1.

Script: history\_container

Type: add

Details: add code to deal with the situation where the event does not have trade information (customized event).

From line 709 to line 713:

for i, field in enumerate(self.fields):

try:

frame\_data[i, j] = sid\_data[field]

except:

continue

2.

Script: tradesimulation.py

Type: add

Details: add code to update customized event information.

From line 224 to line 225:

elif event.type == DATASOURCE\_TYPE.CUSTOM:

self.update\_universe(event)

3.

Script: data\_source.py

Type: add, modified

Details: to accept customized source and to prevent the customized event from feeding into order execution (being regarded as TRADE type by the program). The solution is differentiate the events by event\_type attribute according to types.

From line 48 to line 51:

if 'price' in raw\_row:

event\_type = DATASOURCE\_TYPE.TRADE

else:

event\_type = DATASOURCE\_TYPE.CUSTOM

4.

Script: tradesimulation.py

Type: add

Details: to filter the current\_data for data without historical trade information, to result in a filtered data set for handle\_data.

Line 241:

self.current\_data = BarData({sid:sid\_data for sid, sid\_data \

in self.current\_data.iteritems() if 'price' in self.current\_data[sid]})

5.

Scripts: data\_source.py, protocol.py

Type: add

Details:

Add DELIST type to realize dynamic universe (remove the sid from history container)

Protocol.py

DATASOURCE\_TYPE = Enum(

'AS\_TRADED\_EQUITY',

'MERGER',

'SPLIT',

'DIVIDEND',

'TRADE',

'TRANSACTION',

'ORDER',

'EMPTY',

'DONE',

'CUSTOM',

'BENCHMARK',

'COMMISSION'

'DELIST'

)

data\_source.py

def apply\_mapping(self, raw\_row):

"""

Override this to hand craft conversion of row.

"""

if 'price' in raw\_row:

event\_type = DATASOURCE\_TYPE.TRADE

elif 'delist' in raw\_row:

event\_type = DATASOURCE\_TYPE.DELIST

else:

event\_type = DATASOURCE\_TYPE.CUSTOM

row = raw\_row

row.update({'source\_id': self.get\_hash()})

row.update({'type': event\_type})

return row

6.

Scripts: Tracker.py, algorithm.py, history\_container.py

Type: add

Details: to dynamically change sim\_params.sids and history\_container sids;

set initial sids to watch for history container, add new sids to the container every time it updates.

Tracker.py

if event.type == zp.DATASOURCE\_TYPE.TRADE:

# update last sale

self.trade\_count += 1

if event.sid not in self.sim\_params.sids:

self.sim\_params.sids.add(event.sid)

for perf\_period in self.perf\_periods:

perf\_period.update\_last\_sale(event)

elif event.type == zp.DATASOURCE\_TYPE.DELIST:

if event.sid in self.sim\_params.sids:

self.sim\_params.sids.remove(event.sid)

algorithm.py

In run():

initial\_sids = set(dbProxy.\_get\_sn\_ts(self.period\_start, self.period\_period\_start)['sid'])

self.sim\_params.sids = initial\_sids

def handle\_data(self, data):

self.\_most\_recent\_data = data

if self.history\_container:

self.history\_container.update(self.sim\_params.sids, data, self.datetime)

self.\_handle\_data(self, data)

history\_container.py

in update():

def update(self, current\_sids, data, algo\_dt):

"""

Takes the bar at @algo\_dt's @data, checks to see if we need to roll any

new digests, then adds new data to the buffer panel.

"""

to\_add = current\_sids.difference(self.sids)

self.add\_sids(to\_add)

frame = self.frame\_from\_bardata(data, algo\_dt)

self.update\_last\_known\_values()

self.update\_digest\_panels(algo\_dt, self.buffer\_panel)

self.buffer\_panel.add\_frame(algo\_dt, frame)

7.

Scripts: algorithm.py

Type: add

Set\_commission():

def set\_commission(self, commission):

if not isinstance(commission, (PerShare, PerTrade, PerDollar, PerDollar\_A)):

raise UnsupportedCommissionModel()

8.

Scripts: blotter.py

Type: edit

Details: edit the default slippage model and commission model to use.

def \_\_init\_\_(self):

self.transact = transact\_partial(MySlippage(0.1,0.0), PerDollar\_A())

9.

Scripts: algorithm.py

Type: add

Details: to let user access the current snapshot of universe.

@api\_method

def get\_universe(self):

"""

Returns a current snapshot of sids.

"""

universe\_copy = copy(self.sim\_params.sids)

return universe\_copy

10. trading control

Scripts: algorithm.py, controls.py

Type: edit

Details: to ignore orders instead of raising errors.

controls.py

def fail(self, sid, amount):

"""

Raise a TradingControlViolation with information about the failure.

"""

return False

other classes modified accordingly.

Algorithm.py

if not control.validate(sid,

amount,

self.updated\_portfolio(),

self.get\_datetime(),

self.trading\_client.current\_data):

return False

return True

11. warming\_period

Scripts: algorithm.py trading.py

Type: edit

Details: to skip the warming period in algorithm.py if self.datetime<self.sim\_params.real\_open

def calculate\_real\_open(self, env):

if self.warming\_period is None or self.warming\_period is 0:

return self.calculate\_first\_open(env)

else:

one\_day = datetime.timedelta(days=1)

ndays = 0

real\_open = self.first\_open

while (not env.is\_trading\_day(real\_open)) or (not ndays < self.warming\_period):

real\_open = real\_open + one\_day

ndays += 1

return real\_open

def handle\_data(self, data):

self.\_most\_recent\_data = data

if self.history\_container:

self.history\_container.update(self.sim\_params.sids, data, self.datetime)

if self.sim\_params.real\_open > self.datetime:

return

self.\_handle\_data(self, data)

12. Major Change: about positive cash

Scripts: slippage.py, blotter.py, tradesimulation.py

Type: add, edit

Details: start from tradesimulation, in process\_event(self, event), we now pass in an additional argument “cash”, as the remaining cash (cash itself is updating every time a trade event is processed). The next step, in blotter.py, process\_trade(), we add this argument too and is further passed in process\_transactions(). The third step is in slippage.py: there are two places impacted. The first place is transact\_stub where we add this cash argument as well; the second place is where the logic of the constraint is implemented. First in the general slippage baseclass definition, we have the available\_cash\_for\_bar attribute, similar to volume\_for\_bar. So the implementation of this constraint looks really in the same way as the volume constraint. Second in the derivedclass, in particular MySlippage, we compared the current volume with volume affordable and take the min of the two.

13. Major Change: about twice iterating trade events to make sure long orders are filled as much as possible

Type: add, edit

Details: Now every trade will get processed twice. First time they get processed when iterator is yielding the trade events, and then they get saved in a list. Later they will be processed again by looping through the list.

In myslippage, the process\_order function will update the ‘available\_volume’ field of the trade event (the trade events don’t have this field initially, so the first time the trade is processed, it will be initiated as volume\_limit\*event.volume. This field is to prevent the second time processing of trade event exceeds the max volume available for this bar so we need to make this field an attribute of the event itself instead of a temporary variable ‘remaining\_volume’ as before.

tradesimulation.py

for event in snapshot:

if event.type == DATASOURCE\_TYPE.TRADE:

#update price info for that day

self.update\_universe(event)

any\_trade\_occurred = True

trades\_to\_be\_reprocessed.append(event)

elif event.type == DATASOURCE\_TYPE.CUSTOM:

self.update\_universe(event)

elif event.type == DATASOURCE\_TYPE.BENCHMARK:

benchmark\_event\_occurred = True

#elif event.type == DATASOURCE\_TYPE.CUSTOM:

#self.update\_universe(event)

#elif event.type == DATASOURCE\_TYPE.SPLIT:

#self.algo.blotter.process\_split(event)

# execute orders

self.process\_event(event)

#process twice to make sure buy orders are filled as much as possible

for trade in trades\_to\_be\_reprocessed:

self.process\_event(trade)

slippage.py

def process\_order(self, event, order):

if 'available\_volume' not in event:

max\_volume = self.volume\_limit \* event.volume

event['available\_volume'] = max\_volume

# price impact accounts for the total volume of transactions

# created against the current minute bar

event.available\_volume = event.available\_volume - self.volume\_for\_bar

if event.available\_volume <= 100:

# we can't fill any more transactions

Return

14. dividend tackling

Details: add a new event ‘Dividend’ and process the dividend to let our positions’ information reflect the true market scenario (price, amount).

Scripts: tracker.py, data\_source.py, position.py

Type: add, edit

tracker.py

elif event.type == zp.DATASOURCE\_TYPE.DIVIDEND:

new\_dividend = pd.DataFrame(event.to\_series()).T

self.update\_dividends(new\_dividend)

position.py

def earn\_dividend(self, dividend):

assert dividend['sid'] == self.sid

out = {'id': dividend['id']}

# stock dividend

if dividend['payment\_sid']:

out['payment\_sid'] = dividend['payment\_sid']

out['share\_count'] = floor(self.amount \* dividend['ratio'])

# cash dividend

if dividend['net\_amount']:

out['cash\_amount'] = self.amount \* dividend['net\_amount']

payment\_owed = zp.dividend\_payment(out)

return payment\_owed

data\_source.py

def apply\_mapping(self, raw\_row):

"""

Override this to hand craft conversion of row.

"""

if 'price' in raw\_row:

event\_type = DATASOURCE\_TYPE.TRADE

elif 'delist' in raw\_row:

event\_type = DATASOURCE\_TYPE.DELIST

elif 'payment\_sid' in raw\_row:

event\_type = DATASOURCE\_TYPE.DIVIDEND

else:

event\_type = DATASOURCE\_TYPE.CUSTOM

15. update history with NaN

Scripts: DBProxy.py

Type: edit

Details: prevent the current bar data being stale. Every date, we allow events with NaN price and volume into the update\_universe function so the history digest panel can be updated correctly instead of being updated using the last observation.

Details: