Missing Values

Julia provides support for representing missing values in the statistical sense, that is for situations where no value is available for a variable in an observation, but a valid value theoretically exists. Missing values are represented via the `missing` object, which is the singleton instance of the type `Missing`. `missing` is equivalent to `NULL` in SQL and `NA` in R, and behaves like them in most situations.

Propagation of Missing Values

`missing` values *propagate* automatically when passed to standard mathematical operators and functions. For these functions, uncertainty about the value of one of the operands induces uncertainty about the result. In practice, this means a math operation involving a `missing` value generally returns `missing`

```
julia> missing + 1
missing

julia> "a" * missing
missing

julia> abs(missing)
missing
```

As `missing` is a normal Julia object, this propagation rule only works for functions which have opted in to implement this behavior. This can be achieved either via a specific method defined for arguments of type `Missing`, or simply by accepting arguments of this type, and passing them to functions which propagate them (like standard math operators). Packages should consider whether it makes sense to propagate missing values when defining new functions, and define methods appropriately if that is the case. Passing a `missing` value to a function for which no method accepting arguments of type `Missing` is defined throws a `MethodError`, just like for any other type.

Functions that do not propagate `missing` values can be made to do so by wrapping them in the `passmissing` function provided by the Missings.jl package. For example, `f(x)` becomes `passmissing(f)(x)`.

Equality and Comparison Operators

Standard equality and comparison operators follow the propagation rule presented above: if any of the operands is `missing`, the result is `missing`. Here are a few examples

```
julia> missing == 1
missing

julia> missing == missing
missing

julia> missing < 1
missing

julia> 2 >= missing
missing
```

In particular, note that `missing == missing` returns `missing`, so `==` cannot be used to test whether a value is missing. To test whether `x` is `missing`, use `ismissing(x)`.

Special comparison operators `isequal` and `===` are exceptions to the propagation rule: they always return a `Bool` value, even in the presence of `missing` values, considering `missing` as equal to `missing` and as different from any other value. They can therefore be used to test whether a value is `missing`

```
julia> missing === 1
false

julia> isequal(missing, 1)
false

julia> missing === missing
true

julia> isequal(missing, missing)
true
```

The `isless` operator is another exception: `missing` is considered as greater than any other value. This operator is used by `sort`, which therefore places `missing` values after all other values.

```
julia> isless(1, missing)
true

julia> isless(missing, Inf)
false
```

```
julia> isless(missing, missing)
false
```

Logical operators

Logical (or boolean) operators `|`, `&` and `xor` are another special case, as they only propagate `missing` values when it is logically required. For these operators, whether or not the result is uncertain depends on the particular operation, following the well-established rules of *three-valued logic* which are also implemented by `NULL` in SQL and `NA` in R. This abstract definition actually corresponds to a relatively natural behavior which is best explained via concrete examples.

Let us illustrate this principle with the logical "or" operator `|`. Following the rules of boolean logic, if one of the operands is `true`, the value of the other operand does not have an influence on the result, which will always be `true`

```
julia> true | true
true

julia> true | false
true

julia> false | true
true
```

Based on this observation, we can conclude that if one of the operands is `true` and the other `missing`, we know that the result is `true` in spite of the uncertainty about the actual value of one of the operands. If we had been able to observe the actual value of the second operand, it could only be `true` or `false`, and in both cases the result would be `true`. Therefore, in this particular case, missingness does *not* propagate

```
julia> true | missing
true

julia> missing | true
true
```

On the contrary, if one of the operands is `false`, the result could be either `true` or `false` depending on the value of the other operand. Therefore, if that operand is `missing`, the result has to be `missing` too

```
julia> false | true
true
```

```
julia> true | false
true

julia> false | false
false

julia> false | missing
missing

julia> missing | false
missing
```

The behavior of the logical "and" operator `&` is similar to that of the `|` operator, with the difference that missingness does not propagate when one of the operands is `false`. For example, when that is the case of the first operand

```
julia> false & false
false

julia> false & true
false

julia> false & missing
false
```

On the other hand, missingness propagates when one of the operands is `true`, for example the first one

```
julia> true & true
true

julia> true & false
false

julia> true & missing
missing
```

Finally, the "exclusive or" logical operator `xor` always propagates `missing` values, since both operands always have an effect on the result. Also note that the negation operator `!` returns `missing` when the operand is `missing` just like other unary operators.

Control Flow and Short-Circuiting Operators

Control flow operators including `if`, `while` and the ternary operator `x ? y : z` do not allow for missing values. This is because of the uncertainty about whether the actual value would be `true` or `false` if we could observe it, which implies that we do not know how the program should behave. A `TypeError` is thrown as soon as a `missing` value is encountered in this context

```
julia> if missing
           println("here")
       end
ERROR: TypeError: non-boolean (Missing) used in boolean context
```

For the same reason, contrary to logical operators presented above, the short-circuiting boolean operators `&&` and `||` do not allow for `missing` values in situations where the value of the operand determines whether the next operand is evaluated or not. For example

```
julia> missing || false
ERROR: TypeError: non-boolean (Missing) used in boolean context

julia> missing && false
ERROR: TypeError: non-boolean (Missing) used in boolean context

julia> true && missing && false
ERROR: TypeError: non-boolean (Missing) used in boolean context
```

On the other hand, no error is thrown when the result can be determined without the `missing` values. This is the case when the code short-circuits before evaluating the `missing` operand, and when the `missing` operand is the last one

```
julia> true && missing
missing

julia> false && missing
false
```

Arrays With Missing Values

Arrays containing missing values can be created like other arrays

```
julia> [1, missing]
2-element Array{Union{Missing, Int64},1}:
 1
  missing
```

As this example shows, the element type of such arrays is `Union{Missing, T}`, with `T` the type of the non-missing values. This simply reflects the fact that array entries can be either of type `T` (here, `Int64`) or of type `Missing`. This kind of array uses an efficient memory storage equivalent to an `Array{T}` holding the actual values combined with an `Array{UInt8}` indicating the type of the entry (i.e. whether it is `Missing` or `T`).

Arrays allowing for missing values can be constructed with the standard syntax. Use `Array{Union{Missing, T}}(missing, dims)` to create arrays filled with missing values:

```
julia> Array{Union{Missing, String}}(missing, 2, 3)
2×3 Array{Union{Missing, String},2}:
 missing  missing  missing
 missing  missing  missing
```

An array allowing for `missing` values but which does not contain any such value can be converted back to an array which does not allow for missing values using `convert`. If the array contains `missing` values, a `MethodError` is thrown during conversion

```
julia> x = Union{Missing, String}["a", "b"]
2-element Array{Union{Missing, String},1}:
 "a"
 "b"

julia> convert(Array{String}, x)
2-element Array{String,1}:
 "a"
 "b"

julia> y = Union{Missing, String}[missing, "b"]
2-element Array{Union{Missing, String},1}:
 missing
 "b"

julia> convert(Array{String}, y)
ERROR: MethodError: Cannot `convert` an object of type Missing to an object of type
```

Skipping Missing Values

Since `missing` values propagate with standard mathematical operators, reduction functions return `missing` when called on arrays which contain missing values

```
julia> sum([1, missing])
missing
```

In this situation, use the `skipmissing` function to skip missing values

```
julia> sum(skipmissing([1, missing]))
1
```

This convenience function returns an iterator which filters out `missing` values efficiently. It can therefore be used with any function which supports iterators

```
julia> x = skipmissing([3, missing, 2, 1])
skipmissing(Union{Missing, Int64}[3, missing, 2, 1])

julia> maximum(x)
3

julia> mean(x)
2.0

julia> mapreduce(sqrt, +, x)
4.146264369941973
```

Objects created by calling `skipmissing` on an array can be indexed using indices from the parent array. Indices corresponding to missing values are not valid for these objects and an error is thrown when trying to use them (they are also skipped by `keys` and `eachindex`)

```
julia> x[1]
3

julia> x[2]
ERROR: MissingException: the value at index (2,) is missing
[...]
```

This allows functions which operate on indices to work in combination with `skipmissing`. This is notably the case for search and find functions, which return indices valid for the object returned by

`skipmissing` which are also the indices of the matching entries *in the parent array*

```
julia> findall(==(1), x)
1-element Array{Int64,1}:
 4

julia> findfirst(!iszero, x)
1

julia> argmax(x)
1
```

Use `collect` to extract non-`missing` values and store them in an array

```
julia> collect(x)
3-element Array{Int64,1}:
 3
 2
 1
```

Logical Operations on Arrays

The three-valued logic described above for logical operators is also used by logical functions applied to arrays. Thus, array equality tests using the `==` operator return `missing` whenever the result cannot be determined without knowing the actual value of the `missing` entry. In practice, this means that `missing` is returned if all non-missing values of the compared arrays are equal, but one or both arrays contain missing values (possibly at different positions)

```
julia> [1, missing] == [2, missing]
false

julia> [1, missing] == [1, missing]
missing

julia> [1, 2, missing] == [1, missing, 2]
missing
```

As for single values, use `isequal` to treat `missing` values as equal to other `missing` values but different from non-missing values

```
julia> isequal([1, missing], [1, missing])
```

```
true

julia> isequal([1, 2, missing], [1, missing, 2])
false
```

Functions `any` and `all` also follow the rules of three-valued logic, returning `missing` when the result cannot be determined

```
julia> all([true, missing])
missing

julia> all([false, missing])
false

julia> any([true, missing])
true

julia> any([false, missing])
missing
```

« Multi-dimensional Arrays Networking and Streams »