Programming Data Flow with Python

Song Teng

Why data flow?

Data + Flow = (Clean and Useful) Programs

Design Meeting Notes from Microsoft C# Team

Working with data

Today's programs are connected and trade in **rich, structured data**: it's what's on the wire, it's what apps and services **produce**, **manipulate** and **consume**.

Traditional object-oriented modeling is good for many things, but in many ways it deals rather poorly with this setup: it bunches functionality strongly with the data (through encapsulation), and often relies heavily on mutation of that state. It is "behavior-centric" instead of "data-centric".

Functional programming languages are often better set up for this: data is immutable (representing *information*, not *state*), and is manipulated from the outside, using a freely grow-able and context-dependent set of functions, rather than a fixed set of built-in virtual methods.

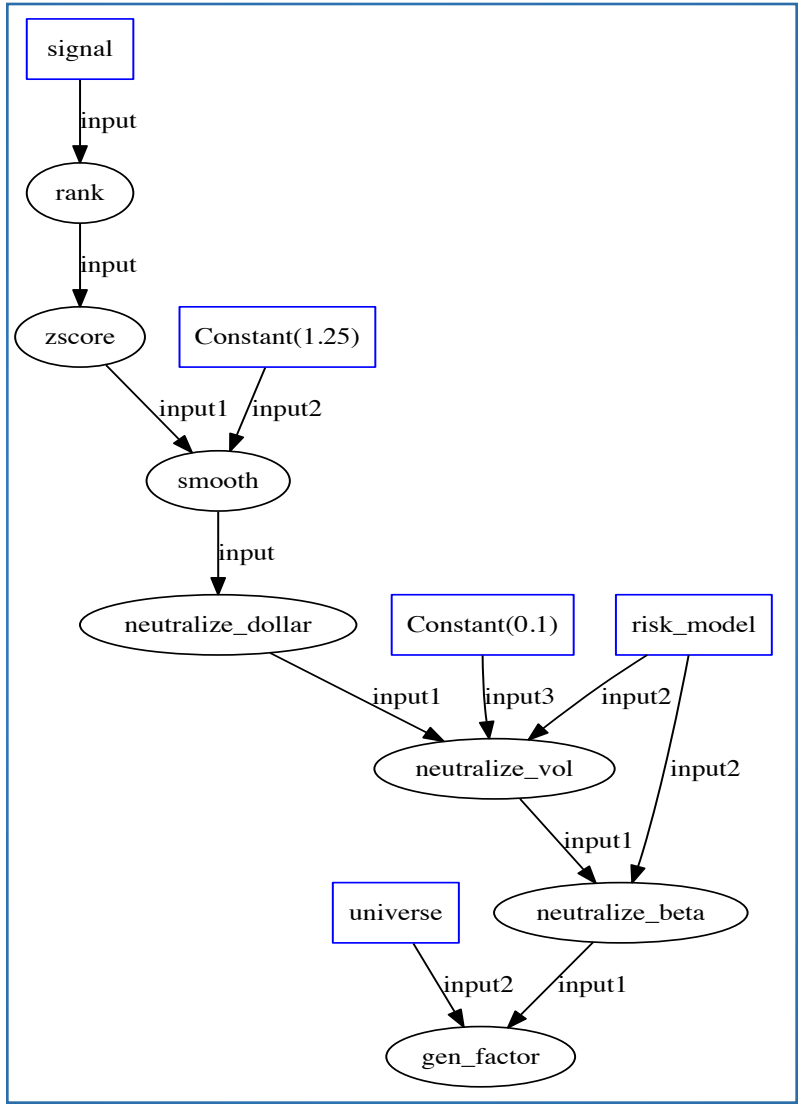
Introducing Pyflow

A functional data flow library in Python

An Example of Transform Sequence for Stock Trading Signals

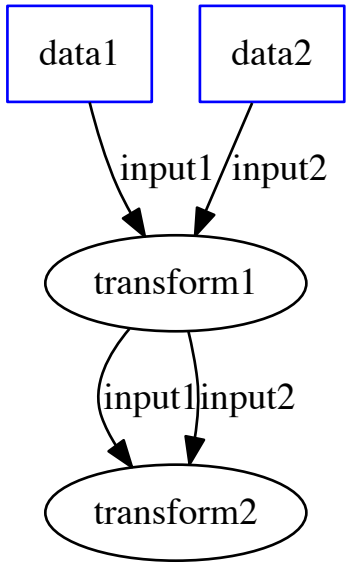
```python
@lift()
def rank(signal):
    return Signal(signal().rank())

@lift()
def zscore(signal):
    d = signal()
    return Signal((d - d.mean()) / d.std())

@lift()
def smooth(signal, scale):
    return Signal(np.tanh(signal() * scale))

@lift()
def neutralize_dollar(signal):
    weights = signal()
    long = weights[weights >= 0].sum()
    short = np.abs(weights[weights < 0].sum())
    return Signal((weights / short).where(weights < 0,  weights / long))

@lift()
def neutralize_beta(signal, risk_model):
    weights = signal()
    betas = risk_model.get_beta()
    res = ols(y=weights, x=betas, intercept=False)
    return Signal(weights - res.beta[0] * betas)

@lift('gen_factor')
def generate_factor(signal, universe):
    return Factor(universe, signal())

@lift()
def neutralize_vol(signal, risk_model, target_vol):
    weights = signal() / risk_model.get_vol()
    portfolio_vol = np.sqrt(weights.dot(risk_model.get_cov()).dot(weights.T))
    return Signal(weights / (portfolio_vol / target_vol))
```

```python
def simple_transform(signal, universe, risk_model,
                     smooth_scale=1.25, target_vol=0.1):

    signal = rank(signal)
    signal = zscore(signal)
    signal = smooth(signal, smooth_scale)
    signal = neutralize_dollar(signal)
    signal = neutralize_vol(signal, risk_model, target_vol)
    signal = neutralize_beta(signal, risk_model)
    return generate_factor(signal, universe)
```



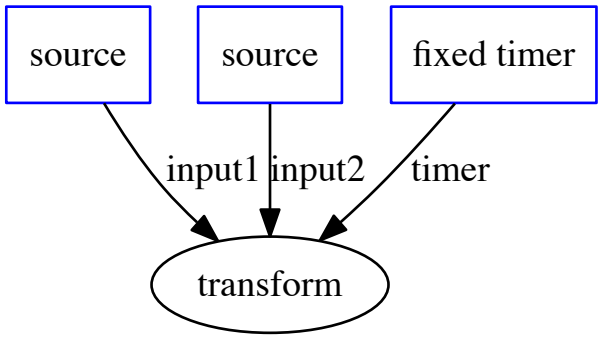
Inside “@lift(...)”

```
class Transform(Flow):
    input1 = Input()
    input2 = Input()

    output1 = Output()
    output2 = Output()

    def __init__(self, input1, input2, name):
        super().__init__(name)

    @when(input1)
    def handle(self):
        self.output1 = self.input1()

    @when(input2)
    def handle(self):
        self.output2 = self.input2()

tx1 = Transform(data1, data2, 'transform1')
tx2 = Transform(tx1.output1, tx1.output2, 'transform2')
```



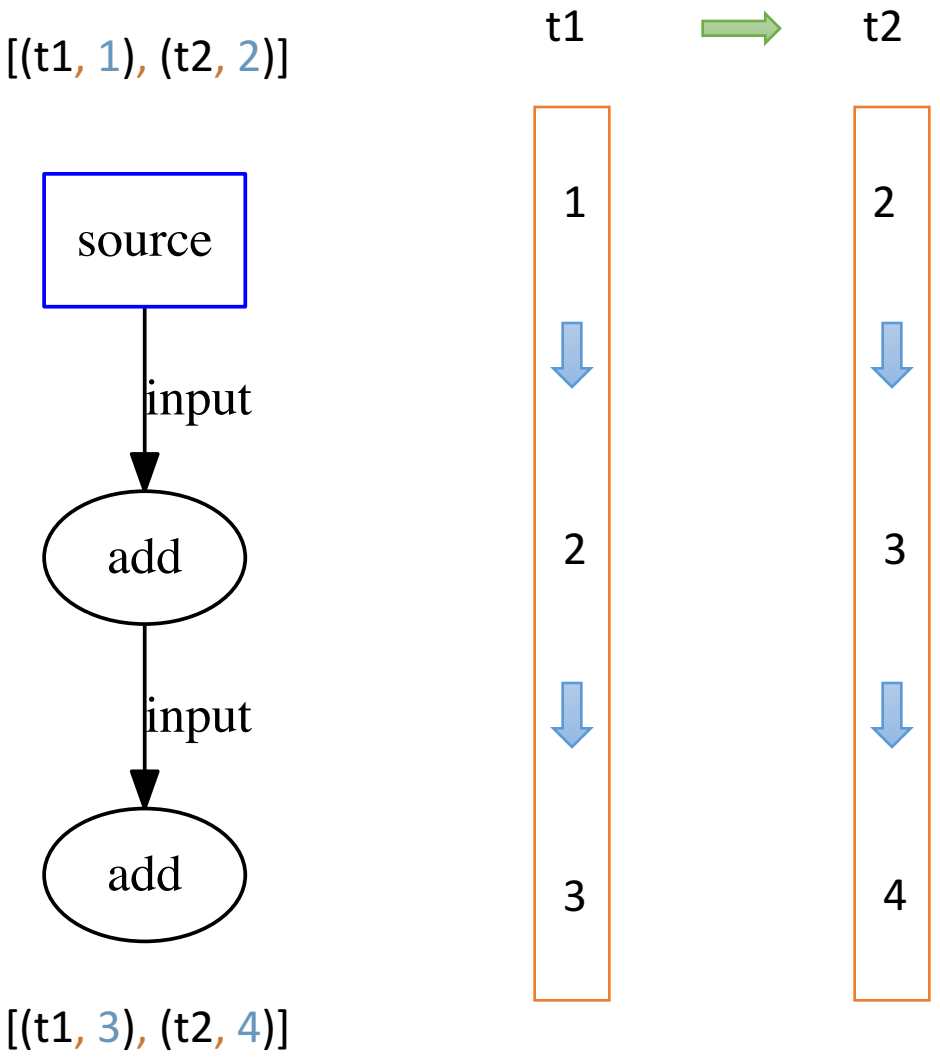
A More Interesting Example

```
class Transform(Flow):
    input1 = Input()
    input2 = Input()

    timer = Timer()

    def __init__(self, input1, input2, default_value):
        super().__init__()
        self.default_value = default_value
        self.done = False

    @when(input1)
    def handle(self):
        self.timer = self.now() + timedelta(minutes=60)

    @when(input2)
    def handle(self):
        if not self.done:
            self << self.input1() * self.input2()

    @when(timer)
    def do_timer(self):
        if not self.input2 and not self.done:
            self << self.input1() * self.default_value
            self.done = True
```



Synchronous Execution Model Enabling Deterministic Concurrency

```
t1 = datetime(2016, 8, 1, 10, 11, 12)
t2 = datetime(2016, 8, 1, 10, 11, 13)

input_series = [(t1, 1), (t2, 2)]

engine = Engine()

data = DataSource(engine, input_series)

@lift()
def add(i):
    return i + 1

tx1 = add(input_series)
tx2 = add(tx1)

engine.start(t1, t2)
```



Cut the Boilerplate with Functional Combinators

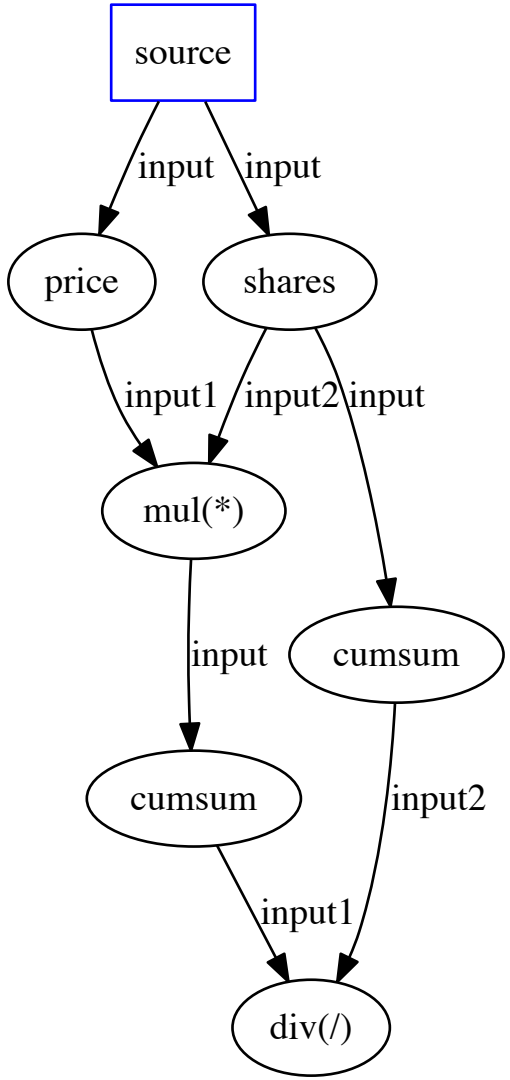
```
class Trade:
    def __init__(self, price, shares):
        self.price = price
        self.shares = shares
```

Before

```
trades = DataSource(trades)
prices = trades.map(lamba t: t.price)
shares = trades.map(lambda t: t.shares)
sp = prices.map2(shares, lambda p: s: p*s)
sp = sp.fold(0, lambda x, a: x + a)
ss = shares.fold(0, lambda x, a: x + a)
vwap = sp.map2(ss, lambda sp, ss: sp / ss)
```

After

```
trades = DataSource(trades)
prices = trades.price
shares = trades.shares
vwap = (shares * prices).sum() / shares.sum()
```



Additional Features

- Feedback to enable cyclic flow
- High-order flow (changing graph of flow while executing)
- Concurrency
- Real-time mode (reacting to real-time events)

Related Work

- Apache Spark
- Reactive Extensions (Rx.Java .....)
- Akka-Stream
- Functional Reactive Programming (FRP)
- Google Cloud Data Flow

The Genesis ...

The synchronous dataflow programming language LUSTRE *

N. Halbwachs, P. Caspi, P. Raymond
IMAG/LGI - Grenoble

D. Pilaud
VERILOG - Grenoble

Abstract

This paper describes the language LUSTRE, which is a dataflow synchronous language, designed for programming reactive systems — such as automatic control and monitoring systems — as well as for describing hardware. The dataflow aspect of LUSTRE makes it very close to usual description tools in these domains (block-diagrams, networks of operators, dynamical samples-systems, etc...), and its synchronous interpretation makes it well suited for handling time in programs. Moreover, this synchronous interpretation allows it to be compiled into an efficient sequential program. Finally, the LUSTRE formalism is very similar to temporal logics. This allows the language to be used for both writing programs and expressing program properties, which results in an original program verification methodology.