Nushell Book

Nushell Book

Nushell Team

Chapter 1

Introduction

Hello, and welcome to the Nushell project. The goal of this project is to take the Unix philosophy of shells, where pipes connect simple commands together, and bring it to the modern style of development. Thus, rather than being either a shell, or a programming language, Nushell connects both by bringing a rich programming language and a full-featured shell together into one package.

Nu takes cues from a lot of familiar territory: traditional shells like bash, object based shells like PowerShell, gradually typed languages like TypeScript, functional programming, systems programming, and more. But rather than trying to be a jack of all trades, Nu focuses its energy on doing a few things well:

- Being a flexible cross-platform shell with a modern feel
- Solving problems as a modern programming language that works with the structure of your data
- Giving clear error messages and clean IDE support

This Book

The book is split into chapters which are further broken down into sections. You can click on the chapter headers to get more information about it.

- Getting Started teaches you how to install Nushell and shows you the ropes. It also explains some of the design principles where Nushell differs from typical shells, such as bash.

- Nu Fundamentals explains basic concepts of the Nushell language.
- Programming in Nu dives more deeply into the language features and shows several ways how to organize and structure your code.
- Nu as a Shell focuses on the shell features, most notably the configuration and environment.
- Coming to Nu is intended to give a quick start for users coming from other shells or languages.
- Design Notes has in-depth explanation of some of the Nushell's design choices.
- (Not So) Advanced includes some more advanced topics (they are not *so* advanced, make sure to check them out, too!).

The Many Parts of Nushell

The Nushell project consists of multiple different repositories and subprojects. You can find all of them under our organization on GitHub[1].

- The main Nushell repository can be found here[2]. It is broken into multiple crates that can be used as independent libraries in your own project, if you wish so.
- The repository of our nushell.sh[3] page, including this book, can be found here[4].
- Nushell has its own line editor which has its own repository[5]
- `nu_scripts`[6] is a place to share scripts and modules with other users until we have some sort of package manager.
- Nana[7] is an experimental effort to explore graphical user interface for Nushell.

[1]https://github.com/nushell
[2]https://github.com/nushell/nushell
[3]https://www.nushell.sh
[4]https://github.com/nushell/nushell.github.io
[5]https://github.com/nushell/reedline
[6]https://github.com/nushell/nu_scripts
[7]https://github.com/nushell/nana

- Awesome Nu[8] contains a list of tools that work with the Nushell ecosystem: plugins, scripts, editor extension, 3rd party integrations, etc.
- Nu Showcase[9] is a place to share works about Nushell, be it blogs, artwork or something else.
- Request for Comment (RFC)[10] serves as a place to propose and discuss major design changes. While currently under-utilized, we expect to use it more as we get closer to and beyond 1.0.

Contributing

We welcome contributions! As you can see[11], there are a lot of places to contribute to. Most repositories contain `CONTRIBUTING.md` file with tips and details that should help you get started (if not, consider contributing a fix!).

Nushell itself is written in Rust[12]. However, you do not have to be a Rust programmer to help. If you know some web development, you can contribute to improving this website or the Nana project. Dataframes can use your data processing expertise.

If you wrote a cool script, plugin or integrated Nushell somewhere, we'd welcome your contribution to `nu_scripts` or Awesome Nu. Discovering bugs with reproduction steps and filing GitHub issues for them is a valuable help, too! You can contribute to Nushell just by using Nushell!

Since Nushell evolves fast, this book is in a constant need of updating. Contributing to this book does not require any special skills aside from a basic familiarity with Markdown. Furthermore, you can consider translating parts of it to your language.

Community

The main place to discuss anything Nushell is our Discord[13]. You can also follow us on Twitter[14] for news and updates. Finally, you can use

[8] https://github.com/nushell/awesome-nu
[9] https://github.com/nushell/showcase
[10] https://github.com/nushell/rfcs
[11] #the-many-parts-of-nushell
[12] https://www.rust-lang.org
[13] https://discord.com/invite/NtAbbGn
[14] https://twitter.com/nu_shell

the GitHub discussions or file GitHub issues.

Chapter 2

Getting Started

Let's get started! :elephant:

First, to be able to use Nushell, we need to install it.

The next sections will give you a short tour of Nushell by example (including how to get help from within Nushell), and show you how to move around your file system.

Finally, because Nushell takes some design decisions that are quite different from typical shells or dynamic scripting languages, make sure to check Thinking in Nu that explains some of these concepts.

Installing Nu

There are lots of ways to get Nu up and running. You can download pre-built binaries from our release page[1], use your favourite package manager[2], or build from source.

The main Nushell binary is named `nu` (or `nu.exe` on Windows). After installation, you can launch it by typing `nu`.

Pre-built binaries

Nu binaries are published for Linux, macOS, and Windows with each GitHub release[3]. Just download, extract the binaries, then copy them to a location on your PATH.

[1] https://github.com/nushell/nushell/releases
[2] https://repology.org/project/nushell/versions
[3] https://github.com/nushell/nushell/releases

Package managers

Nu is available via several package managers:

[4]For macOS and Linux, Homebrew[5] is a popular choice (`brew install nushell`).

For Windows:

- Winget[6] (`winget install nushell`)
- Chocolatey[7] (`choco install nushell`)
- Scoop[8] (`scoop install nu`)

Cross Platform installation:

- npm[9] (`npm install -g nushell` Note that nu plugins are not included if you install in this way)

Build from source

You can also build Nu from source. First, you will need to set up the Rust toolchain and its dependencies.

Installing a compiler suite

For Rust to work properly, you'll need to have a compatible compiler suite installed on your system. These are the recommended compiler suites:

- Linux: GCC or Clang
- macOS: Clang (install Xcode)
- Windows: MSVC (install Visual Studio[10] or the Visual Studio Build Tools[11])
  - Make sure to install the "Desktop development with C++" workload
  - Any Visual Studio edition will work (Community is free)

[4]`https://repology.org/project/nushell/versions`
[5]`https://brew.sh/`
[6]`https://docs.microsoft.com/en-us/windows/package-manager/winget/`
[7]`https://chocolatey.org/`
[8]`https://scoop.sh/`
[9]`https://www.npmjs.com/`
[10]`https://visualstudio.microsoft.com/vs/community/`
[11]`https://visualstudio.microsoft.com/downloads/#build-tools-for-visual-studio-2022`

Installing Rust

If you don't already have Rust on our system, the best way to install it is via rustup[12]. Rustup is a way of managing Rust installations, including managing using different Rust versions.

Nu currently requires the **latest stable (1.66.1 or later)** version of Rust. The best way is to let `rustup` find the correct version for you. When you first open `rustup` it will ask what version of Rust you wish to install:

Once you are ready, press 1 and then enter.

If you'd rather not install Rust via `rustup`, you can also install it via other methods (e.g. from a package in a Linux distro). Just be sure to install a version of Rust that is 1.66.1 or later.

Dependencies

Debian/Ubuntu You will need to install the "pkg-config" and "libssl-dev" package:

RHEL based distros You will need to install "libxcb", "openssl-devel" and "libX11-devel":

macOS Using Homebrew[13], you will need to install "openssl" and "cmake" using:

Build using crates.io[14]

Nu releases are published as source to the popular Rust package registry crates.io[15]. This makes it easy to build and install the latest Nu release with `cargo`:

That's it! The `cargo` tool will do the work of downloading Nu and its source dependencies, building it, and installing it into the cargo bin path.

If you want to install with support for dataframes, you can install using the `--features=dataframe` flag.

[12]https://rustup.rs/
[13]https://brew.sh/
[14]https://crates.io
[15]https://crates.io/

Building from the GitHub repository

You can also build Nu from the latest source on GitHub. This gives you immediate access to the latest features and bug fixes. First, clone the repo:

From there, we can build and run Nu with:

You can also build and run Nu in release mode, which enables more optimizations:

People familiar with Rust may wonder why we do both a “build” and a “run” step if “run” does a build by default. This is to get around a shortcoming of the new `default-run` option in Cargo, and ensure that all plugins are built, though this may not be required in the future.

Default shell

Setting Nu as default shell on your terminal

Terminal	Platform	Instructions
GNOME Terminal	Linux & BSDs	Open `Edit > Preferences`. In the right-hand panel, select the `Command` tab, tick `Run a custom command instead of my shell`, and set `Custom command` to the path to Nu.
GNOME Console	Linux & BSDs	Type the command `gsettings set org.gnome.Console shell "['/usr/bin/nu']"` (replace `/usr/bin/nu` with the path to Nu). Equivalently, use dconf Editor[16] to edit the `/org/gnome/Console/shell` key.
Konsole	Linux & BSDs	Open `Settings > Edit Current Profile`. Set `Command` to the path to Nu.
XFCE Terminal	Linux & BSDs	Open `Edit > Preferences`. Check `Run a custom command instead of my shell`, and set `Custom command` to the path to Nu.
Terminal.app	macOS	Open `Terminal > Preferences`. Ensure you are on the `Profiles` tab, which should be the default tab. In the right-hand panel,

Setting Nu as login shell (Linux, BSD & macOS)

::: warning Nu is still in development and is not intended to be POSIX compliant. Be aware that some programs on your system might assume that your login shell is POSIX[17] compatible. Breaking that assumption can lead to unexpected issues. :::

To set the login shell you can use the `chsh`[18] command. Some Linux distributions have a list of valid shells located in `/etc/shells` and will disallow changing the shell until Nu is in the whitelist. You may see an error similar to the one below if you haven't updated the `shells` file:

You can add Nu to the list of allowed shells by appending your Nu binary to the `shells` file. The path to add can be found with the command `which nu`, usually it is `$HOME/.cargo/bin/nu`.

Quick Tour

The easiest way to see what Nu can do is to start with some examples, so let's dive in.

The first thing you'll notice when you run a command like `ls`[19] is that instead of a block of text coming back, you get a structured table.

The table does more than show the directory in a different way. Just like tables in a spreadsheet, this table allows us to work with the data more interactively.

The first thing we'll do is to sort our table by size. To do this, we'll take the output from `ls`[20] and feed it into a command that can sort tables based on the contents of a column.

You can see that to make this work we didn't pass commandline arguments to `ls`[21]. Instead, we used the `sort-by`[22] command that Nu provides to do the sorting of the output of the `ls`[23] command. To see the biggest files on top, we also used `reverse`[24].

Nu provides many commands that can work on tables. For example, we could use `where`[25] to filter the contents of the `ls`[26] table so that it

[17] https://en.wikipedia.org/wiki/POSIX
[18] https://linux.die.net/man/1/chsh
[19] /commands/docs/ls.md
[20] /commands/docs/ls.md
[21] /commands/docs/ls.md
[22] /commands/docs/sort-by.md
[23] /commands/docs/ls.md
[24] /commands/docs/reverse.md
[25] /commands/docs/where.md
[26] /commands/docs/ls.md

only shows files over 1 kilobyte:

Just as in the Unix philosophy, being able to have commands talk to each other gives us ways to mix-and-match in many different combinations. Let's look at a different command:

You may be familiar with the `ps`[27] command if you've used Linux. With it, we can get a list of all the current processes that the system is running, what their status is, and what their name is. We can also see the CPU load for the processes.

What if we wanted to show the processes that were actively using the CPU? Just like we did with the `ls`[28] command earlier, we can also work with the table that the `ps`[29] command gives back to us:

So far, we've been using `ls`[30] and `ps`[31] to list files and processes. Nu also offers other commands that can create tables of useful information. Next, let's explore `date`[32] and `sys`[33].

Running `date now`[34] gives us information about the current day and time:

To get the date as a table we can feed it into `date to-table`[35]

Running `sys`[36] gives information about the system that Nu is running on:

This is a bit different than the tables we saw before. The `sys`[37] command gives us a table that contains structured tables in the cells instead of simple values. To take a look at this data, we need to *get* the column to view:

The `get`[38] command lets us jump into the contents of a column of the table. Here, we're looking into the "host" column, which contains information about the host that Nu is running on. The name of the OS, the hostname, the CPU, and more. Let's get the name of the users on the system:

Right now, there's just one user on the system named "jt". You'll notice that we can pass a column path (the `host.sessions.name` part)

[27] /commands/docs/ps.md
[28] /commands/docs/ls.md
[29] /commands/docs/ps.md
[30] /commands/docs/ls.md
[31] /commands/docs/ps.md
[32] /commands/docs/date.md
[33] /commands/docs/sys.md
[34] /commands/docs/date_now.md
[35] /commands/docs/date_to-table.md
[36] /commands/docs/sys.md
[37] /commands/docs/sys.md
[38] /commands/docs/get.md

and not just the name of the column. Nu will take the column path and go to the corresponding bit of data in the table.

You might have noticed something else that's different. Rather than having a table of data, we have just a single element: the string "jt". Nu works with both tables of data as well as strings. Strings are an important part of working with commands outside of Nu.

Let's see how strings work outside of Nu in action. We'll take our example from before and run the external `echo`[39] command (the ^ tells Nu to not use the built-in `echo`[40] command):

If this looks very similar to what we had before, you have a keen eye! It is similar, but with one important difference: we've called `^echo` with the value we saw earlier. This allows us to pass data out of Nu into `echo`[41] (or any command outside of Nu, like `git` for example).

Getting Help

Help text for any of Nu's built-in commands can be discovered with the `help`[42] command. To see all commands, run `help commands`[43]. You can also search for a topic by doing `help -f <topic>`.

Moving around your system

Early shells allow you to move around your filesystem and run commands, and modern shells like Nu allow you to do the same. Let's take a look at some of the common commands you might use when interacting with your system.

Viewing directory contents

As we've seen in other chapters, `ls`[44] is a command for viewing the contents of a path. Nu will return the contents as a table that we can use.

The `ls`[45] command also takes an optional argument, to change what you'd like to view. For example, we can list the files that end in ".md"

[39] /commands/docs/echo.md
[40] /commands/docs/echo.md
[41] /commands/docs/echo.md
[42] /commands/docs/help.md
[43] /commands/docs/help_commands.md
[44] /commands/docs/ls.md
[45] /commands/docs/ls.md

Glob patterns (wildcards)

The asterisk (*) in the above optional argument "*.md" is sometimes called a wildcard or a glob. It lets us match anything. You could read the glob "*.md" as "match any filename, so long as it ends with '.md' "

The most general glob is `*`, which will match all paths. More often, you'll see this pattern used as part of another pattern, for example `*.bak` and `temp*`.

In Nushell, we also support double `*` to talk about traversing deeper paths that are nested inside of other directories. For example, `ls **/*` will list all the non-hidden paths nested under the current directory.

Here, we're looking for any file that ends with ".md", and the two asterisks further say "in any directory starting from here".

In other shells (like bash), glob expansion happens in the shell and the invoked program (`ls` in the example above) receives a list of matched files. In Nushell however, the string you enter is passed "as is" to the command, and some commands (like `ls`, `mv`, `cp` and `rm`) interpret their input string as a glob pattern. For example the `ls` command's help page[46] shows that it takes the parameter: `pattern: the glob pattern to use (optional)`.

Globbing syntax in these commands not only supports `*`, but also matching single characters with `?` and character groups with `[...]`[47]. Note that this is a more limited syntax than what the dedicated `glob` Nushell command[48] supports.

Escaping `*`, `?`, `[]` works by quoting them with single quotes or double quotes. To show the contents of a directory named `[slug]`, use `ls "[slug]"` or `ls '[slug]'`. Note that backtick quote doesn't escape glob, for example: <code>cp `test dir/*`</code> will copy all files inside `test dir` to current direcroty.

Changing the current directory

To change from the current directory to a new one, we use the `cd`[49] command. Just as in other shells, we can use either the name of the directory, or if we want to go up a directory we can use the `..` shortcut.

Changing the current working directory can also be done if `cd`[50] is omitted and a path by itself is given:

[46] https://www.nushell.sh/commands/docs/ls.html
[47] https://docs.rs/nu-glob/latest/nu_glob/struct.Pattern.html
[48] https://www.nushell.sh/commands/docs/glob.html
[49] /commands/docs/cd.md
[50] /commands/docs/cd.md

Note: changing the directory with `cd`[51] changes the `PWD` environment variable. This means that a change of a directory is kept to the current block. Once you exit the block, you'll return to the previous directory. You can learn more about working with this in the environment chapter[52].

Filesystem commands

Nu also provides some basic filesystem commands that work cross-platform.

We can move an item from one place to another using the `mv`[53] command:

We can copy an item from one location to another with the `cp`[54] command:

We can remove an item with the `rm`[55] command:

The three commands also can use the glob capabilities we saw earlier with `ls`[56].

Finally, we can create a new directory using the `mkdir`[57] command:

Thinking in Nu

To help you understand - and get the most out of - Nushell, we've put together this section on "thinking in Nushell". By learning to think in Nushell and use the patterns it provides, you'll hit fewer issues getting started and be better setup for success.

So what does it mean to think in Nushell? Here are some common topics that come up with new users of Nushell.

Nushell isn't bash

Nushell is both a programming language and a shell. Because of this, it has its own way of working with files, directories, websites, and more. We've modeled this to work closely with what you may be familiar with other shells. Pipelines work by attaching two commands together:

[51] `/commands/docs/cd.md`
[52] `./environment.md`
[53] `/commands/docs/mv.md`
[54] `/commands/docs/cp.md`
[55] `/commands/docs/rm.md`
[56] `/commands/docs/ls.md`
[57] `/commands/docs/mkdir.md`

```
> ls | length
```

Nushell, for example, also has support for other common capabilities like getting the exit code from previously run commands.

While it does have these amenities, Nushell isn't bash. The bash way of working, and the POSIX style in general, is not one that Nushell supports. For example, in bash, you might use:

```
> echo " hello " > output.txt
```

In Nushell, we use the `>` as the greater-than operator. This fits better with the language aspect of Nushell. Instead, you pipe to a command that has the job of saving content:

```
> "hello" | save output.txt
```

Thinking in Nushell: The way Nushell views data is that data flows through the pipeline until it reaches the user or is handled by a final command. You can simply type data, from strings to JSON-style lists and records, and follow it with `|` to send it through the pipeline. Nushell uses commands to do work and produce more data. Learning these commands and when to use them helps you compose many kinds of pipelines.

Think of Nushell as a compiled language

An important part of Nushell's design and specifically where it differs from many dynamic languages is that Nushell converts the source you give it into something to run, and then runs the result. It doesn't have an `eval` feature which allows you to continue pulling in new source during runtime. This means that tasks like including files to be part of your project need to be known paths, much like includes in compiled languages like C++ or Rust.

For example, the following doesn't make sense in Nushell, and will fail to execute if run as a script:

```
"def abc [] { 1 + 2 }" | save output.nu source "output.
nu" abc
```

The `source`[58] command will grow the source that is compiled, but the `save`[59] from the earlier line won't have had a chance to run. Nushell

[58] /commands/docs/source.md
[59] /commands/docs/save.md

runs the whole block as if it were a single file, rather than running one line at a time. In the example, since the output.nu file is not created until after the 'compilation' step, the `source`[60] command is unable to read definitions from it during parse time.

Another common issue is trying to dynamically create the filename to source from:

```
> source $"($my_path)/common.nu"
```

This doesn't work if `my_path` is a regular runtime variable declared with `let`. This would require the evaluator to run and evaluate the string, but unfortunately Nushell needs this information at compile-time.

However, if `my_path` is a constant[61], then this would work, since the string can be evaluated at compile-time:

```
> const my_path = ([$nu.home-path nushell] | path join)
 > source $"($my_path)/common.nu" # sources /home/user/
nushell/common.nu
```

Thinking in Nushell: Nushell is designed to use a single compile step for all the source you send it, and this is separate from evaluation. This will allow for strong IDE support, accurate error messages, an easier language for third-party tools to work with, and in the future even fancier output like being able to compile Nushell directly to a binary file.

For more in-depth explanation, check How Nushell Code Gets Run.

Variables are immutable

Another common surprise for folks coming from other languages is that Nushell variables are immutable (and indeed some people have started to call them "constants" to reflect this). Coming to Nushell you'll want to spend some time becoming familiar with working in a more functional style, as this tends to help write code that works best with immutable variables.

You might wonder why Nushell uses immutable variables. Early on in Nushell's development we decided to see how long we could go using a more data-focused, functional style in the language. More recently, we added a key bit of functionality into Nushell that made these early experiments show their value: parallelism. By switching from

60 `/commands/docs/source.md`
61 `/book/variables_and_subexpressions#constant-variables`

each[62] to par-each[63] in any Nushell script, you're able to run the corresponding block of code in parallel over the input. This is possible because Nushell's design leans heavily on immutability, composition, and pipelining.

Just because Nushell variables are immutable doesn't mean things don't change. Nushell makes heavy use of the technique of "shadowing". Shadowing means creating a new variable with the same name as a previously declared variable. For example, say you had an $x in scope, and you wanted a new $x that was one greater:

```
let x = $x + 1
```

This new x is visible to any code that follows this line. Careful use of shadowing can make for an easier time working with variables, though it's not required.

Loop counters are another common pattern for mutable variables and are built into most iterating commands, for example you can get both each item and an index of each item using each[64]:

```
> ls | enumerate | each { |it| $"Number ($it.index) is
size ($it.item.size)" }
```

You can also use the reduce[65] command to work in the same way you might mutate a variable in a loop. For example, if you wanted to find the largest string in a list of strings, you might do:

```
> [one, two, three, four, five, six] | reduce {|curr, max|
    if ($curr | str length) > ($max | str length) {
     $curr     } else {          $max     } }
```

Thinking in Nushell: If you're used to using mutable variables for different tasks, it will take some time to learn how to do each task in a more functional style. Nushell has a set of built-in capabilities to help with many of these patterns, and learning them will help you write code in a more Nushell-style. The added benefit of speeding up your scripts by running parts of your code in parallel is a nice bonus.

Nushell's environment is scoped

Nushell takes multiple design cues from compiled languages. One such cue is that languages should avoid global mutable state. Shells have

[62] /commands/docs/each.md
[63] /commands/docs/par-each.md
[64] /commands/docs/each.md
[65] /commands/docs/reduce.md

commonly used global mutation to update the environment, but Nushell steers clear of this approach.

In Nushell, blocks control their own environment. Changes to the environment are scoped to the block where they happen.

In practice, this lets you write some concise code for working with subdirectories, for example, if you wanted to build each sub-project in the current directory, you could run:

```
> ls | each { |it|     cd $it.name     make }
```

The `cd`[66] command changes the `PWD` environment variables, and this variable change does not escape the block, allowing each iteration to start from the current directory and enter the next subdirectory.

Having the environment scoped like this makes commands more predictable, easier to read, and when the time comes, easier to debug. Nushell also provides helper commands like `def --env`[67], `load-env`[68], as convenient ways of doing batches of updates to the environment.

There is one exception here, where `def --env`[69] *allows you to create a command that participates in the caller's environment.*

Thinking in Nushell: - The coding best practice of no global mutable variables extends to the environment in Nushell. Using the built-in helper commands will let you more easily work with the environment in Nushell. Taking advantage of the fact that environments are scoped to blocks can also help you write more concise scripts and interact with external commands without adding things into a global environment you don't need.

Nushell cheat sheet

Data types

```
    > "12" | into int
```

converts string to integer

***

[66] /commands/docs/cd.md
[67] /commands/docs/def.md
[68] /commands/docs/load-env.md
[69] /commands/docs/def.md

```
    > date now | date to-timezone "Europe/London"
```

converts present date to provided time zone

***

```
    > {'name': 'nu', 'stars': 5, 'language': 'Python'}
| upsert language 'Rust'
```

updates a record's language and if none is specified inserts provided value

***

```
    > [one two three] | to yaml
```

converts list of strings to yaml

***

```
    > [[framework, language]; [Django, Python] [Lavarel,
PHP]]
```

prints the table

***

```
    > [{name: 'Robert' age: 34 position: 'Designer'}
   {name: 'Margaret' age: 30 position: 'Software Developer'}
     {name: 'Natalie' age: 50 position: 'Accountant'}
   ] | select name position
```

selects two columns from the table and prints their values

Strings

```
> let name = "Alice"    > $"greetings, ($name)!"
```

prints **`greetings, Alice!`**

***

```
> let string_list = "one,two,three" | split row ","
$string_list                    0   one          1
two          2   three
```

splits the string with specified delimiter and saves the list to **`string_list`** *variable*

***

```
"Hello, world!" | str contains "o, w"
```

checks if a string contains a substring and returns **`boolean`**

***

```
let str_list = [zero one two]    $str_list | str join
','
```

joins the list of strings using provided delimiter

***

```
    > 'Hello World!' | str substring 4..8
```

created a slice from a given string with start (4) and end (8) indices

***

```
    > 'Nushell 0.80' | parse '{shell} {version}'
      #    shell    version                           0
Nushell   0.80
```

parses the string to columns

```
    > "acronym,long\nAPL,A Programming Language" | from
csv
```

parses comma separated values (csv)

```
    > $'(ansi purple_bold)This text is a bold purple!(ansi
reset)'
```

ansi command colors the text (alsways end with `ansi reset` to reset color to default)

Lists

```
    > [foo bar baz] | insert 1 'beeze'
  0   foo          1   beeze        2   bar           3
baz
```

inserts `beeze` value at st index in the list

***

```
    > [1, 2, 3, 4] | update 1 10
```

updates 2nd value to 10

***

```
    > let numbers = [1, 2, 3, 4, 5]     > $numbers | prepend
0
```

adds value at the beginning of the list

***

```
    > let numbers = [1, 2, 3, 4, 5]     > $numbers | append
6
```

adds value at the end of the list

***

```
    > let flowers = [cammomile marigold rose forget-me-
not]     > let flowers = ($flowers | first 2)     > $flowers
```

creates slice of first two values from `flowers` list

***

```
    > let planets = [Mercury Venus Earth Mars Jupiter Saturn
Uranus Neptune]     > $planets | each { |it| $"($it) is
a planet of solar system" }
```

iterates over a list; `it` is current list value

***

```
    > $planets | enumerate | each { |it| $"($it.index +
1) - ($it.item)" }
```

iterates over a list and provides index and value in `it`

***

```
    > let scores = [3 8 4]     > $"total = ($scores | reduce
{ |it, acc| $acc + $it })"
```

reduces the list to a single value, `reduce` gives access to accumulator that is applied to each element in the list

***

```
    > $"total = ($scores | reduce --fold 1 { |it, acc|
$acc * $it })"
```

initial value for accumulator value can be set with `--fold`

***

```
    > let planets = [Mercury Venus Earth Mars Jupiter Saturn
Uranus Neptune]     > $planets.2     > Earth
```

gives access to the 3rd item in the list

***

```
    > let planets = [Mercury Venus Earth Mars Jupiter Saturn
Uranus Neptune]     > $planets | any {|it| $it | str starts-
with "E" }     > true
```

checks if any string in the list starts with **`E`**

***

```
    > let cond = {|x| $x < 0 }; [-1 -2 9 1] | take while
$cond                0   -1        1   -2
```

creates slice of items that satisfy provided condition

Tables

```
    > ls | sort-by size
```

sorting table by size of files

***

```
    > ls | sort-by size | first 5
```

sorting table by size of files and show first 5 entries

***

```
    > let $a = [[first_column second_column third_column];
[foo bar snooze]]    > let $b = [[first_column second_-
column third_column]; [hex seeze feeze]]    > $a | append
$b                                     #   first_column
  second_column   third_column
      0   foo            bar             snooze
     1   hex            seeze           feeze
```

concatenate two tables with same columns

***

```
    > let teams_scores = [[team score plays]; ['Boston
Celtics' 311 3] ['Golden State Warriors', 245 2]]    >
$teams_scores | drop column
   #          team            score
      0   Boston Celtics           311        1   Golden
State Warriors    245
```

remove the last column of a table

Files & Filesystem

```
    > start file.txt
```

opens a text file with the default text editor

***

```
    > 'lorem ipsum ' | save file.txt
```

saves a string to text file

***

```
> 'dolor sit amet' | save --append file.txt
```

appends a string to the end of file.txt

***

```
> { a: 1, b: 2 } | save file.json
```

saves a record to file.json

***

```
> glob **/*.{rs,toml} --depth 2
```

searches for `.rs` and `.toml` files recursively up to 2 folders deep

***

```
> watch . --glob=**/*.rs {|| cargo test }
```

runs cargo test whenever a Rust file changes

***

Custom Commands

```
def greet [name: string] {        $"hello ($name)"
}
```

custom command with parameter type set to string

***

```
    def greet [name = "nushell"] {        $"hello ($name-
)"    }
```

custom command with default parameter set to nushell

***

```
    def greet [         name: string         --age: int
    ] {         [$name $age]     }     > greet world --
age 10
```

passing named parameter by defining flag for custom commands

***

```
    def greet [         name: string         --age (-a)
: int         --twice     ] {         if $twice {
      [$name $age $name $age]         } else {
    [$name $age]         }     }     > greet -a 10 --twice
hello
```

using flag as a switch with a shorthand flag (-a) for the age

***

```
    def greet [...name: string] {         print "hello
all:"         for $n in $name {            print $n
     }     }     > greet earth mars jupiter venus
```

custom command which takes any number of positional arguments using rest params

Variables & Subexpressions

```
 > let val = 42     > print $val     42
```

an immutable variable cannot change its value after declaration

***

```
 > let val = 42     > do { let val = 101;  $val }
101     > $val     42
```

shadowing variable (declaring variable with the same name in a different scope)

***

```
 > mut val = 42     > $val += 27     > $val     69
```

declaring a mutable variable with mut key word

***

```
 > mut x = 0     > [1 2 3] | each { $x += 1 }
```

closures and nested defs cannot capture mutable variables from their environment. This expression results in error.

```
 > const plugin = 'path/to/plugin'     > register $plugin
```

a constant variable is immutable value which is fully evaluated at parse-time

***

```
    > let files = (ls)     > $files.name?.0?
```

using question mark operator to return null instead of error if provided path is incorrect

***

```
    > let big_files = (ls | where size > 10kb)     >  $big_-
files
```

using subexpression by wrapping the expression with parentheses ()

***

Modules

```
    > module greetings {        export def hello [name:
 string] {            $"hello ($name)!"        }
   export def hi [where: string] {            $"hi ($where)
!"        }    }     > use greetings hello     > hello
"world"
```

using inline module

***

```
    # greetings.nu    export-env {        $env.MYNAME
= "Arthur, King of the Britons"    }    export def hello
[] {        $"hello ($env.MYNAME)"    }    > use greeti-
ngs.nu    > $env.MYNAME    Arthur, King of the Britons
    > greetings hello    hello Arthur, King of the Britons!
```

importing module from file and using its environment in current scope

***

```
    # greetings.nu     export def hello [name: string]
{        $"hello ($name)!"    }    export def hi [where:
 string] {         $"hi ($where)!"     }    export def
main [] {         "greetings and salutations!"     }
  > use greetings.nu     > greetings     greetings and
salutations!     > greetings hello world    hello world!
```

using main command in module

***

Chapter 3

Nu Fundamentals

This chapter explains some of the fundamentals of the Nushell programming language. After going through it, you should have an idea how to write simple Nushell programs.

Nushell has a rich type system. You will find typical data types such as strings or integers and less typical data types, such as cell paths. Furthermore, one of the defining features of Nushell is the notion of *structured data* which means that you can organize types into collections: lists, records, or tables. Contrary to the traditional Unix approach where commands communicate via plain text, Nushell commands communicate via these data types. All of the above is explained in Types of Data.

Loading Data explains how to read common data formats, such as JSON, into *structured data*. This includes our own "NUON" data format.

Just like Unix shells, Nushell commands can be composed into pipelines to pass and modify a stream of data.

Some data types have interesting features that deserve their own sections: strings, lists, and tables. Apart from explaining the features, these sections also show how to do some common operations, such as composing strings or updating values in a list.

Finally, Command Reference[1] lists all the built-in commands with brief descriptions. Note that you can also access this info from within Nushell using the `help`[2] command.

[1] `/commands/`
[2] `/commands/docs/help.md`

Types of Data

Traditionally, Unix shell commands have communicated with each other using strings of text: one command would write text to standard output (often abbreviated 'stdout') and the other would read text from standard input (or 'stdin'), allowing the two commands to communicate.

Nu embraces this approach, and expands it to include other types of data, in addition to strings.

Like many programming languages, Nu models data using a set of simple, and structured data types. Simple data types include integers, floats, strings, booleans, dates. There are also special types for filesizes and time durations.

The `describe`[3] command returns the type of a data value:

```
> 42 | describe
```

[3] /commands/docs/describe.md

Types at a glance

Type	Example
Integers	`-65535`
Decimals (floats)	`9.9999`, `Infinity`
Strings	<code>“hole 18”, ‘hole 18’, ‘hole 18‘, hole18</code>
Booleans	`true`
Dates	`2000-01-01`
Durations	`2min + 12sec`
File sizes	`64mb`
Ranges	`0..4`, `0..<5`, `0..`, `..4`
Binary	`0x[FE FF]`
Lists	`[0 1 'two' 3]`
Records	`{name:"Nushell", lang: "Rust"}`
Tables	`[{x:12, y:15}, {x:8, y:9}]`, `[[x, y]; [12, 15], [8, 9]]`
Closures	`{|e| $e + 1 | into string }`, `{ $in.name.0 | path exists }`
Blocks	`if true { print "hello!" }`, `loop { print "press ctrl-c to exit" }`
Null	`null`

Integers

Examples of integers (i.e. “round numbers”) include 1, 0, -5, and 100. You can parse a string into an integer with the `into int`[4] command

```
> "-5" | into int
```

Decimals (floats)

Decimal numbers are numbers with some fractional component. Examples include 1.5, 2.0, and 15.333. You can cast a string into a Float with the `into float`[5] command

[4] `/commands/docs/into_int.md`
[5] `/commands/docs/into_float.md`

```
> "1.2" | into float
```

Strings

A string of characters that represents text. There are a few ways these can be constructed:

- Double quotes
  - `"Line1\nLine2\n"`
- Single quotes `'She said "Nushell is the future".'`
- Dynamic string interpolation
  - `$"6 x 7 = (6 * 7)"`
  - `ls | each { |it| $"($it.name) is ($it.size)" }`
- Bare strings
  - `print hello`
  - `[foo bar baz]`

See Working with strings and Handling Strings[6] for details.

Booleans

There are just two boolean values: `true` and `false`. Rather than writing the values directly, they often result from a comparison:

```
> let mybool = 2 > 1 > $mybool true > let mybool = ($env.
HOME | path exists) > $mybool true
```

Dates

Dates and times are held together in the Date value type. Date values used by the system are timezone-aware, and by default use the UTC timezone.

Dates are in three forms, based on the RFC 3339 standard:

- A date:

[6] https://www.nushell.sh/book/loading_data.html#handling-strings

    - 2022-02-02

- A date and time (in GMT):
    - 2022-02-02T14:30:00
- A date and time with timezone:
    - 2022-02-02T14:30:00+05:00

Durations

Durations represent a length of time. This chart shows all durations currently supported:

Duration	Length
`1ns`	one nanosecond
`1us`	one microsecond
`1ms`	one millisecond
`1sec`	one second
`1min`	one minute
`1hr`	one hour
`1day`	one day
`1wk`	one week

You can make fractional durations:

```
> 3.14day 3day 3hr 21min
```

And you can do calculations with durations:

```
> 30day / 1sec  # How many seconds in 30 days? 2592000
```

File sizes

Nushell also has a special type for file sizes. Examples include `100b`, `15kb`, and `100mb`.

The full list of filesize units are:

- `b`: bytes
- `kb`: kilobytes (aka 1000 bytes)
- `mb`: megabytes

- `gb`: gigabytes
- `tb`: terabytes
- `pb`: petabytes
- `eb`: exabytes
- `kib`: kibibytes (aka 1024 bytes)
- `mib`: mebibytes
- `gib`: gibibytes
- `tib`: tebibytes
- `pib`: pebibytes
- `eib`: exbibytes

As with durations, you can make fractional file sizes, and do calculations:

> `> 1Gb / 1b 1000000000 > 1Gib / 1b 1073741824 > (1Gib / 1b) == 2 ** 30 true`

Ranges

A range is a way of expressing a sequence of integer or float values from st
:: tip

You can also easily create lists of characters with a form similar to ranges
::

Specifying the step

You can specify the step of a range with the form <start>..<second>..<end>, where the step between values in the range is the distance between the <start> and <second> values, which numerically is <second> - <start>. For example, the range `2..5..11` means the numbers 2, 5, 8, and 11 because the step is <second> - <first> = 5 - 2 = 3. The third value is 5 + 3 = 8 and the fourth value is 8 + 3 = 11.

`seq`[9] can also create sequences of numbers, and provides an alternate way of specifying the step with three parameters. It's called with `seq $start $step $end` where the step amount is the second parameter rather than being the second parameter minus the first parameter. So `2..5..9` would be equivalent to `seq 2 3 9`.

[9] `/commands/docs/seq.md`

Inclusive and non-inclusive ranges

Ranges are inclusive by default, meaning that the ending value is counted as part of the range. The range `1..3` includes the number `3` as the last value in the range.

Sometimes, you may want a range that is limited by a number but doesn't use that number in the output. For this, you can use `..<` instead of `..`. For example, `1..<5` is the numbers 1, 2, 3, and 4.

Open-ended ranges

Ranges can also be open-ended. You can remove the start or the end of the range to make it open-ended.

Let's say you wanted to start counting at 3, but you didn't have a specific end in mind. You could use the range `3..` to represent this. When you use a range that's open-ended on the right side, remember that this will continue counting for as long as possible, which could be a very long time! You'll often want to use open-ended ranges with commands like `take`[10], so you can take the number of elements you want from the range.

You can also make the start of the range open. In this case, Nushell
:: warning

Watch out for displaying open-ended ranges like just entering `3..` int
::

Binary data

Binary data, like the data from an image file, is a group of raw bytes.

You can write binary as a literal using any of the `0x[...]`, `0b[...]`, or `0o[...]` forms:

```
> 0x[1F FF]  # Hexadecimal > 0b[1 1010] # Binary > 0o[377]
  # Octal
```

Incomplete bytes will be left-padded with zeros.

Structured data

Structured data builds from the simple data. For example, instead of a single integer, structured data gives us a way to represent multiple

[10] `/commands/docs/take.md`

integers in the same value. Here's a list of the currently supported structured data types: records, lists and tables.

Records

Records hold key-value pairs, which associate string keys with various data values. Record syntax is very similar to objects in JSON. However, commas are *not* required to separate values if Nushell can easily distinguish them!

```
> {name: sam rank: 10}          name   sam    rank   10
```

As these can sometimes have many fields, a record is printed up-down rath

::tip A record is identical to a single row of a table (see below). You can think of a record as essentially being a "one-row table", with each of its keys as a column (although a true one-row table is something distinct from a record).

This means that any command that operates on a table's rows *also* operates on records. For instance, insert[11], which adds data to each of a table's rows, can be used with records:

```
> {x:3 y:1} | insert z 0        x   3    y   1    z   0
```

:::

You can iterate over records by first transposing it into a table:

```
> {name: sam, rank: 10} | transpose key value
  #   key    value                      0   name    sam      1
rank     10
```

Accessing records' data is done by placing a `.` before a string, which is usually a bare string:

```
> {x:12 y:4}.x 12
```

However, if a record has a key name that can't be expressed as a bare string, or resembles an integer (see lists, below), you'll need to use more explicit string syntax, like so:

[11] /commands/docs/insert.md

```
> {"1":true " ":false}." " false
```

To make a copy of a record with new fields, you can use the spread operator[12] (...):

```
> let data = { name: alice, age: 50 }> { ...$data, hobby:
 cricket }                name    alice      age     50
     hobby   cricket
```

Lists

Lists are ordered sequences of data values. List syntax is very similar to arrays in JSON. However, commas are *not* required to separate values if Nushell can easily distinguish them!

```
> [sam fred george]            0    sam         1    fred
  2   george
```

:::tip Lists are equivalent to the individual columns of tables. You can think of a list as essentially being a "one-column table" (with no column name). Thus, any command which operates on a column *also* operates on a list. For instance, `where`[13] can be used with lists:

```
> [bell book candle] | where ($it =~ 'b')          0    bell
    1    book
```

:::

Accessing lists' data is done by placing a `.` before a bare integer:

```
> [a b c].1 b
```

To get a sub-list from a list, you can use the `range`[14] command:

```
> [a b c d e f] | range 1..3         0    b     1    c     2
  d
```

To append one or more lists together, optionally with values interspersed in between, you can use the spread operator[15] (...):

[12] /book/operators#spread-operator
[13] /commands/docs/where.md
[14] /commands/docs/range.md
[15] /book/operators#spread-operator

```
> let x = [1 2] > [...$x 3 ...(4..7 | take 2)]        0
  1    1   2    2   3    3   4    4   5
```

Tables

The table is a core data structure in Nushell. As you run commands, you'll see that many of them return tables as output. A table has both rows and columns.

We can create our own tables similarly to how we create a list. Because tables also contain columns and not just values, we pass in the name of the column values:

```
> [[column1, column2]; [Value1, Value2] [Value3, Value4]]
               #   column1   column2                  0
  Value1    Value2     1   Value3    Value4

```

You can also create a table as a list of records, JSON-style:

```
> [{name: sam, rank: 10}, {name: bob, rank: 7}]
  #   name   rank               0   sam      10    1   bob
      7
```

:::tip Internally, tables are simply **lists of records**. This means that any command which extracts or isolates a specific row of a table will produce a record. For example, `get 0`, when used on a list, extracts the first value. But when used on a table (a list of records), it extracts a record:

```
> [{x:12, y:5}, {x:3, y:6}] | get 0        x   12    y
  5
```

This is true regardless of which table syntax you use:

```
[[x,y];[12,5],[3,6]] | get 0        x   12    y   5

```

:::

Cell Paths

You can combine list and record data access syntax to navigate tables. When used on tables, these access chains are called "cell paths".

You can access individual rows by number to obtain records:

Moreover, you can also access entire columns of a table by name, to obtain lists:

```
> [{x:12 y:5} {x:4 y:7} {x:2 y:2}].x          0   12    1
   4    2    2
```

Of course, these resulting lists don't have the column names of the table. To remove columns from a table while leaving it as a table, you'll commonly use the `select`[16] command with column names:

```
> [{x:0 y:5 z:1} {x:4 y:7 z:3} {x:2 y:2 z:0}] | select
y z           #   y   z            0   5   1    1   7   3
2   2   0
```

To remove rows from a table, you'll commonly use the `select`[17] command with row numbers, as you would with a list:

```
> [{x:0 y:5 z:1} {x:4 y:7 z:3} {x:2 y:2 z:0}] | select
1 2             #   x   y   z              0   4   7   3
1   2   2   0
```

Optional cell paths By default, cell path access will fail if it can't access the requested row or column. To suppress these errors, you can add `?` to a cell path member to mark it as *optional*:

```
> [{foo: 123}, {}].foo?          0   123    1
```

When using optional cell path members, missing data is replaced with `null`.

Closures

Closures are anonymous functions that can be passed a value through parameters and *close over* (i.e. use) a variable outside their scope.

For example, in the command `each { |it| print $it }` the closure is the portion contained in curly braces, `{ |it| print $it }`. Closure parameters are specified between a pair of pipe symbols (for example, `|it|`) if necessary. You can also use a pipeline input as `$in` in most closures instead of providing an explicit parameter: `each { print $in }`

Closures itself can be bound to a named variable and passed as a parameter. To call a closure directly in your code use the `do`[18] command.

[16] /commands/docs/select.md
[17] /commands/docs/select.md
[18] /commands/docs/do.md

```
# Assign a closure to a variable let greet = { |name| print
$"Hello ($name)"} do $greet "Julian"
```

Closures are a useful way to represent code that can be executed on each row of data. It is idiomatic to use `$it` as a parameter name in `each`[19] blocks, but not required; `each { |x| print $x }` works the same way as `each { |it| print $it }`.

Blocks

Blocks don't close over variables, don't have parameters, and can't be passed as a value. However, unlike closures, blocks can access mutable variable in the parent closure. For example, mutating a variable inside the block used in an `if`[20] call is valid:

```
mut x = 1 if true {     $x += 1000 } print $x
```

Null

Finally, there is `null` which is the language's "nothing" value, similar to JSON's "null". Whenever Nushell would print the `null` value (outside of a string or data structure), it prints nothing instead. Hence, most of Nushell's file system commands (like `save`[21] or `cd`[22]) produce `null`.

You can place `null` at the end of a pipeline to replace the pipeline's output with it, and thus print nothing:

```
git checkout featurebranch | null
```

:::warning
`null` is not the same as the absence of a value! It is possible for a table to be produced that has holes in some of its rows. Attempting to access this value will not produce `null`, but instead cause an error:

```
> [{a:1 b:2} {b:1}]             #   a    b             0
 1   2    1      1             > [{a:1 b:2} {b:1}].1.a Er-
ror: nu::shell::column_not_found   × Cannot find column
    [entry #15:1:1]  1   [{a:1 b:2} {b:1}].1.a    ·
```

[19] /commands/docs/each.md
[20] /commands/docs/if.md
[21] /commands/docs/save.md
[22] /commands/docs/cd.md

```
                    ·                          cannot find column
     ·                    value originates here
```

If you would prefer this to return `null`, mark the cell path member as *optional* like `.1.a?`.

The absence of a value is (as of Nushell 0.71) printed as the emoji in interactive output. :::

Loading data

Earlier, we saw how you can use commands like `ls`[23], `ps`[24], `date`[25], and `sys`[26] to load information about your files, processes, time of date, and the system itself. Each command gives us a table of information that we can explore. There are other ways we can load in a table of data to work with.

Opening files

One of Nu's most powerful assets in working with data is the `open`[27] command. It is a multi-tool that can work with a number of different data formats. To see what this means, let's try opening a json file:

In a similar way to `ls`[28], opening a file type that Nu understands will give us back something that is more than just text (or a stream of bytes). Here we open a "package.json" file from a JavaScript project. Nu can recognize the JSON text and parse it to a table of data.

If we wanted to check the version of the project we were looking at, we can use the `get`[29] command.

```
> open editors/vscode/package.json | get version 1.0.0
```

Nu currently supports the following formats for loading data directly into tables:

- csv
- eml

[23] /commands/docs/ls.md
[24] /commands/docs/ps.md
[25] /commands/docs/date.md
[26] /commands/docs/sys.md
[27] /commands/docs/open.md
[28] /commands/docs/ls.md
[29] /commands/docs/get.md

- ics
- ini
- json
- nuon[30]
- ods
- SQLite databases[31]
- ssv
- toml
- tsv
- url
- vcf
- xlsx / xls
- xml
- yaml / yml

::: tip Did you know? Under the hood `open` will look for a `from ...` subcommand in your scope which matches the extension of your file. You can thus simply extend the set of supported file types of `open` by creating your own `from ...` subcommand. :::

But what happens if you load a text file that isn't one of these? Let's try it:

```
> open README.md
```

We're shown the contents of the file.

Below the surface, what Nu sees in these text files is one large string. Next, we'll talk about how to work with these strings to get the data we need out of them.

[30] `#nuon`
[31] `#sqlite`

NUON

Nushell Object Notation (NUON) aims to be for Nushell what JavaScript Object Notation (JSON) is for JavaScript. That is, NUON code is a valid Nushell code that describes some data structure. For example, this is a valid NUON (example from the default configuration file[32]):

```
{  menus: [    # Configuration for default nushell menus
    # Note the lack of source parameter    {       name:
 completion_menu       only_buffer_difference: false
    marker: "| "       type: {           layout: columnar
          columns: 4          col_width: 20   # Optional
value. If missing all the screen width is used to calculate
column width          col_padding: 2       }      style:
 {           text: green          selected_text: green_-
reverse          description_text: yellow      }    }
  ] }
```

You might notice it is quite similar to JSON, and you're right. **NUON is a superset of JSON!** That is, any JSON code is a valid NUON code, therefore a valid Nushell code. Compared to JSON, NUON is more "human-friendly". For example, comments are allowed and commas are not required.

One limitation of NUON currently is that it cannot represent all of the Nushell data types. Most notably, NUON does not allow to serialize blocks.

Handling Strings

An important part of working with data coming from outside Nu is that it's not always in a format that Nu understands. Often this data is given to us as a string.

Let's imagine that we're given this data file:

```
> open people.txt Octavia | Butler | Writer Bob | Ross
| Painter Antonio | Vivaldi | Composer
```

Each bit of data we want is separated by the pipe ('|') symbol, and each person is on a separate line. Nu doesn't have a pipe-delimited file format by default, so we'll have to parse this ourselves.

The first thing we want to do when bringing in the file is to work with it a line at a time:

[32] https://github.com/nushell/nushell/blob/main/crates/nu-utils/src/sample_config/default_config.nu

```
> open people.txt | lines                       0   Octavia
| Butler | Writer  1   Bob | Ross | Painter  2   Antonio
| Vivaldi | Composer
```

We can see that we're working with the lines because we're back into a list. Our next step is to see if we can split up the rows into something a little more useful. For that, we'll use the `split`[33] command. `split`[34], as the name implies, gives us a way to split a delimited string. We will use `split`[35]'s `column` subcommand to split the contents across multiple columns. We tell it what the delimiter is, and it does the rest:

```
> open people.txt | lines | split column "|"
 #   column1    column2    column3
 0   Octavia     Butler     Writer  1   Bob          Ross
       Painter  2   Antonio     Vivaldi     Composer
```

That *almost* looks correct. It looks like there's an extra space there. Let's `trim`[36] that extra space:

```
> open people.txt | lines | split column "|" | str trim
                     #   column1   column2   column3
 0   Octavia   Butler    Writer  1   Bob       Ross      Painter
 2   Antonio   Vivaldi   Composer
```

Not bad. The `split`[37] command gives us data we can use. It also goes ahead and gives us default column names:

```
> open people.txt | lines | split column "|" | str trim
| get column1          0   Octavia  1   Bob  2   Antonio
```

We can also name our columns instead of using the default names:

```
> open people.txt | lines | split column "|" first_name
last_name job | str trim                         #   first_-
name   last_name   job                         0   Octavia
     Butler      Writer  1   Bob          Ross        Painter
 2   Antonio      Vivaldi     Composer
```

[33] /commands/docs/split.md
[34] /commands/docs/split.md
[35] /commands/docs/split.md
[36] /commands/docs/str_trim.md
[37] /commands/docs/split.md

Now that our data is in a table, we can use all the commands we've used on tables before:

```
> open people.txt | lines | split column "|" first_name
last_name job | str trim | sort-by first_name
 #   first_name   last_name   job
0   Antonio      Vivaldi     Composer  1   Bob          Ross
      Painter  2   Octavia      Butler      Writer
```

There are other commands you can use to work with strings:

- `str`[38]
- `lines`[39]
- `size`[40]

There is also a set of helper commands we can call if we know the data has a structure that Nu should be able to understand. For example, let's open a Rust lock file:

```
> open Cargo.lock # This file is automatically @generated
by Cargo. # It is not intended for manual editing. [[pack-
age]] name = "adhoc_derive" version = "0.1.2"
```

The "Cargo.lock" file is actually a .toml file, but the file extension isn't .toml. That's okay, we can use the `from`[41] command using the `toml` subcommand:

The `from`[42] command can be used for each of the structured data text formats that Nu can open and understand by passing it the supported format as a subcommand.

Opening in raw mode

While it's helpful to be able to open a file and immediately work with a table of its data, this is not always what you want to do. To get to the underlying text, the `open`[43] command can take an optional `--raw` flag:

[38] /commands/docs/str.md
[39] /commands/docs/lines.md
[40] /commands/docs/size.md
[41] /commands/docs/from.md
[42] /commands/docs/from.md
[43] /commands/docs/open.md

```
> open Cargo.toml --raw [package]

           name = "nu" version = "0.1.3" authors = ["Yehuda
Katz <wycats@gmail.com>", "Jonathan Turner <jonathan.d.
turner@gmail.com>"] description = "A shell for the GitHub
era" license = "MIT"
```

SQLite

SQLite databases are automatically detected by `open`[44], no matter what their file extension is. You can open a whole database:

```
> open foo.db
```

Or `get`[45] a specific table:

```
> open foo.db | get some_table
```

Or run any SQL query you like:

```
> open foo.db | query db "select * from some_table"
```

(Note: some older versions of Nu use `into db | query` instead of `query db`)

Fetching URLs

In addition to loading files from your filesystem, you can also load URLs by using the `http get`[46] command. This will fetch the contents of the URL from the internet and return it:

Pipelines

One of the core designs of Nu is the pipeline, a design idea that traces its roots back decades to some of the original philosophy behind Unix. Just as Nu extends from the single string data type of Unix, Nu also extends the idea of the pipeline to include more than just text.

[44] /commands/docs/open.md
[45] /commands/docs/get.md
[46] /commands/docs/http.md

Basics

A pipeline is composed of three parts: the input, the filter, and the output.

```
> open "Cargo.toml" | inc package.version --minor | save
"Cargo_new.toml"
```

The first command, `open "Cargo.toml"`, is an input (sometimes also called a “source” or “producer”). This creates or loads data and feeds it into a pipeline. It’s from input that pipelines have values to work with. Commands like `ls`[47] are also inputs, as they take data from the filesystem and send it through the pipelines so that it can be used.

The second command, `inc package.version --minor`, is a filter. Filters take the data they are given and often do something with it. They may change it (as with the `inc`[48] command in our example), or they may do another operation, like logging, as the values pass through.

The last command, `save "Cargo_new.toml"`, is an output (sometimes called a “sink”). An output takes input from the pipeline and does some final operation on it. In our example, we save what comes through the pipeline to a file as the final step. Other types of output commands may take the values and view them for the user.

The `$in` variable will collect the pipeline into a value for you, allowing you to access the whole stream as a parameter:

```
> [1 2 3] | $in.1 * $in.2 6
```

Multi-line pipelines

If a pipeline is getting a bit long for one line, you can enclose it within `(` and `)` to create a subexpression:

```
(     "01/22/2021" |     parse "{month}/{day}/{year}" |
    get year )
```

Also see Subexpressions[49]

[47] /commands/docs/ls.md
[48] /commands/docs/inc.md
[49] https://www.nushell.sh/book/variables_and_subexpressions.html#subexpressions

Semicolons

Take this example:

```
> line1; line2 | line3
```

Here, semicolons are used in conjunction with pipelines. When a semicolon is used, no output data is produced to be piped. As such, the `$in` variable will not work when used immediately after the semicolon.

- As there is a semicolon after `line1`, the command will run to completion and get displayed on the screen.
- `line2` | `line3` is a normal pipeline. It runs, and its contents are displayed after `line1`'s contents.

Working with external commands

Nu commands communicate with each other using the Nu data types (see types of data), but what about commands outside of Nu? Let's look at some examples of working with external commands:

`internal_command | external_command`

Data will flow from the internal_command to the external_command. This data will get converted to a string, so that they can be sent to the `stdin` of the external_command.

`external_command | internal_command`

Data coming from an external command into Nu will come in as bytes that Nushell will try to automatically convert to UTF-8 text. If successful, a stream of text data will be sent to internal_command. If unsuccessful, a stream of binary data will be sent to internal command. Commands like `lines`[50] help make it easier to bring in data from external commands, as it gives discrete lines of data to work with.

`external_command_1 | external_command_2`

Nu works with data piped between two external commands in the same way as other shells, like Bash would. The `stdout` of external_command_1 is connected to the `stdin` of external_command_2. This lets data flow naturally between the two commands.

Behind the scenes

You may have wondered how we see a table if `ls`[51] is an input and not an output. Nu adds this output for us automatically using another

[50] `/commands/docs/lines.md`
[51] `/commands/docs/ls.md`

command called table[52]. The table[53] command is appended to any pipeline that doesn't have an output. This allows us to see the result.

In effect, the command:

```
> ls
```

And the pipeline:

```
> ls | table
```

Are one and the same.

> ***Note*** *the sentence* are one and the same *above only applies for the graphical output in the shell, it does not mean the two data structures are them same*
>
> ```
> > (ls) == (ls | table) false
> ```
>
> `ls | table` *is not even structured data!*

Output result to external commands

Sometimes you want to output Nushell structured data to an external command for further processing. However, Nushell's default formatting options for structured data may not be what you want. For example, you want to find a file named "tutor" under "/usr/share/vim/runtime" and check its ownership

```
> ls /usr/share/nvim/runtime/
   #                  name                      type    size
      modified
   0   /usr/share/nvim/runtime/autoload        dir     4.
1 KB   2 days ago     ..........  ..........  .........
.   31   /usr/share/nvim/runtime/tools           dir
 4.1 KB   2 days ago      32   /usr/share/nvim/runtime/
tutor            dir     4.1 KB   2 days ago
   #                  name                      type    size
      modified
```

You decided to use `grep` and pipe[54] the result to external `^ls`

52/commands/docs/table.md
53/commands/docs/table.md
54https://www.nushell.sh/book/pipelines.html

```
> ls /usr/share/nvim/runtime/ | get name | ^grep tutor
| ^ls -la $in ls: cannot access ''$'\342\224\202'' 32 '$'\342\224\202''
/usr/share/nvim/runtime/tutor        '$'\342\224\202\n':
 No such file or directory
```

What's wrong? Nushell renders lists and tables (by adding a border with characters like , , ,) before piping them as text to external commands. If that's not the behavior you want, you must explicitly convert the data to a string before piping it to an external. For example, you can do so with `to text`[55]:

```
> ls /usr/share/nvim/runtime/ | get name | to text | ^grep
tutor | tr -d '\n' | ^ls -la $in total 24 drwxr-xr-x@
5 pengs  admin  160 14 Nov 13:12 . drwxr-xr-x@  4 pengs
 admin  128 14 Nov 13:42 en -rw-r--r--@  1 pengs  admin
 5514 14 Nov 13:42 tutor.tutor -rw-r--r--@  1 pengs  admin
 1191 14 Nov 13:42 tutor.tutor.json
```

(Actually, for this simple usage you can just use `find`[56])

```
> ls /usr/share/nvim/runtime/ | get name | find tutor |
^ls -al $in
```

Command Output in Nushell

Unlike external commands, Nushell commands are akin to functions. Most Nushell commands do not print anything to `stdout` and instead just return data.

```
> do { ls; ls; ls; "What?!" }
```

This means that the above code will not display the files under the current directory three times. In fact, running this in the shell will only display `"What?!"` because that is the value returned by the `do` command in this example. However, using the system `^ls` command instead of `ls` would indeed print the directory thrice because `^ls` does print its result once it runs.

Knowing when data is displayed is important when using configuration variables that affect the display output of commands such as `table`.

[55] /commands/docs/to_text.md
[56] /commands/docs/find.md

```
> do { $env.config.table.mode = none; ls }
```

For instance, the above example sets the `$env.config.table.mode` configuration variable to `none`, which causes the `table` command to render data without additional borders. However, as it was shown earlier, the command is effectively equivalent to

```
> do { $env.config.table.mode = none; ls } | table
```

Because Nushell `$env` variables are scoped[57], this means that the `table` command in the example is not affected by the environment modification inside the `do` block and the data will not be shown with the applied configuration.

When displaying data early is desired, it is possible to explicitly apply `| table` inside the scope, or use the `print` command.

```
> do { $env.config.table.mode = none; ls | table }> do
{ $env.config.table.mode = none; print (ls) }
```

Working with strings

Strings in Nushell help to hold text data for later use. This can include file names, file paths, names of columns, and much more. Strings are so common that Nushell offers a couple ways to work with them, letting you pick what best matches your needs.

[57] https://www.nushell.sh/book/environment.html#scoping

String formats at a glance

Format of string	Example	Escapes	Notes
Single-quoted string	`'[^\n]+'`	None	Cannot contain any `'`
Backtick string	<code>‘[^\n]+‘</code>	None	Cannot contain any backticks ‘
Double-quoted string	`"The\nEnd"`	C-style backslash escapes	All backslashes must be escaped
Bare string	`ozymandias`	None	Can only contain “word” characters; Cannot be used in command position
Single-quoted interpolation	`$'Captain ($name)'`	None	Cannot contain any `'` or unmatched `()`
Double-quoted interpolation	`$"Captain ($name)"`	C-style backslash escapes	All backslashes and `()` must be escaped

Single-quoted strings

The simplest string in Nushell is the single-quoted string. This string uses the `'` character to surround some text. Here’s the text for hello world as a single-quoted string:

```
> 'hello world' hello world > 'The end' The end
```

Single-quoted strings don’t do anything to the text they’re given, making them ideal for holding a wide range of text data.

Backtick-quoted strings

Single-quoted strings, due to not supporting any escapes, cannot contain any single-quote characters themselves. As an alternative, backtick strings using the <code>‘</code> character also exist:

```
> `no man's land` no man's land > `no man's land` no man's
land
```

Of course, backtick strings cannot contain any backticks themselves. Otherwise, they are identical to single-quoted strings.

Double-quoted Strings

For more complex strings, Nushell also offers double-quoted strings. These strings use the `"` character to surround text. They also support the ability escape characters inside the text using the `\` character.

For example, we could write the text hello followed by a new line and then world, using escape characters and a double-quoted string:

```
> "hello\nworld" hello world
```

Escape characters let you quickly add in a character that would otherwise be hard to type.

Nushell currently supports the following escape characters:

- `\"` - double-quote character
- `\'` - single-quote character
- `\\` - backslash
- `\/` - forward slash
- `\b` - backspace
- `\f` - formfeed
- `\r` - carriage return
- `\n` - newline (line feed)
- `\t` - tab
- `\u{X...}` - a single unicode character, where X... is 1-6 hex digits (0-9, A-F)

Bare strings

Like other shell languages (but unlike most other programming languages) strings consisting of a single ‘word’ can also be written without any quotes:

```
> print hello hello > [hello] | describe list<string>
```

But be careful - if you use a bare word plainly on the command line (that is, not inside a data structure or used as a command parameter) or inside round brackets `( )`, it will be interpreted as an external command:

```
> hello Error: nu::shell::external_command  × External
command failed     [entry #5:1:1]  1  hello   ·
 ·     executable was not found        help: program
not found
```

Also, many bare words have special meaning in nu, and so will not be interpreted as a string:

```
> true | describe bool > [true] | describe list<bool> >
[trueX] | describe list<string> > trueX | describe Error:
 nu::shell::external_command  × External command failed
    [entry #5:1:1]  1  trueX | describe   ·        ·
   executable was not found        help: program not
found
```

So, while bare strings are useful for informal command line usage, when programming more formally in nu, you should generally use quotes.

Strings as external commands

You can place the ^ sigil in front of any string (including a variable) to have Nushell execute the string as if it was an external command:

```
^'C:\Program Files\exiftool.exe' > let foo = 'C:\Program
Files\exiftool.exe' > ^$foo
```

You can also use the `run-external`[58] command for this purpose, which provides additional flags and options.

Appending and Prepending to strings

There are various ways to pre, or append strings. If you want to add something to the beginning of each string closures are a good option:

[58] /commands/docs/run-external.md

```
['foo', 'bar'] | each {|s| '~/' ++ $s} # ~/foo, ~/bar ['foo',
'bar'] | each {|s| '~/' + $s} # ~/foo, ~/bar
```

You can also use a regex to replace the beginning or end of a string:

```
['foo', 'bar'] | str replace -r '^' '~/'# ~/foo, ~/bar ['foo',
'bar'] | str replace -r '$' '~/'# foo~/, bar~/
```

If you want to get one string out of the end then `str join` is your friend:

```
"hello" | append "world!" | str join " " # hello world!
```

You can also use reduce:

```
1..10 | reduce -f "" {|it, acc| $acc + ($it | into string)
 + " + "} # 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 +
```

Though in the cases of strings, especially if you don't have to operate on the strings, it's usually easier and more correct (notice the extra + at the end in the example above) to use `str join`.

Finally you could also use string interpolation, but that is complex enough that it is covered in its own subsection below.

String interpolation

More complex string use cases also need a new form of string: string interpolation. This is a way of building text from both raw text and the result of running expressions. String interpolation combines the results together, giving you a new string.

String interpolation uses `$" "` and `$' '` as ways to wrap interpolated text.

For example, let's say we have a variable called `$name` and we want to greet the name of the person contained in this variable:

```
> let name = "Alice" > $"greetings, ($name)" greetings,
Alice
```

By wrapping expressions in `()`, we can run them to completion and use the results to help build the string.

String interpolation has both a single-quoted, `$' '`, and a double-quoted, `$" "`, form. These correspond to the single-quoted and double-quoted strings: single-quoted string interpolation doesn't support escape characters while double-quoted string interpolation does.

As of version 0.61, interpolated strings support escaping parentheses, so that the `(` and `)` characters may be used in a string without Nushell trying to evaluate what appears between them:

```
> $"2 + 2 is (2 + 2) \(you guessed it!)" 2 + 2 is 4 (you
guessed it!)
```

Interpolated strings can be evaluated at parse time, but if they include values whose formatting depends on your configuration and your `config.nu` hasn't been loaded yet, they will use the default configuration. So if you have something like this in your `config.nu`, `x` will be `"2.0 KB"` even if your config says to use `MB` for all file sizes (datetimes will similarly use the default config).

```
> const x = $"(2kb)"
```

Splitting strings

The `split row`[59] command creates a list from a string based on a delimiter.

```
> "red,green,blue" | split row ","          0   red
1   green    2   blue
```

The `split column`[60] command will create a table from a string based on a delimiter. This applies generic column names to the table.

```
> "red,green,blue" | split column ","
  #   column1   column2   column3
0   red         green     blue
```

Finally, the `split chars`[61] command will split a string into a list of characters.

```
> 'aeiou' | split chars       0   a    1   e    2   i
3   o    4   u
```

[59] /commands/docs/split_row.md
[60] /commands/docs/split_column.md
[61] /commands/docs/split_chars.md

The `str`[62] command

Many string functions are subcommands of the `str`[63] command. You can get a full list using `help str`.

For example, you can look if a string contains a particular substring using `str contains`[64]:

```
> "hello world" | str contains "o wo" true
```

(You might also prefer, for brevity, the `=~` operator (described below).)

Trimming strings

You can trim the sides of a string with the `str trim`[65] command. By default, the `str trim`[66] commands trims whitespace from both sides of the string. For example:

```
> '       My   string   ' | str trim My   string
```

You can specify on which side the trimming occurs with the `--right` and `--left` options. (`-r` and `-l` being the short-form options respectively)

To trim a specific character, use `--char <Character>` or `-c <Character>` to specify the character to trim.

Here's an example of all the options in action:

```
> '=== Nu shell ===' | str trim -r -c '=' === Nu shell
```

Substrings

Substrings are slices of a string. They have a startpoint and an endpoint. Here's an example of using a substring:

```
> 'Hello World!' | str index-of 'o' 4 > 'Hello World!'
| str index-of 'r' 8 > 'Hello World!' | str substring 4.
.8 o Wo
```

[62] /commands/docs/str.md
[63] /commands/docs/str.md
[64] /commands/docs/str_contains.md
[65] /commands/docs/str_trim.md
[66] /commands/docs/str_trim.md

String padding

With the `fill`[67] command you can add padding to a string. Padding adds characters to string until it's a certain length. For example:

```
> '1234' | fill -a right -c '0' -w 10 0000001234 > '1234'
| fill -a left -c '0' -w 10 | str length 10
```

Reversing strings

This can be done easily with the `str reverse`[68] command.

```
> 'Nushell' | str reverse llehsuN > ['Nushell' 'is' 'cool']
| str reverse           0   llehsuN    1   si          2
  looc
```

String parsing

With the `parse`[69] command you can parse a string into columns. For example:

```
> 'Nushell 0.80' | parse '{shell} {version}'
  #    shell    version                      0   Nushell   0.
80                      > 'where all data is structured!'
| parse --regex '(?P<subject>\w*\s?\w+) is (?P<adjective>\w+)
'                    #   subject    adjective
  0   all data   structured
```

If a string is known to contain comma-separated, tab-separated or multi-space-separated data, you can use `from csv`[70], `from tsv`[71] or `from ssv`[72]:

```
> "acronym,long\nAPL,A Programming Language" | from csv
                            #   acronym           long
                                   0   APL       A Programming
Language                         > "name  duration\nonestop.
mid   4:06" | from ssv                          #      name
   duration                        0   onestop.mid   4:06
```

[67] /commands/docs/fill.md
[68] /commands/docs/str_reverse.md
[69] /commands/docs/parse.md
[70] /commands/docs/from_csv.md
[71] /commands/docs/from_tsv.md
[72] /commands/docs/from_ssv.md

```
                      > "rank\tsuit\nJack\tSpades\nAce\tClubs"
| from tsv                 #   rank    suit
0   Jack   Spades    1   Ace    Clubs
```

String comparison

In addition to the standard `==` and `!=` operators, a few operators exist for specifically comparing strings to one another.

Those familiar with Bash and Perl will recognise the regex comparison operators:

```
> 'APL' =~ '^\w{0,3}$' true > 'FORTRAN' !~ '^\w{0,3}$' true
```

Two other operators exist for simpler comparisons:

```
> 'JavaScript' starts-with 'Java' true > 'OCaml' ends-with
'Caml' true
```

Converting strings

There are multiple ways to convert strings to and from other types.

To string

1. Using `into string`[73]. e.g. `123 | into string`
2. Using string interpolation. e.g. `$'(123)'`

From string

1. Using `into <type>`[74]. e.g. `'123' | into int`

Coloring strings

You can color strings with the `ansi`[75] command. For example:

```
> $'(ansi purple_bold)This text is a bold purple!(ansi
reset)'
```

`ansi purple_bold` makes the text a bold purple `ansi reset` resets the coloring to the default. (Tip: You should always end colored strings with `ansi reset`)

[73] /commands/docs/into_string.md
[74] /commands/docs/into.md
[75] /commands/docs/ansi.md

Working with lists

Creating lists

A list is an ordered collection of values. You can create a `list` with square brackets, surrounded values separated by spaces and/or commas (for readability). For example, `[foo bar baz]` or `[foo, bar, baz]`.

Updating lists

You can `update`[76] and `insert`[77] values into lists as they flow through the pipeline, for example let's insert the value `10` into the middle of a list:

```
> [1, 2, 3, 4] | insert 2 10 # [1, 2, 10, 3, 4]
```

We can also use `update`[78] to replace the 2nd element with the value `10`.

```
> [1, 2, 3, 4] | update 1 10 # [1, 10, 3, 4]
```

Removing or adding items from list

In addition to `insert`[79] and `update`[80], we also have `prepend`[81] and `append`[82]. These let you insert to the beginning of a list or at the end of the list, respectively.

For example:

```
let colors = [yellow green] let colors = ($colors | prepend
red) let colors = ($colors | append purple) let colors
= ($colors ++ "blue") let colors = ("black" ++ $colors)
$colors # [black red yellow green purple blue]
```

In case you want to remove items from list, there are many ways. `skip`[83] allows you skip first rows from input, while `drop`[84] allows you to skip specific numbered rows from end of list.

[76] /commands/docs/update.md
[77] /commands/docs/insert.md
[78] /commands/docs/update.md
[79] /commands/docs/insert.md
[80] /commands/docs/update.md
[81] /commands/docs/prepend.md
[82] /commands/docs/append.md
[83] /commands/docs/skip.md
[84] /commands/docs/drop.md

```
let colors = [red yellow green purple] let colors = ($colors
| skip 1) let colors = ($colors | drop 2) $colors # [yellow]
```

We also have `last`[85] and `first`[86] which allow you to `take`[87] from the end or beginning of the list, respectively.

```
let colors = [red yellow green purple black magenta] let
colors = ($colors | last 3) $colors # [purple black magenta]
```

And from the beginning of a list,

```
let colors = [yellow green purple] let colors = ($colors
| first 2) $colors # [yellow green]
```

Iterating over lists

To iterate over the items in a list, use the `each`[88] command with a block of Nu code that specifies what to do to each item. The block parameter (e.g. `|it|` in `{ |it| print $it }`) is the current list item, but the `enumerate`[89] filter can be used to provide `index` and `item` values if needed. For example:

```
let names = [Mark Tami Amanda Jeremy] $names | each { |it|
$"Hello, ($it)!" } # Outputs "Hello, Mark!" and three more
similar lines. $names | enumerate | each { |it| $"($it.
index + 1) - ($it.item)" } # Outputs "1 - Mark", "2 - Tami",
etc.
```

The `where`[90] command can be used to create a subset of a list, effectively filtering the list based on a condition.

The following example gets all the colors whose names end in "e".

```
let colors = [red orange yellow green blue purple] $colors
| where ($it | str ends-with 'e') # The block passed to
```

[85] /commands/docs/last.md
[86] /commands/docs/first.md
[87] /commands/docs/take.md
[88] /commands/docs/each.md
[89] /commands/docs/enumerate.md
[90] /commands/docs/where.md

```
`where` must evaluate to a boolean. # This outputs the
list [orange blue purple].
```

In this example, we keep only values higher than 7.

```
let scores = [7 10 8 6 7] $scores | where $it > 7 # [10
8]
```

The `reduce`[91] command computes a single value from a list. It uses a block which takes 2 parameters: the current item (conventionally named `it`) and an accumulator (conventionally named `acc`). To specify an initial value for the accumulator, use the `--fold` (`-f`) flag. To change `it` to have `index` and `item` values, use the `enumerate`[92] filter. For example:

```
let scores = [3 8 4] $"total = ($scores | reduce { |it,
acc| $acc + $it })" # total = 15 $"total = ($scores | math
sum)" # easier approach, same result $"product = ($scores
| reduce --fold 1 { |it, acc| $acc * $it })" # product
= 96 $scores | enumerate | reduce --fold 0 { |it, acc|
$acc + $it.index * $it.item } # 0*3 + 1*8 + 2*4 = 16
```

Accessing the list

To access a list item at a given index, use the `$name.index` form where `$name` is a variable that holds a list.

For example, the second element in the list below can be accessed with `$names.1`.

```
let names = [Mark Tami Amanda Jeremy] $names.1 # gives
Tami
```

If the index is in some variable `$index` we can use the `get` command to extract the item from the list.

```
let names = [Mark Tami Amanda Jeremy] let index = 1 $names
| get $index # gives Tami
```

The `length`[93] command returns the number of items in a list. For example, `[red green blue] | length` outputs 3.

The `is-empty`[94] command determines whether a string, list, or table is empty. It can be used with lists as follows:

[91] /commands/docs/reduce.md
[92] /commands/docs/enumerate.md
[93] /commands/docs/length.md
[94] /commands/docs/is-empty.md

```
let colors = [red green blue] $colors | is-empty # false
let colors = [] $colors | is-empty # true
```

The `in` and `not-in` operators are used to test whether a value is in a list. For example:

```
let colors = [red green blue] 'blue' in $colors # true 'yellow'
in $colors # false 'gold' not-in $colors # true
```

The `any`[95] command determines if any item in a list matches a given condition. For example:

```
let colors = [red green blue] # Do any color names end
with "e"?  $colors | any {|it| $it | str ends-with "e"
} # true # Is the length of any color name less than 3?
$colors | any {|it| ($it | str length) < 3 } # false let
scores = [3 8 4] # Are any scores greater than 7? $scores
| any {|it| $it > 7 } # true # Are any scores odd? $scores
| any {|it| $it mod 2 == 1 } # true
```

The `all`[96] command determines if every item in a list matches a given condition. For example:

```
let colors = [red green blue] # Do all color names end
with "e"?  $colors | all {|it| $it | str ends-with "e"
} # false # Is the length of all color names greater than
or equal to 3?  $colors | all {|it| ($it | str length)
>= 3 } # true let scores = [3 8 4] # Are all scores greater
than 7? $scores | all {|it| $it > 7 } # false # Are all
scores even? $scores | all {|it| $it mod 2 == 0 } # false
```

Converting the list

The `flatten`[97] command creates a new list from an existing list by adding items in nested lists to the top-level list. This can be called multiple times to flatten lists nested at any depth. For example:

95 /commands/docs/any.md
96 /commands/docs/all.md
97 /commands/docs/flatten.md

```
[1 [2 3] 4 [5 6]] | flatten # [1 2 3 4 5 6] [[1 2] [3 [4
5 [6 7 8]]]] | flatten | flatten | flatten # [1 2 3 4 5
6 7 8]
```

The `wrap`[98] command converts a list to a table. Each list value will be converted to a separate row with a single column:

```
let zones = [UTC CET Europe/Moscow Asia/Yekaterinburg] #
Show world clock for selected time zones $zones | wrap
'Zone' | upsert Time {|it| (date now | date to-timezone
$it.Zone | format date '%Y.%m.%d %H:%M')}
```

Working with tables

One of the common ways of seeing data in Nu is through a table. Nu comes with a number of commands for working with tables to make it convenient to find what you're looking for, and for narrowing down the data to just what you need.

To start off, let's get a table that we can use:

```
> ls                                # name              type
 size      modified                            0  files.
rs        File    4.6 KB   5 days ago  1   lib.rs
 File      330 B   5 days ago  2   lite_parse.rs   File
 6.3 KB   5 days ago  3   parse.rs        File   49.8 KB
 1 day ago   4   path.rs         File    2.1 KB   5 days
ago  5   shapes.rs       File    4.7 KB   5 days ago  6
signature.rs    File    1.2 KB   5 days ago
```

::: tip Changing how tables are displayed Nu will try to expands all table's structure by default. You can change this behavior by changing the `display_output` hook. See hooks[99] for more information. :::

Sorting the data

We can sort a table by calling the `sort-by`[100] command and telling it which columns we want to use in the sort. Let's say we wanted to sort our table by the size of the file:

[98] /commands/docs/wrap.md
[99] /book/hooks.md#changing-how-output-is-displayed
[100] /commands/docs/sort-by.md

```
> ls | sort-by size                            #   name
           type   size     modified
 0   lib.rs         File     330 B   5 days ago  1   signature.
rs    File    1.2 KB   5 days ago  2   path.rs          File
   2.1 KB   5 days ago  3   files.rs        File    4.6 KB
 5 days ago  4   shapes.rs       File    4.7 KB   5 days
ago  5   lite_parse.rs   File    6.3 KB   5 days ago  6
parse.rs        File   49.8 KB   1 day ago
```

We can sort a table by any column that can be compared. For example, we could also have sorted the above using the “name”, “accessed”, or “modified” columns.

Selecting the data you want

We can select data from a table by choosing to select specific columns or specific rows. Let’s `select`[101] a few columns from our table to use:

```
> ls | select name size                        #   name
     size                     0   files.rs         4.6 KB
1   lib.rs               330 B  2   lite_parse.rs    6.3 KB
3   parse.rs           49.8 KB  4   path.rs          2.1 KB
5   shapes.rs           4.7 KB  6   signature.rs     1.2 KB
```

This helps to create a table that’s more focused on what we need. Next, let’s say we want to only look at the 5 smallest files in this directory:

```
> ls | sort-by size | first 5
 #   name           type   size     modified
 0   lib.rs         File    330 B   5 days ago  1   signature.
rs   File   1.2 KB   5 days ago  2   path.rs          File
  2.1 KB   5 days ago  3   files.rs       File   4.6 KB
5 days ago  4   shapes.rs      File   4.7 KB   5 days ago
```

You’ll notice we first sort the table by size to get to the smallest file, and then we use the `first 5` to return the first 5 rows of the table.

You can also `skip`[102] rows that you don’t want. Let’s skip the first two of the 5 rows we returned above:

[101] `/commands/docs/select.md`
[102] `/commands/docs/skip.md`

```
> ls | sort-by size | first 5 | skip 2
 #   name         type   size     modified
 0   path.rs      File   2.1 KB   5 days ago 1   files.rs
   File   4.6 KB   5 days ago 2   shapes.rs   File   4.7
KB   5 days ago
```

We've narrowed it to three rows we care about.

Let's look at a few other commands for selecting data. You may have wondered why the rows of the table are numbers. This acts as a handy way to get to a single row. Let's sort our table by the file name and then pick one of the rows with the `select`[103] command using its row number:

```
> ls | sort-by name                                 #   name
          type   size     modified
 0   files.rs        File    4.6 KB   5 days ago 1   lib.
rs           File     330 B   5 days ago 2   lite_parse.rs
  File    6.3 KB   5 days ago 3   parse.rs        File
49.8 KB   1 day ago 4   path.rs         File    2.1 KB
5 days ago 5   shapes.rs       File    4.7 KB   5 days ago
 6   signature.rs    File    1.2 KB   5 days ago
> ls | sort-by name | select 5
 #   name            type   size      modified
 0   shapes.rs       File    4.7 KB   5 days ago
```

Getting data out of a table

So far, we've worked with tables by trimming the table down to only what we need. Sometimes we may want to go a step further and only look at the values in the cells themselves rather than taking a whole column. Let's say, for example, we wanted to only get a list of the names of the files. For this, we use the `get`[104] command:

```
> ls | get name            0   files.rs 1   lib.rs 2
lite_parse.rs  3   parse.rs  4   path.rs  5   shapes.rs
6   signature.rs
```

We now have the values for each of the filenames.

[103] /commands/docs/select.md
[104] /commands/docs/get.md

This might look like the select[105] command we saw earlier, so let's put that here as well to compare the two:

```
> ls | select name                    #   name              0
files.rs  1   lib.rs  2   lite_parse.rs  3   parse.rs  4
  path.rs  5   shapes.rs  6   signature.rs
```

These look very similar! Let's see if we can spell out the difference between these two commands to make it clear:

- select[106] - creates a new table which includes only the columns specified
- get[107] - returns the values inside the column specified as a list

The one way to tell these apart looking at the table is that the column names are missing, which lets us know that this is going to be a list of values we can work with.

The get[108] command can go one step further and take a path to data deeper in the table. This simplifies working with more complex data, like the structures you might find in a .json file.

Changing data in a table

In addition to selecting data from a table, we can also update what the table has. We may want to combine tables, add new columns, or edit the contents of a cell. In Nu, rather than editing in place, each of the commands in the section will return a new table in the pipeline.

Concatenating Tables

We can concatenate tables using append[109]:

```
let first = [[a b]; [1 2]] let second = [[a b]; [3 4]] $first
| append $second          #   a   b         0   1   2  1   3
  4
```

If the column names are not identical then additionally columns and values will be created as necessary:

[105] /commands/docs/select.md
[106] /commands/docs/select.md
[107] /commands/docs/get.md
[108] /commands/docs/get.md
[109] /commands/docs/append.md

```
let first = [[a b]; [1 2]] let second = [[a b]; [3 4]] let
third = [[a c]; [3 4]] $first | append $second | append
$third           #   a   b    c            0   1    2
1   3    4     2   3       4
```

You can also use the `++` operator as an inline replacement for `append`:

```
$first ++ $second ++ $third           #   a   b    c
 0   1    2     1   3    4     2   3       4
```

Merging Tables

We can use the `merge`[110] command to merge two (or more) tables together

```
let first = [[a b]; [1 2]] let second = [[c d]; [3 4]] $first
| merge $second              #   a   b   c   d
0   1   2   3   4
```

Let's add a third table:

```
> let third = [[e f]; [5 6]]
```

We could join all three tables together like this:

```
> $first | merge $second  | merge $third
 #   a   b   c   d   e   f                  0   1   2   3
4   5   6
```

Or we could use the `reduce`[111] command to dynamically merge all tables:

```
> [$first $second $third] | reduce {|it, acc| $acc | merge
$it }                  #   a   b   c   d   e   f
 0   1   2   3   4   5   6
```

Adding a new column

We can use the `insert`[112] command to add a new column to the table. Let's look at an example:

[110] /commands/docs/merge.md
[111] /commands/docs/reduce.md
[112] /commands/docs/insert.md

```
> open rustfmt.toml          edition   2018
```

Let's add a column called "next_edition" with the value 2021:

```
> open rustfmt.toml | insert next_edition 2021
 edition       2018  next_edition   2021
```

This visual may be slightly confusing, because it looks like what we've just done is add a row. In this case, remember: rows have numbers, columns have names. If it still is confusing, note that appending one more row will make the table render as expected:

```
> open rustfmt.toml | insert next_edition 2021 | append
{edition: 2021 next_edition: 2024}                  #
edition   next_edition                 0      2018
     2021  1      2021           2024
```

Notice that if we open the original file, the contents have stayed the same:

```
> open rustfmt.toml          edition   2018
```

Changes in Nu are functional changes, meaning that they work on values themselves rather than trying to cause a permanent change. This lets us do many different types of work in our pipeline until we're ready to write out the result with any changes we'd like if we choose to. Here we could write out the result using the `save`[113] command:

```
> open rustfmt.toml | insert next_edition 2021 | save rustfmt2.
toml > open rustfmt2.toml             edition       2018
 next_edition   2021
```

Updating a column

In a similar way to the `insert`[114] command, we can also use the `update`[115] command to change the contents of a column to a new value. To see it in action let's open the same file:

```
> open rustfmt.toml          edition   2018
```

And now, let's update the edition to point at the next edition we hope to support:

[113] /commands/docs/save.md
[114] /commands/docs/insert.md
[115] /commands/docs/update.md

```
> open rustfmt.toml | update edition 2021           edition
  2021
```

You can also use the upsert[116] command to insert or update depending on whether the column already exists.

Moving columns

You can use move[117] to move columns in the table. For example, if we wanted to move the "name" column from ls[118] after the "size" column, we could do:

```
> ls | move name --after size
  #    type    size          name          modified
                                       0   dir      256 B
Applications       3 days ago       1   dir      256 B
  Data              2 weeks ago      2   dir      448 B
  Desktop           2 hours ago      3   dir      192 B
  Disks             a week ago       4   dir      416 B
  Documents         4 days ago     ...
```

Renaming columns

You can also rename[119] columns in a table by passing it through the rename command. If we wanted to run ls[120] and rename the columns, we can use this example:

```
> ls | rename filename filetype filesize date
  #        filename        filetype   filesize       date
                                                0   Applications
      dir             256 B   3 days ago        1   Data
            dir             256 B   2 weeks ago      2   Desktop
           dir             448 B   2 hours ago      3   Disks
             dir             192 B   a week ago       4
Documents            dir             416 B   4 days ago    .
..
```

[116]/commands/docs/upsert.md
[117]/commands/docs/move.md
[118]/commands/docs/ls.md
[119]/commands/docs/rename.md
[120]/commands/docs/ls.md

Rejecting/Deleting columns

You can also reject[121] columns in a table by passing it through the reject command. If we wanted to run ls[122] and delete the columns, we can use this example:

```
> ls -l / |reject readonly num_links inode created accessed
modified                                                    #
name     type     target      mode      uid    group    size
                                             0   /bin
symlink   usr/bin   rwxrwxrwx   root   root       7 B
 1   /boot    dir                rwxr-xr-x   root   root
   1.0 KB     2   /dev     dir                rwxr-xr-x
  root   root    4.1 KB     3   /etc     dir
   rwxr-xr-x   root   root    3.6 KB     4   /home    dir
             rwxr-xr-x   root   root      12 B     5
/lib     symlink   usr/lib   rwxrwxrwx   root   root
 7 B     6   /lib64   symlink   usr/lib   rwxrwxrwx   root
  root       7 B     7   /mnt     dir                rwxr-
xr-x   root   root       0 B   ...
```

121/commands/docs/reject.md
122/commands/docs/ls.md

Chapter 4

Programming in Nu

This chapter goes into more detail of Nushell as a programming language. Each major language feature has its own section.

Just like most programming languages allow you to define functions, Nushell uses custom commands for this purpose.

From other shells you might be used to aliases. Nushell's aliases work in a similar way and are a part of the programming language, not just a shell feature.

Common operations can, such as addition or regex search, be done with operators. Not all operations are supported for all data types and Nushell will make sure to let you know.

You can store intermediate results to variables and immediately evaluate subroutines with subexpressions.

The last three sections are aimed at organizing your code:

Scripts are the simplest form of code organization: You just put the code into a file and source it. However, you can also run scripts as standalone programs with command line signatures using the "special" `main` command.

With modules[1], just like in many other programming languages, it is possible to compose your code from smaller pieces. Modules let you define a public interface vs. private commands and you can import custom commands, aliases, and environment variables from them.

Overlays build on top of modules. By defining an overlay, you bring in module's definitions into its own swappable "layer" that gets applied on top of other overlays. This enables features like activating virtual

[1] `modules.md`

environments or overriding sets of default commands with custom variants.

The help message of some built-in commands shows a signature. You can take a look at it to get general rules how the command can be used.

The standard library also has a testing framework if you want to prove your reusable code works perfectly.

Custom commands

Nu's ability to compose long pipelines allows you a lot of control over your data and system, but it comes at the price of a lot of typing. Ideally, you'd be able to save your well-crafted pipelines to use again and again.

This is where custom commands come in.

An example definition of a custom command:

```
def greet [name] {   ['hello' $name] }
```

::: tip The value produced by the last line of a command becomes the command's returned value. In this case, a list containing the string `'hello'` and `$name` is returned. To prevent this, you can place `null` (or the `ignore`[2] command) at the end of the pipeline, like so: `['hello' $name] | null`. Also note that most file system commands, such as `save`[3] or `cd`[4], always output `null`. :::

In this definition, we define the `greet` command, which takes a single parameter `name`. Following this parameter is the block that represents what will happen when the custom command runs. When called, the custom command will set the value passed for `name` as the `$name` variable, which will be available to the block.

To run the above, we can call it like we would call built-in commands:

```
> greet "world"
```

As we do, we also get output just as we would with built-in commands:

[2] `/commands/docs/ignore.md`
[3] `/commands/docs/save.md`
[4] `/commands/docs/cd.md`

```
        0   hello  1   world
```

::: tip If you want to generate a single string, you can use the string interpolation syntax to embed $name in it:

```
def greet [name] {   $"hello ($name)" } greet nushell
```

returns `hello nushell` :::

Command names

In Nushell, a command name is a string of characters. Here are some examples of valid command names: `greet`, `get-size`, `mycommand123`, `my command`, and .

Note: It's common practice in Nushell to separate the words of the command with `-` *for better readability.* For example `get-size` instead of `getsize` or `get_size`.

Sub-commands

You can also define subcommands to commands using a space. For example, if we wanted to add a new subcommand to `str`[5], we can create it by naming our subcommand to start with "str ". For example:

```
def "str mycommand" [] {   "hello" }
```

Now we can call our custom command as if it were a built-in subcommand of `str`[6]:

```
> str mycommand
```

Of course, commands with spaces in their names are defined in the same way:

```
def "custom command" [] {   "this is a custom command with
a space in the name!" }
```

Parameter types

When defining custom commands, you can name and optionally set the type for each parameter. For example, you can write the above as:

[5] `/commands/docs/str.md`
[6] `/commands/docs/str.md`

```
def greet [name: string] {   $"hello ($name)" }
```

The types of parameters are optional. Nushell supports leaving them off and treating the parameter as `any` if so. If you annotated a type on a parameter, Nushell will check this type when you call the function.

For example, let's say you wanted to take in an `int` instead:

```
def greet [name: int] {   $"hello ($name)" } greet world
```

If we try to run the above, Nushell will tell us that the types don't match:

```
error: Type Error     shell:6:7    5   greet world
     ^^^^^ Expected int
```

This can help you guide users of your definitions to call them with only the supported types.

The currently accepted types are (as of version 0.86.0):

- `any`
- `binary`
- `bool`
- `cell-path`
- `closure`
- `datetime`
- `directory`
- `duration`
- `error`
- `filesize`
- `float`
- `glob`
- `int`
- `list`

- `nothing`
- `number`
- `path`
- `range`
- `record`
- `string`
- `table`

Parameters with a default value

To make a parameter optional and directly provide a default value for it you can provide a default value in the command definition.

```
def greet [name = "nushell"] {   $"hello ($name)" }
```

You can call this command either without the parameter or with a value to override the default value:

```
> greet hello nushell > greet world hello world
```

You can also combine a default value with a type requirement[7]:

```
def congratulate [age: int = 18] {   $"Happy birthday!
You are ($age) years old now!" }
```

If you want to check if an optional parameter is present or not and not just rely on a default value use optional positional parameters[8] instead.

Optional positional parameters

By default, positional parameters are required. If a positional parameter is not passed, we will encounter an error:

```
   × Missing required positional argument.     [entry #23:
1:1]  1   greet    ·            ·         missing name
     help: Usage: greet <name>
```

We can instead mark a positional parameter as optional by putting a question mark (`?`) after its name. For example:

[7]`#parameter-types`
[8]`#optional-positional-parameters`

```
def greet [name?: string] {   $"hello ($name)" } greet
```

Making a positional parameter optional does not change its name when accessed in the body. As the example above shows, it is still accessed with `$name`, despite the `?` suffix in the parameter list.

When an optional parameter is not passed, its value in the command body is equal to `null`. We can use this to act on the case where a parameter was not passed:

```
def greet [name?: string] {   if ($name == null) {
"hello, I don't know your name!"   } else {     $"hello
($name)"   } } greet
```

If you just want to set a default value when the parameter is missing it is simpler to use a default value[9] instead.

If required and optional positional parameters are used together, then the required parameters must appear in the definition first.

Flags

In addition to passing positional parameters, you can also pass named parameters by defining flags for your custom commands.

For example:

```
def greet [   name: string   --age: int ] {   [$name $age]
}
```

In the `greet` definition above, we define the `name` positional parameter as well as an `age` flag. This allows the caller of `greet` to optionally pass the `age` parameter as well.

You can call the above using:

```
> greet world --age 10
```

Or:

```
> greet --age 10 world
```

Or even leave the flag off altogether:

```
> greet world
```

[9]#parameters-with-a-default-value

Flags can also be defined to have a shorthand version. This allows you to pass a simpler flag as well as a longhand, easier-to-read flag.

Let's extend the previous example to use a shorthand flag for the `age` value:

```
def greet [  name: string  --age (-a): int ] {  [$name
$age] }
```

Note: Flags are named by their longhand name, so the above example would need to use `$age` and not `$a`.

Now, we can call this updated definition using the shorthand flag:

```
> greet -a 10 hello
```

Flags can also be used as basic switches. This means that their presence or absence is taken as an argument for the definition. Extending the previous example:

```
def greet [  name: string  --age (-a): int  --twice ]
{  if $twice {    [$name $name $age $age]  } else {
   [$name $age]  } }
```

And the definition can be either called as:

```
> greet -a 10 --twice hello
```

Or just without the switch flag:

```
> greet -a 10 hello
```

You can also assign it to true/false to enable/disable the flag too:

```
> greet -a 10 --switch=false > greet -a 10 --switch=true
```

But note that this is not the behavior you want: `> greet -a 10 --switch false`, here the value `false` will pass as a positional argument.

To avoid confusion, it's not allowed to annotate a boolean type on a flag:

```
def greet [    --twice: bool   # Not allowed ] { ... }
```

instead, you should define it as a basic switch: `def greet [--twice] { ... }`

Flags can contain dashes. They can be accessed by replacing the dash with an underscore:

```
def greet [  name: string  --age (-a): int  --two-times
] {  if $two_times {    [$name $name $age $age]  } else
{    [$name $age]  } }
```

Rest parameters

There may be cases when you want to define a command which takes any number of positional arguments. We can do this with a rest parameter, using the following ... syntax:

```
def greet [...name: string] {  print "hello all:"  for
$n in $name {    print $n  } } greet earth mars jupiter
venus
```

We could call the above definition of the `greet` command with any number of arguments, including none at all. All of the arguments are collected into `$name` as a list.

Rest parameters can be used together with positional parameters:

```
def greet [vip: string, ...name: string] {  print $"hello
to our VIP ($vip)"  print "and hello to everybody else:
"  for $n in $name {    print $n  } } #    $vip
    $name #    ---- ------------------------ greet moon
earth mars jupiter venus
```

To pass a list to a rest parameter, you can use the spread operator[10] (...):

```
> let planets = [earth mars jupiter venus] # This is equivalent
to the previous example > greet moon ...$planets
```

Documenting your command

In order to best help users of your custom commands, you can also document them with additional descriptions for the commands and parameters.

Taking our previous example:

```
def greet [  name: string  --age (-a): int ] {  [$name
$age] }
```

Once defined, we can run `help greet` to get the help information for the command:

[10] /book/operators#spread-operator

```
Usage:   > greet <name> {flags} Parameters:   <name> Flags:
   -h, --help: Display this help message   -a, --age <integer>
```

You can see the parameter and flag that we defined, as well as the `-h` help flag that all commands get.

To improve this help, we can add descriptions to our definition that will show up in the help:

```
# A greeting command that can greet the caller def greet
[   name: string      # The name of the person to greet
  --age (-a): int    # The age of the person ] {   [$name
$age] }
```

The comments that we put on the definition and its parameters then

:: warning Note A Nushell comment that continues on the same line for argument documentation purposes requires a space before the `#` pound sign. :::

Now, if we run `help greet`, we're given a more helpful help text:

```
A greeting command that can greet the caller Usage:
> greet <name> {flags} Parameters:   <name> The name of
the person to greet Flags:    -h, --help: Display this
help message   -a, --age <integer>: The age of the person
```

Pipeline Output

Custom commands stream their output just like built-in commands. For example, let's say we wanted to refactor this pipeline:

```
> ls | get name
```

Let's move `ls`[12] into a command that we've written:

```
def my-ls [] { ls }
```

We can use the output from this command just as we would `ls`[13].

[12] /commands/docs/ls.md
[13] /commands/docs/ls.md

```
> my-ls | get name                    0   myscript.nu  1
myscript2.nu  2   welcome_to_nushell.md
```

This lets us easily build custom commands and process their output. Note, that we don't use return statements like other languages. Instead, we build pipelines that output streams of data that can be connected to other pipelines.

Pipeline Input

Custom commands can also take input from the pipeline, just like other commands. This input is automatically passed to the block that the custom command uses.

Let's make our own command that doubles every value it receives as input:

```
def double [] {   each { |it| 2 * $it } }
```

Now, if we call the above command later in a pipeline, we can see what it does with the input:

```
> [1 2 3] | double       0   2  1   4  2   6
```

We can also store the input for later use using the `$in` variable:

```
def nullify [...cols] {   let start = $in   $cols | reduce
--fold $start { |col, df|     $df | upsert $col null
} }
```

Persisting

For information about how to persist custom commands so that they're visible when you start up Nushell, see the configuration chapter and add your startup script.

Aliases

Aliases in Nushell offer a way of doing a simple replacement of command calls (both external and internal commands). This allows you to create a shorthand name for a longer command, including its default arguments.

For example, let's create an alias called `ll` which will expand to `ls -l`.

```
> alias ll = ls -l
```

We can now call this alias:

```
> ll
```

Once we do, it's as if we typed `ls -l`. This also allows us to pass in flags or positional parameters. For example, we can now also write:

```
> ll -a
```

And get the equivalent to having typed `ls -l -a`.

List all loaded aliases

Your useable aliases can be seen in `scope aliases` and `help aliases`.

Persisting

To make your aliases persistent they must be added to your *config.nu* file by running `config nu` to open an editor and inserting them, and then restarting nushell. e.g. with the above `ll` alias, you can add `alias ll = ls -l` anywhere in *config.nu*

```
$env.config = {     # main configuration}alias ll = ls
-l # some other config and script loading
```

Piping in aliases

Note that `alias uuidgen = uuidgen | tr A-F a-f` (to make uuidgen on mac behave like linux) won't work. The solution is to define a command without parameters that calls the system program `uuidgen` via `^`.

```
def uuidgen [] { ^uuidgen | tr A-F a-f }
```

See more in the custom commands section of this book.

Or a more idiomatic example with nushell internal commands

```
def lsg [] { ls | sort-by type name -i | grid -c | str
trim }
```

displaying all listed files and folders in a grid.

Replacing existing commands using aliases

> *Caution! When replacing commands it is best to "back up" the command first and avoid recursion error.*

How to back up a command like `ls`:

```
alias core-ls = ls    # This will create a new alias core-
ls for ls
```

Now you can use `core-ls` as `ls` in your nu-programming. You will see further down how to use `core-ls`.

The reason you need to use alias is because, unlike `def`, aliases are position-dependent. So, you need to "back up" the old command first with an alias, before re-defining it. If you do not backup the command and you replace the command using `def` you get a recursion error.

```
def ls [] { ls }; ls    # Do *NOT* do this! This will throw
a recursion error #output: #Error: nu::shell::recursion_-
limit_reached # #  × Recursion limit (50) reached #
 [C:\Users\zolodev\AppData\Roaming\nushell\config.nu:807:
1] # 807  # 808  def ls [] { ls }; ls #     ·
     #     ·                 This called itself too many
times #
```

The recommended way to replace an existing command is to shadow the command. Here is an example shadowing the `ls` command.

```
# An escape hatch to have access to the original ls command
alias core-ls = ls # Call the built-in ls command with
a path parameter def old-ls [path] {   core-ls $path |
sort-by type name -i } # Shadow the ls command so that
you always have the sort type you want def ls [path?] {
  if $path == null {    old-ls .   } else {    old-ls
$path   } }
```

Operators

Nushell supports the following operators for common math, logic, and string operations:

Operator	Description
`+`	add
`-`	subtract
`*`	multiply
`/`	divide
`//`	floor division
`mod`	modulo
`**`	exponentiation (power)
`==`	equal
`!=`	not equal
`<`	less than
`<=`	less than or equal
`>`	greater than
`>=`	greater than or equal
`=~`	regex match / string contains another
`!~`	inverse regex match / string does *not* contain another
`in`	value in list
`not-in`	value not in list
`not`	logical not
`and`	and two Boolean expressions (short-circuits)
`or`	or two Boolean expressions (short-circuits)
`xor`	exclusive or two boolean expressions
`bit-or`	bitwise or
`bit-xor`	bitwise xor
`bit-and`	bitwise and
`bit-shl`	bitwise shift left
`bit-shr`	bitwise shift right
`starts-with`	string starts with
`ends-with`	string ends with
`++`	append lists

Parentheses can be used for grouping to specify evaluation order or for calling commands and using the results in an expression.

Order of Operations

Operations are evaluated in the following order (from highest precedence to lowest):

- Parentheses (`()`)
- Exponentiation/Power (`**`)
- Multiply (`*`), Divide (`/`), Integer/Floor Division (`//`), and Modulo (`mod`)
- Add (`+`) and Subtract (`-`)
- Bit shifting (`bit-shl`, `bit-shr`)
- Comparison operations (`==`, `!=`, `<`, `>`, `<=`, `>=`), membership tests (`in`, `not-in`, `starts-with`, `ends-with`), regex matching (`=~`, `!~`), and list appending (`++`)
- Bitwise and (`bit-and`)
- Bitwise xor (`bit-xor`)
- Bitwise or (`bit-or`)
- Logical and (`&&`, `and`)
- Logical xor (`xor`)
- Logical or (`||`, `or`)
- Assignment operations

```
> 3 * (1 + 2) 9
```

Types

Not all operations make sense for all data types. If you attempt to perform an operation on non-compatible data types, you will be met with an error message that should explain what went wrong:

```
> "spam" - 1 Error: nu::parser::unsupported_operation (
link)   × Types mismatched for operation.    [entry #49:
1:1]  1   "spam" - 1    ·           ·           int
·          doesn't support these values.    ·      string
        help: Change string or int to be the right types
```

```
and try again.
```

The rules might sometimes feel a bit strict, but on the other hand there should be less unexpected side effects.

Regular Expression / string-contains Operators

The `=~` and `!~` operators provide a convenient way to evaluate regular expressions[14]. You don't need to know regular expressions to use them - they're also an easy way to check whether 1 string contains another.

- `string =~ pattern` returns **true** if `string` contains a match for `pattern`, and **false** otherwise.
- `string !~ pattern` returns **false** if `string` contains a match for `pattern`, and **true** otherwise.

For example:

```
foobarbaz =~ bar # returns true foobarbaz !~ bar # returns
false ls | where name =~ ^nu # returns all files whose
names start with "nu"
```

Both operators use the Rust regex crate's `is_match()` function[15].

Case Sensitivity

Operators are usually case-sensitive when operating on strings. There are a few ways to do case-insensitive work instead:

1. In the regular expression operators, specify the `(?i)` case-insensitive mode modifier:

```
"FOO" =~ "foo" # returns false "FOO" =~ "(?i)foo" # returns
true
```

2. Use the `str contains`[16] command's `--insensitive` flag:

[14] https://cheatography.com/davechild/cheat-sheets/regular-expressions/
[15] https://docs.rs/regex/latest/regex/struct.Regex.html#method.is_match
[16] /commands/docs/str_contains.md

```
"FOO" | str contains --insensitive "foo"
```

3. Convert strings to lowercase with `str downcase`[17] before comparing:

```
("FOO" | str downcase) == ("Foo" | str downcase)
```

Spread operator

Nushell has a spread operator (`...`) for unpacking lists and records. You may be familiar with it if you've used JavaScript before. Some languages use `*` for their spread/splat operator. It can expand lists or records in places where multiple values or key-value pairs are expected.

There are three places you can use the spread operator:

- In list literals[18]
- In record literals[19]
- In command calls[20]

In list literals

Suppose you have multiple lists you want to concatenate together, but you also want to intersperse some individual values. This can be done with `append` and `prepend`, but the spread operator can let you do it more easily.

```
> let dogs = [Spot, Teddy, Tommy] > let cats = ["Mr. Humphrey
Montgomery", Kitten] > [     ...$dogs     Polly     ...
($cats | each { |it| $"($it) \(cat\)" })     ...[Porky
Bessie]     ...Nemo   ]                          0   Spot
                          1   Teddy
     2   Tommy                                  3   Polly
                           4   Mr. Humphrey Montgomery (cat)
   5   Kitten (cat)                          6   Porky
                        7   Bessie
```

[17] /commands/docs/str_downcase.md
[18] #in-list-literals
[19] #in-record-literals
[20] #in-command-calls

```
    8   ...Nemo
```

The below code is an equivalent version using `append`:

```
> $dogs |     append Polly |     append ($cats | each {
|it| $"($it) \(cat\)" }) |     append [Porky Bessie] |
   append ...Nemo
```

Note that each call to `append` results in the creation of a new list, meaning that in this second example, 3 unnecessary intermediate lists are created. This is not the case with the spread operator, so there may be (very minor) performance benefits to using `...` if you're joining lots of large lists together, over and over.

You may have noticed that the last item of the resulting list above is `"...Nemo"`. This is because inside list literals, it can only be used to spread lists, not strings. As such, inside list literals, it can only be used before variables (`...$foo`), subexpressions (`...(foo)`), and list literals (`...[foo]`).

The `...` also won't be recognized as the spread operator if there's any whitespace between it and the next expression:

```
> [ ... [] ]                  0   ...                  1   [list
0 items]
```

This is mainly so that `...` won't be confused for the spread operator in commands such as `mv ... $dir`.

In record literals

Let's say you have a record with some configuration information and you want to add more fields to this record:

```
> let config = { path: /tmp, limit: 5 }
```

You can make a new record with all the fields of `$config` and some new additions using the spread operator. You can use the spread multiple records inside a single record literal.

```
> {    ...$config,    users: [alice bob],    ...{ url:
 example.com },    ...(sys | get mem)   }
  path        /tmp              limit       5
                              users         0   alice
                 1   bob
```

```
url          example.com     total        8.3 GB
    free          2.6 GB          used          5.7 GB
        available    2.6 GB          swap total   2.1 GB
          swap free    18.0 MB          swap used    2.1
GB
```

Similarly to lists, inside record literals, the spread operator can only be used before variables (`...$foo`), subexpressions (`...(foo)`), and record literals (`...{foo:bar}`). Here too, there needs to be no whitespace between the `...` and the next expression for it to be recognized as the spread operator.

In command calls

You can also spread arguments to a command, provided that it either has a rest parameter or is an external command.

Here is an example custom command that has a rest parameter:

```
> def foo [ --flag req opt? ...args ] { [$flag, $req, $opt,
$args] | to nuon }
```

It has one flag (`--flag`), one required positional parameter (`req`), one optional positional parameter (`opt?`), and rest parameter (`args`).

If you have a list of arguments to pass to `args`, you can spread it the same way you'd spread a list inside a list literal[21]. The same rules apply: the spread operator is only recognized before variables, subexpressions, and list literals, and no whitespace is allowed in between.

```
> foo "bar" "baz" ...[1 2 3] # With ..., the numbers are
treated as separate arguments [false, bar, baz, [1, 2,
3]] > foo "bar" "baz" [1 2 3] # Without ..., [1 2 3] is
treated as a single argument [false, bar, baz, [[1, 2,
3]]]
```

A more useful way to use the spread operator is if you have another command with a rest parameter and you want it to forward its arguments to `foo`:

```
> def bar [ ...args ] { foo --flag "bar" "baz" ...$args
} > bar 1 2 3 [true, bar, baz, [1, 2, 3]]
```

You can spread multiple lists in a single call, and also intersperse individual arguments:

[21] `#in-list-literals`

```
> foo "bar" "baz" 1 ...[2 3] 4 5 ...(6..9 | take 2) last
[false, bar, baz, [1, 2, 3, 4, 5, 6, 7, last]]
```

Flags/named arguments can go after a spread argument, just like they can go after regular rest arguments:

```
> foo "bar" "baz" 1 ...[2 3] --flag 4 [true, bar, baz,
[1, 2, 3, 4]]
```

If a spread argument comes before an optional positional parameter, that optional parameter is treated as being omitted:

```
> foo "bar" ...[1 2] "not opt" # The null means no argument
was given for opt [false, bar, null, [1, 2, "not opt"]]
```

Variables and Subexpressions

There are two types of evaluation expressions in Nushell: variables and subexpressions. You know that you're looking at an evaluation expression because it begins with a dollar sign (`$`). This indicates that when Nushell gets the value in this position, it will need to run an evaluation step to process the expression and then use the resulting value. Both evaluation expression forms support a simple form and a 'path' form for working with more complex data.

Variables

The simpler of the two evaluation expressions is the variable. During evaluation, a variable is replaced by its value. After creating a variable, we can refer to it using `$` followed by its name.

Types of Variables

Immutable Variables An immutable variable cannot change its value after declaration. They are declared using the `let` keyword,

```
> let val = 42 > print $val 42
```

However, they can be 'shadowed'. Shadowing means that they are redeclared and their initial value cannot be used anymore within the same scope.

```
> let val = 42                  # declare a variable >
do { let val = 101;  $val }    # in an inner scope, shadow
the variable 101 > $val                            # in
the outer scope the variable remains unchanged 42
```

Mutable Variables A mutable variable is allowed to change its value by assignment. These are declared using the `mut` keyword.

```
> mut val = 42 > $val += 27 > $val 69
```

There are a couple of assignment operators used with mutable variables

Operator	Description
`=`	Assigns a new value to the variable
`+=`	Adds a value to the variable and makes the sum its new value
`-=`	Subtracts a value from the variable and makes the difference its new value
`*=`	Multiplies the variable by a value and makes the product its new value
`/=`	Divides the variable by a value and makes the quotient its new value
`++=`	Appends a list or a value to a variable

Note

1. *`+=`, `-=`, `*=` and `/=` are only valid in the contexts where their root operations are expected to work. For example, `+=` uses addition, so it can not be used for contexts where addition would normally fail*
2. *`++=` requires that either the variable **or** the argument is a list.*

More on Mutability Closures and nested `def`s cannot capture mutable variables from their environment. For example

```
# naive method to count number of elements in a list mut
x = 0 [1 2 3] | each { $x += 1 }   # error: $x is captured
in a closure
```

To use mutable variables for such behaviour, you are encouraged to use the loops

Constant Variables A constant variable is an immutable variable that can be fully evaluated at parse-time. These are useful with commands that need to know the value of an argument at parse time, like `source`[22], `use`[23] and `register`[24]. See how nushell code gets run for a deeper explanation. They are declared using the `const` keyword

```
const plugin = 'path/to/plugin' register $plugin
```

Variable Names

Variable names in Nushell come with a few restrictions as to what characters they can contain. In particular, they cannot contain these characters:

```
.  [  (  {  +  -  *  ^  /  =  !  <  >  &  |
```

It is common for some scripts to declare variables that start with `$`. This is allowed, and it is equivalent to the `$` not being there at all.

```
> let $var = 42 # identical to `let var = 42`
```

Variable Paths

A variable path works by reaching inside of the contents of a variable, navigating columns inside of it, to reach a final value. Let's say instead of `4`, we had assigned a table value:

```
> let my_value = [[name]; [testuser]]
```

We can use a variable path to evaluate the variable `$my_value` and get the value from the `name` column in a single step:

[22] /commands/docs/source.md
[23] /commands/docs/use.md
[24] /commands/docs/register.md

```
> $my_value.name.0 testuser
```

Sometimes, we don't really know the contents of a variable. Accessing values as shown above can result in errors if the path used does not exist. To more robustly handle this, we can use the question mark operator to return `null` in case the path does not exist, instead of an error, then we would write custom logic to handle the `null`.

For example, here, if row `0` does not exist on `name`, then `null` is returned. Without the question mark operator, an error would have been raised instead

```
> let files = (ls) > $files.name.0?
```

The question mark operator can be used to 'guard' any path

```
> let files = (ls) > $files.name?.0?
```

Subexpressions

You can always evaluate a subexpression and use its result by wrapping the expression with parentheses `()`. Note that previous versions of Nushell (prior to 0.32) used `$()`.

The parentheses contain a pipeline that will run to completion, and the resulting value will then be used. For example, `(ls)` would run the `ls`[25] command and give back the resulting table and `(git branch --show-current)` runs the external git command and returns a string with the name of the current branch. You can also use parentheses to run math expressions like `(2 + 3)`.

Subexpressions can also be pipelines and not just single commands. If we wanted to get a table of files larger than ten kilobytes, we could use a subexpression to run a pipeline and assign its result to a variable:

```
> let big_files = (ls | where size > 10kb) > $big_files
                             #     name      type     size
     modified                             0   Cargo.lock
 File   155.3 KB   17 hours ago  1   README.md    File
 15.9 KB   17 hours ago
```

Subexpressions and paths

Subexpressions also support paths. For example, let's say we wanted to get a list of the filenames in the current directory. One way to do this is to use a pipeline:

[25] /commands/docs/ls.md

```
> ls | get name
```

We can do a very similar action in a single step using a subexpression path:

```
> (ls).name
```

It depends on the needs of the code and your particular style which form works best for you.

Short-hand subexpressions (row conditions)

Nushell supports accessing columns in a subexpression using a simple short-hand. You may have already used this functionality before. If, for example, we wanted to only see rows from `ls`[26] where the entry is at least ten kilobytes we could write:

```
> ls | where size > 10kb
```

The `where size > 10kb` is a command with two parts: the command name `where`[27] and the short-hand expression `size > 10kb`. We say short-hand because `size` here is the shortened version of writing `$it.size`. This could also be written in any of the following ways:

```
> ls | where $it.size > 10kb > ls | where ($it.size > 10kb)
 > ls | where {|$x| $x.size > 10kb }
```

For the short-hand syntax to work, the column name must appear on the left-hand side of the operation (like `size` in `size > 10kb`).

Control Flow

Nushell provides several commands that help determine how different

:: tip One thing to note is that all of the commands discussed on this page use blocks[28]. This means you can mutate environmental variables[29] and other mutable variables[30] in them. :::

[26] /commands/docs/ls.md
[27] /commands/docs/where.md
[28] /book/types_of_data.html#blocks
[29] /book/environment.html
[30] http://localhost:8080/book/variables_and_subexpressions.html#mutable-variables

Already covered

Below we cover some commands related to control flow, but before we get to them, it's worthwhile to note there are several pieces of functionality and concepts that have already been covered in other sections that are also related to control flow or that can be used in the same situations. These include:

- Pipes on the pipelines[31] page.
- Closures on the types of data[32] page.
- Iteration commands on the working with lists[33] page. Such as:
  - `each`[34]
  - `where`[35]
  - `reduce`[36]

Choice (Conditionals)

The following commands execute code based on some given condition.

:: tip The choice/conditional commands are expressions so they return values, unlike the other commands on this page. This means the following works.

```
> 'foo' | if $in == 'foo' { 1 } else { 0 } | $in + 2 3
```

:::

`if`

`if`[37] evaluates branching blocks[38] of code based on the results of one or more conditions similar to the "if" functionality in other programming languages. For example:

[31] /book/pipelines.html
[32] /book/types_of_data.html
[33] /book/working_with_lists.html
[34] /commands/docs/each.html
[35] /commands/docs/where.html
[36] /commands/docs/reduce.html
[37] /commands/docs/if.html
[38] /book/types_of_data.html#blocks

```
> if $x > 0 { 'positive' }
```

Returns `'positive`' when the condition is `true` (`$x` is greater than zero) and `null` when the condition is `false` (`$x` is less than or equal to zero).

We can add an `else` branch to the `if` after the first block which executes and returns the resulting value from the `else` block when the condition is `false`. For example:

```
> if $x > 0 { 'positive' } else { 'non-positive' }
```

This time it returns `'positive'` when the condition is `true` (`$x` is greater than zero) and `'non-positive`' when the condition is `false` (`$x` is less than or equal to zero).

We can also chain multiple `if`s together like the following:

```
> if $x > 0 { 'positive' } else if $x == 0 { 'zero' } else
{ "negative" }
```

When the first condition is `true` (`$x` is greater than zero) it will return `'positive'`, when the first condition is `false` and the next condition is `true` (`$x` equals zero) it will return `'zero'`, otherwise it will return `'negative'` (when `$x` is less than zero).

`match`

`match`[39] executes one of several conditional branches based on the value given to match. You can also do some pattern matching[40] to unpack values in composite types like lists and records.

Basic usage of `match`[41] can conditionally run different code like a "switch" statement common in other languages. `match`[42] checks if the value after the word `match`[43] is equal to the value at the start of each branch before the `=>` and if it does, it executes the code after that branch's `=>`.

```
> match 3 {     1 => 'one',     2 => {          let w =
```

[39] /commands/docs/match.html
[40] /cookbook/pattern_matching.html
[41] /commands/docs/match.html
[42] /commands/docs/match.html
[43] /commands/docs/match.html

```
'w'          't' + $w + 'o'     },     3 => 'three',
4 => 'four' } three
```

The branches can either return a single value or, as shown in the second branch, can return the results of a block[44].

Catch all branch You can have also have a catch all condition for if the given value doesn't match any of the other conditions by having a branch whose matching value is `_`.

```
> let foo = match 7 {     1 => 'one',     2 => 'two',
   3 => 'three',     _ => 'other number' } > $foo other
number
```

(Reminder, `match`[45] is an expression which is why we can assign the result to `$foo` here).

Pattern Matching You can "unpack" values from types like lists and records with pattern matching[46]. You can then assign variables to the parts you want to unpack and use them in the matched expressions.

```
> let foo = { name: 'bar', count: 7 } > match $foo {
  { name: 'bar', count: $it } => ($it + 3),    { name:
 _, count: $it } => ($it + 7),     _ => 1 } 10
```

The `_` in the second branch means it matches any record with field `name` and `count`, not just ones where `name` is `'bar'`.

Guards You can also add an additional condition to each branch called a "guard" to determine if the branch should be matched. To do so, after the matched pattern put `if` and then the condition before the `=>`.

```
> let foo = { name: 'bar', count: 7 } > match $foo {
  { name: 'bar', count: $it } if $it < 5 => ($it + 3),
   { name: 'bar', count: $it } if $it >= 5 => ($it + 7)
,     _ => 1 } 14
```

[44] /book/types_of_data.html#blocks
[45] /commands/docs/match.html
[46] /cookbook/pattern_matching.html

***

You can find more details about `match`[47] in the pattern matching cookbook page[48].

Loops

The loop commands allow you to repeat a block of code multiple times.

Loops and other iterating commands

The functionality of the loop commands is similar to commands that apply a closure over elements in a list or table like `each`[49] or `where`[50] and many times you can accomplish the same thing with either. For example:

```
> mut result = [] > for $it in [1 2 3] { $result = ($result
| append ($it + 1)) } > $result       0   2    1   3
2   4        > [1 2 3] | each { $in + 1 }        0   2
  1   3    2   4
```

While it may be tempting to use loops if you're familiar with them in other languages, it is considered more in the Nushell-style[51] (idiomatic) to use commands that apply closures when you can solve a problem either way. The reason for this is because of a pretty big downside with using loops.

Loop disadvantages The biggest downside of loops is that they are statements, unlike `each`[52] which is an expression. Expressions, like `each`[53] always result in some output value, however statements do not.

This means that they don't work well with immutable variables and using immutable variables is considered a more Nushell-style[54]. Without a mutable variable declared beforehand in the example in the

[47] `/commands/docs/match.html`
[48] `https://www.nushell.sh/cookbook/pattern_matching.html`
[49] `/commands/docs/each.html`
[50] `/commands/docs/where.html`
[51] `book/thinking_in_nu.html`
[52] `/commands/docs/each.html`
[53] `/commands/docs/each.html`
[54] `/book/thinking_in_nu.html#variables-are-immutable`

previous section, it would be impossible to use `for`[55] to get the list of numbers with incremented numbers, or any value at all.

Statements also don't work in Nushell pipelines which require some output. In fact Nushell will give an error if you try:

```
> [1 2 3] | for x in $in { $x + 1 } | $in ++ [5 6 7] Er-
ror: nu::parser::unexpected_keyword   × Statement used
in pipeline.     [entry #5:1:1]  1   [1 2 3] | for x in
$in { $x + 1 } | $in ++ [5 6 7]    ·                 ·
            not allowed in pipeline        help: 'for'
keyword is not allowed in pipeline. Use 'for' by itself,
outside of a pipeline.
```

Because Nushell is very pipeline oriented, this means using expression commands like `each`[56] is typically more natural than loop statements.

Loop advantages If loops have such a big disadvantage, why do they exist? Well, one reason is that closures, like `each`[57] uses, can't modify mutable variables in the surrounding environment. If you try to modify a mutable variable in a closure you will get an error:

```
> mut foo = [] > [1 2 3] | each { $foo = ($foo | append
($in + 1)) } Error: nu::parser::expected_keyword   × Capture
of mutable variable.     [entry #8:1:1]  1   [1 2 3] |
each { $foo = ($foo | append ($in + 1)) }    ·
              ·                      capture of mutable
variable
```

If you modify an environmental variable in a closure, you can, but it will only modify it within the scope of the closure, leaving it unchanged everywhere else. Loops, however, use blocks[58] which means they can modify a regular mutable variable or an environmental variable within the larger scope.

```
> mut result = [] > for $it in [1 2 3] { $result = ($result
| append ($it + 1)) } > $result       0   2    1   3
2   4
```

[55] `/commands/docs/each.html`
[56] `/commands/docs/each.html`
[57] `/commands/docs/each.html`
[58] `/book/types_of_data.html#blocks`

`for`

`for`[59] loops over a range or collection like a list or a table.

```
> for x in [1 2 3] { $x * $x | print } 1 4 9
```

Expression command alternatives

- `each`[60]
- `par-each`[61]
- `where`[62]/`filter`[63]
- `reduce`[64]

`while`

`while`[65] loops the same block of code until the given condition is `false`.

```
> mut x = 0; while $x < 10 { $x = $x + 1 }; $x 10
```

Expression command alternatives The "until" and other "while" commands

- `take until`[66]
- `take while`[67]
- `skip until`[68]
- `skip while`[69]

59 /commands/docs/for.html
60 /commands/docs/each.html
61 /commands/docs/par-each.html
62 /commands/docs/where.html
63 /commands/docs/filter.html
64 /commands/docs/reduce.html
65 /commands/docs/while.html
66 /commands/docs/take_until.html
67 /commands/docs/take_while.html
68 /commands/docs/skip_until.html
69 /commands/docs/skip_while.html

`loop`

`loop`[70] loops a block infinitely. You can use `break`[71] (as described in the next section) to limit how many times it loops. It can also be handy for continuously running scripts, like an interactive prompt.

```
> mut x = 0; loop { if $x > 10 { break }; $x = $x + 1 };
$x 11
```

`break`

`break`[72] will stop executing the code in a loop and resume execution after the loop. Effectively "break"ing out of the loop.

```
> for x in 1..10 { if $x > 3 { break }; print $x } 1 2 3
```

`continue`

`continue`[73] will stop execution of the current loop, skipping the rest of the code in the loop, and will go to the next loop. If the loop would normally end, like if `for`[74] has iterated through all the given elements, or if `while`[75]'s condition is now false, it won't loop again and execution will continue after the loop block.

```
> mut x = -1; while $x <= 6 { $x = $x + 1; if $x mod 3
== 0 { continue }; print $x } 1 2 4 5 7
```

Errors

`error make`

`error make`[76] creates an error that stops execution of the code and any code that called it, until either it is handled by a `try`[77] block, or it ends the script and outputs the error message. This functionality is the same as "exceptions" in other languages.

[70] /commands/docs/loop.html
[71] /commands/docs/break.html
[72] /commands/docs/break.html
[73] /commands/docs/continue.html
[74] /commands/docs/for.html
[75] /commands/docs/while.html
[76] /commands/docs/error_make.html
[77] /commands/docs/try.html

```
> print 'printed'; error make { msg: 'Some error info'
}; print 'unprinted' printed Error:   × Some error info
    [entry #9:1:1] 1  print 'printed'; error make { msg:
 'Some error info' }; print 'unprinted'    ·
                 ·                         originates from
here
```

The record passed to it provides some information to the code that catches it or the resulting error message.

You can find more information about `error make`[78] and error concepts on the Creating your own errors page[79].

`try`

`try`[80] will catch errors created anywhere in the `try`[81]'s code block and resume execution of the code after the block.

```
> try { error make { msg: 'Some error info' }}; print 'Resuming'
Resuming
```

This includes catching built in errors.

```
> try { 1 / 0 }; print 'Resuming' Resuming
```

The resulting value will be **`nothing`** if an error occurs and the returned value of the block if an error did not occur.

If you include a **`catch`** block after the `try`[82] block, it will execute the code in the **`catch`** block if an error occurred in the `try`[83] block.

```
> try { 1 / 0 } catch { 'An error happened!' } | $in ++
' And now I am resuming.' An error happened! And now I
am resuming.
```

It will not execute the **`catch`** block if an error did not occur.

Other

`return`

`return`[84] Ends a closure or command early where it is called, without

[78] /commands/docs/error_make.html
[79] /book/creating_errors.html
[80] /commands/docs/try.html
[81] /commands/docs/try.html
[82] /commands/docs/try.html
[83] /commands/docs/try.html
[84] /commands/docs/return.html

running the rest of the command/closure, and returns the given value. Not often necessary since the last value in a closure or command is also returned, but it can sometimes be convenient.

```
def 'positive-check' [it] {     if $it > 0 {         return
'positive'     };     'non-positive' }
```

```
> positive-check 3 positive > positive-check (-3) non-positive
> let positive_check = {|it| if $it > 0 { return 'positive'
}; 'non-positive' }> do $positive_check 3 positive > do
$positive_check (-3) non-positive
```

Scripts

In Nushell, you can write and run scripts in the Nushell language. To run a script, you can pass it as an argument to the `nu` commandline application:

```
> nu myscript.nu
```

This will run the script to completion in a new instance of Nu. You can also run scripts inside the *current* instance of Nu using `source`[85]:

```
> source myscript.nu
```

Let's look at an example script file:

```
# myscript.nu def greet [name] {   ["hello" $name] } greet
"world"
```

A script file defines the definitions for custom commands as well as the main script itself, which will run after the custom commands are defined.

In the above, first `greet` is defined by the Nushell interpreter. This allows us to later call this definition. We could have written the above as:

```
greet "world" def greet [name] {   ["hello" $name] }
```

There is no requirement that definitions have to come before the parts of the script that call the definitions, allowing you to put them where you feel comfortable.

[85] /commands/docs/source.md

How scripts are processed

In a script, definitions run first. This allows us to call the definitions using the calls in the script.

After the definitions run, we start at the top of the script file and run each group of commands one after another.

Script lines

To better understand how Nushell sees lines of code, let's take a look at an example script:

```
a b; c | d
```

When this script is run, Nushell will first run the `a` command to completion and view its results. Next, Nushell will run `b; c | d` following the rules in the "Semicolons" section.

Parameterizing Scripts

Script files can optionally contain a special "main" command. `main` will be run after any other Nu code, and is primarily used to add parameters to scripts. You can pass arguments to scripts after the script name (`nu <script name> <script args>`).

For example:

```
# myscript.nu def main [x: int] {   $x + 10 }
```

```
> nu myscript.nu 100 110
```

Argument Type Interpretation

By default, arguments provided to a script are interpreted with the type `Type::Any`, implying that they are not constrained to a specific data type and can be dynamically interpreted as fitting any of the available data types during script execution.

In the previous example, `main [x: int]` denotes that the argument x should possess an integer data type. However, if arguments are not explicitly typed, they will be parsed according to their apparent data type.

For example:

```
# implicit_type.nu def main [x] {  $"Hello ($x | describe) ($x)" } # explicit_type.nu def main [x: string] {  $"Hello ($x | describe) ($x)" }
```

```
> nu implicit_type.nu +1 Hello int 1 > nu explicit_type.nu +1 Hello string +1
```

Subcommands

A script can have multiple sub-commands like `run`, `build`, etc. which allows to execute a specific main sub-function. The important part is to expose them correctly with `def main [] {}`. See more details in the Custom Command section.

For example:

```
# myscript.nu def "main run" [] {     print "running" } def "main build" [] {     print "building" } # important for the command to be exposed to the outside def main [] {}
```

```
> nu myscript.nu build building > nu myscript.nu run running
```

Shebangs (#!)

On Linux and macOS you can optionally use a shebang[86] to tell the OS that a file should be interpreted by Nu. For example, with the following in a file named `myscript`:

```
#!/usr/bin/env nu "Hello World!"
```

```
> ./myscript Hello World!
```

For script to have access to standard input, `nu` should be invoked with `--stdin` flag:

```
#!/usr/bin/env -S nu --stdin echo $"stdin: ($in)"
```

[86] https://en.wikipedia.org/wiki/Shebang_(Unix)

```
> echo "Hello World!" | ./myscript stdin: Hello World!
```

Modules

Similar to many other programming languages, Nushell also has modules to organize your code. Each module is a "bag" containing a bunch of definitions (typically commands) that you can export (take out of the bag) and use in your current scope. Since Nushell is also a shell, modules allow you to modify environment variables when importing them.

Quick Overview

There are three ways to define a module in Nushell:

1. "inline"
    - `module spam { ... }`
2. from a file
    - using a .nu file as a module
3. from a directory
    - directory must contain a `mod.nu` file

In Nushell, creating a module and importing definitions from a module are two different actions. The former is done using the `module` keyword. The latter using the `use` keyword. You can think of `module` as "wrapping definitions into a bag" and `use` as "opening a bag and taking definitions from it". In most cases, calling `use` will create the module implicitly, so you typically don't need to use `module` that much.

You can define the following things inside a module:

- Commands* (`def`)
- Aliases (`alias`)
- Known externals* (`extern`)
- Submodules (`module`)
- Imported symbols from other modules (`use`)

- Environment setup (`export-env`)

Only definitions marked with `export` are possible to access from outside of the module ("take out of the bag"). Definitions not marked with `export` are allowed but are visible only inside the module (you could call them private). (*`export-env` is special and does not require `export`.*)

**These definitions can also be named `main` (see below).*

"Inline" modules

The simplest (and probably least useful) way to define a module is an "inline" module can be defined like this:

```
module greetings {     export def hello [name: string]
{        $"hello ($name)!"    }    export def hi [where:
 string] {        $"hi ($where)!"    }}use greetings
hello hello "world"
```

You can paste the code into a file and run it with `nu`, or type into the REPL.

First, we create a module (put `hello` and `hi` commands into a "bag" called `greetings`), then we import the `hello` command from the module (find a "bag" called `greetings` and take `hello` command from it) with `use`.

Modules from files

A .nu file can be a module. Just take the contents of the module block from the example above and save it to a file `greetings.nu`. The module name is automatically inferred as the stem of the file ("greetings").

```
# greetings.nu export def hello [name: string] {    $"hello
($name)!" } export def hi [where: string] {    $"hi ($where)
!" }
```

then

```
> use greetings.nu hello > hello
```

The result should be similar as in the previous section.

> ***Note** that the `use greetings.nu hello` call here first implicitly creates the `greetings` module, then takes `hello` from it. You could also write it as `module greetings`.*

nu, use greetings hello. Using module can be useful if you're not interested in any definitions from the module but want to, e.g., re-export the module (export module greet-ings.nu).

Modules from directories

Finally, a directory can be imported as a module. The only condition is that it needs to contain a `mod.nu` file (even empty, which is not particularly useful, however). The `mod.nu` file defines the root module. Any submodules (`export module`) or re-exports (`export use`) must be declared inside the `mod.nu` file. We could write our `greetings` module like this:

In the following examples, / is used at the end to denote that we're importing a directory but it is not required.

```
# greetings/mod.nu export def hello [name: string] {
  $"hello ($name)!" } export def hi [where: string] {
  $"hi ($where)!" }
```

then

```
> use greetings/ hello > hello
```

The name of the module follows the same rule as module created fro

:: tip You can define `main` command inside `mod.nu` to create a command named after the module directory. :::

Import Pattern

Anything after the `use`[87] keyword forms an **import pattern** which controls how the definitions are imported. The import pattern has the following structure `use head members...` where `head` defines the module (name of an existing module, file, or directory). The members are optional and specify what exactly should be imported from the module.

Using our `greetings` example:

```
use greetings
```

imports all symbols prefixed with the `greetings` namespace (can call `greetings hello` and `greetings hi`).

87 /commands/docs/use.md

```
use greetings hello
```

will import the `hello` command directly without any prefix.

```
use greetings [hello, hi]
```

imports multiple definitions<> directly without any prefix.

```
use greetings *
```

will import all names directly without any prefix.

`main`

Exporting a command called `main` from a module defines a command named as the module. Let's extend our `greetings` example:

```
# greetings.nu export def hello [name: string] {    $"hello
($name)!" } export def hi [where: string] {    $"hi ($where)
!" } export def main [] {    "greetings and salutations!"
}
```

then

```
> use greetings.nu > greetings greetings and salutations!
> greetings hello world hello world!
```

The `main` is exported only when

- no import pattern members are used (`use greetings.nu`)
- glob member is used (`use greetings.nu *`)

Importing definitions selectively (`use greetings.nu hello` or `use greetings.nu [hello hi]`) does not define the `greetings` command from `main`. You can, however, selectively import `main` using `use greetings main` (or `[main]`) which defines *only* the `greetings` command without pulling in `hello` or `hi`.

Apart from commands (`def`, `def --env`), known externals (`extern`) can also be named `main`.

Submodules and subcommands

Submodules are modules inside modules. They are automatically created when you call `use` on a directory: Any .nu files inside the directory are implicitly added as submodules of the main module. There are two more ways to add a submodule to a module:

1. Using `export module`
2. Using `export use`

The difference is that `export module some-module` *only* adds the module as a submodule, while `export use some-module` *also* re-exports the submodule's definitions. Since definitions of submodules are available when importing from a module, `export use some-module` is typically redundant, unless you want to re-export its definitions without the namespace prefix.

> ***Note*** *> `module` without `export` defines only a local module, it does not export a submodule.*

Let's illustrate this with an example. Assume three files:

```
# greetings.nu export def hello [name: string] {    $"hello
($name)!" } export def hi [where: string] {    $"hi ($where)
!" } export def main [] {    "greetings and salutations!"
}
```

```
# animals.nu export def dog [] {    "haf" } export def
cat [] {    "meow" }
```

```
# voice.nu export use greetings.nu * export module animals.
nu
```

Then:

```
> use voice.nu > voice animals dog haf > voice animals
cat meow > voice hello world hello world > voice hi there
hi there! > voice greetings greetings and salutations!
```

As you can see, defining the submodule structure also shapes the command line API. In Nushell, namespaces directly folds into subcommands. This is true for all definitions: aliases, commands, known externals, modules.

Environment Variables

Modules can also define an environment using export-env[88]:

```
# greetings.nu export-env {     $env.MYNAME = "Arthur,
King of the Britons" } export def hello [] {     $"hello
($env.MYNAME)" }
```

When use[89] is evaluated, it will run the code inside the export-env[90] block and merge its environment into the current scope:

```
> use greetings.nu > $env.MYNAME Arthur, King of the Britons
> greetings hello hello Arthur, King of the Britons!
```

::: tip You can put a complex code defining your environment without polluting the namespace of the module, for example:

```
    def tmp [] { "tmp" }     def other [] { "other" }
   let len = (tmp | str length)     load-env {
OTHER_ENV: (other)          TMP_LEN: $len     } }
```

Only `$env.TMP_LEN` and `$env.OTHER_ENV` are preserved after evaluating the `export-env` module. :::

Caveats

Like any programming language, Nushell is also a product of a tradeoff and there are limitations to our module system.

`export-env` runs only when the `use` call is *evaluated*

If you also want to keep your variables in separate modules and export their environment, you could try to export use[91] it:

```
# purpose.nu export-env {     $env.MYPURPOSE = "to build
an empire." } export def greeting_purpose [] {     $"Hello
($env.MYNAME). My purpose is ($env.MYPURPOSE)" }
```

and then use it

[88] /commands/docs/export-env.md
[89] /commands/docs/use.md
[90] /commands/docs/export-env.md
[91] /commands/docs/export_use.md

```
> use purpose.nu > purpose greeting_purpose
```

However, this won't work, because the code inside the module is not *evaluated*, only *parsed* (only the `export-env` block is evaluated when you call `use purpose.nu`). To export the environment of `greetings.nu`, you need to add it to the `export-env` module:

```
# purpose.nu export-env {     use greetings.nu     $env.
MYPURPOSE = "to build an empire." } export def greeting_-
purpose [] {     $"Hello ($env.MYNAME). My purpose is (
$env.MYPURPOSE)" }
```

then

```
> use purpose.nu > purpose greeting_purpose Hello Arthur,
King of the Britons. My purpose is to build an empire.
```

Calling `use purpose.nu` ran the `export-env` block inside `purpose.nu` which in turn ran `use greetings.nu` which in turn ran the `export-env` block inside `greetings.nu`, preserving the environment changes.

Module file / command cannot be named after parent module

- Module directory cannot contain .nu file named after the directory (`spam/spam.nu`)
  - Consider a `spam` directory containing both `spam.nu` and `mod.nu`, calling `use spam *` would create an ambiguous situation where the `spam` module would be defined twice.
- Module cannot contain file named after the module
  - Same case as above: Module `spam` containing both `main` and `spam` commands would create an ambiguous situation when exported as `use spam *`.

Examples

This section describes some useful patterns using modules.

Local Definitions

Anything defined in a module without the `export`[92] keyword will work only in the module's scope.

[92] `/commands/docs/export.md`

```
# greetings.nu use tools/utils.nu generate-prefix  # visible
only locally (we assume the file exists) export def hello
[name: string] {     greetings-helper "hello" "world" }
export def hi [where: string] {     greetings-helper "hi"
"there" } def greetings-helper [greeting: string, subject:
 string] {     $"(generate-prefix)($greeting) ($subject)
!" }
```

then

```
> use greetings.nu * > hello "world" hello world! > hi
"there" hi there! > greetings-helper "foo" "bar"  # fails
because 'greetings-helper' is not exported > generate-prefix
 # fails because the command is imported only locally inside
the module
```

Dumping files into directory

A common pattern in traditional shells is dumping and auto-sourcing files from a directory (for example, loading custom completions). In Nushell, doing this directly is currently not possible, but directory modules can still be used.

Here we'll create a simple completion module with a submodule dedicated to some Git completions:

1. Create the completion directory `mkdir ($nu.default-config-dir | path join completions)`
2. Create an empty `mod.nu` for it `touch ($nu.default-config-dir | path join completions mod.nu)`
3. Put the following snippet in `git.nu` under the `completions` directory

```
export extern main [     --version(-v)     -C: string
  # ... etc. ] export extern add [     --verbose(-v)
 --dry-run(-n)     # ... etc. ] export extern checkout
[     branch: string@complete-git-branch ] def complete-
git-branch [] {     # ... code to list git branches }
```

4. Add `export module git.nu` to `mod.nu`
5. Add the parent of the `completions` directory to your NU_LIB_-DIRS inside `env.nu`

```
$env.NU_LIB_DIRS = [     ...     $nu.default-config-dir
]
```

6. import the completions to Nushell in your `config.nu` `use completions *` Now you've set up a directory where you can put your completion files and you should have some Git completions the next time you start Nushell

> ***Note*** *This will use the file name (in our example* `git` *from* `git.nu`*) as the module name. This means some completions might not work if the definition has the base command in it's name. For example, if you defined our known externals in our* `git.nu` *as* `export extern 'git push' []`*, etc. and followed the rest of the steps, you would get subcommands like* `git git push`*, etc. You would need to call* `use completions git *` *to get the desired subcommands. For this reason, using* `main` *as outlined in the step above is the preferred way to define subcommands.*

Setting environment + aliases (conda style)

`def --env` commands, `export-env` block and aliases can be used to dynamically manipulate "virtual environments" (a concept well known from Python).

We use it in our official virtualenv integration `https://github.com/pypa/virtualenv/blob/main/src/virtualenv/activation/nushell/activate.nu`

Another example could be our unofficial Conda module: `https://github.com/nushell/nu_scripts/blob/f86a060c10f132407694e9ba0f536bfe`
`modules/virtual_environments/conda.nu`

> ***Warning*** *Work In Progress*

Hiding

Any custom command or alias, imported from a module or not, can be "hidden", restoring the previous definition. We do this with the `hide`[93] command:

[93] `/commands/docs/hide.md`

```
> def foo [] { "foo" }> foo foo > hide foo > foo  # error!
command not found!
```

The `hide`[94] command also accepts import patterns, just like `use`[95]. The import pattern is interpreted slightly differently, though. It can be one of the following:

`hide foo` or `hide greetings`

- If the name is a custom command or an environment variable, hides it directly. Otherwise:
- If the name is a module name, hides all of its exports prefixed with the module name

`hide greetings hello`

- Hides only the prefixed command / environment variable

`hide greetings [hello, hi]`

- Hides only the prefixed commands / environment variables

`hide greetings *`

- Hides all of the module's exports, without the prefix

> ***Note*** *> `hide` is not a supported keyword at the root of a module (unlike `def` etc.)*

Overlays

Overlays act as "layers" of definitions (custom commands, aliases, environment variables) that can be activated and deactivated on demand. They resemble virtual environments found in some languages, such as Python.

Note: To understand overlays, make sure to check Modules[96] *first as overlays build on top of modules.*

[94] /commands/docs/hide.md
[95] /commands/docs/use.md
[96] modules.md

Basics

First, Nushell comes with one default overlay called `zero`. You can inspect which overlays are active with the `overlay list`[97] command. You should see the default overlay listed there.

To create a new overlay, you first need a module:

```
> module spam {     export def foo [] {         "foo"
  }     export alias bar = "bar"     export-env {
  load-env { BAZ: "baz" }     } }
```

We'll use this module throughout the chapter, so whenever you see o

:: tip The module can be created by any of the three methods described in Modules[98]:

- "inline" modules (used in this example)
- file
- directory :::

To create the overlay, call `overlay use`[99]:

```
> overlay use spam > foo foo > bar bar > $env.BAZ baz >
overlay list        0   zero  1   spam
```

It brought the module's definitions into the current scope and evalua

:: tip In the following sections, the `>` prompt will be preceded by the name of the last active overlay. `(spam)> some-command` means the `spam` overlay is the last active overlay when the command was typed. :::

Removing an Overlay

If you don't need the overlay definitions anymore, call `overlay hide`[103]:

```
(spam)> overlay hide spam (zero)> foo Error: Can't run
executable... (zero)> overlay list        0   zero
```

The overlays are also scoped. Any added overlays are removed at the end of the scope:

[97] `/commands/docs/overlay_list.md`
[98] `modules.md`
[99] `/commands/docs/overlay_use.md`
[103] `/commands/docs/overlay_remove.md`

```
(zero)> do { overlay use spam; foo }  # overlay is active
only inside the block foo (zero)> overlay list       0
  zero
```

The last way to remove an overlay is to call `overlay hide`[104] without an argument which will remove the last active overlay.

Overlays Are Recordable

Any new definition (command, alias, environment variable) is recorded into the last active overlay:

```
(zero)> overlay use spam (spam)> def eggs [] { "eggs" }
```

Now, the `eggs` command belongs to the `spam` overlay. If we remove the overlay, we can't call it anymore:

```
(spam)> overlay hide spam (zero)> eggs Error: Can't run
executable...
```

But we can bring it back!

```
(zero)> overlay use spam (spam)> eggs eggs
```

Overlays remember what you add to them and store that information even

:: tip Sometimes, after adding an overlay, you might not want custom definitions to be added into it. The solution can be to create a new empty overlay that would be used just for recording the custom changes:

```
(zero)> overlay use spam (spam)> module scratchpad { } (
spam)> overlay use scratchpad (scratchpad)> def eggs []
{ "eggs" }
```

The `eggs` command is added into `scratchpad` while keeping `spam` intact.

To make it less verbose, you can use the `overlay new`[105] command:

[104] /commands/docs/overlay_remove.md
[105] /commands/docs/overlay_new.md

```
(zero)> overlay use spam (spam)> overlay new scratchpad
(scratchpad)> def eggs [] { "eggs" }
```

:::

Prefixed Overlays

The `overlay use`[106] command would take all commands and aliases from the module and put them directly into the current namespace. However, you might want to keep them as subcommands behind the module's name. That's what `--prefix` is for:

```
(zero)> module spam {     export def foo [] { "foo" } }
(zero)> overlay use --prefix spam (spam)> spam foo foo
```

Note that this does not apply for environment variables.

Rename an Overlay

You can change the name of the added overlay with the `as` keyword:

```
(zero)> module spam { export def foo [] { "foo" } } (zero)
> overlay use spam as eggs (eggs)> foo foo (eggs)> overlay
hide eggs (zero)>
```

This can be useful if you have a generic script name, such as virtualenv's `activate.nu` but you want a more descriptive name for your overlay.

Preserving Definitions

Sometimes, you might want to remove an overlay, but keep all the custom definitions you added without having to redefine them in the next active overlay:

```
(zero)> overlay use spam (spam)> def eggs [] { "eggs" }
(spam)> overlay hide --keep-custom spam (zero)> eggs eggs
```

The `--keep-custom` flag does exactly that.

One can also keep a list of environment variables that were defined inside an overlay, but remove the rest, using the `--keep-env` flag:

[106] /commands/docs/overlay_use.md

```
(zero)> module spam {     export def foo [] { "foo" }
   export-env { $env.FOO = "foo" } } (zero)> overlay use
spam (spam)> overlay hide spam --keep-env [ FOO ] (zero)
> foo Error: Can't run executable... (zero)> $env.FOO foo
```

Ordering Overlays

The overlays are arranged as a stack. If multiple overlays contain the same definition, say `foo`, the one from the last active one would take precedence. To bring an overlay to the top of the stack, you can call `overlay use`[107] again:

```
(zero)> def foo [] { "foo-in-zero" } (zero)> overlay use
spam (spam)> foo foo (spam)> overlay use zero (zero)> foo
foo-in-zero (zero)> overlay list       0   spam  1   zero
```

Now, the `zero` overlay takes precedence.

Command signature

nu commands can be given explicit signatures; take `str stats`[108] as an example, the signature is like this:

```
def "str stats" []: string -> record { }
```

The type names between the `:` and the opening curly brace `{` describe the command's input/output pipeline types. The input type for a given pipeline, in this case `string`, is given before the `->`; and the output type `record` is given after `->`. There can be multiple input/output types. If there are multiple input/output types, they can be placed within brackets and separated with commas, as in `str join`[109]:

```
def "str join" [separator?: string]: [list -> string, string
-> string] { }
```

It says that the `str join`[110] command takes an optional `string` type argument, and an input pipeline of either a `list` (implying `list<any>`)

[107] /commands/docs/overlay_use.md
[108] /commands/docs/str_stats.md
[109] /commands/docs/str_join.md
[110] /commands/docs/str_join.md

with output type of `string`, or a single `string`, again with output type of `string`.

Some commands don't accept or require data through the input pipeline, thus the input type will be `<nothing>`. The same is true for the output type if the command returns `null` (e.g. `rm`[111] or `hide`[112]):

```
def hide [module: string, members?]: nothing -> nothing
{ }
```

Testing your Nushell code

The standard library has a unit testing framework to ensure that your code works as expected.

Quick start

Have a file, called `test_math.nu`:

```
use std assert #[test] def test_addition [] {     assert
equal (1 + 2) 3 } #[test] def test_skip [] {     assert
skip } #[test] def test_failing [] {     assert false "This
is just for testing" }
```

Run the tests:

```
  use std testing run-tests   run-tests INF|2023-04-12T10:
42:29.099|Running tests in test_math Error:   × This is
just for testing        [C:\wip\test_math.nu:13:1]   13
def test_failing [] {  14        assert false "This is just
for testing"      ·                    ·
It is not true.   15   }        WRN|2023-04-12T10:42:31.
086|Test case test_skip is skipped Error:   × some tests
did not pass (see complete errors above):
test_math test_addition          test_math test_failing
       s test_math test_skip
```

Assert commands

The foundation for every assertion is the `std assert` command. If the condition is not true, it makes an error. For example:

[111] /commands/docs/rm.md
[112] /commands/docs/hide.md

```
  std assert (1 == 2) Error:   × Assertion failed.
  [entry #13:1:1]  1   std assert (1 == 2)    ·
          ·                   It is not true.
```

Optionally, a message can be set to show the intention of the assert command, what went wrong or what was expected:

```
  std assert ($a == 19) $"The lockout code is wrong, received:
 ($a)" Error:   × The lockout code is wrong, received:
13     [entry #25:1:1]  1   std assert ($a == 19) $"The
lockout code is wrong, received: ($a)"    ·
         ·                    It is not true.
```

There are many assert commands, which behave exactly as the base one with the proper operator. The additional value for them is the ability for better error messages.

For example this is not so helpful without additional message:

```
  std assert ($b | str contains $a) Error:   × Assertion
failed.     [entry #35:1:1]  1   assert ($b | str contains
$a)    ·                         ·
It is not true.
```

While with using `assert str contains`:

```
  std assert str contains $b $a Error:   × Assertion failed.
      [entry #34:1:1]  1   assert str contains $b $a
·                            ·
'haystack' does not contain 'a needle'.
```

In general for base `assert` command it is encouraged to always provide the additional message to show what went wrong. If you cannot use any built-in assert command, you can create a custom one with passing the label for `error make`[113] for the `assert` command:

```
def "assert even" [number: int] {    std assert ($number
mod 2 == 0) --error-label {         start: (metadata $number)
.span.start,         end: (metadata $number).span.end,
       text: $"($number) is not an even number",    }}
```

Then you'll have your detailed custom error message:

[113] /commands/docs/error_make.md

```
  let $a = 13  assert even $a Error:   × Assertion failed.
     [entry #37:1:1]  1   assert even $a    ·
      ·                    13 is not an even number
```

Test modules & test cases

The naming convention for test modules is `test_<your_module>.nu` and `test_<test name>` for test cases.

In order for a function to be recognized as a test by the test runner it needs to be annotated with `#[test]`.

The following annotations are supported by the test runner:

- test - test case to be executed during test run
- test-skip - test case to be skipped during test run
- before-all - function to run at the beginning of test run. Returns a global context record that is piped into every test function
- before-each - function to run before every test case. Returns a per-test context record that is merged with global context and piped into test functions
- after-each - function to run after every test case. Receives the context record just like the test cases
- after-all - function to run after all test cases have been executed. Receives the global context record

The standard library itself is tested with this framework, so you can find many examples in the Nushell repository[114].

Setting verbosity

The unit testing framework uses the `log` commands from the standard library to display information, so you can set `NU_LOG_LEVEL` if you want more or less details:

```
  std run-tests  NU_LOG_LEVEL=DEBUG std run-tests  NU_-
LOG_LEVEL=WARNING std run-tests
```

[114] https://github.com/nushell/nushell/blob/main/crates/nu-std/tests/

Best practices

This page is a working document collecting syntax guidelines and best practices we have discovered so far. The goal of this document is to eventually land on a canonical Nushell code style, but as for now it is still work in progress and subject to change. We welcome discussion and contributions.

Keep in mind that these guidelines are not required to be used in external repositories (not ours), you can change them in the way you want, but please be consistent and follow your rules.

All escape sequences should not be interpreted literally, unless directed to do so. In other words, treat something like `\n` like the new line character and not a literal slash followed by `n`.

Formatting

Defaults

It's recommended to assume that by default no spaces or tabs allowed, but the following rules define where they are allowed.

Basic

- **It's recommended to** put one space before and after pipe `|` symbol, commands, subcommands, their options and arguments.

- **It's recommended to** never put several consecutive spaces unless they are part of string.

- **It's recommended to** omit commas between list items.

Correct:

```
'Hello, Nushell! This is a gradient.' | ansi gradient -
-fgstart '0x40c9ff' --fgend '0xe81cff'
```

Incorrect:

```
# - too many spaces after "|": 2 instead of 1 'Hello, Nushell!
This is a gradient.' |  ansi gradient --fgstart '0x40c9ff'
--fgend '0xe81cff'
```

One-line format One-line format is a format for writing all commands in one line.

It's recommended to default to this format:

1. unless you are writing scripts
2. in scripts for lists and records unless they either:
   (a) more than 80 characters long
   (b) contain nested lists or records
3. for pipelines less than 80 characters long not containing items should be formatted with a long format

Rules:

1. parameters:
   (a) **It's recommended to** put one space after comma `,` after block or closure parameter.
   (b) **It's recommended to** put one space after pipe `|` symbol denoting block or closure parameter list end.
2. block and closure bodies:
   (a) **It's recommended to** put one space after opening block or closure curly brace `{` if no explicit parameters defined.
   (b) **It's recommended to** put one space before closing block or closure curly brace `}`.
3. records:
   (a) **It's recommended to** put one space after `:` after record key.
   (b) **It's recommended to** put one space after comma `,` after key value.
4. lists:
   (a) **It's recommended to** put one space after comma `,` after list value.
5. surrounding constructs:

   (a) **It's recommended to** put one space before opening square `[`, curly brace `{`, or parenthesis `(` if preceding symbol is not the same.
   (b) **It's recommended to** put one space after closing square `]`, curly brace `}`, or parenthesis `)` if following symbol is not the same.

Correct:

```
[[status]; [UP] [UP]] | all {|el| $el.status == UP } [1
2 3 4] | reduce {|it, acc| $it + $acc } [1 2 3 4] | reduce
{|it acc| $it + $acc } {x: 1, y: 2} {x: 1 y: 2} [1 2] |
zip [3 4] [] (1 + 2) * 3
```

Incorrect:

```
# too many spaces before "|el|": no space is allowed [[status];
[UP] [UP]] | all { |el| $el.status == UP } # too many spaces
before ",": no space is allowed [1 2 3 4] | reduce {|it
, acc| $it + $acc } # too many spaces before "x": no space
is allowed { x: 1, y: 2} # too many spaces before "[3":
 one space is required [1 2] | zip  [3 4] # too many spaces
before "]": no space is allowed [ ] # too many spaces before
")": no space is allowed (1 + 2 ) * 3
```

Multi-line format Multi-line format is a format for writing all commands in several lines. It inherits all rules from one-line format and modifies them slightly.

It's recommended to default to this format:

1. while you are writing scripts
2. in scripts for lists and records while they either:
   (a) more than 80 characters long
   (b) contain nested lists or records
3. for pipelines more 80 characters long

Rules:

1. general:

   (a) **It's required to omit** trailing spaces.

2. block and closure bodies:

   (a) **It's recommended to** put each body pipeline on a separate line.

3. records:

   (a) **It's recommended to** put each record key-value pair on separate line.

4. lists:

   (a) **It's recommended to** put each list item on separate line.

5. surrounding constructs:

   (a) **It's recommended to** put one `\n` before opening square `[`, curly brace `{`, or parenthesis `(` if preceding symbol is not the and applying this rule produce line with a singular parenthesis.

   (b) **It's recommended to** put one `\n` after closing square `]`, curly brace `}`, or parenthesis `)` if following symbol is not the same and applying this rule produce line with a singular parenthesis.

Correct:

```
[[status]; [UP] [UP]] | all {|el|    $el.status == UP }
[1 2 3 4] | reduce {|it, acc|    $it + $acc } {x: 1, y:
 2} [  {name: "Teresa", age: 24},  {name: "Thomas", age:
 26} ] let selectedProfile = (for it in ($credentials |
transpose name credentials) {    echo $it.name })
```

Incorrect:

```
# too many spaces before "|el|": no space is allowed (like
in one-line format) [[status]; [UP] [UP]] | all { |el|
   # too few "\n" before "}": one "\n" is required
$el.status == UP} # too many spaces before "2": one space
is required (like in one-line format) [1  2 3 4] | reduce
{|it, acc|    $it + $acc } {   # too many "\n" before
"x": one-line format required as no nested lists or record
exist   x: 1,   y: 2 } # too few "\n" before "{": multi-
```

```
line format required as there are two nested records [{name:
 "Teresa", age: 24},  {name: "Thomas", age: 26}] let selectedProfile
= (    # too many "\n" before "foo": no "\n" is allowed
    for it in ($credentials | transpose name credentials)
 {        echo $it.name })
```

Options and parameters of custom commands

- **It's recommended to** keep count of all positional parameters less than or equal to 2, for remaining inputs use options. Assume that command can expect source and destination parameter, like `mv`: source and target file or directory.
- **It's recommended to** use positional parameters unless they can't be used due to rules listed here or technical restrictions. For instance, when there are several kinds of optional parameters (but at least one parameter should be provided) use options. Great example of this is `ansi gradient` command where at least foreground or background must be passed.
- **It's recommended to** provide both long and short options.

Documentation

- **It's recommended to** provide documentation for all exported entities (like custom commands) and their inputs (like custom command parameters and options).

Chapter 5

Nu as a Shell

The Nu Fundamentals and Programming in Nu chapter focused mostly on the language aspects of Nushell. This chapter sheds the light on the parts of Nushell that are related to the Nushell interpreter (the Nushell REPL[1]). Some of the concepts are directly a part of the Nushell programming language (such as environment variables) while others are implemented purely to enhance the interactive experience (such as hooks) and thus are not present when, for example, running a script.

Many parameters of Nushell can be configured. The config itself is stored as an environment variable. Furthermore, Nushell has several different configuration files that are run on startup where you can put custom commands, aliases, etc.

A big feature of any shell are environment variables. In Nushell, environment variables are scoped and can have any type supported by Nushell. This brings in some additional design considerations so please refer to the linked section for more details.

The other sections explain how to work with stdout, stderr and exit codes, how to escape a command call to the external command call, and how to configure 3rd party prompts to work with Nushell.

An interesting feature of Nushell is shells which let you work in multiple directories simultaneously.

Nushell also has its own line editor Reedline. With Nushell's config, it is possible to configure some of the Reedline's features, such as the prompt, keybindings, history, or menus.

It is also possible to define custom signatures for external commands which lets you define custom completions for them (the custom

[1] https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

completions work also for Nushell custom commands).

Coloring and Theming in Nu goes into more detail about how to configure Nushell's appearance.

If you want to schedule some commands to run in the background, Background task in Nu provide a simple guideline for you to follow.

And finally, hooks allow you to insert fragments of Nushell code to run at certain events.

Configuration

Nushell Configuration with `env.nu` and `config.nu`

Nushell uses a configuration system that loads and runs two Nushell script files at launch time:

- `env.nu` is used to define environment variables. These typically get used in the second config file, config.nu.
- `config.nu` is used to add definitions, aliases, and more to the global namespace. It can use the environment variables defined in `env.nu`, which is why there's two separate files.

You can check where Nushell is reading these config files from by calling `$nu.env-path` and `$nu.config-path`.

```
> $nu.env-path /Users/FirstNameLastName/Library/Application
Support/nushell/env.nu
```

(You can think of the Nushell config loading sequence as executing two REPL[2] lines on startup: `source /path/to/env.nu` and `source /path/to/config.nu`. Therefore, using `env.nu` for environment and `config.nu` for other config is just a convention.)

When you launch Nushell without these files set up, Nushell will prompt you to download the `default env.nu`[3] and `default config.nu`[4].

You can browse the default files for default values of environment variables and a list of all configurable settings.

[2] https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

[3] https://github.com/nushell/nushell/blob/main/crates/nu-utils/src/sample_config/default_env.nu

[4] https://github.com/nushell/nushell/blob/main/crates/nu-utils/src/sample_config/default_config.nu

Configuring `$env.config`

Nushell's main settings are kept in the `config` environment variable as a record. This record can be created using:

```
$env.config = {  ... }
```

You can also shadow `$env.config` and update it:

```
$env.config = ($env.config | upsert <field name> <field
value>)
```

By convention, this variable is defined in the `config.nu` file.

Environment

You can set environment variables for the duration of a Nushell session using the `$env.<var> = <val>` structure inside the `env.nu` file. For example:

```
$env.FOO = 'BAR'
```

(Although $env.config is an environment variable, it is still defined by convention inside config.nu.)

These are some important variables to look at for Nushell-specific settings:

- `LS_COLORS`: Sets up colors per file type in ls
- `PROMPT_COMMAND`: Code to execute for setting up the prompt (block or string)
- `PROMPT_COMMAND_RIGHT`: Code to execute for setting up the right prompt (block)
- `PROMPT_INDICATOR = " "`: The indicator printed after the prompt (by default ">"-like Unicode symbol)
- `PROMPT_INDICATOR_VI_INSERT = ": "`
- `PROMPT_INDICATOR_VI_NORMAL = "  "`
- `PROMPT_MULTILINE_INDICATOR = "::: "`

Configurations with built-in commands

Starting with release v0.64 of Nushell, we have introduced two new commands(`config nu`[5] and `config env`[6]) which help you quickly edit nu configurations with your preferred text editor/IDE

Nushell follows underneath orders to locate the editor:

1. `$config.buffer_editor`
2. `$env.EDITOR`
3. `$env.VISUAL`

Note: Previous versions of Nushell were launching `notepad` on windows, otherwise `nano` when these variables weren't found. We removed defaulting to `notepad` on Windows since `notepad` is now distributed via the Windows Store and there will be a possibility of not having `notepad` at all.

Color Config section

You can learn more about setting up colors and theming in the associated chapter.

Remove Welcome Message

To remove the welcome message, you need to edit your `config.nu` by typing `config nu` in your terminal, then you go to the global configuration `$env.config` and set `show_banner` option to false, like this:

Configuring Nu as a login shell

To use Nu as a login shell, you'll need to configure the `$env` variable. This sets up the environment for external programs.

To get an idea of which environment variables are set up by your current login shell, start a new shell session, then run nu in that shell.

You can then configure some `$env.<var> = <val>` that setup the same environment variables in your nu login shell. Use this command to generate some `$env.<var> = <val>` for all the environment variables:

[5] `/commands/docs/config_nu.md`
[6] `/commands/docs/config_env.md`

```
$env | reject config | transpose key val | each {|r| echo
$"$env.($r.key) = '($r.val)'"} | str join (char nl)
```

This will print out `$env.<var> = <val>` lines, one for each environment variable along with its setting. You may not need all of them, for instance the `PS1` variable is bash specific.

Next, on some distros you'll also need to ensure Nu is in the /etc/shells list:

```
> cat /etc/shells # /etc/shells: valid login shells /
bin/sh /bin/dash /bin/bash /bin/rbash /usr/bin/screen /
usr/bin/fish /home/jonathan/.cargo/bin/nu
```

With this, you should be able to `chsh` and set Nu to be your login shell. After a logout, on your next login you should be greeted with a shiny Nu prompt.

Configuration with `login.nu`

If Nushell is used as a login shell, you can use a specific configuration file which is only sourced in this case. Therefore a file with name `login.nu` has to be in the standard configuration directory.

The file `login.nu` is sourced after `env.nu` and `config.nu`, so that you can overwrite those configurations if you need.

There is an environment variable `$nu.loginshell-path` containing the path to this file.

macOS: Keeping `/usr/bin/open` as `open`

Some tools (e.g. Emacs) rely on an `open`[7] command to open files on Mac. As Nushell has its own `open`[8] command which has different semantics and shadows `/usr/bin/open`, these tools will error out when trying to use it. One way to work around this is to define a custom command for Nushell's `open`[9] and create an alias for the system's `open`[10] in your `config.nu` file like this:

```
def nuopen [arg, --raw (-r)] { if $raw { open -r $arg }
else { open $arg } } alias open = ^open
```

[7] /commands/docs/open.md
[8] /commands/docs/open.md
[9] /commands/docs/open.md
[10] /commands/docs/open.md

The ^ symbol *escapes* the Nushell `open` command, which invokes the operating system's `open` command. For more about escape and ^ see the chapter about escapes.

PATH configuration

In Nushell, the PATH environment variable[11] (Path on Windows) is a list of paths. To append a new path to it, you can use `$env.<var> = <val>` and `append`[12] in `env.nu`:

```
$env.PATH = ($env.PATH | split row (char esep) | append
'/some/path')
```

This will append `/some/path` to the end of PATH; you can also use `prepend`[13] to add entries to the start of PATH.

Note the `split row (char esep)` step. We need to add it because in `env.nu`, the environment variables inherited from the host process are still strings. The conversion step of environment variables to Nushell values happens after reading the config files (see also the Environment section). After that, for example in the Nushell REPL when `PATH`/`Path` is a list , you can use `append`[14]/`prepend`[15] directly.

To prepend a new path only if not already listed, one can add to `env.nu`:

```
# create a new string holding the desired path let my_-
path = ( $nu.home-path | path join "bin" ) # return $env.
PATH if $my_path is already listed, return $env.PATH with
$my_path prepended otherwise $env.PATH = ( if $my_path
in $env.PATH { $env.PATH } else { $env.PATH | prepend $my_-
path } )
```

Homebrew

Homebrew[16] is a popular package manager that often requires PATH configuration. To add it to your Nushell PATH:

[11] https://en.wikipedia.org/wiki/PATH_(variable)
[12] /commands/docs/append.md
[13] /commands/docs/prepend.md
[14] /commands/docs/append.md
[15] /commands/docs/prepend.md
[16] https://brew.sh/

```
# macOS ARM64 (Apple Silicon) $env.PATH = ($env.PATH |
split row (char esep) | prepend '/opt/homebrew/bin') #
Linux $env.PATH = ($env.PATH | split row (char esep) |
prepend '/home/linuxbrew/.linuxbrew/bin')
```

Environment

A common task in a shell is to control the environment that external applications will use. This is often done automatically, as the environment is packaged up and given to the external application as it launches. Sometimes, though, we want to have more precise control over what environment variables an application sees.

You can see the current environment variables in the $env variable:

```
~> $env | table -e

  ENV_CONVERSIONS
                                          PATH    from_-
string   <Closure 32>
                 to_string    <Closure 34>



                                          Path    from_-
string   <Closure 36>
                 to_string    <Closure 38>



  HOME                               /Users/jelle
                         LSCOLORS
      GxFxCxDxBxegedabagaced                      | ...
                              | ...
                  |

```

In Nushell, environment variables can be any value and have any type. You can see the type of an env variable with the describe command, for example: `$env.PROMPT_COMMAND | describe`.

To send environment variables to external applications, the values will need to be converted to strings. See Environment variable conversions[17] on how this works.

[17] #environment-variable-conversions

The environment is initially created from the Nu configuration files and from the environment that Nu is run inside of.

Setting environment variables

There are several ways to set an environment variable:

$env.VAR assignment

Using the `$env.VAR = "val"` is the most straightforward method

```
> $env.FOO = 'BAR'
```

So, if you want to extend the Windows `Path` variable, for example, you could do that as follows.

```
$env.Path = ($env.Path | prepend 'C:\path\you\want\to\add')
```

Here we've prepended our folder to the existing folders in the Path, so it will have the highest priority. If you want to give it the lowest priority instead, you can use the `append`[18] command.

`load-env`[19]

If you have more than one environment variable you'd like to set, you can use `load-env`[20] to create a table of name/value pairs and load multiple variables at the same time:

```
> load-env { "BOB": "FOO", "JAY": "BAR" }
```

One-shot environment variables

These are defined to be active only temporarily for a duration of executing a code block. See Single-use environment variables for details.

Calling a command defined with `def --env`[21]

See Defining environment from custom commands for details.

[18] `/commands/docs/append.md`
[19] `/commands/docs/load-env.md`
[20] `/commands/docs/load-env.md`
[21] `/commands/docs/def.md`

Using module's exports

See Modules[22] for details.

Reading environment variables

Individual environment variables are fields of a record that is stored in the `$env` variable and can be read with `$env.VARIABLE`:

```
> $env.FOO BAR
```

Sometimes, you may want to access an environmental variable which might be unset. Consider using the question mark operator to avoid an error:

```
> $env.FOO | describe Error: nu::shell::column_not_found
  × Cannot find column      [entry #1:1:1]  1   $env.FOO
   ·          ·           cannot find column 'FOO'    ·
    value originates here       > $env.FOO? | describe noth-
ing > $env.FOO? | default "BAR" BAR
```

Alternatively, you can check for the presence of an environmental variable with `in`:

```
> $env.FOO BAR > if "FOO" in $env { >     echo $env.FOO
> } BAR
```

Scoping

When you set an environment variable, it will be available only in the current scope (the block you're in and any block inside of it).

Here is a small example to demonstrate the environment scoping:

```
> $env.FOO = "BAR" > do {     $env.FOO = "BAZ"     $env.
FOO == "BAZ" } true > $env.FOO == "BAR" true
```

Changing directory

Common task in a shell is to change directory with the `cd`[23] command. In Nushell, calling `cd`[24] is equivalent to setting the `PWD` environment variable. Therefore, it follows the same rules as other environment variables (for example, scoping).

[22] `modules.md`
[23] `/commands/docs/cd.md`
[24] `/commands/docs/cd.md`

Single-use environment variables

A common shorthand to set an environment variable once is available, inspired by Bash and others:

```
> FOO=BAR $env.FOO BAR
```

You can also use `with-env`[25] to do the same thing more explicitly:

```
> with-env { FOO: BAR } { $env.FOO } BAR
```

The `with-env`[26] command will temporarily set the environment variable to the value given (here: the variable "FOO" is given the value "BAR"). Once this is done, the block will run with this new environment variable set.

Permanent environment variables

You can also set environment variables at startup so they are available for the duration of Nushell running. To do this, set an environment variable inside the Nu configuration file. For example:

```
# In config.nu $env.FOO = 'BAR'
```

Defining environment from custom commands

Due to the scoping rules, any environment variables defined inside a custom command will only exist inside the command's scope. However, a command defined as `def --env`[27] instead of `def`[28] (it applies also to `export def`[29], see Modules[30]) will preserve the environment on the caller's side:

```
> def --env foo [] {    $env.FOO = 'BAR' } > foo > $env.
FOO BAR
```

[25] /commands/docs/with-env.md
[26] /commands/docs/with-env.md
[27] /commands/docs/def.md
[28] /commands/docs/def.md
[29] /commands/docs/export_def.md
[30] modules.md

Environment variable conversions

You can set the `ENV_CONVERSIONS` environment variable to convert other environment variables between a string and a value. For example, the default environment config[31] includes conversion of PATH (and Path used on Windows) environment variables from a string to a list. After both `env.nu` and `config.nu` are loaded, any existing environment variable specified inside `ENV_CONVERSIONS` will be translated according to its `from_string` field into a value of any type. External tools require environment variables to be strings, therefore, any non-string environment variable needs to be converted first. The conversion of value -> string is set by the `to_string` field of `ENV_CONVERSIONS` and is done every time an external command is run.

Let's illustrate the conversions with an example. Put the following in your config.nu:

```
$env.ENV_CONVERSIONS = {     # ... you might have Path
and PATH already there, add:     FOO : {         from_
string: { |s| $s | split row '-' }        to_string: {
|v| $v | str join '-' }     } }
```

Now, within a Nushell instance:

```
> with-env { FOO : 'a-b-c' } { nu }  # runs Nushell with
FOO env. var. set to 'a-b-c' > $env.FOO   0   a   1   b
  2   c
```

You can see the `$env.FOO` is now a list in a new Nushell instance with the updated config. You can also test the conversion manually by

```
> do $env.ENV_CONVERSIONS.FOO.from_string 'a-b-c'
```

Now, to test the conversion list -> string, run:

```
> nu -c '$env.FOO' a-b-c
```

Because `nu` is an external program, Nushell translated the `[ a b c ]` list according to `ENV_CONVERSIONS.FOO.to_string` and passed it to the `nu` process. Running commands with `nu -c` does not load the config file, therefore the env conversion for `FOO` is missing and it is displayed as a plain string -- this way we can verify the translation was successful.

[31]https://github.com/nushell/nushell/blob/main/crates/nu-utils/src/sample_config/default_env.nu

You can also run this step manually by `do $env.ENV_CONVERSIONS.FOO.to_string [a b c]`

(Important! The environment conversion string -> value happens ***after*** *the env.nu and config.nu are evaluated. All environment variables in env.nu and config.nu are still strings unless you set them manually to some other values.)*

Removing environment variables

You can remove an environment variable only if it was set in the current scope via `hide-env`[32]:

```
> $env.FOO = 'BAR' ... > hide-env FOO
```

The hiding is also scoped which both allows you to remove an environment variable temporarily and prevents you from modifying a parent environment from within a child scope:

```
> $env.FOO = 'BAR' > do {     hide-env FOO     # $env.FOO
does not exist   } > $env.FOO BAR
```

Stdout, Stderr, and Exit Codes

An important piece of interop between Nushell and external commands is working with the standard streams of data coming from the external.

The first of these important streams is stdout.

Stdout

Stdout is the way that most external apps will send data into the pipeline or to the screen. Data sent by an external app to its stdout is received by Nushell by default if it's part of a pipeline:

```
> external | str join
```

The above would call the external named `external` and would redirect the stdout output stream into the pipeline. With this redirection, Nushell can then pass the data to the next command in the pipeline, here `str join`[33].

Without the pipeline, Nushell will not do any redirection, allowing it to print directly to the screen.

[32] /commands/docs/hide_env.md
[33] /commands/docs/str_join.md

Stderr

Another common stream that external applications often use to print error messages is stderr. By default, Nushell does not do any redirection of stderr, which means that by default it will print to the screen.

You can force Nushell to do a redirection by using `do { ... } | complete`. For example, if we wanted to call the external above and redirect its stderr, we would write:

```
> do { external } | complete
```

Exit code

Finally, external commands have an "exit code". These codes help give a hint to the caller whether the command ran successfully.

Nushell tracks the last exit code of the recently completed external in one of two ways. The first way is with the `LAST_EXIT_CODE` environment variable.

```
> do { external } > $env.LAST_EXIT_CODE
```

The second uses a command called `complete`[34].

Using the `complete`[35] command

The `complete`[36] command allows you to run an external to completion, and gather the stdout, stderr, and exit code together in one record.

If we try to run the external `cat` on a file that doesn't exist, we can see what `complete`[37] does with the streams, including the redirected stderr:

```
> do { cat unknown.txt } | complete
 stdout
    stderr      cat: unknown.txt: No such file or directory
  exit_code   1
```

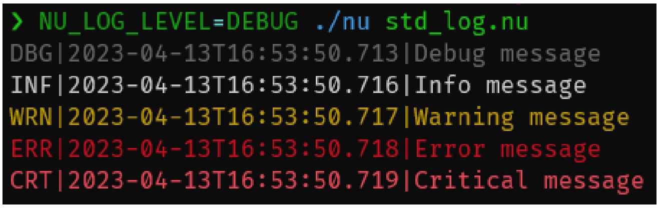
[34] /commands/docs/complete.md
[35] /commands/docs/complete.md
[36] /commands/docs/complete.md
[37] /commands/docs/complete.md

echo, print, and log commands

The echo[38] command is mainly for *pipes*. It returns its arguments, ignoring the piped-in value. There is usually little reason to use this over just writing the values as-is.

In contrast, the print[39] command prints the given values to stdout as plain text. It can be used to write to standard error output, as well. Unlike echo[40], this command does not return any value (`print | describe` will return "nothing"). Since this command has no output, there is no point in piping it with other commands.

The standard library[41] has commands to write out messages in different logging levels. For example:



```
> NU_LOG_LEVEL=DEBUG ./nu std_log.nu
DBG|2023-04-13T16:53:50.713|Debug message
INF|2023-04-13T16:53:50.716|Info message
WRN|2023-04-13T16:53:50.717|Warning message
ERR|2023-04-13T16:53:50.718|Error message
CRT|2023-04-13T16:53:50.719|Critical message
```

The log level for output can be set with the `NU_LOG_LEVEL` environment variable:

```
NU_LOG_LEVEL=DEBUG nu std_log.nu
```

Using out>, err> to redirect stdout and stderr to files

If you want to redirect output to file, you can just type something like this:

```
cat unknown.txt out> out.log err> err.log
```

If you want to redirect both stdout and stderr to the same file, just type something like this:

```
cat unknown.txt out+err> log.log
```

[38] /commands/docs/echo.md
[39] /commands/docs/print.md
[40] /commands/docs/echo.md
[41] /book/standard_library.md

Raw streams

Both stdout and stderr are represented as “raw streams” inside of Nushell. These are streams of bytes rather than structured data, which are what internal Nushell commands use.

Because streams of bytes can be difficult to work with, especially given how common it is to use output as if it was text data, Nushell attempts to convert raw streams into text data. This allows other commands to pull on the output of external commands and receive strings they can further process.

Nushell attempts to convert to text using UTF-8. If at any time the conversion fails, the rest of the stream is assumed to always be bytes.

If you want more control over the decoding of the byte stream, you can use the `decode`[42] command. The `decode`[43] command can be inserted into the pipeline after the external, or other raw stream-creating command, and will handle decoding the bytes based on the argument you give decode. For example, you could decode shift-jis text this way:

```
> 0x[8a 4c] | decode shift-jis
```

Escaping to the system

Nu provides a set of commands that you can use across different OSes (“internal” commands), and having this consistency is helpful. Sometimes, though, you want to run an external command that has the same name as an internal Nu command. To run the external `ls`[44] or `date`[45] command, for example, you use the caret (^) command. Escaping with the caret prefix calls the command that’s in the user’s PATH (e.g. `/bin/ls` instead of Nu’s internal `ls`[46] command).

Nu internal command:

```
> ls
```

Escape to external command:

```
> ^ls
```

[42] /commands/docs/decode.md
[43] /commands/docs/decode.md
[44] /commands/docs/ls.md
[45] /commands/docs/date.md
[46] /commands/docs/ls.md

Windows note

When running an external command on Windows, nushell used to[47] use Cmd.exe[48] to run the command, as a number of common commands on Windows are actually shell builtins and not available as separate executables. Coming from CMD.EXE contains a list of these commands and how to map them to nushell native concepts.

How to configure 3rd party prompts

nerdfonts

nerdfonts are not required but they make the presentation much better.
site[49]
repo[50]

oh-my-posh

site[51]
repo[52]
If you like oh-my-posh[53], you can use oh-my-posh with Nushell with a few steps. It works great with Nushell. How to setup oh-my-posh with Nushell:

1. Install Oh My Posh and download oh-my-posh's themes following guide[54].

2. Download and install a nerd font[55].

3. Generate the .oh-my-posh.nu file. By default it will be generated to your home directory. You can use `--config` to specify a theme, other wise, oh-my-posh comes with a default theme.

[47] https://www.nushell.sh/blog/2022-08-16-nushell-0_67.html#windows-cmd-exe-changes-rgwood
[48] https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/cmd
[49] https://www.nerdfonts.com
[50] https://github.com/ryanoasis/nerd-fonts
[51] https://ohmyposh.dev/
[52] https://github.com/JanDeDobbeleer/oh-my-posh
[53] https://ohmyposh.dev/
[54] https://ohmyposh.dev/docs/installation/linux
[55] https://github.com/ryanoasis/nerd-fonts

4. Initialize oh-my-posh prompt by adding in ~/.config/nushell/config.nu(or the path output by `$nu.config-path`) to source ~/.oh-my-posh.nu.

```
# Generate the .oh-my-posh.nu file > oh-my-posh init nu
--config ~/.poshthemes/M365Princess.omp.json # Initialize
oh-my-posh.nu at shell startup by adding this line in your
config.nu file > source ~/.oh-my-posh.nu
```

For MacOS users:

1. You can install oh-my-posh by `brew`, just following the guide here[56]

2. Download and install a nerd font[57].

3. Set the PROMPT_COMMAND in the file output by `$nu.config-path`, here is a code snippet:

```
let posh_dir = (brew --prefix oh-my-posh | str trim) let
posh_theme = $'($posh_dir)/share/oh-my-posh/themes/' #
Change the theme names to: zash/space/robbyrussel/powerline/
powerlevel10k_lean/ # material/half-life/lambda Or double
lines theme: amro/pure/spaceship, etc. # For more [Themes
demo](https://ohmyposh.dev/docs/themes) $env.PROMPT_COMMAND
= { || oh-my-posh prompt print primary --config $'($posh_-
theme)/zash.omp.json' } # Optional $env.PROMPT_INDICATOR
= $"(ansi y)$> (ansi reset)"
```

Starship

site[58]

repo[59]

1. Follow the links above and install Starship.

2. Install nerdfonts depending on your preferences.

3. Use the config example below. Make sure to set the `STARSHIP_SHELL` environment variable.

[56] https://ohmyposh.dev/docs/installation/macos
[57] https://github.com/ryanoasis/nerd-fonts
[58] https://starship.rs/
[59] https://github.com/starship/starship

::: tip An alternate way to enable Starship is described in the Starship Quick Install[60] instructions.

The link above is the official integration of Starship and Nushell and is the simplest way to get Starship running without doing anything manual:

- Starship will create its own configuration / environment setup script
- you simply have to create it in `env.nu` and `use` it in `config.nu` :::

Here's an example config section for Starship:

```
$env.STARSHIP_SHELL = "nu" def create_left_prompt [] {
   starship prompt --cmd-duration $env.CMD_DURATION_MS
$'--status=($env.LAST_EXIT_CODE)' } # Use nushell functions
to define your right and left prompt $env.PROMPT_COMMAND
= { || create_left_prompt } $env.PROMPT_COMMAND_RIGHT =
"" # The prompt indicators are environmental variables
that represent # the state of the prompt $env.PROMPT_INDICATOR
= "" $env.PROMPT_INDICATOR_VI_INSERT = ": " $env.PROMPT_-
INDICATOR_VI_NORMAL = " " $env.PROMPT_MULTILINE_INDICATOR
= "::: "
```

Now restart Nu.

```
nushell on  main is  v0.60.0 via  v1.59.0
```

Purs

repo[61]

Shells in shells

Working in multiple directories

While it's common to work in one directory, it can be handy to work in multiple places at the same time. For this, Nu offers the concept of "shells". As the name implies, they're a way of running multiple shells

[60] https://starship.rs/#nushell
[61] https://github.com/xcambar/purs

in one, allowing you to quickly jump between working directories and more.

To get started, let's enter a directory:

```
/home/jonathan/Source/nushell(main)> enter ../book /home/
jonathan/Source/book(main)> ls
 #    name                 type   size     modified
  0   404.html             File    429 B   2 hours ago
1   CONTRIBUTING.md      File    955 B   2 hours ago   2
 Gemfile              File   1.1 KB   2 hours ago   3
Gemfile.lock         File   6.9 KB   2 hours ago
```

Entering is similar to changing directories (as we saw with the cd[62] command). This allows you to jump into a directory to work in it. Instead of changing the directory, we now are in two directories. To see this more clearly, we can use the shells[63] command to list the current directories we have active:

```
/home/jonathan/Source/book(main)> enter ../music /home/
jonathan/Source/music(main)> shells
 #   active                path
0   false    /home/jonathan/Source/nushell  1   false
/home/jonathan/Source/book  2   true     /home/jonathan/
Source/music
```

The shells[64] command shows us there are three shells: our original "nushell" source directory, "book" directory and "music" directory which is currently active.

We can jump between these shells with the n[65], p[66] and g[67] shortcuts, short for "next", "previous" and "goto":

```
/home/jonathan/Source/music(main)> p /home/jonathan/Source/
book(main)> n /home/jonathan/Source/music(main)> g 0 /home/
jonathan/Source/nushell(main)>
```

We can see the directory changing, but we're always able to get back to a previous directory we were working on. This allows us to work in multiple directories in the same session.

[62] /commands/docs/cd.md
[63] /commands/docs/shells.md
[64] /commands/docs/shells.md
[65] /commands/docs/n.md
[66] /commands/docs/p.md
[67] /commands/docs/g.md

Exiting the shell

You can leave a shell you have `enter`[68]ed using the `dexit` command.

You can always quit Nu, even if multiple shells are active, using `exit`.

Reedline, Nu's line editor

Nushell's line editor Reedline[69] is a cross-platform line reader designed to be modular and flexible. The engine is in charge of controlling the command history, validations, completions, hints and screen paint.

Configuration

Editing mode

Reedline allows you to edit text using two modes: vi and emacs. If not specified, the default edit mode is emacs mode. In order to select your favorite you need to modify your config file and write down your preferred mode.

For example:

```
 $env.config = {     ...     edit_mode: emacs     ...
}
```

Default keybindings Each edit mode comes with the usual keybinding for vi and emacs text editing.

Emacs and Vi Insert keybindings

[68] /commands/docs/enter.md
[69] https://github.com/nushell/reedline

Key	Event
Esc	Esc
Backspace	Backspace
End	Move to end of line
End	Complete history hint
Home	Move to line start
Ctr + c	Cancel current line
Ctr + l	Clear screen
Ctr + r	Search history
Ctr + Right	Complete history word
Ctr + Right	Move word right
Ctr + Left	Move word left
Up	Move menu up
Up	Move up
Down	Move menu down
Down	Move down
Left	Move menu left
Left	Move left
Right	History hint complete
Right	Move menu right
Right	Move right
Ctr + b	Move menu left
Ctr + b	Move left
Ctr + f	History hint complete
Ctr + f	Move menu right
Ctr + f	Move right
Ctr + p	Move menu up
Ctr + p	Move up
Ctr + n	Move menu down
Ctr + n	Move down

Vi Normal keybindings

Key	Event
Ctr + c	Cancel current line
Ctr + l	Clear screen
Up	Move menu up
Up	Move up
Down	Move menu down
Down	Move down
Left	Move menu left
Left	Move left
Right	Move menu right
Right	Move right

Besides the previous keybindings, while in Vi normal mode you can use the classic vi mode of executing actions by selecting a motion or an action. The available options for the combinations are:

Vi Normal motions

Key	motion
w	Word
d	Line end
0	Line start
$	Line end
f	Right until char
t	Right before char
F	Left until char
T	Left before char

Vi Normal actions

Key	action
d	Delete
p	Paste after
P	Paste before
h	Move left
l	Move right
j	Move down
k	Move up
w	Move word right
b	Move word left
i	Enter Vi insert at current char
a	Enter Vi insert after char
0	Move to start of line
^	Move to start of line
$	Move to end of line
u	Undo
c	Change
x	Delete char
s	History search
D	Delete to end
A	Append to end

Command history

As mentioned before, Reedline manages and stores all the commands that are edited and sent to Nushell. To configure the max number of records that Reedline should store you will need to adjust this value in your config file:

```
$env.config = {     ...     history: {       ...
max_size: 1000       ...     }    ...   }
```

Customizing your prompt

Reedline prompt is also highly customizable. In order to construct your perfect prompt, you could define the next environment variables in your config file:

```
# Use nushell functions to define your right and left prompt
def create_left_prompt [] {    let path_segment = ($env.
PWD)    $path_segment } def create_right_prompt [] {
  let time_segment = ([       (date now | format date
'%m/%d/%Y %r')     ] | str join)    $time_segment } $env.
PROMPT_COMMAND = { create_left_prompt } $env.PROMPT_COMMAND_-
RIGHT = { create_right_prompt }
```

::: tip You don't have to define the environment variables using Nushell functions. You can use simple strings to define them. :::

You can also customize the prompt indicator for the line editor by modifying the next env variables.

```
$env.PROMPT_INDICATOR = " " $env.PROMPT_INDICATOR_VI_INSERT
= ": " $env.PROMPT_INDICATOR_VI_NORMAL = " " $env.PROMPT_-
MULTILINE_INDICATOR = "::: "
```

::: tip The prompt indicators are environment variables that represent the state of the prompt :::

Keybindings

Reedline keybindings are powerful constructs that let you build chains of events that can be triggered with a specific combination of keys.

For example, let's say that you would like to map the completion menu to the `Ctrl + t` keybinding (default is `tab`). You can add the next entry to your config file.

```
  $env.config = {     ...     keybindings: [       {
     name: completion_menu         modifier: control
      keycode: char_t        mode: emacs         event:
 { send: menu name: completion_menu }      }    ]
...   }
```

After loading this new `config.nu`, your new keybinding (`Ctrl + t`) will open the completion command.

Each keybinding requires the next elements:

- name: Unique name for your keybinding for easy reference in `$config.keybindings`
- modifier: A key modifier for the keybinding. The options are:
  - none
  - control
  - alt
  - shift

    - shift_alt
    - alt_shift
    - control_alt
    - alt_control
    - control_shift
    - shift_control
    - control_alt_shift
    - control_shift_alt

- keycode: This represent the key to be pressed
- mode: emacs, vi_insert, vi_normal (a single string or a list. e.g. [`vi_insert vi_normal`])
- event: The type of event that is going to be sent by the keybinding. The options are:
    - send
    - edit
    - until

::: tip All of the available modifiers, keycodes and events can be foun

:: tip The keybindings added to `vi_insert` mode will be available when the line editor is in insert mode (when you can write text), and the keybindings marked with `vi_normal` mode will be available when in normal (when the cursor moves using h, j, k or l) :::

The event section of the keybinding entry is where the actions to be performed are defined. In this field you can use either a record or a list of records. Something like this

```
  ...   event: { send: Enter }  ...
```

or

```
  ...   event: [    { edit: Clear }    { send: Enter
}  ]  ...
```

The first keybinding example shown in this page follows the first case; a single event is sent to the engine.

The next keybinding is an example of a series of events sent to the engine. It first clears the prompt, inserts a string and then enters that value

```
   $env.config = {     ...     keybindings: [    {
  name: change_dir_with_fzf       modifier: CONTROL
   keycode: Char_t       mode: emacs       event:[
     { edit: Clear }           { edit: InsertString,
          value: "cd (ls | where type == dir | each { |it|
$it.name} | str join (char nl) | fzf | decode utf-8 | str
trim)"          }          { send: Enter }         ]
    }    ...   }
```

One disadvantage of the previous keybinding is the fact that the inserted text will be processed by the validator and saved in the history, making the keybinding a bit slow and populating the command history with the same command. For that reason there is the `executehost-command` type of event. The next example does the same as the previous one in a simpler way, sending a single event to the engine

```
   $env.config = {     ...      keybindings: [     {
    name: change_dir_with_fzf       modifier: CONTROL
     keycode: Char_y       mode: emacs       event: {
       send: executehostcommand,         cmd: "cd (ls |
where type == dir | each { |it| $it.name} | str join (char
nl) | fzf | decode utf-8 | str trim)"       }     }   ]
     ...   }
```

Before we continue you must have noticed that the syntax changes for edits and sends, and for that reason it is important to explain them a bit more. A `send` is all the `Reedline` events that can be processed by the engine and an `edit` are all the `EditCommands` that can be processed by the engine.

Send type

To find all the available options for `send` you can use

```
keybindings list | where type == events
```

And the syntax for `send` events is the next one

```
     ...       event: { send: <NAME OF EVENT FROM LIST>
}     ...
```

::: tip You can write the name of the events with capital letters. The keybinding parser is case insensitive :::

There are two exceptions to this rule: the `Menu` and `ExecuteHostCommand`. Those two events require an extra field to be complete. The `Menu` needs the name of the menu to be activated (completion_menu or history_menu)

```
    ...       event: {        send: menu        name:
  completion_menu      }    ...
```

and the `ExecuteHostCommand` requires a valid command that will be sent to the engine

```
    ...       event: {        send: executehostcommand
        cmd: "cd ~"      }    ...
```

It is worth mentioning that in the events list you will also see `Edit([])` , `Multiple([])` and `UntilFound([])`. These options are not available for the parser since they are constructed based on the keybinding definition. For example, a `Multiple([])` event is built for you when defining a list of records in the keybinding's event. An `Edit([])` event is the same as the `edit` type that was mentioned. And the `UntilFound([])` event is the same as the `until` type mentioned before.

Edit type

The `edit` type is the simplification of the `Edit([])` event. The `event` type simplifies defining complex editing events for the keybindings. To list the available options you can use the next command

```
keybindings list | where type == edits
```

The usual syntax for an `edit` is the next one

```
    ...       event: { edit: <NAME OF EDIT FROM LIST>
} ...
```

The syntax for the edits in the list that have a `()` changes a little bit. Since those edits require an extra value to be fully defined. For example, if we would like to insert a string where the prompt is located, then you will have to use

```
    ...       event: {        edit: insertstring
     value: "MY NEW STRING"      }    ...
```

or say you want to move right until the first `S`

```
     ...       event: {        edit: moverightuntil
      value: "S"       }     ...
```

As you can see, these two types will allow you to construct any type of keybinding that you require

Until type

To complete this keybinding tour we need to discuss the `until` type for event. As you have seen so far, you can send a single event or a list of events. And as we have seen, when a list of events is sent, each and every one of them is processed.

However, there may be cases when you want to assign different events to the same keybinding. This is especially useful with Nushell menus. For example, say you still want to activate your completion menu with `Ctrl + t` but you also want to move to the next element in the menu once it is activated using the same keybinding.

For these cases, we have the `until` keyword. The events listed inside the until event will be processed one by one with the difference that as soon as one is successful, the event processing is stopped.

The next keybinding represents this case.

```
    $env.config = {     ...     keybindings: [       {
       name: completion_menu         modifier: control
        keycode: char_t        mode: emacs        event:
   {          until: [            { send: menu name: completion_-
  menu }             { send: menunext }          ]
    }      }    ]     ...   }
```

The previous keybinding will first try to open a completion menu. If the menu is not active, it will activate it and send a success signal. If the keybinding is pressed again, since there is an active menu, then the next event it will send is MenuNext, which means that it will move the selector to the next element in the menu.

As you can see the `until` keyword allows us to define two events for the same keybinding. At the moment of this writing, only the Menu events allow this type of layering. The other non menu event types will always return a success value, meaning that the `until` event will stop as soon as it reaches the command.

For example, the next keybinding will always send a `down` because that event is always successful

```
  $env.config = {     ...     keybindings: [       {
     name: completion_menu         modifier: control
      keycode: char_t         mode: emacs          event:
 {          until: [              { send: down }
      { send: menu name: completion_menu }
{ send: menunext }            ]         }       }     ]
   ...   }
```

Removing a default keybinding

If you want to remove a certain default keybinding without replacing it with a different action, you can set `event: null`.

e.g. to disable screen clearing with `Ctrl + l` for all edit modes

```
  $env.config = {     ...     keybindings: [       {
     modifier: control          keycode: char_l
mode: [emacs, vi_normal, vi_insert]        event: null
      }    ]     ...   }
```

Troubleshooting keybinding problems

Your terminal environment may not always propagate your key combinations on to nushell the way you expect it to. You can use the command `keybindings listen`[71] to figure out if certain keypresses are actually received by nushell, and how.

Menus

Thanks to Reedline, Nushell has menus that can help you with your day to day shell scripting. Next we present the default menus that are always available when using Nushell

Help menu

The help menu is there to ease your transition into Nushell. Say you are putting together an amazing pipeline and then you forgot the internal command that would reverse a string for you. Instead of deleting your pipe, you can activate the help menu with `F1`. Once active just type keywords for the command you are looking for and the menu will show you commands that match your input. The matching is done on the name of the commands or the commands description.

[71] `/commands/docs/keybindings_listen.md`

To navigate the menu you can select the next element by using `tab`, you can scroll the description by pressing left or right and you can even paste into the line the available command examples.

The help menu can be configured by modifying the next parameters

```
  $env.config = {     ...     menus = [       ...
{        name: help_menu        only_buffer_difference:
 true # Search is done on the text written after activating
the menu        marker: "? "                # Indicator
that appears with the menu is active        type: {
          layout: description      # Type of menu
       columns: 4              # Number of columns where
the options are displayed            col_width: 20
        # Optional value. If missing all the screen width
is used to calculate column width           col_padding:
 2          # Padding between columns           selection_-
rows: 4        # Number of rows allowed to display found
options            description_rows: 10     # Number of
rows allowed to display command description        }
       style: {            text: green
   # Text style            selected_text: green_reverse
 # Text style for selected option           description_-
text: yellow      # Text style for description
}       }       ...     ]     ...
```

Completion menu

The completion menu is a context sensitive menu that will present suggestions based on the status of the prompt. These suggestions can range from path suggestions to command alternatives. While writing a command, you can activate the menu to see available flags for an internal command. Also, if you have defined your custom completions for external commands, these will appear in the menu as well.

The completion menu by default is accessed by pressing `tab` and it can be configured by modifying these values from the config object:

```
  $env.config = {     ...     menus: [       ...
  {         name: completion_menu        only_buffer_-
difference: false # Search is done on the text written
after activating the menu        marker: "| "
        # Indicator that appears with the menu is active
        type: {            layout: columnar         #
Type of menu            columns: 4              # Number
of columns where the options are displayed
col_width: 20            # Optional value. If missing
all the screen width is used to calculate column width
           col_padding: 2           # Padding between
columns        }        style: {            text: green
                  # Text style           selected_text:
 green_reverse  # Text style for selected option
      description_text: yellow      # Text style for description
        }      }      ...    ]     ...
```

By modifying these parameters you can customize the layout of your menu to your liking.

History menu

The history menu is a handy way to access the editor history. When activating the menu (default `Ctrl+r`) the command history is presented in reverse chronological order, making it extremely easy to select a previous command.

The history menu can be configured by modifying these values from the config object:

```
  $env.config = {     ...     menus = [       ...
{         name: history_menu        only_buffer_difference:
 true # Search is done on the text written after activating
the menu        marker: "? "                # Indicator
that appears with the menu is active        type: {
          layout: list             # Type of menu
       page_size: 10           # Number of entries that
will presented when activating the menu         }
   style: {            text: green                  #
Text style             selected_text: green_reverse  #
Text style for selected option            description_-
text: yellow      # Text style for description
}       }       ...     ]     ...
```

When the history menu is activated, it pulls `page_size` records from the history and presents them in the menu. If there is space in the terminal, when you press `Ctrl+x` again the menu will pull the same number of records and append them to the current page. If it isn't possible to present all the pulled records, the menu will create a new page. The pages can be navigated by pressing `Ctrl+z` to go to previous page or `Ctrl+x` to go to next page.

Searching the history To search in your history you can start typing key words for the command you are looking for. Once the menu is activated, anything that you type will be replaced by the selected command from your history. for example, say that you have already typed this

```
let a = ()
```

you can place the cursor inside the `()` and activate the menu. You can filter the history by typing key words and as soon as you select an entry, the typed words will be replaced

```
let a = (ls | where size > 10MiB)
```

Menu quick selection Another nice feature of the menu is the ability to quick select something from it. Say you have activated your menu and it looks like this

```
> 0: ls | where size > 10MiB 1: ls | where size > 20MiB
2: ls | where size > 30MiB 3: ls | where size > 40MiB
```

Instead of pressing down to select the fourth entry, you can type `!3` and press enter. This will insert the selected text in the prompt position, saving you time scrolling down the menu.

History search and quick selection can be used together. You can activate the menu, do a quick search, and then quick select using the quick selection character.

User defined menus

In case you find that the default menus are not enough for you and you have the need to create your own menu, Nushell can help you with that.

In order to add a new menu that fulfills your needs, you can use one of the default layouts as a template. The templates available in nushell are columnar, list or description.

The columnar menu will show you data in a columnar fashion adjusting the column number based on the size of the text displayed in your columns.

The list type of menu will always display suggestions as a list, giving you the option to select values using `!` plus number combination.

The description type will give you more space to display a description for some values, together with extra information that could be inserted into the buffer.

Let's say we want to create a menu that displays all the variables created during your session, we are going to call it `vars_menu`. This menu will use a list layout (layout: list). To search for values, we want to use only the things that are written after the menu has been activated (only_buffer_difference: true).

With that in mind, the desired menu would look like this

```
  $env.config = {     ...     menus = [        ...
{          name: vars_menu        only_buffer_difference:
 true          marker: "# "         type: {
layout: list               page_size: 10        }
 style: {            text: green            selected_-
text: green_reverse             description_text: yellow
        }         source: { |buffer, position|
    $nu.scope.vars             | where name =~ $buffer
           | sort-by name             | each { |it| {value:
 $it.name description: $it.type} }        }      }
   ...     ]     ...
```

As you can see, the new menu is identical to the `history_menu` previously described. The only huge difference is the new field called `source`[72]. The `source`[73] field is a nushell definition of the values you want to display in the menu. For this menu we are extracting the data from `$nu.scope.vars` and we are using it to create records that will be used to populate the menu.

The required structure for the record is the next one

```
{   value:       # The value that will be inserted in the
buffer   description: # Optional. Description that will
be display with the selected value   span: {      # Optional.
 Span indicating what section of the string will be replaced
by the value     start:     end:    }   extra: [string]
# Optional. A list of strings that will be displayed with
```

[72] /commands/docs/source.md
[73] /commands/docs/source.md

```
the selected value. Only works with a description menu}
```

For the menu to display something, at least the `value` field has to be present in the resulting record.

In order to make the menu interactive, these two variables are available in the block: `$buffer` and `$position`. The `$buffer` contains the value captured by the menu, when the option `only_buffer_difference` is true, `$buffer` is the text written after the menu was activated. If `only_buffer_difference` is false, `$buffer` is all the string in line. The `$position` variable can be used to create replacement spans based on the idea you had for your menu. The value of `$position` changes based on whether `only_buffer_difference` is true or false. When true, `$position` is the starting position in the string where text was inserted after the menu was activated. When the value is false, `$position` indicates the actual cursor position.

Using this information, you can design your menu to present the information you require and to replace that value in the location you need it. The only thing extra that you need to play with your menu is to define a keybinding that will activate your brand new menu.

Menu keybindings

In case you want to change the default way both menus are activated, you can change that by defining new keybindings. For example, the next two keybindings assign the completion and history menu to `Ctrl+t` and `Ctrl+y` respectively

```
  $env.config = {    ...    keybindings: [      {
     name: completion_menu        modifier: control
      keycode: char_t       mode: [vi_insert vi_normal]
        event: {          until: [            { send:
 menu name: completion_menu }           { send: menupagenext
}          ]        }      }      {        name: history_-
menu        modifier: control        keycode: char_y
       mode: [vi_insert vi_normal]        event: {
       until: [            { send: menu name: history_-
menu }            { send: menupagenext }          ]
      }       }    ]     ...   }
```

Externs

Calling external commands is a fundamental part of using Nushell as a shell (and often using Nushell as a language). There's a problem, though: Nushell can't help with finding errors in the call, completions, or syntax highlighting with external commands.

This is where `extern` comes in. The `extern` keyword allows you to write a full signature for the command that lives outside of Nushell so that you get all the benefits above. If you take a look at the default config, you'll notice that there are a few extern calls in there. Here's one of them:

```
  export extern "git push" [    remote?: string@"nu-complete
git remotes",  # the name of the remote    refspec?: string@"nu-
complete git branches" # the branch / refspec    --verbose(
-v)                              # be more verbose
 --quiet(-q)                                # be more
quiet    --repo: string                               #
repository    --all
    # push all refs    --mirror
             # mirror all refs    --delete(-d)
                        # delete refs    --tags
                               # push tags (can't be used
with --all or --mirror)    --dry-run(-n)
                 # dry run    --porcelain
                    # machine-readable output    --force(
-f)                                # force updates
 --force-with-lease: string                 # require
old value of ref to be at this value    --recurse-submodules:
 string               # control recursive pushing of submodules
    --thin                                     # use thin
pack    --receive-pack: string                       #
receive pack program    --exec: string
              # receive pack program    --set-upstream(
-u)                            # set upstream for git pull/
status    --progress
# force progress reporting    --prune
                    # prune locally removed refs    --
no-verify                                # bypass pre-
push hook    --follow-tags
   # push missing but relevant tags    --signed: string
```

```
                             # GPG sign the push    --atomic
                                     # request atomic transaction
on remote side    --push-option(-o): string
        # option to transmit    --ipv4(-4)
                          # use IPv4 addresses only    --ipv6(
-6)                                  # use IPv6 addresses
only   ]
```

You'll notice this gives you all the same descriptive syntax that internal co

:: warning Note A Nushell comment that continues on the same line for argument documentation purposes requires a space before the `#` pound sign. :::

Types and custom completions

In the above example, you'll notice some types are followed by `@` followed by the name of a command. We talk more about custom completions in their own section.

Both the type (or shape) of the argument and the custom completion tell Nushell about how to complete values for that flag or position. For example, setting a shape to `path` allows Nushell to complete the value to a filepath for you. Using the `@` with a custom completion overrides this default behavior, letting the custom completion give you full completion list.

Format specifiers

Positional parameters can be made optional with a `?` (as seen above) the remaining parameters can be matched with `...` before the parameter name, which will return a list of arguments.

```
  export extern "git add" [     ...pathspecs: glob
# …   ]
```

Limitations

There are a few limitations to the current `extern` syntax. In Nushell, flags and positional arguments are very flexible: flags can precede positional arguments, flags can be mixed into positional arguments, and flags can follow positional arguments. Many external commands are not this flexible. There is not yet a way to require a particular ordering of flags and positional arguments to the style required by the external.

The second limitation is that some externals require flags to be passed using `=` to separate the flag and the value. In Nushell, the `=` is a convenient optional syntax and there's currently no way to require its use.

Custom completions

Custom completions allow you to mix together two features of Nushell: custom commands and completions. With them, you're able to create commands that handle the completions for positional parameters and flag parameters. These custom completions work both for custom commands and known external, or `extern`, commands.

There are two parts to a custom command: the command that handles a completion and attaching this command to the type of another command using `@`.

Example custom completion

Let's look at an example:

```
> def animals [] { ["cat", "dog", "eel" ] }> def my-command
[animal: string@animals] { print $animal }>| my-command
cat                 dog                 eel
```

In the first line, we create a custom command that will return a list of three different animals. These are the values we'd like to use in the completion. Once we've created this command, we can now use it to provide completions for other custom commands and `extern`[74]s.

In the second line, we use `string@animals`. This tells Nushell two things: the shape of the argument for type-checking and the custom completion to use if the user wants to complete values at that position.

On the third line, we type the name of our custom command `my-command` followed by hitting space and then the `<tab>` key. This brings up our completions. Custom completions work the same as other completions in the system, allowing you to type `e` followed by the `<tab>` key and get "eel" automatically completed.

Modules and custom completions

You may prefer to keep your custom completions away from the public API for your code. For this, you can combine modules and custom

[74] /commands/docs/extern.md

completions.

Let's take the example above and put it into a module:

```
module commands {     def animals [] {         ["cat",
"dog", "eel" ]     }     export def my-command [animal:
 string@animals] {         print $animal     } }
```

In our module, we've chosen to export only the custom command `my-command` but not the custom completion `animals`. This allows users of this module to call the command, and even use the custom completion logic, without having access to the custom completion. This keeps the API cleaner, while still offering all the same benefits.

This is possible because custom completion tags using `@` are locked-in as the command is first parsed.

Context aware custom completions

It is possible to pass the context to the custom completion command. This is useful in situations where it is necessary to know previous arguments or flags to generate accurate completions.

Let's apply this concept to the previous example:

```
module commands {     def animals [] {         ["cat",
"dog", "eel" ]     }     def animal-names [context: string]
{         {             cat: ["Missy", "Phoebe"]
     dog: ["Lulu", "Enzo"]            eel: ["Eww", "Slippy"]
        } | get -i ($context | split words | last)
}    export def my-command [        animal: string@animals
        name: string@animal-names    ] {        print
$"The ($animal) is named ($name)."    } }
```

Here, the command `animal-names` returns the appropriate list of names. This is because `$context` is a string with where the value is the command that has been typed until now.

```
>| my-command cat                  dog                  eel
>| my-command dog Lulu                  Enzo >my-command
dog enzo The dog is named Enzo
```

On the second line, once we press the `<tab>` key, the argument `"my-command dog"` is passed to the `animal-names` command as context.

Custom completion and `extern`[75]

A powerful combination is adding custom completions to known `extern` commands. These work the same way as adding a custom completion to a custom command: by creating the custom completion and then attaching it with a `@` to the type of one of the positional or flag arguments of the `extern`.

If you look closely at the examples in the default config, you'll see this:

```
export extern "git push" [     remote?: string@"nu-complete
git remotes",  # the name of the remote     refspec?: string@"nu-
complete git branches" # the branch / refspec      ... ]
```

Custom completions will serve the same role in this example as in the previous examples. The examples above call into two different custom completions, based on the position the user is currently in.

Custom descriptions

As an alternative to returning a list of strings, a completion function can also return a list of records with a `value` and `description` field.

```
def my_commits [] {     [          { value: "5c2464", description:
 "Add .gitignore" },          { value: "f3a377", description:
 "Initial commit" }     ] }
```

Note
with the following snippet

```
def my-command [commit: string@my_commits] {
   print $commit }
```

be aware that, even though the completion menu will show you something like

```
>_ my-command <TAB> 5c2464  Add .gitignore f3a377
 Initial commit
```

only the value, i.e. "5c2464" or "f3a377", will be used in the command arguments!

[75] `/commands/docs/extern.md`

External completions

External completers can also be integrated, instead of relying solely on Nushell ones.

For this, set the `external_completer` field in `config.nu` to a closure which will be evaluated if no Nushell completions were found.

```
> $env.config.completions.external = { >     enable: true
>     max_results: 100 >     completer: $completer > }
```

You can configure the closure to run an external completer, such as carapace[76].

When the closure returns unparsable json (e.g. an empty string) it defaults to file completion.

An external completer is a function that takes the current command as a string list, and outputs a list of records with `value` and `description` keys, like custom completion functions.

> ***Note*** *This closure will accept the current command as a list. For example, typing* `my-command --arg1 <tab>` *will receive* `[my-command --arg1 " "]`*.*

This example will enable carapace external completions:

```
let carapace_completer = {|spans|     carapace $spans.0
nushell ...$spans | from json }
```

More examples of custom completers can be found in the cookbook[77].

Coloring and Theming in Nu

Many parts of Nushell's interface can have their color customized. All of these can be set in the `config.nu` configuration file. If you see the hash/hashtag/pound mark `#` in the config file it means the text after it is commented out.

1. table borders
2. primitive values
3. shapes (this is the command line syntax)

[76] https://github.com/rsteube/carapace-bin
[77] ../cookbook/external_completers.md

4. prompt
5. LS_COLORS

Table borders

Table borders are controlled by the `$env.config.table.mode` setting in `config.nu`. Here is an example:

```
> $env.config = {    table: {         mode: rounded
 } }
```

Here are the current options for `$env.config.table.mode`:

- `rounded` # of course, this is the best one :)
- `basic`
- `compact`
- `compact_double`
- `light`
- `thin`
- `with_love`
- `reinforced`
- `heavy`
- `none`
- `other`

Color symbologies

***

- `r` - normal color red's abbreviation
- `rb` - normal color red's abbreviation with bold attribute
- `red` - normal color red

- `red_bold` - normal color red with bold attribute

- `"#ff0000"` - "#hex" format foreground color red (quotes are required)

- `{ fg: "#ff0000" bg: "#0000ff" attr: b }` - "full #hex" format foreground red in "#hex" format with a background of blue in "#hex" format with an attribute of bold abbreviated.

`attributes`

***

code	meaning
l	blink
b	bold
d	dimmed
h	hidden
i	italic
r	reverse
s	strikethrough
u	underline
n	nothing
	defaults to nothing

normal colors and abbreviations

code	name
g	green
gb	green_bold
gu	green_underline
gi	green_italic
gd	green_dimmed
gr	green_reverse
gbl	green_blink
gst	green_strike
lg	light_green
lgb	light_green_bold
lgu	light_green_underline
lgi	light_green_italic
lgd	light_green_dimmed
lgr	light_green_reverse
lgbl	light_green_blink
lgst	light_green_strike
r	red
rb	red_bold
ru	red_underline
ri	red_italic
rd	red_dimmed
rr	red_reverse
rbl	red_blink
rst	red_strike
lr	light_red
lrb	light_red_bold
lru	light_red_underline
lri	light_red_italic
lrd	light_red_dimmed
lrr	light_red_reverse
lrbl	light_red_blink
lrst	light_red_strike
u	blue
ub	blue_bold
uu	blue_underline
ui	blue_italic
ud	blue_dimmed
ur	blue_reverse
ubl	blue_blink
ust	blue_strike
lu	light_blue
lub	light_blue_bold
luu	light_blue_underline

`"#hex"` format

***

The "#hex" format is one way you typically see colors represented. It's simply the `#` character followed by 6 characters. The first two are for `red`, the second two are for `green`, and the third two are for `blue`. It's important that this string be surrounded in quotes, otherwise Nushell thinks it's a commented out string.

Example: The primary `red` color is `"#ff0000"` or `"#FF0000"`. Upper and lower case in letters shouldn't make a difference.

This `"#hex"` format allows us to specify 24-bit truecolor tones to different parts of Nushell.

`full "#hex"` format

***

The `full "#hex"` format is a take on the `"#hex"` format but allows one to specify the foreground, background, and attributes in one line.

Example: `{ fg: "#ff0000" bg: "#0000ff" attr: b }`

- foreground of red in "#hex" format
- background of blue in "#hex" format
- attribute of bold abbreviated

`Primitive values`

***

Primitive values are things like `int` and `string`. Primitive values and shapes can be set with a variety of color symbologies seen above.

This is the current list of primitives. Not all of these are configurable. The configurable ones are marked with *.

primitive	default color	configurable
`any`		
`binary`	Color::White.normal()	*
`block`	Color::White.normal()	*
`bool`	Color::White.normal()	*
`cellpath`	Color::White.normal()	*
`condition`		
`custom`		
`date`	Color::White.normal()	*
`duration`	Color::White.normal()	*
`expression`		
`filesize`	Color::White.normal()	*
`float`	Color::White.normal()	*
`glob`		
`import`		
`int`	Color::White.normal()	*
`list`	Color::White.normal()	*
`nothing`	Color::White.normal()	*
`number`		
`operator`		
`path`		
`range`	Color::White.normal()	*
`record`	Color::White.normal()	*
`signature`		
`string`	Color::White.normal()	*
`table`		
`var`		
`vardecl`		
`variable`		

special "primitives" (not really primitives but they exist solely for coloring)

primitive	default color	configurable
`leading_trailing_space_bg`	Color::Rgb(128, 128, 128))	*
`header`	Color::Green.bold()	*
`empty`	Color::Blue.normal()	*
`row_index`	Color::Green.bold()	*
`hints`	Color::DarkGray.normal()	*

Here's a small example of changing some of these values.

```
> let config = {     color_config: {         separator:
 purple         leading_trailing_space_bg: "#ffffff"
      header: gb        date: wd        filesize: c
      row_index: cb        bool: red        int: green
        duration: blue_bold        range: purple
   float: red        string: white        nothing: red
        binary: red        cellpath: cyan        hints:
 dark_gray    } }
```

Here's another small example using multiple color syntaxes with some comments.

```
> let config = {     color_config: {         separator:
 "#88b719" # this sets only the foreground color like PR
#486         leading_trailing_space_bg: white # this sets
only the foreground color in the original style
 header: { # this is like PR #489            fg: "#B01455",
# note, quotes are required on the values with hex colors
            bg: "#ffb900", # note, commas are not required,
it could also be all on one line            attr: bli
# note, there are no quotes around this value. it works
with or without quotes        }        date: "#75507B"
        filesize: "#729fcf"         row_index: {
      # note, that this is another way to set only the
foreground, no need to specify bg and attr
fg: "#e50914"        }    } }
```

Shape values

As mentioned above, `shape` is a term used to indicate the syntax coloring.

Here's the current list of flat shapes.

shape	default style	configurable
`shape_block`	fg(Color::Blue).bold()	*
`shape_bool`	fg(Color::LightCyan)	*
`shape_custom`	bold()	*
`shape_external`	fg(Color::Cyan)	*
`shape_externalarg`	fg(Color::Green).bold()	*
`shape_filepath`	fg(Color::Cyan)	*
`shape_flag`	fg(Color::Blue).bold()	*
`shape_float`	fg(Color::Purple).bold()	*
`shape_garbage`	fg(Color::White).on(Color::Red).bold()	*
`shape_globpattern`	fg(Color::Cyan).bold()	*
`shape_int`	fg(Color::Purple).bold()	*
`shape_-internalcall`	fg(Color::Cyan).bold()	*
`shape_list`	fg(Color::Cyan).bold()	*
`shape_literal`	fg(Color::Blue)	*
`shape_nothing`	fg(Color::LightCyan)	*
`shape_operator`	fg(Color::Yellow)	*
`shape_range`	fg(Color::Yellow).bold()	*
`shape_record`	fg(Color::Cyan).bold()	*
`shape_signature`	fg(Color::Green).bold()	*
`shape_string`	fg(Color::Green)	*
`shape_string_interpolation`	fg(Color::Cyan).bold()	*
`shape_table`	fg(Color::Blue).bold()	*
`shape_variable`	fg(Color::Purple)	*

Here's a small example of how to apply color to these items. Anything not specified will receive the default color.

```
> $env.config = {     color_config: {         shape_garbage:
 { fg: "#FFFFFF" bg: "#FF0000" attr: b}         shape_-
bool: green         shape_int: { fg: "#0000ff" attr: b}
    }}
```

Prompt configuration and coloring

The Nushell prompt is configurable through these environment variables and config items:

- `PROMPT_COMMAND`: Code to execute for setting up the prompt (block)

- PROMPT_COMMAND_RIGHT: Code to execute for setting up the *RIGHT* prompt (block) (see oh-my.nu in nu_scripts)
- PROMPT_INDICATOR = " ": The indicator printed after the prompt (by default ">"-like Unicode symbol)
- PROMPT_INDICATOR_VI_INSERT = ": "
- PROMPT_INDICATOR_VI_NORMAL = "v "
- PROMPT_MULTILINE_INDICATOR = "::: "
- **render_right_prompt_on_last_line**: Bool value to enable or disable the right prompt to be rendered on the last line of the prompt

Example: For a simple prompt one could do this. Note that PROMPT_COMMAND requires a **block** whereas the others require a **string**.

```
> $env.PROMPT_COMMAND = { build-string (date now | format
date '%m/%d/%Y %I:%M:%S%.3f') ': ' (pwd | path basename)
 }
```

If you don't like the default PROMPT_INDICATOR you could change it like this.

```
> $env.PROMPT_INDICATOR = "> "
```

If you're using **starship**, you'll most likely want to show the right prompt on the last line of the prompt, just like zsh or fish. You could modify the **config.nu** file, just set **render_right_prompt_on_last_line** to true:

```
config {     render_right_prompt_on_last_line = true
  ... }
```

Coloring of the prompt is controlled by the **block** in PROMPT_COMMAND where you can write your own custom prompt. We've written a slightly fancy one that has git statuses located in the nu_scripts repo[78].

[78] https://github.com/nushell/nu_scripts/blob/main/modules/prompt/oh-my.nu

Transient prompt

If you want a different prompt displayed for previously entered commands, you can use Nushell's transient prompt feature. This can be useful if your prompt has lots of information that is unnecessary to show for previous lines (e.g. time and Git status), since you can make it so that previous lines show with a shorter prompt.

Each of the `PROMPT_*` variables has a corresponding `TRANSIENT_PROMPT_*` variable to be used for changing that segment when displaying past prompts: `TRANSIENT_PROMPT_COMMAND`, `TRANSIENT_PROMPT_COMMAND_RIGHT`, `TRANSIENT_PROMPT_INDICATOR`, `TRANSIENT_PROMPT_INDICATOR_VI_INSERT`, `TRANSIENT_PROMPT_INDICATOR_VI_NORMAL`, `TRANSIENT_PROMPT_MULTILINE_INDICATOR`. By default, the `PROMPT_*` variables are used for displaying past prompts.

For example, if you want to make past prompts show up without a left prompt entirely and leave only the indicator, you can use:

```
> $env.TRANSIENT_PROMPT_COMMAND = ""
```

If you want to go back to the normal left prompt, you'll have to unset `TRANSIENT_PROMPT_COMMAND`:

```
> hide-env TRANSIENT_PROMPT_COMMAND
```

LS_COLORS colors for the ls[79] command

Nushell will respect and use the `LS_COLORS` environment variable setting on Mac, Linux, and Windows. This setting allows you to define the color of file types when you do a `ls`[80]. For instance, you can make directories one color, _.md markdown files another color, _.toml files yet another color, etc. There are a variety of ways to color your file types.

There's an exhaustive list here[81], which is overkill, but gives you an rudimentary understanding of how to create a ls_colors file that `dircolors` can turn into a `LS_COLORS` environment variable.

This[82] is a pretty good introduction to `LS_COLORS`. I'm sure you can find many more tutorials on the web.

I like the `vivid` application and currently have it configured in my `config.nu` like this. You can find `vivid` here[83].

[79] /commands/docs/ls.md
[80] /commands/docs/ls.md
[81] https://github.com/trapd00r/LS_COLORS
[82] https://www.linuxhowto.net/how-to-set-colors-for-ls-command/
[83] https://github.com/sharkdp/vivid

`$env.LS_COLORS = (vivid generate molokai | str trim)`

If `LS_COLORS` is not set, nushell will default to a built-in `LS_COLORS` setting, based on 8-bit (extended) ANSI colors.

Theming

Theming combines all the coloring above. Here's a quick example of one we put together quickly to demonstrate the ability to theme. This is a spin on the `base16` themes that we see so widespread on the web.

The key to making theming work is to make sure you specify all themes and colors you're going to use in the `config.nu` file *before* you declare the `let config =` line.

```
# let's define some colors let base00 = "#181818" # Default
Background let base01 = "#282828" # Lighter Background
(Used for status bars, line number and folding marks) let
base02 = "#383838" # Selection Background let base03 =
"#585858" # Comments, Invisibles, Line Highlighting let
base04 = "#b8b8b8" # Dark Foreground (Used for status bars)
 let base05 = "#d8d8d8" # Default Foreground, Caret, Delimiters,
Operators let base06 = "#e8e8e8" # Light Foreground (Not
often used) let base07 = "#f8f8f8" # Light Background (
Not often used) let base08 = "#ab4642" # Variables, XML
Tags, Markup Link Text, Markup Lists, Diff Deleted let
base09 = "#dc9656" # Integers, Boolean, Constants, XML
Attributes, Markup Link Url let base0a = "#f7ca88" # Classes,
Markup Bold, Search Text Background let base0b = "#a1b56c"
# Strings, Inherited Class, Markup Code, Diff Inserted let
base0c = "#86c1b9" # Support, Regular Expressions, Escape
Characters, Markup Quotes let base0d = "#7cafc2" # Functions,
Methods, Attribute IDs, Headings let base0e = "#ba8baf"
# Keywords, Storage, Selector, Markup Italic, Diff Changed
let base0f = "#a16946" # Deprecated, Opening/Closing Embedded
Language Tags, e.g. <?php ?> # we're creating a theme here
that uses the colors we defined above. let base16_theme
= {     separator: $base03     leading_trailing_space_-
bg: $base04     header: $base0b     date: $base0e     filesize:
 $base0d     row_index: $base0c     bool: $base08     int:
 $base0b     duration: $base08     range: $base08     float:
 $base08     string: $base04     nothing: $base08     binary:
 $base08     cellpath: $base08     hints: dark_gray
# shape_garbage: { fg: $base07 bg: $base08 attr: b} # base16
white on red    # but i like the regular white on red
for parse errors    shape_garbage: { fg: "#FFFFFF" bg:
 "#FF0000" attr: b}    shape_bool: $base0d    shape_-
int: { fg: $base0e attr: b}    shape_float: { fg: $base0e
attr: b}    shape_range: { fg: $base0a attr: b}    shape_-
internalcall: { fg: $base0c attr: b}    shape_external:
 $base0c    shape_externalarg: { fg: $base0b attr: b}
   shape_literal: $base0d    shape_operator: $base0a
   shape_signature: { fg: $base0b attr: b}    shape_string:
 $base0b    shape_filepath: $base0d    shape_globpattern:
 { fg: $base0d attr: b}    shape_variable: $base0e
 shape_flag: { fg: $base0d attr: b}    shape_custom: {attr:
 b} } # now let's apply our regular config settings but
also apply the "color_config:" theme that we specified
above.  let config = {   filesize_metric: true   table_
mode: rounded # basic, compact, compact_double, light,
thin, with_love, rounded, reinforced, heavy, none, other
  use_ls_colors: true  color_config: $base16_theme # <-
- this is the theme  use_grid_icons: true  footer_mode:
 always #always, never, number_of_rows, auto  animate_-
prompt: false  float_precision: 2  use_ansi_coloring:
 true  filesize_format: "b" # b, kb, kib, mb, mib, gb,
gib, tb, tib, pb, pib, eb, eib, auto  edit_mode: emacs
# vi  max_history_size: 10000  log_level: error }
```

if you want to go full-tilt on theming, you'll want to theme all the items I mentioned at the very beginning, including LS_COLORS, and the prompt. Good luck!

Working on light background terminal

Nushell's default config file contains a light theme definition, if you are working on a light background terminal, you can apply the light theme easily.

```
# in $nu.config-path $env.config = {  ...  color_config:
 $dark_theme  # if you want a light theme, replace `$dark_-
theme` to `$light_theme`  ... }
```

You can just change it to light theme by replacing `$dark_theme` to `$light_theme`

```
# in $nu.config-path $env.config = {   ...   color_config:
 $light_theme   # if you want a light theme, replace `$dark_-
theme` to `$light_theme`   ... }
```

Accessibility

It's often desired to have the minimum amount of decorations when using a screen reader. In those cases, it's possible to disable borders and other decorations for both table and errors with the following options:

```
# in $nu.config-path $env.config = {   ...   table: {
 ...      mode: "none"    ...    }   error_style: "plain"
  ... }
```

Line editor menus (completion, history, help…)

Reedline (Nu's line editor) style is not using the `color_config` key. Instead, each menu has its own style to be configured separately. See the section dedicated to Reedline's menus configuration to learn more on this.

Hooks

Hooks allow you to run a code snippet at some predefined situations. They are only available in the interactive mode (REPL[84]), they do not work if you run a Nushell with a script (`nu script.nu`) or commands (`nu -c "print foo"`) arguments.

Currently, we support these types of hooks:

- `pre_prompt` : Triggered before the prompt is drawn
- `pre_execution` : Triggered before the line input starts executing
- `env_change` : Triggered when an environment variable changes
- `display_output` : A block that the output is passed to (experimental).
- `command_not_found` : Triggered when a command is not found.

To make it clearer, we can break down Nushell's execution cycle. The steps to evaluate one line in the REPL mode are as follows:

[84] https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

1. Check for `pre_prompt` hooks and run them
2. Check for `env_change` hooks and run them
3. Display prompt and wait for user input
4. After user typed something and pressed "Enter": Check for `pre_execution` hooks and run them
5. Parse and evaluate user input
6. If a command is not found: Run the `command_not_found` hook. If it returns a string, show it.
7. If `display_output` is defined, use it to print command output
8. Return to 1.

Basic Hooks

To enable hooks, define them in your config:

```
$env.config = {     # ...other config...     hooks: {
       pre_prompt: { print "pre prompt hook" }
pre_execution: { print "pre exec hook" }         env_change:
 {             PWD: {|before, after| print $"changing directory
from ($before) to ($after)" }         }     } }
```

Try putting the above to your config, running Nushell and moving around your filesystem. When you change a directory, the `PWD` environment variable changes and the change triggers the hook with the previous and the current values stored in `before` and `after` variables, respectively.

Instead of defining just a single hook per trigger, it is possible to define a **list of hooks** which will run in sequence:

```
$env.config = {     ...other config...     hooks: {
    pre_prompt: [            { print "pre prompt hook"
}             { print "pre prompt hook2" }         ]
     pre_execution: [             { print "pre exec hook"
}             { print "pre exec hook2" }         ]
   env_change: {            PWD: [                {|before,
after| print $"changing directory from ($before) to ($after)
" }                 {|before, after| print $"changing directory
from ($before) to ($after) 2" }             ]         }
```

```
    } }
```

Also, it might be more practical to update the existing config with new hooks, instead of defining the whole config from scratch:

```
$env.config = ($env.config | upsert hooks {     pre_prompt:
 ...      pre_execution: ...      env_change: {
PWD: ...     } })
```

Changing Environment

One feature of the hooks is that they preserve the environment. Environment variables defined inside the hook **block** will be preserved in a similar way as `def --env`. You can test it with the following example:

```
> $env.config = ($env.config | upsert hooks {     pre_-
prompt: { $env.SPAM = "eggs" } }) > $env.SPAM eggs
```

The hook blocks otherwise follow the general scoping rules, i.e., commands, aliases, etc. defined within the block will be thrown away once the block ends.

Conditional Hooks

One thing you might be tempted to do is to activate an environment whenever you enter a directory:

```
$env.config = ($env.config | upsert hooks {     env_change:
 {         PWD: [               {|before, after|
       if $after == /some/path/to/directory {
           load-env { SPAM: eggs }                 }
          }          ]     } })
```

This won't work because the environment will be active only within the `if`[85] block. In this case, you could easily rewrite it as `load-env ( if $after == ... { ... } else { {} })` but this pattern is fairly common and later we'll see that not all cases can be rewritten like this.

To deal with the above problem, we introduce another way to define a hook -- **a record**:

[85] `/commands/docs/if.md`

```
$env.config = ($env.config | upsert hooks {     env_change:
 {        PWD: [            {                condition:
 {|before, after| $after == /some/path/to/directory }
               code: {|before, after| load-env { SPAM:
eggs } }             }         ]     } })
```

When the hook triggers, it evaluates the `condition` block. If it returns `true`, the `code` block will be evaluated. If it returns `false`, nothing will happen. If it returns something else, an error will be thrown. The `condition` field can also be omitted altogether in which case the hook will always evaluate.

The `pre_prompt` and `pre_execution` hook types also support the conditional hooks but they don't accept the `before` and `after` parameters.

Hooks as Strings

So far a hook was defined as a block that preserves only the environment, but nothing else. To be able to define commands or aliases, it is possible to define the `code` field as **a string**. You can think of it as if you typed the string into the REPL and hit Enter. So, the hook from the previous section can be also written as

```
> $env.config = ($env.config | upsert hooks {     pre_-
prompt: '$env.SPAM = "eggs"' }) > $env.SPAM eggs
```

This feature can be used, for example, to conditionally bring in definitions based on the current directory:

```
$env.config = ($env.config | upsert hooks {     env_change:
 {        PWD: [            {                condition:
 {|_, after| $after == /some/path/to/directory }
          code: 'def foo [] { print "foo" }'
  }             {                condition: {|before,
_| $before == /some/path/to/directory }
 code: 'hide foo'             }         ]     } })
```

When defining a hook as a string, the `$before` and `$after` variables are set to the previous and current environment variable value, respectively, similarly to the previous examples:

```
$env.config = ($env.config | upsert hooks {    env_change:
 {        PWD: {            code: 'print $"changing directory
from ($before) to ($after)"'        }    } }
```

Examples

Adding a single hook to existing config

An example for PWD env change hook:

```
$env.config = ($env.config | upsert hooks.env_change.PWD
{|config|    let val = ($config | get -i hooks.env_change.
PWD)    if $val == null {        $val | append {|before,
after| print $"changing directory from ($before) to ($after)
" }    } else {        [            {|before, after|
print $"changing directory from ($before) to ($after)"
}        ]    } })
```

Automatically activating an environment when entering a directory

This one looks for `test-env.nu` in a directory

```
$env.config = ($env.config | upsert hooks.env_change.PWD
{    [        {            condition: {|_, after|
             ($after == '/path/to/target/dir'
           and ($after | path join test-env.nu | path exists)
)            }            code: "overlay use test-env.
nu"        }        {            condition: {|before,
after|                ('/path/to/target/dir' not-in $after
                    and '/path/to/target/dir' in $before
                    and 'test-env' in (overlay list))
           }            code: "overlay hide test-env -
-keep-env [ PWD ]"        }    ] })
```

Filtering or diverting command output

You can use the `display_output` hook to redirect the output of commands. You should define a block that works on all value types. The output of external commands is not filtered through `display_output`.

This hook can display the output in a separate window, perhaps as rich HTML text. Here is the basic idea of how to do that:

```
$env.config = ($env.config | upsert hooks {     display_-
output: { to html --partial --no-color | save --raw /tmp/
nu-output.html } })
```

You can view the result by opening `file:///tmp/nu-output.html` in a web browser. Of course this isn't very convenient unless you use a browser that automatically reloads when the file changes. Instead of the `save`[86] command, you would normally customize this to send the HTML output to a desired window.

Changing how output is displayed

You can change to default behavior of how output is displayed by using the `display_output` hook. Here is an example that changes the default display behavior to show a table 1 layer deep if the terminal is wide enough, or collapse otherwise:

```
$env.config = ($env.config | upsert hooks {     display_-
output: {if (term size).columns >= 100 { table -ed 1 }
else { table }} })
```

`command_not_found` hook in *Arch Linux*

The following hook uses the `pkgfile` command, to find which packages commands belong to in *Arch Linux*.

```
$env.config = {     ...other config...     hooks: {
     ...other hooks...          command_not_found: {
         |cmd_name| (                    try {
         let pkgs = (pkgfile --binaries --verbose $cmd_-
name)                         if ($pkgs | is-empty) {
                        return null                      }
                       (                            $"(ansi $env.
config.color_config.shape_external)($cmd_name)(ansi reset)
 " +                            $"may be found in the following
packages:\n($pkgs)"                          )
   }              )          }     }}
```

86 /commands/docs/save.md

Background tasks with Nu

Currently, Nushell doesn't have built-in background task management feature, but you can make it "support" background task with some tools, here are some examples:

1. Using a third-party task management tools, like pueue[87]
2. Using a terminal multiplexer, like tmux[88] or zellij[89]

Using nu with pueue

The module borrows the power of pueue[90], it is possible to schedule background tasks to pueue, and manage those tasks (such as viewing logs, killing tasks, or getting the running status of all tasks, creating groups, pausing tasks etc etc)

Unlike terminal multiplexer, you don't need to attach to multiple tmux sessions, and get task status easily.

Here we provide a nushell module[91] to work with pueue easiler.

Here is a setup example to make Nushell "support" background tasks:

1. Install pueue
2. run `pueued`, you can refer to start-the-daemon page[92] for more information.
3. Put the task.nu[93] file under `$env.NU_LIB_DIRS`.
4. Add a line to the `$nu.config-path` file: `use task.nu`
5. Restart Nushell.

Then you will get some commands to schedule background tasks. (e.g: `task spawn`, `task status`, `task log`)

[87] https://github.com/Nukesor/pueue
[88] https://github.com/tmux/tmux/wiki
[89] https://zellij.dev/
[90] https://github.com/Nukesor/pueue
[91] https://github.com/nushell/nu_scripts/tree/main/modules/background_task
[92] https://github.com/Nukesor/pueue/wiki/Get-started#start-the-daemon
[93] https://github.com/nushell/nu_scripts/blob/main/modules/background_task/task.nu

Cons: It spawns a new Nushell interpreter to execute every single task, so it doesn't inherit current scope's variables, custom commands, alias definition. It only inherits environment variables whose value can be converted to a string. Therefore, if you want to use custom commands or variables, you have to `use`[94] or `def`[95] them within the given block.

Using nu with terminal multiplexer

You can choose and install a terminal multiplexer and use it.

It allows you to easily switch between multiple programs in one terminal, detach them (they continue to run in the background) and reconnect them to a different terminal. As a result, it is very flexible and usable.

[94] `/commands/docs/use.md`
[95] `/commands/docs/def.md`

Chapter 6

Coming to Nu

If you are familiar with other shells or programming languages, you might find this chapter useful to get up to speed.

Coming from Bash shows how some patterns typical for Bash, or POSIX shells in general, can be mapped to Nushell. Similarly, Coming from CMD.EXE shows how built-in commands in the Windows Command Prompt can be mapped to Nushell.

Similar comparisons are made for some other shells and domain-specific languages, imperative languages, and functional languages. A separate comparison is made specifically for operators.

Coming from Bash

If you're coming from `Git Bash` on Windows, then the external commands you're used to (bash, grep, etc) will not be available in `nu` by default (unless you had explicitly made them available in the Windows Path environment variable). To make these commands available in `nu` as well, add the following line to your `config.nu` with either `append` or `prepend`.

```
$env.Path = ($env.Path | prepend 'C:\Program Files\Git\usr\bin')
```

Note: this table assumes Nu 0.60.0 or later.

Bash	Nu	Task
`ls`	`ls`	Lists the files in the current directory
`ls <dir>`	`ls <dir>`	Lists the files in the given directory
`ls pattern*`	`ls pattern*`	Lists files that match a given pattern
`ls -la`	`ls --long --all` or `ls -la`	List files with all available information, including hidden files
`ls -d */`	`ls | where type == dir`	List directories
`find . -name *.rs`	`ls **/*.rs`	Find recursively all files that match a given pattern
`find . -name Makefile | xargs vim`	`ls **/Makefile | get name | vim $in`	Pass values as command parameters
`cd <directory>`	`cd <directory>`	Change to the given directory
`cd`	`cd`	Change to the home directory
`cd -`	`cd -`	Change to the previous directory
`mkdir <path>`	`mkdir <path>`	Creates the given path
`mkdir -p <path>`	`mkdir <path>`	Creates the given path, creating parents as necessary
`touch test.txt`	`touch test.txt`	Create a file
`> <path>`	`| save --raw <path>`	Save string into a file
`>> <path>`	`| save --raw --append <path>`	Append string to a file
`> /dev/null`	`| ignore`	Discard command output
`> /dev/null 2>&1`	`out+err> /dev/null`	Discard command output, including stderr
`command arg1 arg2 2>&1 | less`	`run-external --redirect-combine command [arg1 arg2] | less`	Pipe stdout+stderr of a command into less, output updated live
`cat <path>`	`open --raw <path>`	Display the contents

Coming from CMD.EXE

This table was last updated for Nu 0.67.0.

CMD.EXE	Nu	Task
`ASSOC`		Displays or modifies file extension associations
`BREAK`		Trigger debugger breakpoint
`CALL <filename.bat>`	`<filename.bat>`	Run a batch program
	`nu <filename>`	Run a nu script in a fresh context
	`source <filename>`	Run a nu script in this context
	`use <filename>`	Run a nu script as a module
`CD` or `CHDIR`	`$env.PWD`	Get the present working directory
`CD <directory>`	`cd <directory>`	Change the current directory
`CD /D <drive:directory>`	`cd <drive:directory>`	Change the current directory
`CLS`	`clear`	Clear the screen
`COLOR`		Set the console default foreground/background colors
	`ansi {flags} (code)`	Output ANSI codes to change color
`COPY <source> <destination>`	`cp <source> <destination>`	Copy files
`COPY <file1>+<file2> <destination>`	`[<file1>, <file2>] | each { open --raw } | str join | save --raw <destination>`	Append multiple files into one
`DATE /T`	`date now`	Get the current date
`DATE`		Set the date
`DEL <file>` or `ERASE <file>`	`rm <file>`	Delete files
`DIR`	`ls`	List files in the current directory
`ECHO <message>`	`print <message>`	Print the given values to stdout
`ECHO ON`		Echo executed commands to stdout

Before Nu version 0.67, Nu used to[2] use CMD.EXE to launch external commands, which meant that the above builtins could be run as an `^external` command. As of version 0.67, however, Nu no longer uses CMD.EXE to launch externals, meaning the above builtins cannot be run from within Nu, except for `ASSOC`, `CLS`, `ECHO`, `FTYPE`, `MKLINK`, `PAUSE`, `START`, `VER`, and `VOL`, which are explicitly allowed to be interpreted by CMD if no executable by that name exists.

Nu map from other shells and domain specific languages

The idea behind this table is to help you understand how Nu builtins and plugins relate to other known shells and domain specific languages. We've tried to produce a map of relevant Nu commands and what their equivalents are in other languages. Contributions are welcome.

Note: this table assumes Nu 0.43 or later.

[2] https://www.nushell.sh/blog/2022-08-16-nushell-0_67.html#windows-cmd-exe-changes-rgwood

Nushell	SQL	.Net LINQ (C#)	PowerShell (without external modules)	Bash
alias			alias	alias
append		Append	-Append	
math avg	avg	Average	Measure-Object, measure	
calc, `<math expression>`	math operators	Aggregate, Average, Count, Max, Min, Sum		bc
cd			Set-Location, cd	cd
clear			Clear-Host	clear
config			$Profile	vi .bashrc, .profile
cp			Copy-Item, cp, copy	cp
date	NOW() / getdate()	DateTime class	Get-Date	date
du				du
each	cursor		ForEach-Object, foreach, for	
exit			exit	exit
http		HttpClient,WebClient, HttpWebRequest/Response	Invoke-WebRequest	wget
first	top, limit	First, FirstOrDefault	Select-Object -First	head
format		String.Format	String.Format	
from	import flatfile, openjson, cast(variable as xml)		Import/ConvertFrom-{Csv,Xml,Html,Json}	
get		Select	(cmd).column	
group-by	group by	GroupBy,	Group-	

Nu map from imperative languages

The idea behind this table is to help you understand how Nu built-ins and plugins relate to imperative languages. We've tried to produce a map of programming-relevant Nu commands and what their equivalents are in other languages. Contributions are welcome.

Note: this table assumes Nu 0.43 or later.

Nushell	Python	Kotlin (Java)	C++	Rust
append	list.append, set.add	add	push_back, emplace_back	push, push_back
math avg	statistics.mean			
calc, = math	math operators	math operators	math operators	math operators
count	len	size, length	length	len
cp	shutil.copy			fs::copy
date	datetime.date.today	java.time.LocalDate.now		
drop	list[:-3]			
du	shutil.disk_usage			
each	for	for	for	for
exit	exit	System.exit, kotlin.system.exitProcess	exit	exit
http get	urllib.request.urlopen			
first	list[:x]	List[0], peek	vector[0], top	Vec[0]
format	format	format	format	format!
from	csv, json, sqlite3			
get	dict[“key”]	Map[“key”]	map[“key”]	HashMap[“key”], get, entry
group-by	itertools.groupby	groupBy		group_by
headers	keys			
help	help			
insert	dict[“key”] = val		map.insert({ 20, 130 })	map.insert(“key”, val)
is-empty	is None, is []	isEmpty	empty	is_empty
take	list[:x]			&Vec[..x]
take until	itertools.takewhile			
take while	itertools.takewhile			
kill	os.kill			
last	list[-x:]			&Vec[Vec.len()-1]
lines	split, splitlines	split	views::split	split, split_whitespace, rsplit, lines
ls	os.listdir			fs::read_-

Nu map from functional languages

The idea behind this table is to help you understand how Nu builtins and plugins relate to functional languages. We've tried to produce a map of relevant Nu commands and what their equivalents are in other languages. Contributions are welcome.

Note: this table assumes Nu 0.43 or later.

Nushell	Clojure	Tablecloth (Ocaml / Elm)	Haskell
append	conj, into, concat	append, (++), concat, concatMap	(++)
into binary	Integer/toHexString		showHex
count	count	length, size	length, size
date	java.time.LocalDate/now		
each	map, mapv, iterate	map, forEach	map, mapM
exit	System/exit		
first	first	head	head
format	format		Text.Printf.printf
group-by	group-by		group, groupBy
help	doc		
is-empty	empty?	isEmpty	
last	last, peek, take-last	last	last
lines			lines, words, split-with
match		match (Ocaml), case (Elm)	case
nth	nth	Array.get	lookup
open	with-open		
transpose	(apply mapv vector matrix)		transpose
prepend	cons	cons, ::	::
print	println		putStrLn, print
range, 1..10	range	range	1..10, 'a'..'f'
reduce	reduce, reduce-kv	foldr	foldr
reverse	reverse, rseq	reverse, reverseInPlace	reverse
select	select-keys		
shuffle	shuffle		
size	count		size, length
skip	rest	tail	tail
skip until	drop-while		
skip while	drop-while	dropWhile	dropWhile, dropWhileEnd
sort-by	sort, sort-by,	sort, sortBy,	sort, sortBy

Nushell operator map

The idea behind this table is to help you understand how Nu operators relate to other language operators. We've tried to produce a map of all the nushell operators and what their equivalents are in other languages. Contributions are welcome.

Note: this table assumes Nu 0.14.1 or later.

Nushell	SQL	Python	.NET LINQ (C#)	PowerShell	Bash
==	=	==	==	-eq, -is	-eq
!=	!=, <>	!=	!=	-ne, -isnot	-ne
<	<	<	<	-lt	-lt
<=	<=	<=	<=	-le	-le
>	>	>	>	-gt	-gt
>=	>=	>=	>=	-ge	-ge
=~	like	re, in, startswith	Contains, StartsWith	-like, -contains	=~
!~	not like	not in	Except	-notlike, -notcontains	! “str1” =~ “str2”
+	+	+	+	+	+
-	-	-	-	-	-
	*	*	*	*	*
/	/	/	/	/	/
*	pow	**	Power	Pow	**
in	in	re, in, startswith	Contains, StartsWith	-In	case in
not-in	not in	not in	Except	-NotIn	
and	and	and	&&	-And, &&	-a, &&
or	or	or	||	-Or, ||	-o, ||

Chapter 7

Design Notes

This chapter intends to give more in-depth overview of certain aspects of Nushell's design. The topics are not necessary for a basic usage, but reading them will help you understand how Nushell works and why.

We intend to expand this chapter in the future. If there is some topic that you find confusing and hard to understand, let us know. It might be a good candidate for a page here.

How Nushell Code Gets Run explains what happens when you run Nushell source code. It explains how Nushell is in many ways closer to classic compiled languages, like C or Rust, than to other shells and dynamic languages and hopefully clears some confusion that stems from that.

How Nushell Code Gets Run

As you probably noticed, Nushell behaves quite differently from other shells and dynamic languages. In Thinking in Nu, we advise you to *think of Nushell as a compiled language* but we do not give much insight into why. This section hopefully fills the gap.

First, let's give a few example which you might intuitively try but which do not work in Nushell.

1. Sourcing a dynamic path (note that a constant would work, see parse-time evaluation[1])

[1] `#parse-time-evaluation`

```
let my_path = 'foo' source $"($my_path)/common.nu"
```

2. Write to a file and source it in a single script

```
"def abc [] { 1 + 2 }" | save output.nu source "output.
nu"
```

3. Change a directory and source a path within (even though the file exists)

```
if ('spam/foo.nu' | path exists) {    cd spam    source-
env foo.nu }
```

The underlying reason why all of the above examples won't work is a strict separation of **parsing and evaluation** steps by **disallowing eval function**. In the rest of this section, we'll explain in detail what it means, why we're doing it, and what the implications are. The explanation aims to be as simple as possible, but it might help if you've written a program in some language before.

Parsing and Evaluation

Interpreted Languages

Let's start with a simple "hello world" Nushell program:

```
# hello.nu print "Hello world!"
```

When you run `nu hello.nu`, Nushell's interpreter directly runs the program and prints the result to the screen. This is similar (on the highest level) to other languages that are typically interpreted, such as Python or Bash. If you write a similar "hello world" program in any of these languages and call `python hello.py` or `bash hello.bash`, the result will be printed to the screen. We can say that interpreters take the program in some representation (e.g., a source code), run it, and give you the result:

```
source code --> interpreting --> result
```

Under the hood, Nushell's interpreter is split into two parts, like this:

```
1. source code --> parsing --> Intermediate Representation
(IR) 2. IR --> evaluating --> result
```

First, the source code is analyzed by the parser and converted into an intermediate representation (IR), which in Nushell's case are just some data structures. Then, these data structures are passed to the engine which evaluates them and produces the result. This is nothing unusual. For example, Python's source code is typically converted into bytecode[2] before evaluation.

Compiled Languages

On the other side are languages that are typically "compiled", such as C, C++, or Rust. Assuming a simple "hello world"[3] in Rust

```
// main.rs fn main() {     println!("Hello, world!
"); }
```

you first need to *compile* the program into machine code instructions[4] and store the binary file to a disk (`rustc main.rs`). Then, to produce a result, you need to run the binary (`./main`), which passes the instructions to the CPU:

```
1. source code --> compiler --> machine code 2. machine
code --> CPU --> result
```

You can see the compile-run sequence is not that much different from the parse-evaluate sequence of an interpreter. You begin with a source code, parse (or compile) it into some IR (or machine code), then evaluate (or run) the IR to get a result. You could think of machine code as just another type of IR and the CPU as its interpreter.

One big difference, however, between interpreted and compiled languages is that interpreted languages typically implement an *eval function* while compiled languages do not. What does it mean?

Eval Function

Most languages considered as "dynamic" or "interpreted" have an eval function, for example Python (it has two, eval[5] and exec[6]) or Bash[7]. It

[2]https://en.wikipedia.org/wiki/Bytecode
[3]https://doc.rust-lang.org/stable/book/ch01-02-hello-world.html
[4]https://en.wikipedia.org/wiki/Machine_code
[5]https://docs.python.org/3/library/functions.html#eval
[6]https://docs.python.org/3/library/functions.html#exec
[7]https://linux.die.net/man/1/bash

is used to take source code and interpret it within a running interpreter. This can get a bit confusing, so let's give a Python example:

```
# hello_eval.py print ( " Hello world! " ) eval ( " print(
'Hello eval!') " )
```

When you run the file (`python hello_eval.py`), you'll see two messages: "Hello world!" and "Hello eval!". Here is what happened:

1. Parse the whole source code
2. Evaluate `print("Hello world!")`
3. To evaluate `eval("print('Hello eval!')")`: 3.1. Parse `print('Hello eval!')` 3.2. Evaluate `print('Hello eval!')`

Of course, you can have more fun and try `eval("eval(\"print('Hello eval!')\")")` and so on...

You can see the eval function adds a new "meta" layer into the code execution. Instead of parsing the whole source code, then evaluating it, there is an extra parse-eval step during the evaluation. This means that the IR produced by the parser (whatever it is) can be further modified during the evaluation.

We've seen that without `eval`, the difference between compiled and interpreted languages is actually not that big. This is exactly what we mean by thinking of Nushell as a compiled language[8]: Despite Nushell being an interpreted language, its lack of `eval` gives it characteristics and limitations typical for traditional compiled languages like C or Rust. We'll dig deeper into what it means in the next section.

Implications

Consider this Python example:

```
exec ( " def hello(): print('Hello eval!') " ) hello ( )
```

Note: We're using `exec` instead of `eval` because it can execute all valid Python code, not just expressions. The principle is similar, though.

What happens:

1. Parse the whole source code

[8] https://www.nushell.sh/book/thinking_in_nu.html#think-of-nushell-as-a-compiled-language

2. To evaluate `exec("def hello(): print('Hello eval!')")`: 2.1. Parse `def hello(): print('Hello eval!')` 2.2 Evaluate `def hello(): print('Hello eval!')`

3. Evaluate `hello()`

Note, that until step 2.2, the interpreter has no idea a function `hello` exists! This makes static analysis of dynamic languages challenging. In the example, the existence of `hello` function cannot be checked just by parsing (compiling) the source code. You actually need to go and evaluate (run) the code to find out. While in a compiled language, missing function is a guaranteed compile error, in a dynamic interpreted language, it is a runtime error (which can slip unnoticed if the line calling `hello()` is, for example, behind an `if` condition and does not get executed).

In Nushell, there are **exactly two steps**:

1. Parse the whole source code

2. Evaluate the whole source code

This is the complete parse-eval sequence.

Not having `eval`-like functionality prevents `eval`-related bugs from happening. Calling a non-existent function is 100% guaranteed parse-time error in Nushell. Furthermore, after the parse step, we have a deep insight into the program and we're 100% sure it is not going to change during evaluation. This trivially allows for powerful and reliable static analysis and IDE integration which is challenging to achieve with more dynamic languages. In general, you have more peace of mind when scaling Nushell programs to bigger applications.

Before going into examples, one note about the "dynamic" and "static" terminology. Stuff that happens at runtime (during evaluation, after parsing) is considered "dynamic". Stuff that happens before running (during parsing / compilation) is called "static". Languages that have more stuff (such as `eval`, type checking, etc.) happening at runtime are sometimes called "dynamic". Languages that analyze most of the information (type checking, data ownership[9], etc.) before evaluating the program are sometimes called "static". The whole debate can get quite confusing, but for the purpose of this text, the main difference between a "static" and "dynamic" language is whether it has or has not the eval function.

[9] https://doc.rust-lang.org/stable/book/ch04-00-understanding-ownership.html

Common Mistakes

By insisting on strict parse-evaluation separation, we lose much of a flexibility users expect from dynamic interpreted languages, especially other shells, such as bash, fish, zsh and others. This leads to the examples at the beginning of this page not working. Let's break them down one by one

Note: The following examples use `source`[10], but similar conclusions apply to other commands that parse Nushell source code, such as `use`[11], `overlay use`[12], `hide`[13], `register`[14] or `source-env`[15].

1. Sourcing a dynamic path

```
let my_path = 'foo' source $"($my_path)/common.nu"
```

Let's break down what would need to happen for this to work:

1. Parse `let my_path = 'foo'` and `source $"($my_path)/config.nu"`
2. To evaluate `source $"($my_path)/common.nu"`:
   (a) Parse `$"($my_path)/common.nu"`
   (b) Evaluate `$"($my_path)/common.nu"` to get the file name
   (c) Parse the contents of the file
   (d) Evaluate the contents of the file

You can see the process is similar to the `eval` functionality we talked about earlier. Nesting parse-evaluation cycles into the evaluation is not allowed in Nushell.

To give another perspective, here is why it is helpful to *think of Nushell as a compiled language.* Instead of

```
let my_path = 'foo' source $"($my_path)/common.nu"
```

imagine it being written in some typical compiled language, such as C++

[10] /commands/docs/source.md
[11] /commands/docs/use.md
[12] /commands/docs/overlay_use.md
[13] /commands/docs/hide.md
[14] /commands/docs/register.md
[15] /commands/docs/source-env.md

```
#include  < string > std :: string  my_path ( " foo " ) ;
#include  < my_path + "/common.h" >
```

or Rust

```
let  my_path  =   " foo " ; use   format! ( " {} ::common "
,  my_path ) ;
```

If you've ever written a simple program in any of these languages, you can see these examples do not make a whole lot of sense. You need to have all the source code files ready and available to the compiler beforehand.

2. Write to a file and source it in a single script

```
"def abc [] { 1 + 2 }" | save output.nu source "output.
nu"
```

Here, the sourced path is static (= known at parse-time) so everything should be fine, right? Well... no. Let's break down the sequence again:

1. Parse the whole source code
   (a) Parse `"def abc [] { 1 + 2 }" | save output.nu`
   (b) Parse `source "output.nu"` - 1.2.1. Open `output.nu` and parse its contents
2. Evaluate the whole source code
   (a) Evaluate `"def abc [] { 1 + 2 }" | save output.nu` to generate `output.nu`
   (b) ...wait what???

We're asking Nushell to read `output.nu` before it even exists. All the source code needs to be available to Nushell at parse-time, but `output.nu` is only generated during evaluation. Again, it helps here to *think of Nushell as a compiled language.*

3. Change a directory and source a path within

(We assume the `spam/foo.nu` file exists.)

```
if ('spam/foo.nu' | path exists) {    cd spam    source-
env foo.nu }
```

This one is similar to the previous example. `cd spam` changes the directory *during evaluation* but `source-env`[16] attempts to open and read `foo.nu` during parsing.

REPL

REPL[17] is what happens when you run `nu` without any file. You launch an interactive prompt. By

```
> some code...
```

we denote a REPL entry followed by pressing Enter. For example

```
> print "Hello world!" Hello world! > ls # prints files
and directories...
```

means the following:

1. Launch `nu`
2. Type `print "Hello world!"`, press Enter
3. Type `ls`[18], press Enter

Hopefully, that's clear. Now, when you press Enter, these things happen:

1. Parse the line input
2. Evaluate the line input
3. Merge the environment (such as the current working directory) to the internal Nushell state
4. Wait for another input

In other words, each REPL invocation is its own separate parse-evaluation sequence. By merging the environment back to the Nushell's state, we maintain continuity between the REPL invocations.

To give an example, we showed that

[16] /commands/docs/source-env.md
[17] https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop
[18] /commands/docs/ls.md

```
cd spam source-env foo.nu
```

does not work because the directory will be changed *after* source-env[19] attempts to read the file. Running these commands as separate REPL entries, however, works:

```
> cd spam > source-env foo.nu # yay, works!
```

To see why, let's break down what happens in the example:

1. Launch `nu`
2. Parse `cd spam`
3. Evaluate `cd spam`
4. **Merge environment (including the current directory) into the Nushell state**
5. Parse `source-env foo.nu`
6. Evaluate `source-env foo.nu`
7. Merge environment (including the current directory) into the Nushell state

When source-env[20] tries to open `foo.nu` during the parsing in step 5., it can do so because the directory change from step 3. was merged into the Nushell state in step 4. and therefore is visible in the following parse-evaluation cycles.

Parse-time Evaluation

While it is impossible to add parsing into the evaluation, we can add *a little bit* of evaluation into parsing. This feature has been added only recently[21] and we're going to expand it as needed.

One pattern that this unlocks is being able to source[22]/use[23]/etc. a path from a "variable". We've seen that

[19] /commands/docs/source-env.md
[20] /commands/docs/source-env.md
[21] https://github.com/nushell/nushell/pull/7436
[22] /commands/docs/source.md
[23] /commands/docs/use.md

```
let some_path = $nu.default-config-dir source $"($some_-
path)/common.nu"
```

does not work, but we can do the following:

```
const some_path = $nu.default-config-dir source $"($some_-
path)/config.nu"
```

We can break down what is happening again:

1. Parse the whole source code
   (a) Parse `const some_path = $nu.default-config-dir`
      i. Evaluate* `$nu.default-config-dir` to `/home/user/.config/nushell` and store it as a `some_path` constant
   (b) Parse `source $"($some_path)/config.nu"`
      i. Evaluate* `$some_path`, see that it is a constant, fetch it
      ii. Evaluate* `$"($some_path)/config.nu"` to `/home/user/.config/nushell/config.nu`
      iii. Parse the `/home/user/.config/nushell/config.nu` file
2. Evaluate the whole source code
   (a) Evaluate `const some_path = $nu.default-config-dir` (i.e., add the `/home/user/.config/nushell` string to the runtime stack as `some_path` variable)
   (b) Evaluate `source $"($some_path)/config.nu"` (i.e., evaluate the contents of `/home/user/.config/nushell/config.nu`)

This still does not violate our rule of not having an eval function, because an eval function adds additional parsing to the evaluation step. With parse-time evaluation we're doing the opposite.

Also, note the * in steps 1.1.1. and 1.2.1. The evaluation happening during parsing is very restricted and limited to only a small subset of what is normally allowed during a regular evaluation. For example, the following is not allowed:

```
const foo_contents = (open foo.nu)
```

By allowing *everything* during parse-time evaluation, we could set ourselves up to a lot of trouble (think of generating an infinite stream in a subexpression...). Generally, only a simple expressions *without side effects* are allowed, such as string literals or integers, or composite types of these literals (records, lists, tables).

Compiled ("static") languages also tend to have a way to convey some logic at compile time, be it C's preprocessor, Rust's macros, or Zig's comptime[24]. One reason is performance (if you can do it during compilation, you save the time during runtime) which is not as important for Nushell because we always do both parsing and evaluation, we do not store the parsed result anywhere (yet?). The second reason is similar to Nushell's: Dealing with limitations caused by the absence of the eval function.

Conclusion

Nushell operates in a scripting language space typically dominated by "dynamic" "interpreted" languages, such as Python, bash, zsh, fish, etc. While Nushell is also "interpreted" in a sense that it runs the code immediately, instead of storing the intermediate representation (IR) to a disk, one feature sets it apart from the pack: It does not have an **eval function**. In other words, Nushell cannot parse code and manipulate its IR during evaluation. This gives Nushell one characteristic typical for "static" "compiled" languages, such as C or Rust: All the source code must be visible to the parser beforehand, just like all the source code must be available to a C or Rust compiler. For example, you cannot `source`[25] or `use`[26] a path computed "dynamically" (during evaluation). This is surprising for users of more traditional scripting languages, but it helps to *think of Nushell as a compiled language.*

[24] `https://kristoff.it/blog/what-is-zig-comptime`
[25] `/commands/docs/source.md`
[26] `/commands/docs/use.md`

Chapter 8

(Not So) Advanced

While the "Advanced" title might sound daunting and you might be tempted to skip this chapter, in fact, some of the most interesting and powerful features can be found here.

Besides the built-in commands, Nushell has a standard library.

Nushell operates on *structured data*. You could say that Nushell is a "data-first" shell and a programming language. To further explore the data-centric direction, Nushell includes a full-featured dataframe processing engine using Polars[1] as the backend. Make sure to check the Dataframes documentation if you want to process large data efficiently directly in your shell.

Values in Nushell contain some extra metadata. This metadata can be used, for example, to create custom errors.

Thanks to Nushell's strict scoping rules, it is very easy to iterate over collections in parallel which can help you speed up long-running scripts by just typing a few characters.

You can interactively explore data with the `explore`[2] command.

Finally, you can extend Nushell's functionality with plugins. Almost anything can be a plugin as long as it communicates with Nushell in a protocol that Nushell understands.

[1] `https://github.com/pola-rs/polars`
[2] `/commands/docs/explore.md`

Standard library (preview)

The standard library is located on the Git repository[3]. At the moment it is an alpha development stage. You can find more documentation there.

Dataframes

::: warning To use the dataframe support you need a fully-featured build with `cargo build --features dataframe`. Starting with version 0.72, dataframes are *not* included with binary releases of Nushell. See the installation instructions[4] for further details. :::

As we have seen so far, Nushell makes working with data its main priority. `Lists` and `Tables` are there to help you cycle through values in order to perform multiple operations or find data in a breeze. However, there are certain operations where a row-based data layout is not the most efficient way to process data, especially when working with extremely large files. Operations like group-by or join using large datasets can be costly memory-wise, and may lead to large computation times if they are not done using the appropriate data format.

For this reason, the `DataFrame` structure was introduced to Nushell. A `DataFrame` stores its data in a columnar format using as its base the Apache Arrow[5] specification, and uses Polars[6] as the motor for performing extremely fast columnar operations[7].

You may be wondering now how fast this combo could be, and how could it make working with data easier and more reliable. For this reason, let's start this page by presenting benchmarks on common operations that are done when processing data.

Benchmark comparisons

For this little benchmark exercise we will be comparing native Nushell commands, dataframe Nushell commands and Python Pandas[8] commands. For the time being don't pay too much attention to the `Dataframe` commands[9]. They will be explained in later sections of this page.

[3] https://github.com/nushell/nushell/tree/main/crates/nu-std
[4] /book/installation.md
[5] https://arrow.apache.org/
[6] https://github.com/pola-rs/polars
[7] https://h2oai.github.io/db-benchmark/
[8] https://pandas.pydata.org/
[9] /commands/categories/dataframe.md

System Details: The benchmarks presented in this section were run using a machine with a processor Intel(R) Core(TM) i7-10710U (CPU @1.10GHz 1.61 GHz) and 16 gb of RAM.

All examples were run on Nushell version 0.33.1. (Command names are updated to Nushell 0.78)

File information

The file that we will be using for the benchmarks is the New Zealand business demography[10] dataset. Feel free to download it if you want to follow these tests.

The dataset has 5 columns and 5,429,252 rows. We can check that by using the `dfr ls` command:

```
  let df = (dfr open .\Data7602DescendingYearOrder.csv)
dfr ls                          #    name     columns    rows
                          0    $df            5    5429252
```

We can have a look at the first lines of the file using `first`[11]:

```
  $df | dfr first                                        #   anzsic06
   Area      year   geo_count   ec_count
  0   A           A100100   2000            96          130
```

...and finally, we can get an idea of the inferred data types:

```
  $df | dfr dtypes                   #    column     dtype
                   0   anzsic06    str       1   Area
  str       2   year        i64       3   geo_count   i64
    4   ec_count    i64
```

Loading the file

Let's start by comparing loading times between the various methods. First, we will load the data using Nushell's `open`[12] command:

[10] `https://www.stats.govt.nz/assets/Uploads/New-Zealand-business-demography-statis`
`New-Zealand-business-demography-statistics-At-February-2020/`
`Download-data/Geographic-units-by-industry-and-statistical-area-2000-2020-descendi`
`zip`

[11] `/commands/docs/first.md`

[12] `/commands/docs/open.md`

```
 timeit {open .\Data7602DescendingYearOrder.csv} 30sec
479ms 614us 400ns
```

Loading the file using native Nushell functionality took 30 seconds. Not bad for loading five million records! But we can do a bit better than that.

Let's now use Pandas. We are going to use the next script to load the file:

```
import pandas as pd df = pd . read_csv ( " Data7602DescendingYearOrde
csv " )
```

And the benchmark for it is:

```
 timeit {python load.py} 2sec 91ms 872us 900ns
```

That is a great improvement, from 30 seconds to 2 seconds. Nicely done, Pandas!

Probably we can load the data a bit faster. This time we will use Nushell's `dfr open` command:

```
 timeit {dfr open .\Data7602DescendingYearOrder.csv} 601ms
700us 700ns
```

This time it took us 0.6 seconds. Not bad at all.

Group-by comparison

Let's do a slightly more complex operation this time. We are going to group the data by year, and add groups using the column `geo_count`.

Again, we are going to start with a Nushell native command.

:: tip If you want to run this example, be aware that the next command will use a large amount of memory. This may affect the performance of your system while this is being executed. :::

```
 timeit { open .\Data7602DescendingYearOrder.csv | group-
by year | transpose header rows | upsert rows { get rows
| math sum } | flatten } 6min 30sec 622ms 312us
```

So, six minutes to perform this aggregated operation.

Let's try the same operation in pandas:

```
import pandas as pd df = pd . read_csv ( " Data7602DescendingYea
csv " ) res = df . groupby ( " year " ) [ " geo_count " ]
. sum ( ) print ( res )
```

And the result from the benchmark is:

```
timeit {python .\load.py} 1sec 966ms 954us 800ns
```

Not bad at all. Again, pandas managed to get it done in a fraction of the time.

To finish the comparison, let's try Nushell dataframes. We are going to put all the operations in one `nu` file, to make sure we are doing similar operations:

```
let df = (dfr open Data7602DescendingYearOrder.csv) let
res = ($df | dfr group-by year | dfr agg (dfr col geo_-
count | dfr sum)) $res
```

and the benchmark with dataframes is:

```
timeit {source load.nu} 557ms 658us 500ns
```

Luckily Nushell dataframes managed to halve the time again. Isn't that great?

As you can see, Nushell's Dataframe commands[13] are as fast as the most common tools that exist today to do data analysis. The commands that are included in this release have the potential to become your go-to tool for doing data analysis. By composing complex Nushell pipelines, you can extract information from data in a reliable way.

Working with Dataframes

After seeing a glimpse of the things that can be done with Dataframe commands[14], now it is time to start testing them. To begin let's create a sample CSV file that will become our sample dataframe that we will be using along with the examples. In your favorite file editor paste the next lines to create out sample csv file.

```
int_1,int_2,float_1,float_2,first,second,third,word 1,11,0.
1,1.0,a,b,c,first 2,12,0.2,1.0,a,b,c,second 3,13,0.3,2.
0,a,b,c,third 4,14,0.4,3.0,b,a,c,second 0,15,0.5,4.0,b,a,a,third
6,16,0.6,5.0,b,a,a,second 7,17,0.7,6.0,b,c,a,third 8,18,0.
```

[13] /commands/categories/dataframe.md
[14] /commands/categories/dataframe.md

```
8,7.0,c,c,b,eight 9,19,0.9,8.0,c,c,b,ninth 0,10,0.0,9.0,c,c,b,ninth
```

Save the file and name it however you want to, for the sake of these examples the file will be called `test_small.csv`.

Now, to read that file as a dataframe use the `dfr open` command like this:

```
let df = (dfr open test_small.csv)
```

This should create the value `$df` in memory which holds the data we just c

:: tip The command `dfr open` can read either **csv** or **parquet** files. :::

To see all the dataframes that are stored in memory you can use

```
  dfr ls                     #   name   columns   rows
  0   $df            8     10
```

As you can see, the command shows the created dataframes together with basic information about them.

And if you want to see a preview of the loaded dataframe you can send the dataframe variable to the stream

```
  $df                                                      #   int_-
1   int_2   float_1   float_2   first   second   third
word                                                         0
  1      11      0.10      1.00   a       b        c
first     1        2      12       0.20      1.00   a
b        c       second    2       3      13      0.30
  2.00   a       b        c       third     3       4
 14      0.40      3.00   b       a         c       second
   4      0      15      0.50      4.00   b       a
 a       third     5       6      16      0.60      5.00
 b       a        a       second    6       7      17
  0.70      6.00   b       c        a       third     7
    8      18      0.80      7.00   c       c        b
   eight     8       9      19      0.90      8.00   c
   c       b       ninth     9       0      10      0.00
     9.00   c       c        b       ninth
```

With the dataframe in memory we can start doing column operations with

:: tip If you want to see all the dataframe commands that are available you can use `scope commands | where category =~ dataframe` :::

Basic aggregations

Let's start with basic aggregations on the dataframe. Let's sum all the columns that exist in `df` by using the `aggregate` command

```
  $df | dfr sum
#   int_1   int_2   float_1   float_2   first   second   third
  word                                                  0
 40     145      4.50     46.00

```

As you can see, the aggregate function computes the sum for those columns where a sum makes sense. If you want to filter out the text column, you can select the columns you want by using the `dfr select`[15] command

```
  $df | dfr sum | dfr select int_1 int_2 float_1 float_-
2                           #   int_1   int_2   float_1   float_-
2                            0      40     145      4.50
  46.00
```

You can even store the result from this aggregation as you would store any other Nushell variable

```
  let res = ($df | dfr sum | dfr select int_1 int_2 float_-
1 float_2)
```

::: tip Type `let res = ( !! )` and press enter. This will auto complete the previously executed command. Note the space between (and !!. :::

And now we have two dataframes stored in memory

```
  dfr ls                      #   name   columns   rows
  0   $res         4       1    1   $df            8     10

```

Pretty neat, isn't it?

You can perform several aggregations on the dataframe in order to extract basic information from the dataframe and do basic data analysis on your brand new dataframe.

[15] `/commands/docs/dfr_select.md`

Joining a DataFrame

It is also possible to join two dataframes using a column as reference. We are going to join our mini dataframe with another mini dataframe. Copy these lines in another file and create the corresponding dataframe (for these examples we are going to call it `test_small_a.csv`)

```
int_1,int_2,float_1,float_2,first 9,14,0.4,3.0,a 8,13,0.
3,2.0,a 7,12,0.2,1.0,a 6,11,0.1,0.0,b
```

We use the `dfr open` command to create the new variable

```
  let df_a = (dfr open test_small_a.csv)
```

Now, with the second dataframe loaded in memory we can join them using the column called `int_1` from the left dataframe and the column `int_1` from the right dataframe

```
  $df | dfr join $df_a int_1 int_1
  #   int_1   int_2   float_1   float_2   first   second
third    word    int_2_x   float_1_x   float_2_x   first_-
x
  0       6      16      0.60      5.00   b       a
a       second         11         0.10         0.00   b
     1       7      17      0.70      6.00   b       c
   a       third         12         0.20         1.00   a
        2       8      18      0.80      7.00   c       c
       b       eight         13         0.30         2.00
a           3       9      19      0.90      8.00   c
  c        b       ninth         14         0.40         3.00
  a
```

::: tip In Nu when a command has multiple arguments that are expecting multiple values we use brackets `[]` to enclose those values. In the case of `dfr join`[16] we can join on multiple columns as long as they have the same type. :::

For example:

```
  $df | dfr join $df_a [int_1 first] [int_1 first]
  #   int_1   int_2   float_1   float_2   first   second
third    word    int_2_x   float_1_x   float_2_x
  0       6      16      0.60      5.00   b       a
```

[16] `/commands/docs/dfr_join.md`

```
 a       second        11         0.10         0.00

```

By default, the join command does an inner join, meaning that it will keep the rows where both dataframes share the same value. You can select a left join to keep the missing rows from the left dataframe. You can also save this result in order to use it for further operations.

DataFrame group-by

One of the most powerful operations that can be performed with a DataFrame is the dfr group-by[17]. This command will allow you to perform aggregation operations based on a grouping criteria. In Nushell, a GroupBy is a type of object that can be stored and reused for multiple aggregations. This is quite handy, since the creation of the grouped pairs is the most expensive operation while doing group-by and there is no need to repeat it if you are planning to do multiple operations with the same group condition.

To create a GroupBy object you only need to use the dfr_group-by[18] command

```
   let group = ($df | dfr group-by first)  $group
   LazyGroupBy   apply aggregation to complete execution
 plan
```

When printing the GroupBy object we can see that it is in the background a lazy operation waiting to be completed by adding an aggregation. Using the GroupBy we can create aggregations on a column

```
   $group | dfr agg (dfr col int_1 | dfr sum)
   #   first   int_1               0   b          17    1
   a             6    2   c           17
```

or we can define multiple aggregations on the same or different columns

```
   $group | dfr agg [  (dfr col int_1 | dfr n-unique)  (
 dfr col int_2 | dfr min)  (dfr col float_1 | dfr sum)
 (dfr col float_2 | dfr count)  ] | dfr sort-by first
   #   first   int_1   int_2   float_1   float_2
   0   a           3      11      0.60          3    1   b
         4      14      2.20          4    2   c

```

[17]/commands/docs/dfr_group-by.md
[18]/commands/docs/dfr_group-by.md

```
3       10       1.70           3
```

As you can see, the `GroupBy` object is a very powerful variable and it is worth keeping in memory while you explore your dataset.

Creating Dataframes

It is also possible to construct dataframes from basic Nushell primitives, such as integers, decimals, or strings. Let's create a small dataframe using the command `dfr into-df`.

```
  let a = ([[a b]; [1 2] [3 4] [5 6]] | dfr into-df)  $a
```

::: tip For the time being, not all of Nushell primitives can be converted into a dataframe. This will change in the future, as the dataframe feature matures :::

We can append columns to a dataframe in order to create a new variable. As an example, let's append two columns to our mini dataframe `$a`

```
  let a2 = ($a | dfr with-column $a.a --name a2 | dfr with-
column $a.a --name a3)  $a2                 #   a   b   a2
  a3                    0   1   2    1    1    1   3   4    3
   3    2   5   6    5    5
```

Nushell's powerful piping syntax allows us to create new dataframes by taking data from other dataframes and appending it to them. Now, if you list your dataframes you will see in total four dataframes

```
  dfr ls                       #   name    columns   rows
  0   $a2            4      3    1   $df_a          5      4
    2   $df            8      10    3   $a               2
3    4   $res             4       1
```

One thing that is important to mention is how the memory is being optimized while working with dataframes, and this is thanks to **Apache Arrow** and **Polars**. In a very simple representation, each column in a DataFrame is an Arrow Array, which is using several memory specifications in order to maintain the data as packed as possible (check Arrow columnar format[19]). The other optimization trick is the fact

[19] https://arrow.apache.org/docs/format/Columnar.html

that whenever possible, the columns from the dataframes are shared between dataframes, avoiding memory duplication for the same data. This means that dataframes `$a` and `$a2` are sharing the same two columns we created using the `dfr into-df` command. For this reason, it isn't possible to change the value of a column in a dataframe. However, you can create new columns based on data from other columns or dataframes.

Working with Series

A `Series` is the building block of a `DataFrame`. Each Series represents a column with the same data type, and we can create multiple Series of different types, such as float, int or string.

Let's start our exploration with Series by creating one using the `dfr into-df` command:

```
  let new = ([9 8 4] | dfr into-df)  $new        #   0
       0   9    1   8    2   4
```

We have created a new series from a list of integers (we could have done the same using floats or strings)

Series have their own basic operations defined, and they can be used to create other Series. Let's create a new Series by doing some arithmetic on the previously created column.

```
  let new_2 = ($new * 3 + 10)  $new_2        #   0
  0   37    1   34    2   22
```

Now we have a new Series that was constructed by doing basic opera

:: tip If you want to see how many variables you have stored in memory you can use `scope variables` :::

Let's rename our previous Series so it has a memorable name

```
  let new_2 = ($new_2 | dfr rename "0" memorable)  $new_-
 2           #   memorable               0          37
 1          34    2          22
```

We can also do basic operations with two Series as long as they have the same data type

```
  $new - $new_2                  #   sub_0_memorable
  0               -28    1               -26    2
```

```
        -18
```

And we can add them to previously defined dataframes

```
  let new_df = ($a | dfr with-column $new --name new_col)
   $new_df              #   a   b   new_col
  0   1   2         9    1   3   4         8    2   5   6
      4
```

The Series stored in a Dataframe can also be used directly, for example, we can multiply columns `a` and `b` to create a new Series

```
  $new_df.a * $new_df.b           #   mul_a_b
  0         2    1        12    2        30
```

and we can start piping things in order to create new columns and dataframes

```
  let $new_df = ($new_df | dfr with-column ($new_df.a *
$new_df.b / $new_df.new_col) --name my_sum)  $new_df
  #   a   b   new_col   my_sum                     0   1
  2         9        0    1   3   4         8        1
2   5   6         4        7
```

Nushell's piping system can help you create very interesting workflows.

Series and masks

Series have another key use in when working with `DataFrames`, and it is the fact that we can build boolean masks out of them. Let's start by creating a simple mask using the equality operator

```
  let mask = ($new == 8)  $mask         #     0
  0   false    1   true     2   false
```

and with this mask we can now filter a dataframe, like this

```
  $new_df | dfr filter-with $mask                         #
  a   b   new_col   my_sum                      0   3   4
        8        1
```

Now we have a new dataframe with only the values where the mask was true.

The masks can also be created from Nushell lists, for example:

```
  let mask1 = ([true true false] | dfr into-df)  $new_-
df | dfr filter-with $mask1                          #   a
b   new_col   my_sum                           0   1   2
   9        0    1   3   4         8        1
```

To create complex masks, we have the AND

```
  $mask and $mask1           #   and_0_0            0
false      1   true       2   false
```

and OR operations

```
  $mask or $mask1           #   or_0_0            0   true
     1   true      2   false
```

We can also create a mask by checking if some values exist in other Series. Using the first dataframe that we created we can do something like this

```
  let mask3 = ($df | dfr col first | dfr is-in [b c])
$mask3


                                             input       #
   expr       value



                          0   column    first


                        1   literal   Series[list]




                                                      function
  IsIn
                                                  options
FunctionOptions { collect_groups: ApplyFlat, input_wildcard_-
expansion: false, auto_explode: tru              e, fmt_-
str: "", cast_to_supertypes: true, allow_rename: false,
pass_name_to_apply: false }
```

and this new mask can be used to filter the dataframe

```
  $df | dfr filter-with $mask3
  #   int_1   int_2   float_1   float_2   first   second
third    word
 0       4      14      0.40      3.00   b       a
c       second    1       0      15      0.50      4.00
b       a        a       third     2       6      16
0.60      5.00   b       a         a       second    3
  7      17      0.70      6.00   b       c         a
third     4       8      18      0.80      7.00   c
c        b       eight     5       9      19      0.90
  8.00   c       c        b       ninth     6       0
 10      0.00      9.00   c       c        b       ninth
```

Another operation that can be done with masks is setting or replacing a va

:: warning This is example is not updated to recent Nushell versions. :::

```
  $df | dfr get first | dfr set new --mask ($df.first =~
a)          #   string           0   new       1   new
     2   new       3   b         4   b         5   b
    6   b         7   c         8   c         9   c
```

Series as indices

Series can be also used as a way of filtering a dataframe by using them as a list of indices. For example, let's say that we want to get rows 1, 4, and 6 from our original dataframe. With that in mind, we can use the next command to extract that information

```
  let indices = ([1 4 6] | dfr into-df)   $df | dfr take
$indices                                                  #
int_1   int_2   float_1   float_2   first   second   third
   word                                                   0
     2      12      0.20      1.00   a       b        c
    second    1       0      15      0.50      4.00   b
    a        a       third     2       7      17      0.70
    6.00   b       c        a       third
```

The command dfr take[20] is very handy, especially if we mix it with other commands. Let's say that we want to extract all rows for the first duplicated element for column `first`. In order to do that, we can use the command `dfr arg-unique` as shown in the next example

```
  let indices = ($df | dfr get first | dfr arg-unique)
$df | dfr take $indices
  #   int_1   int_2   float_1   float_2   first   second
third    word
  0       1      11      0.10      1.00   a       b
c       first     1       4      14      0.40      3.00
b       a        c       second    2       8      18
0.80      7.00   c       c        b       eight
```

Or what if we want to create a new sorted dataframe using a column

:: tip The same result could be accomplished using the command sort[21] :::

```
  let indices = ($df | dfr get word | dfr arg-sort)  $df
| dfr take $indices
  #   int_1   int_2   float_1   float_2   first   second
third    word
  0       8      18      0.80      7.00   c       c
b       eight     1       1      11      0.10      1.00
a       b        c       first     2       9      19
0.90      8.00   c       c        b       ninth     3
  0      10      0.00      9.00   c       c        b
ninth     4       2      12      0.20      1.00   a
b       c        second    5       4      14      0.40
  3.00   b       a        c       second    6       6
 16      0.60      5.00   b       a        a       second
   7       3      13      0.30      2.00   a       b
  c       third     8       0      15      0.50      4.00
  b       a        a       third     9       7      17
  0.70      6.00   b       c        a       third
```

And finally, we can create new Series by setting a new value in the marked indices. Have a look at the next command

[20] /commands/docs/dfr_take.md
[21] /commands/docs/sort.md

```
  let indices = ([0 2] | dfr into-df);  $df | dfr get int_-
1 | dfr set-with-idx 123 --indices $indices          #
  int_1           0     123    1       2    2     123
3       4    4       0    5       6    6       7    7
   8    8       9    9       0
```

Unique values

Another operation that can be done with `Series` is to search for unique values in a list or column. Lets use again the first dataframe we created to test these operations.

The first and most common operation that we have is `value_-counts`. This command calculates a count of the unique values that exist in a Series. For example, we can use it to count how many occurrences we have in the column `first`

```
  $df | dfr get first | dfr value-counts
#   first   counts                    0   b            4
1   a             3    2   c            3
```

As expected, the command returns a new dataframe that can be used to do more queries.

Continuing with our exploration of `Series`, the next thing that we can do is to only get the unique unique values from a series, like this

```
  $df | dfr get first | dfr unique          #   first
  0   c        1   b        2   a
```

Or we can get a mask that we can use to filter out the rows where data is unique or duplicated. For example, we can select the rows for unique values in column `word`

```
  $df | dfr filter-with ($df | dfr get word | dfr is-unique)
                                                #   int_1   int_-
2   float_1   float_2   first   second   third   word
  0       1      11      0.10      1.00   a       b
c       first    1       8      18      0.80      7.00   c
      c        b       eight

```

Or all the duplicated ones

```
  $df | dfr filter-with ($df | dfr get word | dfr is-duplicated)
                                                      #   int_1
int_2   float_1   float_2   first   second   third    word
                                                        0      2
     12      0.20      1.00   a       b        c       second
   1       3      13      0.30      2.00   a       b
  c       third     2       4      14      0.40      3.00
  b       a         c       second    3       0      15
  0.50      4.00   b       a        a       third     4
    6      16      0.60      5.00   b       a         a
   second    5       7      17      0.70      6.00   b
   c        a       third     6       9      19      0.90
     8.00   c       c        b       ninth     7       0
    10      0.00      9.00   c       c        b      ninth
```

Lazy Dataframes

Lazy dataframes are a way to query data by creating a logical plan. The advantage of this approach is that the plan never gets evaluated until you need to extract data. This way you could chain together aggregations, joins and selections and collect the data once you are happy with the selected operations.

Let's create a small example of a lazy dataframe

```
  let a = ([[a b]; [1 a] [2 b] [3 c] [4 d]] | dfr into-lazy)
   $a                                              plan
            DF ["a", "b"]; PROJECT */2 COLUMNS; SELECTION:
 "None"
                              optimized_plan    DF ["a",
"b"]; PROJECT */2 COLUMNS; SELECTION: "None"
```

As you can see, the resulting dataframe is not yet evaluated, it stays as a set of instructions that can be done on the data. If you were to collect that dataframe you would get the next result

```
  $a | dfr collect          #   a   b            0   1
a    1   2   b    2   3   c    3   4   d
```

as you can see, the collect command executes the plan and creates a nushell table for you.

All dataframes operations should work with eager or lazy dataframes. They are converted in the background for compatibility. However, to take advantage of lazy operations if is recommended to only use lazy operations with lazy dataframes.

To find all lazy dataframe operations you can use

```
$nu.scope.commands | where category =~ lazyframe
```

With your lazy frame defined we can start chaining operations on it. For example this

```
  $a |  dfr reverse |  dfr with-column [   ((dfr col a)
 * 2 | dfr as double_a)   ((dfr col a) / 2 | dfr as half_-
a)  ] | dfr collect                    #   a   b   double_-
a   half_a                      0   4   d         8
   2    1   3   c         6        1    2   2   b
  4        1    3   1   a         2        0
```

:::tip You can use the line buffer editor to format your queries (`ctr + o`) easily :::

This query uses the lazy reverse command to invert the dataframe and the `dfr with-column` command to create new two columns using `expressions`. An `expression` is used to define an operation that is executed on the lazy frame. When put together they create the whole set of instructions used by the lazy commands to query the data. To list all the commands that generate an expression you can use

```
scope commands | where category =~ expression
```

In our previous example, we use the `dfr col` command to indicate that column `a` will be multiplied by 2 and then it will be aliased to the name `double_a`. In some cases the use of the `dfr col` command can be inferred. For example, using the `dfr select` command we can use only a string

```
> $a | dfr select a | dfr collect
```

or the `dfr col` command

```
> $a | dfr select (dfr col a) | dfr collect
```

Let's try something more complicated and create aggregations from a lazy dataframe

```
 let a = ( [[name value]; [one 1] [two 2] [one 1] [two
3]] | dfr into-lazy )  $a |  dfr group-by name |  dfr
agg [   (dfr col value | dfr sum | dfr as sum)   (dfr
col value | dfr mean | dfr as mean)  ] | dfr collect
 #   name   sum   mean                    0   two     5
2.50    1   one     2   1.00
```

And we could join on a lazy dataframe that hasn't being collected. Let's join the resulting group by to the original lazy frame

```
 let a = ( [[name value]; [one 1] [two 2] [one 1] [two
3]] | dfr into-lazy )  let group = ($a  | dfr group-by
name  | dfr agg [   (dfr col value | dfr sum | dfr as
sum)    (dfr col value | dfr mean | dfr as mean)  ])
$a | dfr join $group name name | dfr collect
 #   name   value   sum   mean                       0
one         1     2   1.00    1   two         2     5   2.50
  2   one         1     2   1.00    3   two         3     5
 2.50
```

As you can see lazy frames are a powerful construct that will let you query data using a flexible syntax, resulting in blazing fast results.

Dataframe commands

So far we have seen quite a few operations that can be done using `DataFrame`s commands. However, the commands we have used so far are not all the commands available to work with data and be assured that there will be more as the feature becomes more stable.

The next list shows the available dataframe commands with their des

::: warning This list may be outdated. To get the up-to-date command list, see Dataframe[22] Lazyframe[23] and Dataframe Or Lazyframe[24] command categories. :::

[22] /commands/categories/dataframe.md

[23] /commands/categories/lazyframe.md

[24] /commands/categories/dataframe_or_lazyframe.md

Command Name	Applies To	Description	Nushell Equivalent
aggregate	DataFrame, GroupBy, Series	Performs an aggregation operation on a dataframe, groupby or series object	math
all-false	Series	Returns true if all values are false	
all-true	Series	Returns true if all values are true	all
arg-max	Series	Return index for max value in series	
arg-min	Series	Return index for min value in series	
arg-sort	Series	Returns indexes for a sorted series	
arg-true	Series	Returns indexes where values are true	
arg-unique	Series	Returns indexes for unique values	
count-null	Series	Counts null values	
count-unique	Series	Counts unique value	
drop	DataFrame	Creates a new dataframe by dropping the selected columns	drop
drop-duplicates	DataFrame	Drops duplicate values in dataframe	
drop-nulls	DataFrame, Series	Drops null values in dataframe	
dtypes	DataFrame	Show	

Future of Dataframes

We hope that by the end of this page you have a solid grasp of how to use the dataframe commands. As you can see they offer powerful operations that can help you process data faster and natively.

However, the future of these dataframes is still very experimental. New commands and tools that take advantage of these commands will be added as they mature.

Keep visiting this book in order to check the new things happening to dataframes and how they can help you process data faster and efficiently.

Metadata

In using Nu, you may have come across times where you felt like there was something extra going on behind the scenes. For example, let's say that you try to open a file that Nu supports only to forget and try to convert again:

```
> open Cargo.toml | from toml error: Expected a string
from pipeline - shell:1:18 1 | open Cargo.toml | from toml
  |                   ^^^^^^^^^ requires string input -
 shell:1:5 1 | open Cargo.toml | from toml   |      ---
------- object originates from here
```

The error message tells us not only that what we gave from toml[25] wasn't a string, but also where the value originally came from. How would it know that?

Values that flow through a pipeline in Nu often have a set of additional information, or metadata, attached to them. These are known as tags, like the tags on an item in a store. These tags don't affect the data, but they give Nu a way to improve the experience of working with that data.

Let's run the open[26] command again, but this time, we'll look at the tags it gives back:

```
> metadata (open Cargo.toml)                    span   {record
2 fields}
```

Currently, we track only the span of where values come from. Let's take a closer look at that:

[25] /commands/docs/from_toml.md
[26] /commands/docs/open.md

```
> metadata (open Cargo.toml) | get span              start
  212970    end     212987
```

The span "start" and "end" here refer to where the underline will be in the line. If you count over 5, and then count up to 15, you'll see it lines up with the "Cargo.toml" filename. This is how the error we saw earlier knew what to underline.

Creating your own errors

Using the metadata information, you can create your own custom error messages. Error messages are built of multiple parts:

- The title of the error
- The label of error message, which includes both the text of the label and the span to underline

You can use the `error make`[27] command to create your own error messages. For example, let's say you had your own command called `my-command` and you wanted to give an error back to the caller about something wrong with a parameter that was passed in.

First, you can take the span of where the argument is coming from:

```
let span = (metadata $x).span;
```

Next, you can create an error using the `error make`[28] command. This command takes in a record that describes the error to create:

```
error make {msg: "this is fishy", label: {text: "fish right
here", span: $span } }
```

Together with your custom command, it might look like this:

```
def my-command [x] {     let span = (metadata $x).span;
    error make {          msg: "this is fishy",
label: {              text: "fish right here",
   span: (metadata $x).span         }     } }
```

When called with a value, we'll now see an error message returned:

[27] /commands/docs/error_make.md
[28] /commands/docs/error_make.md

```
> my-command 100 Error:   × this is fishy     [entry #5:
1:1]  1   my-command 100    ·                 ·
          fish right here
```

Parallelism

Nushell now has early support for running code in parallel. This allows you to process elements of a stream using more hardware resources of your computer.

You will notice these commands with their characteristic `par-` naming. Each corresponds to a non-parallel version, allowing you to easily write code in a serial style first, and then go back and easily convert serial scripts into parallel scripts with a few extra characters.

par-each

The most common parallel command is `par-each`[29], a companion to the `each`[30] command.

Like `each`[31], `par-each`[32] works on each element in the pipeline as it comes in, running a block on each. Unlike `each`[33], `par-each`[34] will do these operations in parallel.

Let's say you wanted to count the number of files in each sub-directory of the current directory. Using `each`[35], you could write this as:

```
> ls | where type == dir | each { |it|     { name: $it.
name, len: (ls $it.name | length) } }
```

We create a record for each entry, and fill it with the name of the directory and the count of entries in that sub-directory.

On your machine, the times may vary. For this machine, it took 21 milliseconds for the current directory.

Now, since this operation can be run in parallel, let's convert the above to parallel by changing `each`[36] to `par-each`[37]:

[29] /commands/docs/par-each.md
[30] /commands/docs/each.md
[31] /commands/docs/each.md
[32] /commands/docs/par-each.md
[33] /commands/docs/each.md
[34] /commands/docs/par-each.md
[35] /commands/docs/each.md
[36] /commands/docs/each.md
[37] /commands/docs/par-each.md

```
> ls | where type == dir | par-each { |it|     { name:
$it.name, len: (ls $it.name | length) } }
```

On this machine, it now runs in 6ms. That's quite a difference!

As a side note: Because environment variables are scoped, you can use `par-each`[38] to work in multiple directories in parallel (notice the `cd`[39] command):

```
> ls | where type == dir | par-each { |it|     { name:
$it.name, len: (cd $it.name; ls | length) } }
```

You'll notice, if you look at the results, that they come back in different orders each run (depending on the number of hardware threads on your system). As tasks finish, and we get the correct result, we may need to add additional steps if we want our results in a particular order. For example, for the above, we may want to sort the results by the "name" field. This allows both `each`[40] and `par-each`[41] versions of our script to give the same result.

Plugins

Nu can be extended using plugins. Plugins behave much like Nu's built-in commands, with the added benefit that they can be added separately from Nu itself.

Nu plugins are executables; Nu launches them as needed and communicates with them over stdin, stdout, and stderr[42]. Nu plugins can use either JSON or MSGPACK as their communication encoding.

Adding a plugin

To add a plugin, call the `register`[43] command to tell Nu where to find it. As you do, you'll need to also tell Nushell what encoding the plugin uses.

Please note that the plugin name needs to start with `nu_plugin_`, Nu uses the name prefix to detect plugins.

Linux+macOS:

[38] /commands/docs/par-each.md
[39] /commands/docs/cd.md
[40] /commands/docs/each.md
[41] /commands/docs/par-each.md
[42] https://en.wikipedia.org/wiki/Standard_streams
[43] /commands/docs/register.md

```
> register ./my_plugins/nu_plugin_cool
```

Windows:

```
> register .\my_plugins\nu_plugin_cool.exe
```

When `register`[44] is called:

1. Nu launches the plugin, and waits for the plugin to tell Nu which communication encoding it should use
2. Nu sends it a "Signature" message over stdin
3. The plugin responds via stdout with a message containing its signature (name, description, arguments, flags, and more)
4. Nu saves the plugin signature in the file at `$nu.plugin-path`, so registration is persisted across multiple launches

Once registered, the plugin is available as part of your set of commands:

```
> help commands | where command_type == "plugin"
```

Examples

Nu's main repo contains example plugins that are useful for learning how the plugin protocol works:

- Rust[45]
- Python[46]

Debugging

The simplest way to debug a plugin is to print to stderr; plugins' standard error streams are redirected through Nu and displayed to the user.

Help

Nu's plugin documentation is a work in progress. If you're unsure about something, the #plugins channel on the Nu Discord[47] is a great place to ask questions!

[44] /commands/docs/register.md
[45] https://github.com/nushell/nushell/tree/main/crates/nu_plugin_example
[46] https://github.com/nushell/nushell/blob/main/crates/nu_plugin_python
[47] https://discord.gg/NtAbbGn

More details

The plugin chapter in the contributor book[48] offers more details on the intricacies of how plugins work from a software developer point of view.

`explore`

Explore is a table pager, just like `less` but for table structured data.

Signature

```
> explore --head --index --reverse --peek
```

Parameters

- `--head {boolean}`: turn off column headers
- `--index`: show row indexes (by default it's not showed)
- `--reverse`: start from the last row
- `--peek`: returns a last used value, so it can be used in next pipelines

Get Started

```
ls | explore -i
```

So the main point of `explore`[49] is `:table` (Which you see on the above screenshot).

You can interact with it via `<Left>`, `<Right>`, `<Up>`, `<Down>` *arrow keys*.

You can inspect a underlying values by entering into cursor mode. You can press either `<i>` or `<Enter>` to do so. Then using *arrow keys* you can choose a necessary cell. And you'll be able to see it's underlying structure.

You can obtain more information about the various aspects of it by `:help`.

[48] `/contributor-book/plugins.md`
[49] `/commands/docs/explore.md`

Commands

`explore`[50] has a list of built in commands you can use. Commands are run through pressing `<:>` and then a command name.

To find out the comprehensive list of commands you can type `:help`.

Config

You can configure many things (including styles and colors), via config. You can find an example configuration in `default-config.nu`[51].

Examples

Peeking a value

```
$nu | explore --peek
```

`:try` command

There's an interactive environment which you can use to navigate through data using `nu`.

Keeping the chosen value by `$nu` Remember you can combine it with `--peek`.

Regular expressions

Regular expressions in Nushell's commands are handled by the `rust-lang/regex` crate. If you want to know more, check the crate documentation: "regex[52]".

[50] `/commands/docs/explore.md`
[51] `https://github.com/nushell/nushell/blob/main/crates/nu-utils/src/sample_config/default_config.nu`
[52] `https://github.com/rust-lang/regex`

Chapter 9

Command Reference

To see all commands in Nushell, please run `help commands`

alias

version: 0.90.2

usage:

Alias a command (with optional flags) to a new name.

Signature

```
> alias (name) (initial_value)
```

Parameters

- `name`: Name of the alias.
- `initial_value`: Equals sign followed by value.

Input/output types:

input	output
nothing	nothing

Examples

Alias ll to ls -l

```
> alias ll = ls -l
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

all

version: 0.90.2

usage:

Test if every element of the input fulfills a predicate expression.

Signature

`> all (predicate)`

Parameters

- `predicate`: A closure that must evaluate to a boolean.

Input/output types:

input	output
list<any>	bool

Examples

Check if each row's status is the string 'UP'

```
> [[status]; [UP] [UP]] | all {|el| $el.
status == UP }
```

Check that each item is a string

```
> [ foo bar 2 baz ] | all { || ( $ in | describe)
== ' string ' }
```

Check that all values are equal to twice their index

```
> [ 0 2 4 6 ] | enumerate | all { |i| $ i .item
== $ i .index * 2 }
```

Check that all of the values are even, using a stored closure

```
> let cond = { |el| ( $ el mod 2) == 0 } ; [2 4 6 8]
  | all $ cond
```

ansi

version: 0.90.2

usage:

Output ANSI codes to change color and style of text.

Signature

`> ansi (code) --escape --osc --list`

Parameters

- `code`: The name of the code to use (from `ansi -l`).
- `--escape`: escape sequence without the escape character(s) ('\x1b[' is not required)
- `--osc`: operating system command (osc) escape sequence without the escape character(s) ('\x1b]' is not required)
- `--list`: list available ansi code names

Input/output types:

input	output
nothing	any

Examples

Change color to green (see how the next example text will be green!)

```
> ansi green
```

Reset the color

```
> ansi reset
```

Use different colors and styles in the same text

```
> $'(ansi red_bold)Hello(ansi reset) (ansi green_dimmed)
Nu(ansi reset) (ansi purple_italic)World(ansi reset)'
```

The same example as above with short names

```
> $'(ansi rb)Hello(ansi reset) (ansi gd)Nu(ansi reset)
 (ansi pi)World(ansi reset)'
```

Use escape codes, without the '\x1b['

```
> $"(ansi --escape '3;93;41m')Hello(ansi reset)"   #
 italic bright yellow on red background
```

Use structured escape codes

```
> let bold_blue_on_red = {  # `fg`, `bg`, `attr` are
the acceptable keys, all other keys are considered invalid
and will throw errors.          fg: '#0000ff'
bg: '#ff0000'         attr: b      }     $"(ansi --
escape $bold_blue_on_red)Hello, Nu World!(ansi reset)
"
```

Notes

```
An introduction to what ANSI escape sequences are can be
found in the Wikipedia page of [ANSI escape code](https:
//en.wikipedia.org/wiki/ANSI_escape_code). Escape sequences
usual values:                                    #     type
    normal   bright    name
   0   foreground      30       90   black       1   foreground
      31       91   red          2   foreground      32
    92   green        3   foreground        33       93   yellow
     4   foreground        34       94   blue         5   foreground
      35       95   magenta     5   foreground      35
    95   purple     6   foreground      36       96   cyan
       7   foreground      37       97   white      8
foreground      39            default    9   background
     40      100   black     10   background      41
   101   red        11   background      42      102   green
     12   background      43      103   yellow     13
background      44      104   blue       14   background
     45      105   magenta    14   background      45
   105   purple     15   background      46      106   cyan
     16   background      47      107   white      17
background      49            default
Escape sequences attributes:
  #   id   abbreviation          description
  0    0                 reset / normal display
1    1   b              bold or increased intensity
2    2   d              faint or decreased intensity
3    3   i              italic on (non-mono font)
4    4   u              underline on
5    5   l              slow blink on
6    6                  fast blink on
7    7   r              reverse video on
8    8   h              nondisplayed (invisible) on
9    9   s              strike-through on
Operating system commands:
  #   id                description
  0     0   Set window title and icon name          1
   1   Set icon name                              2    2
Set window title                            3    4   Set/read
color palette                   4    9   iTerm2 Grown notificatio
             5    10   Set foreground color (x11 color
spec)    6    11   Set background color (x11 color spec)
    7   ...   others
```

Subcommands:

name	type	usage
`ansi gradient`[1]	Builtin	Add a color gradient (using ANSI color codes) to the given string.
`ansi link`[2]	Builtin	Add a link (using OSC 8 escape sequence) to the given string.
`ansi strip`[3]	Builtin	Strip ANSI escape sequences from a string.

ansi gradient

version: 0.90.2

usage:

Add a color gradient (using ANSI color codes) to the given string.

Signature

```
> ansi gradient ...rest --fgstart --fgend --bgstart --bgend
```

Parameters

- `...rest`: for a data structure input, add a gradient to strings at the given cell paths
- `--fgstart {string}`: foreground gradient start color in hex (0x123456)
- `--fgend {string}`: foreground gradient end color in hex
- `--bgstart {string}`: background gradient start color in hex
- `--bgend {string}`: background gradient end color in hex

[1] `/commands/docs/ansi_gradient.md`
[2] `/commands/docs/ansi_link.md`
[3] `/commands/docs/ansi_strip.md`

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

draw text in a gradient with foreground start and end colors

```
> ' Hello, Nushell! This is a gradient. ' | ansi
gradient --fgstart ' 0x40c9ff ' --fgend ' 0xe81cff
'
```

draw text in a gradient with foreground start and end colors and background start and end colors

```
> ' Hello, Nushell! This is a gradient. ' | ansi
gradient --fgstart ' 0x40c9ff ' --fgend ' 0xe81cff
' --bgstart ' 0xe81cff ' --bgend ' 0x40c9ff '
```

draw text in a gradient by specifying foreground start color - end color is assumed to be black

```
> ' Hello, Nushell! This is a gradient. ' | ansi
gradient --fgstart ' 0x40c9ff '
```

draw text in a gradient by specifying foreground end color - start color is assumed to be black

```
> ' Hello, Nushell! This is a gradient. ' | ansi
gradient --fgend ' 0xe81cff '
```

ansi link

version: 0.90.2

usage:

Add a link (using OSC 8 escape sequence) to the given string.

Signature

```
> ansi link ...rest --text
```

Parameters

- `...rest`: For a data structure input, add links to all strings at the given cell paths.
- `--text {string}`: Link text. Uses uri as text if absent. In case of tables, records and lists applies this text to all elements

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

Create a link to open some file

```
>  'file:///file.txt'  | ansi link --text 'Open
Me!'
```

Create a link without text

```
>  'https://www.nushell.sh/'  | ansi link
```

Format a table column into links

```
>  [[url text]; [https://example.com Text]]  |
ansi link url
```

ansi strip

version: 0.90.2

usage:

Strip ANSI escape sequences from a string.

Signature

```
> ansi strip ...rest
```

Parameters

- `...rest`: For a data structure input, remove ANSI sequences from strings at the given cell paths.

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

Strip ANSI escape sequences from a string

```
> $'(ansi green)(ansi cursor_on)hello' | ansi
strip
```

any

version: 0.90.2

usage:

Tests if any element of the input fulfills a predicate expression.

Signature

```
> any (predicate)
```

Parameters

- `predicate`: A closure that must evaluate to a boolean.

Input/output types:

input	output
list<any>	bool

Examples

Check if any row's status is the string 'DOWN'

```
> [[status]; [UP] [DOWN] [UP]] | any {
|el| $el.status == DOWN }
```

Check that any item is a string

```
> [1 2 3 4] | any {|| ($in | describe) == '
string' }
```

Check if any value is equal to twice its own index

```
> [9 8 7 6] | enumerate | any {|i| $i.item
== $i.index * 2 }
```

Check if any of the values are odd, using a stored closure

```
> let cond = {|e| $e mod 2 == 1 }; [2 4 1 6 8]
| any $cond
```

append

version: 0.90.2

usage:

Append any number of rows to a table.

Signature

```
> append (row)
```

Parameters

- `row`: The row, list, or table to append.

Input/output types:

input	output
any	list<any>

Examples

Append one int to a list

```
> [0 1 2 3] | append 4
```

Append a list to an item

```
> 0 | append [1 2 3]
```

Append a list of string to a string

```
> "a" | append ["b"]
```

Append three int items

```
> [0 1] | append [2 3 4]
```

Append ints and strings

```
> [0 1] | append [2 nu 4 shell]
```

Append a range of ints to a list

```
> [0 1] | append 2..4
```

Notes

```
Be aware that this command 'unwraps' lists passed to it.
 So, if you pass a variable to it, and you want the variable's
contents to be appended without being unwrapped, it's wise
to pre-emptively wrap the variable in a list, like so:
`append [$val]`. This way, `append` will only unwrap the
outer list, and leave the variable's contents untouched.
```

ast

version: 0.90.2

usage:

Print the abstract syntax tree (ast) for a pipeline.

Signature

```
> ast (pipeline) --json --minify
```

Parameters

- `pipeline`: The pipeline to print the ast for.
- `--json`: serialize to json
- `--minify`: minify the nuon or json output

Input/output types:

input	output
string	record

Examples

Print the ast of a string

```
> ast 'hello'
```

Print the ast of a pipeline

```
> ast 'ls | where name =~ README'
```

Print the ast of a pipeline with an error

```
> ast 'for x in 1..10 { echo $x '
```

Print the ast of a pipeline with an error, as json, in a nushell table

```
> ast 'for x in 1..10 { echo $x ' --json | get
block | from json
```

Print the ast of a pipeline with an error, as json, minified

```
> ast 'for x in 1..10 { echo $x ' --json --minify
```

bits

version: 0.90.2

usage:

Various commands for working with bits.

Signature

```
> bits
```

Input/output types:

input	output
nothing	string

Notes

> ```
> You must use one of the following subcommands. Using this
> command as-is will only produce this help message.
> ```

Subcommands:

name	type	usage
`bits and`[4]	Builtin	Performs bitwise and for ints.
`bits not`[5]	Builtin	Performs logical negation on each bit.
`bits or`[6]	Builtin	Performs bitwise or for ints.
`bits rol`[7]	Builtin	Bitwise rotate left for ints.
`bits ror`[8]	Builtin	Bitwise rotate right for ints.
`bits shl`[9]	Builtin	Bitwise shift left for ints.
`bits shr`[10]	Builtin	Bitwise shift right for ints.
`bits xor`[11]	Builtin	Performs bitwise xor for ints.

bits and

version: 0.90.2

usage:

Performs bitwise and for ints.

Signature

```
> bits and (target)
```

[4] /commands/docs/bits_and.md
[5] /commands/docs/bits_not.md
[6] /commands/docs/bits_or.md
[7] /commands/docs/bits_rol.md
[8] /commands/docs/bits_ror.md
[9] /commands/docs/bits_shl.md
[10] /commands/docs/bits_shr.md
[11] /commands/docs/bits_xor.md

Parameters

- `target`: target int to perform bit and

Input/output types:

input	output
int	int
list<int>	list<int>

Examples

Apply bits and to two numbers

```
> 2 | bits and 2
```

Apply logical and to a list of numbers

```
> [4 3 2] | bits and 2
```

bits not

version: 0.90.2

usage:

Performs logical negation on each bit.

Signature

```
> bits not --signed --number-bytes
```

Parameters

- `--signed`: always treat input number as a signed number
- `--number-bytes {string}`: the size of unsigned number in bytes, it can be 1, 2, 4, 8, auto

Input/output types:

input	output
int	int
list<int>	list<int>

Examples

Apply logical negation to a list of numbers

```
> [4 3 2] | bits not
```

Apply logical negation to a list of numbers, treat input as 2 bytes number

```
> [4 3 2] | bits not --number-bytes '2'
```

Apply logical negation to a list of numbers, treat input as signed number

```
> [4 3 2] | bits not --signed
```

bits or

version: 0.90.2

usage:

Performs bitwise or for ints.

Signature

`> bits or (target)`

Parameters

- `target`: target int to perform bit or

Input/output types:

input	output
int	int
list<int>	list<int>

Examples

Apply bits or to two numbers

```
> 2 | bits or 6
```

Apply logical or to a list of numbers

```
> [8 3 2] | bits or 2
```

bits rol

version: 0.90.2

usage:

Bitwise rotate left for ints.

Signature

`> bits rol (bits) --signed --number-bytes`

Parameters

- `bits`: number of bits to rotate left
- `--signed`: always treat input number as a signed number
- `--number-bytes {string}`: the word size in number of bytes, it can be 1, 2, 4, 8, auto, default value `8`

Input/output types:

input	output
int	int
list<int>	list<int>

Examples

Rotate left a number with 2 bits

```
> 17 | bits rol 2
```

Rotate left a list of numbers with 2 bits

```
> [5 3 2] | bits rol 2
```

bits ror

version: 0.90.2

usage:

Bitwise rotate right for ints.

Signature

```
> bits ror (bits) --signed --number-bytes
```

Parameters

- `bits`: number of bits to rotate right
- `--signed`: always treat input number as a signed number
- `--number-bytes {string}`: the word size in number of bytes, it can be 1, 2, 4, 8, auto, default value `8`

Input/output types:

input	output
int	int
list<int>	list<int>

Examples

Rotate right a number with 60 bits

```
> 17 | bits ror 60
```

Rotate right a list of numbers of one byte

```
> [15 33 92] | bits ror 2 --number-bytes '1'
```

bits shl

version: 0.90.2

usage:

Bitwise shift left for ints.

Signature

```
> bits shl (bits) --signed --number-bytes
```

Parameters

- `bits`: number of bits to shift left
- `--signed`: always treat input number as a signed number
- `--number-bytes {string}`: the word size in number of bytes, it can be 1, 2, 4, 8, auto, default value `8`

Input/output types:

input	output
int	int
list<int>	list<int>

Examples

Shift left a number by 7 bits

```
> 2 | bits shl 7
```

Shift left a number with 1 byte by 7 bits

```
> 2 | bits shl 7 --number-bytes '1'
```

Shift left a signed number by 1 bit

```
> 0x7F | bits shl 1 --signed
```

Shift left a list of numbers

```
> [5 3 2] | bits shl 2
```

bits shr

version: 0.90.2

usage:

Bitwise shift right for ints.

Signature

`> bits shr (bits) --signed --number-bytes`

Parameters

- `bits`: number of bits to shift right
- `--signed`: always treat input number as a signed number
- `--number-bytes {string}`: the word size in number of bytes, it can be 1, 2, 4, 8, auto, default value `8`

Input/output types:

input	output
int	int
list<int>	list<int>

Examples

Shift right a number with 2 bits

```
> 8 | bits shr 2
```

Shift right a list of numbers

```
> [15 35 2] | bits shr 2
```

bits xor

version: 0.90.2

usage:

Performs bitwise xor for ints.

Signature

`> bits xor (target)`

Parameters

- `target`: target int to perform bit xor

Input/output types:

input	output
int	int
list<int>	list<int>

Examples

Apply bits xor to two numbers

```
> 2 | bits xor 2
```

Apply logical xor to a list of numbers

```
> [8 3 2] | bits xor 2
```

break

version: 0.90.2

usage:

Break a loop.

Signature

```
> break
```

Input/output types:

input	output
nothing	nothing

Examples

Break out of a loop

```
> loop { break }
```

bytes

version: 0.90.2

usage:

Various commands for working with byte data.

Signature

```
> bytes
```

Input/output types:

input	output
nothing	string

Notes

> ```
> You must use one of the following subcommands. Using this
> command as-is will only produce this help message.
> ```

Subcommands:

name	type	usage
`bytes add`[12]	Builtin	Add specified bytes to the input.
`bytes at`[13]	Builtin	Get bytes defined by a range.
`bytes build`[14]	Builtin	Create bytes from the arguments.
`bytes collect`[15]	Builtin	Concatenate multiple binary into a single binary, with an optional separator between each.
`bytes ends-with`[16]	Builtin	Check if bytes ends with a pattern.
`bytes index-of`[17]	Builtin	Returns start index of first occurrence of pattern in bytes, or -1 if no match.
`bytes length`[18]	Builtin	Output the length of any bytes in the pipeline.
`bytes remove`[19]	Builtin	Remove bytes.
`bytes replace`[20]	Builtin	Find and replace binary.
`bytes reverse`[21]	Builtin	Reverse the bytes in the pipeline.
`bytes starts-with`[22]	Builtin	Check if bytes starts with a pattern.

[12] `/commands/docs/bytes_add.md`
[13] `/commands/docs/bytes_at.md`
[14] `/commands/docs/bytes_build.md`
[15] `/commands/docs/bytes_collect.md`
[16] `/commands/docs/bytes_ends-with.md`
[17] `/commands/docs/bytes_index-of.md`
[18] `/commands/docs/bytes_length.md`
[19] `/commands/docs/bytes_remove.md`
[20] `/commands/docs/bytes_replace.md`
[21] `/commands/docs/bytes_reverse.md`
[22] `/commands/docs/bytes_starts-with.md`

bytes add

version: 0.90.2

usage:

Add specified bytes to the input.

Signature

```
> bytes add (data) ...rest --index --end
```

Parameters

- `data`: The binary to add.
- `...rest`: For a data structure input, add bytes to the data at the given cell paths.
- `--index {int}`: index to insert binary data
- `--end`: add to the end of binary

Input/output types:

input	output
binary	binary
list<binary>	list<binary>
record	record
table	table

Examples

Add bytes `0x[AA]` to `0x[1F FF AA AA]`

```
> 0x[1F FF AA AA] | bytes add 0x[AA]
```

Add bytes `0x[AA BB]` to `0x[1F FF AA AA]` at index 1

```
> 0x[1F FF AA AA] | bytes add 0x[AA BB] --
index 1
```

Add bytes `0x[11]` to `0x[FF AA AA]` at the end

```
> 0x[FF AA AA] | bytes add 0x[11] --end
```

Add bytes `0x[11 22 33]` to `0x[FF AA AA]` at the end, at index 1(the index is start from end)

```
> 0x[FF AA BB] | bytes add 0x[11 22 33] --end
 --index 1
```

bytes at

version: 0.90.2

usage:

Get bytes defined by a range.

Signature

`> bytes at (range) ...rest`

Parameters

- `range`: The range to get bytes.
- `...rest`: For a data structure input, get bytes from data at the given cell paths.

Input/output types:

input	output
binary	binary
list<binary>	list<binary>
record	record
table	table

Examples

Get a subbytes `0x[10 01]` from the bytes `0x[33 44 55 10 01 13]`

```
>  0x[33 44 55 10 01 13] | bytes at 3..<4
```

Get a subbytes `0x[10 01 13]` from the bytes `0x[33 44 55 10 01 13]`

```
>   0 x [ 33 44 55 10 01 13 ]  |  bytes  at 3..6
```

Get the remaining characters from a starting index

```
>   {  data: 0x [ 33 44 55 10 01 13 ]  }  |  bytes  at
3.. data
```

Get the characters from the beginning until ending index

```
>   0 x [ 33 44 55 10 01 13 ]  |  bytes  at .. < 4
```

Or the characters from the beginning until ending index inside a table

```
>   [ [ ColA ColB ColC ] ;  [ 0x [ 11 12 13 ]  0x [ 14 15
16 ]  0x [ 17 18 19 ] ] ]  |  bytes  at 1.. ColB ColC
```

bytes build

version: 0.90.2

usage:

Create bytes from the arguments.

Signature

`> bytes build ...rest`

Parameters

- `...rest`: List of bytes.

Input/output types:

input	output
nothing	binary

Examples

Builds binary data from 0x[01 02], 0x[03], 0x[04]

```
> bytes build 0x [ 01 02 ] 0x [ 03 ] 0x [ 04 ]
```

bytes collect

version: 0.90.2

usage:

Concatenate multiple binary into a single binary, with an optional separator between each.

Signature

```
> bytes collect (separator)
```

Parameters

- `separator`: Optional separator to use when creating binary.

Input/output types:

input	output
list<binary>	binary

Examples

Create a byte array from input

```
> [ 0x [ 11 ] 0x [ 13 15 ] ] | bytes collect
```

Create a byte array from input with a separator

```
> [ 0x [ 11 ] 0x [ 33 ] 0x [ 44 ] ] | bytes collect
0x [ 01 ]
```

bytes ends-with

version: 0.90.2

usage:

Check if bytes ends with a pattern.

Signature

```
> bytes ends-with (pattern) ...rest
```

Parameters

- `pattern`: The pattern to match.
- `...rest`: For a data structure input, check if bytes at the given cell paths end with the pattern.

Input/output types:

input	output
binary	bool
record	record
table	table

Examples

Checks if binary ends with `0x[AA]`

```
> 0x[1F FF AA AA] | bytes ends-with 0x[AA]
```

Checks if binary ends with `0x[FF AA AA]`

```
> 0x[1F FF AA AA] | bytes ends-with 0x[FF AA AA
]
```

Checks if binary ends with `0x[11]`

```
> 0x[1F FF AA AA] | bytes ends-with 0x[11]
```

bytes index-of

version: 0.90.2

usage:

Returns start index of first occurrence of pattern in bytes, or -1 if no match.

Signature

```
> bytes index-of (pattern) ...rest --all --end
```

Parameters

- `pattern`: The pattern to find index of.
- `...rest`: For a data structure input, find the indexes at the given cell paths.
- `--all`: returns all matched index
- `--end`: search from the end of the binary

Input/output types:

input	output
binary	any
record	record
table	table

Examples

Returns index of pattern in bytes

```
> 0x[33 44 55 10 01 13 44 55] | bytes index-of
0x[44 55]
```

Returns index of pattern, search from end

```
> 0x[33 44 55 10 01 13 44 55] | bytes index-of
--end 0x[44 55]
```

Returns all matched index

```
> 0x[33 44 55 10 01 33 44 33 44] | bytes index-
of --all 0x[33 44]
```

Returns all matched index, searching from end

```
> 0x[33 44 55 10 01 33 44 33 44] | bytes index-
of --all --end 0x[33 44]
```

Returns index of pattern for specific column

```
>   [ [ ColA ColB ColC ] ;  [ 0x [ 11 12 13 ]  0x [ 14 15
16 ]  0x [ 17 18 19 ] ] ]   |   bytes  index-of 0x [ 11 ]
ColA ColC
```

bytes length

version: 0.90.2

usage:

Output the length of any bytes in the pipeline.

Signature

```
> bytes length ...rest
```

Parameters

- `...rest`: For a data structure input, find the length of data at the given cell paths.

Input/output types:

input	output
binary	int
list<binary>	list<int>
record	record
table	table

Examples

Return the length of a binary

```
>   0 x [ 1F FF AA AB ]   |   bytes  length
```

Return the lengths of multiple binaries

```
>   [ 0x [ 1F FF AA AB ]  0x [ 1F ] ]   |   bytes  length
```

bytes remove

version: 0.90.2

usage:

Remove bytes.

Signature

```
> bytes remove (pattern) ...rest --end --all
```

Parameters

- `pattern`: The pattern to find.
- `...rest`: For a data structure input, remove bytes from data at the given cell paths.
- `--end`: remove from end of binary
- `--all`: remove occurrences of finding binary

Input/output types:

input	output
binary	binary
record	record
table	table

Examples

Remove contents

```
>  0x[10 AA FF AA FF] | bytes remove 0x[10 AA]
```

Remove all occurrences of find binary in record field

```
> { data: 0x[10 AA 10 BB 10] } | bytes remove
--all 0x[10] data
```

Remove occurrences of find binary from end

```
> 0x[10 AA 10 BB CC AA 10] | bytes remove --end
 0x[10]
```

Remove find binary from end not found

```
>  0x[10 AA 10 BB CC AA 10]  |  bytes  remove  --end
 0x[11]
```

Remove all occurrences of find binary in table

```
>  [[ColA ColB ColC]; [0x[11 12 13] 0x[14 15 16
] 0x[17 18 19]]]  |  bytes  remove 0x[11]  ColA
ColC
```

bytes replace

version: 0.90.2

usage:

Find and replace binary.

Signature

`> bytes replace (find) (replace) ...rest --all`

Parameters

- `find`: The pattern to find.
- `replace`: The replacement pattern.
- `...rest`: For a data structure input, replace bytes in data at the given cell paths.
- `--all`: replace all occurrences of find binary

Input/output types:

input	output
binary	binary
record	record
table	table

Examples

Find and replace contents

```
> 0 x [ 10 AA FF AA FF ] | bytes replace 0x [ 10 AA ]
 0x [ FF ]
```

Find and replace all occurrences of find binary

```
> 0 x [ 10 AA 10 BB 10 ] | bytes replace -- all 0x
[ 10 ] 0x [ A0 ]
```

Find and replace all occurrences of find binary in table

```
> [ [ ColA ColB ColC ] ; [ 0x [ 11 12 13 ] 0x [ 14 15 16
] 0x [ 17 18 19 ] ] ] | bytes replace -- all 0x [ 11
] 0x [ 13 ] ColA ColC
```

bytes reverse

version: 0.90.2

usage:

Reverse the bytes in the pipeline.

Signature

`> bytes reverse ...rest`

Parameters

- `...rest`: For a data structure input, reverse data at the given cell paths.

Input/output types:

input	output
binary	binary
record	record
table	table

Examples

Reverse bytes `0x[1F FF AA AA]`

```
> 0x[1F FF AA AA] | bytes reverse
```

Reverse bytes `0x[FF AA AA]`

```
> 0x[FF AA AA] | bytes reverse
```

bytes starts-with

version: 0.90.2

usage:

Check if bytes starts with a pattern.

Signature

`> bytes starts-with (pattern) ...rest`

Parameters

- `pattern`: The pattern to match.
- `...rest`: For a data structure input, check if bytes at the given cell paths start with the pattern.

Input/output types:

input	output
binary	bool
record	record
table	table

Examples

Checks if binary starts with `0x[1F FF AA]`

```
> 0x[1F FF AA AA] | bytes starts-with 0x[1F FF
AA]
```

Checks if binary starts with `0x[1F]`

```
> 0x[1F FF AA AA] | bytes starts-with 0x[1F]
```

Checks if binary starts with `0x[1F]`

```
> 0x[1F FF AA AA] | bytes starts-with 0x[11]
```

cal

version: 0.90.2

usage:

Display a calendar.

Signature

`> cal --year --quarter --month --full-year --week-start --month-names`

Parameters

- `--year`: Display the year column
- `--quarter`: Display the quarter column
- `--month`: Display the month column
- `--full-year {int}`: Display a year-long calendar for the specified year
- `--week-start {string}`: Display the calendar with the specified day as the first day of the week
- `--month-names`: Display the month names instead of integers

Input/output types:

input	output
nothing	table

Examples

This month's calendar

```
> cal
```

The calendar for all of 2012

```
> cal --full-year 2012
```

This month's calendar with the week starting on monday

```
> cal --week-start monday
```

cd

version: 0.90.2

usage:

Change directory.

Signature

`> cd (path)`

Parameters

- `path`: The path to change to.

Input/output types:

input	output
nothing	nothing
string	nothing

Examples

Change to your home directory

```
> cd ~
```

Change to the previous working directory ($OLDPWD)

```
> cd -
```

char

version: 0.90.2

usage:

Output special characters (e.g., ‘newline’).

Signature

`> char (character) ...rest --list --unicode --integer`

Parameters

- `character`: The name of the character to output.
- `...rest`: Multiple Unicode bytes.
- `--list`: List all supported character names
- `--unicode`: Unicode string i.e. 1f378
- `--integer`: Create a codepoint from an integer

Input/output types:

input	output
nothing	any

Examples

Output newline

```
> char newline
```

List available characters

```
> char -- list
```

Output prompt character, newline and a hamburger menu character

```
> (char prompt ) + (char newline ) + (char hamburger
)
```

Output Unicode character

```
> char --unicode 1f378
```

Create Unicode from integer codepoint values

```
> char --integer (0x60 + 1 ) ( 0x60 + 2 )
```

Output multi-byte Unicode character

```
> char --unicode 1F468 200D 1F466 200D 1F466
```

clear

version: 0.90.2

usage:

Clear the terminal.

Signature

```
> clear
```

Input/output types:

input	output
nothing	nothing

Examples

Clear the terminal

```
> clear
```

collect

version: 0.90.2

usage:

Collect the stream and pass it to a block.

Signature

```
> collect (closure) --keep-env
```

Parameters

- `closure`: The closure to run once the stream is collected.
- `--keep-env`: let the block affect environment variables

Input/output types:

input	output
any	any

Examples

Use the second value in the stream

```
> [1 2 3] | collect { |x| $x.1 }
```

columns

version: 0.90.2

usage:

Given a record or table, produce a list of its columns' names.

Signature

```
> columns
```

Input/output types:

input	output
record	list<string>
table	list<string>

Examples

Get the columns from the record

```
> { acronym:PWD , meaning: ' Print Working Directory
' } | columns
```

Get the columns from the table

```
> [[name,age,grade]; [bill,20,a]] | columns
```

Get the first column from the table

```
> [[name,age,grade]; [bill,20,a]] | columns |
  first
```

Get the second column from the table

```
> [[name,age,grade]; [bill,20,a]] | columns |
  select 1
```

Notes

```
This is a counterpart to `values`, which produces a list
of columns' values.
```

commandline

version: 0.90.2

usage:

View or modify the current command line input buffer.

Signature

`> commandline (cmd) --cursor --cursor-end --append --insert --replace`

Parameters

- `cmd`: the string to perform the operation with
- `--cursor`: Set or get the current cursor position

- `--cursor-end`: Set the current cursor position to the end of the buffer
- `--append`: appends the string to the end of the buffer
- `--insert`: inserts the string into the buffer at the cursor position
- `--replace`: replaces the current contents of the buffer (default)

Input/output types:

input	output
nothing	nothing
string	string

compact

version: 0.90.2

usage:

Creates a table with non-empty rows.

Signature

```
> compact ...rest --empty
```

Parameters

- `...rest`: The columns to compact from the table.
- `--empty`: also compact empty items like "", {}, and []

Input/output types:

input	output
list<any>	list<any>

Examples

Filter out all records where 'Hello' is null

```
> [["Hello" "World"]; [null 3]] | compact
Hello
```

Filter out all records where 'World' is null

```
> [["Hello" "World"]; [null 3]] | compact
World
```

Filter out all instances of null from a list

```
> [1, null, 2] | compact
```

Filter out all instances of null and empty items from a list

```
> [1, null, 2, "", 3, [], 4, {}, 5] | compact
--empty
```

complete

version: 0.90.2

usage:

Capture the outputs and exit code from an external piped in command in a nushell table.

Signature

```
> complete
```

Input/output types:

input	output
any	record

Examples

Run the external command to completion, capturing stdout and exit_-code

```
> ^external arg1 | complete
```

Run external command to completion, capturing, stdout, stderr and exit_code

```
> do { ^external arg1 } | complete
```

Notes

> In order to capture stdout, stderr, and exit_code, externally piped in commands need to be wrapped with `do`

config

version: 0.90.2

usage:

Edit nushell configuration files.

Signature

```
> config
```

Input/output types:

input	output
nothing	string

Notes

> You must use one of the following subcommands. Using this command as-is will only produce this help message.

Subcommands:

name	type	usage
`config env`[23]	Builtin	Edit nu environment configurations.
`config nu`[24]	Builtin	Edit nu configurations.
`config reset`[25]	Builtin	Reset nushell environment configurations to default, and saves old config files in the config location as oldconfig.nu and oldenv.nu.

config env

version: 0.90.2

usage:

Edit nu environment configurations.

Signature

```
> config env --default
```

Parameters

- `--default`: Print default `env.nu` file instead.

Input/output types:

input	output
nothing	any

[23] `/commands/docs/config_env.md`
[24] `/commands/docs/config_nu.md`
[25] `/commands/docs/config_reset.md`

Examples

allow user to open and update nu env

```
> config env
```

allow user to print default `env.nu` file

```
> config env --default ,
```

allow saving the default `env.nu` locally

```
> config env --default | save -f ~/.config/
nushell/default_env.nu
```

config nu

version: 0.90.2

usage:

Edit nu configurations.

Signature

`> config nu --default`

Parameters

- `--default`: Print default `config.nu` file instead.

Input/output types:

input	output
nothing	any

Examples

allow user to open and update nu config

```
> config nu
```

allow user to print default `config.nu` file

```
> config nu --default ,
```

allow saving the default `config.nu` locally

```
> config nu --default | save -f ~/.config/nushell/
default_config.nu
```

config reset

version: 0.90.2

usage:

Reset nushell environment configurations to default, and saves old config files in the config location as oldconfig.nu and oldenv.nu.

Signature

`> config reset --nu --env --without-backup`

Parameters

- `--nu`: reset only nu config, config.nu
- `--env`: reset only env config, env.nu
- `--without-backup`: do not make a backup

Input/output types:

input	output
nothing	nothing

Examples

reset nushell configuration files

```
> config reset
```

const

version: 0.90.2

usage:

Create a parse-time constant.

Signature

`> const (const_name) (initial_value)`

Parameters

- `const_name`: Constant name.
- `initial_value`: Equals sign followed by constant value.

Input/output types:

input	output
nothing	nothing

Examples

Create a new parse-time constant.

```
> const x = 10
```

Create a composite constant value

```
> const x = { a: 10 , b: 20 }
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

continue

version: 0.90.2

usage:

Continue a loop from the next iteration.

Signature

```
> continue
```

Input/output types:

input	output
nothing	nothing

Examples

Continue a loop from the next iteration

```
> for i in 1..10 { if $i == 5 { continue }; print
$i }
```

cp

version: 0.90.2

usage:

Copy files using uutils/coreutils cp.

Signature

```
> cp ...rest --recursive --verbose --force --interactive --
update --progress --no-clobber --preserve --debug
```

Parameters

- `...rest`: Copy SRC file/s to DEST.
- `--recursive`: copy directories recursively
- `--verbose`: explicitly state what is being done
- `--force`: if an existing destination file cannot be opened, remove it and try again (this option is ignored when the -n option is also used). currently not implemented for windows
- `--interactive`: ask before overwriting files

- `--update`: copy only when the SOURCE file is newer than the destination file or when the destination file is missing
- `--progress`: display a progress bar
- `--no-clobber`: do not overwrite an existing file
- `--preserve {list<string>}`: preserve only the specified attributes (empty list means no attributes preserved) if not specified only mode is preserved possible values: mode, ownership (unix only), timestamps, context, link, links, xattr
- `--debug`: explain how a file is copied. Implies -v

Input/output types:

input	output
nothing	nothing

Examples

Copy myfile to dir_b

```
> cp myfile dir_b
```

Recursively copy dir_a to dir_b

```
> cp -r dir_a dir_b
```

Recursively copy dir_a to dir_b, and print the feedbacks

```
> cp -r -v dir_a dir_b
```

Move many files into a directory

```
> cp *.txt dir_a
```

Copy only if source file is newer than target file

```
> cp -u a b
```

Copy file preserving mode and timestamps attributes

```
> cp --preserve [ mode timestamps ] a b
```

Copy file erasing all attributes

```
> cp --preserve [] a b
```

date

version: 0.90.2

usage:

Date-related commands.

Signature

```
> date
```

Input/output types:

input	output
nothing	string

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

Subcommands:

name	type	usage
date format[26]	Builtin	Removed command: use `format date` instead.
date humanize[27]	Builtin	Print a 'humanized' format for the date, relative to now.
date list-timezone[28]	Builtin	List supported time zones.
date now[29]	Builtin	Get the current date.
date to-record[30]	Builtin	Convert the date into a record.
date to-table[31]	Builtin	Convert the date into a structured table.
date to-timezone[32]	Builtin	Convert a date to a given time zone.

date format

version: 0.90.2

usage:

Removed command: use `format date` instead.

Signature

```
> date format (format string) --list
```

Parameters

- `format string`: The desired date format.

[26] /commands/docs/date_format.md
[27] /commands/docs/date_humanize.md
[28] /commands/docs/date_list-timezone.md
[29] /commands/docs/date_now.md
[30] /commands/docs/date_to-record.md
[31] /commands/docs/date_to-table.md
[32] /commands/docs/date_to-timezone.md

- `--list`: lists strftime cheatsheet

Input/output types:

input	output
datetime	string
string	string

date humanize

version: 0.90.2

usage:

Print a 'humanized' format for the date, relative to now.

Signature

```
> date humanize
```

Input/output types:

input	output
datetime	string
string	string

Examples

Print a 'humanized' format for the date, relative to now.

```
> "2021-10-22 20:00:12 +01:00" | date humanize
```

date list-timezone

version: 0.90.2

usage:

List supported time zones.

Signature

```
> date list-timezone
```

Input/output types:

input	output
nothing	table

Examples

Show timezone(s) that contains ‘Shanghai’

```
> date list-timezone | where timezone =~ Shanghai
```

date now

version: 0.90.2

usage:

Get the current date.

Signature

```
> date now
```

Input/output types:

input	output
nothing	datetime

Examples

Get the current date and display it in a given format string.

```
> date now | format date "%Y-%m-%d %H:%M:%S"
```

Get the time duration from 2019-04-30 to now

```
> (date now) - 2019-05-01
```

Get the time duration since a more accurate time

```
> (date now ) - 2019-05-01T04:12:05.20+08:00
```

Get current time in full RFC3339 format with timezone

```
> date now | debug
```

date to-record

version: 0.90.2

usage:

Convert the date into a record.

Signature

```
> date to-record
```

Input/output types:

input	output
datetime	record
string	record

Examples

Convert the current date into a record.

```
> date to-record
```

Convert the current date into a record.

```
> date now | date to-record
```

Convert a date string into a record.

```
> '2020-04-12T22:10:57.123+02:00' | date to-record
```

Convert a date into a record.

```
> '2020-04-12 22:10:57 +0200' | into datetime |
  date to-record
```

date to-table

version: 0.90.2

usage:

Convert the date into a structured table.

Signature

```
> date to-table
```

Input/output types:

input	output
datetime	table
string	table

Examples

Convert the current date into a table.

```
> date to-table
```

Convert the date into a table.

```
> date now | date to-table
```

Convert a given date into a table.

```
> 2020-04-12T22:10:57.000000789+02:00 | date to-
table
```

Convert a given date into a table.

```
> '2020-04-12 22:10:57 +0200' | into datetime |
  date to-table
```

date to-timezone

version: 0.90.2

usage:

Convert a date to a given time zone.

Signature

`> date to-timezone (time zone)`

Parameters

- `time zone`: Time zone description.

Input/output types:

input	output
datetime	datetime
string	datetime

Examples

Get the current date in UTC+05:00

```
> date now | date to-timezone '+0500'
```

Get the current local date

```
> date now | date to-timezone local
```

Get the current date in Hawaii

```
> date now | date to-timezone US/Hawaii
```

Get the current date in Hawaii

```
> "2020-10-10 10:00:00 +02:00" | date to-timezone
"+0500"
```

Get the current date in Hawaii, from a datetime object

```
> "2020-10-10 10:00:00 +02:00" | into datetime |
  date to-timezone "+0500"
```

Notes

```
Use 'date list-timezone' to list all supported time zones.
```

debug

version: 0.90.2

usage:

Debug print the value(s) piped in.

Signature

```
> debug --raw
```

Parameters

- `--raw`: Prints the raw value representation

Input/output types:

input	output
any	string
list<any>	list<string>

Examples

Debug print a string

```
> 'hello' | debug
```

Debug print a list

```
> ['hello'] | debug
```

Debug print a table

```
> [[version patch]; ['0.1.0' false] ['0.1.1'
 true] ['0.2.0' false]] | debug
```

Subcommands:

name	type	usage
`debug info`[33]	Builtin	View process memory info.

debug info

version: 0.90.2

[33] `/commands/docs/debug_info.md`

usage:

View process memory info.

Signature

`> debug info`

Input/output types:

input	output
nothing	record

Examples

View process information

```
> debug info
```

Notes

```
This command is meant for debugging purposes. It shows
you the process information and system memory information.
```

decode

version: 0.90.2

usage:

Decode bytes into a string.

Signature

`> decode (encoding)`

Parameters

- `encoding`: The text encoding to use.

Input/output types:

input	output
binary	string

Examples

Decode the output of an external command

```
> ^cat myfile.q | decode utf-8
```

Decode an UTF-16 string into nushell UTF-8 string

```
> 0x[00 53 00 6F 00 6D 00 65 00 20 00 44 00 61 00
74 00 61] | decode utf-16be
```

Notes

```
Multiple encodings are supported; here are a few: big5,
euc-jp, euc-kr, gbk, iso-8859-1, utf-16, cp1252, latin5
For a more complete list of encodings please refer to the
encoding_rs documentation link at https://docs.rs/encoding_-
rs/latest/encoding_rs/#statics
```

Subcommands:

name	type	usage
`decode base64`[34]	Builtin	Base64 decode a value.
`decode hex`[35]	Builtin	Hex decode a value.

decode base64

version: 0.90.2

usage:

Base64 decode a value.

[34] /commands/docs/decode_base64.md
[35] /commands/docs/decode_hex.md

Signature

```
> decode base64 ...rest --character-set --binary
```

Parameters

- `...rest`: For a data structure input, decode data at the given cell paths.
- `--character-set {string}`: specify the character rules for encoding the input. Valid values are 'standard', 'standard-no-padding', 'url-safe', 'url-safe-no-padding','binhex', 'bcrypt', 'crypt', 'mutf7'
- `--binary`: Output a binary value instead of decoding payload as UTF-8

Input/output types:

input	output
list<string>	list<any>
record	record
string	any
table	table

Examples

Base64 decode a value and output as UTF-8 string

```
> 'U29tZSBEYXRh' | decode base64
```

Base64 decode a value and output as binary

```
> 'U29tZSBEYXRh' | decode base64 --binary
```

Notes

```
Will attempt to decode binary payload as an UTF-8 string
by default. Use the `--binary(-b)` argument to force binary
output.
```

decode hex

version: 0.90.2

usage:

Hex decode a value.

Signature

```
> decode hex ...rest
```

Parameters

- `...rest`: For a data structure input, decode data at the given cell paths

Input/output types:

input	output
list<string>	list<binary>
record	record
string	binary
table	table

Examples

Hex decode a value and output as binary

```
> '0102030A0a0B' | decode hex
```

Whitespaces are allowed to be between hex digits

```
> '01 02  03 0A 0a 0B' | decode hex
```

def

version: 0.90.2

usage:

Define a custom command.

Signature

```
> def (def_name) (params) (block) --env --wrapped
```

Parameters

- `def_name`: Command name.
- `params`: Parameters.
- `block`: Body of the definition.
- `--env`: keep the environment defined inside the command
- `--wrapped`: treat unknown flags and arguments as strings (requires ...rest-like parameter in signature)

Input/output types:

input	output
nothing	nothing

Examples

Define a command and run it

```
> def say-hi [] { echo 'hi' }; say-hi
```

Define a command and run it with parameter(s)

```
> def say-sth [sth: string] { echo $sth }; say-
sth hi
```

Set environment variable by call a custom command

```
> def --env foo [] { $env.BAR = "BAZ" };
foo; $env.BAR
```

Define a custom wrapper for an external command

```
> def --wrapped my-echo [...rest] { echo $rest
}; my-echo spam
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

default

version: 0.90.2

usage:

Sets a default row's column if missing.

Signature

```
> default (default value) (column name)
```

Parameters

- `default value`: The value to use as a default.
- `column name`: The name of the column.

Input/output types:

input	output
any	any

Examples

Give a default 'target' column to all file entries

```
> ls -la | default 'nothing' target
```

Get the env value of `MY_ENV` with a default value 'abc' if not present

```
> $env | get --ignore-errors MY_ENV | default
  'abc'
```

Replace the `null` value in a list

```
> [1, 2, null, 4] | default 3
```

describe

version: 0.90.2

usage:

Describe the type and structure of the value(s) piped in.

Signature

```
> describe --no-collect --detailed --collect-lazyrecords
```

Parameters

- `--no-collect`: do not collect streams of structured data
- `--detailed`: show detailed information about the value
- `--collect-lazyrecords`: collect lazy records

Input/output types:

input	output
any	any

Examples

Describe the type of a string

```
> 'hello' | describe
```

Describe the type of a record in a detailed way

```
> {shell:'true', uwu:true, features: {bugs:false
, multiplatform:true, speed: 10}, fib: [1 1 2 3 5
8], on_save: {|x| print $'Saving ($x)'}, first_
commit: 2019-05-10, my_duration: (4min + 20sec)} |
describe -d
```

Describe the type of a stream with detailed information

```
> [1 2 3] | each {|i| echo $i} | describe -
d
```

Describe a stream of data, collecting it first

```
> [1 2 3] | each {|i| echo $i} | describe
```

Describe the input but do not collect streams

```
> [1 2 3] | each {|i| echo $i} | describe --no-collect
```

detect columns

version: 0.90.2

usage:

Attempt to automatically split text into multiple columns.

Signature

`> detect columns --skip --no-headers --combine-columns`

Parameters

- `--skip {int}`: number of rows to skip before detecting
- `--no-headers`: don't detect headers
- `--combine-columns {range}`: columns to be combined; listed as a range

Input/output types:

input	output
string	table

Examples

Splits string across multiple columns

```
> 'a b c' | detect columns --no-headers
```

```
> $'c1 c2 c3 c4 c5(char nl)a b c d e' | detect columns --combine-columns 0..1
```

Splits a multi-line string into columns with headers detected

```
> $' c1 c2 c3 c4 c5(char nl)a b c d e ' | detect
columns --combine-columns -2..-1
```

Splits a multi-line string into columns with headers detected

```
> $' c1 c2 c3 c4 c5(char nl)a b c d e ' | detect
columns --combine-columns 2..
```

Parse external ls command and combine columns for datetime

```
> ^ls -lh | detect columns --no-headers --skip
1 --combine-columns 5..7
```

dfr

version: 0.90.2

usage:

Operate with data in a dataframe format.

Signature

```
> dfr
```

Input/output types:

input	output
nothing	string

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

Subcommands:

name	type	usage
`dfr agg`[36]	Builtin	Performs a series of aggregations from a group-by.
`dfr agg-groups`[37]	Builtin	Creates an agg_-groups expression.
`dfr all-false`[38]	Builtin	Returns true if all values are false.
`dfr all-true`[39]	Builtin	Returns true if all values are true.
`dfr append`[40]	Builtin	Appends a new dataframe.
`dfr arg-max`[41]	Builtin	Return index for max value in series.
`dfr arg-min`[42]	Builtin	Return index for min value in series.
`dfr arg-sort`[43]	Builtin	Returns indexes for a sorted series.
`dfr arg-true`[44]	Builtin	Returns indexes where values are true.
`dfr arg-unique`[45]	Builtin	Returns indexes for unique values.
`dfr arg-where`[46]	Builtin	Creates an expression that returns the arguments where expression is true.
`dfr as`[47]	Builtin	Creates an alias expression.
`dfr as-date`[48]	Builtin	Converts string to date.
`dfr as-datetime`[49]	Builtin	Converts string to datetime.
`dfr cache`[50]	Builtin	Caches operations in a new LazyFrame.
`dfr col`[51]	Builtin	Creates a named column expression.
`dfr collect`[52]	Builtin	Collect lazy dataframe into eager dataframe.
`dfr columns`[53]	Builtin	Show dataframe columns.
`dfr concat-str`[54]	Builtin	Creates a concat string expression

[37]/commands/docs/dfr_agg-groups.md
[38]/commands/docs/dfr_all-false.md
[39]/commands/docs/dfr_all-true.md
[40]/commands/docs/dfr_append.md
[41]/commands/docs/dfr_arg-max.md
[42]/commands/docs/dfr_arg-min.md
[43]/commands/docs/dfr_arg-sort.md
[44]/commands/docs/dfr_arg-true.md
[45]/commands/docs/dfr_arg-unique.md
[46]/commands/docs/dfr_arg-where.md
[47]/commands/docs/dfr_as.md
[48]/commands/docs/dfr_as-date.md
[49]/commands/docs/dfr_as-datetime.md
[50]/commands/docs/dfr_cache.md
[51]/commands/docs/dfr_col.md
[52]/commands/docs/dfr_collect.md
[53]/commands/docs/dfr_columns.md
[54]/commands/docs/dfr_concat-str.md
[55]/commands/docs/dfr_concatenate.md
[56]/commands/docs/dfr_contains.md
[57]/commands/docs/dfr_count.md
[58]/commands/docs/dfr_count-null.md
[59]/commands/docs/dfr_cumulative.md
[60]/commands/docs/dfr_datepart.md
[61]/commands/docs/dfr_drop.md
[62]/commands/docs/dfr_drop-duplicates.md
[63]/commands/docs/dfr_drop-nulls.md
[64]/commands/docs/dfr_dtypes.md
[65]/commands/docs/dfr_dummies.md
[66]/commands/docs/dfr_explode.md
[67]/commands/docs/dfr_expr-not.md
[68]/commands/docs/dfr_fetch.md
[69]/commands/docs/dfr_fill-nan.md
[70]/commands/docs/dfr_fill-null.md
[71]/commands/docs/dfr_filter.md
[72]/commands/docs/dfr_filter-with.md
[73]/commands/docs/dfr_first.md
[74]/commands/docs/dfr_flatten.md
[75]/commands/docs/dfr_get.md
[76]/commands/docs/dfr_get-day.md
[77]/commands/docs/dfr_get-hour.md
[78]/commands/docs/dfr_get-minute.md
[79]/commands/docs/dfr_get-month.md
[80]/commands/docs/dfr_get-nanosecond.md
[81]/commands/docs/dfr_get-ordinal.md
[82]/commands/docs/dfr_get-second.md
[83]/commands/docs/dfr_get-week.md
[84]/commands/docs/dfr_get-weekday.md
[85]/commands/docs/dfr_get-year.md
[86]/commands/docs/dfr_group-by.md
[87]/commands/docs/dfr_implode.md

[88] /commands/docs/dfr_into-df.md
[89] /commands/docs/dfr_into-lazy.md
[90] /commands/docs/dfr_into-nu.md
[91] /commands/docs/dfr_is-duplicated.md
[92] /commands/docs/dfr_is-in.md
[93] /commands/docs/dfr_is-not-null.md
[94] /commands/docs/dfr_is-null.md
[95] /commands/docs/dfr_is-unique.md
[96] /commands/docs/dfr_join.md
[97] /commands/docs/dfr_last.md
[98] /commands/docs/dfr_lit.md
[99] /commands/docs/dfr_lowercase.md
[100] /commands/docs/dfr_ls.md
[101] /commands/docs/dfr_max.md
[102] /commands/docs/dfr_mean.md
[103] /commands/docs/dfr_median.md
[104] /commands/docs/dfr_melt.md
[105] /commands/docs/dfr_min.md
[106] /commands/docs/dfr_n-unique.md
[107] /commands/docs/dfr_not.md
[108] /commands/docs/dfr_open.md
[109] /commands/docs/dfr_otherwise.md
[110] /commands/docs/dfr_quantile.md
[111] /commands/docs/dfr_query.md
[112] /commands/docs/dfr_rename.md
[113] /commands/docs/dfr_replace.md
[114] /commands/docs/dfr_replace-all.md
[115] /commands/docs/dfr_reverse.md
[116] /commands/docs/dfr_rolling.md
[117] /commands/docs/dfr_sample.md
[118] /commands/docs/dfr_schema.md
[119] /commands/docs/dfr_select.md
[120] /commands/docs/dfr_set.md
[121] /commands/docs/dfr_set-with-idx.md
[122] /commands/docs/dfr_shape.md
[123] /commands/docs/dfr_shift.md
[124] /commands/docs/dfr_slice.md
[125] /commands/docs/dfr_sort-by.md
[126] /commands/docs/dfr_std.md
[127] /commands/docs/dfr_str-lengths.md
[128] /commands/docs/dfr_str-slice.md
[129] /commands/docs/dfr_strftime.md
[130] /commands/docs/dfr_sum.md
[131] /commands/docs/dfr_summary.md
[132] /commands/docs/dfr_take.md
[133] /commands/docs/dfr_to-arrow.md
[134] /commands/docs/dfr_to-avro.md
[135] /commands/docs/dfr_to-csv.md
[136] /commands/docs/dfr_to-jsonl.md
[137] /commands/docs/dfr_to-parquet.md
[138] /commands/docs/dfr_unique.md

dfr agg-groups

version: 0.90.2

usage:

Creates an agg_groups expression.

Signature

```
> dfr agg-groups
```

Input/output types:

input	output
any	any

Examples

```
>
```

dfr agg

version: 0.90.2

usage:

Performs a series of aggregations from a group-by.

Signature

```
> dfr agg ...rest
```

Parameters

- `...rest`: Expression(s) that define the aggregations to be applied

[139] /commands/docs/dfr_uppercase.md
[140] /commands/docs/dfr_value-counts.md
[141] /commands/docs/dfr_var.md
[142] /commands/docs/dfr_when.md
[143] /commands/docs/dfr_with-column.md

Input/output types:

input	output
any	any

Examples

Group by and perform an aggregation

```
> [[a b]; [1 2] [1 4] [2 6] [2 4]] |
  dfr into-df | dfr group-by a | dfr agg
[ (dfr col b | dfr min | dfr as "b_min")
    (dfr col b | dfr max | dfr as "b_max") (
dfr col b | dfr sum | dfr as "b_sum") ]
```

Group by and perform an aggregation

```
> [[a b]; [1 2] [1 4] [2 6] [2 4]]
 | dfr into-lazy | dfr group-by a | dfr
agg [ (dfr col b | dfr min | dfr as "b_min")
          (dfr col b | dfr max | dfr as "b_max")
    (dfr col b | dfr sum | dfr as "b_sum") ]
   | dfr collect
```

dfr all-false

version: 0.90.2

usage:

Returns true if all values are false.

Signature

```
> dfr all-false
```

Input/output types:

input	output
any	any

Examples

Returns true if all values are false

```
> [false false false] | dfr into-df | dfr all-
false
```

Checks the result from a comparison

```
> let s = ([5 6 2 10] | dfr into-df); let
res = ($s > 9); $res | dfr all-false
```

dfr all-true

version: 0.90.2

usage:

Returns true if all values are true.

Signature

```
> dfr all-true
```

Input/output types:

input	output
any	any

Examples

Returns true if all values are true

```
> [true true true] | dfr into-df | dfr all-true
```

Checks the result from a comparison

```
> let s = ([5 6 2 8] | dfr into-df); let
res = ($s > 9); $res | dfr all-true
```

dfr append

version: 0.90.2

usage:

Appends a new dataframe.

Signature

`> dfr append (other) --col`

Parameters

- `other`: dataframe to be appended
- `--col`: appends in col orientation

Input/output types:

input	output
any	any

Examples

Appends a dataframe as new columns

```
> let a = ([[a b]; [1 2] [3 4]] | dfr into-
df);    $a | dfr append $a
```

Appends a dataframe merging at the end of columns

```
> let a = ([[a b]; [1 2] [3 4]] | dfr into-
df);    $a | dfr append $a --col
```

dfr arg-max

version: 0.90.2

usage:

Return index for max value in series.

Signature

`> dfr arg-max`

Input/output types:

input	output
any	any

Examples

Returns index for max value

```
> [1 3 2] | dfr into-df | dfr arg-max
```

dfr arg-min

version: 0.90.2

usage:

Return index for min value in series.

Signature

```
> dfr arg-min
```

Input/output types:

input	output
any	any

Examples

Returns index for min value

```
> [1 3 2] | dfr into-df | dfr arg-min
```

dfr arg-sort

version: 0.90.2

usage:

Returns indexes for a sorted series.

Signature

```
> dfr arg-sort --reverse --nulls-last --maintain-order
```

Parameters

- `--reverse`: reverse order
- `--nulls-last`: nulls ordered last
- `--maintain-order`: maintain order on sorted items

Input/output types:

input	output
any	any

Examples

Returns indexes for a sorted series

```
> [1 2 2 3 3] | dfr into-df | dfr arg-sort
```

Returns indexes for a sorted series

```
> [1 2 2 3 3] | dfr into-df | dfr arg-sort --
reverse
```

dfr arg-true

version: 0.90.2

usage:

Returns indexes where values are true.

Signature

```
> dfr arg-true
```

Input/output types:

input	output
any	any

Examples

Returns indexes where values are true

```
> [ false true false ] | dfr into-df | dfr arg-
true
```

dfr arg-unique

version: 0.90.2

usage:

Returns indexes for unique values.

Signature

```
> dfr arg-unique
```

Input/output types:

input	output
any	any

Examples

Returns indexes for unique values

```
> [ 1 2 2 3 3 ] | dfr into-df | dfr arg-unique
```

dfr arg-where

version: 0.90.2

usage:

Creates an expression that returns the arguments where expression is true.

Signature

```
> dfr arg-where (column name)
```

Parameters

- `column name`: Expression to evaluate

Input/output types:

input	output
any	any

Examples

Return a dataframe where the value match the expression

```
> let df = ( [ [ a b ] ; [ one 1 ] [ two 2 ] [ three 3
] ] | dfr into-df ) ; $ df | dfr select (dfr
arg-where ((dfr col b ) > = 2 ) | dfr as b_arg )
```

dfr as-date

version: 0.90.2

usage:

Converts string to date.

Signature

```
> dfr as-date (format) --not-exact
```

Parameters

- `format`: formatting date string
- `--not-exact`: the format string may be contained in the date (e.g. foo-2021-01-01-bar could match 2021-01-01)

Input/output types:

input	output
any	any

Examples

Converts string to date

```
> [ " 2021-12-30 "  " 2021-12-31 " ]  |  dfr  into-df
| dfr  as-datetime  " %Y-%m-%d "
```

Notes

```
Format example:          "%Y-%m-%d"    => 2021-12-31
     "%d-%m-%Y"    => 31-12-2021          "%Y%m%d"
=> 2021319 (2021-03-19)
```

dfr as-datetime

version: 0.90.2

usage:

Converts string to datetime.

Signature

```
> dfr as-datetime (format) --not-exact
```

Parameters

- `format`: formatting date time string
- `--not-exact`: the format string may be contained in the date (e.g. foo-2021-01-01-bar could match 2021-01-01)

Input/output types:

input	output
any	any

Examples

Converts string to datetime

```
> [ " 2021-12-30 00:00:00 " " 2021-12-31 00:00:00 " ]
| dfr into-df | dfr as-datetime " %Y-%m-%d %H:%M:
%S "
```

Converts string to datetime with high resolutions

```
> [ " 2021-12-30 00:00:00.123456789 " " 2021-12-31
00:00:00.123456789 " ] | dfr into-df | dfr as-
datetime " %Y-%m-%d %H:%M:%S.%9f "
```

Notes

```
Format example:          "%y/%m/%d %H:%M:%S"  => 21/12/
31 12:54:98          "%y-%m-%d %H:%M:%S"  => 2021-12-31
24:58:01          "%y/%m/%d %H:%M:%S"  => 21/12/31 24:58:
01          "%y%m%d %H:%M:%S"    => 210319 23:58:50
    "%Y/%m/%d %H:%M:%S"  => 2021/12/31 12:54:98
 "%Y-%m-%d %H:%M:%S"  => 2021-12-31 24:58:01          "%Y/
%m/%d %H:%M:%S"  => 2021/12/31 24:58:01          "%Y%m%d
%H:%M:%S"    => 20210319 23:58:50          "%FT%H:%M:%S"
       => 2019-04-18T02:45:55          "%FT%H:%M:%S.%6f"
   => microseconds          "%FT%H:%M:%S.%9f"    => nanoseconds
```

dfr as

version: 0.90.2

usage:

Creates an alias expression.

Signature

`> dfr as (Alias name)`

Parameters

- `Alias name`: Alias name for the expression

Input/output types:

input	output
any	any

Examples

Creates and alias expression

```
> dfr col a | dfr as new_a | dfr into-nu
```

dfr cache

version: 0.90.2

usage:

Caches operations in a new LazyFrame.

Signature

```
> dfr cache
```

Input/output types:

input	output
any	any

Examples

Caches the result into a new LazyFrame

```
> [[a b]; [6 2] [4 2] [2 2]] | dfr into-
df | dfr reverse | dfr cache
```

dfr col

version: 0.90.2

usage:

Creates a named column expression.

Signature

`> dfr col (column name)`

Parameters

- `column name`: Name of column to be used

Input/output types:

input	output
any	any

Examples

Creates a named column expression and converts it to a nu object

```
> dfr col a | dfr into-nu
```

dfr collect

version: 0.90.2

usage:

Collect lazy dataframe into eager dataframe.

Signature

`> dfr collect`

Input/output types:

input	output
any	any

Examples

drop duplicates

```
> [[a b]; [1 2] [3 4]] | dfr into-lazy |
dfr collect
```

dfr columns

version: 0.90.2

usage:

Show dataframe columns.

Signature

```
> dfr columns
```

Input/output types:

input	output
any	any

Examples

Dataframe columns

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr columns
```

dfr concat-str

version: 0.90.2

usage:

Creates a concat string expression.

Signature

```
> dfr concat-str (separator) (concat expressions)
```

Parameters

- `separator`: Separator used during the concatenation
- `concat expressions`: Expression(s) that define the string concatenation

Input/output types:

input	output
any	any

Examples

Creates a concat string expression

```
> let df = ( [ [ a b c ] ; [ one two 1 ] [ three four 2
] ] | dfr into-df ) ; $ df | dfr with-column (
(dfr concat-str " - " [ (dfr col a) (dfr col b) ((dfr
col c) * 2) ] ) | dfr as concat )
```

dfr concatenate

version: 0.90.2

usage:

Concatenates strings with other array.

Signature

`> dfr concatenate (other)`

Parameters

- `other`: Other array with string to be concatenated

Input/output types:

input	output
any	any

Examples

Concatenate string

```
> let other = ( [ za xs cd ] | dfr into-df ) ;
[abc abc abc] | dfr into-df | dfr concatenate $
other
```

dfr contains

version: 0.90.2

usage:

Checks if a pattern is contained in a string.

Signature

```
> dfr contains (pattern)
```

Parameters

- `pattern`: Regex pattern to be searched

Input/output types:

input	output
any	any

Examples

Returns boolean indicating if pattern was found

```
> [abc acb acb] | dfr into-df | dfr contains
ab
```

dfr count-null

version: 0.90.2

usage:

Counts null values.

Signature

```
> dfr count-null
```

Input/output types:

input	output
any	any

Examples

Counts null values

```
> let s = ([1 1 0 0 3 3 4] | dfr into-df); (
$s / $s) | dfr count-null
```

dfr count

version: 0.90.2

usage:

Creates a count expression.

Signature

```
> dfr count
```

Input/output types:

input	output
any	any

Examples

```
>
```

dfr cumulative

version: 0.90.2

usage:

Cumulative calculation for a series.

Signature

```
> dfr cumulative (type) --reverse
```

Parameters

- `type`: rolling operation
- `--reverse`: Reverse cumulative calculation

Input/output types:

input	output
any	any

Examples

Cumulative sum for a series

```
> [1 2 3 4 5] | dfr into-df | dfr cumulative sum
```

dfr datepart

version: 0.90.2

usage:

Creates an expression for capturing the specified datepart in a column.

Signature

```
> dfr datepart (Datepart name)
```

Parameters

- `Datepart name`: Part of the date to capture. Possible values are year, quarter, month, week, weekday, day, hour, minute, second, millisecond, microsecond, nanosecond

Input/output types:

input	output
any	any

Examples

Creates an expression to capture the year date part

```
> [["2021-12-30T01:02:03.123456789"]] | dfr
into-df | dfr as-datetime "%Y-%m-%dT%H:%M:%S.%9f"
| dfr with-column [(dfr col datetime | dfr datepart
year | dfr as datetime_year )]
```

Creates an expression to capture multiple date parts

```
> [["2021-12-30T01:02:03.123456789"]] | dfr into-
df | dfr as-datetime "%Y-%m-%dT%H:%M:%S.%9f" |
              dfr with-column [ (dfr col datetime |
dfr datepart year | dfr as datetime_year ),
     (dfr col datetime | dfr datepart month | dfr as datetime_-
month ),                  (dfr col datetime | dfr datepart
day | dfr as datetime_day ),                  (dfr col datetime
| dfr datepart hour | dfr as datetime_hour ),
       (dfr col datetime | dfr datepart minute | dfr as
datetime_minute ),                  (dfr col datetime |
dfr datepart second | dfr as datetime_second ),
         (dfr col datetime | dfr datepart nanosecond |
dfr as datetime_ns ) ]
```

dfr drop-duplicates

version: 0.90.2

usage:

Drops duplicate values in dataframe.

Signature

```
> dfr drop-duplicates (subset) --maintain --last
```

Parameters

- `subset`: subset of columns to drop duplicates
- `--maintain`: maintain order
- `--last`: keeps last duplicate value (by default keeps first)

Input/output types:

input	output
any	any

Examples

drop duplicates

```
> [[a b]; [1 2] [3 4] [1 2]] | dfr into-df | dfr drop-duplicates
```

dfr drop-nulls

version: 0.90.2

usage:

Drops null values in dataframe.

Signature

`> dfr drop-nulls (subset)`

Parameters

- `subset`: subset of columns to drop nulls

Input/output types:

input	output
any	any

Examples

drop null values in dataframe

```
> let df = ( [ [ a b ] ; [ 1 2 ] [ 3 0 ] [ 1 2 ] ] |
 dfr into-df ) ;     let res = ( $ df .b / $ df .b )
;     let a = ( $ df | dfr with - column $ res --
name res ) ;     $ a | dfr drop-nulls
```

drop null values in dataframe

```
> let s = ( [ 1 2 0 0 3 4 ] | dfr into-df ) ;     ( $
s / $ s ) | dfr drop-nulls
```

dfr drop

version: 0.90.2

usage:

Creates a new dataframe by dropping the selected columns.

Signature

`> dfr drop ...rest`

Parameters

- `...rest`: column names to be dropped

Input/output types:

input	output
any	any

Examples

drop column a

```
> [ [ a b ] ; [ 1 2 ] [ 3 4 ] ] | dfr into-df |
dfr drop a
```

dfr dtypes

version: 0.90.2

usage:

Show dataframe data types.

Signature

```
> dfr dtypes
```

Input/output types:

input	output
any	any

Examples

Dataframe dtypes

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr dtypes
```

dfr dummies

version: 0.90.2

usage:

Creates a new dataframe with dummy variables.

Signature

```
> dfr dummies --drop-first
```

Parameters

- `--drop-first`: Drop first row

Input/output types:

input	output
any	any

Examples

Create new dataframe with dummy variables from a dataframe

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr dummies
```

Create new dataframe with dummy variables from a series

```
> [1 2 2 3 3] | dfr into-df | dfr dummies
```

dfr explode

version: 0.90.2

usage:

Explodes a dataframe or creates a explode expression.

Signature

`> dfr explode ...rest`

Parameters

- `...rest`: columns to explode, only applicable for dataframes

Input/output types:

input	output
any	any

Examples

Explode the specified dataframe

```
> [[id name hobbies]; [1 Mercy [Cycling Knitting
]] [2 Bob [Skiing Football]]] | dfr into-df |
  dfr explode hobbies | dfr collect
```

Select a column and explode the values

```
> [[id name hobbies]; [1 Mercy [Cycling Knitting
]] [2 Bob [Skiing Football]]] | dfr into-df |
  dfr select (dfr col hobbies | dfr explode)
```

dfr expr-not

version: 0.90.2

usage:

Creates a not expression.

Signature

```
> dfr expr-not
```

Input/output types:

input	output
any	any

Examples

Creates a not expression

```
> (dfr col a) > 2) | dfr expr-not
```

dfr fetch

version: 0.90.2

usage:

Collects the lazyframe to the selected rows.

Signature

```
> dfr fetch (rows)
```

Parameters

- `rows`: number of rows to be fetched from lazyframe

Input/output types:

input	output
any	any

Examples

Fetch a rows from the dataframe

```
> [[a b]; [6 2] [4 2] [2 2]] | dfr into-
df | dfr fetch 2
```

dfr fill-nan

version: 0.90.2

usage:

Replaces NaN values with the given expression.

Signature

```
> dfr fill-nan (fill)
```

Parameters

- `fill`: Expression to use to fill the NAN values

Input/output types:

input	output
any	any

Examples

Fills the NaN values with 0

```
> [1 2 NaN 3 NaN] | dfr into-df | dfr fill-nan
0
```

Fills the NaN values of a whole dataframe

```
> [[a b]; [0.2 1] [0.1 NaN]] | dfr into-df
  | dfr fill-nan 0
```

dfr fill-null

version: 0.90.2

usage:

Replaces NULL values with the given expression.

Signature

`> dfr fill-null (fill)`

Parameters

- `fill`: Expression to use to fill the null values

Input/output types:

input	output
any	any

Examples

Fills the null values by 0

```
> [1 2 2 3 3] | dfr into-df | dfr shift 2 |
dfr fill-null 0
```

dfr filter-with

version: 0.90.2

usage:

Filters dataframe using a mask or expression as reference.

Signature

```
> dfr filter-with (mask or expression)
```

Parameters

- `mask or expression`: boolean mask used to filter data

Input/output types:

input	output
any	any

Examples

Filter dataframe using a bool mask

```
> let mask = ( [ true false ] | dfr into-df ) ;
 [[a b] ; [1 2] [ 3 4 ] ] | dfr into-df | dfr
filter-with $ mask
```

Filter dataframe using an expression

```
> [ [ a b ] ; [ 1 2 ] [ 3 4 ] ] | dfr into-df |
dfr filter-with ((dfr col a ) > 1 )
```

dfr filter

version: 0.90.2

usage:

Filter dataframe based in expression.

Signature

```
> dfr filter (filter expression)
```

Parameters

- `filter expression`: Expression that define the column selection

Input/output types:

input	output
any	any

Examples

Filter dataframe using an expression

```
> [[a b]; [6 2] [4 2] [2 2]] | dfr into-
df | dfr filter ((dfr col a) >= 4)
```

dfr first

version: 0.90.2

usage:

Show only the first number of rows or create a first expression

Signature

```
> dfr first (rows)
```

Parameters

- `rows`: starting from the front, the number of rows to return

Input/output types:

input	output
any	any

Examples

Return the first row of a dataframe

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr first
```

Return the first two rows of a dataframe

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr first 2
```

Creates a first expression from a column

```
> dfr col a | dfr first
```

dfr flatten

version: 0.90.2

usage:

An alias for dfr explode.

Signature

`> dfr flatten ...rest`

Parameters

- `...rest`: columns to flatten, only applicable for dataframes

Input/output types:

input	output
any	any

Examples

Flatten the specified dataframe

```
> [[id name hobbies]; [1 Mercy [Cycling Knitting
]] [2 Bob [Skiing Football]]] | dfr into-df |
 dfr flatten hobbies | dfr collect
```

Select a column and flatten the values

```
> [ [ id name hobbies ] ; [ 1 Mercy [ Cycling Knitting
] ] [ 2 Bob [ Skiing Football ] ] ] | dfr into-df |
  dfr select (dfr col hobbies | dfr flatten )
```

dfr get-day

version: 0.90.2

usage:

Gets day from date.

Signature

```
> dfr get-day
```

Input/output types:

input	output
any	any

Examples

Returns day from a date

```
> let dt = ( ' 2020-08-04T16:39:18+00:00 ' | into
datetime -- timezone ' UTC ' ) ; let df = ( [ $ dt
  $ dt ] | dfr into - df ) ; $ df | dfr get-day
```

dfr get-hour

version: 0.90.2

usage:

Gets hour from date.

Signature

```
> dfr get-hour
```

Input/output types:

input	output
any	any

Examples

Returns hour from a date

```
> let dt = ( ' 2020-08-04T16:39:18+00:00 '   |   into
datetime  -- timezone   ' UTC ' ) ;      let  df  =   ( [ $ dt
  $ dt ]  |  dfr into - df ) ;      $ df   |   dfr  get-hour
```

dfr get-minute

version: 0.90.2

usage:

Gets minute from date.

Signature

```
> dfr get-minute
```

Input/output types:

input	output
any	any

Examples

Returns minute from a date

```
> let dt = ( ' 2020-08-04T16:39:18+00:00 '   |   into
datetime  -- timezone   ' UTC ' ) ;      let  df  =   ( [ $
dt   $ dt ]  |  dfr into - df ) ;      $ df   |   dfr  get-
minute
```

dfr get-month

version: 0.90.2

usage:

Gets month from date.

Signature

```
> dfr get-month
```

Input/output types:

input	output
any	any

Examples

Returns month from a date

```
> let dt = ( ' 2020-08-04T16:39:18+00:00 '  |  into
datetime -- timezone  ' UTC ' ) ;    let df  =  ( [ $ dt
 $ dt ]  |  dfr into - df ) ;    $ df  |  dfr  get-month
```

dfr get-nanosecond

version: 0.90.2

usage:

Gets nanosecond from date.

Signature

```
> dfr get-nanosecond
```

Input/output types:

input	output
any	any

Examples

Returns nanosecond from a date

```
> let dt = ( ' 2020-08-04T16:39:18+00:00 '   |   into
datetime  -- timezone   ' UTC ' ) ;      let  df  =   ( [ $
dt   $ dt ]  |  dfr into - df ) ;      $ df  |   dfr  get-
nanosecond
```

dfr get-ordinal

version: 0.90.2

usage:

Gets ordinal from date.

Signature

`> dfr get-ordinal`

Input/output types:

input	output
any	any

Examples

Returns ordinal from a date

```
> let dt = ( ' 2020-08-04T16:39:18+00:00 '   |   into
datetime  -- timezone   ' UTC ' ) ;      let  df  =   ( [ $
dt   $ dt ]  |  dfr into - df ) ;      $ df  |   dfr  get-
ordinal
```

dfr get-second

version: 0.90.2

usage:

Gets second from date.

Signature

```
> dfr get-second
```

Input/output types:

input	output
any	any

Examples

Returns second from a date

```
> let dt = ( ' 2020-08-04T16:39:18+00:00 '   |   into
datetime  -- timezone   ' UTC ' ) ;      let  df  =   ( [ $
dt   $ dt ]  |  dfr into - df ) ;      $ df   |   dfr  get-
second
```

dfr get-week

version: 0.90.2

usage:

Gets week from date.

Signature

```
> dfr get-week
```

Input/output types:

input	output
any	any

Examples

Returns week from a date

```
> let dt = ( ' 2020-08-04T16:39:18+00:00 '   |   into
datetime  -- timezone  ' UTC ' ) ;      let  df  =   ( [ $ dt
  $ dt ]  |  dfr into - df ) ;      $ df   |   dfr  get-week
```

dfr get-weekday

version: 0.90.2

usage:

Gets weekday from date.

Signature

```
> dfr get-weekday
```

Input/output types:

input	output
any	any

Examples

Returns weekday from a date

```
> let dt = ( ' 2020-08-04T16:39:18+00:00 '  |  into
datetime  -- timezone  ' UTC ' ) ;     let  df  =  ( [ $
dt   $ dt ]  |  dfr into - df ) ;     $ df   |   dfr  get-
weekday
```

dfr get-year

version: 0.90.2

usage:

Gets year from date.

Signature

```
> dfr get-year
```

Input/output types:

input	output
any	any

Examples

Returns year from a date

```
> let dt = ( ' 2020-08-04T16:39:18+00:00 '  |  into
datetime  --timezone  ' UTC ' ) ;    let  df  =  ( [ $ dt
 $ dt ]  |  dfr into - df ) ;    $ df  |  dfr  get-year
```

dfr get

version: 0.90.2

usage:

Creates dataframe with the selected columns.

Signature

`> dfr get ...rest`

Parameters

- `...rest`: column names to sort dataframe

Input/output types:

input	output
any	any

Examples

Returns the selected column

```
>  [ [ a b ] ;  [ 1 2 ]  [ 3 4 ] ]  |  dfr  into-df  |
dfr  get a
```

dfr group-by

version: 0.90.2

usage:

Creates a group-by object that can be used for other aggregations.

Signature

```
> dfr group-by ...rest
```

Parameters

- `...rest`: Expression(s) that define the lazy group-by

Input/output types:

input	output
any	any

Examples

Group by and perform an aggregation

```
> [[a b]; [1 2] [1 4] [2 6] [2 4]]     |
  dfr into-df      |  dfr group-by a      |  dfr agg
[          (dfr col b | dfr min | dfr as "b_min")
     (dfr col b | dfr max | dfr as  "b_max")          (
dfr col b | dfr sum | dfr as  "b_sum")       ]
```

Group by and perform an aggregation

```
> [[a b]; [1 2] [1 4] [2 6] [2 4]]
 |  dfr into-lazy      |  dfr group-by a      |  dfr
agg [           (dfr col b | dfr min | dfr as "b_min")
         (dfr col b | dfr max | dfr as "b_max")
    (dfr col b | dfr sum | dfr as "b_sum")       ]
  |  dfr collect
```

dfr implode

version: 0.90.2

usage:

Aggregates a group to a Series.

Signature

```
> dfr implode
```

Input/output types:

input	output
any	any

Examples

```
>
```

dfr into-df

version: 0.90.2

usage:

Converts a list, table or record into a dataframe.

Signature

```
> dfr into-df --schema
```

Parameters

- `--schema {record}`: Polars Schema in format [{name: str}]. CSV, JSON, and JSONL files

Input/output types:

input	output
any	any

Examples

Takes a dictionary and creates a dataframe

```
> [[a b];[1 2] [3 4]] | dfr into-df
```

Takes a list of tables and creates a dataframe

```
> [[1 2 a] [3 4 b] [5 6 c]] | dfr into-df
```

Takes a list and creates a dataframe

```
> [a b c] | dfr into-df
```

Takes a list of booleans and creates a dataframe

```
> [true true false] | dfr into-df
```

Convert to a dataframe and provide a schema

```
> {a: 1, b: {a: [1 2 3]}, c: [a b c]}|
 dfr into-df -s {a: u8, b: {a: list<u64>}, c:
list<str>}
```

dfr into-lazy

version: 0.90.2

usage:

Converts a dataframe into a lazy dataframe.

Signature

```
> dfr into-lazy --schema
```

Parameters

- `--schema {record}`: Polars Schema in format [{name: str}]. CSV, JSON, and JSONL files

Input/output types:

input	output
any	any

Examples

Takes a dictionary and creates a lazy dataframe

```
> [[a b]; [1 2] [3 4]] | dfr into-lazy
```

dfr into-nu

version: 0.90.2

usage:

Converts a dataframe or an expression into into nushell value for access and exploration.

Signature

```
> dfr into-nu --rows --tail
```

Parameters

- `--rows {number}`: number of rows to be shown
- `--tail`: shows tail rows

Input/output types:

input	output
any	any

Examples

Shows head rows from dataframe

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr into-nu
```

Shows tail rows from dataframe

```
> [[a b]; [1 2] [5 6] [3 4]] | dfr into-
df | dfr into-nu --tail --rows 1
```

Convert a col expression into a nushell value

```
> dfr col a | dfr into-nu
```

dfr is-duplicated

version: 0.90.2

usage:

Creates mask indicating duplicated values.

Signature

```
> dfr is-duplicated
```

Input/output types:

input	output
any	any

Examples

Create mask indicating duplicated values

```
> [5 6 6 6 8 8 8] | dfr into-df | dfr is-duplicated
```

Create mask indicating duplicated rows in a dataframe

```
> [[a, b]; [1 2] [1 2] [3 3] [3 3] [1 1]] | dfr into-df | dfr is-duplicated
```

dfr is-in

version: 0.90.2

usage:

Creates an is-in expression.

Signature

```
> dfr is-in (list)
```

Parameters

- `list`: List to check if values are in

Input/output types:

input	output
any	any

Examples

Creates a is-in expression

```
> let df = ( [ [ a b ] ; [ one 1 ] [ two 2 ] [ three 3
] ] | dfr into-df ) ; $ df | dfr with-column (
dfr col a | dfr is-in [ one two ] | dfr as a_in )
```

dfr is-not-null

version: 0.90.2

usage:

Creates mask where value is not null.

Signature

`> dfr is-not-null`

Input/output types:

input	output
any	any

Examples

Create mask where values are not null

```
> let s = ( [ 5 6 0 8 ] | dfr into-df ) ; let
res = ( $ s / $ s ) ; $ res | dfr is-not-null
```

Creates a is not null expression from a column

```
> dfr col a | dfr is-not-null
```

dfr is-null

version: 0.90.2

usage:

Creates mask where value is null.

Signature

```
> dfr is-null
```

Input/output types:

input	output
any	any

Examples

Create mask where values are null

```
> let s = ([5 6 0 8] | dfr into-df); let
res = ($s / $s); $res | dfr is-null
```

Creates a is null expression from a column

```
> dfr col a | dfr is-null
```

dfr is-unique

version: 0.90.2

usage:

Creates mask indicating unique values.

Signature

```
> dfr is-unique
```

Input/output types:

input	output
any	any

Examples

Create mask indicating unique values

```
> [5 6 6 6 8 8 8] | dfr into-df | dfr is-unique
```

Create mask indicating duplicated rows in a dataframe

```
> [[a, b]; [1 2] [1 2] [3 3] [3 3] [1 1
]] | dfr into-df | dfr is-unique
```

dfr join

version: 0.90.2

usage:

Joins a lazy frame with other lazy frame.

Signature

```
> dfr join (other) (left_on) (right_on) --inner --left --outer
--cross --suffix
```

Parameters

- `other`: LazyFrame to join with
- `left_on`: Left column(s) to join on
- `right_on`: Right column(s) to join on
- `--inner`: inner joing between lazyframes (default)
- `--left`: left join between lazyframes
- `--outer`: outer join between lazyframes
- `--cross`: cross join between lazyframes
- `--suffix {string}`: Suffix to use on columns with same name

Input/output types:

input	output
any	any

Examples

Join two lazy dataframes

```
> let df_a = ( [ [ a b c ] ; [ 1  " a "  0 ]  [ 2  " b "  1 ]
  [ 1  " c "  2 ]  [ 1  " c "  3 ] ]  |  dfr  into-lazy ) ;
    let  df_b  =   ( [[ " foo "  " bar "  " ham " ] ; [ 1  "
a "  " let " ] [ 2  " c "  " var " ] [ 3  " c "  " const "
]]  |  dfr into - lazy ) ;     $ df_a  |  dfr  join  $ df_
b  a foo   |   dfr  collect
```

Join one eager dataframe with a lazy dataframe

```
> let df_a = ( [ [ a b c ] ; [ 1  " a "  0 ]  [ 2  " b "  1 ]
  [ 1  " c "  2 ]  [ 1  " c "  3 ] ]  |  dfr  into-df ) ;
   let  df_b  =   ( [[ " foo "  " bar "  " ham " ] ; [ 1  "
a "  " let " ] [ 2  " c "  " var " ] [ 3  " c "  " const "
]]  |  dfr into - lazy ) ;     $ df_a  |  dfr  join  $ df_
b  a foo
```

dfr last

version: 0.90.2

usage:

Creates new dataframe with tail rows or creates a last expression.

Signature

```
> dfr last (rows)
```

Parameters

- `rows`: Number of rows for tail

Input/output types:

input	output
any	any

Examples

Create new dataframe with last rows

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr last 1
```

Creates a last expression from a column

```
> dfr col a | dfr last
```

dfr lit

version: 0.90.2

usage:

Creates a literal expression.

Signature

`> dfr lit (literal)`

Parameters

- `literal`: literal to construct the expression

Input/output types:

input	output
any	any

Examples

Created a literal expression and converts it to a nu object

```
> dfr lit 2 | dfr into-nu
```

dfr lowercase

version: 0.90.2

usage:

Lowercase the strings in the column.

Signature

```
> dfr lowercase
```

Input/output types:

input	output
any	any

Examples

Modifies strings to lowercase

```
> [Abc aBc abC] | dfr into-df | dfr lowercase
```

dfr ls

version: 0.90.2

usage:

Lists stored dataframes.

Signature

```
> dfr ls
```

Input/output types:

input	output
any	any

Examples

Creates a new dataframe and shows it in the dataframe list

```
> let test = ( [ [ a b ] ; [ 1 2 ]  [ 3 4 ] ]  |  dfr
into-df ) ;     ls
```

dfr max

version: 0.90.2

usage:

Creates a max expression or aggregates columns to their max value.

Signature

```
> dfr max
```

Input/output types:

input	output
any	any

Examples

Max value from columns in a dataframe

```
>  [ [ a b ] ;  [ 6 2 ]  [ 1 4 ]  [ 4 1 ] ]  |  dfr  into-
df  |  dfr  max
```

Max aggregation for a group-by

```
>  [ [ a b ] ;  [ one 2 ]  [ one 4 ]  [ two 1 ] ]     |
 dfr  into-df     |  dfr  group-by a     |  dfr  agg (
dfr col b  |  dfr  max )
```

dfr mean

version: 0.90.2

usage:

Creates a mean expression for an aggregation or aggregates columns to their mean value.

Signature

```
> dfr mean
```

Input/output types:

input	output
any	any

Examples

Mean value from columns in a dataframe

```
> [[a b]; [6 2] [4 2] [2 2]] | dfr into-df | dfr mean
```

Mean aggregation for a group-by

```
> [[a b]; [one 2] [one 4] [two 1]] | dfr into-df | dfr group-by a | dfr agg (dfr col b | dfr mean)
```

dfr median

version: 0.90.2

usage:

Aggregates columns to their median value

Signature

```
> dfr median
```

Input/output types:

input	output
any	any

Examples

Median value from columns in a dataframe

```
> [[a b]; [6 2] [4 2] [2 2]] | dfr into-df | dfr median
```

dfr melt

version: 0.90.2

usage:

Unpivot a DataFrame from wide to long format.

Signature

`> dfr melt --columns --values --variable-name --value-name`

Parameters

- `--columns {table}`: column names for melting
- `--values {table}`: column names used as value columns
- `--variable-name {string}`: optional name for variable column
- `--value-name {string}`: optional name for value column

Input/output types:

input	output
any	any

Examples

melt dataframe

```
> [[a b c d]; [x 1 4 a] [y 2 5 b] [z 3 6 c]] | dfr into-df | dfr melt -c [b c] -v [a d]
```

dfr min

version: 0.90.2

usage:

Creates a min expression or aggregates columns to their min value.

Signature

```
> dfr min
```

Input/output types:

input	output
any	any

Examples

Min value from columns in a dataframe

```
> [[a b]; [6 2] [1 4] [4 1]] | dfr into-
df | dfr min
```

Min aggregation for a group-by

```
> [[a b]; [one 2] [one 4] [two 1]]    |
 dfr into-df    | dfr group-by a    | dfr agg (
dfr col b | dfr min)
```

dfr n-unique

version: 0.90.2

usage:

Counts unique values.

Signature

```
> dfr n-unique
```

Input/output types:

input	output
any	any

Examples

Counts unique values

```
> [1 1 2 2 3 3 4] | dfr into-df | dfr n-unique
```

Creates a is n-unique expression from a column

```
> dfr col a | dfr n-unique
```

dfr not

version: 0.90.2

usage:

Inverts boolean mask.

Signature

```
> dfr not
```

Input/output types:

input	output
any	any

Examples

Inverts boolean mask

```
> [true false true] | dfr into-df | dfr not
```

dfr open

version: 0.90.2

usage:

Opens CSV, JSON, JSON lines, arrow, avro, or parquet file to create dataframe.

Signature

`> dfr open (file) --lazy --type --delimiter --no-header --infer-schema --skip-rows --columns --schema`

Parameters

- `file`: file path to load values from
- `--lazy`: creates a lazy dataframe
- `--type {string}`: File type: csv, tsv, json, parquet, arrow, avro. If omitted, derive from file extension
- `--delimiter {string}`: file delimiter character. CSV file
- `--no-header`: Indicates if file doesn't have header. CSV file
- `--infer-schema {number}`: Number of rows to infer the schema of the file. CSV file
- `--skip-rows {number}`: Number of rows to skip from file. CSV file
- `--columns {list<string>}`: Columns to be selected from csv file. CSV and Parquet file
- `--schema {record}`: Polars Schema in format [{name: str}]. CSV, JSON, and JSONL files

Input/output types:

input	output
any	any

Examples

Takes a file name and creates a dataframe

```
> dfr open test.csv
```

dfr otherwise

version: 0.90.2

usage:

Completes a when expression.

Signature

`> dfr otherwise (otherwise expression)`

Parameters

- `otherwise expression`: expression to apply when no when predicate matches

Input/output types:

input	output
any	any

Examples

Create a when conditions

```
> dfr when ((dfr col a ) > 2 ) 4 | dfr otherwise
5
```

Create a when conditions

```
> dfr when ((dfr col a ) > 2 ) 4 | dfr when ((
dfr col a ) < 0 ) 6 | dfr otherwise 0
```

Create a new column for the dataframe

```
> [[a b]; [6 2] [1 4] [4 1]] | dfr into-
lazy | dfr with-column ( dfr when ((dfr col a
) > 2 ) 4 | dfr otherwise 5 | dfr as c )
  | dfr with-column ( dfr when ((dfr col a ) >
 5 ) 10 | dfr when ((dfr col a ) < 2 ) 6 | dfr
otherwise 0 | dfr as d ) | dfr collect
```

dfr quantile

version: 0.90.2

usage:

Aggregates the columns to the selected quantile.

Signature

`> dfr quantile (quantile)`

Parameters

- `quantile`: quantile value for quantile operation

Input/output types:

input	output
any	any

Examples

quantile value from columns in a dataframe

```
> [[a b]; [6 2] [1 4] [4 1]] | dfr into-
df | dfr quantile 0.5
```

dfr query

version: 0.90.2

usage:

Query dataframe using SQL. Note: The dataframe is always named 'df' in your query's from clause.

Signature

`> dfr query (sql)`

Parameters

- `sql`: sql query

Input/output types:

input	output
any	any

Examples

Query dataframe using SQL

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr query 'select a from df'
```

dfr rename

version: 0.90.2

usage:

Rename a dataframe column.

Signature

```
> dfr rename (columns) (new names)
```

Parameters

- `columns`: Column(s) to be renamed. A string or list of strings
- `new names`: New names for the selected column(s). A string or list of strings

Input/output types:

input	output
any	any

Examples

Renames a series

```
> [5 6 7 8] | dfr into-df | dfr rename '0'
new_name
```

Renames a dataframe column

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr rename a a_new
```

Renames two dataframe columns

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr rename [a b] [a_new b_new]
```

dfr replace-all

version: 0.90.2

usage:

Replace all (sub)strings by a regex pattern.

Signature

`> dfr replace-all --pattern --replace`

Parameters

- `--pattern {string}`: Regex pattern to be matched
- `--replace {string}`: replacing string

Input/output types:

input	output
any	any

Examples

Replaces string

```
> [abac abac abac] | dfr into-df | dfr replace-
all --pattern a --replace A
```

dfr replace

version: 0.90.2

usage:

Replace the leftmost (sub)string by a regex pattern.

Signature

```
> dfr replace --pattern --replace
```

Parameters

- `--pattern {string}`: Regex pattern to be matched
- `--replace {string}`: replacing string

Input/output types:

input	output
any	any

Examples

Replaces string

```
> [abc abc abc] | dfr into-df | dfr replace -
-pattern ab --replace AB
```

dfr reverse

version: 0.90.2

usage:

Reverses the LazyFrame

Signature

```
> dfr reverse
```

Input/output types:

input	output
any	any

Examples

Reverses the dataframe.

```
> [[a b]; [6 2] [4 2] [2 2]] | dfr into-df | dfr reverse
```

dfr rolling

version: 0.90.2

usage:

Rolling calculation for a series.

Signature

```
> dfr rolling (type) (window)
```

Parameters

- `type`: rolling operation
- `window`: Window size for rolling

Input/output types:

input	output
any	any

Examples

Rolling sum for a series

```
> [1 2 3 4 5] | dfr into-df | dfr rolling sum
2 | dfr drop-nulls
```

Rolling max for a series

```
> [1 2 3 4 5] | dfr into-df | dfr rolling max
2 | dfr drop-nulls
```

dfr sample

version: 0.90.2

usage:

Create sample dataframe.

Signature

`> dfr sample --n-rows --fraction --seed --replace --shuffle`

Parameters

- `--n-rows {int}`: number of rows to be taken from dataframe
- `--fraction {number}`: fraction of dataframe to be taken
- `--seed {number}`: seed for the selection
- `--replace`: sample with replace
- `--shuffle`: shuffle sample

Input/output types:

input	output
any	any

Examples

Sample rows from dataframe

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr sample --n-rows 1
```

Shows sample row using fraction and replace

```
> [[a b]; [1 2] [3 4] [5 6]] | dfr into-
df | dfr sample --fraction 0.5 --replace
```

dfr schema

version: 0.90.2

usage:

Show schema for a dataframe.

Signature

`> dfr schema --datatype-list`

Parameters

- `--datatype-list`: creates a lazy dataframe

Input/output types:

input	output
any	any

Examples

Dataframe schema

```
> [[a b]; [1 "foo"] [3 "bar"]] | dfr
into-df | dfr schema
```

dfr select

version: 0.90.2

usage:

Selects columns from lazyframe.

Signature

`> dfr select ...rest`

Parameters

- `...rest`: Expression(s) that define the column selection

Input/output types:

input	output
any	any

Examples

Select a column from the dataframe

```
> [[a b]; [6 2] [4 2] [2 2]] | dfr into-
df | dfr select a
```

dfr set-with-idx

version: 0.90.2

usage:

Sets value in the given index.

Signature

`> dfr set-with-idx (value) --indices`

Parameters

- `value`: value to be inserted in series
- `--indices {any}`: list of indices indicating where to set the value

Input/output types:

input	output
any	any

Examples

Set value in selected rows from series

```
> let series = ( [ 4 1 5 2 4 3 ]  |  dfr  into-df ) ;
  let  indices  =   ( [ 0   2 ]  |  dfr into - df ) ;     $
series   |   dfr  set-with-idx 6  -- indices   $ indices
```

dfr set

version: 0.90.2

usage:

Sets value where given mask is true.

Signature

`> dfr set (value) --mask`

Parameters

- `value`: value to be inserted in series
- `--mask {any}`: mask indicating insertions

Input/output types:

input	output
any	any

Examples

Shifts the values by a given period

```
> let s = ( [ 1 2 2 3 3 ] | dfr into-df | dfr
shift 2 ) ;    let mask = ( $ s | dfr is - null ) ;
    $ s | dfr set 0 -- mask $ mask
```

dfr shape

version: 0.90.2

usage:

Shows column and row size for a dataframe.

Signature

```
> dfr shape
```

Input/output types:

input	output
any	any

Examples

Shows row and column shape

```
> [ [ a b ] ; [ 1 2 ] [ 3 4 ] ] | dfr into-df |
dfr shape
```

dfr shift

version: 0.90.2

usage:

Shifts the values by a given period.

Signature

```
> dfr shift (period) --fill
```

Parameters

- `period`: shift period
- `--fill {any}`: Expression used to fill the null values (lazy df)

Input/output types:

input	output
any	any

Examples

Shifts the values by a given period

```
> [1 2 2 3 3] | dfr into-df | dfr shift 2 | dfr drop-nulls
```

dfr slice

version: 0.90.2

usage:

Creates new dataframe from a slice of rows.

Signature

`> dfr slice (offset) (size)`

Parameters

- `offset`: start of slice
- `size`: size of slice

Input/output types:

input	output
any	any

Examples

Create new dataframe from a slice of the rows

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr slice 0 1
```

dfr sort-by

version: 0.90.2

usage:

Sorts a lazy dataframe based on expression(s).

Signature

`> dfr sort-by ...rest --reverse --nulls-last --maintain-order`

Parameters

- `...rest`: sort expression for the dataframe
- `--reverse {list<bool>}`: Reverse sorting. Default is false
- `--nulls-last`: nulls are shown last in the dataframe
- `--maintain-order`: Maintains order during sort

Input/output types:

input	output
any	any

Examples

Sort dataframe by one column

```
> [[a b]; [6 2] [1 4] [4 1]] | dfr into-
df | dfr sort-by a
```

Sort column using two columns

```
> [[a b]; [6 2] [1 1] [1 4] [2 4]] | dfr
  into-df | dfr sort-by [a b] -r [false true]
```

dfr std

version: 0.90.2

usage:

Creates a std expression for an aggregation of std value from columns in a dataframe.

Signature

```
> dfr std
```

Input/output types:

input	output
any	any

Examples

Std value from columns in a dataframe

```
> [[a b]; [6 2] [4 2] [2 2]] | dfr into-
df | dfr std
```

Std aggregation for a group-by

```
> [[a b]; [one 2] [one 2] [two 1] [two 1]]
      | dfr into-df | dfr group-by a | dfr
  agg (dfr col b | dfr std)
```

dfr str-lengths

version: 0.90.2

usage:

Get lengths of all strings.

Signature

```
> dfr str-lengths
```

Input/output types:

input	output
any	any

Examples

Returns string lengths

```
> [ a ab abc ] | dfr into-df | dfr str-lengths
```

dfr str-slice

version: 0.90.2

usage:

Slices the string from the start position until the selected length.

Signature

```
> dfr str-slice (start) --length
```

Parameters

- `start`: start of slice
- `--length {int}`: optional length

Input/output types:

input	output
any	any

Examples

Creates slices from the strings

```
> [ abcded abc321 abc123 ] | dfr into-df | dfr
str-slice 1 --length 2
```

dfr strftime

version: 0.90.2

usage:

Formats date based on string rule.

Signature

```
> dfr strftime (fmt)
```

Parameters

- `fmt`: Format rule

Input/output types:

input	output
any	any

Examples

Formats date

```
> let dt = ('2020-08-04T16:39:18+00:00' | into
datetime --timezone 'UTC'); let df = ([$dt
 $dt] | dfr into-df); $df | dfr strftime
"%Y/%m/%d"
```

dfr sum

version: 0.90.2

usage:

Creates a sum expression for an aggregation or aggregates columns to their sum value.

Signature

```
> dfr sum
```

Input/output types:

input	output
any	any

Examples

Sums all columns in a dataframe

```
> [[a b]; [6 2] [1 4] [4 1]] | dfr into-df | dfr sum
```

Sum aggregation for a group-by

```
> [[a b]; [one 2] [one 4] [two 1]] | dfr into-df | dfr group-by a | dfr agg (dfr col b | dfr sum)
```

dfr summary

version: 0.90.2

usage:

For a dataframe, produces descriptive statistics (summary statistics) for its numeric columns.

Signature

```
> dfr summary --quantiles
```

Parameters

- `--quantiles {table}`: provide optional quantiles

Input/output types:

input	output
any	any

Examples

list dataframe descriptives

```
> [[a b]; [1 1] [1 1]] | dfr into-df |
dfr summary
```

dfr take

version: 0.90.2

usage:

Creates new dataframe using the given indices.

Signature

`> dfr take (indices)`

Parameters

- `indices`: list of indices used to take data

Input/output types:

input	output
any	any

Examples

Takes selected rows from dataframe

```
> let df = ([[a b]; [4 1] [5 2] [4 3]] |
dfr into-df); let indices = ([0 2] | dfr
into-df); $df | dfr take $indices
```

Takes selected rows from series

```
> let series = ([4 1 5 2 4 3] | dfr into-df);
  let indices = ([0 2] | dfr into-df); $
series | dfr take $indices
```

dfr to-arrow

version: 0.90.2

usage:

Saves dataframe to arrow file.

Signature

```
> dfr to-arrow (file)
```

Parameters

- `file`: file path to save dataframe

Input/output types:

input	output
any	any

Examples

Saves dataframe to arrow file

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr to-arrow test.arrow
```

dfr to-avro

version: 0.90.2

usage:

Saves dataframe to avro file.

Signature

```
> dfr to-avro (file) --compression
```

Parameters

- `file`: file path to save dataframe
- `--compression {string}`: use compression, supports deflate or snappy

Input/output types:

input	output
any	any

Examples

Saves dataframe to avro file

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr to-avro test.avro
```

dfr to-csv

version: 0.90.2

usage:

Saves dataframe to CSV file.

Signature

```
> dfr to-csv (file) --delimiter --no-header
```

Parameters

- `file`: file path to save dataframe
- `--delimiter {string}`: file delimiter character
- `--no-header`: Indicates if file doesn't have header

Input/output types:

input	output
any	any

Examples

Saves dataframe to CSV file

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr to-csv test.csv
```

Saves dataframe to CSV file using other delimiter

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr to-csv test.csv --delimiter '|'
```

dfr to-jsonl

version: 0.90.2

usage:

Saves dataframe to a JSON lines file.

Signature

`> dfr to-jsonl (file)`

Parameters

- `file`: file path to save dataframe

Input/output types:

input	output
any	any

Examples

Saves dataframe to JSON lines file

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr to-jsonl test.jsonl
```

dfr to-parquet

version: 0.90.2

usage:

Saves dataframe to parquet file.

Signature

```
> dfr to-parquet (file)
```

Parameters

- `file`: file path to save dataframe

Input/output types:

input	output
any	any

Examples

Saves dataframe to parquet file

```
> [[a b]; [1 2] [3 4]] | dfr into-df |
dfr to-parquet test.parquet
```

dfr unique

version: 0.90.2

usage:

Returns unique values from a dataframe.

Signature

```
> dfr unique --subset --last --maintain-order
```

Parameters

- `--subset {any}`: Subset of column(s) to use to maintain rows (lazy df)
- `--last`: Keeps last unique value. Default keeps first value (lazy df)

- `--maintain-order`: Keep the same order as the original DataFrame (lazy df)

Input/output types:

input	output
any	any

Examples

Returns unique values from a series

```
> [2 2 2 2 2] | dfr into-df | dfr unique
```

Creates a is unique expression from a column

```
> col a | unique
```

dfr uppercase

version: 0.90.2

usage:

Uppercase the strings in the column.

Signature

```
> dfr uppercase
```

Input/output types:

input	output
any	any

Examples

Modifies strings to uppercase

```
> [Abc aBc abC] | dfr into-df | dfr uppercase
```

dfr value-counts

version: 0.90.2

usage:

Returns a dataframe with the counts for unique values in series.

Signature

```
> dfr value-counts
```

Input/output types:

input	output
any	any

Examples

Calculates value counts

```
> [ 5 5 5 5 6 6 ] | dfr into-df | dfr value-
counts
```

dfr var

version: 0.90.2

usage:

Create a var expression for an aggregation.

Signature

```
> dfr var
```

Input/output types:

input	output
any	any

Examples

Var value from columns in a dataframe or aggregates columns to their var value

```
> [[a b]; [6 2] [4 2] [2 2]] | dfr into-
df | dfr var
```

Var aggregation for a group-by

```
> [[a b]; [one 2] [one 2] [two 1] [two 1]]
    | dfr into-df    | dfr group-by a    | dfr
 agg (dfr col b | dfr var)
```

dfr when

version: 0.90.2

usage:

Creates and modifies a when expression.

Signature

`> dfr when (when expression) (then expression)`

Parameters

- `when expression`: when expression used for matching
- `then expression`: expression that will be applied when predicate is true

Input/output types:

input	output
any	any

Examples

Create a when conditions

```
> dfr when ((dfr col a ) > 2 ) 4
```

Create a when conditions

```
> dfr when ((dfr col a ) > 2 ) 4 | dfr when ((
dfr col a ) < 0 ) 6
```

Create a new column for the dataframe

```
> [ [ a b ] ; [ 6 2 ] [ 1 4 ] [ 4 1 ] ] | dfr into-
lazy | dfr with-column ( dfr when ((dfr col a
) > 2 ) 4 | dfr otherwise 5 | dfr as c )
 | dfr with-column ( dfr when ((dfr col a ) >
 5 ) 10 | dfr when ((dfr col a ) < 2 ) 6 | dfr
 otherwise 0 | dfr as d ) | dfr collect
```

dfr with-column

version: 0.90.2

usage:

Adds a series to the dataframe.

Signature

```
> dfr with-column ...rest --name
```

Parameters

- `...rest`: series to be added or expressions used to define the new columns
- `--name {string}`: new column name

Input/output types:

input	output
any	any

Examples

Adds a series to the dataframe

```
> [[a b]; [1 2] [3 4]] | dfr into-df
| dfr with-column ([5 6] | dfr into-df) --name
c
```

Adds a series to the dataframe

```
> [[a b]; [1 2] [3 4]] | dfr into-lazy
    | dfr with-column [ ((dfr col a) * 2 |
dfr as "c") ((dfr col a) * 3 | dfr as "d"
) ] | dfr collect
```

do

version: 0.90.2

usage:

Run a closure, providing it with the pipeline input.

Signature

```
> do (closure) ...rest --ignore-errors --ignore-shell-errors
--ignore-program-errors --capture-errors --env
```

Parameters

- `closure`: The closure to run.
- `...rest`: The parameter(s) for the closure.
- `--ignore-errors`: ignore errors as the closure runs
- `--ignore-shell-errors`: ignore shell errors as the closure runs
- `--ignore-program-errors`: ignore external program errors as the closure runs
- `--capture-errors`: catch errors as the closure runs, and return them
- `--env`: keep the environment defined inside the command

Input/output types:

input	output
any	any

Examples

Run the closure

```
> do { echo hello }
```

Run a stored first-class closure

```
> let text = "I am enclosed"; let hello = {||
echo $text}; do $hello
```

Run the closure and ignore both shell and external program errors

```
> do --ignore-errors { thisisnotarealcommand }
```

Run the closure and ignore shell errors

```
> do --ignore-shell-errors { thisisnotarealcommand
}
```

Run the closure and ignore external program errors

```
> do --ignore-program-errors { nu --commands 'exit
1' }; echo "I'll still run"
```

Abort the pipeline if a program returns a non-zero exit code

```
> do --capture-errors { nu --commands 'exit 1' }
  | myscarycommand
```

Run the closure, with a positional parameter

```
> do {|x| 100 + $x } 77
```

Run the closure, with input

```
> 77 | do {|x| 100 + $in }
```

Run the closure and keep changes to the environment

```
> do --env { $env.foo = 'bar' }; $env.foo
```

drop

version: 0.90.2

usage:

Remove items/rows from the end of the input list/table. Counterpart of `skip`. Opposite of `last`.

Signature

`> drop (rows)`

Parameters

- `rows`: The number of items to remove.

Input/output types:

input	output
list<any>	list<any>
table	table

Examples

Remove the last item of a list

```
> [0,1,2,3] | drop
```

Remove zero item of a list

```
> [0,1,2,3] | drop 0
```

Remove the last two items of a list

```
> [0,1,2,3] | drop 2
```

Remove the last row in a table

```
> [[a, b]; [1, 2] [3, 4]] | drop 1
```

Subcommands:

name	type	usage
`drop column`[144]	Builtin	Remove N columns at the right-hand end of the input table. To remove columns by name, use `reject`.
`drop nth`[145]	Builtin	Drop the selected rows.

drop column

version: 0.90.2

usage:

Remove N columns at the right-hand end of the input table. To remove columns by name, use `reject`.

Signature

```
> drop column (columns)
```

Parameters

- `columns`: Starting from the end, the number of columns to remove.

Input/output types:

input	output
record	record
table	table

Examples

Remove the last column of a table

[144] /commands/docs/drop_column.md
[145] /commands/docs/drop_nth.md

```
> [[lib, extension]; [nu-lib, rs] [nu-core,
rb]] | drop column
```

Remove the last column of a record

```
> {lib: nu-lib , extension: rs} | drop column
```

drop nth

version: 0.90.2

usage:

Drop the selected rows.

Signature

`> drop nth (row number or row range) ...rest`

Parameters

- `row number or row range`: The number of the row to drop or a range to drop consecutive rows.
- `...rest`: The number of the row to drop.

Input/output types:

input	output
list<any>	list<any>

Examples

Drop the first, second, and third row

```
> [sam,sarah,2,3,4,5] | drop nth 0 1 2
```

Drop the first, second, and third row

```
> [0,1,2,3,4,5] | drop nth 0 1 2
```

Drop rows 0 2 4

```
> [0,1,2,3,4,5] | drop nth 0 2 4
```

Drop rows 2 0 4

```
> [0,1,2,3,4,5] | drop nth 2 0 4
```

Drop range rows from second to fourth

```
> [first second third fourth fifth] | drop nth (
1..3)
```

Drop all rows except first row

```
> [0,1,2,3,4,5] | drop nth 1..
```

Drop rows 3,4,5

```
> [0,1,2,3,4,5] | drop nth 3..
```

du

version: 0.90.2

usage:

Find disk usage sizes of specified items.

Signature

`> du (path) --all --deref --exclude --max-depth --min-size`

Parameters

- `path`: Starting directory.
- `--all`: Output file sizes as well as directory sizes
- `--deref`: Dereference symlinks to their targets for size
- `--exclude {glob}`: Exclude these file names
- `--max-depth {int}`: Directory recursion limit
- `--min-size {int}`: Exclude files below this size

Input/output types:

input	output
nothing	table

Examples

Disk usage of the current directory

```
> du
```

each

version: 0.90.2

usage:

Run a closure on each row of the input list, creating a new list with the results.

Signature

```
> each (closure) --keep-empty
```

Parameters

- `closure`: The closure to run.
- `--keep-empty`: keep empty result cells

Input/output types:

input	output
any	any
list<any>	list<any>
table	list<any>

Examples

Multiplies elements in the list

```
> [1 2 3] | each { |e| 2 * $e }
```

Produce a list of values in the record, converted to string

```
> {major:2, minor:1, patch:4} | values | each
  { || into string }
```

Produce a list that has “two” for each 2 in the input

```
> [1 2 3 2] | each { |e| if $e == 2 { "two"
  } }
```

Iterate over each element, producing a list showing indexes of any 2s

```
> [1 2 3] | enumerate | each { |e| if $e.item
== 2 { $"found 2 at ($e.index)!"} }
```

Iterate over each element, keeping null results

```
> [1 2 3] | each --keep-empty { |e| if $e ==
2 { "found 2!"} }
```

Notes

```
Since tables are lists of records, passing a table into
'each' will iterate over each record, not necessarily each
cell within it. Avoid passing single records to this command.
 Since a record is a one-row structure, 'each' will only
run once, behaving similar to 'do'.  To iterate over a
record's values, try converting it to a table with 'transpose'
first.
```

Subcommands:

name	type	usage
each while[146]	Builtin	Run a block on each row of the input list until a null is found, then create a new list with the results.

[146] /commands/docs/each_while.md

each while

version: 0.90.2

usage:

Run a block on each row of the input list until a null is found, then create a new list with the results.

Signature

```
> each while (closure)
```

Parameters

- `closure`: the closure to run

Input/output types:

input	output
list<any>	list<any>

Examples

Produces a list of each element before the 3, doubled

```
> [1 2 3 2 1] | each while {|e| if $e < 3 { $e * 2 } }
```

Output elements until reaching 'stop'

```
> [1 2 stop 3 4] | each while {|e| if $e != 'stop' { $"Output: ($e)" } }
```

Iterate over each element, printing the matching value and its index

```
> [1 2 3] | enumerate | each while {|e| if $e.item < 2 { $"value ($e.item) at ($e.index)!" } }
```

echo

version: 0.90.2

usage:

Returns its arguments, ignoring the piped-in value.

Signature

```
> echo ...rest
```

Parameters

- `...rest`: The values to echo.

Input/output types:

input	output
nothing	any

Examples

Put a list of numbers in the pipeline. This is the same as [1 2 3].

```
> echo 1 2 3
```

Returns the piped-in value, by using the special $in variable to obtain it.

```
> echo $ in
```

Notes

```
When given no arguments, it returns an empty string. When
given one argument, it returns it. Otherwise, it returns
a list of the arguments. There is usually little reason
to use this over just writing the values as-is.
```

encode

version: 0.90.2

usage:

Encode a string into bytes.

Signature

`> encode (encoding) --ignore-errors`

Parameters

- `encoding`: The text encoding to use.
- `--ignore-errors`: when a character isn't in the given encoding, replace with a HTML entity (like `&#127880;`)

Input/output types:

input	output
string	binary

Examples

Encode an UTF-8 string into Shift-JIS

```
> "            " | encode shift-jis
```

Replace characters with HTML entities if they can't be encoded

```
> " " | encode -- ignore-errors shift-jis
```

Notes

> Multiple encodings are supported; here are a few: big5, euc-jp, euc-kr, gbk, iso-8859-1, cp1252, latin5 Note that since the Encoding Standard doesn't specify encoders for utf-16le and utf-16be, these are not yet supported. For a more complete list of encodings, please refer to the encoding_rs documentation link at https://docs.rs/encoding_rs/latest/encoding_rs/#statics

Subcommands:

name	type	usage
`encode base64`[147]	Builtin	Encode a string or binary value using Base64.
`encode hex`[148]	Builtin	Encode a binary value using hex.

encode base64

version: 0.90.2

usage:

Encode a string or binary value using Base64.

Signature

```
> encode base64 ...rest --character-set
```

Parameters

- `...rest`: For a data structure input, encode data at the given cell paths.
- `--character-set {string}`: specify the character rules for encoding the input. Valid values are 'standard', 'standard-no-padding', 'url-safe', 'url-safe-no-padding','binhex', 'bcrypt', 'crypt', 'mutf7'

Input/output types:

input	output
binary	string
list<any>	list<string>
list<binary>	list<string>
list<string>	list<string>
record	record
string	string
table	table

[147] `/commands/docs/encode_base64.md`
[148] `/commands/docs/encode_hex.md`

Examples

Encode binary data

```
> 0 x [ 09 F9 11 02 9D 74 E3 5B D8 41 56 C5 63 56 88 C0
] | encode base64
```

Encode a string with default settings

```
> ' Some Data ' | encode base64
```

Encode a string with the binhex character set

```
> ' Some Data ' | encode base64 -- character-set
binhex
```

encode hex

version: 0.90.2

usage:

Encode a binary value using hex.

Signature

`> encode hex ...rest`

Parameters

- `...rest`: For a data structure input, encode data at the given cell paths

Input/output types:

input	output
binary	string
list<binary>	list<string>
record	record
table	table

Examples

Encode binary data

```
> 0x[09 F9 11 02 9D 74 E3 5B D8 41 56 C5 63 56 88 C0
] | encode hex
```

enumerate

version: 0.90.2

usage:

Enumerate the elements in a stream.

Signature

`> enumerate`

Input/output types:

input	output
any	table

Examples

Add an index to each element of a list

```
> [a, b, c] | enumerate
```

error make

version: 0.90.2

usage:

Create an error.

Signature

`> error make (error_struct) --unspanned`

Parameters

- `error_struct`: The error to create.
- `--unspanned`: remove the origin label from the error

Input/output types:

input	output
nothing	any

Examples

Create a simple custom error

```
> error make { msg: " my custom error message " }
```

Create a more complex custom error

```
> error make {         msg: " my custom error message
"         label: {             text: " my custom label
text "     #  not mandatory unless $.label exists
       #  optional                  span:  {
 #  if $.label.span exists, both start and end must be
present                     start: 123                end:
 456              }          }          help:  " A help
string, suggesting a fix to the user "     #  optional
   }
```

Create a custom error for a custom command that shows the span of the argument

```
> def foo  [ x ]   {          error make  {
msg:  " this is fishy "              label:  {
      text:  " fish right here "                span: (
metadata  $ x ).span              }         }     }
```

every

version: 0.90.2

usage:

Show (or skip) every n-th row, starting from the first one.

Signature

```
> every (stride) --skip
```

Parameters

- `stride`: How many rows to skip between (and including) each row returned.
- `--skip`: skip the rows that would be returned, instead of selecting them

Input/output types:

input	output
list<any>	list<any>

Examples

Get every second row

```
> [1 2 3 4 5] | every 2
```

Skip every second row

```
> [1 2 3 4 5] | every 2 --skip
```

exec

version: 0.90.2

usage:

Execute a command, replacing or exiting the current process, depending on platform.

Signature

```
> exec (command)
```

Parameters

- `command`: The command to execute.

Input/output types:

input	output
nothing	any

Examples

Execute external 'ps aux' tool

```
> exec ps aux
```

Execute 'nautilus'

```
> exec nautilus
```

Notes

```
On Unix-based systems, the current process is replaced
with the command. On Windows based systems, Nushell will
wait for the command to finish and then exit with the command's
exit code.
```

exit

version: 0.90.2

usage:

Exit Nu.

Signature

```
> exit (exit_code)
```

Parameters

- `exit_code`: Exit code to return immediately with.

Input/output types:

input	output
nothing	nothing

Examples

Exit the current shell

```
> exit
```

explain

version: 0.90.2

usage:

Explain closure contents.

Signature

`> explain (closure)`

Parameters

- `closure`: The closure to run.

Input/output types:

input	output
any	any
nothing	any

Examples

Explain a command within a closure

```
> explain { || ls | sort-by name type --ignore-case |
get name } | table -- expand
```

explore

version: 0.90.2

usage:

Explore acts as a table pager, just like `less` does for text.

Signature

```
> explore --head --index --reverse --peek
```

Parameters

- `--head {bool}`: Show or hide column headers (default true)
- `--index`: Show row indexes when viewing a list
- `--reverse`: Start with the viewport scrolled to the bottom
- `--peek`: When quitting, output the value of the cell the cursor was on

Input/output types:

input	output
any	any

Examples

Explore the system information record

```
> sys | explore
```

Explore the output of `ls` without column names

```
> ls | explore --head false
```

Explore a list of Markdown files' contents, with row indexes

```
> glob *.md | each {|| open } | explore --
index
```

Explore a JSON file, then save the last visited sub-structure to a file

```
> open file.json | explore --peek | to json |
  save part.json
```

Notes

```
Press `:` then `h` to get a help menu.
```

export-env

version: 0.90.2

usage:

Run a block and preserve its environment in a current scope.

Signature

```
> export-env (block)
```

Parameters

- `block`: The block to run to set the environment.

Input/output types:

input	output
nothing	nothing

Examples

Set an environment variable

```
> export-env { $env.SPAM = 'eggs' }
```

Set an environment variable and examine its value

```
> export-env { $env.SPAM = 'eggs' }; $env.
SPAM
```

export

version: 0.90.2

usage:

Export definitions or environment variables from a module.

Signature

```
> export
```

Input/output types:

input	output
nothing	nothing

Examples

Export a definition from a module

```
> module utils { export def my-command [] { "hello
" } }; use utils my - command; my - command
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

Subcommands:

name	type	usage
`export alias`[149]	Builtin	Alias a command (with optional flags) to a new name and export it from a module.
`export const`[150]	Builtin	Use parse-time constant from a module and export them from this module.
`export def`[151]	Builtin	Define a custom command and export it from a module.
`export extern`[152]	Builtin	Define an extern and export it from a module.
`export module`[153]	Builtin	Export a custom module from a module.
`export use`[154]	Builtin	Use definitions from a module and export them from this module.

export alias

version: 0.90.2

usage:

Alias a command (with optional flags) to a new name and export it from a module.

[149] /commands/docs/export_alias.md
[150] /commands/docs/export_const.md
[151] /commands/docs/export_def.md
[152] /commands/docs/export_extern.md
[153] /commands/docs/export_module.md
[154] /commands/docs/export_use.md

Signature

```
> export alias (name) (initial_value)
```

Parameters

- `name`: Name of the alias.
- `initial_value`: Equals sign followed by value.

Input/output types:

input	output
nothing	nothing

Examples

Alias ll to ls -l and export it from a module

```
> module spam { export alias ll = ls -l }
```

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

export const

version: 0.90.2

usage:

Use parse-time constant from a module and export them from this module.

Signature

```
> export const (const_name) (initial_value)
```

Parameters

- `const_name`: Constant name.
- `initial_value`: Equals sign followed by constant value.

Input/output types:

input	output
nothing	nothing

Examples

Re-export a command from another module

```
> module spam { export const foo = 3; }     module
 eggs { export use spam foo }     use eggs foo
foo
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

export def

version: 0.90.2

usage:

Define a custom command and export it from a module.

Signature

`> export def (def_name) (params) (block) --env --wrapped`

Parameters

- `def_name`: Command name.
- `params`: Parameters.
- `block`: Body of the definition.
- `--env`: keep the environment defined inside the command
- `--wrapped`: treat unknown flags and arguments as strings (requires ...rest-like parameter in signature)

Input/output types:

input	output
nothing	nothing

Examples

Define a custom command in a module and call it

```
> module spam { export def foo [] { "foo" } };
use spam foo; foo
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

export extern

version: 0.90.2

usage:

Define an extern and export it from a module.

Signature

```
> export extern (def_name) (params)
```

Parameters

- `def_name`: Definition name.
- `params`: Parameters.

Input/output types:

input	output
nothing	nothing

Examples

Export the signature for an external command

```
> export extern echo [text: string]
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

export module

version: 0.90.2

usage:

Export a custom module from a module.

Signature

```
> export module (module) (block)
```

Parameters

- `module`: Module name or module path.
- `block`: Body of the module if 'module' parameter is not a path.

Input/output types:

input	output
nothing	nothing

Examples

Define a custom command in a submodule of a module and call it

```
> module spam {         export module eggs {
    export def foo [] { "foo" }         }     }
  use spam eggs     eggs foo
```

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

export use

version: 0.90.2

usage:

Use definitions from a module and export them from this module.

Signature

```
> export use (module) (members)
```

Parameters

- `module`: Module or module file.
- `members`: Which members of the module to import.

Input/output types:

input	output
nothing	nothing

Examples

Re-export a command from another module

```
> module spam { export def foo [] { "foo" } }
    module eggs { export use spam foo }    use eggs
foo    foo
```

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

extern

version: 0.90.2

usage:

Define a signature for an external command.

Signature

```
> extern (def_name) (params)
```

Parameters

- `def_name`: Definition name.
- `params`: Parameters.

Input/output types:

input	output
nothing	nothing

Examples

Write a signature for an external command

```
> extern echo [ text: string ]
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

fill

version: 0.90.2

usage:

Fill and Align.

Signature

```
> fill --width --alignment --character
```

Parameters

- `--width {int}`: The width of the output. Defaults to 1
- `--alignment {string}`: The alignment of the output. Defaults to Left (Left(l), Right(r), Center(c/m), MiddleRight(cr/mr))
- `--character {string}`: The character to fill with. Defaults to ' ' (space)

Input/output types:

input	output
filesize	string
float	string
int	string
list<any>	list<string>
list<filesize>	list<string>
list<float>	list<string>
list<int>	list<string>
list<string>	list<string>
string	string

Examples

Fill a string on the left side to a width of 15 with the character ' '

```
> 'nushell' | fill --alignment l --character
' ' --width 15
```

Fill a string on the right side to a width of 15 with the character ' '

```
> 'nushell' | fill --alignment r --character
' ' --width 15
```

Fill a string on both sides to a width of 15 with the character ' '

```
> 'nushell' | fill --alignment m --character
' ' --width 15
```

Fill a number on the left side to a width of 5 with the character '0'

```
> 1 | fill --alignment right --character '0'
--width 5
```

Fill a number on both sides to a width of 5 with the character ‘0’

```
> 1.1 | fill --alignment center --character '
0' --width 5
```

Fill a filesize on the left side to a width of 5 with the character ‘0’

```
> 1kib | fill --alignment middle --character '
0' --width 10
```

filter

version: 0.90.2

usage:

Filter values based on a predicate closure.

Signature

`> filter (closure)`

Parameters

- `closure`: Predicate closure.

Input/output types:

input	output
list<any>	list<any>
range	list<any>

Examples

Filter items of a list according to a condition

```
> [1 2] | filter {|x| $x > 1}
```

Filter rows of a table according to a condition

```
> [{a: 1} {a: 2}] | filter {|x| $x.a > 1}
```

Filter rows of a table according to a stored condition

```
> let cond = {|x| $x.a > 1}; [{a: 1} {a: 2}]
  | filter $cond
```

Filter items of a range according to a condition

```
> 9..13 | filter {|el| $el mod 2 != 0}
```

List all numbers above 3, using an existing closure condition

```
> let a = {$in > 3}; [1, 2, 5, 6] | filter $
a
```

Notes

```
This command works similar to 'where' but allows reading
the predicate closure from a variable. On the other hand,
the "row condition" syntax is not supported.
```

find

version: 0.90.2

usage:

Searches terms in the input.

Signature

`> find ...rest --regex --ignore-case --multiline --dotall --columns --invert`

Parameters

- `...rest`: Terms to search.
- `--regex {string}`: regex to match with
- `--ignore-case`: case-insensitive regex mode; equivalent to (?i)

- `--multiline`: multi-line regex mode: ^ and $ match begin/end of line; equivalent to (?m)
- `--dotall`: dotall regex mode: allow a dot . to match newlines \n; equivalent to (?s)
- `--columns {list<string>}`: column names to be searched (with rest parameter, not regex yet)
- `--invert`: invert the match

Input/output types:

input	output
list<any>	list<any>
string	any

Examples

Search for multiple terms in a command output

```
> ls | find toml md sh
```

Search and highlight text for a term in a string

```
> 'Cargo.toml' | find toml
```

Search a number or a file size in a list of numbers

```
> [1 5 3kb 4 3Mb] | find 5 3kb
```

Search a char in a list of string

```
> [moe larry curly] | find l
```

Find using regex

```
> [abc bde arc abf] | find --regex "ab"
```

Find using regex case insensitive

```
> [aBc bde Arc abf] | find --regex "ab" -i
```

Find value in records using regex

```
> [[version name]; ['0.1.0' nushell] ['0.1.1
' fish] ['0.2.0' zsh]] | find --regex "nu"
```

Find inverted values in records using regex

```
> [[version name]; ['0.1.0' nushell] ['0.1.1
' fish] ['0.2.0' zsh]] | find --regex "nu"
 --invert
```

Find value in list using regex

```
> [["Larry", "Moe"], ["Victor", "Marina"]
] | find --regex "rr"
```

Find inverted values in records using regex

```
> [["Larry", "Moe"], ["Victor", "Marina"]
] | find --regex "rr" --invert
```

Remove ANSI sequences from result

```
> [[foo bar]; [abc 123] [def 456]] | find
123 | get bar | ansi strip
```

Find and highlight text in specific columns

```
> [[col1 col2 col3]; [moe larry curly] [larry
curly moe]] | find moe --columns [col1]
```

first

version: 0.90.2

usage:

Return only the first several rows of the input. Counterpart of `last`. Opposite of `skip`.

Signature

```
> first (rows)
```

Parameters

- `rows`: Starting from the front, the number of rows to return.

Input/output types:

input	output
binary	binary
list<any>	any
range	any

Examples

Return the first item of a list/table

```
> [1 2 3] | first
```

Return the first 2 items of a list/table

```
> [1 2 3] | first 2
```

Return the first 2 bytes of a binary value

```
> 0x[01 23 45] | first 2
```

flatten

version: 0.90.2

usage:

Flatten the table.

Signature

```
> flatten ...rest --all
```

Parameters

- `...rest`: Optionally flatten data by column.
- `--all`: flatten inner table one level out

Input/output types:

input	output
list<any>	list<any>
record	table

Examples

flatten a table

```
> [[N, u, s, h, e, l, l]] | flatten
```

flatten a table, get the first item

```
> [[N, u, s, h, e, l, l]] | flatten | first
```

flatten a column having a nested table

```
> [[origin, people]; [Ecuador, ([[name, meal];
['Andres', 'arepa']])]] | flatten --all |
get meal
```

restrict the flattening by passing column names

```
> [[origin, crate, versions]; [World, ([[name];
['nu-cli']]), ['0.21', '0.22']]] | flatten
 versions --all | last | get versions
```

Flatten inner table

```
> { a: b, d: [ 1 2 3 4 ], e: [ 4 3 ] } |
flatten d --all
```

fmt

version: 0.90.2

usage:

Format a number.

Signature

```
> fmt
```

Input/output types:

input	output
number	record

Examples

Get a record containing multiple formats for the number 42

```
> 42 | fmt
```

for

version: 0.90.2

usage:

Loop over a range.

Signature

```
> for (var_name) (range) (block) --numbered
```

Parameters

- `var_name`: Name of the looping variable.
- `range`: Range of the loop.
- `block`: The block to run.
- `--numbered`: return a numbered item ($it.index and $it.item)

Input/output types:

input	output
nothing	nothing

Examples

Print the square of each integer

```
> for x in [1 2 3] { print ($x * $x) }
```

Work with elements of a range

```
> for $x in 1..3 { print $x }
```

Number each item and print a message

```
> for $it in ['bob' 'fred'] --numbered {
print $"($it.index) is ($it.item)" }
```

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

format

version: 0.90.2

usage:

Various commands for formatting data.

Signature

```
> format
```

Input/output types:

input	output
nothing	string

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

Subcommands:

name	type	usage
`format date`[155]	Builtin	Format a given date using a format string.
`format duration`[156]	Builtin	Outputs duration with a specified unit of time.
`format filesize`[157]	Builtin	Converts a column of filesizes to some specified format.
`format pattern`[158]	Builtin	Format columns into a string using a simple pattern.

format date

version: 0.90.2

usage:

Format a given date using a format string.

Signature

```
> format date (format string) --list
```

Parameters

- `format string`: The desired format date.

- `--list`: lists strftime cheatsheet

[155] /commands/docs/format_date.md
[156] /commands/docs/format_duration.md
[157] /commands/docs/format_filesize.md
[158] /commands/docs/format_pattern.md

Input/output types:

input	output
datetime	string
nothing	table
string	string

Examples

Format a given date-time using the default format (RFC 2822).

```
> '2021-10-22 20:00:12 +01:00' | into datetime |
  format date
```

Format a given date-time as a string using the default format (RFC 2822).

```
> "2021-10-22 20:00:12 +01:00" | format date
```

Format the current date-time using a given format string.

```
> date now | format date "%Y-%m-%d %H:%M:%S"
```

Format the current date using a given format string.

```
> date now | format date "%Y-%m-%d %H:%M:%S"
```

Format a given date using a given format string.

```
> "2021-10-22 20:00:12 +01:00" | format date "%Y-
%m-%d"
```

format duration

version: 0.90.2

usage:

Outputs duration with a specified unit of time.

Signature

```
> format duration (format value) ...rest
```

Parameters

- `format value`: The unit in which to display the duration.
- `...rest`: For a data structure input, format duration at the given cell paths.

Input/output types:

input	output
duration	string
list<duration>	list<string>
table	table

Examples

Convert µs duration to the requested second duration as a string

```
> 1000000 µs | format duration sec
```

Convert durations to µs duration as strings

```
> [ 1sec 2sec ] | format duration µs
```

Convert duration to µs as a string if unit asked for was us

```
> 1 sec | format duration us
```

format filesize

version: 0.90.2

usage:

Converts a column of filesizes to some specified format.

Signature

```
> format filesize (format value) ...rest
```

Parameters

- `format value`: The format into which convert the file sizes.
- `...rest`: For a data structure input, format filesizes at the given cell paths.

Input/output types:

input	output
filesize	string
record	record
table	table

Examples

Convert the size column to KB

```
> ls | format filesize KB size
```

Convert the apparent column to B

```
> du | format filesize B apparent
```

Convert the size data to MB

```
> 4 Gb | format filesize MB
```

format pattern

version: 0.90.2

usage:

Format columns into a string using a simple pattern.

Signature

```
> format pattern (pattern)
```

Parameters

- `pattern`: the pattern to output. e.g.) "{foo}: {bar}"

Input/output types:

input	output
record	any
table	list<string>

Examples

Print filenames with their sizes

```
> ls | format pattern '{name}: {size}'
```

Print elements from some columns of a table

```
> [[col1, col2]; [v1, v2] [v3, v4]] | format
pattern '{col2}'
```

from

version: 0.90.2

usage:

Parse a string or binary data into structured data.

Signature

`> from`

Input/output types:

input	output
nothing	string

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

Subcommands:

name	type	usage
`from csv`[159]	Builtin	Parse text as .csv and create table.
`from eml`[160]	Builtin,Plugin	Parse text as .eml and create record.
`from ics`[161]	Builtin,Plugin	Parse text as .ics and create table.
`from ini`[162]	Builtin,Plugin	Parse text as .ini and create table.
`from json`[163]	Builtin	Convert from json to structured data.
`from nuon`[164]	Builtin	Convert from nuon to structured data.
`from ods`[165]	Builtin	Parse OpenDocument Spreadsheet(.ods) data and create table.
`from ssv`[166]	Builtin	Parse text as space-separated values and create a table. The default minimum number of spaces counted as a separator is 2.
`from toml`[167]	Builtin	Parse text as .toml and create record.
`from tsv`[168]	Builtin	Parse text as .tsv and create table.
`from url`[169]	Builtin	Parse url-encoded string as a record.
`from vcf`[170]	Builtin,Plugin	Parse text as .vcf and create table.
`from xlsx`[171]	Builtin	Parse binary Excel(.xlsx) data and create table.
`from xml`[172]	Builtin	Parse text as .xml and create record.
`from yaml`[173]	Builtin	Parse text as .yaml/.yml and create table.
`from yml`[174]	Builtin	Parse text as .yaml/.yml and create table.

from csv

version: 0.90.2

usage:

Parse text as .csv and create table.

Signature

```
> from csv --separator --comment --quote --escape --noheaders
--flexible --no-infer --trim
```

Parameters

- `--separator {string}`: a character to separate columns (either single char or 4 byte unicode sequence), defaults to ','
- `--comment {string}`: a comment character to ignore lines starting with it
- `--quote {string}`: a quote character to ignore separators in strings, defaults to '"'
- `--escape {string}`: an escape character for strings containing the quote character
- `--noheaders`: don't treat the first row as column names
- `--flexible`: allow the number of fields in records to be variable

[160] /commands/docs/from_eml.md
[161] /commands/docs/from_ics.md
[162] /commands/docs/from_ini.md
[163] /commands/docs/from_json.md
[164] /commands/docs/from_nuon.md
[165] /commands/docs/from_ods.md
[166] /commands/docs/from_ssv.md
[167] /commands/docs/from_toml.md
[168] /commands/docs/from_tsv.md
[169] /commands/docs/from_url.md
[170] /commands/docs/from_vcf.md
[171] /commands/docs/from_xlsx.md
[172] /commands/docs/from_xml.md
[173] /commands/docs/from_yaml.md
[174] /commands/docs/from_yml.md

- `--no-infer`: no field type inferencing
- `--trim {string}`: drop leading and trailing whitespaces around headers names and/or field values

Input/output types:

input	output
string	table

Examples

Convert comma-separated data to a table

```
> "ColA,ColB 1,2" | from csv
```

Convert comma-separated data to a table, ignoring headers

```
> open data.txt | from csv --noheaders
```

Convert semicolon-separated data to a table

```
> open data.txt | from csv --separator ';'
```

Convert comma-separated data to a table, ignoring lines starting with '#'

```
> open data.txt | from csv --comment '#'
```

Convert comma-separated data to a table, dropping all possible whitespaces around header names and field values

```
> open data.txt | from csv --trim all
```

Convert comma-separated data to a table, dropping all possible whitespaces around header names

```
> open data.txt | from csv --trim headers
```

Convert comma-separated data to a table, dropping all possible whitespaces around field values

```
> open data.txt | from csv --trim fields
```

from eml

version: 0.90.2

usage:

Parse text as .eml and create record.

Signature

```
> from eml --preview-body
```

Parameters

- `--preview-body {int}`: How many bytes of the body to preview

Input/output types:

input	output
string	record

Examples

Convert eml structured data into record

```
> 'From: test@email.com Subject: Welcome To: someone@somewhere.
com Test' | from eml
```

Convert eml structured data into record

```
> 'From: test@email.com Subject: Welcome To: someone@somewhere.
com Test' | from eml -b 1
```

from ics

version: 0.90.2

usage:

Parse text as .ics and create table.

Signature

```
> from ics
```

Input/output types:

input	output
string	table

Examples

Converts ics formatted string to table

```
> ' BEGIN:VCALENDAR             END:VCALENDAR '   |
from ics
```

from ini

version: 0.90.2

usage:

Parse text as .ini and create table.

Signature

```
> from ini
```

Input/output types:

input	output
string	record

Examples

Converts ini formatted string to record

```
> ' [foo] a=1 b=2 ' | from ini
```

from json

version: 0.90.2

usage:

Convert from json to structured data.

Signature

```
> from json --objects --strict
```

Parameters

- `--objects`: treat each line as a separate value
- `--strict`: follow the json specification exactly

Input/output types:

input	output
string	any

Examples

Converts json formatted string to table

```
> '{ "a": 1 }' | from json
```

Converts json formatted string to table

```
> '{ "a": 1, "b": [1, 2] }' | from json
```

Parse json strictly which will error on comments and trailing commas

```
> '{ "a": 1, "b": 2 }' | from json -s
```

from nuon

version: 0.90.2

usage:

Convert from nuon to structured data.

Signature

```
> from nuon
```

Input/output types:

input	output
string	any

Examples

Converts nuon formatted string to table

```
> '{ a:1 }' | from nuon
```

Converts nuon formatted string to table

```
> '{ a:1, b: [1, 2] }' | from nuon
```

from ods

version: 0.90.2

usage:

Parse OpenDocument Spreadsheet(.ods) data and create table.

Signature

`> from ods --sheets`

Parameters

- `--sheets {list<string>}`: Only convert specified sheets

Input/output types:

input	output
string	table

Examples

Convert binary .ods data to a table

```
> open --raw test.ods | from ods
```

Convert binary .ods data to a table, specifying the tables

```
> open -- raw test.ods | from ods -- sheets [
Spreadsheet1 ]
```

from ssv

version: 0.90.2

usage:

Parse text as space-separated values and create a table. The default minimum number of spaces counted as a separator is 2.

Signature

`> from ssv --noheaders --aligned-columns --minimum-spaces`

Parameters

- `--noheaders`: don't treat the first row as column names
- `--aligned-columns`: assume columns are aligned
- `--minimum-spaces {int}`: the minimum spaces to separate columns

Input/output types:

input	output
string	table

Examples

Converts ssv formatted string to table

```
> ' FOO   BAR 1   2 ' | from ssv
```

Converts ssv formatted string to table but not treating the first row as column names

```
> ' FOO   BAR 1   2 ' | from ssv -- noheaders
```

from toml

version: 0.90.2

usage:

Parse text as .toml and create record.

Signature

```
> from toml
```

Input/output types:

input	output
string	record

Examples

Converts toml formatted string to record

```
> 'a = 1' | from toml
```

Converts toml formatted string to record

```
> 'a = 1 b = [1, 2]' | from toml
```

from tsv

version: 0.90.2

usage:

Parse text as .tsv and create table.

Signature

```
> from tsv --comment --quote --escape --noheaders --flexible
--no-infer --trim
```

Parameters

- `--comment {string}`: a comment character to ignore lines starting with it
- `--quote {string}`: a quote character to ignore separators in strings, defaults to '"'

- `--escape {string}`: an escape character for strings containing the quote character
- `--noheaders`: don't treat the first row as column names
- `--flexible`: allow the number of fields in records to be variable
- `--no-infer`: no field type inferencing
- `--trim {string}`: drop leading and trailing whitespaces around headers names and/or field values

Input/output types:

input	output
string	table

Examples

Convert tab-separated data to a table

```
> " ColA ColB 1 2 " | from tsv
```

Create a tsv file with header columns and open it

```
> $' c1(char tab)c2(char tab)c3(char nl)1(char tab)2(
char tab)3 ' | save tsv-data | open tsv-data |
from tsv
```

Create a tsv file without header columns and open it

```
> $' a1(char tab)b1(char tab)c1(char nl)a2(char tab)b2(
char tab)c2 ' | save tsv-data | open tsv-data |
  from tsv --noheaders
```

Create a tsv file without header columns and open it, removing all unnecessary whitespaces

```
> $' a1(char tab)b1(char tab)c1(char nl)a2(char tab)b2(
char tab)c2 ' | save tsv-data | open tsv-data |
  from tsv --trim all
```

Create a tsv file without header columns and open it, removing all unnecessary whitespaces in the header names

```
> $'a1(char tab)b1(char tab)c1(char nl)a2(char tab)b2(
char tab)c2' | save tsv-data | open tsv-data |
  from tsv --trim headers
```

Create a tsv file without header columns and open it, removing all unnecessary whitespaces in the field values

```
> $'a1(char tab)b1(char tab)c1(char nl)a2(char tab)b2(
char tab)c2' | save tsv-data | open tsv-data |
  from tsv --trim fields
```

from url

version: 0.90.2

usage:

Parse url-encoded string as a record.

Signature

```
> from url
```

Input/output types:

input	output
string	record

Examples

Convert url encoded string into a record

```
> 'bread=baguette&cheese=comt%C3%A9&meat=ham&fat=butter
' | from url
```

from vcf

version: 0.90.2

usage:

Parse text as .vcf and create table.

Signature

```
> from vcf
```

Input/output types:

input	output
string	table

Examples

Converts ics formatted string to table

```
> 'BEGIN:VCARD N:Foo FN:Bar EMAIL:foo@bar.com END:VCARD
' | from vcf
```

from xlsx

version: 0.90.2

usage:

Parse binary Excel(.xlsx) data and create table.

Signature

```
> from xlsx --sheets
```

Parameters

- `--sheets {list<string>}`: Only convert specified sheets

Input/output types:

input	output
binary	table

Examples

Convert binary .xlsx data to a table

```
> open --raw test.xlsx | from xlsx
```

Convert binary .xlsx data to a table, specifying the tables

```
> open --raw test.xlsx | from xlsx --sheets [
Spreadsheet1 ]
```

from xml

version: 0.90.2

usage:

Parse text as .xml and create record.

Signature

`> from xml --keep-comments --keep-pi`

Parameters

- `--keep-comments`: add comment nodes to result
- `--keep-pi`: add processing instruction nodes to result

Input/output types:

input	output
string	record

Examples

Converts xml formatted string to record

```
> '<?xml version="1.0" encoding="UTF-8"?> <note>  <remember>Event</
remember> </note> ' | from xml
```

Notes

```
Every XML entry is represented via a record with tag, attribute
and content fields. To represent different types of entries
different values are written to this fields: 1. Tag entry:
```

> `{tag: <tag name> attrs: {<attr name>: "<string value>" ...} content: [<entries>]}` 2. Comment entry: `{tag: '!' attrs: null content: "<comment string>"}` 3. Processing instruction (PI): `{tag: '?<pi name>' attrs: null content: "<pi content string>"}` 4. Text: `{tag: null attrs: null content: "<text>"}`. Unlike to xml command all null values are always present and text is never represented via plain string. This way content of every tag is always a table and is easier to parse

from yaml

version: 0.90.2

usage:

Parse text as .yaml/.yml and create table.

Signature

`> from yaml`

Input/output types:

input	output
string	any

Examples

Converts yaml formatted string to table

```
> 'a: 1' | from yaml
```

Converts yaml formatted string to table

```
> '[ a: 1, b: [1, 2] ]' | from yaml
```

from yml

version: 0.90.2

usage:

Parse text as .yaml/.yml and create table.

Signature

```
> from yml
```

Input/output types:

input	output
string	any

Examples

Converts yaml formatted string to table

```
> 'a: 1' | from yaml
```

Converts yaml formatted string to table

```
> '[ a: 1, b: [1, 2] ]' | from yaml
```

generate

version: 0.90.2

usage:

Generate a list of values by successively invoking a closure.

Signature

```
> generate (initial) (closure)
```

Parameters

- `initial`: Initial value.
- `closure`: Generator function.

Input/output types:

input	output
list<any>	list<any>
nothing	list<any>

Examples

Generate a sequence of numbers

```
> generate 0 { |i| if $ i <= 10 { { out: $ i , next:
 ( $ i + 2) } } }
```

Generate a stream of fibonacci numbers

```
> generate [ 0, 1 ] { |fib| { out: $ fib .0 , next:
[ $ fib .1, ( $ fib .0 + $ fib .1) ] } } | first 10
```

Notes

```
The generator closure accepts a single argument and returns
a record containing two optional keys: 'out' and 'next'.
 Each invocation, the 'out' value, if present, is added
to the stream. If a 'next' key is present, it is used as
the next argument to the closure, otherwise generation
stops.
```

get

version: 0.90.2

usage:

Extract data using a cell path.

Signature

```
> get (cell_path) ...rest --ignore-errors --sensitive
```

Parameters

- `cell_path`: The cell path to the data.
- `...rest`: Additional cell paths.
- `--ignore-errors`: ignore missing data (make all cell path members optional)
- `--sensitive`: get path in a case sensitive manner

Input/output types:

input	output
list<any>	any
record	any
table	any

Examples

Get an item from a list

```
> [0 1 2] | get 1
```

Get a column from a table

```
> [{A: A0}] | get A
```

Get a cell from a table

```
> [{A: A0}] | get 0.A
```

Extract the name of the 3rd record in a list (same as `ls | $in.name`)

```
> ls | get name.2
```

Extract the name of the 3rd record in a list

```
> ls | get 2.name
```

Extract the cpu list from the sys information record

```
> sys | get cpu
```

Getting Path/PATH in a case insensitive way

```
> $env | get paTH
```

Getting Path in a case sensitive way, won't work for 'PATH'

```
> $env | get --sensitive Path
```

Notes

```
This is equivalent to using the cell path access syntax:
 `$env.OS` is the same as `$env | get OS`. If multiple
cell paths are given, this will produce a list of values.
```

glob

version: 0.90.2

usage:

Creates a list of files and/or folders based on the glob pattern provided.

Signature

`> glob (glob) --depth --no-dir --no-file --no-symlink --exclude`

Parameters

- `glob`: The glob expression.
- `--depth {int}`: directory depth to search
- `--no-dir`: Whether to filter out directories from the returned paths
- `--no-file`: Whether to filter out files from the returned paths
- `--no-symlink`: Whether to filter out symlinks from the returned paths
- `--exclude {list<string>}`: Patterns to exclude from the search: `glob` will not walk the inside of directories matching the excluded patterns.

Input/output types:

input	output
nothing	list<string>

Examples

Search for *.rs files

```
> glob *.rs
```

Search for _.rs and _.toml files recursively up to 2 folders deep

```
> glob **/*.{rs,toml} --depth 2
```

Search for files and folders that begin with uppercase C or lowercase c

```
> glob "[Cc]*"
```

Search for files and folders like abc or xyz substituting a character for ?

```
> glob "{a?c,x?z}"
```

A case-insensitive search for files and folders that begin with c

```
> glob "(?i)c*"
```

Search for files for folders that do not begin with c, C, b, M, or s

```
> glob "[!cCbMs]*"
```

Search for files or folders with 3 a's in a row in the name

```
> glob <a*:3>
```

Search for files or folders with only a, b, c, or d in the file name between 1 and 10 times

```
> glob <[a-d]:1,10>
```

Search for folders that begin with an uppercase ASCII letter, ignoring files and symlinks

```
> glob "[A-Z]*" --no-file --no-symlink
```

Search for files named tsconfig.json that are not in node_modules directories

```
> glob **/tsconfig.json --exclude [**/node_-
modules/**]
```

Search for all files that are not in the target nor .git directories

```
> glob **/* --exclude [**/target/** **/.git/
** */]
```

Notes

```
For more glob pattern help, please refer to https://docs.
rs/crate/wax/latest
```

grid

version: 0.90.2

usage:

Renders the output to a textual terminal grid.

Signature

`> grid --width --color --separator`

Parameters

- `--width {int}`: number of terminal columns wide (not output columns)
- `--color`: draw output with color
- `--separator {string}`: character to separate grid with

Input/output types:

input	output
list<any>	string
record	string

Examples

Render a simple list to a grid

```
> [ 1 2 3 a b c ] | grid
```

The above example is the same as:

```
> [ 1 2 3 a b c ] | wrap name | grid
```

Render a record to a grid

```
> { name: 'foo', b: 1, c: 2 } | grid
```

Render a list of records to a grid

```
> [ { name: 'A', v: 1 } { name: 'B', v: 2 } {
name: 'C', v: 3 } ] | grid
```

Render a table with ‘name’ column in it to a grid

```
> [ [ name patch ] ; [ 0.1.0 false ] [ 0.1.1 true
] [ 0.2.0 false ] ] | grid
```

Notes

```
grid was built to give a concise gridded layout for ls.
 however, it determines what to put in the grid by looking
for a column named 'name'. this works great for tables
and records but for lists we need to do something different.
 such as with '[one two three] | grid' it creates a fake
column called 'name' for these values so that it prints
out the list properly.
```

group-by

version: 0.90.2

usage:

Splits a list or table into groups, and returns a record containing those groups.

Signature

```
> group-by (grouper) --to-table
```

Parameters

- `grouper`: The path to the column to group on.
- `--to-table`: Return a table with "groups" and "items" columns

Input/output types:

input	output
list<any>	any

Examples

Group items by the "type" column's values

```
> ls | group-by type
```

Group items by the "foo" column's values, ignoring records without a "foo" column

```
> open cool.json | group-by foo ?
```

Group using a block which is evaluated against each input value

```
> [ foo.txt bar.csv baz.txt ] | group-by { path
parse | get extension }
```

You can also group by raw values by leaving out the argument

```
> ['1' '3' '1' '3' '2' '1' '1'] |
group-by
```

You can also output a table instead of a record

```
> ['1' '3' '1' '3' '2' '1' '1'] |
group-by --to-table
```

group

version: 0.90.2

usage:

Groups input into groups of `group_size`.

Signature

`> group (group_size)`

Parameters

- `group_size`: The size of each group.

Input/output types:

input	output
list<any>	list<list<any>>

Examples

Group the a list by pairs

```
> [1 2 3 4] | group 2
```

gstat

version: 0.90.2

usage:

Get the git status of a repo

Signature

`> gstat (path)`

Parameters

- `path`: path to repo

Input/output types:

input	output
any	any

hash

version: 0.90.2

usage:

Apply hash function.

Signature

`> hash`

Input/output types:

input	output
nothing	string

Notes

> ```
> You must use one of the following subcommands. Using this
> command as-is will only produce this help message.
> ```

Subcommands:

name	type	usage
`hash md5`[175]	Builtin	Hash a value using the md5 hash algorithm.
`hash sha256`[176]	Builtin	Hash a value using the sha256 hash algorithm.

hash md5

version: 0.90.2

usage:

Hash a value using the md5 hash algorithm.

[175] `/commands/docs/hash_md5.md`
[176] `/commands/docs/hash_sha256.md`

Signature

```
> hash md5 ...rest --binary
```

Parameters

- `...rest`: Optionally md5 hash data by cell path.
- `--binary`: Output binary instead of hexadecimal representation

Input/output types:

input	output
record	record
string	any
table	table

Examples

Return the md5 hash of a string, hex-encoded

```
> 'abcdefghijklmnopqrstuvwxyz' | hash md5
```

Return the md5 hash of a string, as binary

```
> 'abcdefghijklmnopqrstuvwxyz' | hash md5 --binary
```

Return the md5 hash of a file's contents

```
> open ./nu_0_24_1_windows.zip | hash md5
```

hash sha256

version: 0.90.2

usage:

Hash a value using the sha256 hash algorithm.

Signature

```
> hash sha256 ...rest --binary
```

Parameters

- `...rest`: Optionally sha256 hash data by cell path.
- `--binary`: Output binary instead of hexadecimal representation

Input/output types:

input	output
record	record
string	any
table	table

Examples

Return the sha256 hash of a string, hex-encoded

```
> 'abcdefghijklmnopqrstuvwxyz' | hash sha256
```

Return the sha256 hash of a string, as binary

```
> 'abcdefghijklmnopqrstuvwxyz' | hash sha256 --
binary
```

Return the sha256 hash of a file's contents

```
> open ./nu_0_24_1_windows.zip | hash sha256
```

headers

version: 0.90.2

usage:

Use the first row of the table as column names.

Signature

```
> headers
```

Input/output types:

input	output
list<any>	table
table	table

Examples

Sets the column names for a table created by `split column`

```
> "a b c|1 2 3" | split row "|" | split
column " " | headers
```

Columns which don't have data in their first row are removed

```
> "a b c|1 2 3|1 2 3 4" | split row "|" |
split column " " | headers
```

help

version: 0.90.2

usage:

Display help information about different parts of Nushell.

Signature

`> help ...rest --find`

Parameters

- `...rest`: The name of command, alias or module to get help on.
- `--find {string}`: string to find in command names, usage, and search terms

Input/output types:

input	output
nothing	any

Examples

show help for single command, alias, or module

```
> help match
```

show help for single sub-command, alias, or module

```
> help str lpad
```

search for string in command names, usage and search terms

```
> help --find char
```

Notes

```
`help word` searches for "word" in commands, aliases and
modules, in that order.
```

Subcommands:

name	type	usage
`help aliases`[177]	Builtin	Show help on nushell aliases.
`help commands`[178]	Builtin	Show help on nushell commands.
`help escapes`[179]	Builtin	Show help on nushell string escapes.
`help externs`[180]	Builtin	Show help on nushell externs.
`help modules`[181]	Builtin	Show help on nushell modules.
`help operators`[182]	Builtin	Show help on nushell operators.

help aliases

version: 0.90.2

usage:

Show help on nushell aliases.

[177] `/commands/docs/help_aliases.md`
[178] `/commands/docs/help_commands.md`
[179] `/commands/docs/help_escapes.md`
[180] `/commands/docs/help_externs.md`
[181] `/commands/docs/help_modules.md`
[182] `/commands/docs/help_operators.md`

Signature

```
> help aliases ...rest --find
```

Parameters

- `...rest`: The name of alias to get help on.
- `--find {string}`: string to find in alias names and usage

Input/output types:

input	output
nothing	table

Examples

show all aliases

```
> help aliases
```

show help for single alias

```
> help aliases my-alias
```

search for string in alias names and usages

```
> help aliases -- find my-alias
```

help commands

version: 0.90.2

usage:

Show help on nushell commands.

Signature

```
> help commands ...rest --find
```

Parameters

- `...rest`: The name of command to get help on.
- `--find {string}`: string to find in command names, usage, and search terms

Input/output types:

input	output
nothing	table

help escapes

version: 0.90.2

usage:

Show help on nushell string escapes.

Signature

```
> help escapes
```

Input/output types:

input	output
nothing	table

help externs

version: 0.90.2

usage:

Show help on nushell externs.

Signature

```
> help externs ...rest --find
```

Parameters

- `...rest`: The name of extern to get help on.
- `--find {string}`: string to find in extern names and usage

Input/output types:

input	output
nothing	table

Examples

show all externs

```
> help externs
```

show help for single extern

```
> help externs smth
```

search for string in extern names and usages

```
> help externs --find smth
```

help modules

version: 0.90.2

usage:

Show help on nushell modules.

Signature

```
> help modules ...rest --find
```

Parameters

- `...rest`: The name of module to get help on.
- `--find {string}`: string to find in module names and usage

Input/output types:

input	output
nothing	table

Examples

show all modules

```
> help modules
```

show help for single module

```
> help modules my-module
```

search for string in module names and usages

```
> help modules --find my-module
```

Notes

```
When requesting help for a single module, its commands
and aliases will be highlighted if they are also available
in the current scope. Commands/aliases that were imported
under a different name (such as with a prefix after `use
some-module`) will be highlighted in parentheses.
```

help operators

version: 0.90.2

usage:

Show help on nushell operators.

Signature

```
> help operators
```

Input/output types:

input	output
nothing	table

hide-env

version: 0.90.2

usage:

Hide environment variables in the current scope.

Signature

```
> hide-env ...rest --ignore-errors
```

Parameters

- `...rest`: Environment variable names to hide.
- `--ignore-errors`: do not throw an error if an environment variable was not found

Input/output types:

input	output
nothing	nothing

Examples

Hide an environment variable

```
> $env.HZ_ENV_ABC = 1; hide-env HZ_ENV_ABC; 'HZ_ENV_ABC' in (env).name
```

hide

version: 0.90.2

usage:

Hide definitions in the current scope.

Signature

```
> hide (module) (members)
```

Parameters

- `module`: Module or module file.
- `members`: Which members of the module to import.

Input/output types:

input	output
nothing	nothing

Examples

Hide the alias just defined

```
> alias lll = ls -l; hide lll
```

Hide a custom command

```
> def say-hi [] { echo 'Hi!' }; hide say-hi
```

Notes

```
Definitions are hidden by priority: First aliases, then
custom commands. This command is a parser keyword. For
details, check:  https://www.nushell.sh/book/thinking_-
in_nu.html
```

histogram

version: 0.90.2

usage:

Creates a new table with a histogram based on the column name passed in.

Signature

```
> histogram (column-name) (frequency-column-name) --percentage-
type
```

Parameters

- `column-name`: Column name to calc frequency, no need to provide if input is a list.
- `frequency-column-name`: Histogram's frequency column, default to be frequency column output.
- `--percentage-type {string}`: percentage calculate method, can be 'normalize' or 'relative', in 'normalize', defaults to be 'normalize'

Input/output types:

input	output
list<any>	table

Examples

Compute a histogram of file types

```
> ls | histogram type
```

Compute a histogram for the types of files, with frequency column named freq

```
> ls | histogram type freq
```

Compute a histogram for a list of numbers

```
> [1 2 1] | histogram
```

Compute a histogram for a list of numbers, and percentage is based on the maximum value

```
> [1 2 3 1 1 1 2 2 1 1] | histogram --percentage-type relative
```

history

version: 0.90.2

usage:

Get the command history.

Signature

`> history --clear --long`

Parameters

- `--clear`: Clears out the history entries
- `--long`: Show long listing of entries for sqlite history

Input/output types:

input	output
nothing	any

Examples

Get current history length

```
> history | length
```

Show last 5 commands you have ran

```
> history | last 5
```

Search all the commands from history that contains 'cargo'

```
> history | where command =~ cargo | get command
```

Subcommands:

name	type	usage
`history session`[183]	Builtin	Get the command history session.

history session

version: 0.90.2

usage:

Get the command history session.

[183] `/commands/docs/history_session.md`

Signature

```
> history session
```

Input/output types:

input	output
nothing	int

Examples

Get current history session

```
> history session
```

http

version: 0.90.2

usage:

Various commands for working with http methods.

Signature

```
> http
```

Input/output types:

input	output
nothing	string

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

Subcommands:

name	type	usage
http delete[184]	Builtin	Delete the specified resource.
http get[185]	Builtin	Fetch the contents from a URL.
http head[186]	Builtin	Get the headers from a URL.
http options[187]	Builtin	Requests permitted communication options for a given URL.
http patch[188]	Builtin	Patch a body to a URL.
http post[189]	Builtin	Post a body to a URL.
http put[190]	Builtin	Put a body to a URL.

http delete

version: 0.90.2

usage:

Delete the specified resource.

Signature

```
> http delete (URL) --user --password --data --content-type
--max-time --headers --raw --insecure --full --allow-errors
--redirect-mode
```

[184] /commands/docs/http_delete.md
[185] /commands/docs/http_get.md
[186] /commands/docs/http_head.md
[187] /commands/docs/http_options.md
[188] /commands/docs/http_patch.md
[189] /commands/docs/http_post.md
[190] /commands/docs/http_put.md

Parameters

- `URL`: The URL to fetch the contents from.
- `--user {any}`: the username when authenticating
- `--password {any}`: the password when authenticating
- `--data {any}`: the content to post
- `--content-type {any}`: the MIME type of content to post
- `--max-time {int}`: timeout period in seconds
- `--headers {any}`: custom headers you want to add
- `--raw`: fetch contents as text rather than a table
- `--insecure`: allow insecure server connections when using SSL
- `--full`: returns the full response instead of only the body
- `--allow-errors`: do not fail if the server returns an error code
- `--redirect-mode {string}`: What to do when encountering redirects. Default: 'follow'. Valid options: 'follow' ('f'), 'manual' ('m'), 'error' ('e').

Input/output types:

input	output
nothing	any

Examples

http delete from example.com

```
> http delete https://www.example.com
```

http delete from example.com, with username and password

```
> http delete --user myuser --password mypass https:
//www.example.com
```

http delete from example.com, with custom header

```
> http delete --headers [my-header-key my-header-value] https://www.example.com
```

http delete from example.com, with body

```
> http delete --data 'body' https://www.example.com
```

http delete from example.com, with JSON body

```
> http delete --content-type application/json --data { field: value } https://www.example.com
```

Notes

```
Performs HTTP DELETE operation.
```

http get

version: 0.90.2

usage:

Fetch the contents from a URL.

Signature

`> http get (URL) --user --password --max-time --headers --raw --insecure --full --allow-errors --redirect-mode`

Parameters

- `URL`: The URL to fetch the contents from.
- `--user {any}`: the username when authenticating
- `--password {any}`: the password when authenticating
- `--max-time {int}`: timeout period in seconds
- `--headers {any}`: custom headers you want to add
- `--raw`: fetch contents as text rather than a table

- `--insecure`: allow insecure server connections when using SSL
- `--full`: returns the full response instead of only the body
- `--allow-errors`: do not fail if the server returns an error code
- `--redirect-mode {string}`: What to do when encountering redirects. Default: 'follow'. Valid options: 'follow' ('f'), 'manual' ('m'), 'error' ('e').

Input/output types:

input	output
nothing	any

Examples

Get content from example.com

```
> http get https://www.example.com
```

Get content from example.com, with username and password

```
> http get --user myuser --password mypass https://www.example.com
```

Get content from example.com, with custom header

```
> http get --headers [my-header-key my-header-value] https://www.example.com
```

Get content from example.com, with custom headers

```
> http get --headers [my-header-key-A my-header-value-A my-header-key-B my-header-value-B] https://www.example.com
```

Notes

```
Performs HTTP GET operation.
```

http head

version: 0.90.2

usage:

Get the headers from a URL.

Signature

`> http head (URL) --user --password --max-time --headers --insecure --redirect-mode`

Parameters

- `URL`: The URL to fetch the contents from.
- `--user {any}`: the username when authenticating
- `--password {any}`: the password when authenticating
- `--max-time {int}`: timeout period in seconds
- `--headers {any}`: custom headers you want to add
- `--insecure`: allow insecure server connections when using SSL
- `--redirect-mode {string}`: What to do when encountering redirects. Default: 'follow'. Valid options: 'follow' ('f'), 'manual' ('m'), 'error' ('e').

Input/output types:

input	output
nothing	any

Examples

Get headers from example.com

```
> http head https://www.example.com
```

Get headers from example.com, with username and password

```
> http head --user myuser --password mypass https://www.example.com
```

Get headers from example.com, with custom header

```
> http head --headers [my-header-key my-header-value] https://www.example.com
```

Notes

```
Performs HTTP HEAD operation.
```

http options

version: 0.90.2

usage:

Requests permitted communication options for a given URL.

Signature

`> http options (URL) --user --password --max-time --headers --insecure --allow-errors`

Parameters

- `URL`: The URL to fetch the options from.
- `--user {any}`: the username when authenticating
- `--password {any}`: the password when authenticating
- `--max-time {int}`: timeout period in seconds
- `--headers {any}`: custom headers you want to add
- `--insecure`: allow insecure server connections when using SSL
- `--allow-errors`: do not fail if the server returns an error code

Input/output types:

input	output
nothing	any

Examples

Get options from example.com

```
> http options https://www.example.com
```

Get options from example.com, with username and password

```
> http options --user myuser --password mypass https:
//www.example.com
```

Get options from example.com, with custom header

```
> http options --headers [my-header-key my-header
-value] https://www.example.com
```

Get options from example.com, with custom headers

```
> http options --headers [my-header-key-A my-
header-value-A my-header-key-B my-header-value
-B] https://www.example.com
```

Simulate a browser cross-origin preflight request from www.example.com[191] to media.example.com

```
> http options https://media.example.com/api/ --head-
ers [Origin https://www.example.com Access-Control
-Request-Headers "Content-Type, X-Custom-Header"
Access-Control-Request-Method GET]
```

Notes

```
Performs an HTTP OPTIONS request. Most commonly used for
making CORS preflight requests.
```

http patch

version: 0.90.2

usage:

Patch a body to a URL.

[191] `http://www.example.com`

Signature

`> http patch (URL) (data) --user --password --content-type --max-time --headers --raw --insecure --full --allow-errors --redirect-mode`

Parameters

- `URL`: The URL to post to.
- `data`: The contents of the post body.
- `--user {any}`: the username when authenticating
- `--password {any}`: the password when authenticating
- `--content-type {any}`: the MIME type of content to post
- `--max-time {int}`: timeout period in seconds
- `--headers {any}`: custom headers you want to add
- `--raw`: return values as a string instead of a table
- `--insecure`: allow insecure server connections when using SSL
- `--full`: returns the full response instead of only the body
- `--allow-errors`: do not fail if the server returns an error code
- `--redirect-mode {string}`: What to do when encountering redirects. Default: 'follow'. Valid options: 'follow' ('f'), 'manual' ('m'), 'error' ('e').

Input/output types:

input	output
nothing	any

Examples

Patch content to example.com

```
> http patch https://www.example.com 'body'
```

Patch content to example.com, with username and password

```
> http patch --user myuser --password mypass https:
//www.example.com 'body'
```

Patch content to example.com, with custom header

```
> http patch --headers [my-header-key my-header
-value] https://www.example.com
```

Patch content to example.com, with JSON body

```
> http patch --content-type application/json https:/
/www.example.com { field: value }
```

Notes

```
Performs HTTP PATCH operation.
```

http post

version: 0.90.2

usage:

Post a body to a URL.

Signature

```
> http post (URL) (data) --user --password --content-type -
-max-time --headers --raw --insecure --full --allow-errors -
-redirect-mode
```

Parameters

- `URL`: The URL to post to.
- `data`: The contents of the post body.
- `--user {any}`: the username when authenticating
- `--password {any}`: the password when authenticating
- `--content-type {any}`: the MIME type of content to post
- `--max-time {int}`: timeout period in seconds

- `--headers {any}`: custom headers you want to add
- `--raw`: return values as a string instead of a table
- `--insecure`: allow insecure server connections when using SSL
- `--full`: returns the full response instead of only the body
- `--allow-errors`: do not fail if the server returns an error code
- `--redirect-mode {string}`: What to do when encountering redirects. Default: 'follow'. Valid options: 'follow' ('f'), 'manual' ('m'), 'error' ('e').

Input/output types:

input	output
nothing	any

Examples

Post content to example.com

```
> http post https://www.example.com 'body'
```

Post content to example.com, with username and password

```
> http post --user myuser --password mypass https://www.example.com 'body'
```

Post content to example.com, with custom header

```
> http post --headers [my-header-key my-header-value] https://www.example.com
```

Post content to example.com, with JSON body

```
> http post --content-type application/json https://www.example.com { field: value }
```

Notes

```
Performs HTTP POST operation.
```

http put

version: 0.90.2

usage:

Put a body to a URL.

Signature

```
> http put (URL) (data) --user --password --content-type --
max-time --headers --raw --insecure --full --allow-errors -
-redirect-mode
```

Parameters

- `URL`: The URL to post to.
- `data`: The contents of the post body.
- `--user {any}`: the username when authenticating
- `--password {any}`: the password when authenticating
- `--content-type {any}`: the MIME type of content to post
- `--max-time {int}`: timeout period in seconds
- `--headers {any}`: custom headers you want to add
- `--raw`: return values as a string instead of a table
- `--insecure`: allow insecure server connections when using SSL
- `--full`: returns the full response instead of only the body
- `--allow-errors`: do not fail if the server returns an error code
- `--redirect-mode {string}`: What to do when encountering redirects. Default: 'follow'. Valid options: 'follow' ('f'), 'manual' ('m'), 'error' ('e').

Input/output types:

input	output
nothing	any

Examples

Put content to example.com

```
> http put https://www.example.com 'body'
```

Put content to example.com, with username and password

```
> http put --user myuser --password mypass https:/
/www.example.com 'body'
```

Put content to example.com, with custom header

```
> http put --headers [my-header-key my-header-
value] https://www.example.com
```

Put content to example.com, with JSON body

```
> http put --content-type application/json https://
www.example.com { field: value }
```

Notes

```
Performs HTTP PUT operation.
```

if

version: 0.90.2

usage:

Conditionally run a block.

Signature

`> if (cond) (then_block) (else_expression)`

Parameters

- `cond`: Condition to check.
- `then_block`: Block to run if check succeeds.
- `else_expression`: Expression or block to run when the condition is false.

Input/output types:

input	output
any	any

Examples

Output a value if a condition matches, otherwise return nothing

```
> if 2 < 3 { 'yes!' }
```

Output a value if a condition matches, else return another value

```
> if 5 < 3 { 'yes!' } else { 'no!' }
```

Chain multiple if's together

```
> if 5 < 3 { 'yes!' } else if 4 < 5 { 'no!'
  } else { 'okay!' }
```

ignore

version: 0.90.2

usage:

Ignore the output of the previous command in the pipeline.

Signature

```
> ignore
```

Input/output types:

input	output
any	nothing

Examples

Ignore the output of an echo command

```
> echo done | ignore
```

inc

version: 0.90.2

usage:

Increment a value or version. Optionally use the column of a table.

Signature

```
> inc (cell_path) --major --minor --patch
```

Parameters

- `cell_path`: cell path to update
- `--major`: increment the major version (eg 1.2.1 -> 2.0.0)
- `--minor`: increment the minor version (eg 1.2.1 -> 1.3.0)
- `--patch`: increment the patch version (eg 1.2.1 -> 1.2.2)

Input/output types:

input	output
any	any

input

version: 0.90.2

usage:

Get input from the user.

Signature

```
> input (prompt) --bytes-until-any --numchar --suppress-output
```

Parameters

- `prompt`: Prompt to show the user.
- `--bytes-until-any {string}`: read bytes (not text) until any of the given stop bytes is seen
- `--numchar {int}`: number of characters to read; suppresses output
- `--suppress-output`: don't print keystroke values

Input/output types:

input	output
nothing	any

Examples

Get input from the user, and assign to a variable

```
> let user_input = (input )
```

Get two characters from the user, and assign to a variable

```
> let user_input = (input --numchar 2 )
```

Subcommands:

name	type	usage
`input list`[192]	Builtin	Interactive list selection.
`input listen`[193]	Builtin	Listen for user interface event.

input list

version: 0.90.2

usage:

Interactive list selection.

[192] /commands/docs/input_list.md
[193] /commands/docs/input_listen.md

Signature

```
> input list (prompt) --multi --fuzzy --index --display
```

Parameters

- `prompt`: The prompt to display.
- `--multi`: Use multiple results, you can press a to toggle all options on/off
- `--fuzzy`: Use a fuzzy select.
- `--index`: Returns list indexes.
- `--display {cell-path}`: Field to use as display value

Input/output types:

input	output
list<any>	any
range	int

Examples

Return a single value from a list

```
> [1 2 3 4 5] | input list 'Rate it'
```

Return multiple values from a list

```
> [Banana Kiwi Pear Peach Strawberry] | input list
  --multi 'Add fruits to the basket'
```

Return a single record from a table with fuzzy search

```
> ls | input list --fuzzy 'Select the target'
```

Choose an item from a range

```
> 1..10 | input list
```

Return the index of a selected item

```
> [Banana Kiwi Pear Peach Strawberry] | input list
  -- index
```

Choose an item from a table using a column as display value

```
> [[name price]; [Banana 12] [Kiwi 4] [Pear 7
]] | input list -d name
```

Notes

```
Abort with esc or q.
```

input listen

version: 0.90.2

usage:

Listen for user interface event.

Signature

```
> input listen --types --raw
```

Parameters

- `--types {list<string>}`: Listen for event of specified types only (can be one of: focus, key, mouse, paste, resize)
- `--raw`: Add raw_code field with numeric value of keycode and raw_flags with bit mask flags

Input/output types:

input	output
nothing	record<keycode: string, modifiers: list<string>>

Examples

Listen for a keyboard shortcut and find out how nu receives it

```
> input listen --types [key]
```

Notes

```
There are 5 different type of events: focus, key, mouse,
paste, resize. Each will produce a corresponding record,
distinguished by type field:
```

```
{ type: focus event: (gained|lost) }{ type: key key_type:
 <key_type> code: <string> modifiers: [ <modifier> ...
] } { type: mouse col: <int> row: <int> kind: <string>
modifiers: [ <modifier> ... ] } { type: paste content:
<string> } { type: resize col: <int> row: <int> }
```

```
There are 6 `modifier` variants: shift, control, alt, super,
hyper, meta. There are 4 `key_type` variants:     f -
f1, f2, f3 ... keys    char - alphanumeric and special
symbols (a, A, 1, $ ...)     media - dedicated media keys
(play, pause, tracknext ...)     other - keys not falling
under previous categories (up, down, backspace, enter .
..)
```

insert

version: 0.90.2

usage:

Insert a new column, using an expression or closure to create each row's values.

Signature

```
> insert (field) (new value)
```

Parameters

- `field`: The name of the column to insert.
- `new value`: The new value to give the cell(s).

Input/output types:

input	output
list<any>	list<any>
record	record
table	table

Examples

Insert a new entry into a single record

```
> {'name': 'nu', 'stars': 5} | insert
alias 'Nushell'
```

Insert a new column into a table, populating all rows

```
> [[project, lang]; ['Nushell', 'Rust']] |
insert type 'shell'
```

Insert a new column with values computed based off the other columns

```
> [[foo]; [7] [8] [9]] | insert bar {
|row| $row.foo * 2 }
```

Insert a new value into a list at an index

```
> [1 2 4] | insert 2 3
```

Insert a new value at the end of a list

```
> [1 2 3] | insert 3 4
```

inspect

version: 0.90.2

usage:

Inspect pipeline results while running a pipeline.

Signature

```
> inspect
```

Input/output types:

input	output
any	any

Examples

Inspect pipeline results

```
> ls | inspect | get name | inspect
```

into

version: 0.90.2

usage:

Commands to convert data from one type to another.

Signature

```
> into
```

Input/output types:

input	output
nothing	string

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

Subcommands:

name	type	usage
`into binary`[194]	Builtin	Convert value to a binary primitive.
`into bits`[195]	Builtin	Convert value to a binary primitive.
`into bool`[196]	Builtin	Convert value to boolean.
`into cell-path`[197]	Builtin	Convert value to a cell-path.
`into datetime`[198]	Builtin	Convert text or timestamp into a datetime.
`into duration`[199]	Builtin	Convert value to duration.
`into filesize`[200]	Builtin	Convert value to filesize.
`into float`[201]	Builtin	Convert data into floating point number.
`into int`[202]	Builtin	Convert value to integer.
`into record`[203]	Builtin	Convert value to record.
`into sqlite`[204]	Builtin	Convert table into a SQLite database.
`into string`[205]	Builtin	Convert value to string.
`into value`[206]	Builtin	Infer nushell datatype for each cell.

[194] /commands/docs/into_binary.md
[195] /commands/docs/into_bits.md
[196] /commands/docs/into_bool.md
[197] /commands/docs/into_cell-path.md
[198] /commands/docs/into_datetime.md
[199] /commands/docs/into_duration.md
[200] /commands/docs/into_filesize.md
[201] /commands/docs/into_float.md
[202] /commands/docs/into_int.md
[203] /commands/docs/into_record.md
[204] /commands/docs/into_sqlite.md
[205] /commands/docs/into_string.md

into binary

version: 0.90.2

usage:

Convert value to a binary primitive.

Signature

```
> into binary ...rest --compact
```

Parameters

- `...rest`: For a data structure input, convert data at the given cell paths.
- `--compact`: output without padding zeros

Input/output types:

input	output
binary	binary
bool	binary
datetime	binary
filesize	binary
int	binary
number	binary
record	record
string	binary
table	table

Examples

convert string to a nushell binary primitive

```
> 'This is a string that is exactly 52 characters long.
 ' | into binary
```

convert a number to a nushell binary primitive

206 /commands/docs/into_value.md

```
> 1 | into binary
```

convert a boolean to a nushell binary primitive

```
> true | into binary
```

convert a filesize to a nushell binary primitive

```
> ls | where name == LICENSE | get size | into
binary
```

convert a filepath to a nushell binary primitive

```
> ls | where name == LICENSE | get name | path
expand | into binary
```

convert a float to a nushell binary primitive

```
> 1.234 | into binary
```

convert an int to a nushell binary primitive with compact enabled

```
> 10 | into binary --compact
```

into bits

version: 0.90.2

usage:

Convert value to a binary primitive.

Signature

```
> into bits ...rest
```

Parameters

- `...rest`: for a data structure input, convert data at the given cell paths

Input/output types:

input	output
binary	string
bool	string
datetime	string
duration	string
filesize	string
int	string
record	record
string	string
table	table

Examples

convert a binary value into a string, padded to 8 places with 0s

```
> 01 b | into bits
```

convert an int into a string, padded to 8 places with 0s

```
> 1 | into bits
```

convert a filesize value into a string, padded to 8 places with 0s

```
> 1 b | into bits
```

convert a duration value into a string, padded to 8 places with 0s

```
> 1 ns | into bits
```

convert a boolean value into a string, padded to 8 places with 0s

```
> true | into bits
```

convert a datetime value into a string, padded to 8 places with 0s

```
> 2023 -04-17T01:02:03 | into bits
```

convert a string into a raw binary string, padded with 0s to 8 places

```
> ' nushell.sh ' | into bits
```

into bool

version: 0.90.2

usage:

Convert value to boolean.

Signature

```
> into bool ...rest
```

Parameters

- `...rest`: For a data structure input, convert data at the given cell paths.

Input/output types:

input	output
bool	bool
int	bool
list<any>	table
number	bool
record	record
string	bool
table	table

Examples

Convert value to boolean in table

```
> [[value]; ['false'] ['1'] [0] [1.0]
[true]] | into bool value
```

Convert bool to boolean

```
> true | into bool
```

convert int to boolean

```
> 1 | into bool
```

convert float to boolean

```
> 0.3 | into bool
```

convert float string to boolean

```
> '0.0' | into bool
```

convert string to boolean

```
> 'true' | into bool
```

into cell-path

version: 0.90.2

usage:

Convert value to a cell-path.

Signature

`> into cell-path`

Input/output types:

input	output
int	cell-path
list<any>	cell-path
list<record<value: any, optional: bool>>	cell-path

Examples

Convert integer into cell path

```
> 5 | into cell-path
```

Convert string into cell path

```
> 'some.path' | split row '.' | into cell-
path
```

Convert list into cell path

```
> [5 c 7 h] | into cell-path
```

Convert table into cell path

```
> [[value, optional]; [5 true] [c false]] |
into cell-path
```

Notes

```
Converting a string directly into a cell path is intentionally
not supported.
```

into datetime

version: 0.90.2

usage:

Convert text or timestamp into a datetime.

Signature

```
> into datetime ...rest --timezone --offset --format --list
--list-human
```

Parameters

- `...rest`: For a data structure input, convert data at the given cell paths.
- `--timezone {string}`: Specify timezone if the input is a Unix timestamp. Valid options: 'UTC' ('u') or 'LOCAL' ('l')
- `--offset {int}`: Specify timezone by offset from UTC if the input is a Unix timestamp, like '+8', '-4'
- `--format {string}`: Specify expected format of INPUT string to parse to datetime. Use --list to see options
- `--list`: Show all possible variables for use in --format flag
- `--list-human`: Show human-readable datetime parsing examples

Input/output types:

input	output
int	datetime
list<string>	list<datetime>
record	record
string	datetime
table	table

Examples

Convert any standard timestamp string to datetime

```
> '27.02.2021 1:55 pm +0000' | into datetime
```

Convert any standard timestamp string to datetime

```
> '2021-02-27T13:55:40.2246+00:00' | into datetime
```

Convert non-standard timestamp string to datetime using a custom format

```
> '20210227_135540+0000' | into datetime --format
  '%Y%m%d_%H%M%S%z'
```

Convert nanosecond-precision unix timestamp to a datetime with offset from UTC

```
> 1614434140123456789 | into datetime --offset -
5
```

Convert standard (seconds) unix timestamp to a UTC datetime

```
> 1614434140 * 1_000_000_000 | into datetime
```

Convert list of timestamps to datetimes

```
> ["2023-03-30 10:10:07 -05:00", "2023-05-05 13:
43:49 -05:00", "2023-06-05 01:37:42 -05:00"] |
into datetime
```

Parsing human readable datetimes

```
> 'Today at 18:30' | into datetime
```

Parsing human readable datetimes

```
> 'Last Friday at 19:45' | into datetime
```

Parsing human readable datetimes

```
> 'In 5 minutes and 30 seconds' | into datetime
```

into duration

version: 0.90.2

usage:

Convert value to duration.

Signature

`> into duration ...rest --unit`

Parameters

- `...rest`: For a data structure input, convert data at the given cell paths.
- `--unit {string}`: Unit to convert number into (will have an effect only with integer input)

Input/output types:

input	output
duration	duration
int	duration
string	duration
table	table

Examples

Convert duration string to duration value

```
> '7min' | into duration
```

Convert compound duration string to duration value

```
> '1day 2hr 3min 4sec' | into duration
```

Convert table of duration strings to table of duration values

```
> [[value]; ['1sec'] ['2min'] ['3hr'] [
'4day'] ['5wk']] | into duration value
```

Convert duration to duration

```
> 420sec | into duration
```

Convert a number of ns to duration

```
> 1_234_567 | into duration
```

Convert a number of an arbitrary unit to duration

```
> 1_234 | into duration --unit ms
```

Notes

```
Max duration value is i64::MAX nanoseconds; max duration
time unit is wk (weeks).
```

into filesize

version: 0.90.2

usage:

Convert value to filesize.

Signature

```
> into filesize ...rest
```

Parameters

- `...rest`: For a data structure input, convert data at the given cell paths.

Input/output types:

input	output
filesize	filesize
int	filesize
list<any>	list<filesize>
list<filesize>	list<filesize>
list<int>	list<filesize>
list<number>	list<filesize>
list<string>	list<filesize>
number	filesize
record	record
string	filesize
table	table

Examples

Convert string to filesize in table

```
> [ [ device size ] ; [ " /dev/sda1 "  " 200 " ]  [ " /
dev/loop0 "  " 50 " ] ]  |  into  filesize size
```

Convert string to filesize

```
> ' 2 '  |  into  filesize
```

Convert float to filesize

```
> 8 .3  |  into  filesize
```

Convert int to filesize

```
> 5  |  into  filesize
```

Convert file size to filesize

```
> 4 KB  |  into  filesize
```

into float

version: 0.90.2

usage:

Convert data into floating point number.

Signature

```
> into float ...rest
```

Parameters

- `...rest`: For a data structure input, convert data at the given cell paths.

Input/output types:

input	output
bool	float
float	float
int	float
list<any>	list<float>
record	record
string	float
table	table

Examples

Convert string to float in table

```
> [[num]; ['5.01']] | into float num
```

Convert string to floating point number

```
> '1.345' | into float
```

Coerce list of ints and floats to float

```
> [4 -5.9] | into float
```

Convert boolean to float

```
> true | into float
```

into int

version: 0.90.2

usage:

Convert value to integer.

Signature

```
> into int ...rest --radix --endian
```

Parameters

- `...rest`: For a data structure input, convert data at the given cell paths.
- `--radix {number}`: radix of integer
- `--endian {string}`: byte encode endian, available options: native(default), little, big

Input/output types:

input	output
binary	int
bool	int
datetime	int
duration	int
filesize	int
list<any>	list<int>
list<bool>	list<int>
list<datetime>	list<int>
list<duration>	list<int>
list<filesize>	list<int>
list<number>	list<int>
list<string>	list<int>
number	int
record	record
string	int
table	table

Examples

Convert string to int in table

```
> [[num]; ['-5'] [4] [1.5]] | into int
num
```

Convert string to int

```
> '2' | into int
```

Convert float to int

```
> 5.9 | into int
```

Convert decimal string to int

```
> '5.9' | into int
```

Convert file size to int

```
> 4KB | into int
```

Convert bool to int

```
> [false, true] | into int
```

Convert date to int (Unix nanosecond timestamp)

```
> 1983-04-13T12:09:14.123456789-05:00 | into int
```

Convert to int from binary data (radix: 2)

```
> '1101' | into int --radix 2
```

Convert to int from hex

```
> 'FF' | into int --radix 16
```

Convert octal string to int

```
> '0o10132' | into int
```

Convert 0 padded string to int

```
> '0010132' | into int
```

Convert 0 padded string to int with radix 8

```
> '0010132' | into int --radix 8
```

into record

version: 0.90.2

usage:

Convert value to record.

Signature

```
> into record
```

Input/output types:

input	output
datetime	record
duration	record
list<any>	record
range	record
record	record

Examples

Convert from one row table to record

```
> [[value]; [false]] | into record
```

Convert from list to record

```
> [1 2 3] | into record
```

Convert from range to record

```
> 0..2 | into record
```

convert duration to record (weeks max)

```
> (-500day - 4hr - 5sec) | into record
```

convert record to record

```
> {a: 1, b: 2} | into record
```

convert date to record

```
> 2020-04-12T22:10:57+02:00 | into record
```

into sqlite

version: 0.90.2

usage:

Convert table into a SQLite database.

Signature

```
> into sqlite (file-name) --table-name
```

Parameters

- `file-name`: Specify the filename to save the database to.
- `--table-name {string}`: Specify table name to store the data in

Input/output types:

input	output
record	nothing
table	nothing

Examples

Convert ls entries into a SQLite database with 'main' as the table name

```
> ls | into sqlite my_ls.db
```

Convert ls entries into a SQLite database with 'my_table' as the table name

```
> ls | into sqlite my_ls.db -t my_table
```

Convert table literal into a SQLite database with 'main' as the table name

```
> [[name]; [-----] [someone] [=====] [
somename] ['((((']] | into sqlite filename.db
```

Insert a single record into a SQLite database

```
> { foo: bar, baz: quux } | into sqlite filename.
db
```

into string

version: 0.90.2

usage:

Convert value to string.

Signature

```
> into string ...rest --decimals
```

Parameters

- `...rest`: For a data structure input, convert data at the given cell paths.
- `--decimals {int}`: decimal digits to which to round

Input/output types:

input	output
binary	string
bool	string
datetime	string
duration	string
filesize	string
int	string
list<any>	list<string>
number	string
record	record
string	string
table	table

Examples

convert int to string and append three decimal places

```
> 5 | into string --decimals 3
```

convert float to string and round to nearest integer

```
> 1.7 | into string --decimals 0
```

convert float to string

```
> 1.7 | into string --decimals 1
```

convert float to string and limit to 2 decimals

```
> 1.734 | into string --decimals 2
```

convert float to string

```
> 4.3 | into string
```

convert string to string

```
> '1234' | into string
```

convert boolean to string

```
> true | into string
```

convert date to string

```
> '2020-10-10 10:00:00 +02:00' | into datetime |
  into string
```

convert filepath to string

```
> ls Cargo.toml | get name | into string
```

convert filesize to string

```
> 1 KiB | into string
```

convert duration to string

```
> 9 day | into string
```

into value

version: 0.90.2

usage:

Infer nushell datatype for each cell.

Signature

```
> into value --columns --prefer-filesizes
```

Parameters

- `--columns {table}`: list of columns to update
- `--prefer-filesizes`: For ints display them as human-readable file sizes

Input/output types:

input	output
table	table

Examples

Infer Nushell values for each cell.

```
> $table | into value
```

Infer Nushell values for each cell in the given columns.

```
> $table | into value -c [column1, column5]
```

is-admin

version: 0.90.2

usage:

Check if nushell is running with administrator or root privileges.

Signature

```
> is-admin
```

Input/output types:

input	output
nothing	bool

Examples

Return 'iamroot' if nushell is running with admin/root privileges, and 'iamnotroot' if not.

```
> if (is-admin) { "iamroot" } else { "iamnotroot
" }
```

is-empty

version: 0.90.2

usage:

Check for empty values.

Signature

```
> is-empty ...rest
```

Parameters

- `...rest`: The names of the columns to check emptiness.

Input/output types:

input	output
any	bool

Examples

Check if a string is empty

```
> '' | is-empty
```

Check if a list is empty

```
> [] | is-empty
```

Check if more than one column are empty

```
> [[meal size]; [arepa small] [taco '']] |
is-empty meal size
```

is-terminal

version: 0.90.2

usage:

Check if stdin, stdout, or stderr is a terminal.

Signature

`> is-terminal --stdin --stdout --stderr`

Parameters

- `--stdin`: Check if stdin is a terminal
- `--stdout`: Check if stdout is a terminal
- `--stderr`: Check if stderr is a terminal

Input/output types:

input	output
nothing	bool

Examples

Return “terminal attached” if standard input is attached to a terminal, and “no terminal” if not.

```
> if (is-terminal --stdin) { "terminal attached"
  } else { "no terminal" }
```

items

version: 0.90.2

usage:

Given a record, iterate on each pair of column name and associated value.

Signature

`> items (closure)`

Parameters

- `closure`: The closure to run.

Input/output types:

input	output
record	any

Examples

Iterate over each key-value pair of a record

```
> { new: york , san: francisco } | items { |key ,
  value| echo $' ($key) ($value) ' }
```

Notes

```
This is a the fusion of `columns`, `values` and `each`.
```

join

version: 0.90.2

usage:

Join two tables.

Signature

```
> join (right-table) (left-on) (right-on) --inner --left --
right --outer
```

Parameters

- `right-table`: The right table in the join.
- `left-on`: Name of column in input (left) table to join on.
- `right-on`: Name of column in right table to join on. Defaults to same column as left table.
- `--inner`: Inner join (default)
- `--left`: Left-outer join
- `--right`: Right-outer join
- `--outer`: Outer join

Input/output types:

input	output
table	table

Examples

Join two tables

```
> [{a: 1 b: 2}] | join [{a: 1 c: 3}] a
```

keybindings

version: 0.90.2

usage:

Keybindings related commands.

Signature

`> keybindings`

Input/output types:

input	output
nothing	string

Notes

> `You must use one of the following subcommands. Using this command as-is will only produce this help message. For more information on input and keybindings, check: https://www.nushell.sh/book/line_editor.html`

Subcommands:

name	type	usage
`keybindings default`[207]	Builtin	List default keybindings.
`keybindings list`[208]	Builtin	List available options that can be used to create keybindings.
`keybindings listen`[209]	Builtin	Get input from the user.

keybindings default

version: 0.90.2

usage:

List default keybindings.

Signature

```
> keybindings default
```

Input/output types:

input	output
nothing	table

[207] `/commands/docs/keybindings_default.md`
[208] `/commands/docs/keybindings_list.md`
[209] `/commands/docs/keybindings_listen.md`

Examples

Get list with default keybindings

```
> keybindings default
```

keybindings list

version: 0.90.2

usage:

List available options that can be used to create keybindings.

Signature

`> keybindings list --modifiers --keycodes --modes --events --edits`

Parameters

- `--modifiers`: list of modifiers
- `--keycodes`: list of keycodes
- `--modes`: list of edit modes
- `--events`: list of reedline event
- `--edits`: list of edit commands

Input/output types:

input	output
nothing	table

Examples

Get list of key modifiers

```
> keybindings list  -- modifiers
```

Get list of reedline events and edit commands

```
> keybindings list  -e  -d
```

Get list with all the available options

```
> keybindings list
```

keybindings listen

version: 0.90.2

usage:

Get input from the user.

Signature

```
> keybindings listen
```

Input/output types:

input	output
nothing	nothing

Examples

Type and see key event codes

```
> keybindings listen
```

Notes

```
This is an internal debugging tool. For better output,
try `input listen --types [key]`
```

kill

version: 0.90.2

usage:

Kill a process using the process id.

Signature

```
> kill (pid) ...rest --force --quiet --signal
```

Parameters

- `pid`: Process id of process that is to be killed.
- `...rest`: Rest of processes to kill.
- `--force`: forcefully kill the process
- `--quiet`: won't print anything to the console
- `--signal {int}`: signal decimal number to be sent instead of the default 15 (unsupported on Windows)

Input/output types:

input	output
nothing	any

Examples

Kill the pid using the most memory

```
> ps | sort-by mem | last | kill $ in .pid
```

Force kill a given pid

```
> kill -- force 12345
```

Send INT signal

```
> kill - s 2 12345
```

last

version: 0.90.2

usage:

Return only the last several rows of the input. Counterpart of `first`. Opposite of `drop`.

Signature

```
> last (rows)
```

Parameters

- `rows`: Starting from the back, the number of rows to return.

Input/output types:

input	output
binary	binary
list<any>	any

Examples

Return the last 2 items of a list/table

```
> [1,2,3] | last 2
```

Return the last item of a list/table

```
> [1,2,3] | last
```

Return the last 2 bytes of a binary value

```
> 0x[01 23 45] | last 2
```

lazy make

version: 0.90.2

usage:

Create a lazy record.

Signature

```
> lazy make --columns --get-value
```

Parameters

- `--columns {list<string>}`: Closure that gets called when the LazyRecord needs to list the available column names
- `--get-value {closure(string)}`: Closure to call when a value needs to be produced on demand

Input/output types:

input	output
nothing	record

Examples

Create a lazy record

```
> lazy make --columns ["haskell", "futures", "nushell"] --get-value { |lazything| $lazything + "!" }
```

Test the laziness of lazy records

```
> lazy make --columns ["hello"] --get-value { |key| print $"getting ($key)!"; $key | str upcase }
```

Notes

```
Lazy records are special records that only evaluate their
values once the property is requested.        For example,
when printing a lazy record, all of its fields will be
collected. But when accessing        a specific property,
only it will be evaluated.        Note that this is unrelated
to the lazyframes feature bundled with dataframes.
```

length

version: 0.90.2

usage:

Count the number of items in an input list or rows in a table.

Signature

```
> length
```

Input/output types:

input	output
list<any>	int

Examples

Count the number of items in a list

```
> [1 2 3 4 5] | length
```

Count the number of rows in a table

```
> [{a:1 b:2}, {a:2 b:3}] | length
```

let-env

version: 0.90.2

usage:

`let-env FOO = ...` has been removed, use `$env.FOO = ...` instead.

Signature

```
> let-env (var_name) (initial_value)
```

Parameters

- `var_name`: Variable name.
- `initial_value`: Equals sign followed by value.

Input/output types:

input	output
nothing	nothing

let

version: 0.90.2

usage:

Create a variable and give it a value.

Signature

```
> let (var_name) (initial_value)
```

Parameters

- `var_name`: Variable name.
- `initial_value`: Equals sign followed by value.

Input/output types:

input	output
any	nothing

Examples

Set a variable to a value

```
> let x = 10
```

Set a variable to the result of an expression

```
> let x = 10 + 100
```

Set a variable based on the condition

```
> let x = if false { -1 } else { 1 }
```

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

lines

version: 0.90.2

usage:

Converts input to lines.

Signature

```
> lines --skip-empty
```

Parameters

- `--skip-empty`: skip empty lines

Input/output types:

input	output
any	list<string>

Examples

Split multi-line string into lines

```
> $" two\nlines " | lines
```

load-env

version: 0.90.2

usage:

Loads an environment update from a record.

Signature

```
> load-env (update)
```

Parameters

- `update`: The record to use for updates.

Input/output types:

input	output
nothing	nothing
record	nothing

Examples

Load variables from an input stream

```
> { NAME: ABE , AGE: UNKNOWN } | load-env ; $ env .
NAME
```

Load variables from an argument

```
> load-env { NAME: ABE , AGE: UNKNOWN } ; $ env .NAME
```

loop

version: 0.90.2

usage:

Run a block in a loop.

Signature

`> loop (block)`

Parameters

- `block`: Block to loop.

Input/output types:

input	output
nothing	nothing

Examples

Loop while a condition is true

```
> mut x = 0 ;  loop  {  if  $ x  > 10  {  break  } ;  $ x
  =  $ x  + 1  } ;  $ x
```

ls

version: 0.90.2

usage:

List the filenames, sizes, and modification times of items in a directory.

Signature

```
> ls (pattern) --all --long --short-names --full-paths --du
--directory --mime-type
```

Parameters

- `pattern`: The glob pattern to use.
- `--all`: Show hidden files
- `--long`: Get all available columns for each entry (slower; columns are platform-dependent)
- `--short-names`: Only print the file names, and not the path
- `--full-paths`: display paths as absolute paths
- `--du`: Display the apparent directory size ("disk usage") in place of the directory metadata size
- `--directory`: List the specified directory itself instead of its contents
- `--mime-type`: Show mime-type in type column instead of 'file' (based on filenames only; files' contents are not examined)

Input/output types:

input	output
nothing	table

Examples

List visible files in the current directory

```
> ls
```

List visible files in a subdirectory

```
> ls subdir
```

List visible files with full path in the parent directory

```
> ls -f ..
```

List Rust files

```
> ls *.rs
```

List files and directories whose name do not contain ‘bar’

```
> ls -s | where name !~ bar
```

List all dirs in your home directory

```
> ls -a ~ | where type == dir
```

List all dirs in your home directory which have not been modified in 7 days

```
> ls -as ~ | where type == dir and modified < (
(date now) - 7day)
```

List given paths and show directories themselves

```
> ['/path/to/directory' '/path/to/file'] | each
  {|| ls -D $in } | flatten
```

match

version: 0.90.2

usage:

Conditionally run a block on a matched value.

Signature

`> match (value) (match_block)`

Parameters

- `value`: Value to check.
- `match_block`: Block to run if check succeeds.

Input/output types:

input	output
any	any

Examples

Match on a value in range

```
> match 3 { 1..10 => 'yes!' }
```

Match on a field in a record

```
> match {a: 100} { {a: $my_value} => { $my_
value } }
```

Match with a catch-all

```
> match 3 { 1 => { 'yes!' }, _ => { 'no!' 
} }
```

Match against a list

```
> match [1, 2, 3] { [$a, $b, $c] => { $a
 + $b + $c }, _ => 0 }
```

Match against pipeline input

```
> {a: {b: 3}} | match $in {{a: { $b }}
=> ($b + 10) }
```

Match with a guard

```
> match [ 1 2 3 ] {         [ $ x , .. $ y ] if $ x
== 1 => { ' good list ' } ,         _ => { ' not a
very good list ' }    }
```

math

version: 0.90.2

usage:

Use mathematical functions as aggregate functions on a list of numbers or tables.

Signature

```
> math
```

Input/output types:

input	output
nothing	string

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

Subcommands:

name	type	usage
`math abs`[210]	Builtin	Returns the absolute value of a number.
`math arccos`[211]	Builtin	Returns the arccosine of the number.
`math arccosh`[212]	Builtin	Returns the inverse of the hyperbolic cosine function.
`math arcsin`[213]	Builtin	Returns the arcsine of the number.
`math arcsinh`[214]	Builtin	Returns the inverse of the hyperbolic sine function.
`math arctan`[215]	Builtin	Returns the arctangent of the number.
`math arctanh`[216]	Builtin	Returns the inverse of the hyperbolic tangent function.
`math avg`[217]	Builtin	Returns the average of a list of numbers.
`math ceil`[218]	Builtin	Returns the ceil of a number (smallest integer greater than or equal to that number).
`math cos`[219]	Builtin	Returns the cosine of the number.
`math cosh`[220]	Builtin	Returns the hyperbolic cosine of the number.
`math exp`[221]	Builtin	Returns e raised to the power of x.
`math floor`[222]	Builtin	Returns the floor of a number (largest integer less than or equal to that number).
`math ln`[223]	Builtin	Returns the natural logarithm. Base: (math e).
`math log`[224]	Builtin	Returns the logarithm for an arbitrary base.
`math max`[225]	Builtin	Returns the maxi-

math abs

version: 0.90.2

usage:

Returns the absolute value of a number.

Signature

```
> math abs
```

Input/output types:

input	output
duration	duration
list<duration>	list<duration>
list<number>	list<number>
number	number

[211] /commands/docs/math_arccos.md
[212] /commands/docs/math_arccosh.md
[213] /commands/docs/math_arcsin.md
[214] /commands/docs/math_arcsinh.md
[215] /commands/docs/math_arctan.md
[216] /commands/docs/math_arctanh.md
[217] /commands/docs/math_avg.md
[218] /commands/docs/math_ceil.md
[219] /commands/docs/math_cos.md
[220] /commands/docs/math_cosh.md
[221] /commands/docs/math_exp.md
[222] /commands/docs/math_floor.md
[223] /commands/docs/math_ln.md
[224] /commands/docs/math_log.md
[225] /commands/docs/math_max.md
[226] /commands/docs/math_median.md
[227] /commands/docs/math_min.md
[228] /commands/docs/math_mode.md
[229] /commands/docs/math_product.md
[230] /commands/docs/math_round.md
[231] /commands/docs/math_sin.md
[232] /commands/docs/math_sinh.md
[233] /commands/docs/math_sqrt.md
[234] /commands/docs/math_stddev.md
[235] /commands/docs/math_sum.md
[236] /commands/docs/math_tan.md
[237] /commands/docs/math_tanh.md
[238] /commands/docs/math_variance.md

Examples

Compute absolute value of each number in a list of numbers

```
> [-50 -100.0 25] | math abs
```

math arccos

version: 0.90.2

usage:

Returns the arccosine of the number.

Signature

```
> math arccos --degrees
```

Parameters

- `--degrees`: Return degrees instead of radians

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Get the arccosine of 1

```
> 1 | math arccos
```

Get the arccosine of -1 in degrees

```
> -1 | math arccos --degrees
```

math arccosh

version: 0.90.2

usage:

Returns the inverse of the hyperbolic cosine function.

Signature

```
> math arccosh
```

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Get the arccosh of 1

```
> 1 | math arccosh
```

math arcsin

version: 0.90.2

usage:

Returns the arcsine of the number.

Signature

```
> math arcsin --degrees
```

Parameters

- `--degrees`: Return degrees instead of radians

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Get the arcsine of 1

```
> 1 | math arcsin
```

Get the arcsine of 1 in degrees

```
> 1 | math arcsin --degrees
```

math arcsinh

version: 0.90.2

usage:

Returns the inverse of the hyperbolic sine function.

Signature

`> math arcsinh`

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Get the arcsinh of 0

```
> 0 | math arcsinh
```

math arctan

version: 0.90.2

usage:

Returns the arctangent of the number.

Signature

```
> math arctan --degrees
```

Parameters

- `--degrees`: Return degrees instead of radians

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Get the arctangent of 1

```
> 1 | math arctan
```

Get the arctangent of -1 in degrees

```
> - 1 | math arctan -- degrees
```

math arctanh

version: 0.90.2

usage:

Returns the inverse of the hyperbolic tangent function.

Signature

```
> math arctanh
```

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Get the arctanh of 1

```
> 1 | math arctanh
```

math avg

version: 0.90.2

usage:

Returns the average of a list of numbers.

Signature

```
> math avg
```

Input/output types:

input	output
duration	duration
filesize	filesize
list<duration>	duration
list<filesize>	filesize
list<number>	number
number	number
range	number
record	record
table	record

Examples

Compute the average of a list of numbers

```
> [-50 100.0 25] | math avg
```

Compute the average of a list of durations

```
> [2sec 1min] | math avg
```

Compute the average of each column in a table

```
> [[a b]; [1 2] [3 4]] | math avg
```

math ceil

version: 0.90.2

usage:

Returns the ceil of a number (smallest integer greater than or equal to that number).

Signature

```
> math ceil
```

Input/output types:

input	output
list<number>	list<int>
number	int

Examples

Apply the ceil function to a list of numbers

```
> [1.5 2.3 -3.1] | math ceil
```

math cos

version: 0.90.2

usage:

Returns the cosine of the number.

Signature

```
> math cos --degrees
```

Parameters

- `--degrees`: Use degrees instead of radians

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Apply the cosine to

```
> 3.141592 | math cos | math round --precision
4
```

Apply the cosine to a list of angles in degrees

```
> [0 90 180 270 360] | math cos --degrees
```

math cosh

version: 0.90.2

usage:

Returns the hyperbolic cosine of the number.

Signature

```
> math cosh
```

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Apply the hyperbolic cosine to 1

```
> 1 | math cosh
```

math exp

version: 0.90.2

usage:

Returns e raised to the power of x.

Signature

`> math exp`

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Get e raised to the power of zero

```
> 0 | math exp
```

Get e (same as 'math e')

```
> 1 | math exp
```

math floor

version: 0.90.2

usage:

Returns the floor of a number (largest integer less than or equal to that number).

Signature

```
> math floor
```

Input/output types:

input	output
list<number>	list<int>
number	int

Examples

Apply the floor function to a list of numbers

```
> [1.5 2.3 -3.1] | math floor
```

math ln

version: 0.90.2

usage:

Returns the natural logarithm. Base: (math e).

Signature

```
> math ln
```

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Get the natural logarithm of e

```
> 2.7182818 | math ln | math round --precision
 4
```

math log

version: 0.90.2

usage:

Returns the logarithm for an arbitrary base.

Signature

```
> math log (base)
```

Parameters

- `base`: Base for which the logarithm should be computed.

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Get the logarithm of 100 to the base 10

```
> 100 | math log 10
```

Get the log2 of a list of values

```
> [16 8 4] | math log 2
```

math max

version: 0.90.2

usage:

Returns the maximum of a list of values, or of columns in a table.

Signature

```
> math max
```

Input/output types:

input	output
list<any>	any
list<duration>	duration
list<filesize>	filesize
list<number>	number
range	number
record	record
table	record

Examples

Find the maximum of a list of numbers

```
> [-50 100 25] | math max
```

Find the maxima of the columns of a table

```
> [{a: 1 b: 3} {a: 2 b: -1}] | math max
```

Find the maximum of a list of dates

```
> [2022-02-02 2022-12-30 2012-12-12] | math
max
```

math median

version: 0.90.2

usage:

Computes the median of a list of numbers.

Signature

```
> math median
```

Input/output types:

input	output
list<duration>	duration
list<filesize>	filesize
list<number>	number
range	number
record	record
table	record

Examples

Compute the median of a list of numbers

```
> [3 8 9 12 12 15] | math median
```

Compute the medians of the columns of a table

```
> [{a: 1 b: 3} {a: 2 b: -1} {a: -3 b: 5}] |
math median
```

Find the median of a list of file sizes

```
> [5KB 10MB 200B] | math median
```

math min

version: 0.90.2

usage:

Finds the minimum within a list of values or tables.

Signature

```
> math min
```

Input/output types:

input	output
list<any>	any
list<duration>	duration
list<filesize>	filesize
list<number>	number
range	number
record	record
table	record

Examples

Compute the minimum of a list of numbers

```
> [-50 100 25] | math min
```

Compute the minima of the columns of a table

```
> [{a: 1 b: 3} {a: 2 b: -1}] | math min
```

Find the minimum of a list of arbitrary values (Warning: Weird)

```
> [-50 'hello' true] | math min
```

math mode

version: 0.90.2

usage:

Returns the most frequent element(s) from a list of numbers or tables.

Signature

```
> math mode
```

Input/output types:

input	output
list<duration>	list<duration>
list<filesize>	list<filesize>
list<number>	list<number>
table	record

Examples

Compute the mode(s) of a list of numbers

```
> [3 3 9 12 12 15] | math mode
```

Compute the mode(s) of the columns of a table

```
> [{a: 1 b: 3} {a: 2 b: -1} {a: 1 b: 5}] |
math mode
```

math product

version: 0.90.2

usage:

Returns the product of a list of numbers or the products of each column of a table.

Signature

`> math product`

Input/output types:

input	output
list<number>	number
range	number
record	record
table	record

Examples

Compute the product of a list of numbers

```
> [2 3 3 4] | math product
```

Compute the product of each column in a table

```
> [[a b]; [1 2] [3 4]] | math product
```

math round

version: 0.90.2

usage:

Returns the input number rounded to the specified precision.

Signature

```
> math round --precision
```

Parameters

- `--precision {number}`: digits of precision

Input/output types:

input	output
list<number>	list<number>
number	number

Examples

Apply the round function to a list of numbers

```
> [1.5 2.3 -3.1] | math round
```

Apply the round function with precision specified

```
> [1.555 2.333 -3.111] | math round --precision
 2
```

Apply negative precision to a list of numbers

```
> [123, 123.3, -123.4] | math round --precision
 -1
```

math sin

version: 0.90.2

usage:

Returns the sine of the number.

Signature

```
> math sin --degrees
```

Parameters

- `--degrees`: Use degrees instead of radians

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Apply the sine to /2

```
> 3 .141592 / 2 | math sin | math round -- pre-
cision 4
```

Apply the sine to a list of angles in degrees

```
> [ 0 90 180 270 360 ] | math sin - d | math
round -- precision 4
```

math sinh

version: 0.90.2

usage:

Returns the hyperbolic sine of the number.

Signature

```
> math sinh
```

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Apply the hyperbolic sine to 1

```
> 1 | math sinh
```

math sqrt

version: 0.90.2

usage:

Returns the square root of the input number.

Signature

```
> math sqrt
```

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Compute the square root of each number in a list

```
> [9 16] | math sqrt
```

math stddev

version: 0.90.2

usage:

Returns the standard deviation of a list of numbers, or of each column in a table.

Signature

```
> math stddev --sample
```

Parameters

- `--sample`: calculate sample standard deviation (i.e. using N-1 as the denominator)

Input/output types:

input	output
list<number>	number
record	record
table	record

Examples

Compute the standard deviation of a list of numbers

```
> [1 2 3 4 5] | math stddev
```

Compute the sample standard deviation of a list of numbers

```
> [1 2 3 4 5] | math stddev --sample
```

Compute the standard deviation of each column in a table

```
> [[a b]; [1 2] [3 4]] | math stddev
```

math sum

version: 0.90.2

usage:

Returns the sum of a list of numbers or of each column in a table.

Signature

```
> math sum
```

Input/output types:

input	output
list<duration>	duration
list<filesize>	filesize
list<number>	number
range	number
record	record
table	record

Examples

Sum a list of numbers

```
> [1 2 3] | math sum
```

Get the disk usage for the current directory

```
> ls | get size | math sum
```

Compute the sum of each column in a table

```
> [[a b]; [1 2] [3 4]] | math sum
```

math tan

version: 0.90.2

usage:

Returns the tangent of the number.

Signature

```
> math tan --degrees
```

Parameters

- `--degrees`: Use degrees instead of radians

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Apply the tangent to /4

```
> 3.141592 / 4 | math tan | math round --pre-
cision 4
```

Apply the tangent to a list of angles in degrees

```
> [-45 0 45] | math tan --degrees
```

math tanh

version: 0.90.2

usage:

Returns the hyperbolic tangent of the number.

Signature

`> math tanh`

Input/output types:

input	output
list<number>	list<float>
number	float

Examples

Apply the hyperbolic tangent to 10*

```
> 3.141592 * 10 | math tanh | math round --
precision 4
```

math variance

version: 0.90.2

usage:

Returns the variance of a list of numbers or of each column in a table.

Signature

```
> math variance --sample
```

Parameters

- `--sample`: calculate sample variance (i.e. using N-1 as the denominator)

Input/output types:

input	output
list<number>	number
record	record
table	record

Examples

Get the variance of a list of numbers

```
> [1 2 3 4 5] | math variance
```

Get the sample variance of a list of numbers

```
> [1 2 3 4 5] | math variance --sample
```

Compute the variance of each column in a table

```
> [[a b]; [1 2] [3 4]] | math variance
```

merge

version: 0.90.2

usage:

Merge the input with a record or table, overwriting values in matching columns.

Signature

```
> merge (value)
```

Parameters

- `value`: The new value to merge with.

Input/output types:

input	output
record	record
table	table

Examples

Add an 'index' column to the input table

```
> [a b c] | wrap name | merge ( [1 2 3] |
wrap index )
```

Merge two records

```
> {a: 1, b: 2} | merge {c: 3}
```

Merge two tables, overwriting overlapping columns

```
> [{columnA: A0 columnB: B0}] | merge [{columnA:
 'A0*'}]
```

Notes

```
You may provide a column structure to merge When merging
tables, row 0 of the input table is overwritten with values
from row 0 of the provided table, then repeating this process
with row 1, and so on.
```

metadata

version: 0.90.2

usage:

Get the metadata for items in the stream.

Signature

`> metadata (expression)`

Parameters

- `expression`: The expression you want metadata for.

Input/output types:

input	output
any	record

Examples

Get the metadata of a variable

```
> let a = 42 ; metadata $a
```

Get the metadata of the input

```
> ls | metadata
```

mkdir

version: 0.90.2

usage:

Make directories, creates intermediary directories as required.

Signature

`> mkdir ...rest --verbose`

Parameters

- `...rest`: The name(s) of the path(s) to create.
- `--verbose`: print created path(s).

Input/output types:

input	output
nothing	nothing

Examples

Make a directory named foo

```
> mkdir foo
```

Make multiple directories and show the paths created

```
> mkdir -v foo/bar foo2
```

mktemp

version: 0.90.2

usage:

Create temporary files or directories using uutils/coreutils mktemp.

Signature

```
> mktemp (template) --suffix --tmpdir-path --tmpdir --directory
```

Parameters

- `template`: Optional pattern from which the name of the file or directory is derived. Must contain at least three 'X's in last component.
- `--suffix {string}`: Append suffix to template; must not contain a slash.
- `--tmpdir-path {path}`: Interpret TEMPLATE relative to tmpdir-path. If tmpdir-path is not set use $TMPDIR

- `--tmpdir`: Interpret TEMPLATE relative to the system temporary directory.
- `--directory`: Create a directory instead of a file.

Input/output types:

input	output
nothing	string

Examples

Make a temporary file with the given suffix in the current working directory.

```
> mktemp --suffix .txt
```

Make a temporary file named testfile.XXX with the 'X's as random characters in the current working directory.

```
> mktemp testfile.XXX
```

Make a temporary file with a template in the system temp directory.

```
> mktemp -t testfile.XXX
```

Make a temporary directory with randomly generated name in the temporary directory.

```
> mktemp -d
```

module

version: 0.90.2

usage:

Define a custom module.

Signature

```
> module (module) (block)
```

Parameters

- `module`: Module name or module path.
- `block`: Body of the module if 'module' parameter is not a module path.

Input/output types:

input	output
nothing	nothing

Examples

Define a custom command in a module and call it

```
> module spam { export def foo [] { "foo" } };
use spam foo; foo
```

Define an environment variable in a module

```
> module foo { export-env { $env.FOO = "BAZ" }
 }; use foo; $env.FOO
```

Define a custom command that participates in the environment in a module and call it

```
> module foo { export def --env bar [] { $env.
FOO_BAR = "BAZ" } }; use foo bar; bar; $env.FOO_
BAR
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

move

version: 0.90.2

usage:

Move columns before or after other columns.

Signature

```
> move ...rest --after --before
```

Parameters

- `...rest`: The columns to move.
- `--after {string}`: the column that will precede the columns moved
- `--before {string}`: the column that will be the next after the columns moved

Input/output types:

input	output
record	record
table	table

Examples

Move a column before the first column

```
> [[name value index]; [foo a 1] [bar b 2] [baz c 3]] | move index --before name
```

Move multiple columns after the last column and reorder them

```
> [[name value index]; [foo a 1] [bar b 2] [baz c 3]] | move value name --after index
```

Move columns of a record

```
> { name: foo, value: a, index: 1 } | move name --before index
```

mut

version: 0.90.2

usage:

Create a mutable variable and give it a value.

Signature

```
> mut (var_name) (initial_value)
```

Parameters

- `var_name`: Variable name.
- `initial_value`: Equals sign followed by value.

Input/output types:

input	output
any	nothing

Examples

Set a mutable variable to a value, then update it

```
> mut x = 10 ;  $x  = 12
```

Upsert a value inside a mutable data structure

```
> mut a = {b:{c:1}} ;  $a.b.c  = 2
```

Set a mutable variable to the result of an expression

```
> mut x = 10 + 100
```

Set a mutable variable based on the condition

```
> mut x = if false { -1 } else { 1 }
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

mv

version: 0.90.2

usage:

Move files or directories.

Signature

```
> mv (source) (destination) --verbose --force --interactive
--update
```

Parameters

- `source`: The location to move files/directories from.
- `destination`: The location to move files/directories to.
- `--verbose`: make mv to be verbose, showing files been moved.
- `--force`: overwrite the destination.
- `--interactive`: ask user to confirm action
- `--update`: move only when the SOURCE file is newer than the destination file(with -f) or when the destination file is missing

Input/output types:

input	output
nothing	nothing

Examples

Rename a file

```
> mv before.txt after.txt
```

Move a file into a directory

```
> mv test.txt my/subdirectory
```

Move many files into a directory

```
> mv *.txt my/subdirectory
```

nu-check

version: 0.90.2

usage:

Validate and parse input content.

Signature

```
> nu-check (path) --as-module --debug --all
```

Parameters

- `path`: File path to parse.
- `--as-module`: Parse content as module
- `--debug`: Show error messages
- `--all`: Parse content as script first, returns result if success, otherwise, try with module

Input/output types:

input	output
list<any>	bool
string	bool

Examples

Parse a input file as script(Default)

```
> nu-check script.nu
```

Parse a input file as module

```
> nu-check -- as-module  module.nu
```

Parse a input file by showing error message

```
> nu-check  -- debug  script.nu
```

Parse an external stream as script by showing error message

```
> open foo.nu  |  nu-check  -- debug  script.nu
```

Parse an internal stream as module by showing error message

```
> open module.nu  |  lines  |  nu-check  -- debug  --
as-module  module.nu
```

Parse a string as script

```
> $' two(char nl)lines ' | nu-check
```

Heuristically parse which begins with script first, if it sees a failure, try module afterwards

```
> nu-check -a script.nu
```

Heuristically parse by showing error message

```
> open foo.nu | lines | nu-check --all --debug
```

nu-highlight

version: 0.90.2

usage:

Syntax highlight the input string.

Signature

`> nu-highlight`

Input/output types:

input	output
string	string

Examples

Describe the type of a string

```
> 'let x = 3' | nu-highlight
```

open

version: 0.90.2

usage:

Load a file into a cell, converting to table if possible (avoid by appending '--raw').

Signature

```
> open (filename) ...rest --raw
```

Parameters

- `filename`: The filename to use.
- `...rest`: Optional additional files to open.
- `--raw`: open file as raw binary

Input/output types:

input	output
nothing	any
string	any

Examples

Open a file, with structure (based on file extension or SQLite database header)

```
> open myfile.json
```

Open a file, as raw bytes

```
> open myfile.json -- raw
```

Open a file, using the input to get filename

```
> ' myfile.txt ' | open
```

Open a file, and decode it by the specified encoding

```
> open myfile.txt -- raw | decode utf-8
```

Create a custom `from` parser to open newline-delimited JSON files with `open`

```
> def " from ndjson " [ ] { from json -o } ; open
myfile.ndjson
```

Notes

```
Support to automatically parse files with an extension
`.xyz` can be provided by a `from xyz` command in scope.
```

overlay

version: 0.90.2

usage:

Commands for manipulating overlays.

Signature

```
> overlay
```

Input/output types:

input	output
nothing	string

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html   You
must use one of the following subcommands. Using this command
as-is will only produce this help message.
```

Subcommands:

name	type	usage
`overlay hide`[239]	Builtin	Hide an active overlay.
`overlay list`[240]	Builtin	List all active overlays.
`overlay new`[241]	Builtin	Create an empty overlay.
`overlay use`[242]	Builtin	Use definitions from a module as an overlay.

overlay hide

version: 0.90.2

usage:

Hide an active overlay.

Signature

```
> overlay hide (name) --keep-custom --keep-env
```

Parameters

- `name`: Overlay to hide.
- `--keep-custom`: Keep all newly added commands and aliases in the next activated overlay.
- `--keep-env {list<string>}`: List of environment variables to keep in the next activated overlay

Input/output types:

input	output
nothing	nothing

[239] /commands/docs/overlay_hide.md
[240] /commands/docs/overlay_list.md
[241] /commands/docs/overlay_new.md
[242] /commands/docs/overlay_use.md

Examples

Keep a custom command after hiding the overlay

```
> module spam { export def foo [] { "foo" } }
   overlay use spam    def bar [] { "bar" }
overlay hide spam --keep-custom    bar
```

Hide an overlay created from a file

```
>  'export alias f = "foo"'  |  save spam.nu
overlay use spam.nu     overlay hide spam
```

Hide the last activated overlay

```
> module spam { export-env { $env.FOO = "foo" }
  }     overlay use spam     overlay hide
```

Keep the current working directory when removing an overlay

```
> overlay new spam     cd some-dir     overlay hide
--keep-env [ PWD ] spam
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

overlay list

version: 0.90.2

usage:

List all active overlays.

Signature

```
> overlay list
```

Input/output types:

input	output
nothing	list<string>

Examples

Get the last activated overlay

```
> module spam { export def foo [] { "foo" } }
  overlay use spam    overlay list | last
```

Notes

```
The overlays are listed in the order they were activated.
```

overlay new

version: 0.90.2

usage:

Create an empty overlay.

Signature

`> overlay new (name)`

Parameters

- `name`: Name of the overlay.

Input/output types:

input	output
nothing	nothing

Examples

Create an empty overlay

```
> overlay new spam
```

Notes

> ```
> The command will first create an empty module, then add
> it as an overlay. This command is a parser keyword. For
> details, check:  https://www.nushell.sh/book/thinking_-
> in_nu.html
> ```

overlay use

version: 0.90.2

usage:

Use definitions from a module as an overlay.

Signature

```
> overlay use (name) (as) --prefix --reload
```

Parameters

- `name`: Module name to use overlay for.
- `as`: `as` keyword followed by a new name.
- `--prefix`: Prepend module name to the imported commands and aliases
- `--reload`: If the overlay already exists, reload its definitions and environment.

Input/output types:

input	output
nothing	nothing

Examples

Create an overlay from a module

```
> module spam { export def foo [] { "foo" } }
  overlay use spam    foo
```

Create an overlay from a module and rename it

```
> module spam { export def foo [] { "foo" } }
  overlay use spam as spam_new    foo
```

Create an overlay with a prefix

```
> 'export def foo { "foo" }'    overlay use --pre-
fix spam     spam foo
```

Create an overlay from a file

```
> 'export-env { $env.FOO = "foo" }'  |  save spam.
nu     overlay use spam.nu     $env.FOO
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

par-each

version: 0.90.2

usage:

Run a closure on each row of the input list in parallel, creating a new list with the results.

Signature

`> par-each (closure) --threads --keep-order`

Parameters

- `closure`: The closure to run.
- `--threads {int}`: the number of threads to use
- `--keep-order`: keep sequence of output same as the order of input

Input/output types:

input	output
any	any
list<any>	list<any>
table	list<any>

Examples

Multiplies each number. Note that the list will become arbitrarily disordered.

```
> [1 2 3] | par-each {|e| $e * 2 }
```

Multiplies each number, keeping an original order

```
> [1 2 3] | par-each --keep-order {|e| $e *
 2 }
```

Enumerate and sort-by can be used to reconstruct the original order

```
> 1..3 | enumerate | par-each {|p| update item
($p.item * 2)} | sort-by item | get item
```

Output can still be sorted afterward

```
> [foo bar baz] | par-each {|e| $e + '!' }
  | sort
```

Iterate over each element, producing a list showing indexes of any 2s

```
> [1 2 3] | enumerate | par-each { |e| if $e
.item == 2 { $"found 2 at ($e.index)!"} }
```

parse

version: 0.90.2

usage:

Parse columns from string data using a simple pattern.

Signature

```
> parse (pattern) --regex
```

Parameters

- `pattern`: The pattern to match.
- `--regex`: use full regex syntax for patterns

Input/output types:

input	output
list<any>	table
string	table

Examples

Parse a string into two named columns

```
> "hi there" | parse "{foo} {bar}"
```

Parse a string using regex pattern

```
> "hi there" | parse --regex '(?P<foo>\w+) (
?P<bar>\w+)'
```

Parse a string using fancy-regex named capture group pattern

```
> "foo bar." | parse --regex '\s*(?<name>\w+)(
?=\.)'
```

Parse a string using fancy-regex capture group pattern

```
> "foo! bar." | parse --regex '(\w+)(?=\.)|(
\w+)(?=!)'
```

Parse a string using fancy-regex look behind pattern

```
> " @another(foo bar)   " | parse --regex '\s*(
?<=[() ])(@\w+)(\([^)]*\))?\s*'
```

Parse a string using fancy-regex look ahead atomic group pattern

```
> "abcd" | parse --regex '^a(bc(?=d)|b)cd$'
```

path

version: 0.90.2

usage:

Explore and manipulate paths.

Signature

```
> path
```

Input/output types:

input	output
nothing	string

Notes

> You must use one of the following subcommands. Using this command as-is will only produce this help message. There are three ways to represent a path: * As a path literal, e.g., '/home/viking/spam.txt' * As a structured path: a table with 'parent', 'stem', and 'extension' (and * 'prefix' on Windows) columns. This format is produced by the 'path parse' subcommand. * As a list of path parts, e.g., '[/ home viking spam.txt]'. Splitting into parts is done by the `path split` command. All subcommands accept all three variants as an input. Furthermore, the 'path join' subcommand can be used to join the structured path or path parts back into the path literal.

Subcommands:

name	type	usage
path basename[243]	Builtin	Get the final component of a path.
path dirname[244]	Builtin	Get the parent directory of a path.
path exists[245]	Builtin	Check whether a path exists.
path expand[246]	Builtin	Try to expand a path to its absolute form.
path join[247]	Builtin	Join a structured path or a list of path parts.
path parse[248]	Builtin	Convert a path into structured data.
path relative-to[249]	Builtin	Express a path as relative to another path.
path split[250]	Builtin	Split a path into a list based on the system's path separator.
path type[251]	Builtin	Get the type of the object a path refers to (e.g., file, dir, symlink).

path basename

version: 0.90.2

[243] /commands/docs/path_basename.md
[244] /commands/docs/path_dirname.md
[245] /commands/docs/path_exists.md
[246] /commands/docs/path_expand.md
[247] /commands/docs/path_join.md
[248] /commands/docs/path_parse.md
[249] /commands/docs/path_relative-to.md
[250] /commands/docs/path_split.md
[251] /commands/docs/path_type.md

usage:

Get the final component of a path.

Signature

```
> path basename --replace
```

Parameters

- `--replace {string}`: Return original path with basename replaced by this string

Input/output types:

input	output
list<string>	list<string>
string	string

Examples

Get basename of a path

```
> '/home/joe/test.txt' | path basename
```

Get basename of a list of paths

```
> [ /home/joe, /home/doe ] | path basename
```

Replace basename of a path

```
> '/home/joe/test.txt' | path basename --replace
  'spam.png'
```

path dirname

version: 0.90.2

usage:

Get the parent directory of a path.

Signature

```
> path dirname --replace --num-levels
```

Parameters

- `--replace {string}`: Return original path with dirname replaced by this string
- `--num-levels {int}`: Number of directories to walk up

Input/output types:

input	output
list<string>	list<string>
string	string

Examples

Get dirname of a path

```
> '/home/joe/code/test.txt' | path dirname
```

Get dirname of a list of paths

```
> [ /home/joe/test.txt, /home/doe/test.txt ] | path
dirname
```

Walk up two levels

```
> '/home/joe/code/test.txt' | path dirname --num-
levels 2
```

Replace the part that would be returned with a custom path

```
> '/home/joe/code/test.txt' | path dirname --num-
levels 2 --replace /home/viking
```

path exists

version: 0.90.2

usage:

Check whether a path exists.

Signature

```
> path exists --no-symlink
```

Parameters

- `--no-symlink`: Do not resolve symbolic links

Input/output types:

input	output
list<string>	list<bool>
string	bool

Examples

Check if a file exists

```
> '/home/joe/todo.txt' | path exists
```

Check if files in list exist

```
> [ /home/joe/todo.txt, /home/doe/todo.txt ] | path
exists
```

Notes

```
This only checks if it is possible to either `open` or
`cd` to the given path. If you need to distinguish dirs
and files, please use `path type`.
```

path expand

version: 0.90.2

usage:

Try to expand a path to its absolute form.

Signature

```
> path expand --strict --no-symlink
```

Parameters

- `--strict`: Throw an error if the path could not be expanded
- `--no-symlink`: Do not resolve symbolic links

Input/output types:

input	output
list<string>	list<string>
string	string

Examples

Expand an absolute path

```
> '/home/joe/foo/../bar' | path expand
```

Expand a relative path

```
> 'foo/../bar' | path expand
```

Expand a list of paths

```
> [ /foo/../bar, /foo/../baz ] | path expand
```

path join

version: 0.90.2

usage:

Join a structured path or a list of path parts.

Signature

```
> path join ...rest
```

Parameters

- `...rest`: Path to append to the input.

Input/output types:

input	output
list<string>	string
record	string
string	string
table	list<string>

Examples

Append a filename to a path

```
> '/home/viking' | path join spam.txt
```

Append a filename to a path

```
> '/home/viking' | path join spams this_spam.txt
```

Use relative paths, e.g. '..' will go up one directory

```
> '/home/viking' | path join .. folder
```

Use absolute paths, e.g. '/' will bring you to the top level directory

```
> '/home/viking' | path join / folder
```

Join a list of parts into a path

```
> [ '/' 'home' 'viking' 'spam.txt' ] |
path join
```

Join a structured path into a path

```
> { parent: '/home/viking', stem: 'spam',
extension: 'txt' } | path join
```

Join a table of structured paths into a list of paths

```
> [[ parent stem extension ]; [ '/home/viking'
'spam' 'txt' ]] | path join
```

Notes

```
Optionally, append an additional path to the result. It
is designed to accept the output of 'path parse' and 'path
split' subcommands.
```

path parse

version: 0.90.2

usage:

Convert a path into structured data.

Signature

```
> path parse --extension
```

Parameters

- `--extension {string}`: Manually supply the extension (without the dot)

Input/output types:

input	output
list<string>	table
string	record

Examples

Parse a path

```
> '/home/viking/spam.txt' | path parse
```

Replace a complex extension

```
> '/home/viking/spam.tar.gz' | path parse --extension tar.gz | upsert extension { 'txt' }
```

Ignore the extension

```
> '/etc/conf.d' | path parse --extension ''
```

Parse all paths in a list

```
> [ /home/viking.d /home/spam.txt ] | path parse
```

Notes

```
Each path is split into a table with 'parent', 'stem' and
'extension' fields. On Windows, an extra 'prefix' column
is added.
```

path relative-to

version: 0.90.2

usage:

Express a path as relative to another path.

Signature

```
> path relative-to (path)
```

Parameters

- `path`: Parent shared with the input path.

Input/output types:

input	output
list<string>	list<string>
string	string

Examples

Find a relative path from two absolute paths

```
> '/home/viking' | path relative-to '/home'
```

Find a relative path from absolute paths in list

```
> [ /home/viking, /home/spam ] | path relative-to
'/home'
```

Find a relative path from two relative paths

```
> 'eggs/bacon/sausage/spam' | path relative-to '
eggs/bacon/sausage'
```

Notes

```
Can be used only when the input and the argument paths
are either both absolute or both relative. The argument
path needs to be a parent of the input path.
```

path split

version: 0.90.2

usage:

Split a path into a list based on the system's path separator.

Signature

```
> path split
```

Input/output types:

input	output
list<string>	list<list<string>>
string	list<string>

Examples

Split a path into parts

```
> '/home/viking/spam.txt' | path split
```

Split paths in list into parts

```
> [ /home/viking/spam.txt /home/viking/eggs.txt ] |
  path split
```

path type

version: 0.90.2

usage:

Get the type of the object a path refers to (e.g., file, dir, symlink).

Signature

```
> path type
```

Input/output types:

input	output
list<string>	list<string>
string	string

Examples

Show type of a filepath

```
> '.' | path type
```

Show type of a filepaths in a list

```
> ls | get name | path type
```

Notes

```
This checks the file system to confirm the path's object
type. If nothing is found, an empty string will be returned.
```

port

version: 0.90.2

usage:

Get a free port from system.

Signature

```
> port (start) (end)
```

Parameters

- `start`: The start port to scan (inclusive).
- `end`: The end port to scan (inclusive).

Input/output types:

input	output
nothing	int

Examples

get a free port between 3121 and 4000

```
> port 3121 4000
```

get a free port from system

```
> port
```

prepend

version: 0.90.2

usage:

Prepend any number of rows to a table.

Signature

`> prepend (row)`

Parameters

- `row`: The row, list, or table to prepend.

Input/output types:

input	output
any	list<any>

Examples

prepend a list to an item

```
> 0 | prepend [1 2 3]
```

Prepend a list of strings to a string

```
> "a" | prepend ["b"]
```

Prepend one int item

```
> [1 2 3 4] | prepend 0
```

Prepend two int items

```
> [2 3 4] | prepend [0 1]
```

Prepend ints and strings

```
> [2 nu 4 shell] | prepend [0 1 rocks]
```

Prepend a range

```
> [3 4] | prepend 0..2
```

Notes

```
Be aware that this command 'unwraps' lists passed to it.
 So, if you pass a variable to it, and you want the variable's
contents to be prepended without being unwrapped, it's
wise to pre-emptively wrap the variable in a list, like
so: `prepend [$val]`. This way, `prepend` will only unwrap
the outer list, and leave the variable's contents untouched.
```

print

version: 0.90.2

usage:

Print the given values to stdout.

Signature

```
> print ...rest --no-newline --stderr
```

Parameters

- `...rest`: the values to print
- `--no-newline`: print without inserting a newline for the line ending
- `--stderr`: print to stderr instead of stdout

Input/output types:

input	output
any	nothing
nothing	nothing

Examples

Print 'hello world'

```
> print "hello world"
```

Print the sum of 2 and 3

```
> print (2 + 3)
```

Notes

```
Unlike `echo`, this command does not return any value (
`print | describe` will return "nothing"). Since this
command has no output, there is no point in piping it with
other commands. `print` may be used inside blocks of code
(e.g.: hooks) to display text during execution without
interfering with the pipeline.
```

ps

version: 0.90.2

usage:

View information about system processes.

Signature

```
> ps --long
```

Parameters

- `--long`: list all available columns for each entry

Input/output types:

input	output
nothing	table

Examples

List the system processes

```
> ps
```

List the top 5 system processes with the highest memory usage

```
> ps | sort-by mem | last 5
```

List the top 3 system processes with the highest CPU usage

```
> ps | sort-by cpu | last 3
```

List the system processes with 'nu' in their names

```
> ps | where name =~ 'nu'
```

Get the parent process id of the current nu process

```
> ps | where pid == $nu.pid | get ppid
```

query

version: 0.90.2

usage:

Show all the query commands

Signature

`> query`

Input/output types:

input	output
any	any

Subcommands:

| name | type | usage | | -- | -------------- | --- | ------------------------- | | `query db`[252] | Builtin | Query a database using SQL. | | `query json`[253] | Builtin,Plugin | execute json query on json file (open --raw <file> | query json 'query string') | | `query web`[254] | Builtin,Plugin | execute selector query on html/web | | `query xml`[255] | Builtin,Plugin | execute xpath query on xml |

query db

version: 0.90.2

usage:

Query a database using SQL.

Signature

`> query db (SQL)`

Parameters

- `SQL`: SQL to execute against the database.

[252] /commands/docs/query_db.md
[253] /commands/docs/query_json.md
[254] /commands/docs/query_web.md
[255] /commands/docs/query_xml.md

Input/output types:

input	output
any	any

Examples

Execute SQL against a SQLite database

```
> open foo.db | query db "SELECT * FROM Bar"
```

query json

version: 0.90.2

usage:

execute json query on json file (open --raw <file> | query json 'query string')

Signature

`> query json (query)`

Parameters

- `query`: json query

Input/output types:

input	output
any	any

query web

version: 0.90.2

usage:

execute selector query on html/web

Signature

```
> query web --query --as-html --attribute --as-table --inspect
```

Parameters

- `--query {string}`: selector query
- `--as-html`: return the query output as html
- `--attribute {string}`: downselect based on the given attribute
- `--as-table {list<string>}`: find table based on column header list
- `--inspect`: run in inspect mode to provide more information for determining column headers

Input/output types:

input	output
any	any

Examples

Retrieve all `<header>` elements from phoronix.com website

```
> http get https://phoronix.com | query web --query
  'header' | flatten
```

Retrieve a html table from Wikipedia and parse it into a nushell table using table headers as guides

```
> http get https://en.wikipedia.org/wiki/List_of_cities_-
in_India_by_population | query web --as-table
[City 'Population(2011)[3]' 'Population(2001)[3][a]
' 'State or unionterritory' 'Ref']
```

Pass multiple css selectors to extract several elements within single query, group the query results together and rotate them to create a table

```
> http get https://www.nushell.sh | query web --
query 'h2, h2 + p' | each {str join} | group
```

```
 2  |  each  { rotate --ccw tagline description }  |
flatten
```

Retrieve a specific html attribute instead of the default text

```
> http get https://example.org  |  query  web  -- query
 a  -- attribute  href
```

query xml

version: 0.90.2

usage:

execute xpath query on xml

Signature

`> query xml (query)`

Parameters

- `query`: xpath query

Input/output types:

input	output
any	any

random

version: 0.90.2

usage:

Generate a random value.

Signature

`> random`

Input/output types:

input	output
nothing	string

Notes

> ```
> You must use one of the following subcommands. Using this
> command as-is will only produce this help message.
> ```

Subcommands:

name	type	usage
`random bool`[256]	Builtin	Generate a random boolean value.
`random chars`[257]	Builtin	Generate random chars.
`random dice`[258]	Builtin	Generate a random dice roll.
`random float`[259]	Builtin	Generate a random float within a range [min..max].
`random int`[260]	Builtin	Generate a random integer [min..max].
`random uuid`[261]	Builtin	Generate a random uuid4 string.

random bool

version: 0.90.2

usage:

Generate a random boolean value.

[256] `/commands/docs/random_bool.md`
[257] `/commands/docs/random_chars.md`
[258] `/commands/docs/random_dice.md`
[259] `/commands/docs/random_float.md`
[260] `/commands/docs/random_int.md`
[261] `/commands/docs/random_uuid.md`

Signature

`> random bool --bias`

Parameters

- `--bias {number}`: Adjusts the probability of a “true” outcome

Input/output types:

input	output
nothing	bool

Examples

Generate a random boolean value

```
> random bool
```

Generate a random boolean value with a 75% chance of “true”

```
> random bool -- bias 0.75
```

random chars

version: 0.90.2

usage:

Generate random chars.

Signature

`> random chars --length`

Parameters

- `--length {int}`: Number of chars

Input/output types:

input	output
nothing	string

Examples

Generate random chars

```
> random chars
```

Generate random chars with specified length

```
> random chars -- length 20
```

random dice

version: 0.90.2

usage:

Generate a random dice roll.

Signature

```
> random dice --dice --sides
```

Parameters

- `--dice {int}`: The amount of dice being rolled
- `--sides {int}`: The amount of sides a die has

Input/output types:

input	output
nothing	list<any>

Examples

Roll 1 dice with 6 sides each

```
> random dice
```

Roll 10 dice with 12 sides each

```
> random dice -- dice 10 -- sides 12
```

random float

version: 0.90.2

usage:

Generate a random float within a range [min..max].

Signature

```
> random float (range)
```

Parameters

- `range`: Range of values.

Input/output types:

input	output
nothing	float

Examples

Generate a default float value between 0 and 1

```
> random float
```

Generate a random float less than or equal to 500

```
> random float ..500
```

Generate a random float greater than or equal to 100000

```
> random float 100000..
```

Generate a random float between 1.0 and 1.1

```
> random float 1.0..1.1
```

random int

version: 0.90.2

usage:

Generate a random integer [min..max].

Signature

```
> random int (range)
```

Parameters

- `range`: Range of values.

Input/output types:

input	output
nothing	int

Examples

Generate an unconstrained random integer

```
> random int
```

Generate a random integer less than or equal to 500

```
> random int ..500
```

Generate a random integer greater than or equal to 100000

```
> random int 100000..
```

Generate a random integer between 1 and 10

```
> random int 1..10
```

random uuid

version: 0.90.2

usage:

Generate a random uuid4 string.

Signature

`> random uuid`

Input/output types:

input	output
nothing	string

Examples

Generate a random uuid4 string

```
> random uuid
```

range

version: 0.90.2

usage:

Return only the selected rows.

Signature

`> range (rows)`

Parameters

- `rows`: Range of rows to return.

Input/output types:

input	output
list<any>	list<any>

Examples

Get the last 2 items

```
> [0,1,2,3,4,5] | range 4..5
```

Get the last 2 items

```
> [0,1,2,3,4,5] | range (-2)..
```

Get the next to last 2 items

```
> [0,1,2,3,4,5] | range (-3)..-2
```

reduce

version: 0.90.2

usage:

Aggregate a list to a single value using an accumulator closure.

Signature

`> reduce (closure) --fold`

Parameters

- `closure`: Reducing function.
- `--fold {any}`: reduce with initial value

Input/output types:

input	output
list<any>	any
range	any
table	any

Examples

Sum values of a list (same as 'math sum')

```
> [ 1 2 3 4 ] | reduce {|it, acc| $it + $acc
  }
```

Sum values of a list, plus their indexes

```
> [ 8 7 6 ] | enumerate | reduce --fold 0 {
|it, acc| $acc + $it.item + $it.index }
```

Sum values with a starting value (fold)

```
> [ 1 2 3 4 ] | reduce --fold 10 { |it , acc| $
acc + $ it }
```

Replace selected characters in a string with ‘X’

```
> [ i o t ] | reduce --fold " Arthur, King of
the Britons " { |it , acc| $ acc | str replace --all
$ it " X " }
```

Add ascending numbers to each of the filenames, and join with semi-colons.

```
> [ ' foo.gz ' , ' bar.gz ' , ' baz.gz ' ] | enumerate
  | reduce --fold ' ' { |str all| $" ( $ all )(if $
str .index != 0 {'; '})( $ str .index + 1)-( $ str .item)
" }
```

Concatenate a string with itself, using a range to determine the number of times.

```
> let s = " Str " ; 0..2 | reduce --fold ' ' {
|it , acc| $ acc + $ s }
```

register

version: 0.90.2

usage:

Register a plugin.

Signature

`> register (plugin) (signature) --shell`

Parameters

- `plugin`: Path of executable for plugin.
- `signature`: Block with signature description as json object.
- `--shell {path}`: path of shell used to run plugin (cmd, sh, python, etc)

Input/output types:

input	output
nothing	nothing

Examples

Register `nu_plugin_query` plugin from ~/.cargo/bin/ dir

```
> register ~/.cargo/bin/nu_plugin_query
```

Register `nu_plugin_query` plugin from `nu -c` (writes/updates $nu.plugin-path)

```
> let plugin = ((which nu).path.0 | path dirname
  | path join 'nu_plugin_query'); nu -c $'
register ($plugin); version'
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

reject

version: 0.90.2

usage:

Remove the given columns or rows from the table. Opposite of `select`.

Signature

`> reject ...rest --ignore-errors`

Parameters

- `...rest`: The names of columns to remove from the table.
- `--ignore-errors`: ignore missing data (make all cell path members optional)

Input/output types:

input	output
record	record
table	table

Examples

Reject a column in the `ls` table

```
> ls | reject modified
```

Reject a column in a table

```
> [[a, b]; [1, 2]] | reject a
```

Reject a row in a table

```
> [[a, b]; [1, 2] [3, 4]] | reject 1
```

Reject the specified field in a record

```
> {a: 1, b: 2} | reject a
```

Reject a nested field in a record

```
> {a: {b: 3, c: 5}} | reject a.b
```

Reject columns by a provided list of columns

```
> let cols = [size type]; [[name type size]; [Cargo.
toml toml 1kb] [Cargo.lock toml 2kb]] | reject $
cols
```

Reject columns by a list of columns directly

```
> [[name type size]; [Cargo.toml toml 1kb] [
Cargo.lock toml 2kb]] | reject ["size", "type
"]
```

Reject rows by a provided list of rows

```
> let rows = [0 2]; [[name type size]; [Cargo.toml
 toml 1kb] [Cargo.lock toml 2kb] [file.json json 3kb
]] | reject $rows
```

Notes

```
To remove a quantity of rows or columns, use `skip`, `drop`,
or `drop column`.
```

rename

version: 0.90.2

usage:

Creates a new table with columns renamed.

Signature

```
> rename ...rest --column --block
```

Parameters

- `...rest`: The new names for the columns.
- `--column {record}`: column name to be changed
- `--block {closure(any)}`: A closure to apply changes on each column

Input/output types:

input	output
record	record
table	table

Examples

Rename a column

```
> [[a, b]; [1, 2]] | rename my_column
```

Rename many columns

```
> [[a, b, c]; [1, 2, 3]] | rename eggs ham
bacon
```

Rename a specific column

```
> [[a, b, c]; [1, 2, 3]] | rename --column
{ a: ham }
```

Rename the fields of a record

```
> {a: 1 b: 2} | rename x y
```

Rename fields based on a given closure

```
> {abc: 1, bbc: 2} | rename --block { str
replace --all 'b' 'z'}
```

return

version: 0.90.2

usage:

Return early from a function.

Signature

`> return (return_value)`

Parameters

- `return_value`: Optional value to return.

Input/output types:

input	output
nothing	any

Examples

Return early

```
> def foo [] { return }
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

reverse

version: 0.90.2

usage:

Reverses the input list or table.

Signature

```
> reverse
```

Input/output types:

input	output
list<any>	list<any>

Examples

Reverse a list

```
> [0,1,2,3] | reverse
```

Reverse a table

```
> [{a: 1} {a: 2}] | reverse
```

rm

version: 0.90.2

usage:

Remove files and directories.

Signature

```
> rm (filename) ...rest --trash --permanent --recursive --force
--verbose --interactive --interactive-once
```

Parameters

- `filename`: The file or files you want to remove.
- `...rest`: Additional file path(s) to remove.
- `--trash`: move to the platform's trash instead of permanently deleting. not used on android and ios
- `--permanent`: delete permanently, ignoring the 'always_trash' config option. always enabled on android and ios
- `--recursive`: delete subdirectories recursively
- `--force`: suppress error when no file
- `--verbose`: print names of deleted files
- `--interactive`: ask user to confirm action
- `--interactive-once`: ask user to confirm action only once

Input/output types:

input	output
nothing	nothing

Examples

Delete, or move a file to the trash (based on the 'always_trash' config option)

```
> rm file.txt
```

Move a file to the trash

```
> rm --trash file.txt
```

Delete a file permanently, even if the 'always_trash' config option is true

```
> rm --permanent file.txt
```

Delete a file, ignoring 'file not found' errors

```
> rm --force file.txt
```

Delete all 0KB files in the current directory

```
> ls | where size == 0KB and type == file | each
  { rm $in.name } | null
```

roll

version: 0.90.2

usage:

Rolling commands for tables.

Signature

```
> roll
```

Input/output types:

input	output
nothing	string

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

Subcommands:

name	type	usage
roll down[262]	Builtin	Roll table rows down.
roll left[263]	Builtin	Roll record or table columns left.
roll right[264]	Builtin	Roll table columns right.
roll up[265]	Builtin	Roll table rows up.

roll down

version: 0.90.2

usage:

Roll table rows down.

Signature

```
> roll down --by
```

Parameters

- `--by {int}`: Number of rows to roll

Input/output types:

input	output
table	table

Examples

Rolls rows down of a table

```
> [[a b]; [1 2] [3 4] [5 6]] | roll down
```

[262] /commands/docs/roll_down.md
[263] /commands/docs/roll_left.md
[264] /commands/docs/roll_right.md
[265] /commands/docs/roll_up.md

roll left

version: 0.90.2

usage:

Roll record or table columns left.

Signature

```
> roll left --by --cells-only
```

Parameters

- `--by {int}`: Number of columns to roll
- `--cells-only`: rotates columns leaving headers fixed

Input/output types:

input	output
record	record
table	table

Examples

Rolls columns of a record to the left

```
> {a:1 b:2 c:3} | roll left
```

Rolls columns of a table to the left

```
> [[a b c]; [1 2 3] [4 5 6]] | roll left
```

Rolls columns to the left without changing column names

```
> [[a b c]; [1 2 3] [4 5 6]] | roll left -
- cells-only
```

roll right

version: 0.90.2

usage:

Roll table columns right.

Signature

```
> roll right --by --cells-only
```

Parameters

- `--by {int}`: Number of columns to roll
- `--cells-only`: rotates columns leaving headers fixed

Input/output types:

input	output
record	record
table	table

Examples

Rolls columns of a record to the right

```
> {a:1 b:2 c:3} | roll right
```

Rolls columns to the right

```
> [[a b c]; [1 2 3] [4 5 6]] | roll right
```

Rolls columns to the right with fixed headers

```
> [[a b c]; [1 2 3] [4 5 6]] | roll right
--cells-only
```

roll up

version: 0.90.2

usage:

Roll table rows up.

Signature

```
> roll up --by
```

Parameters

- `--by {int}`: Number of rows to roll

Input/output types:

input	output
table	table

Examples

Rolls rows up

```
> [[a b]; [1 2] [3 4] [5 6]] | roll up
```

rotate

version: 0.90.2

usage:

Rotates a table or record clockwise (default) or counter-clockwise (use --ccw flag).

Signature

```
> rotate ...rest --ccw
```

Parameters

- `...rest`: the names to give columns once rotated
- `--ccw`: rotate counter clockwise

Input/output types:

input	output
record	table
table	table

Examples

Rotate a record clockwise, producing a table (like `transpose` but with column order reversed)

```
> {a:1, b:2} | rotate
```

Rotate 2x3 table clockwise

```
> [[a b]; [1 2] [3 4] [5 6]] | rotate
```

Rotate table clockwise and change columns names

```
> [[a b]; [1 2]] | rotate col_a col_b
```

Rotate table counter clockwise

```
> [[a b]; [1 2]] | rotate --ccw
```

Rotate table counter-clockwise

```
> [[a b]; [1 2] [3 4] [5 6]] | rotate --
ccw
```

Rotate table counter-clockwise and change columns names

```
> [[a b]; [1 2]] | rotate --ccw col_a col_b
```

run-external

version: 0.90.2

usage:

Runs external command.

Signature

```
> run-external (command) ...rest --redirect-stdout --redirect-
stderr --redirect-combine --trim-end-newline
```

Parameters

- `command`: External command to run.
- `...rest`: Arguments for external command.
- `--redirect-stdout`: redirect stdout to the pipeline
- `--redirect-stderr`: redirect stderr to the pipeline
- `--redirect-combine`: redirect both stdout and stderr combined to the pipeline (collected in stdout)
- `--trim-end-newline`: trimming end newlines

Input/output types:

input	output
any	any

Examples

Run an external command

```
> run-external "echo" "-n" "hello"
```

Redirect stdout from an external command into the pipeline

```
> run-external --redirect-stdout "echo" "-n" "
hello" | split chars
```

save

version: 0.90.2

usage:

Save a file.

Signature

```
> save (filename) --stderr --raw --append --force --progress
```

Parameters

- `filename`: The filename to use.
- `--stderr {path}`: the filename used to save stderr, only works with `-r` flag
- `--raw`: save file as raw binary
- `--append`: append input to the end of the file
- `--force`: overwrite the destination
- `--progress`: enable progress bar

Input/output types:

input	output
any	nothing

Examples

Save a string to foo.txt in the current directory

```
> 'save me' | save foo.txt
```

Append a string to the end of foo.txt

```
> 'append me' | save --append foo.txt
```

Save a record to foo.json in the current directory

```
> { a: 1, b: 2 } | save foo.json
```

Save a running program's stderr to foo.txt

```
> do -i {} | save foo.txt --stderr foo.txt
```

Save a running program's stderr to separate file

```
> do -i {} | save foo.txt --stderr bar.txt
```

schema

version: 0.90.2

usage:

Show the schema of a SQLite database.

Signature

```
> schema
```

Input/output types:

input	output
any	any

Examples

Show the schema of a SQLite database

```
> open foo.db | schema
```

scope

version: 0.90.2

usage:

Commands for getting info about what is in scope.

Signature

```
> scope
```

Input/output types:

input	output
nothing	string

Subcommands:

name	type	usage
`scope aliases`[266]	Builtin	Output info on the aliases in the current scope.
`scope commands`[267]	Builtin	Output info on the commands in the current scope.
`scope engine-stats`[268]	Builtin	Output stats on the engine in the current state.
`scope externs`[269]	Builtin	Output info on the known externals in the current scope.
`scope modules`[270]	Builtin	Output info on the modules in the current scope.
`scope variables`[271]	Builtin	Output info on the variables in the current scope.

scope aliases

version: 0.90.2

usage:

Output info on the aliases in the current scope.

Signature

```
> scope aliases
```

[266] /commands/docs/scope_aliases.md
[267] /commands/docs/scope_commands.md
[268] /commands/docs/scope_engine-stats.md
[269] /commands/docs/scope_externs.md
[270] /commands/docs/scope_modules.md
[271] /commands/docs/scope_variables.md

Input/output types:

input	output
nothing	any

Examples

Show the aliases in the current scope

```
> scope aliases
```

scope commands

version: 0.90.2

usage:

Output info on the commands in the current scope.

Signature

```
> scope commands
```

Input/output types:

input	output
nothing	any

Examples

Show the commands in the current scope

```
> scope commands
```

scope engine-stats

version: 0.90.2

usage:

Output stats on the engine in the current state.

Signature

```
> scope engine-stats
```

Input/output types:

input	output
nothing	any

Examples

Show the stats on the current engine state

```
> scope engine-stats
```

scope externs

version: 0.90.2

usage:

Output info on the known externals in the current scope.

Signature

```
> scope externs
```

Input/output types:

input	output
nothing	any

Examples

Show the known externals in the current scope

```
> scope externs
```

scope modules

version: 0.90.2

usage:

Output info on the modules in the current scope.

Signature

```
> scope modules
```

Input/output types:

input	output
nothing	any

Examples

Show the modules in the current scope

```
> scope modules
```

scope variables

version: 0.90.2

usage:

Output info on the variables in the current scope.

Signature

```
> scope variables
```

Input/output types:

input	output
nothing	any

Examples

Show the variables in the current scope

```
> scope variables
```

select

version: 0.90.2

usage:

Select only these columns or rows from the input. Opposite of `reject`.

Signature

```
> select ...rest --ignore-errors
```

Parameters

- `...rest`: The columns to select from the table.
- `--ignore-errors`: ignore missing data (make all cell path members optional)

Input/output types:

input	output
list<any>	any
record	record
table	table

Examples

Select a column in a table

```
> [{a: a b: b}] | select a
```

Select a field in a record

```
> {a: a b: b} | select a
```

Select just the `name` column

```
> ls | select name
```

Select the first four rows (this is the same as `first 4`)

```
> ls | select 0 1 2 3
```

Select columns by a provided list of columns

```
> let cols = [name type]; [[name type size]; [Cargo.
toml toml 1kb] [Cargo.lock toml 2kb]] | select $
cols
```

Select columns by a provided list of columns

```
> [[name type size]; [Cargo.toml toml 1kb] [
Cargo.lock toml 2kb]] | select ["name", "type
"]
```

Select rows by a provided list of rows

```
> let rows = [0 2]; [[name type size]; [Cargo.toml
 toml 1kb] [Cargo.lock toml 2kb] [file.json json 3kb
]] | select $rows
```

Notes

```
This differs from `get` in that, rather than accessing
the given value in the data structure, it removes all non-
selected values from the structure. Hence, using `select`
on a table will produce a table, a list will produce a
list, and a record will produce a record.
```

seq

version: 0.90.2

usage:

Output sequences of numbers.

Signature

```
> seq ...rest
```

Parameters

- `...rest`: Sequence values.

Input/output types:

input	output
nothing	list<number>

Examples

sequence 1 to 10

```
> seq 1 10
```

sequence 1.0 to 2.0 by 0.1s

```
> seq 1.0 0.1 2.0
```

sequence 1 to 5, then convert to a string with a pipe separator

```
> seq 1 5 | str join '|'
```

Subcommands:

name	type	usage
`seq char`[272]	Builtin	Print a sequence of ASCII characters.
`seq date`[273]	Builtin	Print sequences of dates.

seq char

version: 0.90.2

usage:

Print a sequence of ASCII characters.

Signature

```
> seq char (start) (end)
```

[272] `/commands/docs/seq_char.md`
[273] `/commands/docs/seq_date.md`

Parameters

- `start`: Start of character sequence (inclusive).
- `end`: End of character sequence (inclusive).

Input/output types:

input	output
nothing	list<string>

Examples

sequence a to e

```
> seq char a e
```

sequence a to e, and put the characters in a pipe-separated string

```
> seq char a e | str join '|'
```

seq date

version: 0.90.2

usage:

Print sequences of dates.

Signature

`> seq date --output-format --input-format --begin-date --end-date --increment --days --reverse`

Parameters

- `--output-format {string}`: prints dates in this format (defaults to %Y-%m-%d)
- `--input-format {string}`: give argument dates in this format (defaults to %Y-%m-%d)
- `--begin-date {string}`: beginning date range

- `--end-date {string}`: ending date
- `--increment {int}`: increment dates by this number
- `--days {int}`: number of days to print
- `--reverse`: print dates in reverse

Input/output types:

input	output
nothing	list<string>

Examples

print the next 10 days in YYYY-MM-DD format with newline separator

```
> seq date --days 10
```

print the previous 10 days in YYYY-MM-DD format with newline separator

```
> seq date --days 10 --reverse
```

print the previous 10 days starting today in MM/DD/YYYY format with newline separator

```
> seq date --days 10 -o '%m/%d/%Y' --reverse
```

print the first 10 days in January, 2020

```
> seq date --begin-date '2020-01-01' --end-date
'2020-01-10'
```

print every fifth day between January 1st 2020 and January 31st 2020

```
> seq date --begin-date '2020-01-01' --end-date
'2020-01-31' --increment 5
```

shuffle

version: 0.90.2

usage:

Shuffle rows randomly.

Signature

```
> shuffle
```

Input/output types:

input	output
list<any>	list<any>

Examples

Shuffle rows randomly (execute it several times and see the difference)

```
> [[version patch]; ['1.0.0' false] ['3.0.1'
  true] ['2.0.0' false]] | shuffle
```

skip

version: 0.90.2

usage:

Skip the first several rows of the input. Counterpart of `drop`. Opposite of `first`.

Signature

```
> skip (n)
```

Parameters

- `n`: The number of elements to skip.

Input/output types:

input	output
list<any>	list<any>
table	table

Examples

Skip the first value of a list

```
> [2 4 6 8] | skip 1
```

Skip two rows of a table

```
> [[editions]; [2015] [2018] [2021]] |
skip 2
```

Notes

```
To skip specific numbered rows, try `drop nth`. To skip
specific named columns, try `reject`.
```

Subcommands:

name	type	usage
`skip until`[274]	Builtin	Skip elements of the input until a predicate is true.
`skip while`[275]	Builtin	Skip elements of the input while a predicate is true.

skip until

version: 0.90.2

usage:

Skip elements of the input until a predicate is true.

Signature

```
> skip until (predicate)
```

Parameters

- `predicate`: The predicate that skipped element must not match.

[274] `/commands/docs/skip_until.md`
[275] `/commands/docs/skip_while.md`

Input/output types:

input	output
list<any>	list<any>
table	table

Examples

Skip until the element is positive

```
> [-2 0 2 -1] | skip until {|x| $x > 0 }
```

Skip until the element is positive using stored condition

```
> let cond = {|x| $x > 0 }; [-2 0 2 -1] |
skip until $cond
```

Skip until the field value is positive

```
> [{a: -2} {a: 0} {a: 2} {a: -1}] | skip
  until {|x| $x.a > 0 }
```

skip while

version: 0.90.2

usage:

Skip elements of the input while a predicate is true.

Signature

`> skip while (predicate)`

Parameters

- `predicate`: The predicate that skipped element must match.

Input/output types:

input	output
list<any>	list<any>
table	table

Examples

Skip while the element is negative

```
> [-2 0 2 -1] | skip while {|x| $x < 0 }
```

Skip while the element is negative using stored condition

```
> let cond = {|x| $x < 0 }; [-2 0 2 -1] |
skip while $cond
```

Skip while the field value is negative

```
> [{a: -2} {a: 0} {a: 2} {a: -1}] | skip
 while {|x| $x.a < 0 }
```

sleep

version: 0.90.2

usage:

Delay for a specified amount of time.

Signature

`> sleep (duration) ...rest`

Parameters

- `duration`: Time to sleep.
- `...rest`: Additional time.

Input/output types:

input	output
nothing	nothing

Examples

Sleep for 1sec

```
> sleep 1sec
```

Sleep for 3sec

```
> sleep 1sec 1sec 1sec
```

Send output after 1sec

```
> sleep 1sec ;  echo  done
```

sort-by

version: 0.90.2

usage:

Sort by the given columns, in increasing order.

Signature

```
> sort-by ...rest --reverse --ignore-case --natural
```

Parameters

- `...rest`: The column(s) to sort by.
- `--reverse`: Sort in reverse order
- `--ignore-case`: Sort string-based columns case-insensitively
- `--natural`: Sort alphanumeric string-based columns naturally (1, 9, 10, 99, 100, ...)

Input/output types:

input	output
list<any>	list<any>
record	table
table	table

Examples

Sort files by modified date

```
> ls | sort-by modified
```

Sort files by name (case-insensitive)

```
> ls | sort-by name --ignore-case
```

Sort a table by a column (reversed order)

```
> [[fruit count]; [apple 9] [pear 3] [orange
7]] | sort-by fruit --reverse
```

sort

version: 0.90.2

usage:

Sort in increasing order.

Signature

`> sort --reverse --ignore-case --values --natural`

Parameters

- `--reverse`: Sort in reverse order
- `--ignore-case`: Sort string-based data case-insensitively
- `--values`: If input is a single record, sort the record by values; ignored if input is not a single record
- `--natural`: Sort alphanumeric string-based values naturally (1, 9, 10, 99, 100, ...)

Input/output types:

input	output
list<any>	list<any>
record	record

Examples

sort the list by increasing value

```
> [2 0 1] | sort
```

sort the list by decreasing value

```
> [2 0 1] | sort --reverse
```

sort a list of strings

```
> [betty amy sarah] | sort
```

sort a list of strings in reverse

```
> [betty amy sarah] | sort --reverse
```

Sort strings (case-insensitive)

```
> [airplane Truck Car] | sort -i
```

Sort strings (reversed case-insensitive)

```
> [airplane Truck Car] | sort -i -r
```

Sort record by key (case-insensitive)

```
> {b: 3, a: 4} | sort
```

Sort record by value

```
> {b: 4, a: 3, c:1} | sort -v
```

source-env

version: 0.90.2

usage:

Source the environment from a source file into the current environment.

Signature

```
> source-env (filename)
```

Parameters

- `filename`: The filepath to the script file to source the environment from.

Input/output types:

input	output
any	any

Examples

Sources the environment from foo.nu in the current context

```
> source-env foo.nu
```

source

version: 0.90.2

usage:

Runs a script file in the current context.

Signature

```
> source (filename)
```

Parameters

- `filename`: The filepath to the script file to source.

Input/output types:

input	output
any	any

Examples

Runs foo.nu in the current context

```
> source foo.nu
```

Runs foo.nu in current context and call the command defined, suppose foo.nu has content: `def say-hi [] { echo 'Hi!' }`

```
> source ./foo.nu ;  say-hi
```

Notes

```
This command is a parser keyword. For details, check:
 https://www.nushell.sh/book/thinking_in_nu.html
```

split-by

version: 0.90.2

usage:

Split a record into groups.

Signature

```
> split-by (splitter)
```

Parameters

- `splitter`: The splitter value to use.

Input/output types:

input	output
record	record

Examples

split items by column named "lang"

```
> {     ' 2019 ' : [         { name: ' andres ' ,
lang: ' rb ' , year: ' 2019 '  } ,         { name: '
jt ' , lang: ' rs ' , year: ' 2019 '  }     ] ,      '
2021 ' :  [          {  name:  ' storm ' ,  lang:  ' rs ' ,
' year ' :  ' 2021 '   }      ]      }  |  split-by  lang
```

split

version: 0.90.2

usage:

Split contents across desired subcommand (like row, column) via the separator.

Signature

```
> split
```

Input/output types:

input	output
nothing	string

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

Subcommands:

name	type	usage
`split chars`[276]	Builtin	Split a string into a list of characters.
`split column`[277]	Builtin	Split a string into multiple columns using a separator.
`split list`[278]	Builtin	Split a list into multiple lists using a separator.
`split row`[279]	Builtin	Split a string into multiple rows using a separator.
`split words`[280]	Builtin	Split a string's words into separate rows.

split chars

version: 0.90.2

usage:

Split a string into a list of characters.

Signature

```
> split chars --grapheme-clusters --code-points
```

Parameters

- `--grapheme-clusters`: split on grapheme clusters
- `--code-points`: split on code points (default; splits combined characters)

[276] /commands/docs/split_chars.md
[277] /commands/docs/split_column.md
[278] /commands/docs/split_list.md
[279] /commands/docs/split_row.md
[280] /commands/docs/split_words.md

Input/output types:

input	output
list<string>	list<list<string>>
string	list<string>

Examples

Split the string into a list of characters

```
> 'hello' | split chars
```

Split on grapheme clusters

```
> '   ' | split chars --grapheme-clusters
```

Split multiple strings into lists of characters

```
> ['hello', 'world'] | split chars
```

split column

version: 0.90.2

usage:

Split a string into multiple columns using a separator.

Signature

```
> split column (separator) ...rest --collapse-empty --regex
```

Parameters

- `separator`: The character or string that denotes what separates columns.
- `...rest`: Column names to give the new columns.
- `--collapse-empty`: remove empty columns
- `--regex`: separator is a regular expression

Input/output types:

input	output
list<string>	table
string	table

Examples

Split a string into columns by the specified separator

```
> 'a--b--c' | split column '--'
```

Split a string into columns of char and remove the empty columns

```
> 'abc' | split column --collapse-empty ''
```

Split a list of strings into a table

```
> ['a-b' 'c-d'] | split column -
```

Split a list of strings into a table, ignoring padding

```
> ['a - b' 'c - d'] | split column --
regex '\s*-\s*'
```

split list

version: 0.90.2

usage:

Split a list into multiple lists using a separator.

Signature

```
> split list (separator) --regex
```

Parameters

- `separator`: The value that denotes what separates the list.
- `--regex`: separator is a regular expression, matching values that can be coerced into a string

Input/output types:

input	output
list<any>	list<list<any>>

Examples

Split a list of chars into two lists

```
> [a, b, c, d, e, f, g] | split list d
```

Split a list of lists into two lists of lists

```
> [[1,2], [2,3], [3,4]] | split list [2,3
]
```

Split a list of chars into two lists

```
> [a, b, c, d, a, e, f, g] | split list a
```

Split a list of chars into lists based on multiple characters

```
> [a, b, c, d, a, e, f, g] | split list --regex
  '(b|e)'
```

split row

version: 0.90.2

usage:

Split a string into multiple rows using a separator.

Signature

```
> split row (separator) --number --regex
```

Parameters

- `separator`: A character or regex that denotes what separates rows.
- `--number {int}`: Split into maximum number of items
- `--regex`: use regex syntax for separator

Input/output types:

input	output
list<string>	list<string>
string	list<string>

Examples

Split a string into rows of char

```
> 'abc' | split row ''
```

Split a string into rows by the specified separator

```
> 'a--b--c' | split row '--'
```

Split a string by '-'

```
> '-a-b-c-' | split row '-'
```

Split a string by regex

```
> 'a   b       c' | split row -r '\s+'
```

split words

version: 0.90.2

usage:

Split a string's words into separate rows.

Signature

`> split words --min-word-length --grapheme-clusters --utf-8-bytes`

Parameters

- `--min-word-length {int}`: The minimum word length
- `--grapheme-clusters`: measure word length in grapheme clusters (requires -l)
- `--utf-8-bytes`: measure word length in UTF-8 bytes (default; requires -l; non-ASCII chars are length 2+)

Input/output types:

input	output
list<string>	list<list<string>>
string	list<string>

Examples

Split the string's words into separate rows

```
> 'hello world' | split words
```

Split the string's words, of at least 3 characters, into separate rows

```
> 'hello to the world' | split words --min-word-
length 3
```

A real-world example of splitting words

```
> http get https://www.gutenberg.org/files/11/11-0.txt
 | str downcase | split words --min-word-length
 2 | uniq --count | sort-by count --reverse |
  first 10
```

start

version: 0.90.2

usage:

Open a folder, file or website in the default application or viewer.

Signature

```
> start (path)
```

Parameters

- `path`: Path to open.

Input/output types:

input	output
nothing	any
string	any

Examples

Open a text file with the default text editor

```
> start file.txt
```

Open an image with the default image viewer

```
> start file.jpg
```

Open the current directory with the default file manager

```
> start .
```

Open a pdf with the default pdf viewer

```
> start file.pdf
```

Open a website with default browser

```
> start https://www.nushell.sh
```

stor

version: 0.90.2

usage:

Various commands for working with the in-memory sqlite database.

Signature

```
> stor
```

Input/output types:

input	output
nothing	string

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

Subcommands:

name	type	usage
`stor create`[281]	Builtin	Create a table in the in-memory sqlite database.
`stor delete`[282]	Builtin	Delete a table or specified rows in the in-memory sqlite database.
`stor export`[283]	Builtin	Export the in-memory sqlite database to a sqlite database file.
`stor import`[284]	Builtin	Import a sqlite database file into the in-memory sqlite database.
`stor insert`[285]	Builtin	Insert information into a specified table in the in-memory sqlite database.
`stor open`[286]	Builtin	Opens the in-memory sqlite database.
`stor reset`[287]	Builtin	Reset the in-memory database by dropping all tables.
`stor update`[288]	Builtin	Update information in a specified table in the in-memory sqlite database.

[281] `/commands/docs/stor_create.md`
[282] `/commands/docs/stor_delete.md`
[283] `/commands/docs/stor_export.md`
[284] `/commands/docs/stor_import.md`
[285] `/commands/docs/stor_insert.md`
[286] `/commands/docs/stor_open.md`
[287] `/commands/docs/stor_reset.md`
[288] `/commands/docs/stor_update.md`

stor create

version: 0.90.2

usage:

Create a table in the in-memory sqlite database.

Signature

```
> stor create --table-name --columns
```

Parameters

- `--table-name {string}`: name of the table you want to create
- `--columns {record}`: a record of column names and datatypes

Input/output types:

input	output
nothing	table

Examples

Create an in-memory sqlite database with specified table name, column names, and column data types

```
> stor create --table-name nudb --columns {bool1: bool, int1: int, float1: float, str1: str, datetime1: datetime}
```

stor delete

version: 0.90.2

usage:

Delete a table or specified rows in the in-memory sqlite database.

Signature

```
> stor delete --table-name --where-clause
```

Parameters

- `--table-name {string}`: name of the table you want to insert into
- `--where-clause {string}`: a sql string to use as a where clause without the WHERE keyword

Input/output types:

input	output
nothing	table

Examples

Delete a table from the in-memory sqlite database

```
> stor delete -- table-name nudb
```

Delete some rows from the in-memory sqlite database with a where clause

```
> stor delete -- table-name nudb -- where-clause "
int1 == 5 "
```

stor export

version: 0.90.2

usage:

Export the in-memory sqlite database to a sqlite database file.

Signature

```
> stor export --file-name
```

Parameters

- `--file-name {string}`: file name to export the sqlite in-memory database to

Input/output types:

input	output
nothing	table

Examples

Export the in-memory sqlite database

```
> stor export --file-name nudb.sqlite
```

stor import

version: 0.90.2

usage:

Import a sqlite database file into the in-memory sqlite database.

Signature

`> stor import --file-name`

Parameters

- `--file-name {string}`: file name to export the sqlite in-memory database to

Input/output types:

input	output
nothing	table

Examples

Import a sqlite database file into the in-memory sqlite database

```
> stor import --file-name nudb.sqlite
```

stor insert

version: 0.90.2

usage:

Insert information into a specified table in the in-memory sqlite database.

Signature

```
> stor insert --table-name --data-record
```

Parameters

- `--table-name {string}`: name of the table you want to insert into
- `--data-record {record}`: a record of column names and column values to insert into the specified table

Input/output types:

input	output
nothing	table

Examples

Insert data the in-memory sqlite database using a data-record of column-name and column-value pairs

```
> stor insert -- table-name nudb -- data-record {
bool1: true , int1: 5 , float1: 1.1 , str1: fdncred ,
datetime1: 2023-04-17 }
```

stor open

version: 0.90.2

usage:

Opens the in-memory sqlite database.

Signature

```
> stor open
```

Input/output types:

input	output
nothing	any

Examples

Open the in-memory sqlite database

```
> stor open
```

stor reset

version: 0.90.2

usage:

Reset the in-memory database by dropping all tables.

Signature

```
> stor reset
```

Input/output types:

input	output
nothing	table

Examples

Reset the in-memory sqlite database

```
> stor reset
```

stor update

version: 0.90.2

usage:

Update information in a specified table in the in-memory sqlite database.

Signature

```
> stor update --table-name --update-record --where-clause
```

Parameters

- `--table-name {string}`: name of the table you want to insert into
- `--update-record {record}`: a record of column names and column values to update in the specified table
- `--where-clause {string}`: a sql string to use as a where clause without the WHERE keyword

Input/output types:

input	output
nothing	table

Examples

Update the in-memory sqlite database

```
> stor update -- table-name nudb -- update-record {
str1: nushell datetime1: 2020-04-17 }
```

Update the in-memory sqlite database with a where clause

```
> stor update -- table-name nudb -- update-record {
str1: nushell datetime1: 2020-04-17 } -- where-clause
" bool1 = 1 "
```

str

version: 0.90.2

usage:

Various commands for working with string data.

Signature

```
> str
```

Input/output types:

input	output
nothing	string

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

Subcommands:

name	type	usage
`str camel-case`[289]	Builtin	Convert a string to camelCase.
`str capitalize`[290]	Builtin	Capitalize first letter of text.
`str contains`[291]	Builtin	Checks if string input contains a substring.
`str distance`[292]	Builtin	Compare two strings and return the edit distance/Levenshtein distance.
`str downcase`[293]	Builtin	Make text lowercase.
`str ends-with`[294]	Builtin	Check if an input ends with a string.
`str escape-glob`[295]	Builtin	Escape glob pattern.
`str expand`[296]	Builtin	Generates all possible combinations defined in brace expansion syntax.
`str index-of`[297]	Builtin	Returns start index of first occurrence of string in input, or -1 if no match.
`str join`[298]	Builtin	Concatenate multiple strings into a single string, with an optional separator between each.
`str kebab-case`[299]	Builtin	Convert a string to kebab-case.
`str length`[300]	Builtin	Output the length of any strings in the pipeline.
`str pascal-case`[301]	Builtin	Convert a string to PascalCase.
`str replace`[302]	Builtin	Find and replace text.
`str reverse`[303]	Builtin	Reverse every string in the pipeline.
`str screaming-snake-case`[304]	Builtin	Convert a string to SCREAMING

str camel-case

version: 0.90.2

usage:

Convert a string to camelCase.

Signature

```
> str camel-case ...rest
```

Parameters

- `...rest`: For a data structure input, convert strings at the given cell paths

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

[290] /commands/docs/str_capitalize.md
[291] /commands/docs/str_contains.md
[292] /commands/docs/str_distance.md
[293] /commands/docs/str_downcase.md
[294] /commands/docs/str_ends-with.md
[295] /commands/docs/str_escape-glob.md
[296] /commands/docs/str_expand.md
[297] /commands/docs/str_index-of.md
[298] /commands/docs/str_join.md
[299] /commands/docs/str_kebab-case.md
[300] /commands/docs/str_length.md
[301] /commands/docs/str_pascal-case.md
[302] /commands/docs/str_replace.md
[303] /commands/docs/str_reverse.md
[304] /commands/docs/str_screaming-snake-case.md
[305] /commands/docs/str_snake-case.md
[306] /commands/docs/str_starts-with.md
[307] /commands/docs/str_stats.md
[308] /commands/docs/str_substring.md
[309] /commands/docs/str_title-case.md
[310] /commands/docs/str_trim.md
[311] /commands/docs/str_upcase.md

Examples

convert a string to camelCase

```
> 'NuShell' | str camel-case
```

convert a string to camelCase

```
> 'this-is-the-first-case' | str camel-case
```

convert a string to camelCase

```
> 'this_is_the_second_case' | str camel-case
```

convert a column from a table to camelCase

```
> [[lang, gems]; [nu_test, 100]] | str camel-
case lang
```

str capitalize

version: 0.90.2

usage:

Capitalize first letter of text.

Signature

```
> str capitalize ...rest
```

Parameters

- `...rest`: For a data structure input, convert strings at the given cell paths.

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

Capitalize contents

```
> 'good day' | str capitalize
```

Capitalize contents

```
> 'anton' | str capitalize
```

Capitalize a column in a table

```
> [[lang, gems]; [nu_test, 100]] | str capitalize
lang
```

str contains

version: 0.90.2

usage:

Checks if string input contains a substring.

Signature

```
> str contains (string) ...rest --ignore-case --not
```

Parameters

- `string`: The substring to find.
- `...rest`: For a data structure input, check strings at the given cell paths, and replace with result.
- `--ignore-case`: search is case insensitive
- `--not`: does not contain

Input/output types:

input	output
list<string>	list<bool>
record	record
string	bool
table	table

Examples

Check if input contains string

```
> 'my_library.rb' | str contains '.rb'
```

Check if input contains string case insensitive

```
> 'my_library.rb' | str contains --ignore-case
'.RB'
```

Check if input contains string in a record

```
> { ColA: test, ColB: 100 } | str contains 'e'
 ColA
```

Check if input contains string in a table

```
>  [[ColA ColB]; [test 100]] | str contains -
-ignore-case 'E' ColA
```

Check if input contains string in a table

```
>  [[ColA ColB]; [test hello]] | str contains
'e' ColA ColB
```

Check if input string contains 'banana'

```
> 'hello' | str contains 'banana'
```

Check if list contains string

```
> [one two three] | str contains o
```

Check if list does not contain string

```
> [one two three] | str contains --not o
```

str distance

version: 0.90.2

usage:

Compare two strings and return the edit distance/Levenshtein distance.

Signature

```
> str distance (compare-string) ...rest
```

Parameters

- `compare-string`: The first string to compare.
- `...rest`: For a data structure input, check strings at the given cell paths, and replace with result.

Input/output types:

input	output
record	record
string	int
table	table

Examples

get the edit distance between two strings

```
> 'nushell' | str distance 'nutshell'
```

Compute edit distance between strings in table and another string, using cell paths

```
> [{a: 'nutshell' b: 'numetal'}] | str
distance 'nushell' 'a' 'b'
```

Compute edit distance between strings in record and another string, using cell paths

```
> {a: 'nutshell' b: 'numetal'} | str distance
'nushell' a b
```

str downcase

version: 0.90.2

usage:

Make text lowercase.

Signature

```
> str downcase ...rest
```

Parameters

- `...rest`: For a data structure input, convert strings at the given cell paths.

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

Downcase contents

```
> 'NU' | str downcase
```

Downcase contents

```
> 'TESTa' | str downcase
```

Downcase contents

```
> [[ColA ColB]; [Test ABC]] | str downcase
ColA
```

Downcase contents

```
> [[ColA ColB]; [Test ABC]] | str downcase
ColA ColB
```

str ends-with

version: 0.90.2

usage:

Check if an input ends with a string.

Signature

```
> str ends-with (string) ...rest --ignore-case
```

Parameters

- `string`: The string to match.
- `...rest`: For a data structure input, check strings at the given cell paths, and replace with result.
- `--ignore-case`: search is case insensitive

Input/output types:

input	output
list<string>	list<bool>
record	record
string	bool
table	table

Examples

Checks if string ends with '.rb'

```
> 'my_library.rb' | str ends-with '.rb'
```

Checks if strings end with '.txt'

```
> ['my_library.rb', 'README.txt'] | str ends-
with '.txt'
```

Checks if string ends with '.RB', case-insensitive

```
> 'my_library.rb' | str ends-with --ignore-case
 '.RB'
```

str escape-glob

version: 0.90.2

usage:

Escape glob pattern.

Signature

```
> str escape-glob ...rest
```

Parameters

- `...rest`: For a data structure input, turn strings at the given cell paths into substrings.

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

escape glob pattern before list

```
> let f = 'test[a]'; ls ($f | str escape-glob
)
```

str expand

version: 0.90.2

usage:

Generates all possible combinations defined in brace expansion syntax.

Signature

```
> str expand --path
```

Parameters

- `--path`: Replaces all backslashes with double backslashes, useful for Path.

Input/output types:

input	output
list<string>	list<list<string>>
string	list<string>

Examples

Define a range inside braces to produce a list of string.

```
> "{3..5}" | str expand
```

Ignore the next character after the backslash (”)

```
> 'A{B\,,C}' | str expand
```

Commas that are not inside any braces need to be skipped.

```
> 'Welcome\, {home,mon ami}!' | str expand
```

Use double backslashes to add a backslash.

```
> 'A{B\\,C}' | str expand
```

Export comma separated values inside braces ({}) to a string list.

```
> "{apple,banana,cherry}" | str expand
```

If the piped data is path, you may want to use --path flag, or else manually replace the backslashes with double backslashes.

```
> 'C:\{Users,Windows}' | str expand --path
```

Brace expressions can be used one after another.

```
> "A{b,c}D{e,f}G" | str expand
```

Collection may include an empty item. It can be put at the start of the list.

```
> "A{,B,C}" | str expand
```

Empty item can be at the end of the collection.

```
> "A{B,C,}" | str expand
```

Empty item can be in the middle of the collection.

```
> "A{B,,C}" | str expand
```

Also, it is possible to use one inside another. Here is a real-world example, that creates files:

```
> "A{B{1,3},C{2,5}}D" | str expand
```

Notes

```
This syntax may seem familiar with `glob {A,B}.C`. The
difference is glob relies on filesystem, but str expand
is not. Inside braces, we put variants. Then basically
we're creating all possible outcomes.
```

str index-of

version: 0.90.2

usage:

Returns start index of first occurrence of string in input, or -1 if no match.

Signature

```
> str index-of (string) ...rest --grapheme-clusters --utf-8-bytes --range --end
```

Parameters

- `string`: The string to find in the input.
- `...rest`: For a data structure input, search strings at the given cell paths, and replace with result.
- `--grapheme-clusters`: count indexes using grapheme clusters (all visible chars have length 1)

- `--utf-8-bytes`: count indexes using UTF-8 bytes (default; non-ASCII chars have length 2+)
- `--range {range}`: optional start and/or end index
- `--end`: search from the end of the input

Input/output types:

input	output
list<string>	list<int>
record	record
string	int
table	table

Examples

Returns index of string in input

```
>  ' my_library.rb '  |  str  index-of  ' .rb '
```

Count length using grapheme clusters

```
>  '          '  |  str  index-of  --grapheme-clusters
'   '
```

Returns index of string in input within a`rhs open range`

```
>  ' .rb.rb '  |  str  index-of  ' .rb '  --range  1..
```

Returns index of string in input within a lhs open range

```
>  ' 123456 '  |  str  index-of  ' 6 '  --range  ..4
```

Returns index of string in input within a range

```
>  ' 123456 '  |  str  index-of  ' 3 '  --range  1..4
```

Returns index of string in input

```
>  ' /this/is/some/path/file.txt '  |  str  index-of  '
/ '  - e
```

str join

version: 0.90.2

usage:

Concatenate multiple strings into a single string, with an optional separator between each.

Signature

```
> str join (separator)
```

Parameters

- `separator`: Optional separator to use when creating string.

Input/output types:

input	output
list<any>	string
string	string

Examples

Create a string from input

```
> ['nu', 'shell'] | str join
```

Create a string from input with a separator

```
> ['nu', 'shell'] | str join '-'
```

str kebab-case

version: 0.90.2

usage:

Convert a string to kebab-case.

Signature

```
> str kebab-case ...rest
```

Parameters

- `...rest`: For a data structure input, convert strings at the given cell paths

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

convert a string to kebab-case

```
> 'NuShell' | str kebab-case
```

convert a string to kebab-case

```
> 'thisIsTheFirstCase' | str kebab-case
```

convert a string to kebab-case

```
> 'THIS_IS_THE_SECOND_CASE' | str kebab-case
```

convert a column from a table to kebab-case

```
> [[lang, gems]; [nuTest, 100]] | str kebab-
case lang
```

str length

version: 0.90.2

usage:

Output the length of any strings in the pipeline.

Signature

```
> str length ...rest --grapheme-clusters --utf-8-bytes
```

Parameters

- `...rest`: For a data structure input, replace strings at the given cell paths with their length.
- `--grapheme-clusters`: count length using grapheme clusters (all visible chars have length 1)
- `--utf-8-bytes`: count length using UTF-8 bytes (default; all non-ASCII chars have length 2+)

Input/output types:

input	output
list<string>	list<int>
record	record
string	int
table	table

Examples

Return the lengths of a string

```
> 'hello' | str length
```

Count length using grapheme clusters

```
> '        ' | str length --grapheme-clusters
```

Return the lengths of multiple strings

```
> ['hi' 'there'] | str length
```

str pascal-case

version: 0.90.2

usage:

Convert a string to PascalCase.

Signature

`> str pascal-case ...rest`

Parameters

- `...rest`: For a data structure input, convert strings at the given cell paths

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

convert a string to PascalCase

```
> 'nu-shell' | str pascal-case
```

convert a string to PascalCase

```
> 'this-is-the-first-case' | str pascal-case
```

convert a string to PascalCase

```
> 'this_is_the_second_case' | str pascal-case
```

convert a column from a table to PascalCase

```
> [[lang, gems]; [nu_test, 100]] | str pascal-
case lang
```

str replace

version: 0.90.2

usage:

Find and replace text.

Signature

```
> str replace (find) (replace) ...rest --all --no-expand --
regex --multiline
```

Parameters

- `find`: The pattern to find.
- `replace`: The replacement string.
- `...rest`: For a data structure input, operate on strings at the given cell paths.
- `--all`: replace all occurrences of the pattern
- `--no-expand`: do not expand capture groups (like $name) in the replacement string
- `--regex`: match the pattern as a regular expression in the input, instead of a substring
- `--multiline`: multi-line regex mode (implies --regex): ^ and $ match begin/end of line; equivalent to (?m)

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

Find and replace the first occurrence of a substring

```
> 'c:\some\cool\path' | str replace 'c:\some\cool' '~'
```

Find and replace all occurrences of a substring

```
> 'abc abc abc' | str replace --all 'b' 'z'
```

Find and replace contents with capture group using regular expression

```
> 'my_library.rb' | str replace -r '(.+).rb' '$1.nu'
```

Find and replace all occurrences of find string using regular expression

```
> 'abc abc abc' | str replace --all --regex '
b' 'z'
```

Find and replace all occurrences of find string in table using regular expression

```
> [[ColA ColB ColC]; [abc abc ads]] | str
replace --all --regex 'b' 'z' ColA ColC
```

Find and replace all occurrences of find string in record using regular expression

```
> { KeyA: abc, KeyB: abc, KeyC: ads } | str
replace --all --regex 'b' 'z' KeyA KeyC
```

Find and replace contents without using the replace parameter as a regular expression

```
> 'dogs_$1_cats' | str replace -r '\$1' '$2
' -n
```

Use captures to manipulate the input text using regular expression

```
> "abc-def" | str replace -r "(.+)-(.+)" "$
{2}_${1}"
```

Find and replace with fancy-regex using regular expression

```
> 'a successful b' | str replace -r '\b([sS])uc(
?:cs|s?)e(ed(?:ed|ing|s?)|ss(?:es|ful(?:ly)?|i(?:ons?|ve(
?:ly)?)|ors?)?)\b' '${1}ucce$2'
```

Find and replace with fancy-regex using regular expression

```
> 'GHIKK-9+*' | str replace -r '[*[:xdigit:]+]
' 'z'
```

Find and replace on individual lines using multiline regular expression

```
> "non-matching line\n123. one line\n124. another line\n
" | str replace --all --multiline '^[0-9]+\. '
  ''
```

str reverse

version: 0.90.2

usage:

Reverse every string in the pipeline.

Signature

```
> str reverse ...rest
```

Parameters

- `...rest`: For a data structure input, reverse strings at the given cell paths.

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

Reverse a single string

```
> 'Nushell' | str reverse
```

Reverse multiple strings in a list

```
> ['Nushell' 'is' 'cool'] | str reverse
```

str screaming-snake-case

version: 0.90.2

usage:

Convert a string to SCREAMING_SNAKE_CASE.

Signature

```
> str screaming-snake-case ...rest
```

Parameters

- `...rest`: For a data structure input, convert strings at the given cell paths

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

convert a string to SCREAMING_SNAKE_CASE

```
> "NuShell" | str screaming-snake-case
```

convert a string to SCREAMING_SNAKE_CASE

```
> "this_is_the_second_case" | str screaming-snake-
case
```

convert a string to SCREAMING_SNAKE_CASE

```
> "this-is-the-first-case" | str screaming-snake-
case
```

convert a column from a table to SCREAMING_SNAKE_CASE

```
> [[lang, gems]; [nu_test, 100]] | str screaming-
snake-case lang
```

str snake-case

version: 0.90.2

usage:

Convert a string to snake_case.

Signature

```
> str snake-case ...rest
```

Parameters

- `...rest`: For a data structure input, convert strings at the given cell paths

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

convert a string to snake_case

```
> "NuShell" | str snake-case
```

convert a string to snake_case

```
> "this_is_the_second_case" | str snake-case
```

convert a string to snake_case

```
> "this-is-the-first-case" | str snake-case
```

convert a column from a table to snake_case

```
> [[lang, gems]; [nuTest, 100]] | str snake-
case lang
```

str starts-with

version: 0.90.2

usage:

Check if an input starts with a string.

Signature

```
> str starts-with (string) ...rest --ignore-case
```

Parameters

- `string`: The string to match.
- `...rest`: For a data structure input, check strings at the given cell paths, and replace with result.
- `--ignore-case`: search is case insensitive

Input/output types:

input	output
list<string>	list<bool>
record	record
string	bool
table	table

Examples

Checks if input string starts with 'my'

```
> 'my_library.rb' | str starts-with 'my'
```

Checks if input string starts with 'Car'

```
> 'Cargo.toml' | str starts-with 'Car'
```

Checks if input string starts with '.toml'

```
> 'Cargo.toml' | str starts-with '.toml'
```

Checks if input string starts with 'cargo', case-insensitive

```
> 'Cargo.toml' | str starts-with --ignore-case
'cargo'
```

str stats

version: 0.90.2

usage:

Gather word count statistics on the text.

Signature

```
> str stats
```

Input/output types:

input	output
string	record

Examples

Count the number of words in a string

```
> "There are seven words in this sentence" | str
stats
```

Counts unicode characters

```
> '     ' | str stats
```

Counts Unicode characters correctly in a string

```
> "Amélie Amelie" | str stats
```

str substring

version: 0.90.2

usage:

Get part of a string. Note that the start is included but the end is excluded, and that the first character of a string is index 0.

Signature

```
> str substring (range) ...rest --grapheme-clusters --utf-8-
bytes
```

Parameters

- `range`: The indexes to substring [start end].
- `...rest`: For a data structure input, turn strings at the given cell paths into substrings.
- `--grapheme-clusters`: count indexes and split using grapheme clusters (all visible chars have length 1)
- `--utf-8-bytes`: count indexes and split using UTF-8 bytes (default; non-ASCII chars have length 2+)

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

Get a substring "nushell" from the text "good nushell" using a range

```
> 'good nushell' | str substring 5..12
```

Count indexes and split using grapheme clusters

```
> '          ' | str substring --grapheme-clusters
4..6
```

str title-case

version: 0.90.2

usage:

Convert a string to Title Case.

Signature

```
> str title-case ...rest
```

Parameters

- `...rest`: For a data structure input, convert strings at the given cell paths

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

convert a string to Title Case

```
> 'nu-shell' | str title-case
```

convert a string to Title Case

```
> 'this is a test case' | str title-case
```

convert a column from a table to Title Case

```
> [[title, count]; ['nu test', 100]] | str
title-case title
```

str trim

version: 0.90.2

usage:

Trim whitespace or specific character.

Signature

```
> str trim ...rest --char --left --right
```

Parameters

- `...rest`: For a data structure input, trim strings at the given cell paths.
- `--char {string}`: character to trim (default: whitespace)
- `--left`: trims characters only from the beginning of the string
- `--right`: trims characters only from the end of the string

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

Trim whitespace

```
> ' Nu shell ' | str trim
```

Trim a specific character (not the whitespace)

```
> '=== Nu shell ===' | str trim --char '='
```

Trim whitespace from the beginning of string

```
> ' Nu shell ' | str trim --left
```

Trim whitespace from the end of string

```
> ' Nu shell ' | str trim --right
```

Trim a specific character only from the end of the string

```
> '=== Nu shell ===' | str trim --right --char
  '='
```

str upcase

version: 0.90.2

usage:

Make text uppercase.

Signature

```
> str upcase ...rest
```

Parameters

- `...rest`: For a data structure input, convert strings at the given cell paths.

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

Upcase contents

```
> 'nu' | str upcase
```

sys

version: 0.90.2

usage:

View information about the system.

Signature

```
> sys
```

Input/output types:

input	output
nothing	record

Examples

Show info about the system

```
> sys
```

Show the os system name with get

```
> (sys ) .host | get name
```

Show the os system name

```
> (sys ) .host.name
```

table

version: 0.90.2

usage:

Render the table.

Signature

```
> table --theme --index --width --expand --expand-deep --flatten
--flatten-separator --collapse --abbreviated --list
```

Parameters

- `--theme {string}`: set a table mode/theme
- `--index {any}`: enable (true) or disable (false) the #/index column or set the starting index
- `--width {int}`: number of terminal columns wide (not output columns)
- `--expand`: expand the table structure in a light mode
- `--expand-deep {int}`: an expand limit of recursion which will take place, must be used with --expand
- `--flatten`: Flatten simple arrays

- `--flatten-separator {string}`: sets a separator when 'flatten' used
- `--collapse`: expand the table structure in collapse mode. Be aware collapse mode currently doesn't support width control
- `--abbreviated {int}`: abbreviate the data in the table by truncating the middle part and only showing amount provided on top and bottom
- `--list`: list available table modes/themes

Input/output types:

input	output
any	any

Examples

List the files in current directory, with indexes starting from 1

```
> ls | table --index 1
```

Render data in table view

```
> [[a b]; [1 2] [3 4]] | table
```

Render data in table view (expanded)

```
> [[a b]; [1 2] [2 [4 4]]] | table --
expand
```

Render data in table view (collapsed)

```
> [[a b]; [1 2] [2 [4 4]]] | table --
collapse
```

Change the table theme to the specified theme for a single run

```
> [[a b]; [1 2] [2 [4 4]]] | table --theme
 basic
```

Force showing of the #/index column for a single run

```
> [[a b]; [1 2] [2 [4 4]]] | table -i
true
```

Set the starting number of the #/index column to 100 for a single run

```
> [[a b]; [1 2] [2 [4 4]]] | table -i
100
```

Force hiding of the #/index column for a single run

```
> [[a b]; [1 2] [2 [4 4]]] | table -i
false
```

Notes

```
If the table contains a column called 'index', this column
is used as the table index instead of the usual continuous
index.
```

take

version: 0.90.2

usage:

Take only the first n elements of a list, or the first n bytes of a binary value.

Signature

`> take (n)`

Parameters

- `n`: Starting from the front, the number of elements to return.

Input/output types:

input	output
binary	binary
list<any>	list<any>
range	list<number>
table	table

Examples

Return the first item of a list/table

```
> [1 2 3] | take 1
```

Return the first 2 items of a list/table

```
> [1 2 3] | take 2
```

Return the first two rows of a table

```
> [[editions]; [2015] [2018] [2021]] |
take 2
```

Return the first 2 bytes of a binary value

```
> 0x[01 23 45] | take 2
```

Return the first 3 elements of a range

```
> 1..10 | take 3
```

Subcommands:

name	type	usage
`take until`[312]	Builtin	Take elements of the input until a predicate is true.
`take while`[313]	Builtin	Take elements of the input while a predicate is true.

take until

version: 0.90.2

usage:

Take elements of the input until a predicate is true.

[312] /commands/docs/take_until.md
[313] /commands/docs/take_while.md

Signature

`> take until (predicate)`

Parameters

- `predicate`: The predicate that element(s) must not match.

Input/output types:

input	output
list<any>	list<any>
table	table

Examples

Take until the element is positive

```
> [-1 -2 9 1] | take until {|x| $x > 0 }
```

Take until the element is positive using stored condition

```
> let cond = {|x| $x > 0 }; [-1 -2 9 1] |
take until $cond
```

Take until the field value is positive

```
> [{a: -1} {a: -2} {a: 9} {a: 1}] | take
 until {|x| $x.a > 0 }
```

take while

version: 0.90.2

usage:

Take elements of the input while a predicate is true.

Signature

`> take while (predicate)`

Parameters

- `predicate`: The predicate that element(s) must match.

Input/output types:

input	output
list<any>	list<any>
table	table

Examples

Take while the element is negative

```
> [-1 -2 9 1] | take while {|x| $x < 0 }
```

Take while the element is negative using stored condition

```
> let cond = {|x| $x < 0 }; [-1 -2 9 1] |
take while $cond
```

Take while the field value is negative

```
> [{a: -1} {a: -2} {a: 9} {a: 1}] | take
 while {|x| $x.a < 0 }
```

term size

version: 0.90.2

usage:

Returns a record containing the number of columns (width) and rows (height) of the terminal.

Signature

```
> term size
```

Input/output types:

input	output
nothing	record<columns: int, rows: int>

Examples

Return the columns (width) and rows (height) of the terminal

```
> term size
```

Return the columns (width) of the terminal

```
> (term size ) .columns
```

Return the rows (height) of the terminal

```
> (term size ) .rows
```

timeit

version: 0.90.2

usage:

Time the running time of a block.

Signature

`> timeit (command)`

Parameters

- `command`: The command or block to run.

Input/output types:

input	output
any	duration
nothing	duration

Examples

Times a command within a closure

```
> timeit { sleep 500ms }
```

Times a command using an existing input

```
> http get https://www.nushell.sh/book/ | timeit {
split chars }
```

Times a command invocation

```
> timeit ls - la
```

to

version: 0.90.2

usage:

Translate structured data to a format.

Signature

`> to`

Input/output types:

input	output
nothing	string

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

Subcommands:

name	type	usage
`to csv`[314]	Builtin	Convert table into .csv text .
`to html`[315]	Builtin	Convert table into simple HTML.
`to json`[316]	Builtin	Converts table data into JSON text.
`to md`[317]	Builtin	Convert table into simple Markdown.
`to nuon`[318]	Builtin	Converts table data into Nuon (Nushell Object Notation) text.
`to text`[319]	Builtin	Converts data into simple text.
`to toml`[320]	Builtin	Convert record into .toml text.
`to tsv`[321]	Builtin	Convert table into .tsv text.
`to xml`[322]	Builtin	Convert special record structure into .xml text.
`to yaml`[323]	Builtin	Convert table into .yaml/.yml text.

to csv

version: 0.90.2

[314] `/commands/docs/to_csv.md`
[315] `/commands/docs/to_html.md`
[316] `/commands/docs/to_json.md`
[317] `/commands/docs/to_md.md`
[318] `/commands/docs/to_nuon.md`
[319] `/commands/docs/to_text.md`
[320] `/commands/docs/to_toml.md`
[321] `/commands/docs/to_tsv.md`
[322] `/commands/docs/to_xml.md`
[323] `/commands/docs/to_yaml.md`

usage:

Convert table into .csv text .

Signature

```
> to csv --separator --noheaders
```

Parameters

- `--separator {string}`: a character to separate columns, defaults to ‘,’
- `--noheaders`: do not output the columns names as the first row

Input/output types:

input	output
record	string
table	string

Examples

Outputs an CSV string representing the contents of this table

```
> [[foo bar]; [1 2]] | to csv
```

Outputs an CSV string representing the contents of this table

```
> [[foo bar]; [1 2]] | to csv --separator '
;'
```

Outputs an CSV string representing the contents of this record

```
> {a: 1 b: 2} | to csv
```

to html

version: 0.90.2

usage:

Convert table into simple HTML.

Signature

```
> to html --html-color --no-color --dark --partial --theme -
-list
```

Parameters

- `--html-color`: change ansi colors to html colors
- `--no-color`: remove all ansi colors in output
- `--dark`: indicate your background color is a darker color
- `--partial`: only output the html for the content itself
- `--theme {string}`: the name of the theme to use (github, blulocolight, ...)
- `--list`: produce a color table of all available themes

Input/output types:

input	output
any	string

Examples

Outputs an HTML string representing the contents of this table

```
> [[foo bar]; [1 2]] | to html
```

Optionally, only output the html for the content itself

```
> [[foo bar]; [1 2]] | to html --partial
```

Optionally, output the string with a dark background

```
> [[foo bar]; [1 2]] | to html --dark
```

Notes

```
Screenshots of the themes can be browsed here: https://
github.com/mbadolato/iTerm2-Color-Schemes.
```

to json

version: 0.90.2

usage:

Converts table data into JSON text.

Signature

```
> to json --raw --indent --tabs
```

Parameters

- `--raw`: remove all of the whitespace
- `--indent {number}`: specify indentation width
- `--tabs {number}`: specify indentation tab quantity

Input/output types:

input	output
any	string

Examples

Outputs a JSON string, with default indentation, representing the contents of this table

```
> [a b c] | to json
```

Outputs a JSON string, with 4-space indentation, representing the contents of this table

```
> [Joe Bob Sam] | to json --indent 4
```

Outputs an unformatted JSON string representing the contents of this table

```
> [1 2 3] | to json -r
```

to md

version: 0.90.2

usage:

Convert table into simple Markdown.

Signature

```
> to md --pretty --per-element
```

Parameters

- `--pretty`: Formats the Markdown table to vertically align items
- `--per-element`: treat each row as markdown syntax element

Input/output types:

input	output
any	string

Examples

Outputs an MD string representing the contents of this table

```
> [[foo bar]; [1 2]] | to md
```

Optionally, output a formatted markdown string

```
> [[foo bar]; [1 2]] | to md --pretty
```

Treat each row as a markdown element

```
> [{"H1": "Welcome to Nushell" } [[foo bar]
; [1 2]]] | to md --per-element --pretty
```

Render a list

```
> [0 1 2] | to md --pretty
```

to nuon

version: 0.90.2

usage:

Converts table data into Nuon (Nushell Object Notation) text.

Signature

```
> to nuon --raw --indent --tabs
```

Parameters

- `--raw`: remove all of the whitespace (default behaviour and overwrites -i and -t)
- `--indent {number}`: specify indentation width
- `--tabs {number}`: specify indentation tab quantity

Input/output types:

input	output
any	string

Examples

Outputs a NUON string representing the contents of this list, compact by default

```
> [1 2 3] | to nuon
```

Outputs a NUON array of ints, with pretty indentation

```
> [1 2 3] | to nuon --indent 2
```

Overwrite any set option with --raw

```
> [1 2 3] | to nuon --indent 2 --raw
```

A more complex record with multiple data types

```
> {date: 2000-01-01, data: [1 [2 3] 4.56]} |
to nuon --indent 2
```

to text

version: 0.90.2

usage:

Converts data into simple text.

Signature

```
> to text
```

Input/output types:

input	output
any	string

Examples

Outputs data as simple text

```
> 1 | to text
```

Outputs external data as simple text

```
> git help -a | lines | find -r '^ ' | to
text
```

Outputs records as simple text

```
> ls | to text
```

to toml

version: 0.90.2

usage:

Convert record into .toml text.

Signature

```
> to toml
```

Input/output types:

input	output
record	string

Examples

Outputs an TOML string representing the contents of this record

```
> { foo: 1 bar: 'qwe'} | to toml
```

to tsv

version: 0.90.2

usage:

Convert table into .tsv text.

Signature

```
> to tsv --noheaders
```

Parameters

- `--noheaders`: do not output the column names as the first row

Input/output types:

input	output
record	string
table	string

Examples

Outputs an TSV string representing the contents of this table

```
> [[foo bar]; [1 2]] | to tsv
```

Outputs an TSV string representing the contents of this record

```
> {a: 1 b: 2} | to tsv
```

to xml

version: 0.90.2

usage:

Convert special record structure into .xml text.

Signature

```
> to xml --indent --partial-escape --self-closed
```

Parameters

- `--indent {int}`: Formats the XML text with the provided indentation setting
- `--partial-escape`: Only escape mandatory characters in text and attributes
- `--self-closed`: Output empty tags as self closing

Input/output types:

input	output
record	string

Examples

Outputs an XML string representing the contents of this table

```
> {tag: note attributes: {} content : [{tag: remember
attributes: { } content : [ { tag: null attributes:
null content : Event } ] } ] } | to xml
```

When formatting xml null and empty record fields can be omitted and strings can be written without a wrapping record

```
> { tag: note content : [{ tag: remember content : [
Event ] } ] } | to xml
```

Optionally, formats the text with a custom indentation setting

```
> { tag: note content : [{ tag: remember content : [
Event ] } ] } | to xml -- indent 3
```

Produce less escaping sequences in resulting xml

```
> {tag: note attributes: {a: "'qwe'\\"} content:
  ["\"'"]} | to xml --partial-escape
```

Save space using self-closed tags

```
> {tag: root content: [[tag]; [a] [b] [c]]
} | to xml --self-closed
```

Notes

```
Every XML entry is represented via a record with tag, attribute
and content fields. To represent different types of entries
different values must be written to this fields: 1. Tag
entry: `{tag: <tag name> attributes: {<attr name>: "<string
value>" ...} content: [<entries>]}` 2. Comment entry: `{tag:
 '!' attributes: null content: "<comment string>"}` 3.
Processing instruction (PI): `{tag: '?<pi name>' attributes:
 null content: "<pi content string>"}` 4. Text: `{tag:
null attributes: null content: "<text>"}`. Or as plain
`<text>` instead of record. Additionally any field which
is: empty record, empty list or null, can be omitted.
```

to yaml

version: 0.90.2

usage:

Convert table into .yaml/.yml text.

Signature

> to yaml

Input/output types:

input	output
any	string

Examples

Outputs an YAML string representing the contents of this table

```
> [[foo bar]; ["1" "2"]] | to yaml
```

touch

version: 0.90.2

usage:

Creates one or more files.

Signature

`> touch (filename) ...rest --reference --modified --access --no-create`

Parameters

- `filename`: The path of the file you want to create.
- `...rest`: Additional files to create.
- `--reference {string}`: change the file or directory time to the time of the reference file/directory
- `--modified`: change the modification time of the file or directory. If no timestamp, date or reference file/directory is given, the current time is used
- `--access`: change the access time of the file or directory. If no timestamp, date or reference file/directory is given, the current time is used
- `--no-create`: do not create the file if it does not exist

Input/output types:

input	output
nothing	nothing

Examples

Creates “fixture.json”

```
> touch fixture.json
```

Creates files a, b and c

```
> touch a b c
```

Changes the last modified time of “fixture.json” to today’s date

```
> touch -m fixture.json
```

Changes the last modified time of files a, b and c to a date

```
> touch -m -d "yesterday" a b c
```

Changes the last modified time of file d and e to “fixture.json”’s last modified time

```
> touch -m -r fixture.json d e
```

Changes the last accessed time of “fixture.json” to a date

```
> touch -a -d "August 24, 2019; 12:30:30" fixture.
json
```

transpose

version: 0.90.2

usage:

Transposes the table contents so rows become columns and columns become rows.

Signature

```
> transpose ...rest --header-row --ignore-titles --as-record
--keep-last --keep-all
```

Parameters

- `...rest`: The names to give columns once transposed.
- `--header-row`: treat the first row as column names
- `--ignore-titles`: don't transpose the column names into values
- `--as-record`: transfer to record if the result is a table and contains only one row
- `--keep-last`: on repetition of record fields due to `header-row`, keep the last value obtained
- `--keep-all`: on repetition of record fields due to `header-row`, keep all the values obtained

Input/output types:

input	output
record	table
table	any

Examples

Transposes the table contents with default column names

```
> [[c1 c2]; [1 2]] | transpose
```

Transposes the table contents with specified column names

```
> [[c1 c2]; [1 2]] | transpose key val
```

Transposes the table without column names and specify a new column name

```
> [[c1 c2]; [1 2]] | transpose --ignore-titles
 val
```

Transfer back to record with -d flag

```
> {c1: 1, c2: 2} | transpose | transpose --
ignore-titles -r -d
```

try

version: 0.90.2

usage:

Try to run a block, if it fails optionally run a catch block.

Signature

```
> try (try_block) (catch_block)
```

Parameters

- `try_block`: Block to run.
- `catch_block`: Block to run if try block fails.

Input/output types:

input	output
any	any

Examples

Try to run a missing command

```
> try { asdfasdf }
```

Try to run a missing command

```
> try { asdfasdf } catch { 'missing' }
```

tutor

version: 0.90.2

usage:

Run the tutorial. To begin, run: tutor.

Signature

```
> tutor (search) --find
```

Parameters

- `search`: Item to search for, or 'list' to list available tutorials.
- `--find {string}`: Search tutorial for a phrase

Input/output types:

input	output
nothing	string

Examples

Begin the tutorial

```
> tutor begin
```

Search a tutorial by phrase

```
> tutor -- find " $ in "
```

ulimit

version: 0.90.2

usage:

Set or get resource usage limits.

Signature

```
> ulimit (limit) --soft --hard --all --core-size --data-size
--file-size --file-descriptor-count --stack-size --cpu-time
--virtual-memory-size
```

Parameters

- `limit`: Limit value.
- `--soft`: Sets soft resource limit
- `--hard`: Sets hard resource limit
- `--all`: Prints all current limits
- `--core-size`: Maximum size of core files created
- `--data-size`: Maximum size of a process's data segment
- `--file-size`: Maximum size of files created by the shell
- `--file-descriptor-count`: Maximum number of open file descriptors
- `--stack-size`: Maximum stack size
- `--cpu-time`: Maximum amount of CPU time in seconds
- `--virtual-memory-size`: Maximum amount of virtual memory available to each process

Input/output types:

input	output
nothing	any

Examples

Print all current limits

```
> ulimit -a
```

Print specified limits

```
> ulimit --core-size --data-size --file-size
```

Set limit

```
> ulimit --core-size 102400
```

Set stack size soft limit

```
> ulimit -s -S 10240
```

Set virtual memory size hard limit

```
> ulimit -v -H 10240
```

Set core size limit to unlimited

```
> ulimit -c unlimited
```

umkdir

version: 0.90.2

usage:

Create directories, with intermediary directories if required using uutils/coreutils mkdir.

Signature

`> umkdir ...rest --verbose`

Parameters

- `...rest`: The name(s) of the path(s) to create.
- `--verbose`: print a message for each created directory.

Input/output types:

input	output
nothing	nothing

Examples

Make a directory named foo

```
> umkdir foo
```

Make multiple directories and show the paths created

```
> umkdir -v foo/bar foo2
```

umv

version: 0.90.2

usage:

Move files or directories.

Signature

`> umv ...rest --force --verbose --progress --interactive --no-clobber`

Parameters

- `...rest`: Rename SRC to DST, or move SRC to DIR.
- `--force`: do not prompt before overwriting
- `--verbose`: explain what is being done.
- `--progress`: display a progress bar
- `--interactive`: prompt before overwriting
- `--no-clobber`: do not overwrite an existing file

Input/output types:

input	output
nothing	nothing

Examples

Rename a file

```
> umv before.txt after.txt
```

Move a file into a directory

```
> umv test.txt my/subdirectory
```

Move many files into a directory

```
> umv *.txt my/subdirectory
```

uniq-by

version: 0.90.2

usage:

Return the distinct values in the input by the given column(s).

Signature

`> uniq-by ...rest --count --repeated --ignore-case --unique`

Parameters

- `...rest`: The column(s) to filter by.
- `--count`: Return a table containing the distinct input values together with their counts
- `--repeated`: Return the input values that occur more than once
- `--ignore-case`: Ignore differences in case when comparing input values
- `--unique`: Return the input values that occur once only

Input/output types:

input	output
list<any>	list<any>
table	table

Examples

Get rows from table filtered by column uniqueness

```
> [[fruit count]; [apple 9] [apple 2] [pear 3
] [orange 7]] | uniq-by fruit
```

uniq

version: 0.90.2

usage:

Return the distinct values in the input.

Signature

`> uniq --count --repeated --ignore-case --unique`

Parameters

- `--count`: Return a table containing the distinct input values together with their counts
- `--repeated`: Return the input values that occur more than once
- `--ignore-case`: Compare input values case-insensitively
- `--unique`: Return the input values that occur once only

Input/output types:

input	output
list<any>	list<any>

Examples

Return the distinct values of a list/table (remove duplicates so that each value occurs once only)

```
> [2 3 3 4] | uniq
```

Return the input values that occur more than once

```
> [1 2 2] | uniq -d
```

Return the input values that occur once only

```
> [1 2 2] | uniq -- unique
```

Ignore differences in case when comparing input values

```
> ['hello' 'goodbye' 'Hello'] | uniq --
ignore-case
```

Return a table containing the distinct input values together with their counts

```
> [1 2 2] | uniq -- count
```

update

version: 0.90.2

usage:

Update an existing column to have a new value.

Signature

`> update (field) (replacement value)`

Parameters

- `field`: The name of the column to update.
- `replacement value`: The new value to give the cell(s), or a closure to create the value.

Input/output types:

input	output
list<any>	list<any>
record	record
table	table

Examples

Update a column value

```
> {'name': 'nu', 'stars': 5} | update name
'Nushell'
```

Use a closure to alter each value in the 'authors' column to a single string

```
> [[project, authors]; ['nu', ['Andrés', 'JT
', 'Yehuda']]] | update authors {|row| $row.
authors | str join ','}
```

You can also use a simple command to update 'authors' to a single string

```
> [[project, authors]; ['nu', ['Andrés', 'JT
', 'Yehuda']]] | update authors { str join ',
' }
```

Update a value at an index in a list

```
> [1 2 3] | update 1 4
```

Use a closure to compute a new value at an index

```
> [1 2 3] | update 1 {|i| $i + 2 }
```

Subcommands:

name	type	usage
update cells[324]	Builtin	Update the table cells.

update cells

version: 0.90.2

usage:

Update the table cells.

Signature

```
> update cells (closure) --columns
```

[324] /commands/docs/update_cells.md

Parameters

- `closure`: the closure to run an update for each cell
- `--columns {list<any>}`: list of columns to update

Input/output types:

input	output
table	table

Examples

Update the zero value cells to empty strings.

```
> [           [ " 2021-04-16 " ,  " 2021-06-10 " ,  " 2021-
09-18 " ,  " 2021-10-15 " ,  " 2021-11-16 " ,  " 2021-11-17 "
,  " 2021-11-18 " ] ;           [            37,
0,              0,              0,             37,
 0,              0 ]       ]  |  update  cells  {  |value|
         if  $ value  == 0  {                "  "
}  else  {               $ value              }      }
```

Update the zero value cells to empty strings in 2 last columns.

```
> [           [ " 2021-04-16 " ,  " 2021-06-10 " ,  " 2021-
09-18 " ,  " 2021-10-15 " ,  " 2021-11-16 " ,  " 2021-11-17 "
,  " 2021-11-18 " ] ;           [            37,
0,              0,              0,             37,
 0,              0 ]       ]  |  update  cells  - c  [ " 2021-
11-18 " ,  " 2021-11-17 " ]   {  |value|            if  $
value  == 0  {               "  "              }  else  {
             $ value             }      }
```

upsert

version: 0.90.2

usage:

Update an existing column to have a new value, or insert a new column.

Signature

```
> upsert (field) (replacement value)
```

Parameters

- `field`: The name of the column to update or insert.
- `replacement value`: The new value to give the cell(s), or a closure to create the value.

Input/output types:

input	output
list<any>	list<any>
record	record
table	table

Examples

Update a record's value

```
> {'name': 'nu', 'stars': 5} | upsert name
'Nushell'
```

Insert a new entry into a record

```
> {'name': 'nu', 'stars': 5} | upsert
language 'Rust'
```

Update each row of a table

```
> [[name lang]; [Nushell ''] [Reedline '']]
  | upsert lang 'Rust'
```

Insert a new column with values computed based off the other columns

```
> [[foo]; [7] [8] [9]] | upsert bar {
|row| $row.foo * 2 }
```

Upsert into a list, updating an existing value at an index

```
> [1 2 3] | upsert 0 2
```

Upsert into a list, inserting a new value at the end

```
> [1 2 3] | upsert 3 4
```

url

version: 0.90.2

usage:

Various commands for working with URLs.

Signature

```
> url
```

Input/output types:

input	output
nothing	string

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

Subcommands:

name	type	usage
url build-query[325]	Builtin	Converts record or table into query string applying percent-encoding.
url decode[326]	Builtin	Converts a percent-encoded web safe string to a string.
url encode[327]	Builtin	Converts a string to a percent encoded web safe string.
url join[328]	Builtin	Converts a record to url.
url parse[329]	Builtin	Parses a url.

url build-query

version: 0.90.2

usage:

Converts record or table into query string applying percent-encoding.

Signature

```
> url build-query
```

Input/output types:

input	output
record	string
table	string

[325] /commands/docs/url_build-query.md
[326] /commands/docs/url_decode.md
[327] /commands/docs/url_encode.md
[328] /commands/docs/url_join.md
[329] /commands/docs/url_parse.md

Examples

Outputs a query string representing the contents of this record

```
> { mode:normal userid:31415 } | url build-query
```

Outputs a query string representing the contents of this 1-row table

```
> [[foo bar]; ["1" "2"]] | url build-query
```

Outputs a query string representing the contents of this record

```
> {a:"AT&T", b: "AT T"} | url build-query
```

url decode

version: 0.90.2

usage:

Converts a percent-encoded web safe string to a string.

Signature

```
> url decode ...rest
```

Parameters

- `...rest`: For a data structure input, url decode strings at the given cell paths.

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

Decode a url with escape characters

```
> 'https://example.com/foo%20bar' | url decode
```

Decode multiple urls with escape characters in list

```
> ['https://example.com/foo%20bar' 'https://example.
com/a%3Eb' '%E4%B8%AD%E6%96%87%E5%AD%97/eng/12%2034'
] | url decode
```

url encode

version: 0.90.2

usage:

Converts a string to a percent encoded web safe string.

Signature

```
> url encode ...rest --all
```

Parameters

- `...rest`: For a data structure input, check strings at the given cell paths, and replace with result.
- `--all`: encode all non-alphanumeric chars including `/`, `.`, `:`

Input/output types:

input	output
list<string>	list<string>
record	record
string	string
table	table

Examples

Encode a url with escape characters

```
> 'https://example.com/foo bar' | url encode
```

Encode multiple urls with escape characters in list

```
>  [ ' https://example.com/foo bar '   ' https://example.
com/a>b '   '   /eng/12 34 ' ]   |   url  encode
```

Encode all non alphanumeric chars with all flag

```
>   ' https://example.com/foo bar '   |   url  encode  --
all
```

url join

version: 0.90.2

usage:

Converts a record to url.

Signature

```
> url join
```

Input/output types:

input	output
record	string

Examples

Outputs a url representing the contents of this record

```
>  {          " scheme " :  " http " ,           " username
" :  " " ,            " password " :  " " ,           " host " :
  " www.pixiv.net " ,           " port " :  " " ,             "
path " :  " /member_illust.php " ,           " query " :  "
mode=medium&illust_id=99260204 " ,           " fragment " :
  " " ,           " params " :           {              " mode
" :  " medium " ,              " illust_id " :  " 99260204 "
          }      }  |   url  join
```

Outputs a url representing the contents of this record

```
> {          " scheme " :  " http " ,          " username
" :  " user " ,          " password " :  " pwd " ,          "
host " :  " www.pixiv.net " ,          " port " :  " 1234 "
,          " query " :  " test=a " ,          " fragment " :
  " "      }  |  url  join
```

Outputs a url representing the contents of this record

```
> {          " scheme " :  " http " ,          " host " :
" www.pixiv.net " ,          " port " :  " 1234 " ,
  " path " :  " user " ,          " fragment " :  " frag "
   }  |  url  join
```

url parse

version: 0.90.2

usage:

Parses a url.

Signature

```
> url parse ...rest
```

Parameters

- `...rest`: Optionally operate by cell path.

Input/output types:

input	output
record	record
string	record
table	table

Examples

Parses a url

```
> 'http://user123:pass567@www.example.com:8081/foo/bar?param1=section
' | url parse
```

use

version: 0.90.2

usage:

Use definitions from a module, making them available in your shell.

Signature

```
> use (module) ...rest
```

Parameters

- `module`: Module or module file.
- `...rest`: Which members of the module to import.

Input/output types:

input	output
nothing	nothing

Examples

Define a custom command in a module and call it

```
> module spam { export def foo [] { "foo" } };
use spam foo; foo
```

Define a custom command that participates in the environment in a module and call it

```
> module foo { export def --env bar [] { $env.
FOO_BAR = "BAZ" } }; use foo bar; bar; $env.FOO_
BAR
```

Use a plain module name to import its definitions qualified by the module name

```
> module spam { export def foo [] { "foo" }; export def bar [] { "bar" } }; use spam; (spam foo) + (spam bar)
```

Specify * to use all definitions in a module

```
> module spam { export def foo [] { "foo" }; export def bar [] { "bar" } }; use spam *; (foo) + (bar)
```

To use commands with spaces, like subcommands, surround them with quotes

```
> module spam { export def 'foo bar' [] { "baz" } }; use spam 'foo bar'; foo bar
```

To use multiple definitions from a module, wrap them in a list

```
> module spam { export def foo [] { "foo" }; export def 'foo bar' [] { "baz" } }; use spam ['foo', 'foo bar']; (foo) + (foo bar)
```

Notes

```
See `help std` for the standard library module. See `help modules` to list all available modules. This command is a parser keyword. For details, check: https://www.nushell.sh/book/thinking_in_nu.html
```

values

version: 0.90.2

usage:

Given a record or table, produce a list of its columns' values.

Signature

```
> values
```

Input/output types:

input	output
record	list<any>
table	list<any>

Examples

Get the values from the record (produce a list)

```
> { mode:normal userid:31415 } | values
```

Values are ordered by the column order of the record

```
> { f:250 g:191 c:128 d:1024 e:2000 a:16 b:32 } |
values
```

Get the values from the table (produce a list of lists)

```
> [[name meaning]; [ls list] [mv move] [cd '
change directory']] | values
```

Notes

```
This is a counterpart to `columns`, which produces a list
of columns' names.
```

version

version: 0.90.2

usage:

Display Nu version, and its build configuration.

Signature

```
> version
```

Input/output types:

input	output
nothing	record

Examples

Display Nu version

```
> version
```

view

version: 0.90.2

usage:

Various commands for viewing debug information.

Signature

```
> view
```

Input/output types:

input	output
nothing	string

Notes

> You must use one of the following subcommands. Using this command as-is will only produce this help message.

Subcommands:

name	type	usage
`view files`[330]	Builtin	View the files registered in nushell's EngineState memory.
`view source`[331]	Builtin	View a block, module, or a definition.
`view span`[332]	Builtin	View the contents of a span.

[330] /commands/docs/view_files.md
[331] /commands/docs/view_source.md
[332] /commands/docs/view_span.md

view files

version: 0.90.2

usage:

View the files registered in nushell's EngineState memory.

Signature

```
> view files
```

Input/output types:

input	output
nothing	table<filename: string, start: int, end: int, size: int>

Examples

View the files registered in Nushell's EngineState memory

```
> view files
```

View how Nushell was originally invoked

```
> view files | get 0
```

Notes

```
These are files parsed and loaded at runtime.
```

view source

version: 0.90.2

usage:

View a block, module, or a definition.

Signature

```
> view source (item)
```

Parameters

- `item`: Name or block to view.

Input/output types:

input	output
nothing	string

Examples

View the source of a code block

```
> let abc = { || echo 'hi' }; view source $abc
```

View the source of a custom command

```
> def hi [] { echo 'Hi!' }; view source hi
```

View the source of a custom command, which participates in the caller environment

```
> def --env foo [] { $env.BAR = 'BAZ' };
view source foo
```

View the source of a custom command with flags and arguments

```
> def test [a?:any --b:int ...rest:string] {
echo 'test' }; view source test
```

View the source of a module

```
> module mod-foo { export-env { $env.FOO_ENV = '
BAZ' } }; view source mod-foo
```

View the source of an alias

```
> alias hello = echo hi; view source hello
```

view span

version: 0.90.2

usage:

View the contents of a span.

Signature

```
> view span (start) (end)
```

Parameters

- `start`: Start of the span.
- `end`: End of the span.

Input/output types:

input	output
nothing	string

Examples

View the source of a span. 1 and 2 are just example values. Use the return of debug --raw to get the actual values

```
> some | pipeline | or | variable | debug -
- raw ; view span 1 2
```

Notes

```
This command is meant for debugging purposes. It allows
you to view the contents of nushell spans. One way to
get spans is to pipe something into 'debug --raw'. Then
you can use the Span { start, end } values as the start
and end values for this command.
```

watch

version: 0.90.2

usage:

Watch for file changes and execute Nu code when they happen.

Signature

```
> watch (path) (closure) --debounce-ms --glob --recursive -
-verbose
```

Parameters

- `path`: The path to watch. Can be a file or directory.
- `closure`: Some Nu code to run whenever a file changes. The closure will be passed `operation`, `path`, and `new_path` (for renames only) arguments in that order.
- `--debounce-ms {int}`: Debounce changes for this many milliseconds (default: 100). Adjust if you find that single writes are reported as multiple events
- `--glob {string}`: Only report changes for files that match this glob pattern (default: all files)
- `--recursive {bool}`: Watch all directories under `<path>` recursively. Will be ignored if `<path>` is a file (default: true)
- `--verbose`: Operate in verbose mode (default: false)

Input/output types:

input	output
nothing	table

Examples

Run `cargo test` whenever a Rust file changes

```
> watch . --glob=**/*.rs {|| cargo test }
```

Watch all changes in the current directory

```
> watch . { |op, path, new_path| $"($op) ($path)
) ($new_path)"}
```

Log all changes in a directory

```
> watch /foo/bar { |op, path| $"($op) - ($path)
(char nl)" | save --append changes_in_bar.log }
```

Note: if you are looking to run a command every N units of time, this can be accomplished with a loop and sleep

```
> loop { command; sleep duration }
```

where

version: 0.90.2

usage:

Filter values based on a row condition.

Signature

```
> where (row_condition)
```

Parameters

- `row_condition`: Filter condition.

Input/output types:

input	output
list<any>	list<any>
range	any
table	table

Examples

Filter rows of a table according to a condition

```
> [{a: 1} {a: 2}] | where a > 1
```

Filter items of a list according to a condition

```
> [1 2] | where {|x| $x > 1}
```

List all files in the current directory with sizes greater than 2kb

```
> ls | where size > 2kb
```

List only the files in the current directory

```
> ls | where type == file
```

List all files with names that contain “Car”

```
> ls | where name =~ "Car"
```

List all files that were modified in the last two weeks

```
> ls | where modified >= (date now) - 2wk
```

Find files whose filenames don’t begin with the correct sequential number

```
> ls | where type == file | sort-by name --nat-
ural | enumerate | where {|e| $e.item.name !~
 $'^($e.index + 1)' } | each {|| get item }
```

Find case-insensitively files called “readme”, without an explicit closure

```
> ls | where ($it.name | str downcase) =~
readme
```

same as above but with regex only

```
> ls | where name =~ '(?i)readme'
```

Notes

```
This command works similar to 'filter' but allows extra
shorthands for working with tables, known as "row conditions".
 On the other hand, reading the condition from a variable
is not supported.
```

which

version: 0.90.2

usage:

Finds a program file, alias or custom command.

Signature

```
> which (application) ...rest --all
```

Parameters

- `application`: Application.
- `...rest`: Additional applications.
- `--all`: list all executables

Input/output types:

input	output
nothing	table

Examples

Find if the 'myapp' application is available

```
> which myapp
```

while

version: 0.90.2

usage:

Conditionally run a block in a loop.

Signature

```
> while (cond) (block)
```

Parameters

- `cond`: Condition to check.
- `block`: Block to loop if check succeeds.

Input/output types:

input	output
nothing	nothing

Examples

Loop while a condition is true

```
> mut x = 0; while $x < 10 { $x = $x + 1 }
```

whoami

version: 0.90.2

usage:

Get the current username using uutils/coreutils whoami.

Signature

```
> whoami
```

Input/output types:

input	output
nothing	string

Examples

Get the current username

```
> whoami
```

window

version: 0.90.2

usage:

Creates a sliding window of `window_size` that slide by n rows/elements across input.

Signature

```
> window (window_size) --stride --remainder
```

Parameters

- `window_size`: The size of each window.
- `--stride {int}`: the number of rows to slide over between windows
- `--remainder`: yield last chunks even if they have fewer elements than size

Input/output types:

input	output
list<any>	list<list<any>>

Examples

A sliding window of two elements

```
> [1 2 3 4] | window 2
```

A sliding window of two elements, with a stride of 3

```
> [1, 2, 3, 4, 5, 6, 7, 8] | window 2 --stride
3
```

A sliding window of equal stride that includes remainder. Equivalent to chunking

```
> [1, 2, 3, 4, 5] | window 3 --stride 3 --re-
mainder
```

with-env

version: 0.90.2

usage:

Runs a block with an environment variable set.

Signature

```
> with-env (variable) (block)
```

Parameters

- `variable`: The environment variable to temporarily set.
- `block`: The block to run once the variable is set.

Input/output types:

input	output
any	any

Examples

Set the MYENV environment variable

```
> with-env [MYENV "my env value"] { $env.MYENV
}
```

Set by primitive value list

```
> with-env [X Y W Z] { $env.X }
```

Set by single row table

```
> with-env [[X W]; [Y Z]] { $env.W }
```

Set by key-value record

```
> with-env {X: "Y", W: "Z"} { [$env.X $env
.W] }
```

wrap

version: 0.90.2

usage:

Wrap the value into a column.

Signature

```
> wrap (name)
```

Parameters

- `name`: The name of the column.

Input/output types:

input	output
any	record
list<any>	table
range	table

Examples

Wrap a list into a table with a given column name

```
> [1 2 3] | wrap num
```

Wrap a range into a table with a given column name

```
> 1..3 | wrap num
```

zip

version: 0.90.2

usage:

Combine a stream with the input.

Signature

```
> zip (other)
```

Parameters

- `other`: The other input.

Input/output types:

input	output
list<any>	list<list<any>>
range	list<list<any>>

Examples

Zip two lists

```
> [ 1 2 ]  |  zip  [ 3 4 ]
```

Zip two ranges

```
> 1 ..3  |  zip  4..6
```

Rename .ogg files to match an existing list of filenames

```
> glob  * .ogg  |  zip  [ ' bang.ogg ' ,  ' fanfare.ogg '
, ' laser.ogg ' ]  |  each  { || mv  $ in .0  $ in .1  }
```