datap Specification

Christoph Glur

2016-04-21

Contents

Example **10**

> "If I could do it all again, I'd be a plumber."
>
> – Albert Einstein

Preliminary Remarks

Document Version

- datap version: 0.1
- Document Version: 0.1
- Date: 2016-04-21
- License:

Scope

datap is a YAML format to define configurable, modular data processes. datap configurations can be used to acquire, pre-process, quality-assure, and merge data.

datap is language neutral.

In practice, each datap setup will consist of the following elements:

1. One or more datap configuration files.
2. One or more code libraries in the programming language of your choice. These libraries do the actual units of work.
3. A datap interpreter, in the programming language of your choice. The interpreter parses the configuration file, and maps processing steps defined in (1) to actual library functions available in (2).

This document is about the first part only: the datap configuration files.

Syntax Description Conventions

In this document, the datap syntax is described using the following conventions:

- `>`: a reference to a specific datap element
- `[]`: optional elements
- `$`: replace the following string with an appropriate name
- `n*`: repeat the element n times
- `|`: or

datap Syntax

context

A datap `>context` is defined in a single YAML document. A YAML document can contain at most one `>context`.

A `>context` spans a tree whose nodes are each one the following types of *joints*:

- `>tap`: entry point to data, can have parameters
- `>structure`: organise taps into hierarchies
- flow control:
  - `>pipe`: combine joints serially
  - `>junction`: combine multiple joints into one
- data processing:
  - `>processor`: unit of work (data acquisition and pre-processing)
  - `>factory`: functional programming construct
- error handling:
  - `>warning`
  - `>error`

The flow of data is from leafs towards the root, and ends at a `>tap`. Thus, each sub-tree below a `>tap` defines the processing steps of a `>tap`. We use the term *upstream* to denote joints that are in a sub-tree relative to a given joint. We use *downstream* to denote joints that are in the joint's ancestry.

variables

Variables can be defined in a `>structure`, `>tap`, `>pipe`, and `>junction` in a given `>context`.

A `>variables` section is an *associative list*, called "variables". Each variable is an entry in that list, with the *key* defining the variable *name*, and the *value* defining the variable *value*:

```
>structure|>tap|>pipe|>junction
  variables:
    n* $variableName: $value
```

The names of *special variable references* cannot be used as variable name (namely: "inflow", "inflowfun", "context")

Example:

```
Closing Prices:
  type: structure
  variables:
    series: Close
    startDate: 2000-01-01
```

You can overwrite a variable value in an upstream joint.

reference

variable and *parameter* reference

A `>reference` has an *@* prefix, and refers to a downstream* variable*, a* parameter*, a* special variable reference*, or a* macro**.*

You can use a `>reference` in a *parameter*, an *argument*, or in another *variable*.

```
>parameters|>arguments|>variables:
  $name: @$variableReferenceName
```

Or, for unnamed `>arguments`:

```
>arguments:
  - @$variableReferenceName
```

For example:

```
AAPL:
  type: tap
  variables:
    #variable reference
    #maxNaRatioDefault must be defined upstream
    maxNaRatio: '@maxNaRatioDefault'
    yahooSymbol: AAPL
    quandlCode: 'YAHOO/AAPL'
  pipe: *QYPipe
```

special variable reference

The following variable references can be used without defining the variables downstream:

- `@inflow`
- `@inflowfun`
- `@context`

They are *reserved words* and cannot be used as variable names.

@inflow

The *@inflow* reference refers to the output of the upstream joints. For a *pipe*, there is a single upstream joint. For a *junction*, there can be more than one. In that case, the *@inflow* refers to the set of upstream outputs.

Example:

```
MinLength:
  type: error
  function: MinLength
  arguments:
    timeseries: '@inflow'
    minLength: 10
```

@inflowfun

The *@inflowfun* reference refers to the upstream joints. This is particularly useful in connection with *factory* joints.

Example:

```
Cache:
  type: factory
  function: Cache
  arguments:
    f: '@inflowfun'
    timeout: 3600
```

@context

The *@context* reference refers to its surrounding `>context`.

It is useful to source data from within a `>context`, and to re-use it as an input into another `>tap`.

For example:

```
Tap:
  type: processor
  function: Tap
  arguments:
    context: '@context'
    tapPath: 'Closing Prices/Indices/SPX'
```

Macro references

A *macro* is a custom function that is interpreted by the *datap interpreter*, and whose return value is substituted into the macro reference dynamically at call-time of the `>tap`.

```
@$macroName(n* $parameterName[,])
```

For example, the datapR interpreter provides a macro *Today*, taking no arguments. Here, it is used to make sure that the *default argument* for the *endDate* parameter of the *Ones* `>tap` is set to today, dynamically at call-time of the `>tap`:

```
Ones:
  type: tap
  parameters:
    startDate: 2000-01-01
    endDate: '@Today()'
```

attributes

Attributes can contain information and/or meta data that is not part of the datap processing. For example, you can store a long name, description, etc. The datap interpreter may then provide additional functionality, e.g. to find a `>tap` by attribute.

```
>pipe|>junction|>processor|>factory|>warning|>error|>structure
  n* $attributeName: $value
```

> Consequentially, attributes are any key value pair for which the key name is not "parameters" or "variables". Also, an attribute cannot be a named associative list, otherwise it would be interpreted as a structure.

Joints

Joints are the building blocks of any datap configuration, as explained in the *Context* section.

structure

`>structure` joints fulfil two purposes:

- they define a hierarchy of other joints, especially `>tap`
- they provide a scope to `>variables`

In terms of data processing, structures are of no relevance.

```
[>structure]
  $structureName:
    type: structure
    [>attributes]
    [>variables]
    n* >structure|>tap
```

Consequentially:

- a structure may never be upstream from a `>tap`
- a structure has no other recognizable type declaration than being a named associative list. Thus, any named associative list inside a structure is itself a structure.
- a pipe may be defined directly on a structure, without a tap. Such a pipe will not be accessible through the context, and its only purpose is to define a re-usable module

tap

A `>tap` defines an entry point to specific data, within a context.

Conceptually, you can think of a tap as a public function: when you open a tap (think "call the function"), data pours out (think: "data is returned as an output/return value").

```
[>structure]
  $tapName:
    type: tap
    [>attributes]
    [>variables]
    [>parameters]
    >pipe|>junction|>processor
```

There are only `>structure` joints downstream from a tap. There are no other `>tap` joints upstream from a tap.

parameters

A `>parameter` allows a user to provide an argument when calling a `>tap`.

A `>tap` may have 0 to n parameters.

Parameters may have *default arguments*.

```
>tap
  parameters:
    n* $parameterName: [$defaultArgument]
```

For example:

```
AAPL: #tap name
  type: tap
  #attributes
  description: Apple Inc. Stock
  used by: chris
  #parameters
  parameters:
    startDate: 2000-01-01
    endDate: @Today()
    includeWeekends:
  #upstream
  pipe: *Quandl
```

processor

A `>processor` defines a unit of work, such as data acquisition and pre-processing.

```
[>structure]
  $tapName:
    type: tap
    [>attributes]
    [>variables]
    [>parameters]
    >pipe|>junction|>processor
```

function

A datap `>function` is a directive to the datap interpreter how a `>processor`, `>error`, or `>warning` is mapped to an actual function in the actual code library.

```
>processor|>error|>warning
  function: $functionName
```

> Without an interpreter and a code library, the functionName has no semantic. It is just a name!

arguments

The `>arguments` section define what arguments will be passed to a function.

The arguments can be *named* or *unnamed*:

```
>processor|>error|>warning
  arguments:
    n* - $argument | n* $parameterName: $argument
```

Example with named arguments:

```
DownloadQuandl:
  type: processor
  function: Quandl::Quandl
  arguments:
    code: '@quandlCode'
    type: xts
```

Example with unnamed arguments:

```
DownloadQuandl:
  type: processor
  function: Quandl::Quandl
  arguments:
    - '@quandlCode'
    - xts
```

error and *warning*

`>error` and `>warning` joints allow testing the results of the upstream `>processor` joint.

`>error` and `>warning` joints are pass-through: the downstream @inflow and @inflowfun variable references the joint's upstream joint.

An `>error` condition is a directive to the interpreter to stop execution and display an error message. A `>warning` condition is a directive to continue execution, and display a warning message.

```
>pipe
  $errorName:
    type: error
    [>attributes]
    >function
    [>arguments]

>pipe
  $warningName:
    type: warning
    [>attributes]
    >function
    [>arguments]
```

Example:

```
MinLength:
  type: error
  function: MinLength
  arguments:
    timeseries: '@inflow'
    minLength: 10
```

factory

`>factory` adds functional programming elements to datap.

A `>factory` is similar to a `>processor`. The difference is that:

1. a factory's `>function` is executed only once, at `>context` **creation time** (and not at `>tap` **call time**)
2. the result of the `>function` is expected to be itself a `>function`. That `>function` will then be invoked at `>tap` call time.

```
>pipe
  $factoryName:
    type: factory
    [>attributes]
    >function
    [>arguments]
```

Example:

```
Cache:
  type: factory
  function: Cache
  arguments:
    f: '@inflowfun'
    timeout: 3600
```

Interpretation: The *function* of the upstream joint is passed into the Cache function as its *f* argument. *Cache* is expected to be a function factory that returns, as an output a memoised version of @inflowfun.

pipe

A `>pipe` joint lets you arrange a number of upstream joints sequentially.

```
>tap|>pipe|>junction|>module
  $pipeName:
    type: pipe
    [>attributes]
    [>variables]
    n* >pipe|>junction|>processor|>factory|>warning|>error
```

For example, the following `>pipe` first checks if the number of NAs in a series is below an inacceptable threshold (*NA Ratio*), then it backfills missing values (*Fill NAs*):

```
NA handling: &NaHandling
  type: pipe
  Fill NAs:
    type: processor
    function: zoo::na.locf
    arguments:
      object: '@inflow'
  NA Ratio:
    type: warning
    function: NaRatio
    arguments:
      timeseries: '@inflow'
      variable: '@series'
      maxRatio: '@maxNaRatio'
```

junction

A `>junction` merges multiple upstream joints into a single stream.

Unlike the `>pipe`, the `>junction` has a `>function`, which is a directive how to merge the upstream joints.

```
>pipe|>junction|>tap|>module
  $junctionName:
    type: junction
    [>attributes]
    [>variables]
    >function
    [>arguments]
    n* >pipe|>junction|>processor
```

module

Modularization is achieved with YAML *anchors* and *references*. Modules that are not used in a tap can be put in a module section.

```
[>module]
  $moduleName:
    type: module
    [>attributes]
    n* >pipe|>junction|>module
```

For example:

```
modules:
  type: module
  #this module has no tap
  #it only serves as anchors for other taps
  NA handling: &NaHandling
    type: pipe
    Fill NAs:
      type: processor
      function: zoo::na.locf
      arguments:
        object: '@inflow'
    NA Ratio:
      type: warning
      function: NaRatio
      arguments:
        timeseries: '@inflow'
        variable: '@series'
        maxRatio: '@maxNaRatio'
```

Example

```
modules:
  type: module
```

```
#these modules have no tap
#they only serve as anchors for other taps
NA handling: &NaHandling
  type: pipe
  Fill NAs:
    type: processor
    function: zoo::na.locf
    arguments:
      object: '@inflow'
  NA Ratio:
    type: warning
    function: NaRatio
    arguments:
      timeseries: '@inflow'
      variable: '@series'
      maxRatio: '@maxNaRatio'
Quandle and Yahoo download: &QYPipe
  type: pipe
  attributes:
    description: |
      This defines a reusable process
      to download prices from Quandl,
      overwrite missings with data
      from Yahoo, do NA handling and
      more. Return value: an xts object
  Cache:
    type: factory
    function: Cache
    arguments:
      f: '@inflowfun'
      timeout: 3600
  GetSeries:
    type: processor
    function: magrittr::use_series
    arguments:
      a: '@inflow'
      b: '@series'
  NAs: *NaHandling
  Regularize:
      type: processor
      function: Regularize
      arguments:
        xts: '@inflow'
  Combine:
    type: junction
    function: Combine
    arguments:
      listofxts: '@inflow'
    Quandl:
      type: pipe
      MinLength:
        type: error
        function: MinLength
        arguments:
```

```
            timeseries: '@inflow'
            minLength: 10
        DownloadQuandl:
          type: processor
          function: Quandl::Quandl
          arguments:
            code: "@quandlCode"
            type: xts
      Yahoo:
        type: pipe
        MinLength:
          type: warning
          function: MinLength
          arguments:
            timeseries: '@inflow'
            minLength: 10
        SetNames:
          type: processor
          function: SetNames
          arguments:
            x: '@inflow'
            names: [Open, High, Low, Close, Volume, 'Adjusted Close']
        DownloadYahoo:
          type: processor
          function: quantmod::getSymbols
          arguments:
            Symbols: "@yahooSymbol"
            auto.assign: FALSE
## taps
Closing Prices:
  type: structure
  variables:
    series: Close
    maxNaRatioDefault: 0.25
  Indices:
    type: structure
    SPX:
      type: tap
      attributes:
        longname: "S&P 500 daily close"
        description: |
          Quandl, fill missing values with Yahoo.
          Backfill weekends and holidays.
          Cache for an hour.
          Warn if newest value older than a day.
      parameters:
        #parameterName: defaultArgument
        dteRange: 1990-01-01/2010-01-01
      variables:
        #variableName: value
        series: '@series'
        maxNaRatio: '@maxNaRatioDefault'
        yahooSymbol: "^GSPC"
        quandlCode: "YAHOO/INDEX_GSPC"
```

```
      Pipe:
        type: pipe
        DateRange:
          type: processor
          function: magrittr::extract
          arguments:
            - '@inflow'
            - '@dteRange'
        QYPipe: *QYPipe
  Single Stocks:
    type: structure
    AAPL:
      type: tap
      attributes:
        longname: "Apple"
        description: |
          Apple stock price
      variables:
        series: '@series'
        maxNaRatio: '@maxNaRatioDefault'
        yahooSymbol: "AAPL"
        quandlCode: "YAHOO/AAPL"
      pipe: *QYPipe
    MSFT:
      type: tap
      attributes:
        longname: "Microsoft"
      variables:
        series: '@series'
        maxNaRatio: 0.0
        yahooSymbol: "MSFT"
        quandlCode: "YAHOO/MSFT"
      pipe: *QYPipe
  Fabricated:
    type: structure
    variables:
      startDateDefault: 1990-01-01
    Ones:
      type: tap
      parameters:
        startDate: '@startDateDefault'
        endDate: '@Today()'
      GetOnes:
        type: processor
        function: Ones
        arguments:
          startDate: '@startDate'
          endDate: '@endDate'
          colname: '@series'
Technical Indicators:
  type: structure
  MATap:
    type: tap
    attributes:
```

```
    longname: "Moving Average"
    description: |
      This demonstrates how to create
      taps based on other taps
  parameters:
    tapPath:
    periods: 10
    ...:
  pipe:
    type: pipe
    SMA:
      type: processor
      function: TTR::SMA
      arguments:
        x: '@inflow'
        'n': '@periods'
    Tap:
      type: processor
      function: Tap
      arguments:
        context: '@context'
        tapPath: '@tapPath'
        ...: '@...'
MA:
  type: tap
  attributes:
    longname: "Moving Average"
    description: |
      This demonstrates how to create
      taps without a data source. Use it
      as a function, with any xts as an
      input!
  parameters:
    series:
    periods: 10
  Transform:
    type: processor
    function: TTR::SMA
    arguments:
      x: '@series'
      'n': '@periods'
```