Noteworthy Differences from other Languages

Noteworthy differences from MATLAB

Although MATLAB users may find Julia's syntax familiar, Julia is not a MATLAB clone. There are major syntactic and functional differences. The following are some noteworthy differences that may trip up Julia users accustomed to MATLAB:

- Julia arrays are indexed with square brackets, `A[i,j]`.
- Julia arrays are not copied when assigned to another variable. After `A = B`, changing elements of `B` will modify `A` as well.
- Julia values are not copied when passed to a function. If a function modifies an array, the changes will be visible in the caller.
- Julia does not automatically grow arrays in an assignment statement. Whereas in MATLAB `a(4) = 3.2` can create the array `a = [0 0 0 3.2]` and `a(5) = 7` can grow it into `a = [0 0 0 3.2 7]`, the corresponding Julia statement `a[5] = 7` throws an error if the length of `a` is less than 5 or if this statement is the first use of the identifier `a`. Julia has `push!` and `append!`, which grow `Vectors` much more efficiently than MATLAB's `a(end+1) = val`.
- The imaginary unit `sqrt(-1)` is represented in Julia as `im`, not `i` or `j` as in MATLAB.
- In Julia, literal numbers without a decimal point (such as `42`) create integers instead of floating point numbers. As a result, some operations can throw a domain error if they expect a float; for example, `julia> a = -1; 2^a` throws a domain error, as the result is not an integer (see the FAQ entry on domain errors for details).
- In Julia, multiple values are returned and assigned as tuples, e.g. `(a, b) = (1, 2)` or `a, b = 1, 2`. MATLAB's `nargout`, which is often used in MATLAB to do optional work based on the number of returned values, does not exist in Julia. Instead, users can use optional and keyword arguments to achieve similar capabilities.
- Julia has true one-dimensional arrays. Column vectors are of size `N`, not `Nx1`. For example, `rand(N)` makes a 1-dimensional array.
- In Julia, `[x,y,z]` will always construct a 3-element array containing `x`, `y` and `z`.
  - To concatenate in the first ("vertical") dimension use either `vcat(x,y,z)` or separate with semicolons (`[x; y; z]`).
  - To concatenate in the second ("horizontal") dimension use either `hcat(x,y,z)` or separate with spaces (`[x y z]`).

    - To construct block matrices (concatenating in the first two dimensions), use either `hvcat` or combine spaces and semicolons (`[a b; c d]`).
- In Julia, `a:b` and `a:b:c` construct `AbstractRange` objects. To construct a full vector like in MATLAB, use `collect(a:b)`. Generally, there is no need to call `collect` though. An `AbstractRange` object will act like a normal array in most cases but is more efficient because it lazily computes its values. This pattern of creating specialized objects instead of full arrays is used frequently, and is also seen in functions such as `range`, or with iterators such as `enumerate`, and `zip`. The special objects can mostly be used as if they were normal arrays.
- Functions in Julia return values from their last expression or the `return` keyword instead of listing the names of variables to return in the function definition (see The return Keyword for details).
- A Julia script may contain any number of functions, and all definitions will be externally visible when the file is loaded. Function definitions can be loaded from files outside the current working directory.
- In Julia, reductions such as `sum`, `prod`, and `max` are performed over every element of an array when called with a single argument, as in `sum(A)`, even if `A` has more than one dimension.
- In Julia, parentheses must be used to call a function with zero arguments, like in `rand()`.
- Julia discourages the use of semicolons to end statements. The results of statements are not automatically printed (except at the interactive prompt), and lines of code do not need to end with semicolons. `println` or `@printf` can be used to print specific output.
- In Julia, if `A` and `B` are arrays, logical comparison operations like `A == B` do not return an array of booleans. Instead, use `A .== B`, and similarly for the other boolean operators like `<`, `>`.
- In Julia, the operators `&`, `|`, and `⊻` (`xor`) perform the bitwise operations equivalent to `and`, `or`, and `xor` respectively in MATLAB, and have precedence similar to Python's bitwise operators (unlike C). They can operate on scalars or element-wise across arrays and can be used to combine logical arrays, but note the difference in order of operations: parentheses may be required (e.g., to select elements of `A` equal to 1 or 2 use `(A .== 1) .| (A .== 2)`).
- In Julia, the elements of a collection can be passed as arguments to a function using the splat operator `...`, as in `xs=[1,2]; f(xs...)`.
- Julia's `svd` returns singular values as a vector instead of as a dense diagonal matrix.
- In Julia, `...` is not used to continue lines of code. Instead, incomplete expressions automatically continue onto the next line.
- In both Julia and MATLAB, the variable `ans` is set to the value of the last expression issued in an interactive session. In Julia, unlike MATLAB, `ans` is not set when Julia code is run in non-interactive mode.
- Julia's `structs` do not support dynamically adding fields at runtime, unlike MATLAB's `classes`. Instead, use a `Dict`.
- In Julia each module has its own global scope/namespace, whereas in MATLAB there is just one

global scope.

- In MATLAB, an idiomatic way to remove unwanted values is to use logical indexing, like in the expression `x(x>3)` or in the statement `x(x>3) = []` to modify `x` in-place. In contrast, Julia provides the higher order functions `filter` and `filter!`, allowing users to write `filter(z->z>3, x)` and `filter!(z->z>3, x)` as alternatives to the corresponding transliterations `x[x.>3]` and `x = x[x.>3]`. Using `filter!` reduces the use of temporary arrays.
- The analogue of extracting (or "dereferencing") all elements of a cell array, e.g. in `vertcat(A{:})` in MATLAB, is written using the splat operator in Julia, e.g. as `vcat(A...)`.
- In Julia, the `adjoint` function performs conjugate transposition; in MATLAB, `adjoint` provides the "adjugate" or classical adjoint, which is the transpose of the matrix of cofactors.

Noteworthy differences from R

One of Julia's goals is to provide an effective language for data analysis and statistical programming. For users coming to Julia from R, these are some noteworthy differences:

- Julia's single quotes enclose characters, not strings.
- Julia can create substrings by indexing into strings. In R, strings must be converted into character vectors before creating substrings.
- In Julia, like Python but unlike R, strings can be created with triple quotes `""" ... """`. This syntax is convenient for constructing strings that contain line breaks.
- In Julia, varargs are specified using the splat operator `...`, which always follows the name of a specific variable, unlike R, for which `...` can occur in isolation.
- In Julia, modulus is `mod(a, b)`, not `a %% b`. `%` in Julia is the remainder operator.
- In Julia, not all data structures support logical indexing. Furthermore, logical indexing in Julia is supported only with vectors of length equal to the object being indexed. For example:
  - In R, `c(1, 2, 3, 4)[c(TRUE, FALSE)]` is equivalent to `c(1, 3)`.
  - In R, `c(1, 2, 3, 4)[c(TRUE, FALSE, TRUE, FALSE)]` is equivalent to `c(1, 3)`.
  - In Julia, `[1, 2, 3, 4][[true, false]]` throws a `BoundsError`.
  - In Julia, `[1, 2, 3, 4][[true, false, true, false]]` produces `[1, 3]`.
- Like many languages, Julia does not always allow operations on vectors of different lengths, unlike R where the vectors only need to share a common index range. For example, `c(1, 2, 3, 4) + c(1, 2)` is valid R but the equivalent `[1, 2, 3, 4] + [1, 2]` will throw an error in Julia.
- Julia allows an optional trailing comma when that comma does not change the meaning of code. This can cause confusion among R users when indexing into arrays. For example, `x[1,]` in R would return the first row of a matrix; in Julia, however, the comma is ignored, so `x[1,] == x[1]`, and will return the first element. To extract a row, be sure to use `:`, as in `x[1,:]`.

- Julia's `map` takes the function first, then its arguments, unlike `lapply(<structure>, function, ...)` in R. Similarly Julia's equivalent of `apply(X, MARGIN, FUN, ...)` in R is `mapslices` where the function is the first argument.
- Multivariate apply in R, e.g. `mapply(choose, 11:13, 1:3)`, can be written as `broadcast(binomial, 11:13, 1:3)` in Julia. Equivalently Julia offers a shorter dot syntax for vectorizing functions `binomial.(11:13, 1:3)`.
- Julia uses `end` to denote the end of conditional blocks, like `if`, loop blocks, like `while`/ `for`, and functions. In lieu of the one-line `if ( cond ) statement`, Julia allows statements of the form `if cond; statement; end`, `cond && statement` and `!cond || statement`. Assignment statements in the latter two syntaxes must be explicitly wrapped in parentheses, e.g. `cond && (x = value)`.
- In Julia, `<-`, `<<-` and `->` are not assignment operators.
- Julia's `->` creates an anonymous function.
- Julia constructs vectors using brackets. Julia's `[1, 2, 3]` is the equivalent of R's `c(1, 2, 3)`.
- Julia's `*` operator can perform matrix multiplication, unlike in R. If `A` and `B` are matrices, then `A * B` denotes a matrix multiplication in Julia, equivalent to R's `A %*% B`. In R, this same notation would perform an element-wise (Hadamard) product. To get the element-wise multiplication operation, you need to write `A .* B` in Julia.
- Julia performs matrix transposition using the `transpose` function and conjugated transposition using the `'` operator or the `adjoint` function. Julia's `transpose(A)` is therefore equivalent to R's `t(A)`. Additionally a non-recursive transpose in Julia is provided by the `permutedims` function.
- Julia does not require parentheses when writing `if` statements or `for`/`while` loops: use `for i in [1, 2, 3]` instead of `for (i in c(1, 2, 3))` and `if i == 1` instead of `if (i == 1)`.
- Julia does not treat the numbers `0` and `1` as Booleans. You cannot write `if (1)` in Julia, because `if` statements accept only booleans. Instead, you can write `if true`, `if Bool(1)`, or `if 1==1`.
- Julia does not provide `nrow` and `ncol`. Instead, use `size(M, 1)` for `nrow(M)` and `size(M, 2)` for `ncol(M)`.
- Julia is careful to distinguish scalars, vectors and matrices. In R, `1` and `c(1)` are the same. In Julia, they cannot be used interchangeably.
- Julia's `diag` and `diagm` are not like R's.
- Julia cannot assign to the results of function calls on the left hand side of an assignment operation: you cannot write `diag(M) = fill(1, n)`.
- Julia discourages populating the main namespace with functions. Most statistical functionality for Julia is found in packages under the JuliaStats organization. For example:
  - Functions pertaining to probability distributions are provided by the Distributions package.
  - The DataFrames package provides data frames.
  - Generalized linear models are provided by the GLM package.

- Julia provides tuples and real hash tables, but not R-style lists. When returning multiple items, you should typically use a tuple or a named tuple: instead of `list(a = 1, b = 2)`, use `(1, 2)` or `(a=1, b=2)`.
- Julia encourages users to write their own types, which are easier to use than S3 or S4 objects in R. Julia's multiple dispatch system means that `table(x::TypeA)` and `table(x::TypeB)` act like R's `table.TypeA(x)` and `table.TypeB(x)`.
- In Julia, values are not copied when assigned or passed to a function. If a function modifies an array, the changes will be visible in the caller. This is very different from R and allows new functions to operate on large data structures much more efficiently.
- In Julia, vectors and matrices are concatenated using `hcat`, `vcat` and `hvcat`, not `c`, `rbind` and `cbind` like in R.
- In Julia, a range like `a:b` is not shorthand for a vector like in R, but is a specialized `AbstractRange` object that is used for iteration without high memory overhead. To convert a range into a vector, use `collect(a:b)`.
- Julia's `max` and `min` are the equivalent of `pmax` and `pmin` respectively in R, but both arguments need to have the same dimensions. While `maximum` and `minimum` replace `max` and `min` in R, there are important differences.
- Julia's `sum`, `prod`, `maximum`, and `minimum` are different from their counterparts in R. They all accept an optional keyword argument `dims`, which indicates the dimensions, over which the operation is carried out. For instance, let `A = [1 2; 3 4]` in Julia and `B <- rbind(c(1,2),c(3,4))` be the same matrix in R. Then `sum(A)` gives the same result as `sum(B)`, but `sum(A, dims=1)` is a row vector containing the sum over each column and `sum(A, dims=2)` is a column vector containing the sum over each row. This contrasts to the behavior of R, where separate `colSums(B)` and `rowSums(B)` functions provide these functionalities. If the `dims` keyword argument is a vector, then it specifies all the dimensions over which the sum is performed, while retaining the dimensions of the summed array, e.g. `sum(A, dims=(1,2)) == hcat(10)`. It should be noted that there is no error checking regarding the second argument.
- Julia has several functions that can mutate their arguments. For example, it has both `sort` and `sort!`.
- In R, performance requires vectorization. In Julia, almost the opposite is true: the best performing code is often achieved by using devectorized loops.
- Julia is eagerly evaluated and does not support R-style lazy evaluation. For most users, this means that there are very few unquoted expressions or column names.
- Julia does not support the `NULL` type. The closest equivalent is `nothing`, but it behaves like a scalar value rather than like a list. Use `x === nothing` instead of `is.null(x)`.
- In Julia, missing values are represented by the `missing` object rather than by `NA`. Use `ismissing(x)` (or `ismissing.(x)` for element-wise operation on vectors) instead of `is.na(x)`. The `skipmissing` function is generally used instead of `na.rm=TRUE` (though in some particular

cases functions take a `skipmissing` argument).

- Julia lacks the equivalent of R's `assign` or `get`.
- In Julia, `return` does not require parentheses.
- In R, an idiomatic way to remove unwanted values is to use logical indexing, like in the expression `x[x>3]` or in the statement `x = x[x>3]` to modify `x` in-place. In contrast, Julia provides the higher order functions `filter` and `filter!`, allowing users to write `filter(z->z>3, x)` and `filter!(z->z>3, x)` as alternatives to the corresponding transliterations `x[x.>3]` and `x = x[x.>3]`. Using `filter!` reduces the use of temporary arrays.

Noteworthy differences from Python

- Julia's `for`, `if`, `while`, etc. blocks are terminated by the `end` keyword. Indentation level is not significant as it is in Python. Unlike Python, Julia has no `pass` keyword.
- Strings are denoted by double quotation marks (`"text"`) in Julia (with three double quotation marks for multi-line strings), whereas in Python they can be denoted either by single (`'text'`) or double quotation marks (`"text"`). Single quotation marks are used for characters in Julia (`'c'`).
- String concatenation is done with `*` in Julia, not `+` like in Python. Analogously, string repetition is done with `^`, not `*`. Implicit string concatenation of string literals like in Python (e.g. `'ab' 'cd' == 'abcd'`) is not done in Julia.
- Python Lists—flexible but slow—correspond to the Julia `Vector{Any}` type or more generally `Vector{T}` where `T` is some non-concrete element type. "Fast" arrays like Numpy arrays that store elements in-place (i.e., `dtype` is `np.float64`, `[('f1', np.uint64), ('f2', np.int32)]`, etc.) can be represented by `Array{T}` where `T` is a concrete, immutable element type. This includes built-in types like `Float64`, `Int32`, `Int64` but also more complex types like `Tuple{UInt64,Float64}` and many user-defined types as well.
- In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
- Julia's slice indexing includes the last element, unlike in Python. `a[2:3]` in Julia is `a[1:3]` in Python.
- Julia does not support negative indices. In particular, the last element of a list or array is indexed with `end` in Julia, not `-1` as in Python.
- Julia requires `end` for indexing until the last element. `x[1:]` in Python is equivalent to `x[2:end]` in Julia.
- Julia's range indexing has the format of `x[start:step:stop]`, whereas Python's format is `x[start:(stop+1):step]`. Hence, `x[0:10:2]` in Python is equivalent to `x[1:2:10]` in Julia. Similarly, `x[::-1]` in Python, which refers to the reversed array, is equivalent to `x[end:-1:1]` in Julia.
- In Julia, indexing a matrix with arrays like `X[[1,2], [1,3]]` refers to a sub-matrix that contains the intersections of the first and second rows with the first and third columns. In Python, `X[[1,2],`

`[1,3]]` refers to a vector that contains the values of cell `[1,1]` and `[2,3]` in the matrix. `X[[1,2], [1,3]]` in Julia is equivalent with `X[np.ix_([0,1],[0,2])]` in Python. `X[[0,1], [0,2]]` in Python is equivalent with `X[[CartesianIndex(1,1), CartesianIndex(2,3)]]` in Julia.

- Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.
- Julia arrays are column major (Fortran ordered) whereas NumPy arrays are row major (C-ordered) by default. To get optimal performance when looping over arrays, the order of the loops should be reversed in Julia relative to NumPy (see relevant section of Performance Tips).
- Julia's updating operators (e.g. `+=`, `-=`, ...) are *not in-place* whereas NumPy's are. This means `A = [1, 1]; B = A; B += [3, 3]` doesn't change values in `A`, it rather rebinds the name `B` to the result of the right-hand side `B = B + 3`, which is a new array. For in-place operation, use `B .+= 3` (see also dot operators), explicit loops, or `InplaceOps.jl`.
- Julia evaluates default values of function arguments every time the method is invoked, unlike in Python where the default values are evaluated only once when the function is defined. For example, the function `f(x=rand()) = x` returns a new random number every time it is invoked without argument. On the other hand, the function `g(x=[1,2]) = push!(x,3)` returns `[1,2,3]` every time it is called as `g()`.
- In Julia `%` is the remainder operator, whereas in Python it is the modulus.
- In Julia, the commonly used `Int` type corresponds to the machine integer type (`Int32` or `Int64`), unlike in Python, where `int` is an arbitrary length integer. This means in Julia the `Int` type will overflow, such that `2^64 == 0`. If you need larger values use another appropriate type, such as `Int128`, `BigInt` or a floating point type like `Float64`.
- The imaginary unit `sqrt(-1)` is represented in Julia as `im`, not `j` as in Python.
- In Julia, the exponentiation operator is `^`, not `**` as in Python.
- Julia uses `nothing` of type `Nothing` to represent a null value, whereas Python uses `None` of type `NoneType`.
- In Julia, the standard operators over a matrix type are matrix operations, whereas, in Python, the standard operators are element-wise operations. When both `A` and `B` are matrices, `A * B` in Julia performs matrix multiplication, not element-wise multiplication as in Python. `A * B` in Julia is equivalent with `A @ B` in Python, whereas `A * B` in Python is equivalent with `A .* B` in Julia.
- The adjoint operator `'` in Julia returns an adjoint of a vector (a lazy representation of row vector), whereas the transpose operator `.T` over a vector in Python returns the original vector (non-op).
- In Julia, a function may contain multiple concrete implementations (called *Methods*), selected via multiple dispatch, whereas functions in Python have a single implementation (no polymorphism).
- There are no classes in Julia. Instead they are structures (mutable or immutable), containing data but no methods.

- Calling a method of a class in Python (`a = MyClass(x), x.func(y)`) corresponds to a function call in Julia, e.g. `a = MyStruct(x), func(x::MyStruct, y)`. In general, multiple dispatch is more flexible and powerful than the Python class system.
- Julia structures may have exactly one abstract supertype, whereas Python classes can inherit from one or more (abstract or concrete) superclasses.
- The logical Julia program structure (Packages and Modules) is independent of the file strucutre (`include` for additional files), whereas the Python code structure is defined by directories (Packages) and files (Modules).
- The ternary operator `x > 0 ? 1 : -1` in Julia corresponds to conditional expression in Python `1 if x > 0 else -1`.
- In Julia the `@` symbol refers to a macro, whereas in Python it refers to a decorator.
- Exception handling in Julia is done using `try — catch — finally`, instead of `try — except — finally`. In contrast to Python, it is not recommended to use exception handling as part of the normal workflow in Julia due to performance reasons.
- In Julia loops are fast, there is no need to write "vectorized" code for performance reasons.
- Be careful with non-constant global variables in Julia, especially in tight loops. Since you can write close-to-metal code in Julia (unlike Python), the effect of globals can be drastic (see Performance Tips).
- In Python, the majority of values can be used in logical contexts (e.g. `if "a":` means the following block is executed, and `if "":` means it is not). In Julia, you need explicit conversion to `Bool` (e.g. `if "a"` throws an exception). If you want to test for a non-empty string in Julia, you would explicitly write `if !isempty("")`.
- In Julia, a new local scope is introduced by most code blocks, including loops and `try — catch — finally`. Note that comprehensions (list, generator, etc.) introduce a new local scope both in Python and Julia, whereas `if` blocks do not introduce a new local scope in both languages.

Noteworthy differences from C/C++

- Julia arrays are indexed with square brackets, and can have more than one dimension `A[i,j]`. This syntax is not just syntactic sugar for a reference to a pointer or address as in C/C++. See the Julia documentation for the syntax for array construction (it has changed between versions).
- In Julia, indexing of arrays, strings, etc. is 1-based not 0-based.
- Julia arrays are not copied when assigned to another variable. After `A = B`, changing elements of `B` will modify `A` as well. Updating operators like `+=` do not operate in-place, they are equivalent to `A = A + B` which rebinds the left-hand side to the result of the right-hand side expression.
- Julia arrays are column major (Fortran ordered) whereas C/C++ arrays are row major ordered by default. To get optimal performance when looping over arrays, the order of the loops should be

reversed in Julia relative to C/C++ (see relevant section of Performance Tips).

- Julia values are not copied when assigned or passed to a function. If a function modifies an array, the changes will be visible in the caller.
- In Julia, whitespace is significant, unlike C/C++, so care must be taken when adding/removing whitespace from a Julia program.
- In Julia, literal numbers without a decimal point (such as `42`) create signed integers, of type `Int`, but literals too large to fit in the machine word size will automatically be promoted to a larger size type, such as `Int64` (if `Int` is `Int32`), `Int128`, or the arbitrarily large `BigInt` type. There are no numeric literal suffixes, such as `L`, `LL`, `U`, `UL`, `ULL` to indicate unsigned and/or signed vs. unsigned. Decimal literals are always signed, and hexadecimal literals (which start with `0x` like C/C++), are unsigned. Hexadecimal literals also, unlike C/C++/Java and unlike decimal literals in Julia, have a type based on the *length* of the literal, including leading 0s. For example, `0x0` and `0x00` have type `UInt8`, `0x000` and `0x0000` have type `UInt16`, then literals with 5 to 8 hex digits have type `UInt32`, 9 to 16 hex digits type `UInt64` and 17 to 32 hex digits type `UInt128`. This needs to be taken into account when defining hexadecimal masks, for example `~0xf == 0xf0` is very different from `~0x000f == 0xfff0`. 64 bit `Float64` and 32 bit `Float32` bit literals are expressed as `1.0` and `1.0f0` respectively. Floating point literals are rounded (and not promoted to the `BigFloat` type) if they can not be exactly represented. Floating point literals are closer in behavior to C/C++. Octal (prefixed with `0o`) and binary (prefixed with `0b`) literals are also treated as unsigned.
- String literals can be delimited with either `"` or `"""`, `"""` delimited literals can contain `"` characters without quoting it like `"\""`. String literals can have values of other variables or expressions interpolated into them, indicated by `$variablename` or `$(expression)`, which evaluates the variable name or the expression in the context of the function.
- `//` indicates a `Rational` number, and not a single-line comment (which is `#` in Julia)
- `#=` indicates the start of a multiline comment, and `=#` ends it.
- Functions in Julia return values from their last expression(s) or the `return` keyword. Multiple values can be returned from functions and assigned as tuples, e.g. `(a, b) = myfunction()` or `a, b = myfunction()`, instead of having to pass pointers to values as one would have to do in C/C++ (i.e. `a = myfunction(&b)`.
- Julia does not require the use of semicolons to end statements. The results of expressions are not automatically printed (except at the interactive prompt, i.e. the REPL), and lines of code do not need to end with semicolons. `println` or `@printf` can be used to print specific output. In the REPL, `;` can be used to suppress output. `;` also has a different meaning within `[ ]`, something to watch out for. `;` can be used to separate expressions on a single line, but are not strictly necessary in many cases, and are more an aid to readability.
- In Julia, the operator `⊻` (`xor`) performs the bitwise XOR operation, i.e. `^` in C/C++. Also, the bitwise operators do not have the same precedence as C/++, so parenthesis may be required.
- Julia's `^` is exponentiation (pow), not bitwise XOR as in C/C++ (use `⊻`, or `xor`, in Julia)

- Julia has two right-shift operators, `>>` and `>>>`. `>>>` performs an arithmetic shift, `>>` always performs a logical shift, unlike C/C++, where the meaning of `>>` depends on the type of the value being shifted.
- Julia's `->` creates an anonymous function, it does not access a member via a pointer.
- Julia does not require parentheses when writing `if` statements or `for`/`while` loops: use `for i in [1, 2, 3]` instead of `for (int i=1; i <= 3; i++)` and `if i == 1` instead of `if (i == 1)`.
- Julia does not treat the numbers `0` and `1` as Booleans. You cannot write `if (1)` in Julia, because `if` statements accept only booleans. Instead, you can write `if true`, `if Bool(1)`, or `if 1==1`.
- Julia uses `end` to denote the end of conditional blocks, like `if`, loop blocks, like `while`/ `for`, and functions. In lieu of the one-line `if ( cond ) statement`, Julia allows statements of the form `if cond; statement; end`, `cond && statement` and `!cond || statement`. Assignment statements in the latter two syntaxes must be explicitly wrapped in parentheses, e.g. `cond && (x = value)`, because of the operator precedence.
- Julia has no line continuation syntax: if, at the end of a line, the input so far is a complete expression, it is considered done; otherwise the input continues. One way to force an expression to continue is to wrap it in parentheses.
- Julia macros operate on parsed expressions, rather than the text of the program, which allows them to perform sophisticated transformations of Julia code. Macro names start with the `@` character, and have both a function-like syntax, `@mymacro(arg1, arg2, arg3)`, and a statement-like syntax, `@mymacro arg1 arg2 arg3`. The forms are interchangeable; the function-like form is particularly useful if the macro appears within another expression, and is often clearest. The statement-like form is often used to annotate blocks, as in the distributed `for` construct: `@distributed for i in 1:n; #= body =#; end`. Where the end of the macro construct may be unclear, use the function-like form.
- Julia has an enumeration type, expressed using the macro `@enum(name, value1, value2, ...)` For example: `@enum(Fruit, banana=1, apple, pear)`
- By convention, functions that modify their arguments have a `!` at the end of the name, for example `push!`.
- In C++, by default, you have static dispatch, i.e. you need to annotate a function as virtual, in order to have dynamic dispatch. On the other hand, in Julia every method is "virtual" (although it's more general than that since methods are dispatched on every argument type, not only `this`, using the most-specific-declaration rule).

Noteworthy differences from Common Lisp

- Julia uses 1-based indexing for arrays by default, and it can also handle arbitrary index offsets.
- Functions and variables share the same namespace (“Lisp-1”).

- There is a `Pair` type, but it is not meant to be used as a `COMMON-LISP:CONS`. Various iterable collections can be used interchangeably in most parts of the language (eg splatting, tuples, etc). `Tuple`s are the closest to Common Lisp lists for *short* collections of heterogeneous elements. Use `NamedTuple`s in place of alists. For larger collections of homogeneous types, `Array`s and `Dict`s should be used.
- The typical Julia workflow for prototyping also uses continuous manipulation of the image, implemented with the Revise.jl package.
- Bignums are supported, but conversion is not automatic; ordinary integers overflow.
- Modules (namespaces) can be hierarchical. `import` and `using` have a dual role: they load the code and make it available in the namespace. `import` for only the module name is possible (roughly equivalent to `ASDF:LOAD-OP`). Slot names don't need to be exported separately. Global variables can't be assigned to from outside the module (except with `eval(mod, :(var = val))` as an escape hatch).
- Macros start with `@`, and are not as seamlessly integrated into the language as Common Lisp; consequently, macro usage is not as widespread as in the latter. A form of hygiene for macros is supported by the language. Because of the different surface syntax, there is no equivalent to `COMMON-LISP:&BODY`.
- *All* functions are generic and use multiple dispatch. Argument lists don't have to follow the same template, which leads to a powerful idiom (see `do`). Optional and keyword arguments are handled differently. Method ambiguities are not resolved like in the Common Lisp Object System, necessitating the definition of a more specific method for the intersection.
- Symbols do not belong to any package, and do not contain any values *per se*. `M.var` evaluates the symbol `var` in the module `M`.
- A functional programming style is fully supported by the language, including closures, but isn't always the idiomatic solution for Julia. Some workarounds may be necessary for performance when modifying captured variables.

« Frequently Asked Questions Unicode Input »