Interactive Utilities

`Base.Docs.apropos` — Function

```
apropos(string)
```

Search through all documentation for a string, ignoring case.

`InteractiveUtils.varinfo` — Function

```
varinfo(m::Module=Main, pattern::Regex=r"")
```

Return a markdown table giving information about exported global variables in a module, optionally restricted to those matching `pattern`.

The memory consumption estimate is an approximate lower bound on the size of the internal structure of the object.

`InteractiveUtils.versioninfo` — Function

```
versioninfo(io::IO=stdout; verbose::Bool=false)
```

Print information about the version of Julia in use. The output is controlled with boolean keyword arguments:

- `verbose`: print all additional information

`InteractiveUtils.methodswith` — Function

```
methodswith(typ[, module or function]; supertypes::Bool=false])
```

Return an array of methods with an argument of type `typ`.

The optional second argument restricts the search to a particular module or function (the default is all top-level modules).

If keyword `supertypes` is `true`, also return arguments with a parent type of `typ`, excluding type `Any`.

`InteractiveUtils.subtypes` — Function

```
subtypes(T::DataType)
```

Return a list of immediate subtypes of DataType `T`. Note that all currently loaded subtypes are included, including those not visible in the current module.

Examples

```
julia> subtypes(Integer)
3-element Array{Any,1}:
 Bool
 Signed
 Unsigned
```

`InteractiveUtils.supertypes` — Function

```
supertypes(T::Type)
```

Return a tuple `(T, ..., Any)` of `T` and all its supertypes, as determined by successive calls to the `supertype` function, listed in order of `<:` and terminated by `Any`.

Examples

```
julia> supertypes(Int)
(Int64, Signed, Integer, Real, Number, Any)
```

`InteractiveUtils.edit` — Method

```
edit(path::AbstractString, line::Integer=0)
```

Edit a file or directory optionally providing a line number to edit the file at. Return to the `julia` prompt when you quit the editor. The editor can be changed by setting `JULIA_EDITOR`, `VISUAL` or `EDITOR` as an environment variable.

See also: (`define_editor`)[@ref]

`InteractiveUtils.edit` — Method

```
edit(function, [types])
edit(module)
```

Edit the definition of a function, optionally specifying a tuple of types to indicate which method to edit. For modules, open the main source file. The module needs to be loaded with `using` or `import` first.

Julia 1.1

`edit` on modules requires at least Julia 1.1.

To ensure that the file can be opened at the given line, you may need to call `define_editor` first.

`InteractiveUtils.@edit` — Macro

```
@edit
```

Evaluates the arguments to the function or macro call, determines their types, and calls the `edit` function on the resulting expression.

`InteractiveUtils.define_editor` — Function

```
define_editor(fn, pattern; wait=false)
```

Define a new editor matching `pattern` that can be used to open a file (possibly at a given line number) using `fn`.

The `fn` argument is a function that determines how to open a file with the given editor. It should take three arguments, as follows:

- `cmd` - a base command object for the editor
- `path` - the path to the source file to open
- `line` - the line number to open the editor at

Editors which cannot open to a specific line with a command may ignore the `line` argument. The `fn` callback must return either an appropriate `Cmd` object to open a file or `nothing` to indicate that they cannot edit this file. Use `nothing` to indicate that this editor is not appropriate for the current environment and another editor should be attempted. It is possible to add more general editing hooks that need not spawn external commands by pushing a callback directly to the vector `EDITOR_CALLBACKS`.

The `pattern` argument is a string, regular expression, or an array of strings and regular expressions. For the `fn` to be called, one of the patterns must match the value of `EDITOR`, `VISUAL` or `JULIA_EDITOR`. For strings, the string must equal the `basename` of the first word of the editor command, with its extension, if any, removed. E.g. "vi" doesn't match "vim -g" but matches "/usr/bin/vi -m"; it also matches `vi.exe`. If `pattern` is a regex it is matched against all of the editor command as a shell-escaped string. An array pattern matches if any of its items match. If multiple editors match, the one added most recently is used.

By default julia does not wait for the editor to close, running it in the background. However, if the editor is terminal based, you will probably want to set `wait=true` and julia will wait for the editor to close before resuming.

If one of the editor environment variables is set, but no editor entry matches it, the default editor entry is invoked:

```
(cmd, path, line) -> `$cmd $path`
```

Note that many editors are already defined. All of the following commands should already work:

- emacs
- emacsclient
- vim
- nvim
- nano

- textmate
- mate
- kate
- subl
- atom
- notepad++
- Visual Studio Code
- open
- pycharm
- bbedit

Example:

The following defines the usage of terminal-based `emacs`:

```
define_editor(
    r"\bemacs\b.*\s(-nw|--no-window-system)\b", wait=true) do cmd, path, line
    `$cmd +$line $path`
end
```

Julia 1.4

`define_editor` was introduced in Julia 1.4.

`InteractiveUtils.less` — Method

```
less(file::AbstractString, [line::Integer])
```

Show a file using the default pager, optionally providing a starting line number. Returns to the `julia` prompt when you quit the pager.

`InteractiveUtils.less` — Method

```
less(function, [types])
```

Show the definition of a function using the default pager, optionally specifying a tuple of types to indicate which method to see.

`InteractiveUtils.@less` — Macro

```
@less
```

Evaluates the arguments to the function or macro call, determines their types, and calls the `less` function on the resulting expression.

`InteractiveUtils.@which` — Macro

```
@which
```

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns the `Method` object for the method that would be called for those arguments. Applied to a variable, it returns the module in which the variable was bound. It calls out to the `which` function.

`InteractiveUtils.@functionloc` — Macro

```
@functionloc
```

Applied to a function or macro call, it evaluates the arguments to the specified call, and returns a tuple `(filename,line)` giving the location for the method that would be called for those arguments. It calls out to the `functionloc` function.

`InteractiveUtils.@code_lowered` — Macro

```
@code_lowered
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_lowered` on the resulting expression.

`InteractiveUtils.@code_typed` — Macro

```
@code_typed
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_typed` on the resulting expression. Use the optional argument `optimize` with

```
@code_typed optimize=true foo(x)
```

to control whether additional optimizations, such as inlining, are also applied.

`InteractiveUtils.code_warntype` — Function

```
code_warntype([io::IO], f, types; debuginfo=:default)
```

Prints lowered and type-inferred ASTs for the methods matching the given generic function and type signature to `io` which defaults to `stdout`. The ASTs are annotated in such a way as to cause "non-leaf" types to be emphasized (if color is available, displayed in red). This serves as a warning of potential type instability. Not all non-leaf types are particularly problematic for performance, so the results need to be used judiciously. In particular, unions containing either `missing` or `nothing` are displayed in yellow, since these are often intentional.

Keyword argument `debuginfo` may be one of `:source` or `:none` (default), to specify the verbosity of code comments.

See `@code_warntype` for more information.

`InteractiveUtils.@code_warntype` — Macro

```
@code_warntype
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_warntype` on the resulting expression.

InteractiveUtils.code_llvm — Function

```
code_llvm([io=stdout,], f, types; raw=false, dump_module=false, optimize=true,
```

Prints the LLVM bitcodes generated for running the method matching the given generic function and type signature to `io`.

If the `optimize` keyword is unset, the code will be shown before LLVM optimizations. All metadata and dbg.* calls are removed from the printed bitcode. For the full IR, set the `raw` keyword to true. To dump the entire module that encapsulates the function (with declarations), set the `dump_module` keyword to true. Keyword argument `debuginfo` may be one of source (default) or none, to specify the verbosity of code comments.

InteractiveUtils.@code_llvm — Macro

```
@code_llvm
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_llvm` on the resulting expression. Set the optional keyword arguments `raw`, `dump_module`, `debuginfo`, `optimize` by putting them and their value before the function call, like this:

```
@code_llvm raw=true dump_module=true debuginfo=:default f(x)
@code_llvm optimize=false f(x)
```

`optimize` controls whether additional optimizations, such as inlining, are also applied. `raw` makes all metadata and dbg.* calls visible. `debuginfo` may be one of `:source` (default) or `:none`, to specify the verbosity of code comments. `dump_module` prints the entire module that encapsulates the function.

InteractiveUtils.code_native — Function

```
code_native([io=stdout,], f, types; syntax=:att, debuginfo=:default)
```

Prints the native assembly instructions generated for running the method matching the given generic function and type signature to `io`. Switch assembly syntax using `syntax` symbol parameter set to `:att` for AT&T syntax or `:intel` for Intel syntax. Keyword argument `debuginfo` may be one of source (default) or none, to specify the verbosity of code comments.

InteractiveUtils.@code_native — Macro

```
@code_native
```

Evaluates the arguments to the function or macro call, determines their types, and calls `code_native` on the resulting expression.

Set the optional keyword argument `debuginfo` by putting it before the function call, like this:

```
@code_native debuginfo=:default f(x)
```

`debuginfo` may be one of `:source` (default) or `:none`, to specify the verbosity of code comments.

InteractiveUtils.clipboard — Function

```
clipboard(x)
```

Send a printed form of `x` to the operating system clipboard ("copy").

```
clipboard() -> AbstractString
```

Return a string with the contents of the operating system clipboard ("paste").

« Future LibGit2 »

Powered by Documenter.jl and the Julia Programming Language.