Style Guide

The following sections explain a few aspects of idiomatic Julia coding style. None of these rules are absolute; they are only suggestions to help familiarize you with the language and to help you choose among alternative designs.

Write functions, not just scripts

Writing code as a series of steps at the top level is a quick way to get started solving a problem, but you should try to divide a program into functions as soon as possible. Functions are more reusable and testable, and clarify what steps are being done and what their inputs and outputs are. Furthermore, code inside functions tends to run much faster than top level code, due to how Julia's compiler works.

It is also worth emphasizing that functions should take arguments, instead of operating directly on global variables (aside from constants like `pi`).

Avoid writing overly-specific types

Code should be as generic as possible. Instead of writing:

```
Complex{Float64}(x)
```

it's better to use available generic functions:

```
complex(float(x))
```

The second version will convert `x` to an appropriate type, instead of always the same type.

This style point is especially relevant to function arguments. For example, don't declare an argument to be of type `Int` or `Int32` if it really could be any integer, expressed with the abstract type `Integer`. In fact, in many cases you can omit the argument type altogether, unless it is needed to disambiguate from other method definitions, since a `MethodError` will be thrown anyway if a type is passed that does not support any of the requisite operations. (This is known as duck typing.)

For example, consider the following definitions of a function `addone` that returns one plus its argument:

```
addone(x::Int) = x + 1                  # works only for Int
addone(x::Integer) = x + oneunit(x)     # any integer type
addone(x::Number) = x + oneunit(x)      # any numeric type
addone(x) = x + oneunit(x)              # any type supporting + and oneunit
```

The last definition of `addone` handles any type supporting `oneunit` (which returns 1 in the same type as `x`, which avoids unwanted type promotion) and the `+` function with those arguments. The key thing to realize is that there is *no performance penalty* to defining *only* the general `addone(x) = x + oneunit(x)`, because Julia will automatically compile specialized versions as needed. For example, the first time you call `addone(12)`, Julia will automatically compile a specialized `addone` function for `x::Int` arguments, with the call to `oneunit` replaced by its inlined value `1`. Therefore, the first three definitions of `addone` above are completely redundant with the fourth definition.

Handle excess argument diversity in the caller

Instead of:

```
function foo(x, y)
    x = Int(x); y = Int(y)
    ...
end
foo(x, y)
```

use:

```
function foo(x::Int, y::Int)
    ...
end
foo(Int(x), Int(y))
```

This is better style because `foo` does not really accept numbers of all types; it really needs `Int`s.

One issue here is that if a function inherently requires integers, it might be better to force the caller to decide how non-integers should be converted (e.g. floor or ceiling). Another issue is that declaring more specific types leaves more "space" for future method definitions.

Append ! to names of functions that modify their arguments

Instead of:

```
function double(a::AbstractArray{<:Number})
    for i = firstindex(a):lastindex(a)
        a[i] *= 2
    end
    return a
end
```

use:

```
function double!(a::AbstractArray{<:Number})
    for i = firstindex(a):lastindex(a)
        a[i] *= 2
    end
    return a
end
```

Julia Base uses this convention throughout and contains examples of functions with both copying and modifying forms (e.g., `sort` and `sort!`), and others which are just modifying (e.g., `push!`, `pop!`, `splice!`). It is typical for such functions to also return the modified array for convenience.

Avoid strange type `Union`s

Types such as `Union{Function,AbstractString}` are often a sign that some design could be cleaner.

Avoid elaborate container types

It is usually not much help to construct arrays like the following:

```
a = Vector{Union{Int,AbstractString,Tuple,Array}}(undef, n)
```

In this case `Vector{Any}(undef, n)` is better. It is also more helpful to the compiler to annotate specific uses (e.g. `a[i]::Int`) than to try to pack many alternatives into one type.

Use naming conventions consistent with Julia `base/`

- modules and type names use capitalization and camel case: `module SparseArrays`, `struct UnitRange`.

- functions are lowercase (`maximum`, `convert`) and, when readable, with multiple words squashed together (`isequal`, `haskey`). When necessary, use underscores as word separators. Underscores are also used to indicate a combination of concepts (`remotecall_fetch` as a more efficient implementation of `fetch(remotecall(...))`) or as modifiers.
- conciseness is valued, but avoid abbreviation (`indexin` rather than `indxin`) as it becomes difficult to remember whether and how particular words are abbreviated.

If a function name requires multiple words, consider whether it might represent more than one concept and might be better split into pieces.

Write functions with argument ordering similar to Julia Base

As a general rule, the Base library uses the following order of arguments to functions, as applicable:

1. Function argument. Putting a function argument first permits the use of `do` blocks for passing multiline anonymous functions.
2. I/O stream. Specifying the `IO` object first permits passing the function to functions such as `sprint`, e.g. `sprint(show, x)`.
3. Input being mutated. For example, in `fill!(x, v)`, `x` is the object being mutated and it appears before the value to be inserted into `x`.
4. Type. Passing a type typically means that the output will have the given type. In `parse(Int, "1")`, the type comes before the string to parse. There are many such examples where the type appears first, but it's useful to note that in `read(io, String)`, the `IO` argument appears before the type, which is in keeping with the order outlined here.
5. Input not being mutated. In `fill!(x, v)`, `v` is *not* being mutated and it comes after `x`.
6. Key. For associative collections, this is the key of the key-value pair(s). For other indexed collections, this is the index.
7. Value. For associative collections, this is the value of the key-value pair(s). In cases like `fill!(x, v)`, this is `v`.
8. Everything else. Any other arguments.
9. Varargs. This refers to arguments that can be listed indefinitely at the end of a function call. For example, in `Matrix{T}(undef, dims)`, the dimensions can be given as a `Tuple`, e.g. `Matrix{T}(undef, (1,2))`, or as `Vararg`s, e.g. `Matrix{T}(undef, 1, 2)`.
10. Keyword arguments. In Julia keyword arguments have to come last anyway in function definitions; they're listed here for the sake of completeness.

The vast majority of functions will not take every kind of argument listed above; the numbers merely

denote the precedence that should be used for any applicable arguments to a function.

There are of course a few exceptions. For example, in `convert`, the type should always come first. In `setindex!`, the value comes before the indices so that the indices can be provided as varargs.

When designing APIs, adhering to this general order as much as possible is likely to give users of your functions a more consistent experience.

Don't overuse try-catch

It is better to avoid errors than to rely on catching them.

Don't parenthesize conditions

Julia doesn't require parens around conditions in `if` and `while`. Write:

```
if a == b
```

instead of:

```
if (a == b)
```

Don't overuse `...`

Splicing function arguments can be addictive. Instead of `[a..., b...]`, use simply `[a; b]`, which already concatenates arrays. `collect(a)` is better than `[a...]`, but since `a` is already iterable it is often even better to leave it alone, and not convert it to an array.

Don't use unnecessary static parameters

A function signature:

```
foo(x::T) where {T<:Real} = ...
```

should be written as:

```
foo(x::Real) = ...
```

instead, especially if `T` is not used in the function body. Even if `T` is used, it can be replaced with `typeof(x)` if convenient. There is no performance difference. Note that this is not a general caution against static parameters, just against uses where they are not needed.

Note also that container types, specifically may need type parameters in function calls. See the FAQ Avoid fields with abstract containers for more information.

Avoid confusion about whether something is an instance or a type

Sets of definitions like the following are confusing:

```
foo(::Type{MyType}) = ...
foo(::MyType) = foo(MyType)
```

Decide whether the concept in question will be written as `MyType` or `MyType()`, and stick to it.

The preferred style is to use instances by default, and only add methods involving `Type{MyType}` later if they become necessary to solve some problem.

If a type is effectively an enumeration, it should be defined as a single (ideally immutable struct or primitive) type, with the enumeration values being instances of it. Constructors and conversions can check whether values are valid. This design is preferred over making the enumeration an abstract type, with the "values" as subtypes.

Don't overuse macros

Be aware of when a macro could really be a function instead.

Calling `eval` inside a macro is a particularly dangerous warning sign; it means the macro will only work when called at the top level. If such a macro is written as a function instead, it will naturally have access to the run-time values it needs.

Don't expose unsafe operations at the interface level

If you have a type that uses a native pointer:

```
mutable struct NativeType
    p::Ptr{UInt8}
    ...
```

```
end
```

don't write definitions like the following:

```
getindex(x::NativeType, i) = unsafe_load(x.p, i)
```

The problem is that users of this type can write `x[i]` without realizing that the operation is unsafe, and then be susceptible to memory bugs.

Such a function should either check the operation to ensure it is safe, or have `unsafe` somewhere in its name to alert callers.

Don't overload methods of base container types

It is possible to write definitions like the following:

```
show(io::IO, v::Vector{MyType}) = ...
```

This would provide custom showing of vectors with a specific new element type. While tempting, this should be avoided. The trouble is that users will expect a well-known type like `Vector()` to behave in a certain way, and overly customizing its behavior can make it harder to work with.

Avoid type piracy

"Type piracy" refers to the practice of extending or redefining methods in Base or other packages on types that you have not defined. In some cases, you can get away with type piracy with little ill effect. In extreme cases, however, you can even crash Julia (e.g. if your method extension or redefinition causes invalid input to be passed to a `ccall`). Type piracy can complicate reasoning about code, and may introduce incompatibilities that are hard to predict and diagnose.

As an example, suppose you wanted to define multiplication on symbols in a module:

```
module A
import Base.*
*(x::Symbol, y::Symbol) = Symbol(x,y)
end
```

The problem is that now any other module that uses `Base.*` will also see this definition. Since `Symbol` is defined in Base and is used by other modules, this can change the behavior of unrelated code unexpectedly. There are several alternatives here, including using a different function name, or

wrapping the `Symbol`s in another type that you define.

Sometimes, coupled packages may engage in type piracy to separate features from definitions, especially when the packages were designed by collaborating authors, and when the definitions are reusable. For example, one package might provide some types useful for working with colors; another package could define methods for those types that enable conversions between color spaces. Another example might be a package that acts as a thin wrapper for some C code, which another package might then pirate to implement a higher-level, Julia-friendly API.

Be careful with type equality

You generally want to use `isa` and `<:` for testing types, not `==`. Checking types for exact equality typically only makes sense when comparing to a known concrete type (e.g. `T == Float64`), or if you *really, really* know what you're doing.

Do not write `x->f(x)`

Since higher-order functions are often called with anonymous functions, it is easy to conclude that this is desirable or even necessary. But any function can be passed directly, without being "wrapped" in an anonymous function. Instead of writing `map(x->f(x), a)`, write `map(f, a)`.

Avoid using floats for numeric literals in generic code when possible

If you write generic code which handles numbers, and which can be expected to run with many different numeric type arguments, try using literals of a numeric type that will affect the arguments as little as possible through promotion.

For example,

```
julia> f(x) = 2.0 * x
f (generic function with 1 method)

julia> f(1//2)
1.0

julia> f(1/2)
1.0

julia> f(1)
```

```
2.0
```

while

```
julia> g(x) = 2 * x
g (generic function with 1 method)

julia> g(1//2)
1//1

julia> g(1/2)
1.0

julia> g(1)
2
```

As you can see, the second version, where we used an `Int` literal, preserved the type of the input argument, while the first didn't. This is because e.g. `promote_type(Int, Float64) == Float64`, and promotion happens with the multiplication. Similarly, `Rational` literals are less type disruptive than `Float64` literals, but more disruptive than `Int`s:

```
julia> h(x) = 2//1 * x
h (generic function with 1 method)

julia> h(1//2)
1//1

julia> h(1/2)
1.0

julia> h(1)
2//1
```

Thus, use `Int` literals when possible, with `Rational{Int}` for literal non-integer numbers, in order to make it easier to use your code.

« Workflow Tips Frequently Asked Questions »