Flatline Documentation

Release 1.0

The BigML Team

May 02, 2017

Contents

Flatline is a lispy language for the specification of values to be extracted or generated from an input dataset, using a finite sliding window of input rows.

In BigML, it is used either as a row filter specifier or as a field generator.

In the former case, the input consists of dataset rows on which a single, boolean expression is computed, and only those for which the result is true are kept in the output dataset.

When used to generate new datasets from given ones, a list of Flatline expressions is provided, each one generating either a value or a list of values, which are then concatenated together to conform the output rows (each value representing therefore a field in the generated dataset).

CHAPTER 1

Flatline user manual

S-expressions vs. JSON

Flatline expressions in this manual use its lisp-like syntax, based on symbolic expressions or *sexps*. When sending them to BigML via our API, you can also use their JSON representation, which is trivially obtained by using JSON lists for each paranthesised sexp. For instance:

```
(if (< (f "a") 3) 0 4) => ["if", ["<", ["f", "a"], 3], 0, 4]
```

Literal values

Constant numbers, symbols, booleans and strings, using Java/Clojure syntax are valid expressions.

Examples:

```
1258
2.349
this-is-a-symbol
"a string"
true
false
```

Counters

While running over an input dataset, Flatline keeps track of the (zero-based) number of the input row that's being used, which can be accessed with the function `row-number`, which takes no arguments:

```
(row-number) => current input row (0-based)
```

A typical use of this function is to generate a unique identifier for each row. The row number will start at 0 unless you skip some rows of the input dataset, and increase by one on each new row (unless you're specifying a input row step when generating a dataset).

Field accessors

Field values

Input field values are accessed using the `field` operator:

```
(field <field-designator> [<shift>])
<field-designator> := field id | field name | column_number
<shift> := integer expression
```

where `<field-designator>` can be either the identifier, name or column number of the desired field, and the optional `<shift>` (an integer, defaulting to 0) denotes the offset with respect to the current input row.

So, for instance, these sexps denote field values extracted from the current row:

```
(field 0)
(field 0 0)
(field "000004")
(field "a field name" 0)
```

while

```
(field "000001" -2)
```

denotes the value of the cell corresponding to a field with identifier "000001" two rows *before* the current one. Positive shift values denote rows after the current one.

```
(field "a field" 3)
(field "another field" 2)
```

For convenience, and since `field` is probably going to be your most often user operator, it can be abbreviated to `f`:

```
(f "000001" -2)
(f 3 1)
(f "a field" 23)
```

We also provide a predicate, `missing?`, that will tell you whether the value of the field for the given row (taking into account the shift, if any) is a missing token:

```
(missing? <field-designator> [<shift>])
```

E.g.:

```
(missing? "species")
(missing? "000001" -2)
(missing? 3 1)
(missing? "a field" 23)
```

will all yield boolean values. For backwards compatibility, `missing` is an alias for `missing?`.

Randomized field values

There are two Flatline functions that will let you generate a random value in the domain of a given field, given its designator:

```
(random-value <field-designator>)
(weighted-random-value <field-designator>)
```

e.g.

```
(random-value "age")
(weighted-random-value "000001")
(weighted-random-value 3)
```

Both functions generate a value with the constrain that it belongs to the domain of the given field, but while `random-value` uses a uniform probability of the field's range of values, `weighted-random-value` uses de distribution of the field values (as computed in its histogram) as the probability measure for the random generator.

These two functions work for numeric, categorical and text fields, with generated values satisfying:

- For numeric fields, generated values are in the interval `[(minimum <fid>), (maximum <fid>)]`
- For categorical fields, generated values belong to the set `(categories <fid>)`
- For text fields, we generate terms in the field's tag cloud (generated values correspond to single terms in the cloud).
- Datetime **parent** fields are not supported, since they don't have a defined distribution: you can use any of their numeric children for generating values following their distributions.

A common use of these functions is replacing missing values with random data, which in Flatline you could write as, say:

```
(if (missing? "00000") (random-value "000000") (f "000000"))
```

We provide a shortcut for those common operations with the functions `ensure-value` and `ensure-weighted-value`:

```
(ensure-value <fdes>) :=
  (if (missing? <fdes>) (random-value <fdes>) (field <fdes>))

(ensure-weighted-value <fdes>) :=
  (if (missing? <fdes>) (weighted-random-value <fdes>) (field <fdes>))
```

We them, our example above can be simply written as:

```
(ensure-value "000000")
```

or, if you want that the generated random values follow the same distribution as the field "000000":

```
(ensure-weighted-value "000000")
```

Normalized field values

For numeric fields, it's often useful to normalize their values to a standard interval (usually [0, 1]). To that end, you can use the Flatline primitive `normalize`, which takes as arguments the designator for the field you want to normalize and, optionally, the two bounds of the resulting interval:

```
(normalize <id> [<from> <to>])
=> (+ from (* (- to from)
              (/ (- (f id) (minimum id))
                 (- (maximum id) (minimum id)))))
```

For instance:

```
(normalize "000001") ;; = (normalize "000001" 0 1)
(normalize "width" -1 1)
(normalize "length" 8 23)
```

As shown in the formula above, `normalize` linearly maps the minimum value of the field to `from` (0 by default) and the maximum value to `to` (1 by default).

Besides this linear normalization, it's also common to standardize numeric data values by mapping them to a gaussian, according to the equation:

```
x[i] -> (x[i] - mean(x)) / standard_deviation(x)
```

or, in flatline terms:

```
(/ (- (f <id>) (mean <id>)) (standard-deviation <id>))
```

This normalization function is called the Z score, and we provide it as the function `z-score`:

```
(z-score <field-designator>)
```

E.g.:

```
(z-score "000034")
(z-score "a numeric field")
(z-score 23)
```

As with `normalize`, the field used must have a numeric optype.

Vectorized categorical or text fields

It may be useful to convert categorical or text fields to numeric values for models which accept only numeric data as input. This can be accomplished with the Flatline primitive `vectorize`:

```
(vectorize <field-designator> [<max-fields>])
```

For categorical fields, the output is a binary indicator vector. In other words, it is a list of numeric fields, one per possible categorical value, and for each instance, the numeric field corresponding to the category of that instance will have a value of `1`, whereas the remaining numeric fields will have a value of `0`.

For text fields, the output is a list of numeric fields, each corresponding to a term in the field's tag cloud. The value of each field is the number of times that term appears in that instance.

A numeric expression or literal can be passed as an optional second argument to limit the number of generated fields to the *n* most frequent categories or text terms.

Field properties

Summary properties

Field descriptors contain lots of properties with metadata about the field, including its summary. These propeties (when they're atomic) can be accessed via `field-prop`:

```
(field-prop <type> <field-descriptor> <property-name> ...)
<type> := string | numeric | boolean
```

For instance, you can access the name for field "00023" via:

```
(field-prop string "00023" name)
```

or the value of the nested property missing_count inside the summary with:

```
(field-prop numeric "00023" summary missing_count)
```

We provide several shortcuts for concrete summary properties, to save you typing:

```
(maximum <field-designator>)
(mean <field-designator>)
(median <field-designator>)
(minimum <field-designator>)
(missing-count <field-designator>)
(population <field-designator>)
(sum <field-designator>)
(sum-squares <field-designator>)
(standard-deviation <field-designator>)
(variance <field-designator>)

(preferred? <field-designator>)

(category-count <field-designator> <category>)
(bin-center <field-designator> <bin-number>)
(bin-count <field-designator> <bin-number>)
```

As you can see, the category and count accessors take an additional parameter designating either the category (a string or order number) and the bin (a 0-based integer index) you refer to:

```
(category-count "species" "Iris-versicolor")
(category-count "species" (f "000004"))
(bin-count "age" (f "bin-selector"))
(bin-center "000003" 3)
(bin-center (field "field-selector") 4)
```

Discretization of numeric fields

A simple way to discretize a numeric field is to assign a label to each of a finite set of segments, defined by a sequence of upper bounds. For instance:

```
(let (v (f "age"))
  (cond (< v 2) "baby"
        (< v 10) "child"
        (< v 20) "teenager"
        "adult"))
```

Flatline provides a shortcut for the above expression via its `segment-label` primitive:

```
(segment-label "000000" "baby" 2 "child" 10 "teenager" 20 "adult")
```

As you can see, the first argument is the field designator (as usual, a name, column number or identifier), followed by alternating labels and upper bounds. More generally:

```
(segment-label <fdes> <l1> <b1> ... <ln-1> <bn-1> <ln>)
<l1> ... <ln> strings, <b1> ... <bn-1> numbers
=> (cond (< (f <fdes>) <b1>) <l1>
         (< (f <fdes>) <b2>) <l2>
         ...
         (< (f <fdes>) <bn-1>) <ln-1>
         <ln>)
```

The alternating labels and bounds must be constant strings and numbers. If you want to use segments of equal length between the minimum and maximum value of the field, you can omit the upper bounds and give simply the list of labels, e.g.

```
(segment-label 0 "1st fourth" "2nd fourth" "3rd fourth" "4th fourth")
```

which would be equivalent to:

```
(let (max (maximum 0)
      min (minimum 0)
      step (/ (- max min) 4))
  (segment-label 0 "1st fourth" (+ min step)
                   "2nd fourth" (+ min step step)
                   "3rd fourth" (+ min step step step)
                   "4th fourth"))
```

or, in general:

```
(segment-label <fdes> <l1> ... <ln>) with  <l1> ... <ln> strings

=> (let (min (minimum <fdes>)
         step (- (maximum <fdes>) min)
         shift (- (f <fdes>) min))
     (cond (< shift step) <l1>
           (< shift (* 2 step)) <l2>
           ...
           (< shift (* (- n 1) step)) <ln-1>
           <ln>))
```

Items and itemsets

A common operation on fields of optype *items* is to check whether they contain a list of items. That can be used, for instance, to filter the rows of a dataset that satisfy a given association rule, but calling `contains-items?` with the list of items in the antecedent and consequent of the desired rule.

```
(contains-items? <field-designator> <item_0> ... <item_n>)
;; with <item_i> of type string for i in [0, n]
```

The `contains-items` primitive takes as first argument the descriptor of the field we want to check (which must have optype items), followed by the one or more items we want to check, which must all have type string. For instance, the predicate:

```
(contains-items? "000000" "blue" "green" "darkblue")
```

will filter the rows whose first column satisfies the association rule `blue, green -> darkblue`.

It is also possible to check whether an items field contains *only* the given list of items (in any order), using `equal-to-items?`, which works exactly as `contains-items?` except for the fact that it's exclusive:

```
(equals-to-items? <field-designator> <item_0> ... <item_n>)
;; with <item_i> of type string for i in [0, n]
```

Field population, percentiles &co for numeric fields

Besides direct readings from the field summaries, there exist other derived statistical properties available as Flatline functions. In particular, these are the ones related to population percentiles and their distribution for *numeric* fields:

```
(percentile <field-designator> <fraction>)    ;; fraction in [0.0, 1.0]
(within-percentiles? <field-designator> <lower> <upper>)
(population-fraction <field-designator> <sexp>)
(percentile-label <field-designator> <label-0> ... <label-n>)
```

The first one, `percentile`, gives you the value that a numeric field must have in order to be in the given population fraction. Thus, you could use, for instance, the following predicate in a filter to remove outliers:

```
(<= (percentile "age" 0.5) (f "age") (percentile "age" 0.95))
```

We provide syntactic sugar for the above expression via `within-percentiles?`:

```
(within-percentiles? "age" 0.5 0.95)
```

Related to percentile is `population-fraction`, which, given a field identifier and a value, computes the number of instances of this field whose value is less than the given one. As with the case of `percentile`, the designated field must be numeric.

Finally, `percentile-label` computes the percentile the input value belongs to and generates the label you provided. For instance, this generator:

```
(percentile-label "000023" "1st" "2nd" "3rd" "4th")
```

will generate the label "1st" if the value of the field 000023 is in the first population "quartile" (since we're providing 4 labels, we use 4 segments), "2nd" to the second, etc. The sexp above is equivalent to:

```
(cond (within-percentiles? "000023" 0 0.25) "1st"
      (within-percentiles? "000023" 0.25 0.5) "2nd"
      (within-percentiles? "000023" 0.5 0.75) "3rd"
      "4th")
```

and, as you see, it easily generalizes to any number of labels: if you had provided 5 labels we'd be computing "quintiles"; had them been 10, the labels would correspond to "deciles," and so forth. As with all functions in this section, the target field must be numeric.

Note that we're using scare quotes around quartile, quintiles, etc. above. That's because `percentile-label` will assign to each value the label of the lowest percentile it belongs to, and therefore, it won't really discretize your variable by exact quantiles: if the population is skewed around a value, so it'll be the resulting labels' population.

Strings and regular expressions

Coercion and substrings

Any value can be coerced to a string using the `str` operator, which will also concatenate the corresponding strings when called with more than one argument:

```
(str <sexp0> ...)
```

For instance:

```
(str 1 "hello " (field "a"))   ;; =>  "1hello <value of field a>"
(str "value_" (+ 3 4) "/" (name "000001"))  ;; => "value_7/a"
```

It is also possible to take a substring of a string value using the `subs` operator:

```
(subs <string> <start> [<end>])
<start> in [0 (length <string>))
<end> in (0 (length <string>)]
```

It returns the substring of `<string>` beginning at start inclusive, and ending at end (defaults to length of string), exclusive.

String utilities

The number of characters in a string value is given by `length`:

```
(length <string>)
```

e.g.

```
(length "abc") => 3
(length "") => 0
```

Note that the length of a missing value is a missing value, not zero.

The primitive `levenshtein` computes, as an integer, the distance between two given string values:

```
(levenshtein <str-sexp0> <str-sexp1>)
```

Arbitrary arguments are allowed, provided they're strings:

```
(levenshtein (f 0) "a random string")
(if (< (levenshtein (f 0) "bluething") 5) "bluething" (f 0))
```

You can also compute the number of times a word appears in a given string by means of the `occurrences` function. It takes an input string and the term to look for as mandatory parameters, and, optionally, a language code, and a boolean controlling case sensitivity:

```
(occurrences <string> <term> [<case-insensitive?> <lang>])
<case-insensitive?> := true | false (defaults to false)
<lang> := "en" | "es" | "ca" | "nl" | "none" (defaults to "none")
```

By default, terms matching is case sensitive and exact. The optional third argument is a boolean flag to turn on case insensitivity. Finally, if you provide a fourth constant argument specifying one of the known languages (English, Spanish, Catalan or Dutch), words are compared using their stems (e.g., in English, "day" and "days" will be considered the same term).

For instance:

```
(occurrences "howdy woman, howdy" "howdy") => 2
(occurrences "howdy woman" "Man" true) => 0
(occurrences "howdy man" "Man" true) => 1
(occurrences "hola, Holas" "hola" true "es") => 2
```

Hashing functions

There are several hashing functions available: `md5`, `sha1` and `sha256`. These functions act on the stream of bytes of their input string and return a string representing the bytes that the cryptographic digest they name produces, in their hexadecimal representation:

```
(md5 <string>) => string of length 32
(sha1 <string>) => string of length 40
(sha256 <string>) => string of length 64
```

e.g.

```
(md5 "a text") => "b229386ec4627869d2c71b7df3c9600a"
(sha1 "a text") => "7081f2babbafff16b4bae16282859c844baa14ef"
(sha256 "") =>
"e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"
```

As shown, the returned strings use charaters in `[0-9a-f]` to represent the values of the output bytes: md5 produces 16 bytes (128 bits), sha-1 produces 20 bytes (160 bits) and sha-256 produces 32 bytes (256 bits).

Regular expression matching

The `matches?` function takes a regular expression as a string and a form evaluating to a string and returns a boolean telling you if the latter matches the former.

```
(matches? <string> <regex-string>)  => boolean
<regex-string> := a string form representing a regular expression
<string> := a string expression to be tested against the regexp
```

Regular expressions follow the Perl and Java syntax and extensions (see for instance this summary), including flags modifiers such as "(?i)" to turn on case-insensitive mode.

For instance, to check if the field "name" contains the word "Hal" anywhere, you could use:

```
(matches? (field "name") ".*\\sHal\\s.*")
(matches? (field "name") "(?i).*\\shal\\s.*")
```

where the second form performs case-insensitive pattern matching.

It's possible to use non-constant string values for the regular expression, but take into account that any special character in the string will be treated as such when it's converted to a regular expression. If what you want is to match literally the contents of a field, use `re-quote`:

```
(re-quote <string>)  => regexp that matches <string> literally
```

and then you can write things like:

```
(if (matches? (f "result") (re-quote (f "target"))) "GOOD" "MISS")
```

and you can use the string concatenation operator `str` to construct regular expressions strings out of smaller pieces:

```
(matches? (f "name") (str "^" (re-quote (f "salutation")) "\\s *$"))
```

Regular expression search and replace

Given a string expression, you can substitute matches of a given regexp by a given replacement string using `replace` and `replace-first`:

```
(replace <string> <regexp> <replacement>)
(replace-first <string> <regexp> <replacement>)
```

e.g.:

```
(replace "Target string ss" "\\Ws" "S") => "TargetStringSs"
```

The replacement is literal, except that "$1", "$2", etc. in the replacement string are substituted with the string that matched the corresponding parenthesized group in the pattern. If you want them to appear literally in the replacement string, just use "\$1" and the like.

For example:

```
(replace "Almost Pig Latin" "\\b(\\w)(\\w+)\\b" "$2$1ay")
 => "lmostAay igPay atinLay"
```

While `replace` replaces all occurrences of the regular expression, `replace-first` stops after the first match:

```
(replace-first "swap first two words" "(\\w+)(\\s+)(\\w+)" "$3$2$1")
 => "first swap two words"
```

Text analysis

Flatline provides a primitive function, `language`, that tries to detect the language of a given string value. It returns the ISO 639 code of the detected language, as a string.

```
(language <string>) => <ISO 639 string code>
```

For instance:

```
(language "this is an English phrase") => "en"
```

Note that language detectors will do in general a very poor job for short texts, and that we currently limit the set of detected languages to those used in BigML's text analysis facility (English, Spanish, Catalan or Dutch as of this writing, represented as "en", "es", "ca" and "nl", respectively.)

Relational operators and equality

You can compare numeric and datetime values with any of the relational operators <, <=, >, and >=, which can be applied to two or more arguments and always result in a boolean value. For example:

```
(< (field 0) (field 1))
(<= (field 0 -1) (field 0) (field 0 1))
(> (field "date") "07-14-1969")
(>= 23 (f "000004" -2))
```

The equality (=) and inequality (!=) operators can be applied to operators of any kind:

```
(= "Dante" (field "Author"))
(= 1300 (field "Year"))
(= (field "Year" -2) (field "Year" -1) (field "Year"))
(!= (field "00033" -1) (field "00033" 1))
```

Comparing numerical values can be tricky, especially when they're the result of mathematical operations, but Flatline makes an effort to be sensible and considers things like 1 and 1.0 equal (for numeric values, it actually uses Clojure's == operator); but of course it cannot fix rounding errors or the like for you!

Logical operators

The basic logical connectives `and`, `or` and `not`, acting on boolean values, are available, with their usual meanings.

```
(and (= 3 (field 1)) (= "meh" (f "a")) (< (f "pregnancies") 5))
(not true)
```

For additional convenience, `and` and `or` can be applied to lists (described below):

```
(and (list <sexp0> ... <sexpn>)) := (and <sexp0> ... <sexpn>)
(or (list <sexp0> ... <sexpn>)) := (or <sexp0> ... <sexpn>)
```

Arithmetical operators

The usual arithmetical operators `+`, `-`, `*` and `/` taking any number of arguments (or zero, for+and*') are available. Of course their operands must evaluate to a numeric value; otherwise, the result will be nil, representing a missing value.

When not coerced, the result of the `/` operator has type `double`. If needed, you can transform it to an integer via the coercion function `integer` or use instead the integer division operator `div` (see below).

Numerical coercions

You can coerce arbitrary values to explicit numeric types. When the input sexp is a string (or a category name), we try to parse it as a number and afterwards perform a pure numerical coercion if needed. Boolean values are mapped to 0 (false) and 1 (true).

```
(integer <sexp>)
(real <sexp>)
```

If the input value cannot be coerced to a number the result is a missing value.

Mathematical functions

We provide a host of mathematical functions:

```
(max <x0> ... <xn>)
(min <x0> ... <xn>)

(abs <x>)      ;; Absolute value
(mod <n> <m>) ;; Modulus
(div <n> <m>) ;; Integer division (quotient)
(sqrt <x>)
(pow <x> <n>)
(square <x>)  ;; (* <x> <x>)

(ln <x>)       ;; Natural logarithm
(log <x>)      ;; Base-2 logarithm
(log10 <x>)    ;; Base-10 logarithm
(exp <x>)      ;; Exponential

(ceil <x>)
(floor <x>)
(round <x>)

(cos <x>)      ;; <x> := radians
(sin <x>)      ;; <x> := radians
(tan <x>)      ;; <x> := radians

(to-radians <x>) ;; <x> := degrees
(to-degrees <x>) ;; <x> := radians

(acos <x>)
(asin <x>)
(atan <x>)

(cosh <x>)
(sinh <x>)
(tanh <x>)
```

As well as two primitives for generating random numbers:

```
(rand)             ;; a random double in [0, 1)
(rand-int <n>)     ;; a random integer in [0, n) or (n, 0]
```

Currently there's no way of specifying the seed used for random number generation, but it's coming shortly to a selected data generation language very near to you.

Regression

It's also possible to compute the slope, intercept and Pearson coeffient of the linear regression of a set of points given as a list of alternating x and y coordinates:

```
(linear-regression <x0> <y0> <x1> <y1> ... <xn> <yn>)
   => (<slope> <intercept> <pearson>) ;; 3 double values
```

e.g.

```
(linear-regression 1 1 2 2 3 3 4 4) => (1.0 0 1.0)
(linear-regression 2.0 3.1 2.3 3.3 24.3 45.2) => (1.89 -0.87 0.9999)
```

Statistical functions

The function `chi-square-p-value` computes the p-value of a Chi-square distribution with the given number of degrees of freedom and a given cut value:

```
(chi-square-p-value <d> <x>)
  ;; => <p-value>, with <d> integer <x> a number
```

Thus, the value `x` passes the Chi-square test if the value returned by `(chi-square-p-value d x)` is less than or equal to `x`. For instance, the expression:

```
(<= (chi-square-p-value 2 (field "000000")) 0.05)
```

will compute a boolean that tells you whether the field "000000" passes a Chi-square test for two degrees of freedom with significance level 0.05.

Dates and times

Epoch fields

A numerical field can be interpreted as an *epoch*, that is, the number of **milliseconds** since 1970. Flatline provides the following functions to expand an epoch to its date-time components:

```
(epoch-year <n>)
(epoch-month <n>)
(epoch-day <n>)
(epoch-weekday <n>)
(epoch-hour <n>)
(epoch-minute <n>)
(epoch-second <n>)
(epoch-millisecond <n>)

(epoch-fields <n>)
  => (list (epoch-year <n>) (epoch-month <n>) (epoch-day <n>)
           (epoch-weekday <n>) (epoch-hour <n>) (epoch-minute <n>)
           (epoch-second <n>) (epoch-millisecond <n>))
<n> ::= numerical value
```

For instance:

```
(epoch-fields (f "milliseconds"))
(epoch-year (* 1000 (f "seconds")))
```

The epoch functions also accept negative integers, which represent dates prior to 1970.

The day of the week (given by `epoch-weekday`) is a number from 1 (Monday) to 7 (Sunday).

Datetime arithmetic

Since epochs are just integers, date arithmetic can be performed at that level by simply using Flatline's arithmetic operations.

As a convenience, if a field of type `datetime` is used in an arithmetic operation, it's automatically converted to an epoch (i.e., an integer value) for you. For instance, the two following expressions for computing the number of seconds since 1970 are equivalent:

```
(/ (f "a-datetime-string") 1000)
(/ (epoch (f "a-datetime-string")) 1000)
```

Datetime parsing

Conversely, string values representing dates can be transformed to a numerical epoch by using the `epoch` coercion function:

```
(epoch <str>)
(epoch <str> <format>)
```

If you don't specify a datetime format for parsing, we try a long list of available formats in sequence, which is less efficient than if you provide the format explicitly. Datetime format specifiers follow the well known *JodaTime* specification for datetime patterns.

For instance:

```
(epoch-fields (epoch "1969-14-07T06:00:12")) => [1969 14 07 06 00 12 0]
(epoch-hour (epoch "11~22~30" "hh~mm~ss")) => 11
```

The datetime formate pattern letters are:

```
Symbol  Meaning                      Presentation  Examples
------  -------                      ------------  -------
 G       era                          text          AD
 C       century of era (>=0)         number        20
 Y       year of era (>=0)            year          1996

 x       weekyear                     year          1996
 w       week of weekyear             number        27
 e       day of week                  number        2
 E       day of week                  text          Tuesday; Tue

 y       year                         year          1996
 D       day of year                  number        189
 M       month of year                month         July; Jul; 07
 d       day of month                 number        10

 a       halfday of day               text          PM
 K       hour of halfday (0~11)       number        0
 h       clockhour of halfday (1~12)  number        12

 H       hour of day (0~23)           number        0
 k       clockhour of day (1~24)      number        24
 m       minute of hour               number        30
 s       second of minute             number        55
 S       fraction of second           number        978
```

```
z       time zone                    text          Pacific Standard Time; PST
Z       time zone offset/id          zone          -0800; -08:00; America/Los_Angeles

'       escape for text              delimiter
''      single quote                 literal       '
```

The count of pattern letters determine the format, according to the following rules:

- Text: If the number of pattern letters is 4 or more, the full form is used; otherwise a short or abbreviated form is used if available. Thus, "EEEE" might output "Monday" whereas "E" might output "Mon" (the short form of Monday).
- Number: The minimum number of digits. Shorter numbers are zero-padded to this amount. Thus, "HH" might output "09" whereas "H" might output "9" (for the hour-of-day of 9 in the morning).
- Year: Numeric presentation for year and weekyear fields are handled specially. For example, if the count of `y` is 2, the year will be displayed as the zero-based year of the century, which is two digits.
- Month: 3 or over, use text, otherwise use number. Thus, "MM" might output "03" whereas "MMM" might output "Mar" (the short form of March) and "MMMM" might output "March".
- Zone: `Z` outputs offset without a colon, `ZZ` outputs the offset with a colon, `ZZZ` or more outputs the zone id.
- Zone names: Time zone names (`z`) cannot be parsed.

Any characters in the pattern that are not in the ranges of `['a'..'z']` and `['A'..'Z']` will be treated as quoted text. For instance, characters like `:`, `.`, `` `,#and?` `` will appear in the resulting time text even they are not embraced within single quotes.

Local bindings

You can define lexically scoped variables using the `let` special form:

```
(let <bindings> <body>)
<bindings> := (<varname0> <val0> ...  <varnamen> <valn>)
<body> := <expression with varname0 ... varnamen>
```

The binding values are evaluated sequentially and can then be referenced in the body of the let expression by their names:

```
(let (x (+ (window "a" -10 10))
      a (/ (* x 3) 4.34)
      y (if (< a 10) "Good" "Bad"))
  (list x (str (f 10) "-" y) a y))
```

As shown in the example above, value expressions can use any identifier previously defined in the same list:

```
(let (x 43
      x (+ x 1)
      y (+ x 1))
  (list x y))  =>  (44 45)
```

Finally, let expressions can nested and they can appear wherever a Flatline expression is valid:

```
(list (let (z (f 0)) (* 2 (* z z) (log z)))
      (let (pi 3.141592653589793 r (f "radius")) (* 4 pi r r)))
```

Control structures

Conditionals

The `if` operator can be applied to a boolean conditional to yield one of a couple of values, with the "else" clause being optional:

```
(if <cond> <then> [<else>])
<cond> := boolean value
```

You can use arbitrary expressions for `<cond>`, `<then>` and `<else>`, with the only restriction that `<cond>` must be a boolean value. If not provided, `<else>` defaults to a "nil" value that denotes a missing token.

```
(if (< (field "age") 18) "non-adult" "adult")

(if (= "oh" (field "000000")) "OH")

(if (> (field "000001") (mean "000001"))
    "above average"
    (if (< (field "000001") (mean "000001"))
        "below average"
        "mediocre"))
```

Flatline won't let you give `<then>` and `<else>` different types.

Another caveat is that in Flatline boolean expressions can have 3 values, namely `true`, `false` and **missing**. If the `<cond>` in an `if` expression is a missing value, **the whole expression will evaluate to a missing value**. That means that, for instance:

```
(if (< (f 0) 3) 0 1)
```

will evaluate to null (and *not* to 1) when the field 0 has a missing value. That's because the `<else>` branch is not even evaluated. Therefore:

```
(if (< (f 0) 3) 0 (if (missing? 0) 2 1))
```

will again evaluate to null when the field 0 is missing: it will *not* evaluate to 2, because the `<else>` branch is never reached. If you need to test for a missing value, the test must always come first:

```
(if (missing? 0) 2 (if (< (f 0) 3) 0 1))
```

We also provide the `cond` operator, which allows a more compact representation of a chain of nested `if` clauses:

```
(cond <cond0> <then0>
      <cond1> <then1>
      ... ...
      <default>) := (if <cond0> <then0> (if <cond1> <then1> ... <default>))
```

Conditions are checked in order, and the first one that matches provides the value of the `cond` expression. If none of the conditions is met, the expression evaluates to `<default>` or nil (missing token) if it's not provided.

For instance:

```
(cond (> (f "000001") (mean "000001")) "above average"
      (= (f "000001") (mean "000001")) "below average"
      "mediocre")
```

```
(cond (or (= "a" (f 0)) (= "a+" (f 0))) 1
      (or (= "b" (f 0)) (= "b+" (f 0))) 0
      (or (= "c" (f 0)) (= "c+" (f 0))) -1)

(cond (< (f "age") 2) "baby"
      (and (<= 2 (f "age") 10) (= "F" (f "sex"))) "girl"
      (and (<= 2 (f "age") 10) (= "M" (f "sex"))) "boy"
      (< 10 (f "age") 20) "teenager"
      "adult")
```

The same caveat with `if` regarding missing values applies to `cond`: **if any of the conditions evaluates to a missing value, the whole expression evaluates to a missing value**. Therefore, checks using `missing?` must always come first:

```
;;; CORRECT
(cond (missing? "age") 0
      (< (f "age") 10) 1
      2)

;;; INCORRECT (the missing? test is never reached)
(cond (< (f "age")) 1
      (missing? "age") 0
      2)
```

Lists

It's possible to create a list of values using the `list` operator:

```
(list <sexp-0> ...)
```

with the values any valid Flatline expression, e.g.:

```
(list (field "age")
      (field "weight" -1)
      (population "age"))

(list 1.23
      (if (< (field "age") 10) "child" "adult")
      (field 3))
```

and we also provide the classical `cons` to create a list from its head and tail, which can in turn be accessed via `head` and `tail`:

```
(cons <head> <tail>)
<tail> := list
(head <list>)
(tail <list>)
```

so that:

```
(head (cons x lst)) ==> x
(tail (cons x lst)) ==> lst
```

It's also possible to access the nth element of a list using its 0-based position index:

```
(nth <list> <pos>)
<pos> := positive integer
```

When the given position is out of bounds, the expression evaluates to nil (a missing token).

List operators

Given a list value, you can count its elements, obtain their mode and, when its values are numeric, compute the maximum minimum and average:

```
(count <list>)          ;; (count (list (f 1) (f 2))) => 2
(mode <list>)           ;; (mode (list a b b c b a c c c)) => "c"
(max <list>)            ;; (max (list -1 2 -2 0.38)) => 2
(min <list>)            ;; (min (list -1.3 2 1)) => -1.3
(avg <list>)            ;; (avg (list -1 -2 1 2 0.8 -0.8)) => 0
```

And, as we have mentioned, the arithmetic operators `+`, `-`, `*` and `/` are, like `max` and `min`, overloaded to distribute over the elements of a numeric list:

```
(+ (list x y ...)) := (+ x y ...)
(- (list x y ...)) := (- x y ...)
(* (list x y ...)) := (* x y ...)
(/ (list x y ...)) := (/ x y ...)
```

Finally, you can check whether a value appears in a list using the `in` operator:

```
(in <x> (<x0> <x1> ... <xn>))
```

which evaluates to `true` if any of the `<xi>` equals `<x>`, e.g.:

```
(in 3 (1 2 3 2)) => true
(in "abc" (1 2 3)) => false
(in (f "size") ("X" "XXL"))
```

Maps and filters

It's also possible to apply an expression template (a Flatline expression with one free variable, marked as _) to each element of a list, yielding a list of results, using the `map` primitive:

```
(map <fn> (list <a0> <a1> ... <an>))
  := (list (call <fn> <a0>) (call <fn> <a1>) ... (call <fn> <an>))
<fn> := expression template
```

An expression template is any valid Flatline expression that uses _ as a placeholder:

```
(< _ 3)
(+ (f "000001" _) 3)
(< -18 _ (f 3))
```

and when you `call` a template with an argument, a new expression is generated by the simple device of substituting the argument for _ in the template. For instance:

```
(map (* 2 _) (list (f 0 -1) (f 0) (f 0 1)))
```

expands to

```
((* 2 (f 0 1)) (* 2 (f 0)) (* 2 (f 0 1)))
```

A second common list transformation is `filter`, which allows you to apply a predicate to each element of a list and retain only those values that satisfy it:

```
(filter <fn> (list <a0> ... <an>)) := [ai | (call <fn> <ai>) is true]
```

For instance,

```
(+ (filter (< _ 3) (fields "a" "b" "c")))
```

will add those values of the fields with names a, b and c whose values are less than three.

Currently, maps and filter are implemented as macro expansions (for simplicity, and also for performance) and their second argument must therefore be a `list`, `fields` or `window` (see below) form. If needed, future versions of Flatline will provide slow real functions.

Field lists and windows

(Almost) all fields

We provide several primitives for creating lists of field values. The first one is `all`, which specifies that all input fields should be copied, without any modification. For cases where you want to copy all but a few fields, there's `all-but`, which takes as argument designators of those fields *not* to include in the list:

```
(all) := (list (f 0) ... (f <field-count>))
(all-but  <fd0> ... <fdn>)
  := (list (f i0) ... (f in)) | i0...in not in fd0...fdn
```

and, conversely, `fields`, which lets you select a list of fields from the current input row:

```
(fields <field-designator> ... <field-designator-n>) :=
   (list (f <field-designator>) .. <field-designator-n>)
```

In both `all-but` and `fields`, fields can be designated, as usual, with either their identifier, name or `column_number`:

```
(all-but "id" "000023")
(fields "000003" 3 "a field" "another" "0002a3b-3")
```

Sometimes one needs to fill-in missing values in one pass: an easy way for that is provided by the function `all-with-defaults`, that copies all input rows, but replacing missing values with given ones:

```
(all-with-defaults <field-designator-0> <field-value-0>
                   <field-designator-1> <field-value-1>
                   ...
                   <field-designator-n> <field-value-n>)
```

The list of designator/value pairs does not need to be exhaustive or ordered, and again the designator can be a field id, name, or column number:

```
(all-with-defaults "species" "Iris-versicolor"
                   "petal-width" 2.8
                   "000002" 0)
```

It is also possible to provide a default for all missing numeric fields in a row at once, using `all-with-numeric-default`:

```
(all-with-numeric-default <value>)
<value> := "mean" | "median" | "minimum" | "maximum" | <number>
```

As shown, we can specify that missing numeric fields be filled with their mean, median, minimum or maximum values (as read from their respective field descriptors) or with any concrete numeric value. For example:

```
(all-with-numeric-default "median")
(all-with-numeric-default 0)
```

A word of caution: for the case of concrete values, the given number is cast to the datatype of the target field, i.e., it'll be mapped to value range of the given field (for instance, if you give a default value of 128 and a field of type `int8` is missing, it'll receive the value `-1`).

Windows

In addition to horizontally selecting different fields in the same row, we can keep the field fixed and select vertical windows of its value, via the `window` and related operators. They're just syntactic sugar over the shifted field accessors we've already seen:

```
(window <field-designator> <start> <end>)

 := (list (f <field-designator> <start>)
          (f <field-designator> <start + 1>)
          ...
          (f <field-designator> <end>))
```

So, for instance, the window:

```
(window "000001" -1 2)
```

denotes the list of values:

```
(list (f "000001" -1) (f "000001" 0) (f "000001" 1) (f "000001" 2))
```

As shown, both start and end must be integers, and the values corresponding to their shifts are included in the resulting list.

It's possible to apply arithmetic operators, `filter` and `map` to any window. For instance, you could compute the average of the last 3 values of a field as:

```
(/ (+ (window "Temp" -2 0) 3))
```

Or convert all the values to Fahrenheit degrees and select those below 99.9 with:

```
(filter (< _ 99.9) (map (+ 32 (* 1.8 _)) (window "Temp" -2 0)))
```

In addition to the plain `window` generator, we provide some other convenience window primitives computing, respectively, the average value of the values in a window, their sum and the sequence of their differences:

```
(avg-window <field-designator> <start> <end>)
  := (/ (+ (window <field-designator> <start> <end>)) (+ 1 (- <start> <end>))


(sum-window <field-designator> <start> <end>)
  := (+ (window <field-designator> <start> <end>))


(diff-window <fdes> <start> <end>)
  := (list (- (f <fdes> <start>) (f <fdes> (- <start> 1)))
           (- (f <fdes> (- <start> 1)) (f <fdes> (- <start> 2)))
           ...
           (- (f <fdes> (- <end> 1)) (f <fdes> <end>)))
```

These window generator forms can also be combined with `filter`, `map` and all the other window operators.

Conditional window limits

There are scenarios in which you might be interested in forming a window whose width depends on some condition. For instance, say you want to compute the average of a temperature for the last four minutes in a dataset with aperiodic entries: `cond-window` to the rescue:

```
(let (now (f "epoch"))
  (avg (cond-window "temperature" (< (- (f "epoch") now) 240))))
```

As you see in this example, `cond-window` takes a field designator and a predicate; the latter is applied sequentially to the current and future rows (up to a standard maximum value), and a list of the values of the requested fields for the rows satisfying the predicate is returned.

```
(cond-window <fdes> <sexp>)
  := (list (f <fdesc> 0) ... (f <fdesc> n)) | for [0..n] (<sexp>)
```

Note that, as mentioned, `<sexp>` is a Flatline expression computed with the corresponding (future) full row as input.

CHAPTER 2

Quick reference

Field accessors and properties

Access to input field values:

```
(field <field-designator> [<shift>])
(f <field-designator> [<shift>])
(fields <field-designator> ... <field-designator-n>)
(random-field-value <field-designator>)
(weighted-random-field-value <field-designator>)
(ensure-value <field-designator>)
(ensure-weighted-value <field-designator>)
```

All fields in a row:

```
(all)
(all-but <field-designator> ... <field-designator-n>)
(all-with-defaults <field-designator-0> <field-value-0>
                   <field-designator-1> <field-value-1>
                   ...
                   <field-designator-n> <field-value-n>)
(all-with-numeric-default ["mean" "median" "minimum" "maximum" <number>]
```

Row properties:

```
(row-number) ;;  current row number, 0-based
```

Field properties:

```
(bin-center <field-designator> <bin-number>)  ;;  number
(bin-count <field-designator> <bin-number>) ;;  number
(category-count <field-designator> <category>) ;;  number
(maximum <field-designator>) ;;  number
(mean <field-designator>) ;;  number
(median <field-designator>) ;;  number
```

```
(minimum <field-designator>) ;;  number
(missing? <field-designator> [<shift>]) ;;  boolean
(missing-count <field-designator>) ;;  number
(preferred? <field-designator>) ;;  boolean
(population <field-designator>) ;;  integer
(sum <field-designator>) ;;  number
(sum-squares <field-designator>) ;;  number
(variance <field-designator>) ;;  number
(standard-deviation <field-designator>) ;;  number
```

Normalization:

```
(normalize <id> [<from> <to>]) ;; [from to] defaults to [0, 1]
(z-score <id>)
```

Percentiles and population:

```
(percentile <field-designator> <fraction>) ;;  number
(population-fraction <field-designator> <fraction>) ;;  integer
(within-percentiles? <field-designator> <lower> <upper>) ;;  boolean
(percentile-label <field-designator> <label-0> ... <label-n>)
```

Segments:

```
(segment-label <field-designator>
               <label-1> <bound-1>
               ...
               <label-n-1> <bound-n-1>
               <label-n>)
(segment-label <field-designator> <label-1> <label-2> ... <label-n>)
```

Vectorize categorical and text fields:

```
(vectorize <field-designator> [<max-fields>])
```

Items:

```
(contains-items? <field-designator> <item0> ... <itemn>)
(equal-to-items? <field-designator> <item0> ... <itemn>)
```

Clustering:

```
(row-distance <list-of-field-values> [<list-of-field-values> <weights>])
(row-distance-squared <list-of-field-values> [<list-of-field-values> <weights>])
```

Strings and regular expressions

Conversion of any value to a string:

```
(str <sexp0> ...) ;;  string
```

Substrings:

```
(subs <string> <start> [<end>]) ;;  string
```

Regexps:

```
(matches? <string> <regex-string>)  ;;  boolean
(re-quote <string>)  ;;  regexp that matches <string> literally
(replace <string> <regexp> <replacement>) ;;  string
(replace-first <string> <regexp> <replacement>) ;;  string
```

Utilities:

```
(length <string>) ;;  integer
(levenshtein <str-sexp0> <str-sexp1>)  ;;  number
(occurrences <string> <term> [<case-insensitive?> <lang>]) ;;  number
(language <string>) ;;  ["en", "es", "ca", "nl"]
```

Hashing:

```
(md5 <string>) ;;  string of length 32
(sha1 <string>) ;;  string of length 40
(sha256 <string>) ;;  string of length 64
```

Math and logic

Arithmetic operators:

```
+ - * / div mod
```

Relational operators:

```
< <= > >= = !=
```

Logical operators:

```
and or not
```

Mathematical functions:

```
(abs <x>)     ;; Absolute value
(acos <x>)
(asin <x>)
(atan <x>)
(ceil <x>)
(cos <x>)     ;; <x> := radians
(cosh <x>)
(exp <x>)     ;; Exponential
(floor <x>)
(ln <x>)      ;; Natural logarithm
(log <x>)     ;; Natural logarithm
(log2 <x>)    ;; Base-2 logarithm
(log10 <x>)   ;; Base-10 logarithm
(max <x0> ... <xn>)
(min <x0> ... <xn>)
(mod <n> <m>) ;; Modulus
(div <n> <m>) ;; Integer division (quotient)
(pow <x> <n>)
(rand)            ;; a random double in [0, 1)
(rand-int <n>)    ;; a random integer in [0, n) or (n, 0]
```

```
(round <x>)
(sin <x>)      ;; <x> := radians
(sinh <x>)
(sqrt <x>)
(square <x>)   ;; (* <x> <x>)
(tan <x>)      ;; <x> := radians
(tanh <x>)
(to-degrees <x>) ;; <x> := radians
(to-radians <x>) ;; <x> := degrees
(linear-regression <x1> <y1> ... <xn> <yn>) ;; slope, intercept, pearson
(chi-square-p-value <degrees of freedom> <value>)
```

Coercions

```
(integer <sexp>) ;;  integer
(real <sexp>) ;;  real
;; (integer true) = 1, (integer false) = 0
```

Dates and time

Functions taking a number representing the *epoch*, i.e., the number of **milliseconds** since Jan 1st 1970.

```
(epoch-year <n>) ;;  number
(epoch-month <n>) ;;  number
(epoch-day <n>) ;;  number
(epoch-weekday <n>) ;;  number
(epoch-hour <n>) ;;  number
(epoch-minute <n>) ;;  number
(epoch-second <n>) ;;  number
(epoch-millisecond <n>) ;;  number
(epoch-fields <n>) ;;  list of numbers
```

Any string can be coerced to an epoch:

```
(epoch <string> [<format>])
```

Conditionals and local variables

Conditionals:

```
(if <cond> <then> [<else>])

(cond <cond0> <then0>
      <cond1> <then1>
      ... ...
      <default>)
```

For example:

```
(cond (> (f "000001") (mean "000001")) "above average"
      (= (f "000001") (mean "000001")) "below average"
      "mediocre")
```

Local variables:

```
(let <bindings> <body>)
<bindings> := (<varname0> <val0> ...  <varnamen> <valn>)
<body> := <expression with varname0 ... varnamen>
```

For example:

```
(let (x (+ (window "a" -10 10))
      a (/ (* x 3) 4.34)
      y (if (< a 10) "Good" "Bad"))
  (list x (str (f 10) "-" y) a y))
```

Lists

Creation and elememt access:

```
(list <sexp-0> ... <sexp-n>) ;;  list of given values
(cons <head> <tail>) ;;  list
(head <list>) ;;  first element
(tail <list>) ;;  list sans first element
(nth <list> <n>) ;;   0-based nth element
```

Inclusion:

```
(in <value> <list>) ;;  boolean
```

Properties of lists:

```
(count <list>)          ;; (count (list (f 1) (f 2))) => 2
(mode <list>)           ;; (mode (list a b b c b a c c c))  => "c"
(max <list>)            ;; (max (list -1 2 -2 0.38))  => 2
(min <list>)            ;; (min (list -1.3 2 1))  => -1.3
(avg <list>)            ;; (avg (list -1 -2 1 2 0.8 -0.8)) => 0
```

List transformations:

```
(map <fn> (list <a0> <a1> ... <an>))
(filter <fn> (list <a0> ... <an>))
```

Field lists and windows:

```
(fields <field-designator> ... <field-designator-n>)
(window <field-designator> <start> <end>)
(avg-window <field-designator> <start> <end>)  ;; average of values
(sum-window <field-designator> <start> <end>) ;; sum of values
(diff-window <fdes> <start> <end>) ;; differences of consecutive values
(cond-window <fdes> <sexp>)        ;; values that satisfy boolean sexp
```