，如将下述内容写入 `macd.py` 文件：

```
import talib

__config__ = {
  "base": {
    "accounts": {
      "STOCK": 100000,
    },
    "start_date": "20190101",
    "end_date": "20191231",
  },
  "mod": {
    "sys_analyser": {
      "plot": True,
      "benchmark": "000300.SH"
    }
  }
}

def init(context):
    context.stock = "000001.sz"

    context.SHORTPERIOD = 12
```

```

context.LONGPERIOD = 26
context.SMOOTHPERIOD = 9
context.OBSERVATION = 100

def handle_bar(context, bar_dict):
    prices = history_bars(context.stock, context.OBSERVATION, '1d',
        'close_price')
    macd, macd_signal, _ = talib.MACD(
        prices, context.SHORTPERIOD, context.LONGPERIOD, context.SMOOTHPERIOD
    )

    if macd[-1] > macd_signal[-1] and macd[-2] < macd_signal[-2]:
        order_target_percent(context.stock, 1)

    if macd[-1] < macd_signal[-1] and macd[-2] > macd_signal[-2]:
        if get_position(context.stock).quantity > 0:
            order_target_percent(context.stock, 0)

```

可以直接使用如下命令运行策略，仅仅需要使用 `-f` 参数指定策略文件，不再需要传入更多参数：

```
deepquant run -f macd.py
```

定时器

除了约定函数以供策略逻辑在市场发生变动时运行外，DeepQuant 还提供了定时器功能以供策略逻辑周期性地执行。

定时器暂只支持**股票日**、**分钟级回测**。

定时器的使用方式是在 `init` 中通过定时器接口注册函数，被注册的函数会在符合指定的“时间规则”时被调用，如：

```

# 每日开市时打印当前剩余资金

#scheduler调用的函数需要包括context, bar_dict两个输入参数
def log_cash(context, bar_dict):
    logger.info("Remaning cash: %r" % context.portfolio.cash)

def init(context):
    #...
    # 每天运行一次
    scheduler.run_daily(log_cash)

```

除每日运行之外，定时器还支持注册每周、每月运行的函数，并且支持指定如“每月的第 N 个交易日”或“每天的第 N 分钟”的时间规则。详细的使用方法可查阅 [scheduler 定时器接口手册](#)。

读取本地持仓权重运行回测

支持读取本地持仓权重样例运行回测，举例如本地调仓权重样例如下：

TRADE_DT	TICKER	NAME	TARGET_WEIGHT
20191202	000001.SZ	平安银行	0.03

TRADE_DT	TICKER	NAME	TARGET_WEIGHT
20191202	002916.SZ	深南电路	0.02
...
20200102	002916.SZ	深南电路	0.02

简单样例策略如下，若需要一个完整的策略范例请点击：[根据本地持仓权重运行回测范例](#)

```
import pandas
import numpy

import deepquant.quest.datac as yhdatac

# pip install -i https://pypi.tuna.tsinghua.edu.cn/simple xlrd
# deepquant run -f holding_target_position_simplified.py

__config__ = {
    "base": {
        "start_date": "20191201",
        "end_date": "20200930",
        "accounts": {
            "stock": 100000000,
        },
    },
}

# 在这个方法中编写任何的初始化逻辑。context对象将会在你的算法策略的任何方法之间做传递。
def init(context):
    df = pandas.read_excel('调仓权重样例.xlsx', dtype={
        'TARGET_WEIGHT': numpy.float, 'TICKER': numpy.str, 'TRADE_DT': numpy.int
    })
    df['TICKER'] = df['TICKER'].apply(lambda x: yhdatac.id_convert(x) if ".OF"
    not in x else x)
    context.target = {d: t.set_index("TICKER")["TARGET_WEIGHT"].to_dict() for d,
    t in df.groupby("TRADE_DT")}

# 你选择的证券的数据更新将会触发此段逻辑，例如日或分钟历史数据切片或者是实时数据切片更新
def handle_bar(context, bar_dict):
    today = context.now.year * 10000 + context.now.month * 100 + context.now.day
    if today not in context.target:
        return
    order_target_portfolio(context.target[today])
```

常见问题

为什么部分 API 与 Datac 中的 API 同名但用法不同？

□Datac 的 API 与 DeepQuant 提供的 API 使用场景不同。DeepQuant 提供的 API 通常在策略内调用，故更多考虑的是如何更方便地调取到“当前时间”的数据以及如何避免策略无意间调用到未来数据。

如果希望在策略中调用 datac 的 API，需要显式地引入 datac 包，如：

```
import deepquant.quest.datac as yhdatac
import deepquant.data.interface.gid as gid

gid.init('user', 'password')
print(yhdatac.get_price('000001.SZ', start_date='2015-04-01', end_date='2015-04-12'))
```

DeepQuant 中的接口是线程安全的吗？

不是。请勿在多线程环境中运行 DeepQuant 或在策略中开启子线程。任何情况下每个进程中同一时间应只有一个策略实例在运行，否则可能会导致 DeepQuant 出现不可预测的行为。

为什么我已经在终端配置了 DeepQuant 的 License，但在 IDE/编辑器中依然会遇到 License 不生效的情况？

该问题通常会发生在 Linux/macOS 中。

DeepQuant 通过环境变量存储 License 等配置信息。在 Linux 和 macOS 中，环境变量是通过在 bash 启动文件（`.bash_profile`、`.bashrc`、`.zshrc`）中添加命令的方式设置的。若您使用的 IDE 或编辑器因为某些原因未能读取到环境变量，则会出现执行策略或脚本时报出无权限错误的情况。

您需要了解：

- `deepquant license` 命令会设置 `DEEPQUANT_LICENSE` 环境变量

解决问题的方法：

1. 首先确认您的终端内能够正确读取到上述环境变量
2. 尝试在上述终端内启动 IDE
3. 在 IDE 内执行代码以确认能否读取上述环境变量

若问题依旧，您可以：

1. 尝试升级 IDE 版本，并联系 IDE 提供方寻求帮助
2. 在 IDE 的设置中或您的脚本中手动配置上述环境变量

示例策略

多股票 RSI 算法示例

```
import talib

def init(context):

    context.s1 = "000001.SZ"
```

```

context.s2 = "601988.SH"
context.s3 = "000068.SZ"
context.stocks = [context.s1, context.s2, context.s3]

context.TIME_PERIOD = 14
context.HIGH_RSI = 85
context.LOW_RSI = 30
context.ORDER_PERCENT = 0.3

def handle_bar(context, bar_dict):
    # 对我们选中的股票集合进行loop，运算每一只股票的RSI数值
    for stock in context.stocks:
        # 读取历史数据
        prices = history_bars(stock, context.TIME_PERIOD+1, '1d', 'close_price')

        # 用Talib计算RSI值
        rsi_data = talib.RSI(prices, timeperiod=context.TIME_PERIOD)[-1]

        cur_position = context.portfolio.positions[stock].quantity
        # 用剩余现金的30%来购买新的股票
        target_available_cash = context.portfolio.cash * context.ORDER_PERCENT

        # 当RSI大于设置的上限阈值，清仓该股票
        if rsi_data > context.HIGH_RSI and cur_position > 0:
            order_target_value(stock, 0)

        # 当RSI小于设置的下限阈值，用剩余cash的一定比例补仓该股
        if rsi_data < context.LOW_RSI:
            logger.info("target available cash caled: " +
                str(target_available_cash))
            # 如果剩余的现金不够一手 - 100shares，那么会被DeepQuant 的order management
            # system reject掉
            order_value(stock, target_available_cash)

```

商品期货跨品种配对交易

该策略为分钟级别回测。运用了简单的移动平均以及布林带（Bollinger Bands）作为交易信号产生源。有关对冲比率（HedgeRatio）的确定，您可以在我们的研究平台上面通过 import statsmodels.api as sm 引入 statsmodels 中的 OLS 方法进行线性回归估计。具体估计窗口，您可以根据自己策略需要自行选择。

策略中的移动窗口选择为 60 分钟，即在每天开盘 60 分钟内不做任何交易，积累数据计算移动平均值。当然，这一移动窗口也可以根据自身需要进行灵活选择。下面例子中使用了黄金与白银两种商品期货进行配对交易。简单起见，例子中期货的价格并未做对数差处理。

```

# 可以自己import我们平台支持的第三方python模块，比如pandas、numpy等。
import numpy as np

# 在这个方法中编写任何的初始化逻辑。context对象将会在你的算法策略的任何方法之间做传递。
def init(context):
    context.s1 = 'AG1612'
    context.s2 = 'AU1612'

```

```

# 设置全局计数器
context.counter = 0

# 设置滚动窗口
context.window = 60

# 设置对冲手数,通过研究历史数据进行价格序列回归得到该值
context.ratio = 15

context.up_cross_up_limit = False
context.down_cross_down_limit = False

# 设置入场临界值
context.entry_score = 2

# 初始化时订阅合约行情。订阅之后的合约行情会在handle_bar中进行更新
subscribe([context.s1, context.s2])

# before_trading此函数会在每天交易开始前被调用,当天只会被调用一次
def before_trading(context):
    # 样例商品期货在回测区间内有夜盘交易,所以在每日开盘前将计数器清零
    context.counter = 0

# 你选择的期货数据更新将会触发此段逻辑,例如日线或分钟线更新
def handle_bar(context, bar_dict):

    # 获取当前一对合约的仓位情况。如尚未有仓位,则对应持仓量都为0
    position_a = context.portfolio.positions[context.s1]
    position_b = context.portfolio.positions[context.s2]

    context.counter += 1
    # 当累积满一定数量的bar数据时候,进行交易逻辑的判断
    if context.counter > context.window:

        # 获取当天历史分钟线价格队列
        price_array_a = history_bars(context.s1, context.window, '1m',
                                     'close_price')
        price_array_b = history_bars(context.s2, context.window, '1m',
                                     'close_price')

        # 计算价差序列、其标准差、均值、上限、下限
        spread_array = price_array_a - context.ratio * price_array_b
        std = np.std(spread_array)
        mean = np.mean(spread_array)
        up_limit = mean + context.entry_score * std
        down_limit = mean - context.entry_score * std

        # 获取当前bar对应合约的收盘价格并计算价差
        price_a = bar_dict[context.s1].close_price
        price_b = bar_dict[context.s2].close_price
        spread = price_a - context.ratio * price_b

        # 如果价差低于预先计算得到的下限,则为建仓信号,'买入'价差合约
        if spread <= down_limit and not context.down_cross_down_limit:

```

```

# 可以通过logger打印日志
logger.info('spread: {}, mean: {}, down_limit: {}'.format(spread,
mean, down_limit))
logger.info('创建买入价差中...')

# 获取当前剩余的应建仓的数量
qty_a = 1 - position_a.buy_quantity
qty_b = context.ratio - position_b.sell_quantity

# 由于存在成交不超过下一bar成交量25%的限制,所以可能要通过多次发单成交才能够成功建
仓

if qty_a > 0:
    buy_open(context.s1, qty_a)
if qty_b > 0:
    sell_open(context.s2, qty_b)
if qty_a == 0 and qty_b == 0:
    # 已成功建立价差的'多仓'
    context.down_cross_down_limit = True
    logger.info('买入价差仓位创建成功!')

# 如果价差向上回归移动平均线,则为平仓信号
if spread >= mean and context.down_cross_down_limit:
    logger.info('spread: {}, mean: {}, down_limit: {}'.format(spread,
mean, down_limit))
    logger.info('对买入价差仓位进行平仓操作中...')

# 由于存在成交不超过下一bar成交量25%的限制,所以可能要通过多次发单成交才能够成功建
仓

qty_a = position_a.buy_quantity
qty_b = position_b.sell_quantity
if qty_a > 0:
    sell_close(context.s1, qty_a)
if qty_b > 0:
    buy_close(context.s2, qty_b)
if qty_a == 0 and qty_b == 0:
    context.down_cross_down_limit = False
    logger.info('买入价差仓位平仓成功!')

# 如果价差高于预先计算得到的上限,则为建仓信号,'卖出'价差合约
if spread >= up_limit and not context.up_cross_up_limit:
    logger.info('spread: {}, mean: {}, up_limit: {}'.format(spread, mean,
up_limit))
    logger.info('创建卖出价差中...')
    qty_a = 1 - position_a.sell_quantity
    qty_b = context.ratio - position_b.buy_quantity
    if qty_a > 0:
        sell_open(context.s1, qty_a)
    if qty_b > 0:
        buy_open(context.s2, qty_b)
    if qty_a == 0 and qty_b == 0:
        context.up_cross_up_limit = True
        logger.info('卖出价差仓位创建成功!')

# 如果价差向下回归移动平均线,则为平仓信号
if spread < mean and context.up_cross_up_limit:

```



```

        logger.info('spread: {}, mean: {}, up_limit: {}'.format(spread, mean,
up_limit))

        logger.info('对卖出价差仓位进行平仓操作中...')
        qty_a = position_a.sell_quantity
        qty_b = position_b.buy_quantity
        if qty_a > 0:
            buy_close(context.s1, qty_a)
        if qty_b > 0:
            sell_close(context.s2, qty_b)
        if qty_a == 0 and qty_b == 0:
            context.up_cross_up_limit = False
        logger.info('卖出价差仓位平仓成功!')

```

期权回测样例

通过沪深 300 股指期权认购认沽评价构造指数的空头，结合股沪深 300 股指期货多头进行对冲买入并持有策略。

```

import rqalpha_plus
import rqalpha_mod_option

__config__ = {
    "base": {
        "start_date": "20200101",
        "end_date": "20200221",
        "frequency": '1d',
        "accounts": {
            # 股指期权使用 future 账户
            "future": 1000000
        }
    },
    "mod": {
        "option": {
            "enabled": True,
            "exercise_slippage": 0
        },
        "sys_simulation": {
            "enabled": True,
            "matching_type": 'current_bar',
            "volume_limit": False,
            "volume_percent": 0,
        },
        "sys_analyser": {
            "plot": True,
        },
    },
}

def init(context):
    context.s1 = 'IO2002C3900'
    context.s2 = 'IO2002P3900'
    context.s3 = 'IF2002'

    subscribe(context.s1)
    subscribe(context.s2)

```

```

subscribe(context.s3)

context.counter = 0
print('***** INIT *****')

def before_trading(context):
    pass

def handle_bar(context, bar_dict):
    context.counter += 1
    if context.counter == 1:
        sell_open(context.s1, 3)
        buy_open(context.s2, 3)
        buy_open(context.s3, 1)

def after_trading(context):
    pass

```

转债平价溢价率作为信号的分钟回测

```

import numpy as np

__config__ = {
    "base": {
        "start_date": "20180601",
        "end_date": "20180610",
        "frequency": '1m',
        "accounts": {
            "stock": 1000000 # 可转债使用 stock 账号
        }
    },
    "mod": {
        "sys_simulation": {
            "enabled": True,
            "matching_type": 'current_bar',
            # 是否允许涨跌停状态下买入、卖出
            "price_limit": False,
            # 是否开启成交量限制
            "volume_limit": False,
        },
        "convertible": {
            "enabled": True,
            # 设置转债回测的佣金费率
            "commission_rate": 0,
            # 设置转债回测的最小佣金
            "min_commission": 0,
        },
        "sys_analyser": {
            "plot": True,
        },
    },
}

def init(context):
    context.o = "110030.SH"

```

```

subscribe(context.o)
context.count = 0
context.exercise_flag = False
context.stock_id = instruments(context.o).stock_code
context.conversion_value = 0

def handle_bar(context, bar_dict):
    context.count += 1
    cb_price = bar_dict[context.o].close_price
    stock_price = bar_dict[context.stock_id].close_price
    # 转债的转股价值
    context.conversion_value = 100/7.24 * stock_price

    # 转债的平价溢价率
    ratio = cb_price / context.conversion_value - 1
    quantity = get_position(context.o, POSITION_DIRECTION.LONG).quantity
    if ratio < 0.31 and quantity < 2000:
        print('当前可转债平价溢价率为 {}'.format(ratio))
        order_shares(context.o, 100)

    if ratio > 0.36 and quantity > 0:
        print('当前可转债平价溢价率为 {}'.format(ratio))
        order_shares(context.o, -1*quantity)

```

公募基金回测简单样例

```

INIT_CASH = 100000

__config__ = {
    "base": {
        "start_date": "20190105",
        "end_date": "20200809",
        "accounts": {
            "stock": INIT_CASH
        }
    },
    "mod": {
        "sys_progress": {
            "enabled": True,
            "show": True
        }, "sys_analyser": {
            "enabled": True,
            "plot": True
        }, "fund": {
            # 基金申购前端费率
            "fee_ratio": 0.015,
            # 基金份额到账时间
            "subscription_receiving_days": 1,
            # 赎回金回款时间
            "redemption_receiving_days": 3,
            # 申购金额上下限检查限制
            "subscription_limit": True,
            # 申购状态检查限制
            "status_limit": True,
        },
    },
}

```

```

        'sys_simulation': {
            'enabled': True,
            'matching_type': 'current_bar',

        },
    }

}

# 在这个方法中编写任何的初始化逻辑。context对象将会在你的算法策略的任何方法之间做传递。
def init(context):
    logger.info("init")
    context.s1 = "004241.SZ"

    context.fired = False

def before_trading(context):
    pass

# 你选择的证券的数据更新将会触发此段逻辑，例如日或分钟历史数据切片或者是实时数据切片更新
def handle_bar(context, bar_dict):

    if not context.fired:

        subscribe_value(context.s1, INIT_CASH)
        context.fired = True

    if context.portfolio.total_returns > 0.4 or context.portfolio.total_returns <
-0.2:
        context.quantity = get_position(context.s1).quantity
        if context.quantity > 0:
            redeem_shares(context.s1, context.quantity)

```

黄金现货回测样例

- 引入黄金现货 AUTD.SGEX 合约和黄金期货主力合约 AU2006 进行配对交易。
- 两个合约的合约乘数相同，都是 1000，所以价差数量比例为 1:1。合约乘数可以通过 datac.instruments 查询到，对应字段为 contract_multiplier
- 计算期货、现货历史价差的最大最小值，如果当前价差超过历史 10 日最大价差，认为价差即将收敛，做多现货做空期货。

```

__config__ = {
    'base': {
        'start_date': '20200101',
        'end_date': '20200321',
        'frequency': '1d',
        # 保证金倍率。基于基础保证金水平进行调整
        'margin_multiplier': 1,
        # 商品现货回测这里使用 stock 账户
        'accounts': {

```

```

        'stock': 1000000,
        'future': 1000000,
    },
    # 期货交易佣金设置
    'future_info': {
        # 期货品种，如不设置，则按照默认费用进行收取
        'AU': {
            # 平仓费率
            'close_commission_ratio': 0.00005,
            # 开仓费率
            'open_commission_ratio': 0.00005,
            # 平今费率
            'close_commission_today_ratio': 0,
            # BY_MONEY 为按照名义价值收取，BY_VOLUME 为根据成交合约张数收取
            'commission_type': 'BY_MONEY',
        },
    },
},
'mod': {
    'spot': {
        'enabled': True,
        'commission_multiplier': 0,
    },
    'sys_simulation': {
        'enabled': True,
        # 是否开启信号模式。如果开启，限价单将按照指定价格成交，并且不受撮合成交量限制
        'signal': False,
        'matching_type': 'current_bar',
        'volume_limit': True,
        'volume_percent': 0.001,
    },
    'sys_analyser': {
        'plot': True,
    },
}
}

def init(context):
    context.s1 = 'AUTD.SGEX'
    context.s2 = 'AU2006'
    subscribe(context.s1)
    context.counter = 0

def handle_bar(context, bar_dict):
    # 通过 bar_dict 获得当日数据计算当日价差
    current_spread = bar_dict[context.s2].close - bar_dict[context.s1].close

    # 通过 history_bars 获得历史价格序列，计算移动窗口历史价差的最大、最小值
    spot_price = history_bars(context.s1, 10, '1d', 'close_price')
    future_price = history_bars(context.s2, 10, '1d', 'close_price')
    max_spread = max(future_price - spot_price)
    min_spread = min(future_price - spot_price)

    if current_spread >= max_spread:
        print('当前价差为 {} 大于过去10天历史最大价差 {}, 买入现货卖出期
货'.format(current_spread, max_spread))

```

```

        buy_open(context.s1, 5)
        sell_open(context.s2, 5)

    if current_spread < min_spread:
        print('当前价差为 {} 小于过去10天历史最小价差 {}, 买入期货卖出出现
        货'.format(current_spread, min_spread))
        buy_open(context.s2, 5)
        sell_open(context.s1, 5)

```

优化器回测样例

对于回测中使用优化器的场景，DeepQuant 做了简单封装，用户无需传入时间参数，策略中的优化器 API 参数见：[portfolio_optimize](#)

```

__config__ = {
    'base': {
        'accounts': {
            'stock': 10000000,
        },
        'start_date': "20170101",
        'end_date': "20200101",
        'frequency': '1d',
    },
    "mod": {
        "optimizer2": {
            "enabled": True,
        },
        'sys_analyser': {
            'enabled': True,
            'benchmark': '000300.SH',
        },
    },
}

def rebalance(context, bar_dict):
    cons = [
        wildcardIndustryConstraint(lower_limit=-0.01, upper_limit=0.1,
        relative=True,
                                classification=IndustryClassification.ZX,
        hard=False),
        wildcardStyleConstraint(lower_limit=-0.3, upper_limit=0.3, relative=True,
        hard=False)
    ]
    pool = [s for s in index_components('000906.SH') if not is_suspended(s)]
    s = portfolio_optimize(pool, cons=cons, benchmark='000300.SH')
    s = s[s > 0.0001]

    for order_book_id, position in context.stock_account.positions.items():
        if order_book_id not in s:
            order_target_value(order_book_id, 0)

    s = s.sort_values()
    portfolio_value = context.portfolio.total_value

    for order_book_id, weight in s.items():
        order_target_value(order_book_id, portfolio_value * weight)

```

```
def init(context):
    scheduler.run_monthly(rebalance, 1)
```

根据本地持仓权重运行回测范例

这里的样例与前面的精简版相比考虑了更复杂的场景，例如若调仓当天因为风控等原因发单失败，第二个交易日会继续发单，仅供用户参考。

```
import pandas
import numpy
from deepquant.quest.apis import *

__config__ = {
    "base": {
        "start_date": "20191201",
        "end_date": "20200930",
        "accounts": {
            "stock": 100000000,
        },
    },
}

def read_tables_df():
    # need pandas version 0.21.0+
    # need xlrd
    d_type = {'NAME': numpy.str, 'TARGET_WEIGHT': numpy.float, 'TICKER':
numpy.str, 'TRADE_DT': numpy.int}
    columns_name = ["TRADE_DT", "TICKER", "NAME", "TARGET_WEIGHT"]
    df = pandas.read_excel(r'调仓权重样例.xlsx', dtype=d_type)
    if not df.columns.isin(d_type.keys()).all():
        raise TypeError("xlsx文件格式必须有{}四列".format(list(d_type.keys())))
    for date, weight_data in df.groupby("TRADE_DT"):
        if round(weight_data["TARGET_WEIGHT"].sum(), 6) > 1:
            raise ValueError("权重之和出错，请检查{}日的权重".format(date))
    # 转换为order_book_id
    df['TICKER'] = df['TICKER'].apply(lambda x: yhdatac.id_convert(x) if ".OF"
not in x else x)
    return df

def on_order_failure(context, event):
    # 拒单时，未成功下单的标的放入第二天下单队列中
    order_book_id = event.order.order_book_id
    context.next_target_queue.append(order_book_id)

# 在这个方法中编写任何的初始化逻辑。context对象将会在你的算法策略的任何方法之间做传递。
def init(context):
    import rqalpha
    import rqalpha_mod_fund
    df = read_tables_df() # 调仓权重文件
    context.target_weight = df
```

```

context.adjust_days = set(context.target_weight.TRADE_DT.to_list()) # 需要调
仓的日期
context.target_queue = [] # 当日需要调仓标的队列
context.next_target_queue = [] # 次日需要调仓标的队列
context.current_target_table = dict() # 当前持仓权重比例
subscribe_event(EVENT.ORDER_CREATION_REJECT, on_order_failure)
subscribe_event(EVENT.ORDER_UNSOLICITED_UPDATE, on_order_failure)

# before_trading此函数会在每天策略交易开始前被调用，当天只会被调用一次
def before_trading(context):
    def dt_2_int_dt(dt):
        return dt.year * 10000 + dt.month * 100 + dt.day

    dt = dt_2_int_dt(context.now)
    if dt in context.adjust_days:
        today_df = context.target_weight[context.target_weight.TRADE_DT ==
dt].set_index("TICKER").sort_values(
            "TARGET_WEIGHT")
        context.target_queue = today_df.index.to_list() # 更新需要调仓的队列
        context.current_target_table = today_df["TARGET_WEIGHT"].to_dict()
        context.next_target_queue.clear()
        # 非目标持仓 需要清空
        for i in context.portfolio.positions.keys():
            if i not in context.target_queue:
                # 非目标权重持仓 需要清空
                context.target_queue.insert(0, i)
            else:
                # 当前持仓权重大于目标持仓权重 需要优先卖出获得资金
                equity = context.portfolio.positions[i].long.equity +
context.portfolio.positions[i].short.equity
                total_value =
context.portfolio.accounts[instruments(i).account_type].total_value
                current_percent = equity / total_value
                if current_percent > context.current_target_table[i]:
                    context.target_queue.remove(i)
                    context.target_queue.insert(0, i)

# 你选择的证券的数据更新将会触发此段逻辑，例如日或分钟历史数据切片或者是实时数据切片更新
def handle_bar(context, bar_dict):
    if context.target_queue:
        for _ticker in context.target_queue:
            _target_weight = context.current_target_table.get(_ticker, 0)
            o = order_target_percent(_ticker, round(_target_weight, 6))
            if o is None:
                logger.info("{}下单失败，该标将于次日下单".format(_ticker))
                context.next_target_queue.append(_ticker)
            else:
                logger.info("{}下单成功，现下占比{}%".format(_ticker,
round(_target_weight, 6) * 100))
                # 下单完成 下单失败的在队列context.next_target_queue中
                context.target_queue.clear()

# after_trading函数会在每天交易结束后被调用，当天只会被调用一次

```



```
def after_trading(context):
    if context.next_target_queue:
        context.target_queue += context.next_target_queue
        context.next_target_queue.clear()
    if context.target_queue:
        logger.info("未完成调仓的标的:{}".format(context.target_queue))

if __name__ == '__main__':
    from deepquant.quest.alpha import run_func

    run_func(init=init, before_trading=before_trading,
after_trading=after_trading, handle_bar=handle_bar,
            config=__config__)
```

更新履历

DeepQuant 版本	发布时间	新功能及改善	Bug 修复
0.5.68	2025-4-18	1. DeepQuant 带有回测功能的首版本发布	