Blue: a Style Guide for Julia

Code Formatting

Module Imports

Module imports should occur at the top of the file or right after a `module` declaration.
Files loaded via an `include` should avoid specifying their own module imports and should instead add them to the file in which they were included (e.g. "src/Example.jl" or "test/runtests.jl").

A module import should only specify a single package per line.
The lines should be ordered alphabetically by the package/module name (note: relative imports precede absolute imports).

```
# Yes:
using A
using B

# No:
using A, B

# No:
using B
using A
```

Imports which explicitly declare what to bring into scope should be grouped into: modules, constants, types, macros, and functions.
These groupings should be specified in that order and each group's contents should be sorted alphabetically, typically with modules on a separate line.
As pseudo-code:

```
using Example: $(sort(modules)...)
using Example: $(sort(constants)...), $(sort(types)...), $(sort(macros)...),
$(sort(functions)...)
```

If you are only explicitly importing a few items you can alternatively use the following one-line form:

```
using Example: $(sort(modules)...), $(sort(constants)...), $(sort(types)...),
$(sort(macros)...), $(sort(functions)...)
```

In some scenarios there may be alternate ordering within a group which makes more logical sense.
For example when explicitly importing subtypes of `Period` you may want to sort them in units largest to smallest:

```
using Dates: Year, Month, Week, Day, Hour, Minute, Second, Millisecond
```

Explicit import lines which exceed the line length should use line-continuation or multiple import statements for the same package.
Using multiple import statements should be preferred when creating alternate groupings:

```
# Yes:
using AVeryLongPackage: AVeryLongType, AnotherVeryLongType,
a_very_long_function,
    another_very_long_function

# Yes:
using AVeryLongPackage: AVeryLongType, AnotherVeryLongType
using AVeryLongPackage: a_very_long_function, another_very_long_function

# No:
using AVeryLongPackage:
    AVeryLongType,
    AnotherVeryLongType,
    a_very_long_function,
    another_very_long_function

# No:
using AVeryLongPackage: AVeryLongType
using AVeryLongPackage: AnotherVeryLongType
using AVeryLongPackage: a_very_long_function
using AVeryLongPackage: another_very_long_function
```

Note: Prefer the use of imports with explicit declarations when writing packages.
Doing so will make maintaining the package easier by knowing what functionality the package is importing and when dependencies can safely be dropped.

Prefer the use of `using` over `import` to ensure that extension of a function is always explicit and on purpose:

```
# Yes:
using Example

Example.hello(x::Monster) = "Aargh! It's a Monster!"
Base.isreal(x::Ghost) = false

# No:
import Base: isreal
import Example: hello

hello(x::Monster) = "Aargh! It's a Monster!"
isreal(x::Ghost) = false
```

If you do require the use of `import` then create separate groupings for `import` and `using` statements divided by a blank line:

```
# Yes:
import A: a
import C

using B
using D: d

# No:
import A: a
using B
import C
using D: d
```

Function Exports

All functions that are intended to be part of the public API should be exported.
All function exports should occur at the top of the main module file, after module imports.
Avoid splitting a single export over multiple lines; either define one export per line, or group them by theme.

```
# Yes:
export foo
export bar
export qux

# Yes:
export get_foo, get_bar
export solve_foo, solve_bar

# No:
export foo,
    bar,
    qux
```

Global Variables

Global variables should be avoided whenever possible. When required, global variables should be `const`s and have an all uppercase name separated with underscores (e.g. `MY_CONSTANT`).
They should be defined at the top of the file, immediately after imports and exports but before an `__init__` function.
If you truly want mutable global style behaviour you may want to look into mutable containers or closures.

Function Naming

Names of functions should describe an action or property irrespective of the type of the argument; the argument's type provides this information instead.
For example, `submit_bid(bid)` should be `submit(bid::Bid)` and `bids_in_batch(batch)` should be `bids(batch::Batch)`.

Names of functions should usually be limited to one or two lowercase words separated by underscores.
If you find it hard to shorten your function names without losing information, you may need to factor more information into the type signature or split the function's responsibilities into two or more functions.
In general, shorter functions with clearly-defined responsibilities are preferred.

NOTE: Functions that are only intended for internal use should be marked with a leading underscore (e.g., `_internal_utility_function(x, y)`).
Although it should be much less common, the same naming convention can be used for internal types and constants as well (e.g., `_MyInternalType`, `_MY_CONSTANT`).
Marking a function as internal or private lets other people know that they shouldn't expect any kind of API stability from that functionality.

Method Definitions

Only use short-form function definitions when they fit on a single line:

```
# Yes:
foo(x::Int64) = abs(x) + 3

# No:
foobar(array_data::AbstractArray{T}, item::T) where {T<:Int64} = T[
    abs(x) * abs(item) + 3 for x in array_data
]
```

```
# Yes:
function foobar(array_data::AbstractArray{T}, item::T) where T<:Int64
    return T[abs(x) * abs(item) + 3 for x in array_data]
end

# No:
foobar(
    array_data::AbstractArray{T},
    item::T,
) where {T<:Int64} = T[abs(x) * abs(item) + 3 for x in array_data]
```

When using long-form functions always use the `return` keyword:

```
# Yes:
function fnc(x::T) where T
    result = zero(T)
    result += fna(x)
    return result
end

# No:
function fnc(x::T) where T
    result = zero(T)
    result += fna(x)
end
```

```
# Yes:
function Foo(x, y)
    return new(x, y)
end

# No:
function Foo(x, y)
    new(x, y)
end
```

When using the `return` keyword always explicitly return a value, even if it is `return nothing`.

```
# Yes:
function maybe_do_thing()
    # code
    return nothing
end

# No:
function maybe_do_thing()
    # code
    return
end
```

Functions definitions with parameter lines which exceed 92 characters should separate each parameter by a newline and indent by one-level:

```
# Yes:
function foobar(
    df::DataFrame,
    id::Symbol,
    variable::Symbol,
    value::AbstractString,
    prefix::AbstractString="",
)
    # code
end

# Ok:
function foobar(df::DataFrame, id::Symbol, variable::Symbol,
value::AbstractString, prefix::AbstractString="")
    # code
end

# No: Don't put any args on the same line as the open parenthesis if they won't
all fit.
function foobar(df::DataFrame, id::Symbol, variable::Symbol,
value::AbstractString,
    prefix::AbstractString="")

    # code
end

# No: All args should be on a new line in this case.
```

```
function foobar(
    df::DataFrame, id::Symbol, variable::Symbol, value::AbstractString,
    prefix::AbstractString=""
)
    # code
end

# No: Indented too much.
function foobar(
        df::DataFrame,
        id::Symbol,
        variable::Symbol,
        value::AbstractString,
        prefix::AbstractString="",
    )
    # code
end
```

If all of the arguments fit inside the 92 character limit then you can place them on 1 line. Similarly, you can follow the same rule if you break up positional and keyword arguments across two lines.

```
# Ok:
function foobar(
    df::DataFrame, id::Symbol, variable::Symbol, value::AbstractString;
prefix::String=""
)
    # code
end

# Ok: Putting all args and all kwargs on separate lines is fine.
function foobar(
    df::DataFrame, id::Symbol, variable::Symbol, value::AbstractString;
    prefix::String=""
)
    # code
end

# Ok: Putting all positional args on 1 line and each kwarg on separate lines.
function foobar(
    df::DataFrame, id::Symbol, variable::Symbol, value::AbstractString;
    prefix="I'm a long default setting that probably shouldn't exist",
    msg="I'm another long default setting that probably shouldn't exist",
)
    # code
end

# No: Because the separate line is more than 92 characters.
function foobar(
    df::DataFrame, id::Symbol, variable::Symbol, value::AbstractString;
prefix::AbstractString=""
)
    # code
end
```

```
# No: The args and kwargs should be split up.
function foobar(
    df::DataFrame, id::Symbol, variable::Symbol,
    value::AbstractString; prefix::AbstractString=""
)
    # code
end

# No: The kwargs are more than 92 characters.
function foobar(
    df::DataFrame, id::Symbol, variable::Symbol, value::AbstractString;
    prefix="I'm a long default setting that probably shouldn't exist", msg="I'm
another long default settings that probably shouldn't exist",
)
    # code
end
```

Keyword Arguments

When calling a function always separate your keyword arguments from your positional arguments with a semicolon.
This avoids mistakes in ambiguous cases (such as splatting a `Dict`).

```
# Yes:
xy = foo(x; y=3)
ab = foo(; a=1, b=2)

# Ok:
ab = foo(a=1, b=2)

# No:
xy = foo(x, y=3)
```

Whitespace

- Avoid extraneous whitespace immediately inside parentheses, square brackets or braces.

  ```
  # Yes:
  spam(ham[1], [eggs])

  # No:
  spam( ham[ 1 ], [ eggs ] )
  ```

- Avoid extraneous whitespace immediately before a comma or semicolon:

  ```
  # Yes:
  if x == 4 @show(x, y); x, y = y, x end

  # No:
  if x == 4 @show(x , y) ; x , y = y , x end
  ```

- Avoid whitespace around `:` in ranges. Use brackets to clarify expressions on either side.

```
# Yes:
ham[1:9]
ham[9:-3:0]
ham[1:step:end]
ham[lower:upper-1]
ham[lower:upper - 1]
ham[lower:(upper + offset)]
ham[(lower + offset):(upper + offset)]

# No:
ham[1: 9]
ham[9 : -3: 1]
ham[lower : upper - 1]
ham[lower + offset:upper + offset]  # Avoid as it is easy to read as
`ham[lower + (offset:upper) + offset]`
```

- Avoid using more than one space around an assignment (or other) operator to align it with another:

```
# Yes:
x = 1
y = 2
long_variable = 3

# No:
x             = 1
y             = 2
long_variable = 3
```

- Surround most binary operators with a single space on either side: assignment (`=`), updating operators (`+=`, `-=`, etc.), numeric comparisons operators (`==`, `<`, `>`, `!=`, etc.), lambda operator (`->`). Binary operators that may be excluded from this guideline include: the range operator (`:`), rational operator (`//`), exponentiation operator (`^`), optional arguments/keywords (e.g. `f(x=1; y=2)`).

```
# Yes:
i = j + 1
submitted += 1
x^2 < y

# No:
i=j+1
submitted +=1
x^2<y
```

- Avoid using whitespace between unary operands and the expression:

```
# Yes:
-1
[1 0 -1]

# No:
- 1
[1 0 - 1]  # Note: evaluates to `[1 -1]`
```

- Avoid extraneous empty lines. Avoid empty lines between single line method definitions and otherwise separate functions with one empty line, plus a comment if required:

```
# Yes:
# Note: an empty line before the first long-form `domaths` method is
optional.
domaths(x::Number) = x + 5
domaths(x::Int) = x + 10
function domaths(x::String)
    return "A string is a one-dimensional extended object postulated in
string theory."
end

dophilosophy() = "Why?"

# No:
domath(x::Number) = x + 5

domath(x::Int) = x + 10
```

```
function domath(x::String)
    return "A string is a one-dimensional extended object postulated in string
theory."
end

dophilosophy() = "Why?"
```

- Function calls which cannot fit on a single line within the line limit should be broken up such that the lines containing the opening and closing brackets are indented to the same level while the parameters of the function are indented one level further.
  In most cases the arguments and/or keywords should each be placed on separate lines.
  Note that this rule conflicts with the typical Julia convention of indenting the next line to align with the open bracket in which the parameter is contained.
  If working in a package with a different convention follow the convention used in the package over using this guideline.

```
  # Yes:
  f(a, b)
  constraint = conic_form!(
      SOCElemConstraint(temp2 + temp3, temp2 - temp3, 2 * temp1),
      unique_conic_forms,
```

```
)

# No:
# Note: `f` call is short enough to be on a single line
f(
    a,
    b,
)
constraint = conic_form!(SOCElemConstraint(temp2 + temp3,
                                           temp2 - temp3, 2 * temp1),
                         unique_conic_forms)
```

- Assignments using expanded notation for arrays or tuples, or function calls should have the first open bracket on the same line assignment operator and the closing bracket should match the indentation level of the assignment.
  Alternatively you can perform assignments on a single line when they are short:

```
# Yes:
arr = [
    1,
    2,
    3,
]
arr = [
    1, 2, 3,
]
result = func(
    arg1,
    arg2,
)
arr = [1, 2, 3]

# No:
arr =
[
    1,
    2,
    3,
]
arr =
[
    1, 2, 3,
]
arr = [
    1,
    2,
    3,
    ]
```

- Nested arrays or tuples that are in expanded notation should have the opening and closing brackets at the same indentation level:

```
# Yes:
x = [
```

```
    [
        1, 2, 3,
    ],
    [
        "hello",
        "world",
    ],
    ['a', 'b', 'c'],
]

# No:
y = [
    [
        1, 2, 3,
    ], [
        "hello",
        "world",
    ],
]
z = [[
        1, 2, 3,
    ], [
        "hello",
        "world",
    ],
]
```

- Always include the trailing comma when working with expanded arrays, tuples or functions notation.
  This allows future edits to easily move elements around or add additional elements.
  The trailing comma should be excluded when the notation is only on a single-line:

```
# Yes:
arr = [
    1,
    2,
    3,
]
result = func(
    arg1,
    arg2,
)
arr = [1, 2, 3]

# No:
arr = [
    1,
    2,
    3
]
result = func(
    arg1,
    arg2
)
arr = [1, 2, 3,]
```

- Triple-quotes and triple-backticks written over multiple lines should be indented.
  As triple-quotes use the indentation of the lowest indented line (excluding the opening quotes) the least indented line in the string or ending quotes should be indented once.
  Triple-backticks should also follow this style even though the indentation does not matter for them.

```
# Yes:
str = """
    hello
    world!
    """
cmd = ```
    program
        --flag value
        parameter
```

No:

str = """
hello
world!
"""
cmd = `program`

```
        --flag value
        parameter
```

```

```

- Assignments using triple-quotes or triple-backticks should have the opening quotes on the same line as the assignment operator.

```
# Yes:
str2 = """
       hello
   world!
   """

# No:
str2 =
   """
       hello
   world!
   """
```

- Group similar one line statements together.

```
# Yes:
foo = 1
bar = 2
baz = 3

# No:
foo = 1

bar = 2

baz = 3
```

- Use blank-lines to separate different multi-line blocks.

```
# Yes:
if foo
    println("Hi")
end

for i in 1:10
    println(i)
end

# No:
if foo
    println("Hi")
end
for i in 1:10
    println(i)
end
```

- After a function definition, and before an end statement do not include a blank line.

```
# Yes:
function foo(bar::Int64, baz::Int64)
    return bar + baz
end

# No:
function foo(bar::Int64, baz::Int64)

    return bar + baz
end

# No:
function foo(bar::In64, baz::Int64)
    return bar + baz

end
```

- Use line breaks between control flow statements and returns.

```
# Yes:
```

```
function foo(bar; verbose=false)
    if verbose
        println("baz")
    end

    return bar
end

# Ok:
function foo(bar; verbose=false)
    if verbose
        println("baz")
    end
    return bar
end
```

NamedTuples

The `=` character in `NamedTuple`s should be spaced as in keyword arguments.
No space should be put between the name and its value.
`NamedTuple`s should not be prefixed with `;` at the start.
The empty `NamedTuple` should be written `NamedTuple()` not `(;)`

```
# Yes:
xy = (x=1, y=2)
x = (x=1,)  # Trailing comma required for correctness.
x = (; kwargs...)  # Semicolon required to splat correctly.

# No:
xy = (x = 1, y = 2)
xy = (;x=1,y=2)
x = (; x=1)
```

Numbers

Floating-point numbers should always include a leading and/or trailing zero:

```
# Yes:
0.1
2.0
3.0f0

# No:
.1
2.
3.f0
```

Ternary Operator

Ternary operators (`?:`) should generally only consume a single line.
Do not chain multiple ternary operators.
If chaining many conditions, consider using an `if-elseif-else` conditional, dispatch, or a dictionary.

```
# Yes:
foobar = foo == 2 ? bar : baz

# No:
foobar = foo == 2 ?
    bar :
    baz
foobar = foo == 2 ? bar : foo == 3 ? qux : baz
```

As an alternative, you can use a compound boolean expression:

```
# Yes:
foobar = if foo == 2
    bar
else
    baz
end

foobar = if foo == 2
    bar
elseif foo == 3
    qux
else
    baz
end
```

For loops

For loops should always use `in`, never `=` or `∈`.
This also applies to list and generator comprehensions

```
# Yes
for i in 1:10
    #...
end

[foo(x) for x in xs]

# No:
for i = 1:10
    #...
end

[foo(x) for x ∈ xs]
```

Modules

Normally a file that includes the definition of a module, should not include any other code that runs outside that module.
i.e. the module should be declared at the top of the file with the `module` keyword and `end` at the bottom of the file.
No other code before, or after (except for module docstring before).
In this case the code with in the module block should **not** be indented.

Sometimes, e.g. for tests, or for namespacing an enumeration, it *is* desirable to declare a submodule midway through a file.
In this case the code within the submodule **should** be indented.

Type annotation

Annotations for function definitions should be as general as possible.

```
# Yes:
splicer(arr::AbstractArray, step::Integer) = arr[begin:step:end]

# No:
splicer(arr::Array{Int}, step::Int) = arr[begin:step:end]
```

Using as generic types as possible allows for a variety of inputs and allows your code to be more general:

```
julia> splicer(1:10, 2)
1:2:9

julia> splicer([3.0, 5, 7, 9], 2)
2-element Array{Float64,1}:
 3.0
 7.0
```

Annotations on type fields need to be given a little more thought.
Using specific concrete types for fields allows Julia to optimize the memory layout but can reduce flexibility.
For example lets take a look at the type `MySubString` which allows us to work with a subsection of a string without having to copy the data:

```
mutable struct MySubString <: AbstractString
    string::AbstractString
    offset::Integer
    endof::Integer
end
```

We want the type to be able to hold any subtype of `AbstractString` but do we need to have `offset` and `endof` to be able to hold any subtype of `Integer`?
Really, no, we should be ok to use `Int` here (`Int64` on 64-bit systems and `Int32` on 32-bit systems).
Note that even though we're using `Int` a user can still do things like `MySubString("foobar", 0x4, 0x6);` as provided `offset` and `endof` values will be converted to an `Int`.

```
mutable struct MySubString <: AbstractString
    string::AbstractString
    offset::Int
    endof::Int
end
```

If we truly care about performance there is one more thing we can do by making our type parametric.
The current definition of `MySubString` allows us to modify the `string` field at any time with any subtype of `AbstractString`.
Using a parametric type allows us to use any subtype of `AbstractString` upon construction but the field type will be set to something concrete (like `String`) and cannot be changed for the lifetime of the instance.

```
mutable struct MySubString{T<:AbstractString} <: AbstractString
    string::T
    offset::Integer
    endof::Integer
end
```

Overall, it is best to keep the types general to start with and later optimize them using parametric types.
Optimizing too early in the code design process can impact your ability to refactor the code early on.

Package version specifications

For simplicity and consistency with the wider Julia community, avoid including the default caret specifier when specifying package version requirements.

```
# Yes:
DataFrames = "0.17"

# No:
DataFrames = "^0.17"
```

Comments

Comments should be used to state the intended behaviour of code.
This is especially important when the code is doing something clever that may not be obvious upon first inspection.
Avoid writing comments that state exactly what the code obviously does.

```
# Yes:
x = x + 1      # Compensate for border

# No:
x = x + 1      # Increment x
```

Comments that contradict the code are much worse than no comments.
Always make a priority of keeping the comments up-to-date with code changes!

Comments should be complete sentences.
If a comment is a phrase or sentence, its first word should be capitalized, unless it is an identifier that begins with a lower case letter (never alter the case of identifiers!).

If a comment is short, the period at the end can be omitted.
Block comments generally consist of one or more paragraphs built out of complete sentences, and each sentence should end in a period.

Comments should be separated by at least two spaces from the expression and have a single space after the `#`.

When referencing Julia in documentation note that "Julia" refers to the programming language while "julia" (typically in backticks, e.g. `julia`) refers to the executable.

Only use inline comments if they fit within the line length limit. If your comment cannot be fitted inline then place the comment above the content to which it refers:

```
# Yes:

# Number of nodes to predict. Again, an issue with the workflow order. Should be
updated
# after data is fetched.
p = 1

# No:

p = 1  # Number of nodes to predict. Again, an issue with the workflow order.
Should be
# updated after data is fetched.
```

Documentation

It is recommended that most modules, types and functions should have docstrings.
That being said, only exported functions are required to be documented.
Avoid documenting methods like `==` as the built in docstring for the function already covers the details well.
Try to document a function and not individual methods where possible as typically all methods will have similar docstrings.
If you are adding a method to a function which already has a docstring only add a docstring if the behaviour of your function deviates from the existing docstring.

Docstrings are written in Markdown and should be concise.
Docstring lines should be wrapped at 92 characters.

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

When types or methods have lots of parameters it may not be feasible to write a concise docstring.
In these cases it is recommended you use the templates below.
Note if a section doesn't apply or is overly verbose (for example "Throws" if your function doesn't throw an exception) it can be excluded.
It is recommended that you have a blank line between the headings and the content when the content is of sufficient length.
Try to be consistent within a docstring whether you use this additional whitespace.
Note that the additional space is only for reading raw markdown and does not affect the rendered version.

Type Template (should be skipped if is redundant with the constructor(s) docstring):

```
"""
    MyArray{T,N}

My super awesome array wrapper!

# Fields
- `data::AbstractArray{T,N}`: stores the array being wrapped
- `metadata::Dict`: stores metadata about the array
"""
struct MyArray{T,N} <: AbstractArray{T,N}
    data::AbstractArray{T,N}
    metadata::Dict
end
```

Function Template (only required for exported functions):

```
"""
    mysearch(array::MyArray{T}, val::T; verbose=true) where {T} -> Int

Searches the `array` for the `val`. For some reason we don't want to use Julia's
builtin search :)

# Arguments
- `array::MyArray{T}`: the array to search
- `val::T`: the value to search for

# Keywords
- `verbose::Bool=true`: print out progress details

# Returns
- `Int`: the index where `val` is located in the `array`

# Throws
- `NotFoundError`: I guess we could throw an error if `val` isn't found.
"""
function mysearch(array::AbstractArray{T}, val::T) where T
    ...
end
```

If your method contains lots of arguments or keywords you may want to exclude them from the method signature on the first line and instead use `args...` and/or `kwargs...`.

```
"""
    Manager(args...; kwargs...) -> Manager

A cluster manager which spawns workers.

# Arguments

- `min_workers::Integer`: The minimum number of workers to spawn or an exception
is thrown
- `max_workers::Integer`: The requested number of workers to spawn

# Keywords

- `definition::AbstractString`: Name of the job definition to use. Defaults to
the
    definition used within the current instance.
- `name::AbstractString`: ...
- `queue::AbstractString`: ...
"""
function Manager(...)
    ...
end
```

Feel free to document multiple methods for a function within the same docstring.
Be careful to only do this for functions you have defined.

```
"""
    Manager(max_workers; kwargs...)
    Manager(min_workers:max_workers; kwargs...)
    Manager(min_workers, max_workers; kwargs...)

A cluster manager which spawns workers.

# Arguments

- `min_workers::Int`: The minimum number of workers to spawn or an exception is
thrown
- `max_workers::Int`: The requested number of workers to spawn

# Keywords

- `definition::AbstractString`: Name of the job definition to use. Defaults to
the
    definition used within the current instance.
- `name::AbstractString`: ...
- `queue::AbstractString`: ...
"""
function Manager end
```

If the documentation for bullet-point exceeds 92 characters the line should be wrapped and slightly indented.
Avoid aligning the text to the `:`.

```
"""
...

# Keywords
- `definition::AbstractString`: Name of the job definition to use. Defaults to
the
    definition used within the current instance.
"""
```

For additional details on documenting in Julia see the official documentation.

For documentation written in Markdown files such as `README.md` or `docs/src/index.md` use exactly one sentence per line.

Test Formatting

Testsets

Julia provides test sets which allows developers to group tests into logical groupings.
Test sets can be nested and ideally packages should only have a single "root" test set.
It is recommended that the "runtests.jl" file contains the root test set which contains the remainder of the tests:

```
@testset "PkgExtreme" begin
    include("arithmetic.jl")
    include("utils.jl")
end
```

Comparisons

Most tests are written in the form `@test x == y`.
Since the `==` function doesn't take types into account tests like the following are valid: `@test 1.0 == 1`.
Avoid adding visual noise into test comparisons:

```
# Yes:
@test value == 0

# No:
@test value == 0.0
```

Performance and Optimization

Several of these tips are contained within Julia's Performance Tips.

Much of Julia's performance gains come from being able to specialize functions on their input types.
Putting variables and functionality in the global namespace or module's namespace thwarts this. One consequence of this is that conventional MATLAB-style scripts will result in surprisingly slow code.
There are two ways to mitigate this:

- Move as much functionality into functions as possible.
- Declare global variables as constants using `const`.

Remember that the first time you call a function with a certain type signature it will compile that function for the given input types.
Compilation is sometimes a significant portion of time, so avoid profiling/timing functions on their first run. Note that the `@benchmark` and `@btime` macros from the BenchmarkTools package can be useful as they run the function many times and report summary statistics of time and memory allocation, alleviating the need to run the function first before benchmarking.

References

[1] https://github.com/invenia/BlueStyle.