Documentation

Julia enables package developers and users to document functions, types and other objects easily via a built-in documentation system since Julia 0.4.

The basic syntax is simple: any string appearing at the toplevel right before an object (function, macro, type or instance) will be interpreted as documenting it (these are called *docstrings*). Note that no blank lines or comments may intervene between a docstring and the documented object. Here is a basic example:

```
"Tell whether there are too foo items in the array."
foo(xs::Array) = ...
```

Documentation is interpreted as Markdown, so you can use indentation and code fences to delimit code examples from text. Technically, any object can be associated with any other as metadata; Markdown happens to be the default, but one can construct other string macros and pass them to the `@doc` macro just as well.

Note

Markdown support is implemented in the `Markdown` standard library and for a full list of supported syntax see the documentation.

Here is a more complex example, still using Markdown:

~~~
"""
    bar(x[, y])

Compute the Bar index between `x` and `y`. If `y` is missing, compute
the Bar index between all pairs of columns of `x`.

# Examples
```julia-repl
julia> bar([1, 2], [1, 2])
1
```
"""
~~~

```
function bar(x, y) ...
```

As in the example above, we recommend following some simple conventions when writing documentation:

1. Always show the signature of a function at the top of the documentation, with a four-space indent so that it is printed as Julia code.

   This can be identical to the signature present in the Julia code (like `mean(x::AbstractArray)`), or a simplified form. Optional arguments should be represented with their default values (i.e. `f(x, y=1)`) when possible, following the actual Julia syntax. Optional arguments which do not have a default value should be put in brackets (i.e. `f(x[, y])` and `f(x[, y[, z]])`). An alternative solution is to use several lines: one without optional arguments, the other(s) with them. This solution can also be used to document several related methods of a given function. When a function accepts many keyword arguments, only include a `<keyword arguments>` placeholder in the signature (i.e. `f(x; <keyword arguments>)`), and give the complete list under an `# Arguments` section (see point 4 below).
2. Include a single one-line sentence describing what the function does or what the object represents after the simplified signature block. If needed, provide more details in a second paragraph, after a blank line.

   The one-line sentence should use the imperative form ("Do this", "Return that") instead of the third person (do not write "Returns the length...") when documenting functions. It should end with a period. If the meaning of a function cannot be summarized easily, splitting it into separate composable parts could be beneficial (this should not be taken as an absolute requirement for every single case though).
3. Do not repeat yourself.

   Since the function name is given by the signature, there is no need to start the documentation with "The function `bar`...": go straight to the point. Similarly, if the signature specifies the types of the arguments, mentioning them in the description is redundant.
4. Only provide an argument list when really necessary.

   For simple functions, it is often clearer to mention the role of the arguments directly in the description of the function's purpose. An argument list would only repeat information already provided elsewhere. However, providing an argument list can be a good idea for complex functions with many arguments (in particular keyword arguments). In that case, insert it after the general description of the function, under an `# Arguments` header, with one `-` bullet for each argument. The list should mention the types and default values (if any) of the arguments:

   ```
   """
   ```

```
...
# Arguments
- `n::Integer`: the number of elements to compute.
- `dim::Integer=1`: the dimensions along which to perform the computation.
...
"""
```

5. Provide hints to related functions.

   Sometimes there are functions of related functionality. To increase discoverability please provide a short list of these in a `See also:` paragraph.

   ```
   See also: [`bar!`](@ref), [`baz`](@ref), [`baaz`](@ref)
   ```

6. Include any code examples in an `# Examples` section.

   Examples should, whenever possible, be written as *doctests*. A *doctest* is a fenced code block (see Code blocks) starting with ` ```jldoctest ` and contains any number of `julia>` prompts together with inputs and expected outputs that mimic the Julia REPL.

   Note

   Doctests are enabled by `Documenter.jl`. For more detailed documentation see Documenter's manual.

   For example in the following docstring a variable `a` is defined and the expected result, as printed in a Julia REPL, appears afterwards:

   ~~~
   """
   Some nice documentation here.

   # Examples
   ```jldoctest
   julia> a = [1 2; 3 4]
   2×2 Array{Int64,2}:
    1  2
    3  4
   ```
   """
   ~~~

   Warning

Calling `rand` and other RNG-related functions should be avoided in doctests since they will not produce consistent outputs during different Julia sessions. If you would like to show some random number generation related functionality, one option is to explicitly construct and seed your own `MersenneTwister` (or other pseudorandom number generator) and pass it to the functions you are doctesting.

Operating system word size (`Int32` or `Int64`) as well as path separator differences (`/` or `\`) will also affect the reproducibility of some doctests.

Note that whitespace in your doctest is significant! The doctest will fail if you misalign the output of pretty-printing an array, for example.

You can then run `make -C doc doctest=true` to run all the doctests in the Julia Manual and API documentation, which will ensure that your example works.

To indicate that the output result is truncated, you may write `[...]` at the line where checking should stop. This is useful to hide a stacktrace (which contains non-permanent references to lines of julia code) when the doctest shows that an exception is thrown, for example:

~~~
```jldoctest
julia> div(1, 0)
ERROR: DivideError: integer division error
[...]
```
~~~

Examples that are untestable should be written within fenced code blocks starting with ` ```julia ` so that they are highlighted correctly in the generated documentation.

Tip

Wherever possible examples should be self-contained and runnable so that readers are able to try them out without having to include any dependencies.

7. Use backticks to identify code and equations.

Julia identifiers and code excerpts should always appear between backticks ` to enable highlighting. Equations in the LaTeX syntax can be inserted between double backticks ``. Use Unicode characters rather than their LaTeX escape sequence, i.e. ` ``α = 1`` ` rather than ` ``\\alpha = 1`` `.

8. Place the starting and ending `"""` characters on lines by themselves.

That is, write:

```
"""
...

...
"""
f(x, y) = ...
```

rather than:

```
"""...

..."""
f(x, y) = ...
```

This makes it clearer where docstrings start and end.

9. Respect the line length limit used in the surrounding code.

   Docstrings are edited using the same tools as code. Therefore, the same conventions should apply. It is recommended that lines are at most 92 characters wide.
10. Provide information allowing custom types to implement the function in an `# Implementation` section. These implementation details are intended for developers rather than users, explaining e.g. which functions should be overridden and which functions automatically use appropriate fallbacks. Such details are best kept separate from the main description of the function's behavior.
11. For long docstrings, consider splitting the documentation with an `# Extended help` header. The typical help-mode will show only the material above the header; you can access the full help by adding a '?' at the beginning of the expression (i.e., "??foo" rather than "?foo").

Accessing Documentation

Documentation can be accessed at the REPL or in IJulia by typing `?` followed by the name of a function or macro, and pressing `Enter`. For example,

```
?cos
?@time
?r""
```

will show documentation for the relevant function, macro or string macro respectively. In Juno using `Ctrl-J, Ctrl-D` will show the documentation for the object under the cursor.

Functions & Methods

Functions in Julia may have multiple implementations, known as methods. While it's good practice for generic functions to have a single purpose, Julia allows methods to be documented individually if necessary. In general, only the most generic method should be documented, or even the function itself (i.e. the object created without any methods by `function bar end`). Specific methods should only be documented if their behaviour differs from the more generic ones. In any case, they should not repeat the information provided elsewhere. For example:

```
"""
    *(x, y, z...)

Multiplication operator. `x * y * z *...` calls this function with multiple
arguments, i.e. `*(x, y, z...)`.
"""
function *(x, y, z...)
    # ... [implementation sold separately] ...
end

"""
    *(x::AbstractString, y::AbstractString, z::AbstractString...)

When applied to strings, concatenates them.
"""
function *(x::AbstractString, y::AbstractString, z::AbstractString...)
    # ... [insert secret sauce here] ...
end

help?> *
search: * .*

  *(x, y, z...)

  Multiplication operator. x * y * z *... calls this function with multiple
  arguments, i.e. *(x,y,z...).

  *(x::AbstractString, y::AbstractString, z::AbstractString...)

  When applied to strings, concatenates them.
```

When retrieving documentation for a generic function, the metadata for each method is concatenated with the `catdoc` function, which can of course be overridden for custom types.

Advanced Usage

The `@doc` macro associates its first argument with its second in a per-module dictionary called `META`.

To make it easier to write documentation, the parser treats the macro name `@doc` specially: if a call to `@doc` has one argument, but another expression appears after a single line break, then that additional expression is added as an argument to the macro. Therefore the following syntax is parsed as a 2-argument call to `@doc`:

```
@doc raw"""
...
"""
f(x) = x
```

This makes it possible to use expressions other than normal string literals (such as the `raw""` string macro) as a docstring.

When used for retrieving documentation, the `@doc` macro (or equally, the `doc` function) will search all `META` dictionaries for metadata relevant to the given object and return it. The returned object (some Markdown content, for example) will by default display itself intelligently. This design also makes it easy to use the doc system in a programmatic way; for example, to re-use documentation between different versions of a function:

```
@doc "..." foo!
@doc (@doc foo!) foo
```

Or for use with Julia's metaprogramming functionality:

```
for (f, op) in ((:add, :+), (:subtract, :-), (:multiply, :*), (:divide, :/))
    @eval begin
        $f(a,b) = $op(a,b)
    end
end
@doc "`add(a,b)` adds `a` and `b` together" add
@doc "`subtract(a,b)` subtracts `b` from `a`" subtract
```

Documentation written in non-toplevel blocks, such as `begin`, `if`, `for`, and `let`, is added to the documentation system as blocks are evaluated. For example:

```
if condition()
    "..."
    f(x) = x
end
```

will add documentation to `f(x)` when `condition()` is `true`. Note that even if `f(x)` goes out of scope at the end of the block, its documentation will remain.

It is possible to make use of metaprogramming to assist in the creation of documentation. When using string-interpolation within the docstring you will need to use an extra `$` as shown with `$($name)`:

```
for func in (:day, :dayofmonth)
    name = string(func)
    @eval begin
        @doc """
            $($name)(dt::TimeType) -> Int64

        The day of month of a `Date` or `DateTime` as an `Int64`.
        """ $func(dt::Dates.TimeType)
    end
end
```

Dynamic documentation

Sometimes the appropriate documentation for an instance of a type depends on the field values of that instance, rather than just on the type itself. In these cases, you can add a method to `Docs.getdoc` for your custom type that returns the documentation on a per-instance basis. For instance,

```
struct MyType
    value::String
end

Docs.getdoc(t::MyType) = "Documentation for MyType with value $(t.value)"

x = MyType("x")
y = MyType("y")
```

`?x` will display "Documentation for MyType with value x" while `?y` will display "Documentation for MyType with value y".

Syntax Guide

This guide provides a comprehensive overview of how to attach documentation to all Julia syntax constructs for which providing documentation is possible.

In the following examples `"..."` is used to illustrate an arbitrary docstring.

`$` and `\` characters

The `$` and `\` characters are still parsed as string interpolation or start of an escape sequence in docstrings too. The `raw""` string macro together with the `@doc` macro can be used to avoid having to escape them. This is handy when the docstrings include LaTeX or Julia source code examples containing interpolation:

~~~
@doc raw"""
```math
\LaTeX
```
"""
function f end
~~~

Functions and Methods

```
"..."
function f end

"..."
f
```

Adds docstring `"..."` to the function `f`. The first version is the preferred syntax, however both are equivalent.

```
"..."
f(x) = x

"..."
function f(x)
    x
end

"..."
f(x)
```

Adds docstring `"..."` to the method `f(::Any)`.

```
"..."
f(x, y = 1) = x + y
```

Adds docstring `"..."` to two `Method`s, namely `f(::Any)` and `f(::Any, ::Any)`.

Macros

```
"..."
macro m(x) end
```

Adds docstring `"..."` to the `@m(::Any)` macro definition.

```
"..."
:(@m)
```

Adds docstring `"..."` to the macro named `@m`.

Types

```
"..."
abstract type T1 end

"..."
mutable struct T2
    ...
end

"..."
struct T3
    ...
end
```

Adds the docstring `"..."` to types `T1`, `T2`, and `T3`.

```
"..."
struct T
    "x"
    x
    "y"
    y
```

```
end
```

Adds docstring `"..."` to type `T`, `"x"` to field `T.x` and `"y"` to field `T.y`. Also applicable to `mutable struct` types.

Modules

```
"..."
module M end

module M

"..."
M

end
```

Adds docstring `"..."` to the `Module M`. Adding the docstring above the `Module` is the preferred syntax, however both are equivalent.

```
"..."
baremodule M
# ...
end

baremodule M

import Base: @doc

"..."
f(x) = x

end
```

Documenting a `baremodule` by placing a docstring above the expression automatically imports `@doc` into the module. These imports must be done manually when the module expression is not documented. Empty `baremodule`s cannot be documented.

Global Variables

```
"..."
```

```
const a = 1

"..."
b = 2

"..."
global c = 3
```

Adds docstring `"..."` to the `Binding`s `a`, `b`, and `c`.

`Binding`s are used to store a reference to a particular `Symbol` in a `Module` without storing the referenced value itself.

Note

When a `const` definition is only used to define an alias of another definition, such as is the case with the function `div` and its alias `÷` in `Base`, do not document the alias and instead document the actual function.

If the alias is documented and not the real definition then the docsystem (`?` mode) will not return the docstring attached to the alias when the real definition is searched for.

For example you should write

```
"..."
f(x) = x + 1
const alias = f
```

rather than

```
f(x) = x + 1
"..."
const alias = f
```

```
"..."
sym
```

Adds docstring `"..."` to the value associated with `sym`. However, it is preferred that `sym` is documented where it is defined.

Multiple Objects

```
"..."
a, b
```

Adds docstring `"..."` to `a` and `b` each of which should be a documentable expression. This syntax is equivalent to

```
"..."
a

"..."
b
```

Any number of expressions many be documented together in this way. This syntax can be useful when two functions are related, such as non-mutating and mutating versions `f` and `f!`.

Macro-generated code

```
"..."
@m expression
```

Adds docstring `"..."` to the expression generated by expanding `@m expression`. This allows for expressions decorated with `@inline`, `@noinline`, `@generated`, or any other macro to be documented in the same way as undecorated expressions.

Macro authors should take note that only macros that generate a single expression will automatically support docstrings. If a macro returns a block containing multiple subexpressions then the subexpression that should be documented must be marked using the `@__doc__` macro.

The `@enum` macro makes use of `@__doc__` to allow for documenting `Enum`s. Examining its definition should serve as an example of how to use `@__doc__` correctly.

`Core.@__doc__` — Macro

```
@__doc__(ex)
```

Low-level macro used to mark expressions returned by a macro that should be documented. If more than one expression is marked then the same docstring is applied to each expression.

```
macro example(f)
    quote
        $(f)() = 0
        @__doc__ $(f)(x) = 1
        $(f)(x, y) = 2
    end |> esc
end
```

`@__doc__` has no effect when a macro that uses it is not documented.

« Modules Metaprogramming »