Quantopian 日本語翻訳プロジェクト

Tokyo Quantopian User Group

2020 年 07 月 18 日

Contents:

第 1 章 Getting Started 3
- 1.1 取引アルゴリズムとは？ . . . 3
- 1.2 何から始めればいいですか？ . . . 3
- 1.3 データを探す . . . 5
- 1.4 Pipeline API . . . 8
- 1.5 戦略定義 . . . 11
- 1.6 戦略分析 . . . 12
- 1.7 Algorithm API と 主な関数 . . . 16
- 1.8 アルゴリズムにおけるデータ処理 . . . 18
- 1.9 ポートフォリオマネジメント . . . 21
- 1.10 リスクマネジメント . . . 23
- 1.11 バックテスト で分析する . . . 27

第 2 章 Pipeline 31
- 2.1 なぜ Pipeline なのか？ . . . 31
- 2.2 Research と IDE . . . 32
- 2.3 計算処理 . . . 32
- 2.4 データセット . . . 33
- 2.5 パイプラインを作成する . . . 33
- 2.6 パイプラインを実行する . . . 34
- 2.7 ファクター . . . 34
- 2.8 ファクターの作成 . . . 35
- 2.9 ファクターをパイプラインに追加する . . . 35
- 2.10 Latest . . . 36
- 2.11 デフォルト入力 . . . 37
- 2.12 ファクターの結合 . . . 37
- 2.13 フィルタ . . . 39
- 2.14 比較演算子 . . . 39
- 2.15 **ファクター / クラシファイア** メソッド . . . 39
- 2.16 売買代金（Dollar Volume）フィルタ . . . 40
- 2.17 スクリーニング . . . 41
- 2.18 フィルタの反転 . . . 42

2.19 フィルタの結合 . . . 42
2.20 マスキング . . . 44
2.21 ファクターに対するマスキング . . . 44
2.22 フィルタに対するマスキング . . . 45
2.23 クラシファイア . . . 46
2.24 クラシファイアからフィルタを作る . . . 47
2.25 分位数 . . . 47
2.26 データセット（Dataset）と連結列（BoundColumns） . . . 48
2.27 dtypes . . . 49
2.28 価格データ（Pricing Data） . . . 49
2.29 財務データ（Fundamental Data） . . . 49
2.30 パートナーデータ（Partner Data） . . . 49
2.31 カスタムファクター . . . 50
2.32 全部のせ . . . 54
2.33 パイプライン作成 . . . 55
2.34 IDE への移行 . . . 58
2.35 パイプラインへのアタッチ . . . 59
2.36 パイプライン出力 . . . 59
2.37 Research 環境で作成したパイプラインの使用法 . . . 60

第 3 章 Lectures 67

第 4 章 appendix 69
4.1 用語集 . . . 69

第 5 章 Indices and tables 71

索引 73

警告: このドキュメントは下書きです

重要: 免責事項（Disclaimer）

東京 Quantopian ユーザーグループは、掲載している情報の正確性について万全を期しておりますが、その内容について保証するものではありません。当該和訳は、英文を翻訳したものですので、和訳はあくまでも便宜的なものとして利用し、適宜、英文の原文を参照していただくようお願いします。（英文の原文：https://www.quantopian.com/docs/index）

東京 Quantopian ユーザーグループは、間違い、情報の欠落、あるいは、掲載されている情報の使用に起因して生じる結果に対して一切の責任を負わないものとします。掲載されている全ての情報は、その時点の情報が掲載されており、完全性、正確性、時間の経過、あるいは、情報の使用に起因して生じる結果について一切の責任を負わないものとします。また、あらゆる種類の保証、それが明示されているか示唆されているかにかかわらず、また業務遂行、商品性、あるいは特定の目的への適合性への保証、また、これらに限定されない保証も含め、いかなることも保証するものではありません。

第 1 章

Getting Started

重要: 2020/06/07 更新。本翻訳は、２０２０年５月以前に公開されていた **旧 Tutorial １ Getting Started** です。翻訳作業中に原作が大きく更新され、この翻訳は原作側には存在しません。しかしながら、記載されている内容やスクリプトは現在も使用可能ですので、日本語翻訳サイトには引き続き掲載します。

Quantopian へようこそ。この入門チュートリアルでは、Quantopian でのクオンツトレーディング戦略の研究と開発について説明します。このチュートリアルでは、Quantopian API の基本的な機能を多く取り上げており、Quantopian を初めて利用する方を対象にしています。チュートリアルを始めるために必要なのは、基本的な Python のプログラミングスキルだけです。

1.1 取引アルゴリズムとは？

取引アルゴリズムとは、コンピューターでクオンツトレーディング戦略を実現し、取引対象の資産を売買するためのルールを実装したプログラムのことです。一般に、取引アルゴリズムは、過去のデータについて数学的、統計的に分析して構築したモデルに基づいて作成され、取引の意思決定を行います。

1.2 何から始めればいいですか？

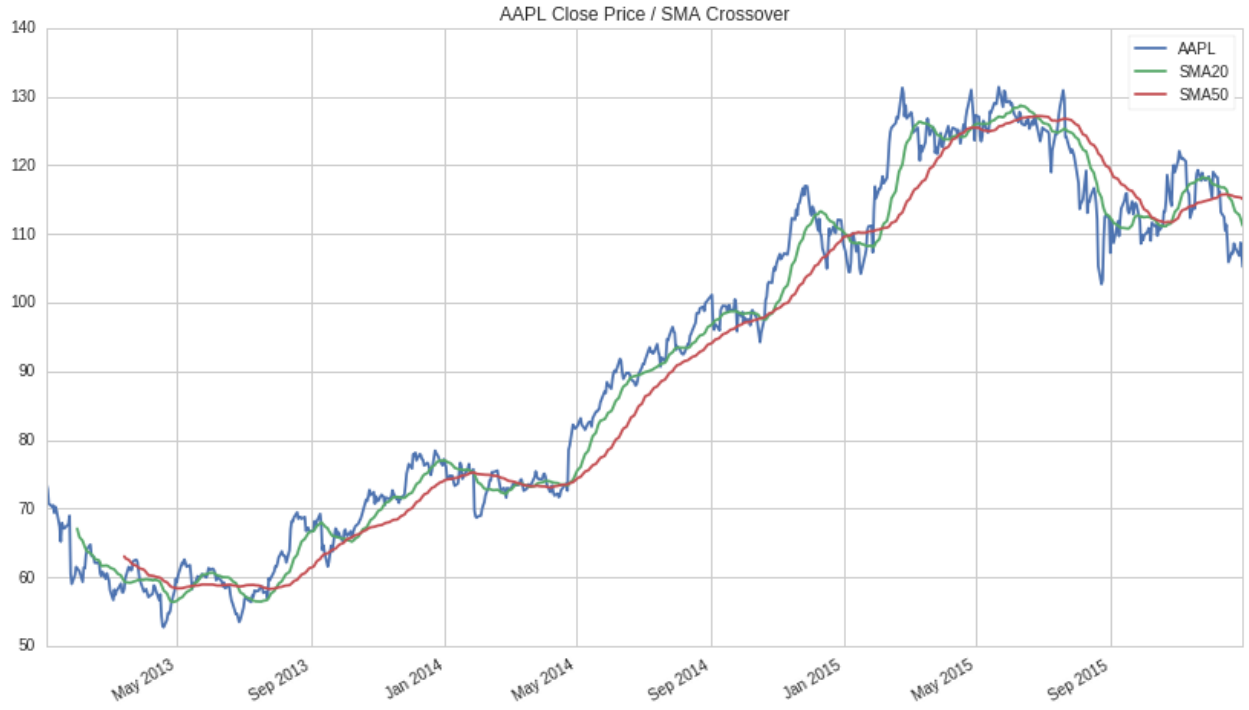
取引アルゴリズムを作成するためには、まず、戦略のベースとなる経済的または統計的な関係を見つけなくてはなりません。Quantopian の Research 環境では、そのために必要な過去のデータセットが提供されており、それを利用して分析を行うことができます。

Research は Jupyter Notebook 環境で提供されており、Python のコードを ' セル' と呼ばれる場所で実行することができます。

例えば、以下のコードは、Apple Inc. (AAPL) の毎日の終値と 20 日と 50 日移動平均線をプロットしています。

```
# Research 環境用関数
from quantopian.research import prices, symbols

# Pandas library: https://pandas.pydata.org/
import pandas as pd

# AAPL の過去の価格データを取得する
aapl_close = prices(
    assets=symbols('AAPL'),
    start='2013-01-01',
    end='2016-01-01',
)

# AAPL の価格データより 20 日と 50 日の移動平均を算出する
aapl_sma20 = aapl_close.rolling(20).mean()
aapl_sma50 = aapl_close.rolling(50).mean()

# 結果を結合して pandas の DataFrame に入れ、描画する
pd.DataFrame({
    'AAPL': aapl_close,
    'SMA20': aapl_sma20,
    'SMA50': aapl_sma50
}).plot(
    title='AAPL Close Price / SMA Crossover'
);
```

上記のコードは、Research 環境で新しいノートブックを開いて、セルにコピーアンドペーストすることで利用できます。そして、このコードのあるセルを選択して Shift+Enter キーを押せばセルを実行でき、次のような出力がなされます。



では早速、Research 環境を使って Quantopian のデータセットを触ってみましょう。次のレッスンでは、取引戦略を定義し、過去のデータに基づいてリターンを効果的に予測できるかどうかを検証してみます。さらに、その結果をもとに、インタラクティブ開発環境（IDE）で取引アルゴリズムを開発し、テストを行ってみましょう。

レッスン 2 から 4 は、Research 環境で行います。

Research 環境を利用するときには、新しいノートブックを作成 して下さい。

1.3 データを探す

Research は、価格、出来高、およびリターンを照会するためのユーティリティ関数を提供します。データは、2002 年から今日現在までの 8000 株以上の米国株式データです。関数は、資産（または資産のリスト）、開始日、終了日を引数として受け取り、日付を index に持つ、pandas の Series （もしくは DataFrame) を返します。

ここで、期間を指定して、AAPL のリターンを `returns` 関数を使って照会してみましょう。

```
# Research 環境用関数
from quantopian.research import returns, symbols

# 時間範囲を指定
period_start = '2014-01-01'
period_end = '2014-12-31'

# 上記の時間範囲で、AAPL のリターンデータを照会する
aapl_returns = returns(
```

(次のページに続く)

(前のページからの続き)

```
    assets=symbols('AAPL'),
    start=period_start,
    end=period_end,
)

# 最初の 10 行のみ表示
aapl_returns.head(10)
```

```
2014-01-02 00:00:00+00:00   -0.014137
2014-01-03 00:00:00+00:00   -0.022027
2014-01-06 00:00:00+00:00    0.005376
2014-01-07 00:00:00+00:00   -0.007200
2014-01-08 00:00:00+00:00    0.006406
2014-01-09 00:00:00+00:00   -0.012861
2014-01-10 00:00:00+00:00   -0.006674
2014-01-13 00:00:00+00:00    0.005043
2014-01-14 00:00:00+00:00    0.020123
2014-01-15 00:00:00+00:00    0.020079
Freq: C, Name: Equity(24 [AAPL]), dtype: float64
```

1.3.1 さまざまなデータ

Quantopian には、価格や出来高のデータだけでなく、企業のファンダメンタルズやセンチメント分析、マクロ経済指標など様々なデータセットが用意されています。データセットは、全部で 50 以上用意されており、詳細については、Quantopian の Data Reference で確認できます。

このチュートリアルでは、センチメントデータを使って株式を選び、取引を評価するところまで扱います。今回は、センチメントデータとして、PsychSignal の StockTwits Trader Mood データセットを使います。この PsychSignal のデータセットは、Stocktwits という株式専門 SNS に投稿されたメッセージの総体的なセンチメントに基づいて、日次で銘柄ごとにブル（強気）とベア（弱気）のスコアを割り当てたものです。

ではまず、`stocktwits` データセットのメッセージの総量とセンチメントスコア（ブルからベアを引いたもの）の列を見ていきましょう。データを照会するには、Quantopian の Pipeline API を使います。Pipeline API は、今後 Research 環境でデータを照会したり分析したりする時に何度も使うことになります。詳しくは次のレッスンや、Pipeline 専用のチュートリアル で学ぶことができます。今のところ知っておく必要があるのは、以下のコードが data pipeline を使用して `stocktwits` を照会してデータを返し、AAPL の結果をプロットしているということだけです。

```
# Pipeline imports
from quantopian.research import run_pipeline
from quantopian.pipeline import Pipeline
from quantopian.pipeline.factors import Returns
from quantopian.pipeline.data.psychsignal import stocktwits
```

(次のページに続く)

(前のページからの続き)

```
# Pipeline 定義
def make_pipeline():

    returns = Returns(window_length=2)
    sentiment = stocktwits.bull_minus_bear.latest
    msg_volume = stocktwits.total_scanned_messages.latest

    return Pipeline(
        columns={
            'daily_returns': returns,
            'sentiment': sentiment,
            'msg_volume': msg_volume,
        },
    )

# Pipeline 実行
data_output = run_pipeline(
    make_pipeline(),
    start_date=period_start,
    end_date=period_end
)

# AAPL のデータだけを取得
aapl_output = data_output.xs(
    symbols('AAPL'),
    level=1
)

# 描画
aapl_output.plot(subplots=True);
```

データセットの中味を検討するときには、データセットと株価の動きを見比べて、何かパターンがないか探して見てください。そうして見つけたパターンが、取引ストラテジーの基礎になるかもしれません。上記の例では、株価の日々のリターン（収益）のスパイク（急激な変化）と `stocktwits` のメッセージ総量のスパイクが、いくつか同じタイミングで起きていることが見て取れますし、リターンのスパイクと AAPL のセンチメントスコアの方向がマッチしている様子もいくつか確認できます。こうして見ると、充分に使えそうな面白いデータセットのようですので、さらにしっかりと統計的に分析して、うまく行くかどうか検証してみましょう。

次のレッスンでは、Pipeline API について詳しく説明します。

1.4 Pipeline API

> **警告:** この Tutorial で使用している PsychSignal Trader Mood データは２０２０年５月で更新終了しました。ご注意下さい。データに関する詳しい情報は PsychSignal Trader Mood (DEPRECATED) を参照して下さい。

Pipeline API は、横断的に資産データ分析を行うための強力なツールです。これにより、複数のデータに対して一連の演算を行い、一度に大量の資産を分析することができます。一般的な Pipeline API の用途として、以下のようなものがあります。

- フィルタリングルールに基づいた資産の選択
- スコアリング関数に基づく資産のランク付け
- ポートフォリオの配分の計算

まず、Pipeline クラスをインポートして、空の pipeline を返す関数を作成します。関数のなかで pipeline を使う形にすれば、複雑な処理をすっきりと扱うことができます。そうしておけば、pipeline を使った仕組みを Research 環境から IDE に移す時にも、関数ごとまとめて扱えてるので便利です。

```
# Pipeline class
from quantopian.pipeline import Pipeline

def make_pipeline():
    # 空の Pipeline を作成し返す。
    return Pipeline()
```

pipeline からデータ出力を取り出すためには、まず、データセットに収録されているデータ項目と、そのデータに対して行いたい演算を指定します。具体的には、日々の終値を取り出す場合、`USEquityPricing` データセットにある close （終値） を用いて、下記のように記述します。

```
# Pipeline class と USEquityPricing dataset  を import
from quantopian.pipeline import Pipeline
from quantopian.pipeline.data import USEquityPricing

def make_pipeline():
    # 日々の最終価格を取得
    close_price = USEquityPricing.close.latest

    # 上記のデータを Pipeline に入れて返す
    return Pipeline(
        columns={
            'close_price': close_price,
        }
    )
```

Pipeline API には、計算機能が予め多数用意されており、移動平均など、データのなかで一定期間を切り出して処理をする演算機能なども利用できます。例えば、Psychsignal の `stocktwits` データセットで提供されている `bull_minus_bear` データについて、3 日移動平均を計算して出力するコードは以下のように定義できます。

```
# Pipeline と データセットをインポート
from quantopian.pipeline import Pipeline
from quantopian.pipeline.data import USEquityPricing
from quantopian.pipeline.data.psychsignal import stocktwits

# 移動平均を計算する関数をインポート
from quantopian.pipeline.factors import SimpleMovingAverage


def make_pipeline():
    # 日々の最終価格を取得
    close_price = USEquityPricing.close.latest

```

(次のページに続く)

(前のページからの続き)

```
    # bull_minus_bear スコアの 3 日移動平均を演算
    sentiment_score = SimpleMovingAverage(
        inputs=[stocktwits.bull_minus_bear],
        window_length=3,
    )

    # pipeline に、最終価格と、センチメントスコアを入れて、返す
    return Pipeline(
        columns={
            'close_price': close_price,
            'sentiment_score': sentiment_score,
        }
    )
```

1.4.1 評価対象となる資産を選ぶ

戦略を開発するうえで、どのような取引対象資産を選ぶかは重要です。つまり、取引対象として考えられる銘柄からなる資産セットを予め用意し、そのなかから銘柄を選んで取引することを考えます。この資産セットのことを、トレーディング・ユニバース（trading universe）と呼びます。

トレーディング・ユニバースは、できるだけ多くの資産が含まれているのが望ましいですが、一方で、不必要な資産については排除しておく必要もあります。例えば、流動性の低い銘柄や、取引困難な銘柄などは外しておきたいところです。そこで便利なのが、予めそのようなことを考慮して用意されている `QTradableStocksUS` ユニバースです。早速、pipeline のスクリーニングパラメータを使って、`QTradableStocksUS` を私達のトレーディング・ユニバースとして設定しましょう。

```
# Pipeline と　データセットをインポート
from quantopian.pipeline import Pipeline
from quantopian.pipeline.data import USEquityPricing
from quantopian.pipeline.data.psychsignal import stocktwits

# 移動平均を計算する関数をインポート
from quantopian.pipeline.factors import SimpleMovingAverage

# 組み込みトレーディング・ユニバースをインポート
from quantopian.pipeline.filters import QTradableStocksUS


def make_pipeline():
    # トレーディング・ユニバースへの参照を作成
    base_universe = QTradableStocksUS()

    # 日々の最終価格を取得
    close_price = USEquityPricing.close.latest
```

(次のページに続く)

(前のページからの続き)

```
    # bull_minus_bear スコアの 3 日移動平均を演算
    sentiment_score = SimpleMovingAverage(
        inputs=[stocktwits.bull_minus_bear],
        window_length=3,
    )

    # pipeline に、最終価格と、センチメントスコア、スクリーニングとして、トレーディング・ユニバースを入れて返す
    return Pipeline(
        columns={
            'close_price': close_price,
            'sentiment_score': sentiment_score,
        },
        screen=base_universe
    )
```

これで pipeline の定義は完了しました。次に、 run_pipeline を使い、期間を指定して pipeline を実行してみましょう。結果は pandas の DataFrame で出力され、そのインデックスが日付と資産名、列は pipeline で定義したカラムとなります。

```
# run_pipeline をインポート
from quantopian.research import run_pipeline

# start_date と end_date を指定してm make_pipeline 関数を実行して pipeline を実行。
pipeline_output = run_pipeline(
    make_pipeline(),
    start_date='2013-01-01',
    end_date='2013-12-31'
)

# 最初の 10 行を表示
pipeline_output.tail(10)
```

次のレッスンでは、アルゴリズムが取引銘柄を選ぶ戦略を構築し、ファクター分析ツールを使って、過去のデータに対する戦略の予測力の評価をします。

1.5 戦略定義

Quantopian のデータへのアクセスと操作の方法を学んだところで、株式のロングショート戦略を行うデータ取得の仕組みを pipeline で構築してみましょう。株式のロングショート戦略とは、一般に、対象銘柄の価格の動き方を比較し、今後最も上昇すると思われる銘柄（ロング) と、下落すると思われる銘柄（ショート ）を組み合わせて利益を狙う戦略のことを言います。

株式のロングショート戦略では、値上がりした銘柄の価格変化と値下がりした銘柄の価格変化の差（スプレッド）が利益になります。つまり、この戦略は、株式の将来的な値動きの大きさについて、うまく順位付けができるとい

うことを前提にしています。このチュートリアルでは、簡単な順位付け方法を使ってみます。

今回の戦略: センチメントスコアの 3 日移動平均を取得し、銘柄別のセンチメントの状態の高さ、あるいは低さが、今後の値動きの高低に影響するものとして売買の判断をします。

1.6 戦略分析

上記の戦略は、 SimpleMovingAverage 関数と、 stocktwits の bull_minus_bear データを使って定義出来ます。先述した pipeline のレッスンと同じように書くことができます。

```
# Pipeline インポート
from quantopian.pipeline import Pipeline
from quantopian.pipeline.data.psychsignal import stocktwits
from quantopian.pipeline.factors import SimpleMovingAverage
from quantopian.pipeline.filters import QTradableStocksUS


# Pipeline 定義
def  make_pipeline():

    base_universe = QTradableStocksUS()

    sentiment_score = SimpleMovingAverage(
        inputs=[stocktwits.bull_minus_bear],
        window_length=3,
    )

    return Pipeline(
        columns={
            'sentiment_score': sentiment_score,
        },
        screen=base_universe
    )
```

まずここでは、説明を簡単にするために、 sentiment_score を使って順位した各銘柄のなかで、上位および下位のそれぞれ 350 銘柄だけについて分析します。具体的には、pipeline フィルターというものを利用し、sentiment_score の出力を、 top と bottom メソッドを使って上位と下位だけ取得するフィルターを作ります。そして、その結果を | オペレータ でつないで和集合を作れば、上位と下位の銘柄を集めたものをまとめられます。次に、そのなかから、私達のトレーディングユニバースにない銘柄を除くために、フィルターとユニバースを & オペレータでつなぎます。

```
# Pipeline インポート
from quantopian.pipeline import Pipeline
from quantopian.pipeline.data.psychsignal import stocktwits
from quantopian.pipeline.factors import SimpleMovingAverage
```

(次のページに続く)

(前のページからの続き)

```
from quantopian.pipeline.filters import QTradableStocksUS

# Pipeline 定義
def  make_pipeline():

    base_universe = QTradableStocksUS()

    sentiment_score = SimpleMovingAverage(
        inputs=[stocktwits.bull_minus_bear],
        window_length=3,
    )

    # センチメントスコアに基づいて上位下位 350 銘柄のみを取得するフィルターを作成
    top_bottom_scores = (
        sentiment_score.top(350) | sentiment_score.bottom(350)
    )

    return Pipeline(
        columns={
            'sentiment_score': sentiment_score,
        },
        # 定義したフィルターとトレーディングユニバースのどちらにも入っている銘柄のみにスクリーニングする
        screen=(
            base_universe
            & top_bottom_scores
        )
    )
```

それでは、3 年間の pipeline を実行して、このあとの分析で使う情報を取り出してみましょう。これには 1 分ほどかかります。

```
# run_pipeline インポート
from quantopian.research import run_pipeline

# 評価する期間を指定
period_start = '2013-01-01'
period_end = '2016-01-01'

# 指定期間で pipeline 実行
pipeline_output = run_pipeline(
    make_pipeline(),
    start_date=period_start,
    end_date=period_end
)
```

この先の分析には、各銘柄のセンチメントデータに加えて、同じ期間の価格のデータも必要です。pipeline が出力する DataFrame の index は銘柄のリストになっていますので、そのリストを `prices` に渡せば価格データを得る

ことが出来ます。

```
# prices 関数をインポート
from quantopian.research import prices

# pipeline が出力した dataframe の index から銘柄リストを取得し、 unique 関数を使って、重複しないリストを取得します。
asset_list = pipeline_output.index.levels[1].unique()

# 銘柄リストに入っている銘柄全てに対して、指定期間の価格を取得します。
asset_prices = prices(
    asset_list,
    start=period_start,
    end=period_end
)
```

次に、Quantopian が作ったオープンソースの分析ツールである、 Alphalens を使って、私達の戦略の品質を検証してみましょう。まず、 get_clean_factor_and_forward_returns 関数を使って、ファクターデータと価格データを組み合わせます。この関数は、ファクターデータを順位付けて分類し、数日間にわたり銘柄を保有したら、収益がいくらになるかを（複数の評価基準日に対して）計算します。ここでは、ファクターデータを上位と下位の半分ずつにわけ、評価基準日から 1 日、5 日、10 日後の収益結果をみます。

```
# Alphalens インポート
import alphalens as al

# センチメントスコアに基づいて、quantile に指定された分位数にわける
factor_data = al.utils.get_clean_factor_and_forward_returns(
    factor=pipeline_output['sentiment_score'],
    prices=asset_prices,
    quantiles=2,
    periods=(1,5,10),
)

# 上から 5 行を表示
factor_data.head(5)
```

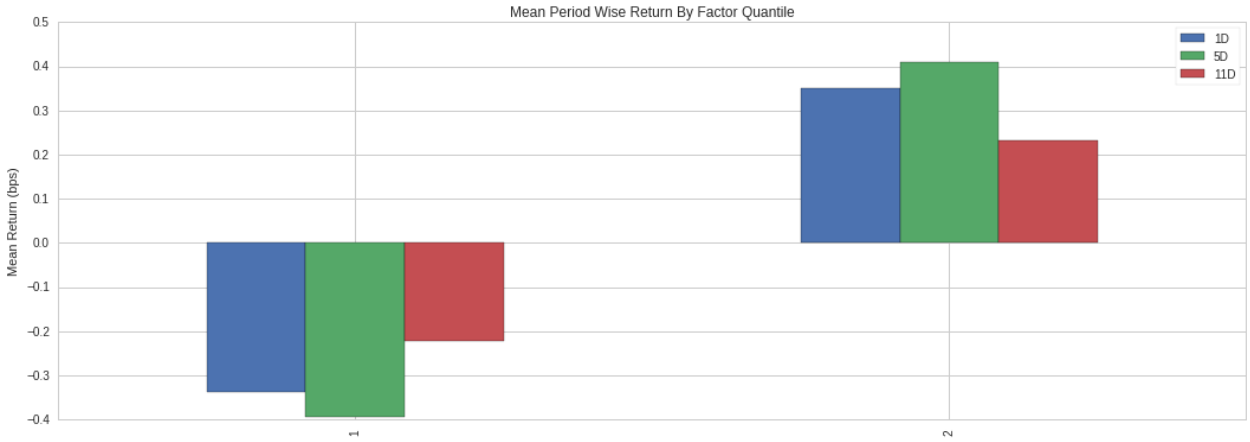
これらの出力結果を、 Alphalens に渡せば、分析や描画を行なうことができます。ではまず、指定した全期間における、平均の収益を四分位ごとに見てみましょう。私達の戦略はロングショート戦略なので、ショートする下位の四分位の収益がネガティブ、ロングする上位の四分位の収益がポジティブであればうまく行くということになります。

```
# ファクターの四分位別に、平均を算出
mean_return_by_q, std_err_by_q = al.performance.mean_return_by_quantile(factor_data)

# 四分位と保有ごとに、平均を描画
al.plotting.plot_quantile_returns_bar(
    mean_return_by_q.apply(
```

(次のページに続く)

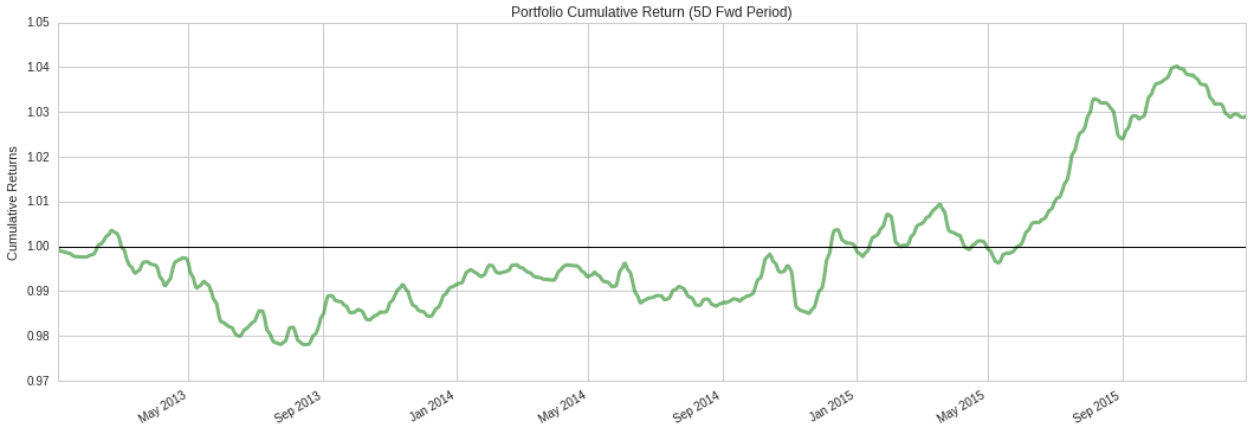
(前のページからの続き)

```
        al.utils.rate_of_return,
        axis=0,
        args=('1D',)
    )
);
```



次に、5 日間保有した場合の累積収益を見てみましょう。ただし今回は、ロングとショートのポートフォリオにファクターでウェイトをかけます。

```
import pandas as pd
# ファクターでウェイト付けしたロングショートのポートフォリオを収益を算出
ls_factor_returns = al.performance.factor_returns(factor_data)

# 5 日間保有した場合の累積収益を描画
al.plotting.plot_cumulative_returns(ls_factor_returns['5D'], '5D', freq=pd.tseries.
↪offsets.BDay());
```



このチャートを見ると、大きなドローダウンの期間がありますね。しかも、この分析では、取引コストやマーケットインパクトをまだ考慮に入れていません。ですので、これはあまり有望な戦略とは言えないようです。より良い戦略にするためには、さらに深い分析を Alphalens で行い、色んなアイデアで試行錯誤していく必要があるでしょ

う。ですが、これはチュートリアルですので、この戦略のままで進めて行きたいと思います。

さて、ここまでのところで、取引戦略を実装し、その性能の検証をしてみました。次のチュートリアルレッスンでは、バックテストの機能を使って、この株式ロングショート戦略のパフォーマンスの検証を行います。バックテストでは IDE で Algorithm API を使っていきます。

前回のレッスンでは、Research の機能を使って、ポートフォリオに組み入れる銘柄を選択し、その資産のアルファスコアを計算するデータパイプラインを作成しました。残りのレッスンでは、Quantopian の Interactive Development Environment (IDE) を使って、取引アルゴリズムを作成します。IDE でも、データパイプラインを使い、アルファスコアを算出して、ポートフォリオを構築していきます。そして、過去データを用いてシミュレーションを行い、より現実的な条件下でのアルゴリズムのパフォーマンスを分析します。このようなヒストリカルシミュレーションは一般的に、"バックテスト"として知られています。

1.7 Algorithm API と 主な関数

次のステップでは、 Quantopian の Algorithm API を使用しながら、取引アルゴリズムの基本構造を構築しましょう。Algorithm API は、注文のスケジュールや実行を簡単に行う事ができる機能や、アルゴリズムのパラメータを初期化したり管理したりする機能を提供しています。ここでは、アルゴリズムで使用する主要な関数をいくつか紹介します。

initialize(context)

アルゴリズムの実行を開始するときに 、 `initialize` という関数は真っ先に一度だけ呼ばれます。その時必ず引数として `context` を渡さなくてはいけません。IDE で用いるパラメータの初期化や、一回限りで行なう初期化や設定のロジックは、すべてこの `initialize` で行う必要があります。

`context` は Python の dictionary を拡張したもので、シミュレーションの過程において、その状況を格納するために使われ、アルゴリズムのどの時点においても参照することができるようにしてあります。このため、IDE での関数呼び出しで共有させたい変数は、グローバル変数を用いるのではなく `context` の中に属性として保存しておきましょう。 `context` に保存する属性はドット（ 例： `context.some_attribute` ）でアクセスして、その内容の設定や参照をすることができます。

before_trading_start(context, data)

`before_trading_start` という関数は、シミュレーションの中で、毎日マーケットが開く前に 1 回だけ呼び出され、引数として `context` と `data` を必要とします。 `context` は `initialize` で使った辞書と同じものです。 `data` は複数の API 関数を格納しているオブジェクトです。それらの関数を使って、任意の資産の現在または過去の価格や数量のデータを調べることができます。`before_trading_start` 関数は、 pipeline の出力結果を取得する場所でもあります。また、その結果として得られたデータをポートフォリオ構築に使用するための前処理も行います。これらについては次のレッスンで説明します。

schedule_function(func, day_rule, time_rule)

Quantopian のアルゴリズムは ニューヨーク証券取引所の取引カレンダー に沿って、午前 9 時 30 分から午後 4 時

の間株式の取引を行います。schedule_function を使うと、指定した日時にユーザーがカスタマイズした関数を実行することができます。例えば、毎週最初の営業日のマーケットオープン時にポートフォリオのリバランスを行う関数は、以下のようにスケジュールすることができます。

```
schedule_function(
    rebalance, ## ユーザーがカスタマイズした関数名
    date_rule=date_rules.week_start(),
    time_rule=time_rules.market_open()
)
```

スケジューリング関数は initialize 関数の中で呼び出す必要があり、決められたスケジュールに従って呼び出される関数（通常ユーザーが作成するもの）と呼び出すタイミングを引数として取ります。この呼び出される関数は、必ず context と data を引数に取るように定義されている必要があります。

利用可能な date_rules と time_rules を使ってどのようにスケジューリングすることができるかは documentation を参照してください。

次に、取引アルゴリズムの基本構造を作ってみましょう。下記のアルゴリズムは、シミュレーションで経過した日数を記録し、日付と時間に応じて異なるメッセージをログに出力しているだけです。この後のレッスンで、 データパイプライン と、取引ロジックをこの基本構造に追加していきます。"Clone" ボタンをクリックすると、IDE にコピーが作成され、アルゴリズムを実行することが出来ます。

重要: Lesson 1 の冒頭で説明したとおり、原作側が Tutorial 1 を大幅に改定した為、このスクリプトをクローンすることは出来ません。よって、 Quantopian にログイン後、Research > Algorithm と進み、 New Algorithm ボタンをおして IDE を開きスクリプトをコピー&ペーストしてお使い下さい。

IDE に入ったら、"Build Algorithm" (左上） または "Run Full Backtest" (右上）をクリックしてバックテストを実行します。

```
# Algorithm API をインポート
import quantopian.algorithm as algo


def initialize(context):
    # アルゴリズムパラメータを初期化
    context.day_count = 0
    context.daily_message = "Day {}."
    context.weekly_message = "Time to place some trades!"

    # rebalance 関数をスケジューリング
    algo.schedule_function(
        rebalance,
        date_rule=algo.date_rules.week_start(),
        time_rule=algo.time_rules.market_open()
```

(次のページに続く)

(前のページからの続き)

```
    )


def before_trading_start(context, data):
    # 毎日、トレード時間が始まる前に必ず実行する
    context.day_count += 1
    log.info(context.daily_message, context.day_count)


def rebalance(context, data):
    # リバランスのロジックを実行する
    log.info(context.weekly_message)
```

取引アルゴリズムの基本的な構造ができたので、前回のレッスンで作成したデータパイプラインをアルゴリズムに追加してみましょう。

1.8 アルゴリズムにおけるデータ処理

次のステップでは、Research で構築したデータパイプラインをアルゴリズムに統合します。Research との重要な違いは、バックテストの間、シミュレーションの進行に合わせて pipeline が必ず毎日実行されることです。したがって、 `start_date` と `end_date` を記述する必要はありません。

アルゴリズムの中でデータパイプラインを使うためには、まずアルゴリズムの `initialize` 関数の中に データパイプライン への参照を追加します。これは `attach_pipeline` メソッドを使って行います。これには 2 つの引数があり、一つめは、 `Pipeline` オブジェクトへの参照（これは、 `make_pipeline` を使って作成します。)、もう一つは、その参照に名前を付けるための任意の **文字列** です。

```
# Algorithm API  をインポート
import quantopian.algorithm as algo


def initialize(context):
    # アルゴリズムを pipeline に 取り付ける
    algo.attach_pipeline(
        make_pipeline(),
        'data_pipe'
    )

    # rebalance 関数をスケジュールする
    algo.schedule_function(
        rebalance,
        date_rule=algo.date_rules.week_start(),
        time_rule=algo.time_rules.market_open()
    )
```

(次のページに続く)

(前のページからの続き)

```
def before_trading_start(context, data):
    pass


def rebalance(context, data):
    pass
```

冒頭で述べたように、パイプラインは、バックテスト期間内の毎日、マーケットが開く前にデータを処理し、出力を生成します。この出力は、 before_trading_start 関数のなかで、 pipeline_output 関数を使って取得することができます。なお、その出力は、 pandas の DataFrame 型のデータになっています。それでは、毎日生成されるパイプラインの出力のうち、最初の 10 行だけ、 rebalance 関数のなかでログに書き出してみましょう。

```
# Algorithm API をインポート
import quantopian.algorithm as algo


def initialize(context):
    # algorithm に pipeline を取り付ける
    algo.attach_pipeline(
        make_pipeline(),
        'data_pipe'
    )

    # rebalance 関数をスケジュールする
    algo.schedule_function(
        rebalance,
        date_rule=algo.date_rules.week_start(),
        time_rule=algo.time_rules.market_open()
    )


def before_trading_start(context, data):
    # pipeline の出力結果を取得。
    # それを、context.pipeline_data 変数へ格納する。
    context.pipeline_data = algo.pipeline_output(
        'data_pipe'
    )


def rebalance(context, data):
    # pipeline の出力結果の最初の 10 行だけをログに出力
    log.info(context.pipeline_data.head(10))
```

では、Research で作った make_pipeline 関数をアルゴリズムに追加してみましょう。このアルゴリズムでは、あらかじめ用意した取引対象銘柄に入っている銘柄のうち、センチメントスコアが付与されている全ての銘柄を

（銘柄数の制限なしで）対象とします。そのため、 sentiment_score の notnull メソッドを使って、センチメントスコアが付与されている銘柄を取り出し、それを & 演算子で取引対象銘柄と組み合わせて、取引すべき銘柄を絞り込みます。

```
# Algorithm API インポート
import quantopian.algorithm as algo

# Pipeline インポート
from quantopian.pipeline import Pipeline
from quantopian.pipeline.data.psychsignal import stocktwits
from quantopian.pipeline.factors import SimpleMovingAverage
from quantopian.pipeline.filters import QTradableStocksUS


def initialize(context):
    # algorithm に pipeline を取り付ける
    algo.attach_pipeline(
        make_pipeline(),
        'data_pipe'
    )

    # rebalance 関数をスケジュールする
    algo.schedule_function(
        rebalance,
        date_rule=algo.date_rules.week_start(),
        time_rule=algo.time_rules.market_open()
    )


def before_trading_start(context, data):
    # pipeline の出力結果を取得。
    # それを、context.pipeline_data 変数へ格納する。
    context.pipeline_data = algo.pipeline_output('data_pipe')


def rebalance(context, data):
    # pipeline の出力結果の最初の 10 行だけをログに出力
    log.info(context.pipeline_data.head(10))


# Pipeline definition
def make_pipeline():

    base_universe = QTradableStocksUS()

    sentiment_score = SimpleMovingAverage(
        inputs=[stocktwits.bull_minus_bear],
        window_length=3,
    )
```

(次のページに続く)

(前のページからの続き)

```
    return Pipeline(
        columns={
            'sentiment_score': sentiment_score,
        },
        screen=(
            base_universe
            & sentiment_score.notnull()
        )
    )
```

ここまでのところで、バックテスト期間中の毎日、取引の対象となる銘柄候補を選び出し、それぞれについて、ポートフォリオ内の資産配分を決定するために使うアルファスコアを得られるようになりました。次のレッスンでは、データパイプラインによって得られたアルファスコアに基づいて、最適なポートフォリオを構築する方法を学びます。

1.9 ポートフォリオマネジメント

前回のレッスンでは、取引アルゴリズムに data pipeline を組み込みました。

次は、一般に、ポートフォリオ最適化 と呼ばれている取り組みを行います。アルゴリズムが対象資産を売買し、ポートフォリオを組み上げていくときには、制約条件やルールに従って、リターンを最大にすることを目指しますが、ここでは、pipeline によって生成したアルファスコアを活用して、リターンを最大化するために最適と見込まれるポートフォリオを作っていきます。

Quantopian の Optimize API を使うと、ユーザーの望む目的関数と制約条件に基づいて、 pipeline からの出力を変換して利用することができます。さらに、 `order_optimal_portfolio` を使うことで、ポートフォリオをターゲットポートフォリオに近づけていくための注文を行なうことができます。

最初のステップは、 目的関数を定義することです。ここでは `MaximizeAlpha` を使用し、アルファスコアに基づいて、資本（資金）を各対象資産に配分することを考えます。

```
# Optimize API module インポート
import quantopian.optimize as opt


def rebalance(context, data):
  # pipeline 出力から alpha を取り出す
  alpha = context.pipeline_data.sentiment_score

  if not alpha.empty:
      # MaximizeAlpha objective を作成
      objective = opt.MaximizeAlpha(alpha)
```

次に、ターゲットポートフォリオが満たすべき制約のリストを指定します。まず、いくつかの閾値を initialize 関数で定義し、それを context 変数に格納してみましょう。

```
# 制約パラメータ
context.max_leverage = 1.0
context.max_pos_size = 0.015
context.max_turnover = 0.95
```

上で定義した閾値を使って rebalance 関数で制約を指定しましょう。

```
# Optimize API module インポート
import quantopian.optimize as opt


def rebalance(context, data):
  # pipeline 出力から alpha を取り出す
  alpha = context.pipeline_data.sentiment_score

  if not alpha.empty:
      # MaximizeAlpha objective を作成
      objective = opt.MaximizeAlpha(alpha)

      # ポジションサイズの制約
      constrain_pos_size = opt.PositionConcentration.with_equal_bounds(
          -context.max_pos_size,
          context.max_pos_size
      )

      # ポートフォリオレバレッジの制約
      max_leverage = opt.MaxGrossExposure(context.max_leverage)

      # ロング（買い持ち）とショート（売り持ち）のサイズをだいたい同じに合わせる
      dollar_neutral = opt.DollarNeutral()

      # ポートフォリオの出来高の制約
      max_turnover = opt.MaxTurnover(context.max_turnover)
```

最後に、 目的関数と制約のリストを order_optimal_portfolio 関数に渡してターゲットポートフォリオを評価し、現在のポートフォリオを最適な状態にするために必要な注文を出します。

```
# Algorithm API インポート
import quantopian.algorithm as algo

# Optimize API インポート
import quantopian.optimize as opt


def rebalance(context, data):
```

(次のページに続く)

(前のページからの続き)

```
    # pipeline 出力から alpha を取り出す
    alpha = context.pipeline_data.sentiment_score

    if not alpha.empty:
        # MaximizeAlpha objective を作成
        objective = opt.MaximizeAlpha(alpha)

        # ポジションサイズの制約
        constrain_pos_size = opt.PositionConcentration.with_equal_bounds(
            -context.max_pos_size,
            context.max_pos_size
        )

        # ポートフォリオレバレッジの制約
        max_leverage = opt.MaxGrossExposure(context.max_leverage)

        # ロング（買い持ち）とショート（売り持ち）のサイズをだいたい同じに合わせる
        dollar_neutral = opt.DollarNeutral()

        # ポートフォリオの出来高の制約
        max_turnover = opt.MaxTurnover(context.max_turnover)

        # 目的関数と制約リストを使ってポートフォリオをリバランスする
        algo.order_optimal_portfolio(
            objective=objective,
            constraints=[
                constrain_pos_size,
                max_leverage,
                dollar_neutral,
                max_turnover,
            ]
        )
```

1.10 リスクマネジメント

ターゲットポートフォリオに制約条件を設定して、ポートフォリオを最適化していくのとあわせ、ここで、ポートフォリオのパフォーマンスに悪い影響を与えやすいリスク要因を避けるような制約条件も設定しておきたいと思います。例えば、 `stocktwits` のセンチメントデータは一時的な性質のものなので、センチメントスコアの急上昇を捉えて投資をするようなアルゴリズムを組むと、逆に急降下のリスクにさらされてしまう可能性があります。Quantopian の Risk Model を使用すれば、一般的なリスク要因に対するポートフォリオのエクスポージャーを管理できます。Risk Model では、資産にまつわる 16 種類のリスク要因に対して評価ができ、11 のセクターリスク要因と 5 つのスタイルリスク要因（短期的な反転を含む）に対応しています。

このリスク評価の情報は `risk_loading_pipeline` 関数を使えば簡単に取得できます。この関数は、Risk Model に定義された各リスク要因の結果をコラムに持つ、 data pipeline を返します。

risk data pipeline は、data pipeline と同じやり方で、識別名を付けてアルゴリズムに登録します。そうすれば `before_trading_start` 関数で risk data pipeline の出力を取得し、それを `context` に保存することができるようになります。

```
# Algorithm API インポート
import quantopian.algorithm as algo

# Risk API method インポート
from quantopian.pipeline.experimental import risk_loading_pipeline

def initialize(context):
    # 制約パラメータ
    context.max_leverage = 1.0
    context.max_pos_size = 0.015
    context.max_turnover = 0.95

    # data pipelines を取り付ける
    algo.attach_pipeline(
        make_pipeline(),
        'data_pipe'
    )
    algo.attach_pipeline(
        risk_loading_pipeline(),
        'risk_pipe'
    )

    # rebalance 関数をスケジュール
    algo.schedule_function(
        rebalance,
        algo.date_rules.week_start(),
        algo.time_rules.market_open(),
    )


def before_trading_start(context, data):
    # pipeline 出力を取得し、context に格納する。
    context.pipeline_data = algo.pipeline_output(
      'data_pipe'
    )

    context.risk_factor_betas = algo.pipeline_output(
      'risk_pipe'
    )
```

次に、ポートフォリオ最適化ロジックに `RiskModelExposure` 制約を追加します。この制約はリスクモデルによって生成されたデータを受け取り、リスクモデルに含まれるひとつひとつの要因に対して、ターゲットポートフォリオに対するエクスポージャーの制限を設定します。

```
# ターゲットポートフォリオのリスクエクスポージャーを制限する。
# デフォルト値は、セクターエクスポージャーの最大値は 0.2、スタイルエクスポージャーの最大値は 0.4
factor_risk_constraints = opt.experimental.RiskModelExposure(
    context.risk_factor_betas,
    version=opt.Newest
)
```

下記のコードが、私たちの戦略とポートフォリオ構築ロジックを記述したアルゴリズムです。このコードでバックテストすることができます。アルゴリズムを clone した後、IDE の右上にある「Run Full Backtest」をクリックして、完全なバックテストを実行してみましょう。

注釈: Quantopian にログイン後、本翻訳の原作ページ https://www.quantopian.com/tutorials/getting-started#lesson7 で、 Clone ボタンを押してコードをクローンして下さい。

```
# Algorithm API インポート
import quantopian.algorithm as algo

# Optimize API インポート
import quantopian.optimize as opt

# Pipeline  インポート
from quantopian.pipeline import Pipeline
from quantopian.pipeline.data.psychsignal import stocktwits
from quantopian.pipeline.factors import SimpleMovingAverage

# built-in universe と Risk API method インポート
from quantopian.pipeline.filters import QTradableStocksUS
from quantopian.pipeline.experimental import risk_loading_pipeline


def initialize(context):
    # 制約パラメータ
    context.max_leverage = 1.0
    context.max_pos_size = 0.015
    context.max_turnover = 0.95

    # data pipelines を取り付ける
    algo.attach_pipeline(
        make_pipeline(),
        'data_pipe'
    )
    algo.attach_pipeline(
        risk_loading_pipeline(),
        'risk_pipe'
    )
```

(次のページに続く)

(前のページからの続き)

```
    # rebalance 関数をスケジュール
    algo.schedule_function(
        rebalance,
        algo.date_rules.week_start(),
        algo.time_rules.market_open(),
    )


def before_trading_start(context, data):
    # pipeline 出力を取得し、context に格納する。
    context.pipeline_data = algo.pipeline_output('data_pipe')

    context.risk_factor_betas = algo.pipeline_output('risk_pipe')


# Pipeline definition
def make_pipeline():

    sentiment_score = SimpleMovingAverage(
        inputs=[stocktwits.bull_minus_bear],
        window_length=3,
        mask=QTradableStocksUS()
    )

    return Pipeline(
        columns={
            'sentiment_score': sentiment_score,
        },
        screen=sentiment_score.notnull()
    )


def rebalance(context, data):
    # pipeline 出力から alpha を取り出す
    alpha = context.pipeline_data.sentiment_score

    if not alpha.empty:
        # MaximizeAlpha objective 作成
        objective = opt.MaximizeAlpha(alpha)

        # ポジションサイズ制約
        constrain_pos_size = opt.PositionConcentration.with_equal_bounds(
            -context.max_pos_size,
            context.max_pos_size
        )

        # ターゲットポートフォリオレバレッジ制約
```

(次のページに続く)

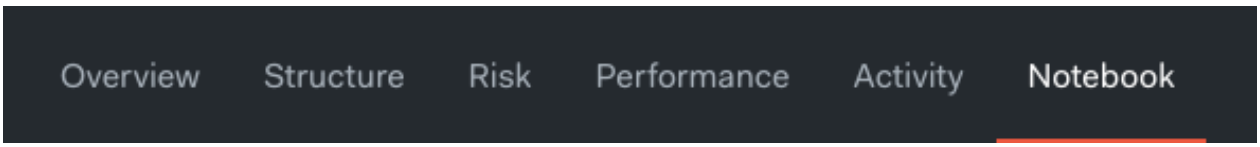
(前のページからの続き)

```
        max_leverage = opt.MaxGrossExposure(context.max_leverage)

        # ロング（買い持ち）とショート（売り持ち）のサイズをだいたい同じに合わせる
        dollar_neutral = opt.DollarNeutral()

        # ポートフォリオの出来高の制約
        max_turnover = opt.MaxTurnover(context.max_turnover)

        # ターゲットポートフォリオのリスクエクスポージャーを制限する。
        # デフォルト値は、セクターエクスポージャーの最大値は 0.2
        # スタイルエクスポージャーの最大値は 0.4
        factor_risk_constraints = opt.experimental.RiskModelExposure(
            context.risk_factor_betas,
            version=opt.Newest
        )

        # 目的関数と制約リストを使ってポートフォリオをリバランスする
        algo.order_optimal_portfolio(
            objective=objective,
            constraints=[
                constrain_pos_size,
                max_leverage,
                dollar_neutral,
                max_turnover,
                factor_risk_constraints,
            ]
        )
```

次のレッスンでは、バックテストの結果をより詳しく分析する方法を学びます。

1.11 バックテスト で分析する

バックテストの実行が終了したら、"Notebook"タブをクリックします。



そうすると、Research notebook が開き、セルの 1 つに以下のようなコードが自動で挿入されます。

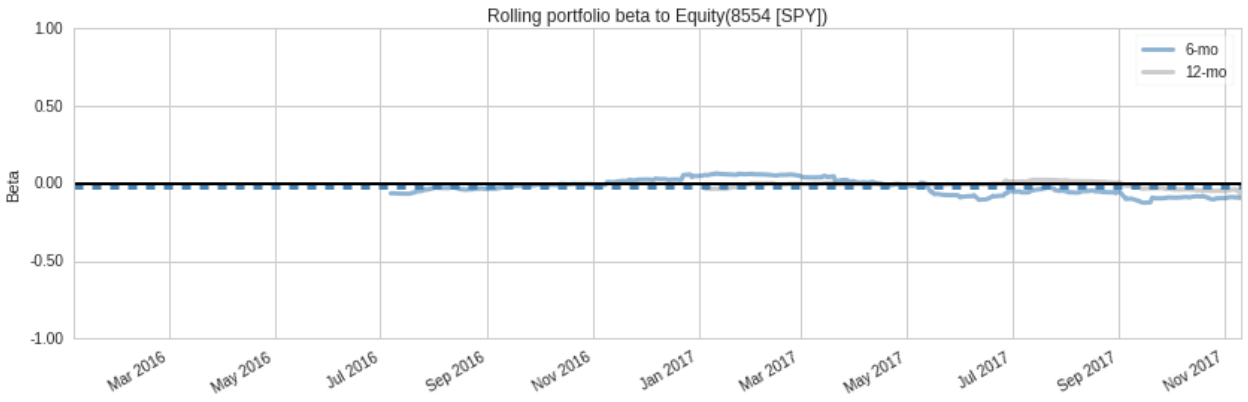
```
bt = get_backtest('5a4e4faec73c4e44f218170a')
bt.create_full_tear_sheet()
```

このセルを実行（Shift+Enter）すると、backtest で生成されたデータが research notebook に読み込まれ、それを使って Pyfolio の tear sheet が作成されます。

注釈: notebook に表示される英数字の文字列は、上記のものとは異なります。この文字列は、Quantopian 内でのバックテストを特定するための識別子です。full backtest を行った時の結果ページの URL にも、この文字列が使われています。full backtest とは、Algorithm IDE 機能の 1 つでストラテジーのパフォーマンスやリスク要因などを詳細に分析するツールです。詳しくは、 ドキュメント を参照してください。

Pyfolio は、Quantopian が開発したポートフォリオとリスク分析を行うオープンソースのツールです。

このツールには、アルゴリズムの動作やリスクエクスポージャーを時間の経過とともによりよく理解するために設計された、多くの可視化ツールが用意されています。例えば以下のプロットは、私達のポートフォリオがマーケットの影響をどのくらい受けているか、一定期間単位で時系列に表示したものです。私たちが株式ロング・ショート取引アルゴリズムを構築しようと思った理由の一つは、市場との相関関係を低く維持することでした。したがって、このプロットでは、backtesting の期間中ずっと、相関係数が概ねゼロ付近になっているのが望ましい姿ということになります。



tear sheet のもう 1 つの面白い点は、パフォーマンスがどの特性から生み出されたものかを表示するところです。以下のプロットは、リターンのうちどれだけが私達の戦略に起因するか、そしてそのうちどれだけが一般的なリスク要因に起因するかを説明するために、Quantopian の Risk Model を使っています。

上のプロットから、ポートフォリオのトータルリターンのほとんどが特定のリターンから来ていることがわかります。これは、アルゴリズムのパフォーマンスが一般的なリスク要因から来ていないことを示唆しています。それは

私達のアルゴリズムにとって良いことです。

これで Quantopian の入門チュートリアルが終了です。おめでとうございます。プラットフォームの API に慣れてきたところで、ご自身の戦略を研究・開発して、 contest に応募してみましょう。

アイデアが必要な場合は、 Lecture Series を試して下さい。金融工学（クオンツ）について色々と学ぶことができます。また、 community で他のメンバーが共有しているアイデアを見るのもよいでしょう。

第 2 章

Pipeline

Quantopian パイプラインのチュートリアルへようこそ。このチュートリアルでは pipeline API を紹介します。もしあなたが Quantopian に不慣れであれば Getting Started Tutorial から始め、少なくとも Python の動作知識を学習しておくことをお薦めします。このチュートリアルはいくつかのレッスンに分かれており、それぞれのレッスンで異なる Pipeline API の機能に触れていきます。レッスン 2 から 11 は Research 環境で動作します。レッスン 12 は IDE 環境で動作します。

2.1 なぜ Pipeline なのか？

多くの取引アルゴリズムは、以下のような構造を持っています：

1. 既知の（大きな）データセット内に存在する資産に対し、一定区間データに基づく N 個のスカラ値を計算する。
2. (1) の計算結果に基づき、取引可能な資産の集合を絞り込む。
3. (2) で絞り込みした資産の集合に対し、望ましい投資比率を計算する。
4. 現在のポートフォリオの投資比率が、(3) で計算した望ましい投資比率となるように発注する。

これらを頑健に実行するためには、いくつかの技術的困難が存在します。そこには、以下のようなものが含まれます。

- 大規模データセットに対する効率的な問い合わせ
- 大規模データセットに対する計算効率
- データ修正作業（株式分割や配当金）
- 資産の上場廃止作業

パイプラインは、さまざまなデータセットのコレクションに対する計算処理を表現するための統一された API を提供することにより、こうした技術的困難を解決します。

2.2 Research と IDE

理想的なアルゴリズムデザインワークフローには、調査（Research）段階と、実装（Implementaion）段階が内在します。Reserch 環境内では、notebook を通じてデータに触れたり、様々なアイデアを素早く試したりすることができます。アルゴリズムは、バックテストが可能な IDE の中で実装されます。

パイプライン API の特徴のひとつは、パイプラインの構築は Research と IDE の両方で同じであることです。2 つの環境内でパイプラインを使用する際の唯一の違いは、その実行方法です。この特徴は、パイプラインを Reserch 環境でデザインし、それを IDE 内のアルゴリズムに単純にコピーアンドペーストできることを可能にしています。このワークフローは、後ほどのレッスンで詳細に議論する予定ですが、チュートリアル全体を通して見ていくことになるでしょう。

2.3 計算処理

パイプラインで表現される計算処理には、「ファクター（Factors）」「フィルタ（Filters）」「クラシファイア（Classifires）」の 3 種類あります。

抽象的にいうと、ファクター、フィルタ、クラシファイアで表されるすべての関数は、とある資産のある一時点における何らかの値を出力します。ファクター、フィルタ、クラシファイアは、出力する値の種類によって区別されます。

2.3.1 ファクター

ファクターとは、とある資産のある一時点から数値を出力する関数のことです。

ファクターの単純な例は、とある証券の直近値段です。証券の銘柄名と特定時点を与えられることによって、その直近の値段が数値として返ってきます。また別の例として、とある証券の 10 日間の平均出来高が挙げられます。ファクターは数値を証券に割り当てるために最も一般的に利用され、いろいろな方法で用いられます。ファクターは、以下のような処理において使われます。

- ターゲットとなる比率の計算
- アルファシグナルの生成
- より複雑なファクターの作成
- フィルタの作成

2.3.2 フィルタ

フィルタとは、とある資産のある一時点からブール型の値を出力する関数のことです。

フィルタの一例は、とある証券の値段が 10 ドル以下かどうかを表す関数です。証券の銘柄名と特定時点を与えられることによって、True または False によって評価されます。フィルタは特定の目的に対し、資産の集合が該当するか否かを表現するために最も一般的に利用されます。

2.3.3 クラシファイア

クラシファイアとは、とある資産のある一時点から分類型の値を出力する関数のことです。

より明確には、クラシファイアは数値で表現することのできない、string 型、あるいは int 型（例えば数値でラベリングされた業種コードなど）を返します。クラシファイアはファクター計算上の複雑な変換を行ううえで、資産をグループ化するために最も一般的に利用されます。クラシファイアの一例は、とある資産の現在取引可能な取引所を返す関数です。

2.4 データセット

パイプライン処理は、四本値や出来高 、 財務データ 、そして コンセンサス予想データ といった 多様なデータ を用いて実行することが可能です。後ほどのレッスンで、こうしたデータセットを見ていくことにします。

典型的なパイプラインには通常、複数の計算処理とデータセットを含んでいます。このチュートリアルでは、10 日間平均値段と 30 日値段の間で大きな値動きがあった流動性の高い証券を選別するパイプラインを構築していきます。

レッスン 2 から 11 は、research 環境で実行します。research を利用可能な状態にするためには Jupyter notebook を新規作成するか、"Get Notebook"をクリックすることでこのレッスンの notebook バージョンを複製することができます。もしあなたががまだ research 環境に不慣れであれば Introduction to Research レクチャーから始めるか、documentation に目を通しておくことをお薦めします。

2.5 パイプラインを作成する

いくつかの import 文を追加するところから始めましょう。まず始めに、 `Pipeline` クラスをインポートします。

```
from quantopian.pipeline import Pipeline
```

新しいセルの中で、パイプラインを作成するための関数を定義します。関数の中にパイプラインの作成を内包することによって、後ほど紹介する、より複雑なパイプラインを構築するための土台が出来上がります。ここで、以下の関数は単純に空のパイプラインを返します。

```
def make_pipeline():
    return Pipeline()
```

新しいセルで make_pipeline() を実行することにより、パイプラインをインスタンス化しましょう:

```
my_pipe = make_pipeline()
```

2.6 パイプラインを実行する

空のパイプラインを参照する my_pipe を作成しました。これを実行し、どのようになっているかを見てみましょう。ただし、パイプラインを実行する前に run_pipeline をインポートする必要があります。これは research 環境だけで使える関数で、指定した期間を通してパイプラインを実行することを可能にします。

```
from quantopian.research import run_pipeline
```

では run_pipeline を使いパイプラインを 1 日（2015-05-05）だけ実行してその中身を表示してみましょう。なお、第 2 引数と第 3 引数はそれぞれシミュレーションの「開始日付」と「終了日付」です。

```
result = run_pipeline(my_pipe, '2015-05-05', '2015-05-05')
```

run_pipeline を実行すると、日付と証券でインデックスされた pandas DataFrame が返ってきます。空のパイプラインがどのようになっているか見てみましょう。

```
result
```

空のパイプラインは列データを持たない DataFrame を出力しています。今回の場合、パイプラインは 2015 年 5 月 5 日に対して 8000 超 (上の表では途中中断) の証券からなるインデックスを持っていますが、列データを何も持っていません。

以降のレッスンでは、パイプラインの出力に対してどのように列を追加していくのか、どのようにフィルタをかけて証券を絞り込んでいくのかをみていきます。

2.7 ファクター

ファクターは、ある資産とある一時点から数値型を返す関数です。

$$F(asset, timestamp) \rightarrow float$$

パイプラインでは ファクター は最も一般的な用語であり、数値として返される計算結果を表します。ファクターには入力値として列データと計算区間が必要です。

パイプラインにおける最も単純なファクターは、ビルトイン・ファクター と呼ばれるものです。ビルトイン・ファクターは、一般的によく使われる計算を実行するために予め用意されています。最初の例として、資産ごとに 10

日間の区間を動かしながら終値の平均値段を計算するファクターを作成しましょう。指定した計算区間（10 日間）を対象に入力データ（終値）の平均値を計算する SipleMovingAverate というビルトイン・ファクターを使います。これを使うためには SimpleMovingAverage ビルトイン・ファクターと、 USEquityPricing dataset をインポートします。

```
from quantopian.pipeline import Pipeline
from quantopian.research import run_pipeline

# 前回のレッスンに加えて、USEquityPricing dataset をインポート
from quantopian.pipeline.data.builtin import USEquityPricing

# 前回のレッスンに加えて、SimpleMovingAverage ビルトイン・ファクターをインポート
from quantopian.pipeline.factors import SimpleMovingAverage
```

2.8 ファクターの作成

前のレッスンで作成した make_pipeline 関数に戻って、 SimpleMovingAverage ファクターをインスタンス化します。SimpleMovingAverage の作成には、SimpleMovingAverage に input（ BoundColumn 型のリスト）と、window_length（移動平均の計算が受け取るデータ日数を表す整数値）という 2 つの引数を渡してコンストラクタを呼び出します。（ BoundColumn については後ほど詳細に触れます。今の段階では、BoundColumn がファクターを作成する上で必要な必要なものであるということだけ知っていれば十分です。）

以下の一行で、証券の 10 日間移動平均を計算する **ファクター** が出来上がります。

```
mean_close_10 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_length=10)
```

ここで重要なのは、ファクターの作成段階では、実際には計算を実行していないという点です。ファクターの作成は、関数を定義するのに似ています。計算を実行するためには、ファクターをパイプラインに追加して、実行する必要があります。

2.9 ファクターをパイプラインに追加する

元となる空のパイプラインを、移動平均を計算するファクターにアップデートさせましょう。まず始めに、ファクターのインスタンス化を make_pipeline 内に移動させます。次に、columns （列名に対してファクター、フィルタ、あるいはクラシファイアを紐づける辞書型）引数を通じて、パイプラインにファクターの計算を指示します。アップデートされた make_pipline 関数は、このような感じになるはずです。

```
def make_pipeline():

    mean_close_10 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_
↪length=10)
```

(次のページに続く)

(前のページからの続き)

```
    return Pipeline(
        columns={
            '10_day_mean_close': mean_close_10
        }
    )
```

中身がどのようになっているかを確認するため、パイプラインを実行して結果を表示させます。

```
result = run_pipeline(make_pipeline(), '2015-05-05', '2015-05-05')
result
```

これでパイプラインの出力に、8000 超の全銘柄（画面上は途中まで）に対して計算された 10 日間終値移動平均の列が追加されました。各行は、該当する証券と該当する日付における計算結果に対応しています。この DataFrame は、 マルチインデックス（第 1 レベルは計算を行った日付を表す日時、第 2 レベルは証券に対応する Equity オブジェクト）を持っています。例えば 1 行目 (2015-05-05 00:00:00+00:00, Equity(2 [AA])) には、2015 年 5 月 5 日の AA （訳者注：AA はアルコア社（アルミニウム、アルミニウム製品およびアルミナの世界的なメーカー）を表す証券コード）の mean_close_10 ファクターの計算結果が格納されます。

もし 1 日よりも長い期間パイプラインを実行すれば、その結果はこのようになります。

```
result = run_pipeline(make_pipeline(), '2015-05-05', '2015-05-07')
result
```

備考：Pipeline.add メソッドを用いることでも同様に Pipeline インスタンスに対してファクターを追加できます。add を使う場合はこのような感じになります：>>> my_pipe = Pipeline() >>> f1 = SomeFactor(…) >>> my_pipe.add(f1, ‘ f1 ’)

2.10 Latest

最もよく使われるビルトイン Factor は、 Latest です。Latest ファクターは、与えられたデータ列中で最も直近の値を取得します。このファクターは非常によく使われるので、他のファクターとは異なる方法でインスタンス化されます。データ列から直近の値を取得するには、 .latest アトリビュートから取得するのが最良の方法です。例として make_pipeline をアップデートして直近終値を取得するファクターを作成してパイプラインに追加してみましょう。

```
def make_pipeline():

    mean_close_10 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_
↪length=10)
    latest_close = USEquityPricing.close.latest

    return Pipeline(
```

(次のページに続く)

(前のページからの続き)

```
        columns={
            '10_day_mean_close': mean_close_10,
            'latest_close_price': latest_close
        }
    )
```

ここで再びパイプラインを作成し実行すると、出力された dataframe には 2 つの列ができます。一方は 10 日間終値移動平均の列で、もう一方は直近の終値の列になっています。

```
result = run_pipeline(make_pipeline(), '2015-05-05', '2015-05-05')
result.head(5)
```

`.latest` は **ファクター** 以外のものを返すことがあります（訳者注：latest_close_price に数値ではない NaN を含む)。これ以外の起こり得る返り値の型については後ほどみていきます。

2.11 デフォルト入力

いくつかのファクターは、変更すべきでないデフォルト入力があります。たとえば VWAP ビルトイン・ファクターは常に `USEquityPricing.close` と `USEquityPricing.volume` から計算されます。ファクターが常に同じ *BoundColumns* から計算される場合、 `input` 引数を明示せずにコンストラクタを呼び出せます。

```
from quantopian.pipeline.factors import VWAP
vwap = VWAP(window_length=10)
```

次のレッスンでは、ファクターの結合を見ていきます。

2.12 ファクターの結合

ファクターは組込み数学演算子（+、-、*など）のいずれかを使い、別のファクターやスカラ値と結合できます。これにより、複数のファクターを結合させた複雑な式を簡単にかけます。例えば 2 つの異なるファクターの平均を取るファクターの作成はシンプルに：

```
>>> f1 = SomeFactor(...)
>>> f2 = SomeOtherFactor(...)
>>> average = (f1 + f2) / 2.0
```

と書けます。このレッスンでは、10 日間平均ファクターと 30 日間平均ファクターを組み合わせた `relative_difference` ファクターを計算するパイプラインを作成します。いつもと同じく、import 文から始めます：

```
from quantopian.pipeline import Pipeline
from quantopian.research import run_pipeline
from quantopian.pipeline.data.builtin import USEquityPricing
from quantopian.pipeline.factors import SimpleMovingAverage
```

今回は、10 日間移動平均と 30 日間移動平均の 2 つのファクターが必要です：

```
mean_close_10 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_length=10)
mean_close_30 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_length=30)
```

では、mean_close_30 ファクターと mean_close_10 ファクターを使って相対変化（percent_difference）を計算するファクターを作成します。

```
percent_difference = (mean_close_10 - mean_close_30) / mean_close_30
```

この例を見ると、 percent_difference はより単純なファクターの組み合わせによって構成されていますが、依然として Factor です。percent_difference をパイプラインの列として追加できます。

それでは percent_difference を列に持つ（終値平均を計算するファクターは列に加えません）パイプラインを作成する make_pipeline を定義しましょう：

```
def make_pipeline():

    mean_close_10 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_
↪length=10)
    mean_close_30 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_
↪length=30)

    percent_difference = (mean_close_10 - mean_close_30) / mean_close_30

    return Pipeline(
        columns={
            'percent_difference': percent_difference
        }
    )
```

新しい出力がどのようになっているか確認します：

```
result = run_pipeline(make_pipeline(), '2015-05-05', '2015-05-05')
result
```

次のレッスンでは、フィルタについて学習します。

```
from quantopian.pipeline import Pipeline
from quantopian.research import run_pipeline
from quantopian.pipeline.data.builtin import USEquityPricing
from quantopian.pipeline.factors import SimpleMovingAverage
```

2.13 フィルタ

フィルタとは、とある資産のある一時点からブール型の値を出力する関数です：

$$F(asset, timestamp) \rightarrow boolean$$

フィルタ は計算過程やパイプラインの最終出力に含まれる証券の絞り込みに使われます。**フィルタ** 作成には 2 つの一般的な方法があります。比較演算子と **ファクター / クラシファイア** メソッドを使う方法です。

2.14 比較演算子

ファクター と **クラシファイア** に用いられる比較演算子から **フィルタ** が作られます。ここまで **クラシファイア** について触れていませんが、とにかく **ファクター** を使った例で進めていきます。以下の例では、直近の終値が 20 ドルを超える場合に `True` を返すフィルタを作成します。

```
last_close_price = USEquityPricing.close.latest
close_price_filter = last_close_price > 20
```

またこの例では、10 日間平均が 30 日間平均を下回るる場合に `True` を返すフィルタを作成しています。

```
mean_close_10 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_length=10)
mean_close_30 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_length=30)
mean_crossover_filter = mean_close_10 < mean_close_30
```

それぞれの証券は日付ごとに `True` または `False` の値を持っていることに留意してください。

2.15 ファクター / クラシファイア メソッド

ファクター と **クラシファイア** クラスが **フィルタ** を作成する方法は多様です。相変わらず **クラシファイア** について触れていませんが **ファクター** を使った例で進めていきます（このあと **クラシファイア** メソッドを見ていきます）。`Factor.top(n)` メソッドは、与えられた **ファクター** における上位 `n` 件の証券に `True` を返す **フィルタ** を作成します。以下の例はすべての利用可能な証券のうち、日付ごとに終値が上位 200 位以内に含まれる銘柄に対し `True` を返すフィルタを作成します。

```
last_close_price = USEquityPricing.close.latest
top_close_price_filter = last_close_price.top(200)
```

フィルタ を返す **ファクター** メソッドの一覧は、ここを参照してください。

フィルタ を返す **クラシファイア** メソッドの一覧は、ここを参照してください。

2.16 売買代金（Dollar Volume）フィルタ

最初の例として、証券の 30 日間の平均売買代金が 10,000,000 ドルより大きい場合に `True` を返すフィルタを作成します。まず、 30 日間の平均売買代金を計算する `AverageDollarVolume` ファクターを作成します。import 文を使って組込みの `AverageDollarVolume` ファクターを利用可能にします。

```
from quantopian.pipeline.factors import AverageDollarVolume
```

次に、AverageDollarVolume ファクターをインスタンス化します。

```
dollar_volume = AverageDollarVolume(window_length=30)
```

`AverageDollarVolume` はデフォルトで `USEquityPricing.close` と `USEquityPricing.volume` を利用しているため `inputs` 引数を指定する必要はありません。さてこれで売買代金ファクターが用意できたので、boolean を使ったフィルタを作成できます。以下の行で、`dollar_volume` が 10,000,000 よりも大きい証券に対して `True` を返すフィルタを作成します：

```
high_dollar_volume = (dollar_volume > 10000000)
```

フィルタの中がどのようになっているか確認するため、前のレッスンで作成したパイプラインにフィルタを追加します。

```
def make_pipeline():

    mean_close_10 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_
↪length=10)
    mean_close_30 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_
↪length=30)

    percent_difference = (mean_close_10 - mean_close_30) / mean_close_30

    dollar_volume = AverageDollarVolume(window_length=30)
    high_dollar_volume = (dollar_volume > 10000000)

    return Pipeline(
        columns={
            'percent_difference': percent_difference,
            'high_dollar_volume': high_dollar_volume
        }
    )
```

パイプラインを作成・実行すると、各証券に対してフィルタの結果を表す boolean 値が入った `high_dollar_volume` 列が作られます。

```
result = run_pipeline(make_pipeline(), '2015-05-05', '2015-05-05')
result
```

2.17 スクリーニング

通常、パイプラインは Quantopian のデータベースに存在するすべての資産を対象に、日付ごとに計算結果を出力します。しかしながら、特定の基準を満たした証券の部分集合だけが必要なケースが頻繁に起こります（例えば、日々の取引が活発で注文が即座に成立するような銘柄のみが必要となることがあります）。パイプラインの中で `screen` キーワードを使うと、パイプラインの実行でフィルタが `False` を返した銘柄をふるい落とすことができます。

出力結果を 30 日間の平均売買代金が 10,000,000 ドルよりも大きい証券だけに絞り込むには、`screen` の引数として `high_dollar_volume` をあてはめるだけです。`make_pipeline` はこのような感じになります：

```
def make_pipeline():

    mean_close_10 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_
↪length=10)
    mean_close_30 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_
↪length=30)

    percent_difference = (mean_close_10 - mean_close_30) / mean_close_30

    dollar_volume = AverageDollarVolume(window_length=30)
    high_dollar_volume = dollar_volume > 10000000

    return Pipeline(
        columns={
            'percent_difference': percent_difference
        },
        screen=high_dollar_volume
    )
```

実行すると、パイプラインの出力にはそれぞれの日付において `high_dollar_volume` フィルタを通過した証券のみが含まれています。例えばこのパイプラインを 2015 年 5 月 5 日に対して実行した出力結果は、約 2,100 銘柄程度となります。

（翻訳者注：以下のソースコードは print 文が python2 の文法であるため、python3 では通常エラーとなる）

```
result = run_pipeline(make_pipeline(), '2015-05-05', '2015-05-05')
print 'Number of securities that passed the filter: %d' % len(result)
result
```

```
Number of securities that passed the filter: 2110
```

2.18 フィルタの反転

~ 演算子はフィルタの反転に使われ、 True 値を False 値に置き換えます（逆もしかり）。例えば、売買代金の少ない証券にフィルタをかける場合は以下のように書きます：

```
low_dollar_volume = ~high_dollar_volume
```

この場合、過去 30 日間の平均売買代金が 10,000,000 ドル以下の銘柄に対して True が返ります。

次のレッスンでは、フィルタの結合について見ていきます。

```
from quantopian.pipeline import Pipeline
from quantopian.research import run_pipeline
from quantopian.pipeline.data.builtin import USEquityPricing
from quantopian.pipeline.factors import SimpleMovingAverage, AverageDollarVolume
```

2.19 フィルタの結合

ファクタと同じく、フィルタも結合できます。フィルタの結合には & （and）や | （or）演算子を使います。例えば直近の終値が 20 ドルよりも高く、かつ、平均売買代金が全銘柄の上位 10 パーセントという条件でスクリーニングを実施したいとしましょう。まず売買代金フィルタを AverageDollarVolume ファクター と percentile_between を使って作成します：

```
dollar_volume = AverageDollarVolume(window_length=30)
high_dollar_volume = dollar_volume.percentile_between(90, 100)
```

（備考） percentile_between は、 **フィルタ** を返す **ファクター** メソッドです。

次に latest_close ファクターを作成し、終値が 20 ドルよりも高い証券に絞り込むフィルタを定義します：

```
latest_close = USEquityPricing.close.latest
above_20 = latest_close > 20
```

そして high_dollar_volume フィルタと above_20 フィルタを & 演算子を使って結合します：

```
tradeable_filter = high_dollar_volume & above_20
```

このフィルタは high_dollar_volume と above_20 の双方が True となる証券に対して True と判定します。それら以外は False となります。類似の演算が | (or) 演算子を使って実行可能です。このフィルタをパイプライン内でスクリーニングに利用したい場合、 screen 引数として tradeable_filter を指定するだけです。

```
def make_pipeline():

```

(次のページに続く)

(前のページからの続き)

```
    mean_close_10 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_
↪length=10)
    mean_close_30 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_
↪length=30)

    percent_difference = (mean_close_10 - mean_close_30) / mean_close_30

    dollar_volume = AverageDollarVolume(window_length=30)
    high_dollar_volume = dollar_volume.percentile_between(90, 100)

    latest_close = USEquityPricing.close.latest
    above_20 = latest_close > 20

    tradeable_filter = high_dollar_volume & above_20

    return Pipeline(
        columns={
            'percent_difference': percent_difference
        },
        screen=tradeable_filter
    )
```

実行すると、パイプラインは約 700 銘柄を出力します。

```
result = run_pipeline(make_pipeline(), '2015-05-05', '2015-05-05')
print 'Number of securities that passed the filter: %d' % len(result)
result
```

課題: print 文を python3 用に書き換えるか検討

```
Number of securities that passed the filter: 737
```

次のレッスンでは、ファクターとフィルタのマスキングについて見ていきます。

```
from quantopian.pipeline import Pipeline
from quantopian.research import run_pipeline
from quantopian.pipeline.data.builtin import USEquityPricing
from quantopian.pipeline.factors import SimpleMovingAverage, AverageDollarVolume
```

2.20 マスキング

パイプラインを処理するときに特定の資産を除外したいことがあります。資産が除外できると便利な事例が 2 つあります：

1. 計算コストが高い評価式を計算したいが、特定の資産の結果だけ確認できれば良い場合。そのような計算コストが高い評価式の例として、回帰係数 (RollingLinearRegressionOfReturns) を計算する **ファクター** が挙げられます。

2. 資産どうしの比較を行いたいが、計算は全資産のうちの一部分だけを対象にして実行したい場合。たとえば **ファクター** メソッドのひとつである `top` を使って益回り (earnings yield) の高い上位 200 銘柄を計算したいが、一定の流動性制約を満たさない資産は除外したいと考える場合が挙げられます。

このようなユースケースに対応するため、すべての **ファクター** と多くの **ファクター** メソッドは、mask 引数を受け取ることができます。ただし、その引数は計算時にどのような資産を対象とすべきかを示す `Filter` になってなければなりません。

2.21 ファクターに対するマスキング

売買代金が 10,000,000 ドルよりも大きい銘柄のみを対象とし、値段変化率が高いまたは低い銘柄を出力するパイプラインを考えてみましょう。これを実現するためには `make_pipeline` 関数を再編集して最初に `high_dollar_volume` フィルタを作成します。そして `high_dollar_volume` フィルタを `SimpleMovingAverage` の `mask` 引数として渡すことで、移動平均を計算するファクターの `mask` とすることができます。

```
# 売買代金を計算するファクター
dollar_volume = AverageDollarVolume(window_length=30)

# 高売買代金フィルタ
high_dollar_volume = (dollar_volume > 10000000)

# 終値移動平均を計算するファクター
mean_close_10 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_length=10,␣
↪mask=high_dollar_volume)
mean_close_30 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_length=30,␣
↪mask=high_dollar_volume)

# 変化率を計算するファクター
percent_difference = (mean_close_10 - mean_close_30) / mean_close_30
```

`high_dollar_volume` フィルタによって `SimpleMovingAverage` をマスキングすることで、終値平均値段のファクターの計算対象はマスキング無しで約 8,000 銘柄となるのに対し、約 2,000 銘柄に絞り込まれます。`mean_close_10` と `mean_close_30` を組み合わせた `percent_defference` ファクターの計算は同じ約 2,000 銘柄を対象に行われます。

2.22 フィルタに対するマスキング

マスキングは top 、 bottom 、 percentile_between といったフィルタを返すメソッドに対しても適用可能です。マスキングは結合処理の早い段階でフィルタを適用したいときに最も役立ちます。例えば売買代金トップ 10% の中で始値が最も高い 50 銘柄を取り出したいとします。さらに、それらの銘柄のうち終値が第 90 分位から第 100 分位の間に含まれる銘柄を取り出したいとします。これらは以下のようにして実現できます：

```
# 売買代金を計算するファクター
dollar_volume = AverageDollarVolume(window_length=30)

# 高売買代金フィルタ
high_dollar_volume = dollar_volume.percentile_between(90,100)

# 始値上位フィルタ （高売買代金フィルタでマスキング）
top_open_price = USEquityPricing.open.latest.top(50, mask=high_dollar_volume)

# 終値分位上位フィルタ （高売買代金フィルタと始値上位 50 銘柄フィルタでマスキング）
high_close_price = USEquityPricing.close.latest.percentile_between(90, 100, mask=top_
↪open_price)
```

これを make_pipeline に追加し、high_close_price でスクリーニングをかけた空のパイプラインを出力します。

```
def make_pipeline():

    # 売買代金を計算するファクター
    dollar_volume = AverageDollarVolume(window_length=30)

    # 高売買代金フィルタ
    high_dollar_volume = dollar_volume.percentile_between(90,100)

    # 始値上位フィルタ （高売買代金フィルタでマスキング）
    top_open_price = USEquityPricing.open.latest.top(50, mask=high_dollar_volume)

    # 終値分位上位フィルタ （高売買代金フィルタと始値上位 50 銘柄フィルタでマスキング）
    high_close_price = USEquityPricing.close.latest.percentile_between(90, 100,␣
↪mask=top_open_price)

    return Pipeline(
        screen=high_close_price
    )
```

2015 年 5 月 15 日付のパイプライン実行結果は、5 銘柄の出力となります。

```
result = run_pipeline(make_pipeline(), '2015-05-05', '2015-05-05')
print 'Number of securities that passed the filter: %d' % len(result)
```

課題: python2 コードの修正どうするか

```
Number of securities that passed the filter: 5
```

上記で行ったようにレイヤにマスクを適用することは、「資産の絞り込み作業」と捉えることができます。次のレッスンでは、クラシファイアについて見ていきます。

```
from quantopian.pipeline import Pipeline
from quantopian.research import run_pipeline
from quantopian.pipeline.factors import SimpleMovingAverage, AverageDollarVolume
```

2.23 クラシファイア

クラシファイアとは、とある資産のある一時点から **文字列** や **整数** ラベルといった 分類型の値を出力 する関数のことです。

$$F(asset, timestamp) \rightarrow category$$

たとえばクラシファイアは当該証券の取引所 ID を表す文字列を出力します。このクラシファイアを作成するには、 `Fundamentals.exchange_id` をインポートして、インスタンスの latest アトリビュートを参照します。

```
from quantopian.pipeline.data import Fundamentals

# Fudamentals.exchange_id は string 型なので、.latest はクラシファイアを返します。
exchange = Fundamentals.exchange_id.latest
```

以前のレッスンでは `latest` アトリビュートは **ファクター** のインスタンスを返していましたが、今回のケースでは `string` を返しているため、`latest` アトリビュートは **クラシファイア** を作成していることがわかります。

同様に、ある証券に対して直近のモーニングスター社による業種コードを返すコードは **クラシファイア** と言えます。このケースでは返り値は `int` となりますが、この整数の値は数値を表すものではなく区分を表すものなので、このコードもまたクラシファイアを作成しています。

直近の業種コードは、組込の `Sector` クラシファイアを使うことで取得できます。

```
from quantopian.pipeline.classifiers.fundamentals import Sector
morningstar_sector = Sector()
```

`Sector` を用いるのと、 `Fundamentals.morningstar_sector_code.latest` は同じ結果となります。

2.24 クラシファイアからフィルタを作る

クラシファイアは isnull 、 eq 、 startwith などのメソッドを使ってフィルタを作成することにも使われます。フィルタを作ることができる **クラシファイア** メソッドの一覧は ここにあります。

たとえばニューヨーク証券取引所で取引されている証券を選択するフィルタが必要な場合、 exchange クラシファイアに eq メソッドを使うことで実現できます。

```
nyse_filter = exchange.eq('NYS')
```

このフィルタは直近の exchange_id が NYS であるものに対して True を返します。

2.25 分位数

クラシファイアは、さまざまな **ファクタ** メソッドからも作り出すことができます。その最たる例が quantiles **（分位数）** メソッドです。このメソッドは引数として分割数（bin）を受け取ります。quantile メソッドは、全ての非数値型でないファクター出力に対して、0 から（bin - 1）までの数値でラベル付けを行うことで、 **クラシファイア**を返します。 非数値（ NaN ）には、-1 がラベル付けされます。四分位数（quartiles） (quantiles(4) と等価) や、五分位数（quintiles） (quantiles(5) と等価)、十分位数（deciles） (quantiles(10) と等価) といった別名メソッドを使うこともできます。

たとえばファクター出力に対して 10 分割（deciles）を行い、その第 10 分位に含まれる銘柄を選択するフィルタはこのようになります：

```
dollar_volume_decile = AverageDollarVolume(window_length=10).deciles()
top_decile = (dollar_volume_decile.eq(9))
```

パイプラインに作成したクラシファイアをセットしてその結果を確認すると以下のようになります：

```
def make_pipeline():
    exchange = Fundamentals.exchange_id.latest
    nyse_filter = exchange.eq('NYS')

    morningstar_sector = Sector()

    dollar_volume_decile = AverageDollarVolume(window_length=10).deciles()
    top_decile = (dollar_volume_decile.eq(9))

    return Pipeline(
        columns={
            'exchange': exchange,
            'sector_code': morningstar_sector,
            'dollar_volume_decile': dollar_volume_decile
        },
```

(次のページに続く)

(前のページからの続き)

```
        screen=(nyse_filter & top_decile)
    )
```

```
result = run_pipeline(make_pipeline(), '2015-05-05', '2015-05-05')
print 'Number of securities that passed the filter: %d' % len(result)
result.head(5)
```

```
Number of securities that passed the filter: 513
```

クラシファイアはファクター出力に対する複雑なグループ化処理を表現することにも役立ちます。demean や、groupby といった集計処理はこのチュートリアルの範囲を超えます。より一歩進んだクラシファイアの使い方については、今後のチュートリアルで取り上げることになるでしょう。

次のレッスンでは、パイプラインの中で使うことができるほかのデータセットを見ていきます。

パイプラインを作成する際、計算に必要な入力データを特定する方法が必要となります。パイプライン計算に必要な入力データは、 Dataset と BoundColumns を使うことで特定できます。

2.26 データセット（Dataset）と連結列（BoundColumns）

DataSets はパイプライン API に対して計算に必要な入力データを、どこからどのように取り出すのかを知らせる単純なオブジェクトのコレクションです。ここまでに Dataset の一例としてすでに USEquityPricing を紹介しました。

BoundColumn とは、DataSet と密接に連結したデータ列を表します。BoundColumn のインスタンスは、DataSets のアトリビュートにアクスすることで動的に作られます。パイプライン計算に対する入力データは、BoundColumn 型でなければなりません。BoundColumn の一例としてすでに USEquityPrincing.close を紹介しました。

DataSets と BoundColumn は、実際のデータを保持していない、ということを理解することが大切です。計算処理を記述してパイプラインに追加した場合、実際の計算処理はパイプラインが実行されるまでは実際には計算を実行していないことを思い出してください。

DataSets と BoundColumn は同じ方法で解釈することができます。これらは単純に計算の入力データを定義するために使われているのです。そしてデータはパイプラインが実行された後に配置されます。

2.27 dtypes

パイプライン計算を定義する場合、使用できる関数や操作を知るために、入力データのデータ型を知っている必要があります。BoundColumn の dtype は、パイプラインが実行されたときのデータ型を教えてくれます。たとえば、 USEquityPricing は float 型の dtype を持っているのでファクターは USequityPricing.close を使って算術計算を実行できます。

BoundColumn の dtype は、計算処理のタイプも決定できます。たとえば latest 計算では、 dtype の型によって計算処理がファクター（ float ）なのか、フィルタ（ bool ）なのか、はたまたクラシファイア（ string または int）なのかが決まります。

2.28 価格データ（Pricing Data）

米国株の株価は、 USEquityPricing データセットに保存されています。 USEquityPricing は 5 つの列を持っています。

- USEquityPricing.open（始値）
- USEquityPriging.high（高値）
- USEquityPricing.low（安値）
- USEquityPricing.close（終値）
- USEquityPricing.volume（出来高）

それぞれの列は float の dtype を持っています。

2.29 財務データ（Fundamental Data）

Quantopian では モーニングスター が提供する多くの財務データを利用できます。財務データは Fundamental データセットにおいて BoundColumn として存在しており、900 種類以上が利用可能です。詳しくは Quantopian Fundamental Reference を参照してください。

2.30 パートナーデータ（Partner Data）

Quantopian では `USEquityPricing`やモーニングスター財務データのほかにも多くのデータセットを利用できます。コンセンサス予想データ、ニュースセンチメントなどがこれらに含まれます。ほとんどのデータセットは quantopian.pipline.data 以下に、プロバイダ名で名前空間が定義されています。

USEquityPricing 同様、そのほかのデータセットもパイプライン計算に利用される列（ BoundColumn ）を持ちます。詳しい情報は、データを使う際のパイプラインの例とともに、 Data Reference に記載されています。

dtype は多岐にわたります。

BoundColumns は次のレッスンで見ていくカスタムファクターで一般的に使われます。

```
from quantopian.pipeline import Pipeline
from quantopian.research import run_pipeline
from quantopian.pipeline.data.builtin import USEquityPricing
from quantopian.pipeline.factors import SimpleMovingAverage, AverageDollarVolume
```

2.31 カスタムファクター

このレッスンでファクターについて最初に触れたとき、ビルトインファクターを見ていきました。しかしながら多くの場合、実施したい計算処理はビルトインファクターとして準備されていません。パイプライン API の最も強力な機能のひとつは、カスタムファクターを自身で作成できる機能です。実施したい計算処理が組み込みに存在しなければ、自分でカスタムファクターを作れます。

概念的には、カスタムファクターはビルトインファクターと同じです。カスタムファクターは、 inputs 、window_length 、および mask をコンストラクタ引数として受け取ることができ、Factor オブジェクトを計算対象日ごとに返します。

ビルトインとして用意されていない複雑な計算例「標準偏差」を例にとってみます。計算区間をずらしながら 標準偏差 を計算するには quantopian.pipeline.CustomFactor をからサブクラスを作成して compute メソッドを実装します。このメソッドのシグネチャ（訳注：メソッドの入力と出力の定義）は：

```
def compute(self, today, asset_ids, out, *inputs):
    ...
```

- *inputs は M 行 N 列の numpy arrays です。ここで M は window_length 、N は銘柄数（ mask を使わない場合は通常 8,000 程度）です。 *inputs は計算区間のデータとなります。ファクターの入力リストに渡される BoundColumn ごとに 1 つの M 行 N 列の配列が存在することに注意してください。各配列のデータ型は対応する BoundColumn の dtype となります
- out は長さ N の空の配列です。 out は計算日ごとのカスタムファクターの計算結果となります。計算ジョブは出力を out に書き出します。
- asset_ids は長さ N の整数 配列 となります。*inputs の列データに対応する銘柄コードが含まれます
- today は pandas タイムスタンプ型 となります。compute が呼び出されたときの日付を表します

当然 *inputs と out が最も一般的に使われます。

パイプラインに追加された CustomFactor のインスタンスは compute メソッドを毎日呼び出します。たとえば過去 5 日間の終値による移動標準偏差を計算するカスタムファクターを定義してみます。まずはじめに、CustomFactor と numpy を import 文に追加します。

```
from quantopian.pipeline import CustomFactor
import numpy
```

次に numpy.nanstd を使い、計算期間に対して標準偏差を計算するカスタムファクターを定義します。

```
class StdDev(CustomFactor):
    def compute(self, today, asset_ids, out, values):
        # NaN を無視して、列ごとの標準偏差を計算する
        out[:] = numpy.nanstd(values, axis=0)
```

最後にカスタムファクターを make_pipeline() の中でインスタンス化します：

```
def make_pipeline():
    std_dev = StdDev(inputs=[USEquityPricing.close], window_length=5)

    return Pipeline(
        columns={
            'std_dev': std_dev
        }
    )
```

パイプラインを実行すると、 以下のようなデータとともに StdDev.compute() が毎日呼び出されます：

- values: 1 つの M 行 N 列の numpy array です。ここで M は 5(計算区間) で、N は約 8,000(計算対象日に存在する銘柄の数) です
- out : 長さ N の空の配列です。この例では、 compute メソッドの実行で、5 日間の終値による移動標準偏差を out に書き込みます

```
result = run_pipeline(make_pipeline(), '2015-05-05', '2015-05-05')
result
```

```
/usr/local/lib/python2.7/dist-packages/numpy/lib/nanfunctions.py:1057: RuntimeWarning:␣
↪Degrees of freedom <= 0 for slice.
  warnings.warn("Degrees of freedom <= 0 for slice.", RuntimeWarning)
```

2.31.1 デフォルト入力

カスタムファクターの作成時、デフォルトの input と window_length を指定できます。たとえば計算期間に対して numpy.nanmean を使って 2 つのデータ列の差の平均を計算する TenDayMeanDifference カスタムファクターを定義してみます。デフォルト input として [USEquityPricing.close, USEquityPricing.open] window_length を、デフォルト window_length として 10 をセットします：

```
class TenDayMeanDifference(CustomFactor):
    # Default inputs.
    inputs = [USEquityPricing.close, USEquityPricing.open]
    window_length = 10
    def compute(self, today, asset_ids, out, close, open):
         # NaN を無視して、列ごとの差の平均を計算する
        out[:] = numpy.nanmean(close - open, axis=0)
```

この例では close と open はそれぞれ、10 行 8000 列の二次元 numpy arrays. です。もし TenDayMeanDifference に対して何も引数を与えずに呼び出せば、ここでセットしたデフォルト入力が使われます。

```
# 10 日間の始値と終値の差の平均を計算する。
close_open_diff = TenDayMeanDifference()
```

デフォルト入力はコンストラクタ呼び出しの際に引数を与えることにより上書きできます。

```
# 10 日間の高値と安値の差の平均を計算する。
high_low_diff = TenDayMeanDifference(inputs=[USEquityPricing.high, USEquityPricing.
↪low])
```

2.31.2 より複雑な例

モメンタム カスタムファクターを作成し、それをフィルタの作成に使ってみます。フィルタをパイプラインの screen として用います。

まず始めに、直近の終値を n 日前の終値で割った Momentum ファクターを定義します。ここで n は window_length となります。

```
class Momentum(CustomFactor):
    # デフォルト入力
    inputs = [USEquityPricing.close]

    # モメンタムの計算
    def compute(self, today, assets, out, close):
        out[:] = close[-1] / close[0]
```

ここで、 Momentum ファクターのインスタンスを 2 つ（10 日間モメンタムと 20 日間モメンタム）作成します。同時に 10 日間、20 日間ともに正のモメンタムを持つ銘柄に対して ``True`` を返す positive_momentum フィルターを作成します。

```
ten_day_momentum = Momentum(window_length=10)
twenty_day_momentum = Momentum(window_length=20)
```

(次のページに続く)

(前のページからの続き)

```
positive_momentum = ((ten_day_momentum > 1) & (twenty_day_momentum > 1))
```

次に、2 つのモメンタムファクターと positive_momentum フィルタを make_pipeline に追加します。また、positive_momentum をパイプラインの screen 引数に対して渡します。

```
def make_pipeline():

    ten_day_momentum = Momentum(window_length=10)
    twenty_day_momentum = Momentum(window_length=20)

    positive_momentum = ((ten_day_momentum > 1) & (twenty_day_momentum > 1))

    std_dev = StdDev(inputs=[USEquityPricing.close], window_length=5)

    return Pipeline(
        columns={
            'std_dev': std_dev,
            'ten_day_momentum': ten_day_momentum,
            'twenty_day_momentum': twenty_day_momentum
        },
        screen=positive_momentum
    )
```

このパイプライン出力は、2 つのモメンタムがともに正である銘柄の、標準偏差、10 日間モメンタム、20 日間モメンタムを出力します。

```
result = run_pipeline(make_pipeline(), '2015-05-05', '2015-05-05')
result
```

カスタムファクターはパイプライン内で独自の計算処理を定義することを可能にします。パートナーデータセットや複数のデータ列を用いた計算を実行するうえでベストの方法になることが多いです。カスタムファクターに関する完全なドキュメントは ここ を参照してください。

次のレッスンでは、これまでに学んだすべての内容を使い、アルゴリズムのためのパイプラインを作成します。

```
from quantopian.pipeline import Pipeline
from quantopian.research import run_pipeline
from quantopian.pipeline.data.builtin import USEquityPricing
from quantopian.pipeline.factors import SimpleMovingAverage, AverageDollarVolume
```

2.32 全部のせ

ここまでパイプライン API の基本コンポーネントについて見てきたので、アルゴリズムで使いたくなるパイプラインの構築をしてみましょう。

まず始めに、パイプライン出力の銘柄数を絞り込むためのフィルタの作成を行います。今回は、以下の条件を全て満たす銘柄のみを抽出するフィルタを作成します。

- プライマリ株式であること（訳注：ここによれば、会社が IPO を行い、現在も活発に取引が行われている最初の株式のことを表す）
- 普通株式（common stock）であること
- 預託証券 (ADR/GDR) ではないこと
- 相対取引 (OTC) されていないこと
- 発行日前取引 ではないこと（訳注：発行日前取引 ）
- 会社名に limited partnership (LP) が含まれないこと
- 企業情報に LP であることが記載されていないこと
- ETF ではないこと (モーニングスターの財務データを利用する)

2.32.1 基準の選択理由

プライマリ株式や普通株式を選択することで、1 企業に対して 1 銘柄のみを選択できます。通常、プライマリ株式は企業の代表的な証券であるため、パイプラインではこれらを選択することにします（訳注：株式には普通株のほかに優先株や劣後株といった複数の種類が存在するため、代表的な株式のみをパイプラインの出力対象としていると考えられます）。

ADR や GDR は異なる取引所で売買される株式に対する米国株式市場における預託証書の取引です。これらには為替変動による固有のリスクが発生することがしばしば起こるのでパイプラインから除外することにします。

OTC や WI、そして LP 株式はほとんどのブローカで取り扱っていないためパイプラインから除外となります。

課題: 以下の一文は Tutorial には存在するが、Notebook には存在しない。掲載すべきか？

証券を比較したりランク付けする場合、ETF には財務データがないため、通常の株式と ETF との比較はほぼ無意味です。ETF の価値は保有する多数の証券から生まれます。りんごとオレンジを比較することがないよう、ETF はパイプラインから除外します。

2.33 パイプライン作成

それぞれの基準毎にフィルタを作成し組み合わせることで、 tradable_stocks フィルタを作成します。まず、モーニングスターの DataSet と IsPrimaryShare ビルトインフィルタをインポートします。

```
from quantopian.pipeline.data import Fundamentals
from quantopian.pipeline.filters.fundamentals import IsPrimaryShare
```

次に、フィルタを定義します。

```
# プライマリ株式に対するフィルタ。IsPrimaryShare はビルトインフィルタです。
primary_share = IsPrimaryShare()

# 普通株式に対するフィルタ（普通株と対照的な株式には「優先株」がある）
# 'ST00000001' は普通株を意味する
common_stock = Fundamentals.security_type.latest.eq('ST00000001')

# ADR ではないものに対するフィルタ。~ 演算子は、フィルタを反転させる
not_depositary = ~Fundamentals.is_depositary_receipt.latest

# OTC ではないものに対するフィルタ。
not_otc = ~Fundamentals.exchange_id.latest.startswith('OTC')

# WI ではないものに対するフィルタ。
not_wi = ~Fundamentals.symbol.latest.endswith('.WI')

# 名前に LP を含まないものに対するフィルタ。.matches は正規表現を使って検索します。
not_lp_name = ~Fundamentals.standard_name.latest.matches('.* L[. ]?P.?$')

# モーニングスターの財務データ中の limited_partnarship フィールドが Null であるものに対するフィルタ。
not_lp_balance_sheet = Fundamentals.limited_partnership.latest.isnull()

# 直近のモーニングスターの時価総額フィールドが Null でない場合は ETF ではない。
have_market_cap = Fundamentals.market_cap.latest.notnull()

# 以上全てのフィルタに適合する銘柄に対するフィルタ
tradeable_stocks = (
    primary_share
    & common_stock
    & not_depositary
    & not_otc
    & not_wi
    & not_lp_name
    & not_lp_balance_sheet
    & have_market_cap
)
```

フィルタを定義する際、これまでのレッスンで取り扱っていない notnull 、 startswith 、 endswidh 、

matches といった Classifier メソッドを使っていることに注意してください。これらのメソッドに関するドキュメントは ここ を参照してください。

次に、20 日間の平均売買代金の上位 30% に対するフィルタを作成します。ここではこのファクターを base_universe と命名します。

```
base_universe = AverageDollarVolume(window_length=20, mask=tradeable_stocks).
↪percentile_between(70, 100)
```

2.33.1 ビルトインユニバース

いまここで、売買代金にもとづいて`売買可能な`銘柄を選択する基本ユニバースを構築しましたが、Quantopian では同様のことを実現するビルトインフィルタを用意しています。その中でも最良で最新のフィルタが、QTradableStocksUS です。

QTradableStocksUS は日々の銘柄ユニバースを選択するビルトインフィルタです。このフィルタは 3 種類のフィルタを通して選択基準を維持する、サイズ制約のない可能な限り流動性の高いユニバースを構築します。QTradableStocksUS は以前の Q500US や Q1500US フィルタとは異なりサイズ上限がありません。

QTradableStocksUS に関するより詳細な銘柄選定基準は、ここを参照してください。

パイプラインを簡略化するため、ここまでに書いた base_universe を QTradableStocksUS ビルトインファクターに置き換えます。まず、import 文が必要です。

```
from quantopian.pipeline.filters import QTradableStocksUS
```

次に base_universe に対して、 QTradableStocksUS をセットします。

```
base_universe = QTradableStocksUS()
```

これで銘柄の絞り込みを行う base_universe が用意できました。次は対象銘柄に適用するファクターの構築に目を向けます。今回は、平均回帰戦略（mean reversion strategy）のためのパイプラインを作成します。この戦略は、10 日間と 30 日感の移動平均値段（終値）を使います。2 つの移動平均の変動率が最も小さい 75 銘柄に対して同じ金額だけ株式を保有する一方、変動率が最も大きい 75 銘柄に対して同じ金額を空売りするアルゴリズムとします。

これを実現するため、 base_universe フィルタをパイプラインのマスクとして適用し、2 つの移動平均ファクタを作成します。そして 2 つのファクターを組み合わせて、変動率を計算するファクターを作成します。

```
# 10 日間の終値移動平均
mean_10 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_length=10,␣
↪mask=base_universe)

# 30 日間の終値移動平均
```

(次のページに続く)

(前のページからの続き)

```
mean_30 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_length=30,␣
↪mask=base_universe)

percent_difference = (mean_10 - mean_30) / mean_30
```

次に percent_difference を使い、上位 75 銘柄と下位 75 銘柄を選択するフィルタをそれぞれ作成します。

```
# 空売りする銘柄を選択するフィルタを作成
shorts = percent_difference.top(75)

# 購入する銘柄を選択するフィルタを作成
longs = percent_difference.bottom(75)
```

shorts と longs を結合して、パイプラインのスクリーニングに使うフィルタを作成します。

```
securities_to_trade = (shorts | longs)
```

コードの前の方にあるフィルタはこの最終フィルタ（ securities_to_trade ）を構築するためのマスクとして使用してきたので、securities_to_trade をスクリーンとして用いるとパイプライン出力される銘柄はこのレッスンの冒頭で見てきた基準（プライマリ株、非 ETF など）を満たします。同様に売買代金も高いものとなります。

（訳注：base_universe はの選択基準は途中で QTradableStockUS に置き換えているため、実際には QTradableStockUS の選択基準を満たしています）

最後に、パイプラインをインスタンス化します。同じ金額だけ購入あるいは空売りをするアルゴリズムとしているので、パイプライン出力する情報は取引を行う銘柄（パイプラインのインデックスとして出力されます）と、それを購入するのか空売りするのかという情報だけです。longs と shorts フィルタをパイプラインに追加し、screen として securities_to_trade をセットします。

```
def make_pipeline():

    # ベースとなるユニバース
    base_universe = QTradableStocksUS()

    # 10 日間の終値移動平均
    mean_10 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_length=10,␣
↪mask=base_universe)

    # 30 日間の終値移動平均
    mean_30 = SimpleMovingAverage(inputs=[USEquityPricing.close], window_length=30,␣
↪mask=base_universe)

    # 変化率ファクタ
    percent_difference = (mean_10 - mean_30) / mean_30
```

(次のページに続く)

(前のページからの続き)

```
    # 空売り銘柄を選択するフィルタ
    shorts = percent_difference.top(75)

    # 購入銘柄を選択するフィルタ
    longs = percent_difference.bottom(75)

    # 売買を行う銘柄を選択するフィルタ
    securities_to_trade = (shorts | longs)

    return Pipeline(
        columns={
            'longs': longs,
            'shorts': shorts
        },
        screen=securities_to_trade
    )
```

パイプラインの実行結果は、2 つの列データを持つ DataFrame が返ってきます。これらの列には銘柄に対する購入するか空売りするかを示す boolean 値が日付ごとに格納されます。

```
result = run_pipeline(make_pipeline(), '2015-05-05', '2015-05-05')
result.head()
```

次のレッスンでは、このパイプラインをアルゴリズムに追加します。

2.34 IDE への移行

ここまでのレッスンは Research 環境でパイプラインを作成して実行してきました。いよいよ IDE 環境へ移行します。まず始まる前に、`Pipeline` をインポートのインポートと空のパイプライン出力を返す `make_pipeline` を作成したスケルトンアルゴリズムを作ります。

（訳注：このレッスンのコードは Research 環境では動作しません。IDE 環境（アルゴリズム）環境で動作します）

```
import quantopian.algorithm as algo
from quantopian.pipeline import Pipeline

def initialize(context):
    my_pipe = make_pipeline()
    algo.attach_pipeline(my_pipe, 'my_pipeline')

def make_pipeline():
    return Pipeline()
```

2.35 パイプラインへのアタッチ

Research 環境では make_pipline はパイプラインオブジェクトのインスタンス化を行い、run_pipeline は実行期間指定の指定とパイプライン実行を行ってきました。アルゴリズムの中でこれを安全に実行することはできないため、どうにかしてアルゴリズムのシミュレーション（訳注：IDE 環境でアルゴリズムのバックテストを行うこと）でパイプラインを実行させる必要があります。シミュレーションでパイプラインを実行させるためには、attach_pipeline を使ってパイプラインをアタッチします。

attatch_pipeline 関数は 2 つの引数を必要とします。 Pipeline オブジェクトへの参照、および文字列による任意のパイプライン名です。attatch_pipeline をインポートして、スケルトンアルゴリズムに作成した空のパイプラインをアタッチします。

```
import quantopian.algorithm as algo
from quantopian.pipeline import Pipeline

def initialize(context):
    my_pipe = make_pipeline()
    algo.attach_pipeline(my_pipe, 'my_pipeline')

def make_pipeline():
    return Pipeline()
```

パイプラインにアタッチできたので、パイプラインは毎日一度実行されるようになりました。もし 2016 年 6 月 6 日（月）から 2016 年 6 月 10 日（金）までアルゴリズムのバックテストやライブトレードを行った場合、パイプラインは毎日 1 回（合計 5 回）実行されます。アタッチしたパイプラインは、実行日ごとに新しい DataFrame を出力します。しかしながら、現在のシミュレーション対象日はパイプライン計算の日付として暗に示されるため出力された DataFrame は日付によるインデクスを持ちません。

2.36 パイプライン出力

日々のパイプライン出力結果は、 before_trading_start の中の pipeline_output で取り出すことができます。pipeline_output は引数としてアタッチしたパイプライン名を必要とし、シミュレーション対象日におけるシミュレーション結果である Dataframe を返します。pipeline_output をインポートし、スケルトンアルゴリズムを修正して日々のパイプライン出力を context にストアできるようにします。

```
import quantopian.algorithm as algo
from quantopian.pipeline import Pipeline

def initialize(context):
    my_pipe = make_pipeline()
    algo.attach_pipeline(my_pipe, 'my_pipeline')

def make_pipeline():
```

(次のページに続く)

(前のページからの続き)

```
    return Pipeline()

def before_trading_start(context, data):
    # パイプライン出力の DataFrame を context にストアする
    context.output = algo.pipeline_output('my_pipeline')
```

これでスケルトンアルゴリズムは約 8000 行と 0 個の列を持つ空の Dataframe を毎日出力するようになりました。出力結果はこのようになります（Research 環境とは異なり、MultiIndex にならないことに注目してください)。

2.37 Research 環境で作成したパイプラインの使用法

以前のレッスン で作成したパイプラインのアルゴリズムに移行するには、Research 環境で使った必要な import 文と、 make_pipeline 関数を、アルゴリズムにコピーするだけです。以下のパイプラインを実行すると 150 行と 2 つの列（longs と shorts ）をもつ Dataframe をシミュレーション対象日ごとに

context にストアします。

```
import quantopian.algorithm as algo
from quantopian.pipeline import Pipeline
from quantopian.pipeline.data.builtin import USEquityPricing
from quantopian.pipeline.factors import SimpleMovingAverage
from quantopian.pipeline.filters import QTradableStocksUS

def initialize(context):
    my_pipe = make_pipeline()
    algo.attach_pipeline(my_pipe, 'my_pipeline')

def make_pipeline():
    """
    パイプラインを作成する
    """

    # ベースとなるユニバースとして、QTradableStockUS をセット
    base_universe = QTradableStocksUS()

    # 10 日間の終値移動平均
    mean_10 = SimpleMovingAverage(
        inputs=[USEquityPricing.close],
        window_length=10,
        mask=base_universe
    )

    # 30 日間の終値移動平均
    mean_30 = SimpleMovingAverage(
        inputs=[USEquityPricing.close],
```

(次のページに続く)

(前のページからの続き)

```
        window_length=30,
        mask=base_universe
    )

    percent_difference = (mean_10 - mean_30) / mean_30

    # 空売り銘柄を選択するフィルタ
    shorts = percent_difference.top(75)

    # 購入銘柄を選択するフィルタ
    longs = percent_difference.bottom(75)

    # 全取引銘柄を選択するフィルタ
    securities_to_trade = (shorts | longs)

    return Pipeline(
        columns={
            'longs': longs,
            'shorts': shorts
        },
        screen=(securities_to_trade),
    )

def before_trading_start(context, data):
    # パイプライン出力の DataFrame を context にストアする
    context.output = algo.pipeline_output('my_pipeline')
```

シミュレーション対象日ごとのパイプライン出力は以下のような感じになります：

購入または空売りを行う銘柄に対する取引量の計算と、発注を行うための関数をいくつか作成し、パイプライン出力で指定します。Getting Started Tutorial で学んだ基本知識を使い、取引量の計算と発注を行う関数を実装します。

```
def compute_target_weights(context, data):
    """
    取引量を計算します。
    """
    # 目標取引量を格納する空のディクショナリの初期化
    # 銘柄と目標取引量をマッピングします。
    weights = {}

    # longs または shorts のリストに銘柄が存在する場合、
    # 全ての銘柄が等しくなるよう取引量を決定します。
    if context.longs and context.shorts:
        long_weight = 0.5 / len(context.longs)
        short_weight = -0.5 / len(context.shorts)
    else:
        return weights
```

(次のページに続く)

(前のページからの続き)

```
    # longs または  shorts のリストの中に保有銘柄が含まれていない場合、
    # ポジションを解消（訳注：保有量をゼロにする）します。
    for security in context.portfolio.positions:
        if security not in context.longs and security not in context.shorts and␣
↪data.can_trade(security):
            weights[security] = 0

    for security in context.longs:
        weights[security] = long_weight

    for security in context.shorts:
        weights[security] = short_weight

    return weights

def before_trading_start(context, data):
    """
    パイプライン出力を取得します。
    """

    # シミュレーション日ごとにパイプライン出力を取得します。
    pipe_results = algo.pipeline_output('my_pipeline')

    # `longs`  列の値が True の場合は購入対象となります。
    # 取引可能かどうかを併せてチェックします。
    context.longs = []
    for sec in pipe_results[pipe_results['longs']].index.tolist():
        if data.can_trade(sec):
            context.longs.append(sec)

    # `shorts`  列の値が True の場合は空売り対象となります。
    # 取引可能かどうかを併せてチェックします。
    context.shorts = []
    for sec in pipe_results[pipe_results['shorts']].index.tolist():
        if data.can_trade(sec):
            context.shorts.append(sec)

def my_rebalance(context, data):
    """
    週 1 度リバランスを実行します。
    """

    # リバランスの際の目標取引量を計算します。
    target_weights = compute_target_weights(context, data)

    # 計算できたらポートフォリオのリバランスを実行します。
    if target_weights:
```

(次のページに続く)

(前のページからの続き)

```
        algo.order_optimal_portfolio(
            objective=opt.TargetWeights(target_weights),
            constraints=[],
        )
```

最後にここまでの成果をひとまとめにします。リバランスの回数は週 1 回とします。

（訳注：Quantopian のオンラインレッスン上では、Clone ボタンをクリックすることで以下のアルゴリズムを自分の IDE 環境にコピーできます）

```
from quantopian.algorithm import order_optimal_portfolio
from quantopian.algorithm import attach_pipeline, pipeline_output
from quantopian.pipeline import Pipeline
from quantopian.pipeline.data.builtin import USEquityPricing
from quantopian.pipeline.factors import SimpleMovingAverage
from quantopian.pipeline.filters import QTradableStocksUS
import quantopian.optimize as opt

def initialize(context):
    # リバランス関数を週の始めの取引開始時に実行します。
    schedule_function(
        my_rebalance,
        date_rules.week_start(),
        time_rules.market_open()
    )

    # パイプラインを作成しアルゴリズムにアタッチします。
    my_pipe = make_pipeline()
    attach_pipeline(my_pipe, 'my_pipeline')

def make_pipeline():
    """
    パイプラインを作成します。
    """

    # ベースとなるユニバースとして QTradableStocksUS をセットします。
    base_universe = QTradableStocksUS()

    # 10 日間の終値移動平均を計算します。
    mean_10 = SimpleMovingAverage(
        inputs=[USEquityPricing.close],
        window_length=10,
        mask=base_universe
    )

    # 30 日間の終値移動平均を計算します。
    mean_30 = SimpleMovingAverage(
```

(次のページに続く)

(前のページからの続き)

```
        inputs=[USEquityPricing.close],
        window_length=30,
        mask=base_universe
    )

    percent_difference = (mean_10 - mean_30) / mean_30

    # 空売り銘柄を選択するフィルタ
    shorts = percent_difference.top(75)

    # 購入銘柄を選択するフィルタ
    longs = percent_difference.bottom(75)

    # 全取引銘柄を選択するフィルタ
    securities_to_trade = (shorts | longs)

    return Pipeline(
        columns={
            'longs': longs,
            'shorts': shorts
        },
        screen=(securities_to_trade),
    )

def compute_target_weights(context, data):
    """
    取引量を計算します。
    """

    # 目標取引量を格納する空のディクショナリの初期化
    # 銘柄と目標取引量をマッピングします。
    weights = {}

    # longs または shorts のリストに銘柄が存在する場合、
    # 全ての銘柄が等しくなるよう取引量を決定します。
    if context.longs and context.shorts:
        long_weight = 0.5 / len(context.longs)
        short_weight = -0.5 / len(context.shorts)
    else:
        return weights

    # longs または  shorts のリストの中に保有銘柄が含まれていない場合、
    # ポジションを解消（訳注：保有量をゼロにする）します。
    for security in context.portfolio.positions:
        if security not in context.longs and security not in context.shorts and data.
↪can_trade(security):
            weights[security] = 0
```

(次のページに続く)

(前のページからの続き)

```
    for security in context.longs:
        weights[security] = long_weight

    for security in context.shorts:
        weights[security] = short_weight

    return weights

def before_trading_start(context, data):
    """
    パイプライン出力を取得します。
    """

    # シミュレーション日ごとにパイプライン出力を取得します。
    pipe_results = pipeline_output('my_pipeline')

    # `longs`  列の値が True の場合は購入対象となります。
    # 取引可能かどうかを併せてチェックします。
    context.longs = []
    for sec in pipe_results[pipe_results['longs']].index.tolist():
        if data.can_trade(sec):
            context.longs.append(sec)

    # `shorts`  列の値が True の場合は空売り対象となります。
    # 取引可能かどうかを併せてチェックします。
    context.shorts = []
    for sec in pipe_results[pipe_results['shorts']].index.tolist():
        if data.can_trade(sec):
            context.shorts.append(sec)

def my_rebalance(context, data):
    """
    週 1 度リバランスを実行します。
    """

    # リバランスの際の目標取引量を計算します。
    target_weights = compute_target_weights(context, data)

    # 計算できたらポートフォリオのリバランスを実行します。
    if target_weights:
        order_optimal_portfolio(
            objective=opt.TargetWeights(target_weights),
            constraints=[],
        )
```

パイプラインをバックテストで実行する場合、全体の計算スピードを向上させるためにバッチ実行で行われる点に注意してください。そのためパフォーマンスチャートは周期的に止まって見えます。

パイプラインチュートリアルは以上です。ぜひ Research 環境でパイプラインを自分自身でデザインし、アルゴリズムで実行してみてください。

第 3 章

Lectures

第 4 章

appendix

4.1 用語集

株式を **ロング** します。

ロング　銘柄を買うこと

ショート　銘柄を売ること

ターンオーバー　売買回転率

第 5 章

Indices and tables

- genindex
- modindex
- search

索引

ショート, 69

ターンオーバー, 69

ロング, 69