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.

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`)

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

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[9] or the Visual Studio Build Tools[10])
  - 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://visualstudio.microsoft.com/vs/community/
[10] https://visualstudio.microsoft.com/downloads/#build-tools-for-visual-studio-2022

Installing Rust

If we don't already have Rust on our system, the best way to install it is via rustup[11]. Rustup is a way of managing Rust installations, including managing using different Rust versions.

Nu currently requires the **latest stable (1.60 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 we are ready, we 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 1.60 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[12], you will need to install "openssl" and "cmake" using:

Build using crates.io[13]

Nu releases are published as source to the popular Rust package registry crates.io[14]. This makes it easy to build+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 so we can run it.

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

Once installed, we can run Nu using the `nu` command:

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

Building from the GitHub repository

We can also build our own Nu from the latest source on GitHub. This gives us 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:

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.

Setting the login shell (*nix)

!!! Nu is still in development, and may not be stable for everyday use. !!!

To set the login shell you can use the `chsh`[15] 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`.

Setting the default shell (Windows Terminal)

If you are using Windows Terminal[16] you can set `nu` as your default shell by adding:

to `"profiles"` in your Terminal Settings (JSON-file). The last thing to do is to change the `"defaultProfile"` to:

Now, `nu` should load on startup of the Windows Terminal.

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` is that instead of a block of text coming back, you get a structured table.

[15] https://linux.die.net/man/1/chsh

[16] https://github.com/microsoft/terminal

The table is more than just showing 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` 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`. Instead, we used the `sort-by` command that Nu provides to do the sorting of the output of the `ls` command. To see the biggest files on top, we also used `reverse`.

Nu provides many commands that can work on tables. For example, we could filter the contents of the `ls` table so that it 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` 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` command earlier, we can also work with the table that the `ps` command gives back to us:

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

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

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

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

This is a bit different than the tables we saw before. The `sys` 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` 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) 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` command (the `^` tells Nu to not use the built-in `echo` 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` (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` command. To see all commands, run `help commands`. 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` is a command for viewing the contents of a path. Nu will return the contents as a table that we can use.

The `ls` 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"

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 addition to `*`, there is also the `?` pattern which will match a single character. For example, you can match the word "port" by using the pattern `p???`.

Changing the current directory

To change from the current directory to a new one, we use the `cd` 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` is omitted and a path by itself is given:

Note: changing the directory with `cd` 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[17].

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` command:

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

We can remove an item with the `rm` command:

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

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

[17] `./environment.md`

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 and because of this 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:

```
> 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` command will grow the source that is compiled, but the `save` from the earlier line won't have had a chance to run. Nushell 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` 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 would require the evaluator to run and evaluate the string, but unfortunately Nushell needs this information at compile-time.

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 `each` to `par-each` 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 the `-n` flag on `each`:

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

You can also use the `reduce` 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 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` 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`, `load-env`, as convenient ways of doing batches of updates to the environment.

* - there is one exception here, where `def-env` 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.

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 lists all the built-in commands with brief descriptions. Note that you can also access this info from within Nushell using the `help` command.

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` command returns the type of a data value:

```
> 42 | describe
```

Types at a glance

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

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` command

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

Decimals

Decimal numbers are numbers with some fractional component. Examples include 1.5, 2.0, and 15.333. You can cast a string into an Decimal with the `into decimal` command

```
> "1.2" | into decimal
```

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[1] for details.

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

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:
  - `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
- `zb`: zettabyte
- `kib`: kibibytes (aka 1024 bytes)
- `mib`: mebibytes
- `gib`: gibibytes
- `tib`: tebibytes
- `pib`: pebibytes
- `eib`: exbibyte
- `zib`: zebibyte

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 values from start to finish. They take the form <start>..<end>. For example, the range `1..3` means the numbers 1, 2, and 3.

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`, 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 will start counting with `0`. For example, the range `..2` is the numbers 0, 1, and 2.

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 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-dowr

::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`, 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:

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

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` 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` command:

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

  0   b
  1   c
  2   d
```

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:

```
> [{langs:[Rust JS Python], releases:60}].0

  langs      [list 3 items]
  releases   60

> [{langs:[Rust JS Python], releases:60}].0.langs.2
Python
```

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` 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` 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
```

There are numerous other commands for selecting and reducing the data in tables, records and lists.

Blocks

Blocks represent a block of code in Nu. For example, in the command `each { |it| print $it }` the block is the portion contained in curly braces, `{ |it| print $it }`. Block parameters are specified between a pair of pipe symbols (for example, `|it|`) if necessary. You can also use `$in` in most blocks instead of providing a parameter: `each { print $in }`

Blocks 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` blocks, but not required; `each { |x| print $x }` works the same way as `each { |it| print $it }`.

Null

Finally, there is `null` (also known as `$nothing`) 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` or `cd`) 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
Error: nu::shell::column_not_found

   × Cannot find column
    [entry #15:1:1]
 1   [{a:1 b:2} {b:1}].1.a
    ·
    ·                       cannot find column
    ·                value originates here
```

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`, `ps`, `date`, and `sys` 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` 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:

```
> open editors/vscode/package.json

 name                 lark
 description          Lark support for VS Code
 author               Lark developers
```

```
 license            MIT
 version            1.0.0
 repository         [row type url]
 publisher          vscode
 categories         [table 0 rows]
 keywords           [table 1 rows]
 engines            [row vscode]
 activationEvents   [table 1 rows]
 main               ./out/extension
 contributes        [row configuration grammars languages]
 scripts            [row compile postinstall test vscode:
prepublish watch]
 devDependencies    [row @types/mocha @types/node tslint
typescript vscode vscode-languageclient]
```

In a similar way to `ls`, 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` 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
- ics
- ini
- json
- nuon[2]

[2] `#nuon`

- ods
- SQLite databases[3]
- ssv
- toml
- tsv
- url
- vcf
- xlsx / xls
- xml
- yaml / yml

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.

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[4]):

```
{
  menus: [
    # Configuration for default nushell menus
    # Note the lack of souce parameter
    {
      name: completion_menu
```

[3] #sqlite

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

```
        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:

```
> 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 table. 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` command. `split`, as the name implies, gives us a way to split a delimited string. We will use `split`'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` 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` 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
```

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`
- `lines`
- `size`

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.
[[package]]
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` command using the `toml` subcommand:

```
> open Cargo.lock | from toml

 metadata   [row 107 columns]
 package    [table 130 rows]
```

The `from` 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 command can take an optional `--raw` flag:

```
> 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`, no matter what their file extension is. You can open a whole database:

```
> open foo.db
```

Or `get` 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 `fetch` command. This will fetch the contents of the URL from the internet and return it:

```
> fetch https://blog.rust-lang.org/feed.xml

 feed   {record 2 fields}
```

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.

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` 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` 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[5]

[5] 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` 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` is an input and not an output. Nu adds this output for us automatically using another command called `table`. The `table` 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.

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[6] the result to external **^ls**

```
> 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`:

```
> 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`)

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

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.

[6] https://www.nushell.sh/book/pipelines.html

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
- `\uXXXX` - a unicode character (replace XXXX with the number of the unicode character)

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` command for this purpose, which provides additional flags and options.

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!)
```

Splitting strings

The `split row` 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` 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` command will split a string into a list of characters.

```
> 'aeiou' | split chars

  0   a
  1   e
  2   i
  3   o
  4   u
```

The `str` command

Many string functions are subcommands of the `str` 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`:

```
> "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` command. By default, the `str trim` 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
```

String padding

With the `str lpad` and `str rpad` commands you can add padding to a string. Padding adds characters to string until it's a certain length. For example:

```
> '1234' | str lpad -l 10 -c '0'
0000001234
> '1234' | str rpad -l 10 -c '0' | str length
10
```

Reversing strings

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

```
> 'Nushell' | str reverse
llehsuN
> ['Nushell' 'is' 'cool'] | str reverse

  0   llehsuN
  1   si
  2   looc
```

String parsing

With the `parse` 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?\
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`, `from tsv` or `from ssv`:

```
> "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

> "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`. e.g. `123 | into string`

2. Using string interpolation. e.g. `$'(123)'`

3. Using `build-string`[7]. e.g. `build-string (123)`

From string

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

[7] `commands/build-string.md`

Coloring strings

You can color strings with the `ansi` 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`)

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` and `insert` 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
```

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

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

In addition to `insert` and `update`, we also have `prepend` and `append`. 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)
$colors # [red yellow green purple]
```

Iterating over lists

To iterate over the items in a list, use the `each` 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 normally the current list item, but the `--numbered` (`-n`) flag can change it to have `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 | each -n { |it| $"($it.index + 1) - ($it.item)"
}
# Outputs "1 - Mark", "2 - Tami", etc.
```

The `where` 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 `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` 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, add the `--numbered` (`-n`) flag. 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 })" # total = 96

$scores | reduce -n { |it, acc| $acc.item + $it.index *
$it.item } # 3 + 1*8 + 2*4 = 19
```

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

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

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

```
# Do any color names end with "e"?
$colors | any ($it | str ends-with "e") # true

# Is the length of any color name less than 3?
$colors | any ($it | str length) < 3 # false

# Are any scores greater than 7?
$scores | any $it > 7 # true

# Are any scores odd?
$scores | any $it mod 2 == 1 # true
```

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

```
# Do all color names end with "e"?
$colors | all ($it | str ends-with "e") # false

# Is the length of all color names greater than or equal
to 3?
$colors | all ($it | str length) >= 3 # true

# Are all scores greater than 7?
$scores | all $it > 7 # false

# Are all scores even?
$scores | all $it mod 2 == 0 # false
```

Converting the list

The `flatten` 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:

```
[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` 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 | date format '%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
```

Sorting the data

We can sort a table by calling the `sort-by` 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:

```
> 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` 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` rows that you don't want. Let's skip the first two of the 5 rows we returned above:

```
> 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` 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` 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.

This might look like the `select` 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` - creates a new table which includes only the columns specified
- `get` - 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` 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 with identical column names using `append`:

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

 #   a   b

 0   1   2
 1   3   4
```

Merging Tables

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

```
> 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 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 command, we can also use the update 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:

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

 edition   2021
```

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

Moving columns

You can use `move` to move columns in the table. For example, if we wanted to move the "name" column from `ls` 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` columns in a table by passing it through the rename command. If we wanted to run `ls` 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

...
```

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 subrountines 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.

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. :::

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:

```
 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`, 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`:

```
> 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:

```
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 0.65.0):

- `any`
- `block`
- `cell-path`
- `duration`
- `path`

- `expr`
- `filesize`
- `glob`
- `int`
- `math`
- `number`
- `operator`
- `range`
- `cond`
- `bool`
- `signature`
- `string`
- `variable`
- `record`
- `list`
- `table`
- `error`

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[2]:

```
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[3] 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:

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

greet
```

[2] #parameter-types
[3] #optional-positional-parameters

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[4] 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:

[4] `#parameters-with-a-default-value`

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

Or:

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

Or even leave the flag off altogether:

```
> greet world
```

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
```

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] {
  "hello all:"
  for $n in $name {
    $n
  }
}

greet earth mars jupiter venus
```

::: tip Each line of a command has its resulting value printed out when run, as long as it isn't `null`. Hence, `"hello all:"` above will be printed out despite not being the return value. To prevent this, you can place `null` (or the `ignore` command) at the end of the pipeline, like so: `"hello all:" | null`. Also note that most file system commands, such as `save` or `cd`, always output `null`. :::

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] {
  $"hello to our VIP ($vip)"
  "and hello to everybody else:"
```

```
  for $n in $name {
    $n
  }
}

#     $vip          $name
#     ---- ------------------------
greet moon earth mars jupiter venus
```

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:

```
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 appear as descriptions inside the `help` of the command.

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` into a command that we've written:

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

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

```
> 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, textual replacement. 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`.

How to write an alias with Pipes

If you want to add a pipe to your alias you must enclose it with parentheses which are a pair of round brackets ( ) used to mark off your set of commands with pipes.

```
alias lsname = (ls | get name)
```

Here is an alias with more than one pipe:

```
alias lt = (ls | sort-by modified -r | sort-by type)
```

List all loaded aliases

Your useable aliases can be seen in `$nu.scope.aliases`.

Persisting

To make your alias persistent it must be added to your *config.nu* file.

For more details about how to persist aliases so that they're visible when you start up Nushell, see the configuration chapter.

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[5]. 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[6].

[5]https://cheatography.com/davechild/cheat-sheets/regular-expressions/
[6]https://docs.rs/regex/latest/regex/struct.Regex.html#method.is_match

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` command's `--insensitive` flag:

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

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

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

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.

If we create a variable, we can print its contents by using `$` to refer to it:

```
> let my_value = 4
> $my_value
4
```

Variables in Nushell are immutable, that means that you cannot change its value after declaration. They can be shadowed in nested block, that results in:

```
> let my_value = 4
> do { let my_value = 5; $my_value }
5
> $my_value
4
```

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:

```
> $my_value.name.0
testuser
```

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` 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 list of filenames larger than ten kilobytes, we can use an subexpression to run a pipeline and assign the result to a variable:

```
> let names_of_big_files = (ls | where size > 10kb)
> $names_of_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:

```
> 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` where the entry is at least ten kilobytes we can write:

```
> ls | where size > 10kb
```

The `where size > 10kb` is a command with two parts: the command name `where` 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 short-hand syntax to work, the column name must appear on the left-hand side of the operation (like `size` in `size > 10kb`).

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`:

```
> 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.

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
```

Shebangs (#!)

On Linux and macOS you can optionally use a shebang[7] 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!
```

Modules

Similar to many other programming languages, Nushell also has modules that let you import custom commands into a current scope. However, since Nushell is also a shell, modules allow you to import environment variables which can be used to conveniently activate/deactivate various environments.

Basics

A simple module can be defined like this:

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

     export def hi [where: string] {
```

[7] https://en.wikipedia.org/wiki/Shebang_(Unix)

```
            $"hi ($where)!"
        }
}
```

or in a file named the same as the module you want to create:

```
# greetings.nu

export def hello [name: string] {
    $"hello ($name)!"
}

export def hi [where: string] {
    $"hi ($where)!"
}
```

We defined `hello` and `hi` custom commands inside a `greetings` module.

The `export` keyword makes it possible to later import the commands from the module.

Similar to `def`, it is also possible to mark `def-env` with the `export` keyword (you can learn more about `def-env` in the Environment chapter).

Using modules

By itself, the module does not do anything. To use what the module exports, we need to `use` it.

```
> use greetings

> greetings hello "world"
hello world!

> greetings hi "there"
hi there!
```

The `hello` and `hi` commands are now available with the `greetings` prefix.

Importing symbols

In general, anything after the `use` keyword forms an **import pattern** which controls how the symbols are imported. The import pattern can be one of the following:

`use greetings`

Imports all symbols with the module name as a prefix (we saw this in the previous example).

`use greetings hello`

The `hello` symbol will be imported directly without any prefix.

`use greetings [ hello, hi ]`

Imports multiple symbols directly without any prefix.

`use greetings *`

You can also use the module name and the `*` glob to import all names directly without any prefix.

Module Files

Nushell lets you implicitly treat a source file as a module. Let's start by saving the body of the module definition into a file:

```
# greetings.nu

export def hello [name: string] {
    $"hello ($name)!"
}

export def hi [where: string] {
    $"hi ($where)!"
}
```

Now, you can call `use` directly on the file:

```
> use greetings.nu

> greetings hello "world"
hello world!

> greetings hi "there"
hi there!
```

Nushell automatically infers the module's name from the stem of the file ("greetings" without the ".nu" extension). You can use any import patterns as described above with the file name instead of the module name.

Local Custom Commands

Any custom commands defined in a module without the `export` keyword will work only in the module's scope:

```
# greetings.nu

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]
{
    $"($greeting) ($subject)!"
}
```

Then, in Nushell we import all definitions from the "greetings.nu":

```
> use greetings.nu *

> hello "world"
hello world!

> hi "there"
hi there!

> greetings-helper "foo" "bar"  # fails because 'greetings-
helper' is not exported
```

Environment Variables

So far we used modules just to import custom commands. However, modules can also define an environment using `export-env`:

```
# greetings.nu

export-env {
    let-env MYNAME = "Arthur, King of the Britons"
}

export def hello [] {
    $"hello ($env.MYNAME)"
}
```

`use` will run the code inside the `export-env` 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 might wonder why we can't just define `let-env` at the top of the module. The reason is that the `export-env {...}` block keeps its scope separate from the rest of the module which makes it more organized. You can put a complex code defining your environment without polluting the namespace of the module, for example:

```
export-env {
    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. :::

If you also want to keep your variables in separate modules and export its environment, you could try to `export use` it:

```
# purpose.nu
export use greetings.nu
export-env {let-env MYPURPOSE = "to build an empire."}

export def greeting_purpose [] {
    $"Hello ($env.MYNAME). My purpose is ($env.MYPURPOSE)
"
}
```

and then use it

```
> use purpose.nu
> purpose greeeting_purpose
```

However, this won't work, because the module would not export its environment unless defined manually, like so:

```
# purpose.nu

# preserves its environment
export-env {
    use greetings.nu
    let-env MYPURPOSE = "to build an empire."
}

export def greeting_purpose [] {
    $"Hello ($env.MYNAME). My purpose is ($env.MYPURPOSE)
"
}
```

Now, everything is exported properly

```
> use purpose.nu
> purpose greeting_purpose
Hello Arthur, King of the Britons. My purpose is to build
an empire.
```

Exporting symbols

Apart from `def` and `def-env`, you can also export `aliases` and `externs`, giving you a way to only use these features when you need. Exporting externs also gives you the ability to hide custom completion commands in a module, so they don't have to be part of the global namespace.

Here's the full list of ways you can export:

- `export def` - export a custom command
- `export def-env` - export a custom environment command
- `export alias` - export an alias
- `export extern` - export a known external definition
- `export use` - use definitions from a module and export them from this module

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` command:

```
> def foo [] { "foo" }

> foo
foo

> hide foo

> foo  # error! command not found!
```

The `hide` command also accepts import patterns, just like `use`. 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

Hiding Environment Variables

Environment variables can be hidden with `hide-env`:

```
> let-env FOO = "FOO"

> $env.FOO
FOO

> hide-env FOO

> $env.FOO  # error! environment variable not found!
```

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[8] first as overlays build on top of modules.

[8] `modules.md`

Basics

First, Nushell comes with one default overlay called `zero`. You can inspect which overlays are active with the `overlay list` 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 `overlay use spam`, assume `spam` is referring to this module.

To create the overlay, call `overlay use`:

```
> 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`[10]:

```
(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:

```
(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`[11] 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:

[10] commands/overlay_remove.md
[11] commands/overlay_remove.md

```
(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

:: 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 new 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` command:

```
(zero)> overlay use spam

(spam)> overlay new scratchpad

(scratchpad)> def eggs [] { "eggs" }
```

:::

Prefixed Overlays

The `overlay use` 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:

```
(zero)> module spam {
    export def foo [] { "foo" }
    export-env { let-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` 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 contains a signature section, take `str distance` as example, the signature is like this:

```
Signatures(Cell paths are supported):
  <string> | str distance <string> -> <int>
```

The first type name before `|` describes the type of input pipeline. The command name is followed by the required argument type(s) for the command. The output type is `int` and given after `->`.

`(Cell paths are supported)` indicates that you can provide cell paths for `str distance` to apply an operation at the given cell path(s) in a nested structure or table, and replace the column or field with the result, like: `ls | str distance 'nushell' 'name'`

Here is another one example, `str join`:

```
Signatures:
  list<string> | str join <string?> -> <string>
```

It says that `str join` command expect input pipeline is a list of string, and take optional `string` type argument, finally the output type is `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`).

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+runs two Nushell script files at launch time: First, `env.nu`, then `config.nu`. Paths to these files can be found by calling `echo $nu.env-path` and `echo $nu.config-path`. `env.nu` is meant to define the environment variables which are then available within `config.nu`. `config.nu` can be used to add definitions, aliases, and more to the global namespace.

(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.

Configuring `$env.config`

Nushell's main settings are kept in the `config` environment variable as a record. This record can be created using:

```
let-env config = {
  ...
}
```

[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

You can also shadow `$env.config` and update it:

```
let-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 `let-env` calls inside the `env.nu` file. For example:

```
let-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` and `config env`) 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`
4. If 1~3 not found, then launch `notepad` for windows, otherwise run `nano`

Color Config section

You can learn more about setting up colors and theming in the associated chapter.

Configuring Nu as a login shell

To use Nu as a login shell, you'll need to configure the `$env` variable. With this, you'll have enough support to run external commands as a login shell.

You can build the full set of environment variables by running Nu inside of another shell, like Bash. Once you're in Nu, you can run a command like this:

```
> env | each { |it| echo $"let-env ($it.name) = '($it.raw)
'" } | str join (char nl)
```

This will print out `let-env` lines, one for each environment variable along with its setting.

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` command to open files on Mac. As Nushell has its own `open` 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` and create an alias for the system's `open` in your `config.nu` file like this:

```
def nuopen [arg, --raw (-r)] { if $raw { open -r $arg }
else { open $arg } }
alias open = ^open
```

PATH configuration

In Nushell, the PATH environment variable[5] (Path on Windows) is a list of paths. To append a new path to it, you can use `let-env` and `append` in `env.nu`:

```
let-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` 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`/`prepend` directly.

[5] https://en.wikipedia.org/wiki/PATH_(variable)

Homebrew

Homebrew[6] is a popular package manager that often requires PATH configuration. To add it to your Nushell PATH:

```
# macOS ARM64 (Apple Silicon)
let-env PATH = ($env.PATH | split row (char esep) | prepend
'/opt/homebrew/bin')

# Linux
let-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 using the `env` command:

```
   #           name                     type
 value                   raw

  16    DISPLAY              string                  :0
                :0
  17    EDITOR               string                  nvim
                nvim
  28    LANG                 string                  en_US.
UTF-8           en_US.UTF-8
  35    PATH                 list<unknown>           [list
16 items]       /path1:/path2:/...
  36    PROMPT_COMMAND       block                   <Block
197>
```

In Nushell, environment variables can be any value and have any type (see the `type` column). The actual value of the env. variable used

[6] https://brew.sh/

within Nushell is under the `value` column. You can query the value directly using the `$env` variable, for example, `$env.PATH | length`. The last `raw` column shows the actual value that will be sent to external applications (see Environment variable conversions[7] for details).

The environment is initially created from the Nu configuration file and from the environment that Nu is run inside of.

Setting environment variables

There are several ways to set an environment variable:

let-env

Using the `let-env` command is the most straightforward method

```
> let-env FOO = 'BAR'
```

'let-env' is similar to the **export** command in Bash.

So, if you want to extend the Windows `Path` variable, for example, you could do that as follows.

```
let-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` command.

load-env

If you have more than one environment variable you'd like to set, you can use `load-env` 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.

[7] #environment-variable-conversions

Calling a command defined with `def-env`

See Defining environment from custom commands for details.

Using module's exports

See Modules[8] 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
```

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:

```
> let-env FOO = "BAR"
> do {
    let-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` command. In Nushell, calling `cd` is equivalent to setting the `PWD` environment variable. Therefore, it follows the same rules as other environment variables (for example, scoping).

[8] `modules.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` to do the same thing more explicitly:

```
> with-env { FOO: BAR } { $env.FOO }
BAR
```

The `with-env` 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
let-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` instead of `def` (it applies also to `export def`, see Modules[9]) will preserve the environment on the caller's side:

```
> def-env foo [] {
    let-env FOO = 'BAR'
}

> foo
```

[9] `modules.md`

```
> $env.FOO
BAR
```

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[10] 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:

```
let-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
```

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

```
  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. You can also run this step manually by `do $env.ENV_CONVERSIONS.FOO.to_string [a b c]`

If we look back at the `env` command, the `raw` column shows the value translated by `ENV_CONVERSIONS.<name>.to_string` and the `value` column shows the value used in Nushell (the result of `ENV_CONVERSIONS.<name>.from_string` in the case of `FOO`). If the value is not a string and does not have `to_string` conversion, it is not passed to an external (see the `raw` column of `PROMPT_COMMAND`). One exception is `PATH` (`Path` on Windows): by default, it converts the string to a list on startup and from a list to a string when running externals if no manual conversions are specified.

*(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`[11]:

[11] `commands/hide_env.html`

```
> let-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:

```
> let-env FOO = 'BAR'
> do {
    hide-env FOO
    # $env.FOO does not exist
  }
> $env.FOO
BAR
```

You can check Modules[12] for more details about hiding.

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`.

Without the pipeline, Nushell will not do any redirection, allowing it to print directly to the screen.

[12] `modules.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 -i { ... }`. For example, if we wanted to call the external above and redirect its stderr, we would write:

```
> do -i { external }
```

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 -i { external }
> $env.LAST_EXIT_CODE
```

The second uses a command called `complete`.

Using the `complete` command

The `complete` 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` does with the streams, including the redirected stderr:

```
> do -i { cat unknown.txt } | complete

 stdout

 stderr      cat: unknown.txt: No such file or directory

 exit_code   1
```

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` command. The `decode` 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` or `date` 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` command).

Nu internal command:

```
> ls
```

Escape to external command:

```
> ^ls
```

Windows note

When running an external command on Windows, nushell used to[13] use Cmd.exe[14] 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[15]

repo[16]

oh-my-posh

site[17]

repo[18]

If you like oh-my-posh[19], 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[20].
2. Download and install a nerd font[21].
3. Set the PROMPT_COMMAND in ~/.config/nushell/config.nu(or the path output by `$nu.config-path`), change `M365Princess.omp.json` to whatever you like Themes demo[22].

[13] https://www.nushell.sh/blog/2022-08-16-nushell-0_67.html#windows-cmd-exe-changes-rgwood

[14] https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/cmd

[15] https://www.nerdfonts.com

[16] https://github.com/ryanoasis/nerd-fonts

[17] https://ohmyposh.dev/

[18] https://github.com/JanDeDobbeleer/oh-my-posh

[19] https://ohmyposh.dev/

[20] https://ohmyposh.dev/docs/installation/linux

[21] https://github.com/ryanoasis/nerd-fonts

[22] https://ohmyposh.dev/docs/themes

```
> let-env PROMPT_COMMAND = { oh-my-posh --config ~/.poshthemes/
M365Princess.omp.json }
```

For MacOS users:

1. You can install oh-my-posh by `brew`, just following the guide here[23]
2. Download and install a nerd font[24].
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)

let-env PROMPT_COMMAND = { oh-my-posh prompt print primary
--config $'($posh_theme)/zash.omp.json' }
# Optional
let-env PROMPT_INDICATOR = $"(ansi y)$> (ansi reset)"
```

Starship

site[25]

repo[26]

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.

[23] https://ohmyposh.dev/docs/macos
[24] https://github.com/ryanoasis/nerd-fonts
[25] https://starship.rs/
[26] https://github.com/starship/starship

Here's an example config section for Starship:

```
let-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
let-env PROMPT_COMMAND = { create_left_prompt }
let-env PROMPT_COMMAND_RIGHT = ""

# The prompt indicators are environmental variables that
represent
# the state of the prompt
let-env PROMPT_INDICATOR = ""
let-env PROMPT_INDICATOR_VI_INSERT = ": "
let-env PROMPT_INDICATOR_VI_NORMAL = " "
let-env PROMPT_MULTILINE_INDICATOR = "::: "
```

Now restart Nu.

```
nushell on  main is  v0.60.0 via  v1.59.0
```

You can learn more about configuring Starship in the official starship configuration documentation[27].

An alternate way to enable Starship is described in the Starship Quick Install[28] instructions.

Purs

repo[29]

[27] https://github.com/starship/starship#step-2-setup-your-shell-to-use-starship
[28] https://starship.rs/#nushell
[29] https://github.com/xcambar/purs

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 in one, allowing you to quickly jump between working directories and more.

To get started, let's enter a directory:

```
/home/jonathant/Source/nushell(main)> enter ../book
/home/jonathant/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` 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` command to list the current directories we have active:

```
/home/jonathan/Source/book(main)> shells

 #  active      name                  path

 0  false    filesystem   /home/jt/Source/nushell
 1  true     filesystem   /home/jt/Source/book
 2  false    filesystem   /home/jt/Source/music
```

The `shells` command shows us there are three shells currently active: our original "nushell" source directory and now this new "book" directory.

We can jump between these shells with the `n`, `p` and `g` shortcuts, short for "next", "previous" and "goto":

```
/home/jonathant/Source/book(main)> n
/home/jonathant/Source/nushell(main)> p
/home/jonathant/Source/book(main)> g 2
/home/jonathant/Source/music(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.

Exiting the shell

You can leave a shell you have `enter`ed using the `exit` command. If this is the last open shell, Nu will quit.

You can always quit Nu, even if multiple shells are active by passing the `--now` flag to the exit command. Like so: `exit --now`

Reedline, Nu's line editor

Nushell's line editor Reedline[30] 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:

```
  let $config = {
    ...
    edit_mode: emacs
    ...
  }
```

[30] https://github.com/nushell/reedline

Default keybindings Each edit mode comes with the usual keybinding for vi and emacs text editing.

Emacs and Vi Insert keybindings

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:

```
let $config = {
  ...
  max_history_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 | date format '%m/%d/%Y %r')
    ] | str join)

    $time_segment
}

let-env PROMPT_COMMAND = { create_left_prompt }
let-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.

```
let-env PROMPT_INDICATOR = " "
let-env PROMPT_INDICATOR_VI_INSERT = ": "
let-env PROMPT_INDICATOR_VI_NORMAL = " "
let-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.

```
let $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
  - control | alt
  - control | alt | shift
- 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

```
let $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

```
  let $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.

```
let $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

```
let $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

```
let $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` 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 `ctr+q`. 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.

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

```
  let $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:

```
let $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+x`) 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:

```
  let $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: 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

```
let $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`. The `source` 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 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

```
let $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, commands outside of Nushell means that Nushell can't help with finding errors in the call, or completions, or syntax highlighting.

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 commands do, letting you describe flags, short flags, positional parameters, types, and more.

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 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`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 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 the 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.

Custom completion and `extern`

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"
}
    ]
}
```

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 block which will be evaluated if no Nushell completions were found. You can configure the block to run an external completer, such as carapace[31].

This example should enable carapace external completions:

```
# config.nu
let carapace_completer = {|spans|
    carapace $spans.0 nushell $spans | from json
}

# The default config record. This is where much of your
global configuration is setup.
let-env config = {
    # ... your config
    completions: {
        external: {
            enable: true
            max_results: 100
            completer: $carapace_completer
        }
    }
```

[31] https://github.com/rsteube/carapace-bin

```
}
```

Multiple completers can be defined as such:

```
let external_completer = {|spans|
  {
    $spans.0: { default_completer $spans | from json }
# default
    ls: { ls_completer $spans | from json }
    git: { git_completer $spans | from json }
  } | get $spans.0 | each {|it| do $it}
}
```

> *When the block returns unparsable json (e.g. an empty string) it defaults to file completion.*

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)
4. prompt
5. LS_COLORS

Table borders

Table borders are controlled by the `table_mode` setting in `config.nu`. Here is an example:

```
> let $config = {
    table_mode: rounded
```

```
}
```

Here are the current options for `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.

```
> let $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`.

```
> let-env PROMPT_COMMAND = { build-string (date now | date
format '%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.

```
> let-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[32].

`LS_COLORS` colors for the `ls` 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`. 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[33], 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[34] 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[35].

`let-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.

[32] https://github.com/nushell/nu_scripts/blob/main/prompt/oh-my.nu
[33] https://github.com/trapd00r/LS_COLORS
[34] https://www.linuxhowto.net/how-to-set-colors-for-ls-command/
[35] https://github.com/sharkdp/vivid

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, zb, zib, 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 applied light theme easily.

```
# in $nu.config-file
let-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-file
let-env config = {
  ...
  color_config: $light_theme   # if you want a light theme,
replace `$dark_theme` to `$light_theme`
  ...
}
```

Hooks

Hooks allow you to run a code snippet at some predefined situations. They are only available in the interactive mode (REPL[36]), 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).

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:

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

[36] https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

4. After user typed something and pressed "Enter": Check for `pre_execution` hooks and run them
5. Parse and evaluate user input
6. If `display_output` is defined, use it to print command output
7. Return to 1.

Basic Hooks

To enable hooks, define them in your config:

```
let-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 filesytem. 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:

```
let-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:

```
let-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:

```
> let-env config = ($env.config | upsert hooks {
    pre_prompt: { let-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:

```
let-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` 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**:

```
let-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

```
> let-env config = ($env.config | upsert hooks {
    pre_prompt: 'let-env SPAM = "eggs"'
})

> $env.SPAM
eggs
```

This feature can be used, for example, to conditionally bring in definitions based on the current directory:

```
let-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:

```
let-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:

```
let-env config = ($env.config | upsert hooks.env_change.
PWD {|config|
    let val = ($config | get -i hooks.env_change.PWD)

    if $val == $nothing {
        $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

```
let-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:

```
let-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 automaticaly reloads when the file changes. Instead of the `save` command, you would normally customize this to send the HTML output to a desired window.

Background task in 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 example:

1. using a third-party task management tools, like pueue[37]
2. using a terminal multiplexer, like tmux[38] or zellij[39]

Using nu with pueue

Borrows the power of pueue[40], 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)

Unlike terminal multiplexer, you don't need to attach to multiple tmux sessions, and get task status easily.

Here we provide a nushell module[41] to work with pueue easiler.

Here is a setup example to make nushell "support" background task:

1. install pueue
2. run `pueued` with default config, you can refer to start-the-daemon page[42] for more information.
3. put the job.nu[43] file under `$env.NU_LIB_DIRS`.
4. add a line to the `$nu.config-path` file: `use job.nu`
5. restart nu.

Then you will get some commands to schedule background tasks. (e.g: `job spawn`, `job status`, `job log`)

Cons note: It spawned a fresh nushell to execute the given command, so it doesn't inherit current scope's variables, custom commands, alias definition, except env variables which can convert value to string. Therefore, if you want to use custom commands or variables, you have to `use` or `define` them within the given block.

[37] https://github.com/Nukesor/pueue
[38] https://github.com/tmux/tmux/wiki
[39] https://zellij.dev/
[40] https://github.com/Nukesor/pueue
[41] https://github.com/nushell/nu_scripts/tree/main/background_task
[42] https://github.com/Nukesor/pueue/wiki/Get-started#start-the-daemon
[43] https://github.com/nushell/nu_scripts/blob/main/background_task/job.nu

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.

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`.

```
let-env Path = ($env.Path | prepend 'C:\Program Files\Git\usr\bin
```

Note: this table assumes Nu 0.60.0 or later.

<table>
<tr><th>Bash</th><th>Nu</th><th>Task</th></tr>
<tr><td>ls</td><td>ls</td><td>Lists the files in the current directory</td></tr>
<tr><td>ls <dir></td><td>ls <dir></td><td>Lists the files in the given directory</td></tr>
<tr><td>ls pattern*</td><td>ls pattern*</td><td>Lists files that match a given pattern</td></tr>
<tr><td>ls -la</td><td>ls --long --all or ls -la</td><td>List files with all available information, including hidden files</td></tr>
<tr><td>ls -d */</td><td>ls | where type == dir</td><td>List directories</td></tr>
<tr><td>find . -name *.rs</td><td>ls **/*.rs</td><td>Find recursively all files that match a given pattern</td></tr>
<tr><td>find . -name Makefile | xargs vim</td><td>ls **/Makefile | get name | vim $in</td><td>Pass values as command parameters</td></tr>
<tr><td>cd <directory></td><td>cd <directory></td><td>Change to the given directory</td></tr>
<tr><td>cd</td><td>cd</td><td>Change to the home directory</td></tr>
<tr><td>cd -</td><td>cd -</td><td>Change to the previous directory</td></tr>
<tr><td>mkdir <path></td><td>mkdir <path></td><td>Creates the given path</td></tr>
<tr><td>mkdir -p <path></td><td>mkdir <path></td><td>Creates the given path, creating parents as necessary</td></tr>
<tr><td>touch test.txt</td><td>touch test.txt</td><td>Create a file</td></tr>
<tr><td>> <path></td><td>| save --raw <path></td><td>Save string into a file</td></tr>
<tr><td>>> <path></td><td>| save --raw --append <path></td><td>Append string to a file</td></tr>
<tr><td>cat <path></td><td>open --raw <path></td><td>Display the contents of the given file</td></tr>
<tr><td></td><td>open <path></td><td>Read a file as structured data</td></tr>
<tr><td>mv <source> <dest></td><td>mv <source> <dest></td><td>Move file to new location</td></tr>
<tr><td>cp <source> <dest></td><td>cp <source> <dest></td><td>Copy file to new location</td></tr>
<tr><td>cp -r <source> <dest></td><td>cp -r <source> <dest></td><td>Copy directory to a new location, recur-</td></tr>
</table>

Coming from CMD.EXE

This table was last updated for Nu 0.67.0.

<table>
<tr><th>CMD.EXE</th><th>Nu</th><th>Task</th></tr>
<tr><td>ASSOC</td><td></td><td>Displays or modifies file extension associations</td></tr>
<tr><td>BREAK</td><td></td><td>Trigger debugger breakpoint</td></tr>
<tr><td>CALL <filename.bat></td><td><filename.bat></td><td>Run a batch program</td></tr>
<tr><td></td><td>nu <filename></td><td>Run a nu script in a fresh context</td></tr>
<tr><td></td><td>source <filename></td><td>Run a nu script in this context</td></tr>
<tr><td></td><td>use <filename></td><td>Run a nu script as a module</td></tr>
<tr><td>CD or CHDIR</td><td>$env.PWD</td><td>Get the present working directory</td></tr>
<tr><td>CD <directory></td><td>cd <directory></td><td>Change the current directory</td></tr>
<tr><td>CD /D <drive:directory></td><td>cd <drive:directory></td><td>Change the current directory</td></tr>
<tr><td>CLS</td><td>clear</td><td>Clear the screen</td></tr>
<tr><td>COLOR</td><td></td><td>Set the console default foreground/background colors</td></tr>
<tr><td></td><td>ansi {flags} (code)</td><td>Output ANSI codes to change color</td></tr>
<tr><td>COPY <source> <destination></td><td>cp <source> <destination></td><td>Copy files</td></tr>
<tr><td>COPY <file1>+<file2> <destination></td><td>[<file1>, <file2>] | each { open --raw } | str join | save --raw <destination></td><td>Append multiple files into one</td></tr>
<tr><td>DATE /T</td><td>date now</td><td>Get the current date</td></tr>
<tr><td>DATE</td><td></td><td>Set the date</td></tr>
<tr><td>DEL <file> or ERASE <file></td><td>rm <file></td><td>Delete files</td></tr>
<tr><td>DIR</td><td>ls</td><td>List files in the current directory</td></tr>
<tr><td>ECHO <message></td><td>print <message></td><td>Print the given values to stdout</td></tr>
<tr><td>ECHO ON</td><td></td><td>Echo executed commands to stdout</td></tr>
</table>

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`, `DIR`, `ECHO`, `FTYPE`, `MKLINK`, `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
fetch		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			
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
fetch	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			
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			

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	re-matches, re-seq, re-find			
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			

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, &&	-a, &&
||	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 sopic 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[1] 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

[1] `book/how_nushell_code_gets_run.md`

```
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

```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 instuctions 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

[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

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 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:

```python
# 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

[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

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
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

[8] https://www.nushell.sh/book/thinking_in_nu.html#think-of-nushell-as-a-compiled-language

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" langauge is whether it has or has not the eval function.

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`, but similar conclusions apply to other commands that parse Nushell source code, such as `use`, `overlay use`, `hide`, `register` or `source-env`.

1. Sourcing a dynamic path

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

Let's break down what would need to happen for this to work (assuming `$my_path` is set somewhere):

[9] https://doc.rust-lang.org/stable/book/ch04-00-understanding-ownership.html

1. Parse `source $"($my_path)/common.nu"`
2. To evaluate `source $"($my_path)/common.nu"`: 2.1. Parse `$"($my_path)/common.nu"` 2.2. Evaluate `$"($my_path)/common.nu"` to get the file name 2.3. Parse the contents of the file 2.4. 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++

```
#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 1.1. Parse `"def abc [] { 1 + 2 }" | save output.nu` 1.2. Parse `source "output.nu"` - 1.2.1. Open `output.nu` and parse its contents

2. Evaluate the whole source code 2.1. Evaluate `"def abc [] { 1 + 2 }" | save output.nu` to generate `output.nu` 2.2. ...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` attempts to open and read `foo.nu` during parsing.

REPL

REPL[10] 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
```

[10] https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop

```
# prints files and directories...
```

means the following:

1. Launch `nu`
2. Type `print "Hello world!"`, press Enter
3. Type `ls`, 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

```
cd spam
source-env foo.nu
```

does not work because the directory will be changed *after* `source-env` 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` 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[11] and we're going to expand it as needed.

One pattern that this unlocks is being able to `source`/`use`/etc. a path from a "variable". We've seen that

```
let some_path = 'foo/common.nu'
source $some_path
```

does not work, but we can do the following:

```
const some_path = 'foo/common.nu'
source $some_path
```

We can break down what is happening again:

1. Parse the whole source code 1.1. Parse `const some_path = 'foo/common.nu'` - 1.1.1. Evaluate* `'foo/common.nu'` and store it as a `some_path` constant 1.2. Parse `source $some_path` - 1.2.1. Evaluate* `$some_path`, see that it is a constant, fetch it - 1.2.2. Parse the `foo/common.nu` file

[11] https://github.com/nushell/nushell/pull/7436

2. Evaluate the whole source code 2.1. Evaluate `const some_path = 'foo/common.nu'` (i.e., add the `foo/common.nu` string to the runtime stack as `some_path` variable) 2.2. Evaluate `source $some_path` (i.e., evaluate the contents of `foo/common.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[12]. 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

[12] `https://kristoff.it/blog/what-is-zig-comptime`

code must be available to a C or Rust compiler. For example, you cannot `source` or `use` 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*.

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.

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 have an overview of `explore` command with explore

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.

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.

[1] `https://github.com/pola-rs/polars`

See the installation instructions[2] 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[3] specification, and uses Polars[4] as the motor for performing extremely fast columnar operations[5].

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[6] commands. For the time being don't pay too much attention to the `dataframe` commands. They will be explained in later sections of this page.

> *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.*

File information

The file that we will be using for the benchmarks is the New Zealand business demography[7] dataset. Feel free to download it if you want to

[2] /book/installation.md

[3] https://arrow.apache.org/

[4] https://github.com/pola-rs/polars

[5] https://h2oai.github.io/db-benchmark/

[6] https://pandas.pydata.org/

[7] https://www.stats.govt.nz/assets/Uploads/New-Zealand-business-demography-statistics/New-Zealand-business-demography-statistics-At-February-2020/Download-data/Geographic-units-by-industry-and-statistical-area-2000-2020-descending-ord

follow these tests.

The dataset has 5 columns and 5,429,252 rows. We can check that by using the `ls-df` command:

```
> let df = (open-df .\Data7602DescendingYearOrder.csv)
> ls-df


 #   name    rows     columns

 0   $df    5429252   5

```

We can have a look at the first lines of the file using `first`:

```
> $df | first


 #   anzsic06    Area     year   geo_count   ec_count

 0   A          A100100   2000          96        130
 1   A          A100200   2000         198        110
 2   A          A100300   2000          42         25
 3   A          A100400   2000          66         40
 4   A          A100500   2000          63         40

```

...and finally, we can get an idea of the inferred data types:

```
> $df | dtypes


 #    column     dtype

 0   anzsic06    str
 1   Area        str
 2   year        i64
 3   geo_count   i64
 4   ec_count    i64
```

`zip`

Loading the file

Let's start by comparing loading times between the various methods. First, we will load the data using Nushell's `open` command:

```
> benchmark {open .\Data7602DescendingYearOrder.csv}


 #          real time

 0   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:

```python
import pandas as pd

df = pd.read_csv("Data7602DescendingYearOrder.csv")
```

And the benchmark for it is:

```
> benchmark {python load.py}


 #         real time

 0   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 `open-df` command:

```
> benchmark {open-df .\Data7602DescendingYearOrder.csv}


 #      real time

 0   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. :::

```
> benchmark {
open .\Data7602DescendingYearOrder.csv
| group-by year
| transpose header rows
| upsert rows { get rows | math sum }
| flatten
}


 #        real time

 0   6min 30sec 622ms 312us

```

So, six minutes to perform this aggregated operation.

Let's try the same operation in pandas:

```python
import pandas as pd

df = pd.read_csv("Data7602DescendingYearOrder.csv")
res = df.groupby("year")["geo_count"].sum()
```

```
print(res)
```

And the result from the benchmark is:

```
> benchmark {python .\load.py}


 #         real time

 0   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 = open-df Data7602DescendingYearOrder.csv
let res = ($df | group-by year | agg (col geo_count | sum)
)
$res
```

and the benchmark with dataframes is:

```
> benchmark {source load.nu}


 #       real time

 0   557ms 658us 500ns

```

Luckily Nushell dataframes managed to halve the time again. Isn't that great?

As you can see, Nushell's `Dataframe` commands 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, 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
8,18,0.8,7.0,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 `open-df` command like this:

```
> let df = open-df test_small.csv
```

This should create the value `$df` in memory which holds the data we

:: tip The command `open-df` can read either **csv** or **parquet** files.
:::

To see all the dataframes that are stored in memory you can use

```
> ls-df


 #   name   rows   columns

 0   $df    10     8
```

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    0.1000    1.0000   a       b
c       first
 1      2      12    0.2000    1.0000   a       b
c       second
 2      3      13    0.3000    2.0000   a       b
c       third
 3      4      14    0.4000    3.0000   b       a
c       second
 4      0      15    0.5000    4.0000   b       a
a       third
 5      6      16    0.6000    5.0000   b       a
a       second
 6      7      17    0.7000    6.0000   b       c
a       third
 7      8      18    0.8000    7.0000   c       c
b       eight
 8      9      19    0.9000    8.0000   c       c
b       ninth
 9      0      10    0.0000    9.0000   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 `$nu.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 | sum


 #   int_1   int_2   float_1   float_2   first   second
third   word

 0      40     145    4.5000   46.0000
```

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 `select` command

```
$df | sum | select int_1 int_2 float_1 float_2


 #   int_1   int_2   float_1   float_2

 0      40     145    4.5000   46.0000
```

You can even store the result from this aggregation as you would store any other Nushell variable

```
> let res = ($df | sum | 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

```
> ls-df


 #   name   rows   columns

 0   $df    10     8
```

```
 1   $res   1      4
```

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.

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 `open-df` command to create the new variable

```
> let df_a = open-df 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 | join $df_a int_1 int_1


 #   int_1   int_2   float_1   float_2   first   second
third    word     int_2_right   float_1_right   float_2_-
right   first_right

 0       6      16    0.6000    5.0000   b       a
a       second             11          0.1000         0.
0000   b
 1       7      17    0.7000    6.0000   b       c
a       third              12          0.2000         1.
```

```
0000   a
 2       8      18    0.8000    7.0000   c       c
b       eight               13          0.3000         2.
0000   a
 3       9      19    0.9000    8.0000   c       c
b       ninth               14          0.4000         3.
0000   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 `join` we can join on multiple columns as long as they have the same type, for example we could have done `$df | join $df_a [int_1 int_2] [int_1 int_2]` :::

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 `group-by`. 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 `group-by` command

```
> let group = ($df | 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 | agg (col int_1 | sum)


 #   first   int_1

 0   a               6
 1   b              17
 2   c              17
```

or we can define multiple aggregations on the same or different columns

```
$group | agg [
(col int_1 | n-unique)
(col int_2 | min)
(col float_1 | sum)
(col float_2 | count)
] | sort-by first


 #   first   int_1       int_2       float_1       float_-
2

 0   a               3          11        0.6000         3
 1   b               4          14        2.2000         4
 2   c               3          10        1.7000         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 `into df`.

```
> let a = ([[a b]; [1 2] [3 4] [5 6]] | into df)
> $a
```

```
 #   b   a

 0   2   1
 1   4   3
 2   6   5
```

::: 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 | with-column $a.a --name a2 | with-column
$a.a --name a3)


 #   b   a   a2   a3

 0   2   1    1    1
 1   4   3    3    3
 2   6   5    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

```
> ls-df


 #    name   rows   columns

 0   $a      3      2
 1   $a2     3      4
 2   $df_a   4      5
 3   $df     10     8
```

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[8]). The other optimization trick is the fact 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 `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 `into df` command:

```
> let new = ([9 8 4] | 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.

[8] https://arrow.apache.org/docs/format/Columnar.html

```
> 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 `$nu.scope.vars` :::

Let's rename our previous Series so it has a memorable name

```
> let new_2 = ($new_2 | 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_0


 0       -28
 1       -26
 2       -18
```

And we can add them to previously defined dataframes

```
> let new_df = ($a | with-column $new --name new_col)
> $new_df


 #   b   a   new_col

 0   2   1         9
 1   4   3         8
 2   6   5         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 | with-column ($new_df.a * $new_-
df.b / $new_df.new_col) --name my_sum)
> let $new_df


 #   b   a   new_col   my_sum

 0   2   1         9        0
 1   4   3         8        1
 2   6   5         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


 #   new_col

 0   false
 1   true
 2   false
```

and with this mask we can now filter a dataframe, like this

```
> $new_df | 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] | into df)
> $new_df | 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 && $mask1

  #   and_new_col_mask

  0   false
  1   true
  2   false
```

and `OR` operations

```
> $mask || $mask1

  #   or_new_col_mask

  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.first | is-in ([b c] | into df))

  #   first

  0   false
  1   false
  2   false
  3   true
```

```
 4   true
 5   true
 6   true
 7   true
 8   true
 9   true
```

and this new mask can be used to filter the dataframe

```
> $df | filter-with $mask3


 #   int_1   int_2   float_1   float_2   first   second
third    word

 0       4      14    0.4000    3.0000   b       a
c       second
 1       0      15    0.5000    4.0000   b       a
a       third
 2       6      16    0.6000    5.0000   b       a
a       second
 3       7      17    0.7000    6.0000   b       c
a       third
 4       8      18    0.8000    7.0000   c       c
b       eight
 5       9      19    0.9000    8.0000   c       c
b       ninth
 6       0      10    0.0000    9.0000   c       c
b       ninth
```

Another operation that can be done with masks is setting or replacing a value from a series. For example, we can change the value in the column `first` where the value is equal to `a`

```
> $df.first | set new --mask ($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] | into df)
> $df | take $indices


 #   int_1   int_2   float_1   float_2   first   second
third    word

 0       2      12    0.2000    1.0000   a       b
c       second
 1       0      15    0.5000    4.0000   b       a
a       third
 2       7      17    0.7000    6.0000   b       c
a       third
```

The command `take` 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 `arg-unique` as shown in the next example

```
> let indices = ($df.first | arg-unique)
> $df | take $indices


 #   int_1   int_2   float_1   float_2   first   second
third    word

 0       1      11    0.1000    1.0000   a       b
c       first
 1       4      14    0.4000    3.0000   b       a
c       second
 2       8      18    0.8000    7.0000   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` :::

```
> let indices = ($df.word | arg-sort)
> $df | take $indices


 #   int_1   int_2   float_1   float_2   first   second
third    word

 0       8      18    0.8000    7.0000   c       c
b       eight
 1       1      11    0.1000    1.0000   a       b
c       first
 2       9      19    0.9000    8.0000   c       c
b       ninth
 3       0      10    0.0000    9.0000   c       c
b       ninth
 4       2      12    0.2000    1.0000   a       b
c       second
 5       4      14    0.4000    3.0000   b       a
c       second
 6       6      16    0.6000    5.0000   b       a
a       second
```

```
 7       3      13    0.3000    2.0000   a       b
c       third
 8       0      15    0.5000    4.0000   b       a
a       third
 9       7      17    0.7000    6.0000   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

```
> let indices = ([0 2] | into df);
> $df.int_1 | 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.first | value-counts


 #   first   counts

 0   b            4
 1   c            3
 2   a            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.first | 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`

```
> $df | filter-with ($df.word | is-unique)


 #   int_1   int_2   float_1   float_2   first   second
third   word

 0       1      11    0.1000    1.0000   a       b
c       first
 1       8      18    0.8000    7.0000   c       c
b       eight

```

Or all the duplicated ones

```
> $df | filter-with ($df.word | is-duplicated)


 #   int_1   int_2   float_1   float_2   first   second
third    word

 0       2      12    0.2000    1.0000   a       b
c       second
 1       3      13    0.3000    2.0000   a       b
c       third
 2       4      14    0.4000    3.0000   b       a
c       second
 3       0      15    0.5000    4.0000   b       a
a       third
 4       6      16    0.6000    5.0000   b       a
a       second
 5       7      17    0.7000    6.0000   b       c
a       third
 6       9      19    0.9000    8.0000   c       c
b       ninth
 7       0      10    0.0000    9.0000   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]] | into lazy)

> $a
```

```
plan              DATAFRAME(in-memory): ["a", "b"];
                      project */2 columns    |    details:
None;
                      selection: "None"
optimized_plan    DATAFRAME(in-memory): ["a", "b"];
                      project */2 columns    |    details:
None;
                      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 | 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
:::   | reverse
:::   | with-column [
```

```
:::       ((col a) * 2 | as double_a)
:::       ((col a) / 2 | as half_a)
:::    ]
:::    | 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 `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

```
$nu.scope.commands | where category =~ expression
```

In our previous example, we use the `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 `col` command can be inferred. For example, using the select command we can use only a string

```
> $a | select a | collect
```

or the `col` command

```
> $a | select (col a) | 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]] | into lazy )
> $a
:::   | group-by name
:::   | agg [
:::       (col value | sum | as sum)
:::       (col value | mean | as mean)
:::     ]
:::   | 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]] | into lazy )
> let group = ($a
:::   | group-by name
:::   | agg [
:::       (col value | sum | as sum)
:::       (col value | mean | as mean)
:::     ])
> $a | join $group name name | 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 descriptions, and whenever possible, their analogous Nushell command.

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. For example, the next step for dataframes is the introduction of Lazy Dataframes. These will allow you to define complex data operations that will not be executed until you decide to “finish” the pipe. This will give Nushell the chance to select the optimal plan to query the data you would be asking for.

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` 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 command again, but this time, we'll look at the tags it gives back:

```
> open Cargo.toml | metadata

 span     {record 2 fields}
```

Currently, we track only the span of where values come from. Let's take a closer look at that:

```
> open Cargo.toml | metadata | get span

 start   5
 end     15
```

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` 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` command. This command takes in a record that describes the error to create:

```
error make {msg: "this is fishy", label: {text: "fish right
here", start: $span.start, end: $span.end } }
```

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",
            start: $span.start,
            end: $span.end
        }
    }
}
```

When called with a value, we'll now see an error message returned:

```
> 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, a companion to the each command.

Like each, par-each works on each element in the pipeline as it comes in, running a block on each. Unlike each, par-each 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, 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 to par-each:

```
> 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 to work in multiple directories in parallel (notice the cd 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 and par-each 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[9]. Nu plugins can use either JSON or MSGPACK as their communication encoding.

Adding a plugin

To add a plugin, call the `register` 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:

```
> register ./my_plugins/nu_plugin_cool
```

Windows:

```
> register .\my_plugins\nu_plugin_cool.exe
```

When `register` is called:

1. Nu launches the plugin, and wait for plugin 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:

[9] https://en.wikipedia.org/wiki/Standard_streams

```
> 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[10]
- Python[11]

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[12] is a great place to ask questions!

`explore`

Explore is a table pager, just like `less` but for table structured data.

Signature

`> table --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

[10] https://github.com/nushell/nushell/tree/main/crates/nu_plugin_example
[11] https://github.com/nushell/nushell/blob/main/crates/nu_plugin_python
[12] https://discord.gg/NtAbbGn

- `--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` 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 chose a nessary 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`.

Commands

`explore` 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`.

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`.

`:tweak` command

The `tweak` command can be used to set config options right inside. For example run

```
:tweak table.split_line red
```

Interactive configuration

```
explore -i
```

`:config-show` command

You can use `:config-show` to inspect currently used config.

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[13]".

[13] https://github.com/rust-lang/regex

Chapter 9

Command Reference

To see all commands in Nushell, please run `help commands`

agg-groups

version: 0.74.0

usage:

creates an agg_groups expression

Signature

`> agg-groups`

Examples

```
>
```

agg

version: 0.74.0

usage:

Performs a series of aggregations from a group-by

Signature

> agg

Examples

Group by and perform an aggregation

```
> [[a b]; [1 2] [1 4] [2 6] [2 4]]
    | into df
    | group-by a
    | agg [
        (col b | min | as "b_min")
        (col b | max | as "b_max")
        (col b | sum | as "b_sum")
     ]
```

Group by and perform an aggregation

```
> [[a b]; [1 2] [1 4] [2 6] [2 4]]
    | into lazy
    | group-by a
    | agg [
        (col b | min | as "b_min")
        (col b | max | as "b_max")
        (col b | sum | as "b_sum")
     ]
    | collect
```

alias

version: 0.74.0

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

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Alias ll to ls -l

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

Make an alias that makes a list of all custom commands

```
> alias customs = ($nu.scope.commands | where is_custom
| get command)
```

all-false

version: 0.74.0

usage:

Returns true if all values are false

Signature

`> all-false`

Examples

Returns true if all values are false

```
> [false false false] | into df | all-false
```

Checks the result from a comparison

```
> let s = ([5 6 2 10] | into df);
    let res = ($s > 9);
    $res | all-false
```

all-true

version: 0.74.0

usage:

Returns true if all values are true

Signature

```
> all-true
```

Examples

Returns true if all values are true

```
> [true true true] | into df | all-true
```

Checks the result from a comparison

```
> let s = ([5 6 2 8] | into df);
    let res = ($s > 9);
    $res | all-true
```

all

version: 0.74.0

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

Examples

Check if each row's status is the string 'UP'

```
> [[status]; [UP] [UP]] | all {|el| $el.status == UP }
```

Check that all values are equal to twice their index

```
> [0 2 4 6] | all {|el ind| $el == $ind * 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: 0.74.0

usage:

Output ANSI codes to change color.

Signature

`> ansi (code) --escape --osc --list`

Parameters

- `code`: the name of the code to use like 'green' or 'reset' to reset the color
- `--escape`: escape sequence without the escape character(s)
- `--osc`: operating system command (ocs) escape sequence without the escape character(s)
- `--list`: list available ansi code names

Notes

```
For escape sequences:
Escape: '\x1b[' is not required for --escape parameter
Format: #(;#)m
Example: 1;31m for bold red or 2;37;41m for dimmed white
fg with red bg
There can be multiple text formatting sequence numbers
separated by a ; and ending with an m where the # is of
the
following values:
    attribute_number, abbreviation, description
    0     reset / normal display
    1  b  bold or increased intensity
    2  d  faint or decreased intensity
    3  i  italic on (non-mono font)
    4  u  underline on
    5  l  slow blink on
    6     fast blink on
    7  r  reverse video on
    8  h  nondisplayed (invisible) on
    9  s  strike-through on

    foreground/bright colors    background/bright colors
    30/90    black              40/100    black
    31/91    red                41/101    red
    32/92    green              42/102    green
    33/93    yellow             43/103    yellow
    34/94    blue               44/104    blue
    35/95    magenta            45/105    magenta
    36/96    cyan               46/106    cyan
    37/97    white              47/107    white
    39       default            49        default
    https://en.wikipedia.org/wiki/ANSI_escape_code

OSC: '\x1b]' is not required for --osc parameter
Example: echo [(ansi -o '0') 'some title' (char bel)] |
str join
Format: #
    0 Set window title and icon name
    1 Set icon name
```

```
    2 Set window title
    4 Set/read color palette
    9 iTerm2 Grown notifications
    10 Set foreground color (x11 color spec)
    11 Set background color (x11 color spec)
    ... others
```

Examples

Change color to green

```
> ansi green
```

Reset the color

```
> ansi reset
```

Use ansi to color text (rb = red bold, gb = green bold, pb = purple bold)

```
> $'(ansi rb)Hello (ansi gb)Nu (ansi pb)World(ansi reset)
'
```

Use ansi to color text (italic bright yellow on red 'Hello' with green bold 'Nu' and purple bold 'World')

```
> [(ansi -e '3;93;41m') Hello (ansi reset) " " (ansi gb)
 Nu " " (ansi pb) World (ansi reset)] | str join
```

Use ansi to color text with a style (blue on red in bold)

```
> $"(ansi -e { fg: '#0000ff' bg: '#ff0000' attr: b })Hello
Nu World(ansi reset)"
```

ansi gradient

version: 0.74.0

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

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 strip

version: 0.74.0

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

Examples

Strip ANSI escape sequences from a string

```
> $'(ansi green)(ansi cursor_on)hello' | ansi strip
```

any

version: 0.74.0

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

Examples

Check if any row's status is the string 'DOWN'

```
> [[status]; [UP] [DOWN] [UP]] | any {|el| $el.status ==
DOWN }
```

Check if any value is equal to twice its own index

```
> [9 8 7 6] | any {|el ind| $el == $ind * 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: 0.74.0

usage:

Appends a new dataframe

Signature

```
> append
```

Examples

Appends a dataframe as new columns

```
> let a = ([[a b]; [1 2] [3 4]] | into df);
    $a | append $a
```

Appends a dataframe merging at the end of columns

```
> let a = ([[a b]; [1 2] [3 4]] | into df);
    $a | append $a --col
```

append

version: 0.74.0

usage:

Append any number of rows to a table.

Signature

```
> append (row)
```

Parameters

- `row`: the row, list, or table to append

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.
```

Examples

Append one Int item

```
> [0,1,2,3] | append 4
```

Append three Int items

```
> [0,1] | append [2,3,4]
```

Append Ints and Strings

```
> [0,1] | append [2,nu,4,shell]
```

arg-max

version: 0.74.0

usage:

Return index for max value in series

Signature

> arg-max

Examples

Returns index for max value

```
> [1 3 2] | into df | arg-max
```

arg-min

version: 0.74.0

usage:

Return index for min value in series

Signature

> arg-min

Examples

Returns index for min value

```
> [1 3 2] | into df | arg-min
```

arg-sort

version: 0.74.0

usage:

Returns indexes for a sorted series

Signature

`> arg-sort`

Examples

Returns indexes for a sorted series

```
> [1 2 2 3 3] | into df | arg-sort
```

Returns indexes for a sorted series

```
> [1 2 2 3 3] | into df | arg-sort -r
```

arg-true

version: 0.74.0

usage:

Returns indexes where values are true

Signature

`> arg-true`

Examples

Returns indexes where values are true

```
> [false true false] | into df | arg-true
```

arg-unique

version: 0.74.0

usage:

Returns indexes for unique values

Signature

```
> arg-unique
```

Examples

Returns indexes for unique values

```
> [1 2 2 3 3] | into df | arg-unique
```

arg-where

version: 0.74.0

usage:

Creates an expression that returns the arguments where expression is true

Signature

```
> arg-where
```

Examples

Return a dataframe where the value match the expression

```
> let df = ([[a b]; [one 1] [two 2] [three 3]] | into df)
;
    $df | select (arg-where ((col b) >= 2) | as b_arg)
```

as-date

version: 0.74.0

usage:

Converts string to date.

Signature

```
> as-date
```

Notes

```
Format example:
        "%Y-%m-%d"     => 2021-12-31
        "%d-%m-%Y"     => 31-12-2021
        "%Y%m%d"       => 2021319 (2021-03-19)
```

Examples

Converts string to date

```
> ["2021-12-30" "2021-12-31"] | into df | as-datetime "%Y-
%m-%d"
```

as-datetime

version: 0.74.0

usage:

Converts string to datetime.

Signature

```
> as-datetime
```

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
```

Examples

Converts string to datetime

```
> ["2021-12-30 00:00:00" "2021-12-31 00:00:00"] | into
df | as-datetime "%Y-%m-%d %H:%M:%S"
```

as

version: 0.74.0

usage:

Creates an alias expression

Signature

```
> as
```

Examples

Creates and alias expression

```
> col a | as new_a | into nu
```

ast

version: 0.74.0

usage:

Print the abstract syntax tree (ast) for a pipeline.

Signature

```
> ast (pipeline)
```

Parameters

- `pipeline`: the pipeline to print the ast for

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 '
```

benchmark

version: 0.74.0

usage:

Time the running time of a block

Signature

`> benchmark (block)`

Parameters

- `block`: the block to run

Examples

Benchmarks a command within a block

> ```
> > benchmark { sleep 500ms }
> ```

bits

version: 0.74.0

usage:

Various commands for working with bits

Signature

`> bits`

Notes

> ```
> You must use one of the following subcommands. Using this
> command as-is will only produce this help message.
> ```

bits and

version: 0.74.0

usage:

Performs bitwise and for integers

Signature

> bits and (target)

Parameters

- `target`: target integer to perform bit and

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: 0.74.0

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

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 -n 2
```

Apply logical negation to a list of numbers, treat input as signed number

```
> [4 3 2] | bits not -s
```

bits or

version: 0.74.0

usage:

Performs bitwise or for integers

Signature

`> bits or (target)`

Parameters

- `target`: target integer to perform bit or

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: 0.74.0

usage:

Bitwise rotate left for integers

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`

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: 0.74.0

usage:

Bitwise rotate right for integers

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`

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 -n 1
```

bits shl

version: 0.74.0

usage:

Bitwise shift left for integers

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`

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 -n 1
```

Shift left a signed number by 1 bit

```
> 0x7F | bits shl 1 -s
```

Shift left a list of numbers

```
> [5 3 2] | bits shl 2
```

bits shr

version: 0.74.0

usage:

Bitwise shift right for integers

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`

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: 0.74.0

usage:

Performs bitwise xor for integers

Signature

`> bits xor (target)`

Parameters

- `target`: target integer to perform bit xor

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: 0.74.0

usage:

Break a loop

Signature

```
> break
```

Notes

> ```
> This command is a parser keyword. For details, check:
>   https://www.nushell.sh/book/thinking_in_nu.html
> ```

Examples

Break out of a loop

> ```
> > loop { break }
> ```

bytes

version: 0.74.0

usage:

Various commands for working with byte data

Signature

```
> bytes
```

Notes

> ```
> You must use one of the following subcommands. Using this
> command as-is will only produce this help message.
> ```

bytes add

version: 0.74.0

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

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] -i 1
```

Add bytes `0x[11]` to `0x[FF AA AA]` at the end

```
> 0x[FF AA AA] | bytes add 0x[11] -e
```

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] -e -i 1
```

bytes at

version: 0.74.0

usage:

Get bytes defined by a range. Note that the start is included but the end is excluded, and that the first byte is index 0.

Signature

```
> bytes at (range) ...rest
```

Parameters

- `range`: the indexes to get bytes
- `...rest`: for a data structure input, get bytes from data at the given cell paths

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]
```

Alternatively, you can use the form

```
> 0x[33 44 55 10 01 13] | bytes at '3,4'
```

Drop the last `n` characters from the string

```
> 0x[33 44 55 10 01 13] | bytes at ',-3'
```

Get the remaining characters from a starting index

```
> 0x[33 44 55 10 01 13] | bytes at '3,'
```

Get the characters from the beginning until ending index

```
> 0x[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: 0.74.0

usage:

Create bytes from the arguments.

Signature

`> bytes build ...rest`

Parameters

- `...rest`: list of bytes

Examples

Builds binary data from 0x[01 02], 0x[03], 0x[04]

```
> bytes build 0x[01 02] 0x[03] 0x[04]
```

bytes collect

version: 0.74.0

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

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: 0.74.0

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

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: 0.74.0

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

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 -e 0x[44
```

```
55]
```

Returns all matched index

```
>  0x[33 44 55 10 01 33 44 33 44] | bytes index-of -a 0x[33
44]
```

Returns all matched index, searching from end

```
>  0x[33 44 55 10 01 33 44 33 44] | bytes index-of -a -
e 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: 0.74.0

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

Examples

Return the lengths of multiple strings

```
> 0x[1F FF AA AB] | bytes length
```

Return the lengths of multiple strings

```
> [0x[1F FF AA AB] 0x[1F]] | bytes length
```

bytes remove

version: 0.74.0

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

Examples

Remove contents

```
> 0x[10 AA FF AA FF] | bytes remove 0x[10 AA]
```

Remove all occurrences of find binary

```
> 0x[10 AA 10 BB 10] | bytes remove -a 0x[10]
```

Remove occurrences of find binary from end

```
> 0x[10 AA 10 BB CC AA 10] | bytes remove -e 0x[10]
```

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: 0.74.0

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

Examples

Find and replace contents

```
> 0x[10 AA FF AA FF] | bytes replace 0x[10 AA] 0x[FF]
```

Find and replace all occurrences of find binary

```
> 0x[10 AA 10 BB 10] | bytes replace -a 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 -a 0x[11] 0x[13] ColA ColC
```

bytes reverse

version: 0.74.0

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

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: 0.74.0

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

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]
```

cache

version: 0.74.0

usage:

Caches operations in a new LazyFrame

Signature

```
> cache
```

Examples

Caches the result into a new LazyFrame

```
> [[a b]; [6 2] [4 2] [2 2]] | into df | reverse | cache
```

cal

version: 0.74.0

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

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: 0.74.0

usage:

Change directory.

Signature

`> cd (path)`

Parameters

- `path`: the path to change to

Examples

Change to your home directory

```
> cd ~
```

Change to a directory via abbreviations

```
> cd d/s/9
```

Change to the previous working directory ($OLDPWD)

```
> cd -
```

char

version: 0.74.0

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

Examples

Output newline

```
> char newline
```

Output prompt character, newline and a hamburger menu character

```
> (char prompt) + (char newline) + (char hamburger)
```

Output Unicode character

```
> char -u 1f378
```

Create Unicode from integer codepoint values

```
> char -i (0x60 + 1) (0x60 + 2)
```

Output multi-byte Unicode character

```
> char -u 1F468 200D 1F466 200D 1F466
```

clear

version: 0.74.0

usage:

Clear the terminal.

Signature

> clear

Examples

Clear the terminal

```
> clear
```

col

version: 0.74.0

usage:

Creates a named column expression

Signature

> col

Examples

Creates a named column expression and converts it to a nu object

```
> col a | into nu
```

collect

version: 0.74.0

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

Examples

Use the second value in the stream

```
> [1 2 3] | collect { |x| $x.1 }
```

collect

version: 0.74.0

usage:

Collect lazy dataframe into eager dataframe

Signature

```
> collect
```

Examples

drop duplicates

```
> [[a b]; [1 2] [3 4]] | into lazy | collect
```

columns

version: 0.74.0

usage:

Show dataframe columns

Signature

```
> columns
```

Examples

Dataframe columns

```
> [[a b]; [1 2] [3 4]] | into df | columns
```

columns

version: 0.74.0

usage:

Given a record or table, produce a list of its columns' names.

Signature

```
> columns
```

Notes

```
This is a counterpart to `values`, which produces a list
of columns' values.
```

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
```

commandline

version: 0.74.0

usage:

View or modify the current command line input buffer

Signature

`> commandline (cmd) --append --insert --replace`

Parameters

- `cmd`: the string to perform the operation with
- `--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)

compact

version: 0.74.0

usage:

Creates a table with non-empty rows.

Signature

`> compact ...rest`

Parameters

- `...rest`: the columns to compact from the table

Examples

Filter out all records where 'Hello' is null (returns nothing)

```
> [["Hello" "World"]; [null 3]]| compact Hello
```

Filter out all records where 'World' is null (Returns the table)

```
> [["Hello" "World"]; [null 3]]| compact World
```

Filter out all instances of nothing from a list (Returns [1,2])

```
> [1, null, 2] | compact
```

complete

version: 0.74.0

usage:

Complete the external piped in, collecting outputs and exit code

Signature

`> complete`

Examples

Run the external completion

```
> ^external arg1 | complete
```

concat-str

version: 0.74.0

usage:

Creates a concat string expression

Signature

```
> concat-str
```

Examples

Creates a concat string expression

```
> let df = ([[a b c]; [one two 1] [three four 2]] | into
df);
    $df | with-column ((concat-str "-" [(col a) (col b)
 ((col c) * 2)]) | as concat)
```

concatenate

version: 0.74.0

usage:

Concatenates strings with other array

Signature

```
> concatenate
```

Examples

Concatenate string

```
> let other = ([za xs cd] | into df);
    [abc abc abc] | into df | concatenate $other
```

config

version: 0.74.0

usage:

Edit nushell configuration files

Signature

```
> config
```

Notes

> ```
> You must use one of the following subcommands. Using this
> command as-is will only produce this help message.
> ```

config env

version: 0.74.0

usage:

Edit nu environment configurations

Signature

```
> config env
```

Examples

allow user to open and update nu env

> ```
> > config env
> ```

config nu

version: 0.74.0

usage:

Edit nu configurations

Signature

`> config nu`

Examples

allow user to open and update nu config

```
> config nu
```

config reset

version: 0.74.0

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

Examples

reset nushell configuration files

```
> config reset
```

const

version: 0.74.0

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

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Create a new parse-time constant.

```
> const x = 10
```

Create a composite constant value

```
> const x = { a: 10, b: 20 }
```

contains

version: 0.74.0

usage:

Checks if a pattern is contained in a string

Signature

> `contains`

Examples

Returns boolean indicating if pattern was found

```
> [abc acb acb] | into df | contains ab
```

continue

version: 0.74.0

usage:

Continue a loop from the next iteration

Signature

```
> continue
```

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Continue a loop from the next iteration

```
> for i in 1..10 { if $i == 5 { continue }; print $i }
```

count-null

version: 0.74.0

usage:

Counts null values

Signature

```
> count-null
```

Examples

Counts null values

```
> let s = ([1 1 0 0 3 3 4] | into df);
    ($s / $s) | count-null
```

count

version: 0.74.0

usage:

creates a count expression

Signature

```
> count
```

Examples

```
>
```

cp

version: 0.74.0

usage:

Copy files.

Signature

```
> cp (source) (destination) --recursive --verbose --interactive
--no-symlink
```

Parameters

- `source`: the place to copy from
- `destination`: the place to copy to
- `--recursive`: copy recursively through subdirectories
- `--verbose`: show successful copies in addition to failed copies (default:false)
- `--interactive`: ask user to confirm action
- `--no-symlink`: no symbolic links are followed, only works if -r is active

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
```

cumulative

version: 0.74.0

usage:

Cumulative calculation for a series

Signature

```
> cumulative
```

Examples

Cumulative sum for a series

```
> [1 2 3 4 5] | into df | cumulative sum
```

date

version: 0.74.0

usage:

Date-related commands

Signature

```
> date
```

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

date format

version: 0.74.0

usage:

Format a given date using a format string.

Signature

```
> date format (format string) --list
```

Parameters

- `format string`: the desired date format
- `--list`: lists strftime cheatsheet

Examples

Format a given date-time as a string using the default format (RFC 2822).

```
> "2021-10-22 20:00:12 +01:00" | date format
```

Format the current date-time using a given format string.

```
> date now | date format "%Y-%m-%d %H:%M:%S"
```

Format the current date using a given format string.

```
> date now | date format "%Y-%m-%d %H:%M:%S"
```

Format a given date using a given format string.

```
> "2021-10-22 20:00:12 +01:00" | date format "%Y-%m-%d"
```

date humanize

version: 0.74.0

usage:

Print a 'humanized' format for the date, relative to now.

Signature

`> date humanize`

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: 0.74.0

usage:

List supported time zones.

Signature

```
> date list-timezone
```

Examples

Show timezone(s) that contains 'Shanghai'

```
> date list-timezone | where timezone =~ Shanghai
```

date now

version: 0.74.0

usage:

Get the current date.

Signature

```
> date now
```

Examples

Get the current date and display it in a given format string.

```
> date now | date format "%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: 0.74.0

usage:

Convert the date into a record.

Signature

> date to-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.

```
> '2020-04-12 22:10:57 +0200' | date to-record
```

date to-table

version: 0.74.0

usage:

Convert the date into a structured table.

Signature

> date to-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.

```
> '2020-04-12 22:10:57 +0200' | date to-table
```

date to-timezone

version: 0.74.0

usage:

Convert a date to a given time zone.

Signature

> date to-timezone (time zone)

Parameters

- `time zone`: time zone description

Notes

```
Use 'date list-timezone' to list all supported time zones.
```

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"
```

debug

version: 0.74.0

usage:

Debug print the value(s) piped in.

Signature

`> debug --raw`

Parameters

- `--raw`: Prints the raw value representation

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
```

decode

version: 0.74.0

usage:

Decode bytes as a string.

Signature

`> decode (encoding)`

Parameters

- `encoding`: the text encoding to use

Notes

```
Multiple encodings are supported, here is an example of
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/0.8.28/
```

```
encoding_rs/#statics
```

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
```

decode base64

version: 0.74.0

usage:

Base64 decode a value

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'
- `--binary`: Output a binary value instead of decoding payload as UTF-8

Notes

```
Will attempt to decode binary payload as an UTF-8 string
by default. Use the `--binary(-b)` argument to force binary
output.
```

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
```

def-env

version: 0.74.0

usage:

Define a custom command, which participates in the caller environment

Signature

`> def-env (def_name) (params) (block)`

Parameters

- `def_name`: definition name
- `params`: parameters
- `block`: body of the definition

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html

=== EXTRA NOTE ===
```

```
All blocks are scoped, including variable definition and
environment variable changes.

Because of this, the following doesn't work:

def-env cd_with_fallback [arg = ""] {
    let fall_back_path = "/tmp"
    if $arg != "" {
        cd $arg
    } else {
        cd $fall_back_path
    }
}

Instead, you have to use cd in the top level scope:

def-env cd_with_fallback [arg = ""] {
    let fall_back_path = "/tmp"
    let path = if $arg != "" {
        $arg
    } else {
        $fall_back_path
    }
    cd $path
}
```

Examples

Set environment variable by call a custom command

```
> def-env foo [] { let-env BAR = "BAZ" }; foo; $env.BAR
```

def

version: 0.74.0

usage:

Define a custom command

Signature

> def (def_name) (params) (body)

Parameters

- `def_name`: definition name
- `params`: parameters
- `body`: body of the definition

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

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
```

default

version: 0.74.0

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

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 -i MY_ENV | default 'abc'
```

Replace the `null` value in a list

```
> [1, 2, null, 4] | default 3
```

describe

version: 0.74.0

usage:

Describe the type and structure of the value(s) piped in.

Signature

`> describe --no-collect`

Parameters

- `--no-collect`: do not collect streams of structured data

Examples

Describe the type of a string

```
> 'hello' | describe
```

Describe a stream of data, collecting it first

```
> [1 2 3] | each {|i| $i} | describe
```

Describe the input but do not collect streams

```
> [1 2 3] | each {|i| $i} | describe --no-collect
```

detect columns

version: 0.74.0

usage:

Attempt to automatically split text into multiple columns

Signature

`> detect columns --skip --no-headers`

Parameters

- `--skip {int}`: number of rows to skip before detecting
- `--no-headers`: don't detect headers

Examples

Splits string across multiple columns

```
> 'a b c' | detect columns -n
```

Splits a multi-line string into columns with headers detected

```
> $'c1 c2 c3(char nl)a b c' | detect columns
```

df-not

version: 0.74.0

usage:

Inverts boolean mask

Signature

```
> df-not
```

Examples

Inverts boolean mask

```
> [true false true] | into df | df-not
```

do

version: 0.74.0

usage:

Run a closure, providing it with the pipeline input

Signature

```
> do (closure) ...rest --ignore-errors --ignore-shell-errors
--ignore-program-errors --capture-errors
```

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

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 -i { thisisnotarealcommand }
```

Run the closure and ignore shell errors

```
> do -s { thisisnotarealcommand }
```

Run the closure and ignore external program errors

```
> do -p { nu -c 'exit 1' }; echo "I'll still run"
```

Abort the pipeline if a program returns a non-zero exit code

```
> do -c { nu -c '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 }
```

drop-duplicates

version: 0.74.0

usage:

Drops duplicate values in dataframe

Signature

```
> drop-duplicates
```

Examples

drop duplicates

```
> [[a b]; [1 2] [3 4] [1 2]] | into df | drop-duplicates
```

drop-nulls

version: 0.74.0

usage:

Drops null values in dataframe

Signature

```
> drop-nulls
```

Examples

drop null values in dataframe

```
> let df = ([[a b]; [1 2] [3 0] [1 2]] | into df);
    let res = ($df.b / $df.b);
    let a = ($df | with-column $res --name res);
    $a | drop-nulls
```

drop null values in dataframe

```
> let s = ([1 2 0 0 3 4] | into df);
    ($s / $s) | drop-nulls
```

drop

version: 0.74.0

usage:

Creates a new dataframe by dropping the selected columns

Signature

`> drop`

Examples

drop column a

```
> [[a b]; [1 2] [3 4]] | into df | drop a
```

drop

version: 0.74.0

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

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
```

drop column

version: 0.74.0

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

Examples

Remove the last column of a table

```
> [[lib, extension]; [nu-lib, rs] [nu-core, rb]] | drop
column
```

drop nth

version: 0.74.0

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

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..
```

dtypes

version: 0.74.0

usage:

Show dataframe data types

Signature

```
> dtypes
```

Examples

Dataframe dtypes

```
> [[a b]; [1 2] [3 4]] | into df | dtypes
```

du

version: 0.74.0

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

Examples

Disk usage of the current directory

```
> du
```

dummies

version: 0.74.0

usage:

Creates a new dataframe with dummy variables

Signature

```
> dummies
```

Examples

Create new dataframe with dummy variables from a dataframe

```
> [[a b]; [1 2] [3 4]] | into df | dummies
```

Create new dataframe with dummy variables from a series

```
> [1 2 2 3 3] | into df | dummies
```

each

version: 0.74.0

usage:

Run a closure on each row of the input list, creating a new list with the results.

Signature

`> each (closure) --keep-empty --numbered`

Parameters

- `closure`: the closure to run
- `--keep-empty`: keep empty result cells
- `--numbered`: iterate with an index (deprecated; use a two-parameter closure instead)

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.
```

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] | each {|el ind| if $el == 2 { $"found 2 at (
$ind)!"} }
```

Iterate over each element, keeping all results

```
> [1 2 3] | each --keep-empty {|e| if $e == 2 { "found
2!"} }
```

each while

version: 0.74.0

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) --numbered
```

Parameters

- `closure`: the closure to run
- `--numbered`: iterate with an index (deprecated; use a two-parameter closure instead)

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] | each while {|el ind| if $el < 2 { $"value (
$el) at ($ind)!"} }
```

echo

version: 0.74.0

usage:

Returns its arguments, ignoring the piped-in value.

Signature

`> echo ...rest`

Parameters

- `...rest`: the values to echo

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.
```

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
```

encode

version: 0.74.0

usage:

Encode an UTF-8 string into other kind of representations.

Signature

`> encode (encoding)`

Parameters

- `encoding`: the text encoding to use

Notes

```
Multiple encodings are supported, here is an example of
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/0.8.28/
encoding_rs/#statics
```

Examples

Encode an UTF-8 string into Shift-JIS

```
> "           " | encode shift-jis
```

encode base64

version: 0.74.0

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'

Examples

Encode binary data

```
> 0x[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
```

enter

version: 0.74.0

usage:

Enters a new shell at the given path.

Signature

`> enter (path)`

Parameters

- `path`: the path to enter as a new shell

Examples

Enter a new shell at path '../dir-foo'

```
> enter ../dir-foo
```

env

version: 0.74.0

usage:

Display current environment variables

Signature

`> env`

Examples

Display current path environment variable

```
> env | where name == PATH
```

Check whether the env variable `MY_ENV_ABC` exists

```
> env | any { |e| $e.name == MY_ENV_ABC }
```

Another way to check whether the env variable `PATH` exists

```
> 'PATH' in (env).name
```

error make

version: 0.74.0

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

Examples

Create a custom error for a custom command

```
> def foo [x] {
      let span = (metadata $x).span;
      error make {msg: "this is fishy", label: {text: "fish
right here", start: $span.start, end: $span.end } }
    }
```

Create a simple custom error for a custom command

```
> def foo [x] {
      error make {msg: "this is fishy"}
    }
```

every

version: 0.74.0

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

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: 0.74.0

usage:

Execute a command, replacing the current process.

Signature

`> exec (command)`

Parameters

- `command`: the command to execute

Notes

```
Currently supported only on Unix-based systems.
```

Examples

Execute external 'ps aux' tool

```
> exec ps aux
```

Execute 'nautilus'

```
> exec nautilus
```

exit

version: 0.74.0

usage:

Exit a Nu shell or exit Nu entirely.

Signature

> exit (exit_code) --now

Parameters

- exit_code: Exit code to return immediately with
- --now: Exit out of all shells immediately (exiting Nu)

Examples

Exit the current shell

```
> exit
```

Exit all shells (exiting Nu)

```
> exit --now
```

explode

version: 0.74.0

usage:

creates an explode expression

Signature

> explode

Examples

```
>
```

explore

version: 0.74.0

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

Notes

```
Press `:` then `h` to get a help menu.
```

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 -i
```

Explore a JSON file, then save the last visited sub-structure to a file

```
> open file.json | explore -p | to json | save part.json
```

export-env

version: 0.74.0

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

Examples

Set an environment variable

```
> export-env { let-env SPAM = 'eggs' }
```

Set an environment variable and examine its value

```
> export-env { let-env SPAM = 'eggs' }; $env.SPAM
```

export

version: 0.74.0

usage:

Export definitions or environment variables from a module.

Signature

```
> export
```

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Export a definition from a module

```
> module utils { export def my-command [] { "hello" } };
use utils my-command; my-command
```

export alias

version: 0.74.0

usage:

Define an alias and export it from a module

Signature

`> export alias (name) (initial_value)`

Parameters

- `name`: name of the alias
- `initial_value`: equals sign followed by value

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

export an alias of ll to ls -l, from a module

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

export def-env

version: 0.74.0

usage:

Define a custom command that participates in the environment and export it from a module

Signature

`> export def-env (name) (params) (block)`

Parameters

- `name`: definition name
- `params`: parameters
- `block`: body of the definition

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html

=== EXTRA NOTE ===
All blocks are scoped, including variable definition and
environment variable changes.

Because of this, the following doesn't work:

export def-env cd_with_fallback [arg = ""] {
    let fall_back_path = "/tmp"
    if $arg != "" {
        cd $arg
    } else {
        cd $fall_back_path
```

```
    }
}

Instead, you have to use cd in the top level scope:

export def-env cd_with_fallback [arg = ""] {
    let fall_back_path = "/tmp"
    let path = if $arg != "" {
        $arg
    } else {
        $fall_back_path
    }
    cd $path
}
```

Examples

Define a custom command that participates in the environment in a module and call it

```
> module foo { export def-env bar [] { let-env FOO_BAR
= "BAZ" } }; use foo bar; bar; $env.FOO_BAR
```

export def

version: 0.74.0

usage:

Define a custom command and export it from a module

Signature

`> export def (name) (params) (block)`

Parameters

- `name`: definition name
- `params`: parameters

- `block`: body of the definition

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Define a custom command in a module and call it

```
> module spam { export def foo [] { "foo" } }; use spam
foo; foo
```

export extern

version: 0.74.0

usage:

Define an extern and export it from a module

Signature

`> export extern (def_name) (params)`

Parameters

- `def_name`: definition name
- `params`: parameters

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Export the signature for an external command

```
> export extern echo [text: string]
```

export use

version: 0.74.0

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

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Re-export a command from another module

```
> module spam { export def foo [] { "foo" } }
    module eggs { export use spam foo }
    use eggs foo
    foo
```

expr-not

version: 0.74.0

usage:

creates a not expression

Signature

```
> expr-not
```

Examples

Creates a not expression

```
> (col a) > 2) | expr-not
```

extern

version: 0.74.0

usage:

Define a signature for an external command

Signature

```
> extern (def_name) (params)
```

Parameters

- `def_name`: definition name
- `params`: parameters

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Write a signature for an external command

```
> extern echo [text: string]
```

fetch

version: 0.74.0

usage:

collects the lazyframe to the selected rows

Signature

```
> fetch
```

Examples

Fetch a rows from the dataframe

```
> [[a b]; [6 2] [4 2] [2 2]] | into df | fetch 2
```

fetch

version: 0.74.0

usage:

Fetch the contents from a URL.

Signature

```
> fetch (URL) --user --password --timeout --headers --raw
```

Parameters

- `URL`: the URL to fetch the contents from
- `--user {any}`: the username when authenticating
- `--password {any}`: the password when authenticating
- `--timeout {int}`: timeout period in seconds
- `--headers {any}`: custom headers you want to add
- `--raw`: fetch contents as text rather than a table

Notes

```
Performs HTTP GET operation.
```

Examples

Fetch content from example.com

```
> fetch https://www.example.com
```

Fetch content from example.com, with username and password

```
> fetch -u myuser -p mypass https://www.example.com
```

Fetch content from example.com, with custom header

```
> fetch -H [my-header-key my-header-value] https://www.
example.com
```

fill-nan

version: 0.74.0

usage:

Replaces NaN values with the given expression

Signature

> fill-nan

Examples

Fills the NaN values with 0

```
> [1 2 NaN 3 NaN] | into df | fill-nan 0
```

Fills the NaN values of a whole dataframe

```
> [[a b]; [0.2 1] [0.1 NaN]] | into df | fill-nan 0
```

fill-null

version: 0.74.0

usage:

Replaces NULL values with the given expression

Signature

> fill-null

Examples

Fills the null values by 0

```
> [1 2 2 3 3] | into df | shift 2 | fill-null 0
```

filter-with

version: 0.74.0

usage:

Filters dataframe using a mask or expression as reference

Signature

```
> filter-with
```

Examples

Filter dataframe using a bool mask

```
> let mask = ([true false] | into df);
    [[a b]; [1 2] [3 4]] | into df | filter-with $mask
```

Filter dataframe using an expression

```
> [[a b]; [1 2] [3 4]] | into df | filter-with ((col a)
 > 1)
```

filter

version: 0.74.0

usage:

Filter values based on a predicate closure.

Signature

```
> filter (closure)
```

Parameters

- `closure`: Predicate closure

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.
```

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

version: 0.74.0

usage:

Filter dataframe based in expression

Signature

```
> filter
```

Examples

Filter dataframe using an expression

```
> [[a b]; [6 2] [4 2] [2 2]] | into df | filter ((col a)
 >= 4)
```

find

version: 0.74.0

usage:

Searches terms in the input.

Signature

`> find ...rest --regex --ignore-case --multiline --dotall --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)
- `--invert`: invert the match

Examples

Search for multiple terms in a command output

```
> ls | find toml md sh
```

Search 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

```
> [[version name]; [0.1.0 nushell] [0.1.1 fish] [0.2.0
zsh]] | find -r "nu"
```

first

version: 0.74.0

usage:

Show only the first number of rows.

Signature

```
> first
```

Examples

Return the first row of a dataframe

```
> [[a b]; [1 2] [3 4]] | into df | first
```

Return the first two rows of a dataframe

```
> [[a b]; [1 2] [3 4]] | into df | first 2
```

first

version: 0.74.0

usage:

creates a first expression

Signature

```
> first
```

Examples

Creates a first expression from a column

```
> col a | first
```

first

version: 0.74.0

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

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: 0.74.0

usage:

creates a flatten expression

Signature

`> flatten`

Examples

```
>
```

flatten

version: 0.74.0

usage:

Flatten the table.

Signature

`> flatten ...rest --all`

Parameters

- `...rest`: optionally flatten data by column
- `--all`: flatten inner table out

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: 0.74.0

usage:

Format a number

Signature

> fmt

Examples

Get a record containing multiple formats for the number 42

```
> 42 | fmt
```

for

version: 0.74.0

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`: returned a numbered item ($it.index and $it.item)

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Echo 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 echo a message

```
> for $it in ['bob' 'fred'] --numbered { print $"($it.index)
  is ($it.item)" }
```

format

version: 0.74.0

usage:

Format columns into a string using a simple pattern.

Signature

`> format (pattern)`

Parameters

- `pattern`: the pattern to output. e.g.) "{foo}: {bar}"

Examples

Print filenames with their sizes

```
> ls | format '{name}: {size}'
```

Print elements from some columns of a table

```
> [[col1, col2]; [v1, v2] [v3, v4]] | format '{col2}'
```

format filesize

version: 0.74.0

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

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

```
> 4Gb | format filesize MB
```

from

version: 0.74.0

usage:

Parse a string or binary data into structured data

Signature

`> from`

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

from csv

version: 0.74.0

usage:

Parse text as .csv and create table.

Signature

`> from csv --separator --noheaders --no-infer --trim`

Parameters

- `--separator {string}`: a character to separate columns, defaults to ‘,’
- `--noheaders`: don’t treat the first row as column names
- `--no-infer`: no field type inferencing
- `--trim {string}`: drop leading and trailing whitespaces around headers names and/or field values

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 comma-separated data to a table, ignoring headers

```
> open data.txt | from csv -n
```

Convert semicolon-separated data to a table

```
> open data.txt | from csv --separator ';'
```

Convert semicolon-separated data to a table, dropping all possible whitespaces around header names and field values

```
> open data.txt | from csv --trim all
```

Convert semicolon-separated data to a table, dropping all possible whitespaces around header names

```
> open data.txt | from csv --trim headers
```

Convert semicolon-separated data to a table, dropping all possible whitespaces around field values

```
> open data.txt | from csv --trim fields
```

from eml

version: 0.74.0

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

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: 0.74.0

usage:

Parse text as .ics and create table.

Signature

`> from ics`

Examples

Converts ics formatted string to table

```
> 'BEGIN:VCALENDAR
END:VCALENDAR' | from ics
```

from ini

version: 0.74.0

usage:

Parse text as .ini and create record

Signature

`> from ini`

Examples

Converts ini formatted string to record

```
> '[foo]
a=1
b=2' | from ini
```

from json

version: 0.74.0

usage:

Convert from json to structured data

Signature

`> from json --objects`

Parameters

- `--objects`: treat each line as a separate value

Examples

Converts json formatted string to table

```
> '{ "a": 1 }' | from json
```

Converts json formatted string to table

```
> '{ "a": 1, "b": [1, 2] }' | from json
```

from nuon

version: 0.74.0

usage:

Convert from nuon to structured data

Signature

`> from nuon`

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: 0.74.0

usage:

Parse OpenDocument Spreadsheet(.ods) data and create table.

Signature

`> from ods --sheets`

Parameters

- `--sheets {list<string>}`: Only convert specified sheets

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 -s [Spreadsheet1]
```

from ssv

version: 0.74.0

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

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 -n
```

from toml

version: 0.74.0

usage:

Parse text as .toml and create record.

Signature

```
> from toml
```

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: 0.74.0

usage:

Parse text as .tsv and create table.

Signature

```
> from tsv --noheaders --no-infer --trim
```

Parameters

- `--noheaders`: don't treat the first row as column names
- `--no-infer`: no field type inferencing
- `--trim {string}`: drop leading and trailing whitespaces around headers names and/or field values

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 -n
```

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: 0.74.0

usage:

Parse url-encoded string as a record.

Signature

```
> from url
```

Examples

Convert url encoded string into a record

```
> 'bread=baguette&cheese=comt%C3%A9&meat=ham&fat=butter'
| from url
```

from vcf

version: 0.74.0

usage:

Parse text as .vcf and create table.

Signature

```
> from vcf
```

Examples

Converts ics formatted string to table

```
> 'BEGIN:VCARD
N:Foo
FN:Bar
EMAIL:foo@bar.com
END:VCARD' | from vcf
```

from xlsx

version: 0.74.0

usage:

Parse binary Excel(.xlsx) data and create table.

Signature

`> from xlsx --sheets`

Parameters

- `--sheets {list<string>}`: Only convert specified sheets

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 -s [Spreadsheet1]
```

from xml

version: 0.74.0

usage:

Parse text as .xml and create record.

Signature

> from xml

Examples

Converts xml formatted string to record

```
> '<?xml version="1.0" encoding="UTF-8"?>
<note>
  <remember>Event</remember>
</note>' | from xml
```

from yaml

version: 0.74.0

usage:

Parse text as .yaml/.yml and create table.

Signature

> from yaml

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: 0.74.0

usage:

Parse text as .yaml/.yml and create table.

Signature

```
> from yml
```

Examples

Converts yaml formatted string to table

```
> 'a: 1' | from yaml
```

Converts yaml formatted string to table

```
> '[ a: 1, b: [1, 2] ]' | from yaml
```

g

version: 0.74.0

usage:

Switch to a given shell, or list all shells if no given shell number.

Signature

```
> g (shell_number)
```

Parameters

- `shell_number`: shell number to change to

Examples

Lists all open shells

```
> g
```

Make two directories and enter new shells for them, use `g` to jump to the specific shell

```
> mkdir foo bar; enter foo; enter ../bar; g 1
```

Use `shells` to show all the opened shells and run `g 2` to jump to the third one

```
> shells; g 2
```

Make two directories and enter new shells for them, use `g -` to jump to the last used shell

```
> mkdir foo bar; enter foo; enter ../bar; g -
```

get-day

version: 0.74.0

usage:

Gets day from date

Signature

```
> get-day
```

Examples

Returns day from a date

```
> let dt = ('2020-08-04T16:39:18+00:00' | into datetime
-z 'UTC');
    let df = ([$dt $dt] | into df);
```

```
    $df | get-day
```

get-hour

version: 0.74.0

usage:

Gets hour from date

Signature

```
> get-hour
```

Examples

Returns hour from a date

```
> let dt = ('2020-08-04T16:39:18+00:00' | into datetime
-z 'UTC');
    let df = ([$dt $dt] | into df);
    $df | get-hour
```

get-minute

version: 0.74.0

usage:

Gets minute from date

Signature

```
> get-minute
```

Examples

Returns minute from a date

```
> let dt = ('2020-08-04T16:39:18+00:00' | into datetime
-z 'UTC');
    let df = ([$dt $dt] | into df);
    $df | get-minute
```

get-month

version: 0.74.0

usage:

Gets month from date

Signature

`> get-month`

Examples

Returns month from a date

```
> let dt = ('2020-08-04T16:39:18+00:00' | into datetime
-z 'UTC');
    let df = ([$dt $dt] | into df);
    $df | get-month
```

get-nanosecond

version: 0.74.0

usage:

Gets nanosecond from date

Signature

`> get-nanosecond`

Examples

Returns nanosecond from a date

```
> let dt = ('2020-08-04T16:39:18+00:00' | into datetime
-z 'UTC');
    let df = ([$dt $dt] | into df);
    $df | get-nanosecond
```

get-ordinal

version: 0.74.0

usage:

Gets ordinal from date

Signature

```
> get-ordinal
```

Examples

Returns ordinal from a date

```
> let dt = ('2020-08-04T16:39:18+00:00' | into datetime
-z 'UTC');
    let df = ([$dt $dt] | into df);
    $df | get-ordinal
```

get-second

version: 0.74.0

usage:

Gets second from date

Signature

```
> get-second
```

Examples

Returns second from a date

```
> let dt = ('2020-08-04T16:39:18+00:00' | into datetime
-z 'UTC');
    let df = ([$dt $dt] | into df);
    $df | get-second
```

get-week

version: 0.74.0

usage:

Gets week from date

Signature

```
> get-week
```

Examples

Returns week from a date

```
> let dt = ('2020-08-04T16:39:18+00:00' | into datetime
-z 'UTC');
    let df = ([$dt $dt] | into df);
    $df | get-week
```

get-weekday

version: 0.74.0

usage:

Gets weekday from date

Signature

```
> get-weekday
```

Examples

Returns weekday from a date

```
> let dt = ('2020-08-04T16:39:18+00:00' | into datetime
-z 'UTC');
    let df = ([$dt $dt] | into df);
    $df | get-weekday
```

get-year

version: 0.74.0

usage:

Gets year from date

Signature

```
> get-year
```

Examples

Returns year from a date

```
> let dt = ('2020-08-04T16:39:18+00:00' | into datetime
-z 'UTC');
    let df = ([$dt $dt] | into df);
    $df | get-year
```

get

version: 0.74.0

usage:

Creates dataframe with the selected columns

Signature

```
> get
```

Examples

Returns the selected column

```
> [[a b]; [1 2] [3 4]] | into df | get a
```

get

version: 0.74.0

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`: when there are empty cells, instead of erroring out, replace them with nothing
- `--sensitive`: get path in a case sensitive manner

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 -s Path
```

glob

version: 0.74.0

usage:

Creates a list of files and/or folders based on the glob pattern provided.

Signature

`> glob (glob) --depth`

Parameters

- `glob`: the glob expression
- `--depth {int}`: directory depth to search

Notes

> For more glob pattern help, please refer to https://github.com/olson-sean-k/wax

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 and 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>
```

grid

version: 0.74.0

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

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.
```

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
```

group-by

version: 0.74.0

usage:

Splits a list or table into groups, and returns a record containing those groups.

Signature

```
> group-by (grouper)
```

Parameters

- `grouper`: the grouper value to use

Examples

Group items by the "type" column's values

```
> ls | group-by type
```

You can also group by raw values by leaving out the argument

```
> ['1' '3' '1' '3' '2' '1' '1'] | group-by
```

group-by

version: 0.74.0

usage:

Creates a group-by object that can be used for other aggregations

Signature

`> group-by`

Examples

Group by and perform an aggregation

```
> [[a b]; [1 2] [1 4] [2 6] [2 4]]
    | into df
    | group-by a
    | agg [
        (col b | min | as "b_min")
        (col b | max | as "b_max")
        (col b | sum | as "b_sum")
     ]
```

Group by and perform an aggregation

```
> [[a b]; [1 2] [1 4] [2 6] [2 4]]
    | into lazy
    | group-by a
    | agg [
        (col b | min | as "b_min")
        (col b | max | as "b_max")
        (col b | sum | as "b_sum")
     ]
    | collect
```

group

version: 0.74.0

usage:

Groups input into groups of `group_size`.

Signature

`> group (group_size)`

Parameters

- `group_size`: the size of each group

Examples

Group the a list by pairs

```
> [1 2 3 4] | group 2
```

gstat

version: 0.74.0

usage:

Get the git status of a repo

Signature

```
> gstat
```

hash

version: 0.74.0

usage:

Apply hash function.

Signature

```
> hash
```

Notes

> ```
> You must use one of the following subcommands. Using this
> command as-is will only produce this help message.
> ```

hash base64

version: 0.74.0

usage:

Deprecated command

Signature

```
> hash base64
```

hash md5

version: 0.74.0

usage:

Hash a value using the md5 hash algorithm

Signature

`> hash md5 ...rest --binary`

Parameters

- `...rest`: optionally md5 hash data by cell path
- `--binary`: Output binary instead of hexadecimal representation

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: 0.74.0

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

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: 0.74.0

usage:

Use the first row of the table as column names.

Signature

`> headers`

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: 0.74.0

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

Notes

```
`help word` searches for "word" in commands, aliases and
modules, in that order.
```

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
```

help aliases

version: 0.74.0

usage:

Show help on nushell aliases.

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

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: 0.74.0

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

help modules

version: 0.74.0

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

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.
```

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
```

help operators

version: 0.74.0

usage:

Show help on nushell operators.

Signature

```
> help operators
```

hide-env

version: 0.74.0

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

Examples

Hide an environment variable

```
> let-env HZ_ENV_ABC = 1; hide-env HZ_ENV_ABC; 'HZ_ENV_ABC' in (env).name
```

hide

version: 0.74.0

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

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
```

Examples

Hide the alias just defined

```
> alias lll = ls -l; hide lll
```

Hide a custom command

```
> def say-hi [] { echo 'Hi!' }; hide say-hi
```

histogram

version: 0.74.0

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 just 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'

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: 0.74.0

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

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 | wrap cmd | where cmd =~ cargo
```

history session

version: 0.74.0

usage:

Get the command history session

Signature

`> history session`

Examples

Get current history session

```
> history session
```

if

version: 0.74.0

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 if check fails

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

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: 0.74.0

usage:

Ignore the output of the previous command in the pipeline

Signature

```
> ignore
```

Examples

Ignore the output of an echo command

```
> echo done | ignore
```

inc

version: 0.74.0

usage:

Increment a value or version. Optionally use the column of a table.

Signature

```
> inc
```

input

version: 0.74.0

usage:

Get input from the user.

Signature

`> input (prompt) --bytes-until --suppress-output`

Parameters

- `prompt`: prompt to show the user
- `--bytes-until {string}`: read bytes (not text) until a stop byte
- `--suppress-output`: don't print keystroke values

Examples

Get input from the user, and assign to a variable

```
> let user_input = (input)
```

insert

version: 0.74.0

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)

Examples

Insert a new entry into a single record

```
> {'name': 'nu', 'stars': 5} | insert alias 'Nushell'
```

Insert a column with values equal to their row index, plus the value of 'foo' in each row

```
> [[foo]; [7] [8] [9]] | insert bar {|el ind| $el.foo +
$ind }
```

into

version: 0.74.0

usage:

Commands to convert data from one type to another.

Signature

`> into`

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

into binary

version: 0.74.0

usage:

Convert value to a binary primitive

Signature

```
> into binary ...rest
```

Parameters

- `...rest`: for a data structure input, convert data at the given cell paths

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

```
> 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 decimal to a nushell binary primitive

```
> 1.234 | into binary
```

into bool

version: 0.74.0

usage:

Convert value to boolean

Signature

`> into bool ...rest`

Parameters

- `...rest`: for a data structure input, convert data at the given cell paths

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 integer to boolean

```
> 1 | into bool
```

convert decimal to boolean

```
> 0.3 | into bool
```

convert decimal string to boolean

```
> '0.0' | into bool
```

convert string to boolean

```
> 'true' | into bool
```

into datetime

version: 0.74.0

usage:

Convert text into a datetime

Signature

`> into datetime ...rest --timezone --offset --format --list`

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 an expected format for parsing strings to datetimes. Use --list to see all possible options
- `--list`: Show all possible variables for use with the --format flag

Examples

Convert to datetime

```
> '27.02.2021 1:55 pm +0000' | into datetime
```

Convert to datetime

```
> '2021-02-27T13:55:40+00:00' | into datetime
```

Convert to datetime using a custom format

```
> '20210227_135540+0000' | into datetime -f '%Y%m%d_%H%M%S%z'
```

Convert timestamp (no larger than 8e+12) to a UTC datetime

```
> 1614434140 | into datetime
```

Convert timestamp (no larger than 8e+12) to datetime using a specified timezone offset (between -12 and 12)

```
> 1614434140 | into datetime -o +9
```

Convert a millisecond-precise timestamp

```
> 1656165681720 | into datetime
```

into decimal

version: 0.74.0

usage:

Convert text into a decimal

Signature

`> into decimal ...rest`

Parameters

- `...rest`: for a data structure input, convert data at the given cell paths

Examples

Convert string to decimal in table

```
> [[num]; ['5.01']] | into decimal num
```

Convert string to decimal

```
> '1.345' | into decimal
```

Convert decimal to decimal

```
> '-5.9' | into decimal
```

Convert boolean to decimal

```
> true | into decimal
```

into df

version: 0.74.0

usage:

Converts a list, table or record into a dataframe

Signature

```
> into df
```

Examples

Takes a dictionary and creates a dataframe

```
> [[a b];[1 2] [3 4]] | into df
```

Takes a list of tables and creates a dataframe

```
> [[1 2 a] [3 4 b] [5 6 c]] | into df
```

Takes a list and creates a dataframe

```
> [a b c] | into df
```

Takes a list of booleans and creates a dataframe

```
> [true true false] | into df
```

into duration

version: 0.74.0

usage:

Convert value to duration

Signature

`> into duration ...rest --convert`

Parameters

- `...rest`: for a data structure input, convert data at the given cell paths
- `--convert {string}`: convert duration into another duration

Notes

```
This command does not take leap years into account, and
every month is assumed to have 30 days.
```

Examples

Convert string to duration in table

```
> [[value]; ['1sec'] ['2min'] ['3hr'] ['4day'] ['5wk']]
| into duration value
```

Convert string to duration

```
> '7min' | into duration
```

Convert string to the requested duration as a string

```
> '7min' | into duration --convert sec
```

Convert duration to duration

```
> 420sec | into duration
```

Convert duration to the requested duration as a string

```
> 420sec | into duration --convert ms
```

into filesize

version: 0.74.0

usage:

Convert value to filesize

Signature

`> into filesize ...rest`

Parameters

- `...rest`: for a data structure input, convert data at the given cell paths

Examples

Convert string to filesize in table

```
> [[bytes]; ['5'] [3.2] [4] [2kb]] | into filesize bytes
```

Convert string to filesize

```
> '2' | into filesize
```

Convert decimal to filesize

```
> 8.3 | into filesize
```

Convert int to filesize

```
> 5 | into filesize
```

Convert file size to filesize

```
> 4KB | into filesize
```

into int

version: 0.74.0

usage:

Convert value to integer

Signature

`> into int ...rest --radix --little-endian`

Parameters

- `...rest`: for a data structure input, convert data at the given cell paths
- `--radix {number}`: radix of integer
- `--little-endian`: use little-endian byte decoding

Examples

Convert string to integer in table

```
> [[num]; ['-5'] [4] [1.5]] | into int num
```

Convert string to integer

```
> '2' | into int
```

Convert decimal to integer

```
> 5.9 | into int
```

Convert decimal string to integer

```
> '5.9' | into int
```

Convert file size to integer

```
> 4KB | into int
```

Convert bool to integer

```
> [false, true] | into int
```

Convert date to integer (Unix timestamp)

```
> 2022-02-02 | into int
```

Convert to integer from binary

```
> '1101' | into int -r 2
```

Convert to integer from hex

```
> 'FF' |  into int -r 16
```

Convert octal string to integer

```
> '0o10132' | into int
```

Convert 0 padded string to integer

```
> '0010132' | into int
```

Convert 0 padded string to integer with radix

```
> '0010132' | into int -r 8
```

into lazy

version: 0.74.0

usage:

Converts a dataframe into a lazy dataframe

Signature

`> into lazy`

Examples

Takes a dictionary and creates a lazy dataframe

```
> [[a b];[1 2] [3 4]] | into lazy
```

into nu

version: 0.74.0

usage:

Converts a section of the dataframe into nushell Table

Signature

`> into nu`

Examples

Shows head rows from dataframe

```
> [[a b]; [1 2] [3 4]] | into df | into nu
```

Shows tail rows from dataframe

```
> [[a b]; [1 2] [5 6] [3 4]] | into df | into nu -t -n
1
```

into nu

version: 0.74.0

usage:

Convert expression into a nu value for access and exploration

Signature

```
> into nu
```

Examples

Convert a col expression into a nushell value

```
> col a | into nu
```

into record

version: 0.74.0

usage:

Convert value to record

Signature

```
> into 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

```
> -500day | 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: 0.74.0

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

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
```

Convert a variety of values in table literal form into a SQLite database

```
> [one 2 5.2 six true 100mib 25sec] | into sqlite variety.
db
```

into string

version: 0.74.0

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

Examples

convert integer to string and append three decimal places

```
> 5 | into string -d 3
```

convert decimal to string and round to nearest integer

```
> 1.7 | into string -d 0
```

convert decimal to string

```
> 1.7 | into string -d 1
```

convert decimal to string and limit to 2 decimals

```
> 1.734 | into string -d 2
```

try to convert decimal to string and provide negative decimal points

```
> 1.734 | into string -d -2
```

convert decimal to string

```
> 4.3 | into string
```

convert string to string

```
> '1234' | into string
```

convert boolean to string

```
> true | into string
```

convert filepath to string

```
> ls Cargo.toml | get name | into string
```

convert filesize to string

```
> 1KiB | into string
```

is-admin

version: 0.74.0

usage:

Check if nushell is running with administrator or root privileges

Signature

```
> is-admin
```

Examples

Return 'iamroot' if nushell is running with admin/root privileges, and 'iamnotroot' if not.

```
> if is-admin { "iamroot" } else { "iamnotroot" }
```

is-duplicated

version: 0.74.0

usage:

Creates mask indicating duplicated values

Signature

```
> is-duplicated
```

Examples

Create mask indicating duplicated values

```
> [5 6 6 6 8 8 8] | into df | is-duplicated
```

Create mask indicating duplicated rows in a dataframe

```
> [[a, b]; [1 2] [1 2] [3 3] [3 3] [1 1]] | into df | is-
duplicated
```

is-empty

version: 0.74.0

usage:

Check for empty values.

Signature

```
> is-empty ...rest
```

Parameters

- `...rest`: the names of the columns to check emptiness

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-in

version: 0.74.0

usage:

Checks if elements from a series are contained in right series

Signature

```
> is-in
```

Examples

Checks if elements from a series are contained in right series

```
> let other = ([1 3 6] | into df);
    [5 6 6 6 8 8 8] | into df | is-in $other
```

is-in

version: 0.74.0

usage:

Creates an is-in expression

Signature

```
> is-in
```

Examples

Creates a is-in expression

```
> let df = ([[a b]; [one 1] [two 2] [three 3]] | into df)
;
    $df | with-column (col a | is-in [one two] | as a_-
in)
```

is-not-null

version: 0.74.0

usage:

Creates mask where value is not null

Signature

```
> is-not-null
```

Examples

Create mask where values are not null

```
> let s = ([5 6 0 8] | into df);
    let res = ($s / $s);
    $res | is-not-null
```

is-not-null

version: 0.74.0

usage:

creates a is not null expression

Signature

```
> is-not-null
```

Examples

Creates a is not null expression from a column

```
> col a | is-not-null
```

is-null

version: 0.74.0

usage:

Creates mask where value is null

Signature

```
> is-null
```

Examples

Create mask where values are null

```
> let s = ([5 6 0 8] | into df);
    let res = ($s / $s);
    $res | is-null
```

is-null

version: 0.74.0

usage:

creates a is null expression

Signature

```
> is-null
```

Examples

Creates a is null expression from a column

```
> col a | is-null
```

is-unique

version: 0.74.0

usage:

Creates mask indicating unique values

Signature

```
> is-unique
```

Examples

Create mask indicating unique values

```
> [5 6 6 6 8 8 8] | into df | is-unique
```

Create mask indicating duplicated rows in a dataframe

```
> [[a, b]; [1 2] [1 2] [3 3] [3 3] [1 1]] | into df | is-
unique
```

join

version: 0.74.0

usage:

Joins a lazy frame with other lazy frame

Signature

```
> join
```

Examples

Join two lazy dataframes

```
> let df_a = ([[a b c];[1 "a" 0] [2 "b" 1] [1 "c" 2] [1
"c" 3]] | into lazy);
    let df_b = ([["foo" "bar" "ham"];[1 "a" "let"] [2 "c"
"var"] [3 "c" "const"]] | into lazy);
    $df_a | join $df_b a foo | 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]] | into df);
    let df_b = ([["foo" "bar" "ham"];[1 "a" "let"] [2 "c"
"var"] [3 "c" "const"]] | into lazy);
    $df_a | join $df_b a foo
```

keybindings

version: 0.74.0

usage:

Keybindings related commands

Signature

`> keybindings`

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

keybindings default

version: 0.74.0

usage:

List default keybindings

Signature

> keybindings default

Examples

Get list with default keybindings

```
> keybindings default
```

keybindings list

version: 0.74.0

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

Examples

Get list of key modifiers

```
> keybindings list -m
```

Get list of reedline events and edit commands

```
> keybindings list -e -d
```

Get list with all the available options

```
> keybindings list
```

keybindings listen

version: 0.74.0

usage:

Get input from the user.

Signature

`> keybindings listen`

Examples

Type and see key event codes

```
> keybindings listen
```

kill

version: 0.74.0

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)

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: 0.74.0

usage:

Creates new dataframe with tail rows or creates a last expression

Signature

`> last`

Examples

Create new dataframe with last rows

```
> [[a b]; [1 2] [3 4]] | into df | last 1
```

last

version: 0.74.0

usage:

creates a last expression

Signature

`> last`

Examples

Creates a last expression from a column

```
> col a | last
```

last

version: 0.74.0

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

Examples

Get the last 2 items

```
> [1,2,3] | last 2
```

Get the last item

```
> [1,2,3] | last
```

length

version: 0.74.0

usage:

Count the number of elements in the input.

Signature

`> length --column`

Parameters

- `--column`: Show the number of columns in a table

Examples

Count the number of items in a list

```
> [1 2 3 4 5] | length
```

Count the number of columns in a table

```
> [{columnA: A0 columnB: B0}] | length -c
```

let-env

version: 0.74.0

usage:

Create an environment variable and give it a value.

Signature

> let-env (var_name) (initial_value)

Parameters

- `var_name`: variable name
- `initial_value`: equals sign followed by value

Examples

Create an environment variable and display it

```
> let-env MY_ENV_VAR = 1; $env.MY_ENV_VAR
```

let

version: 0.74.0

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

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

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 }
```

lines

version: 0.74.0

usage:

Converts input to lines

Signature

`> lines --skip-empty`

Parameters

- `--skip-empty`: skip empty lines

Examples

Split multi-line string into lines

```
> $"two\nlines" | lines
```

list

version: 0.74.0

usage:

Aggregates a group to a Series

Signature

```
> list
```

Examples

```
>
```

lit

version: 0.74.0

usage:

Creates a literal expression

Signature

```
> lit
```

Examples

Created a literal expression and converts it to a nu object

```
> lit 2 | into nu
```

load-env

version: 0.74.0

usage:

Loads an environment update from a record.

Signature

`> load-env (update)`

Parameters

- `update`: the record to use for updates

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: 0.74.0

usage:

Run a block in a loop.

Signature

`> loop (block)`

Parameters

- `block`: block to loop

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Loop while a condition is true

```
> mut x = 0; loop { if $x > 10 { break }; $x = $x + 1 };
$x
```

lowercase

version: 0.74.0

usage:

Lowercase the strings in the column

Signature

```
> lowercase
```

Examples

Modifies strings to lowercase

```
> [Abc aBc abC] | into df | lowercase
```

ls-df

version: 0.74.0

usage:

Lists stored dataframes

Signature

```
> ls-df
```

Examples

Creates a new dataframe and shows it in the dataframe list

```
> let test = ([[a b];[1 2] [3 4]] | into df);
    ls-df
```

ls

version: 0.74.0

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)

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
```

math

version: 0.74.0

usage:

Use mathematical functions as aggregate functions on a list of numbers or tables.

Signature

> math

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

math abs

version: 0.74.0

usage:

Returns the absolute value of a number

Signature

> math abs

Examples

Compute absolute value of each number in a list of numbers

```
> [-50 -100.0 25] | math abs
```

math arccos

version: 0.74.0

usage:

Returns the arccosine of the number.

Signature

`> math arccos --degrees`

Parameters

- `--degrees`: Return degrees instead of radians

Examples

Get the arccosine of 1

```
> 1 | math arccos
```

Get the arccosine of -1 in degrees

```
> -1 | math arccos -d
```

math arccosh

version: 0.74.0

usage:

Returns the inverse of the hyperbolic cosine function.

Signature

`> math arccosh`

Examples

Get the arccosh of 1

```
> 1 | math arccosh
```

math arcsin

version: 0.74.0

usage:

Returns the arcsine of the number.

Signature

> math arcsin --degrees

Parameters

- `--degrees`: Return degrees instead of radians

Examples

Get the arcsine of 1

```
> 1 | math arcsin
```

Get the arcsine of 1 in degrees

```
> 1 | math arcsin -d
```

math arcsinh

version: 0.74.0

usage:

Returns the inverse of the hyperbolic sine function.

Signature

> math arcsinh

Examples

Get the arcsinh of 0

```
> 0 | math arcsinh
```

math arctan

version: 0.74.0

usage:

Returns the arctangent of the number.

Signature

`> math arctan --degrees`

Parameters

- `--degrees`: Return degrees instead of radians

Examples

Get the arctangent of 1

```
> 1 | math arctan
```

Get the arctangent of -1 in degrees

```
> -1 | math arctan -d
```

math arctanh

version: 0.74.0

usage:

Returns the inverse of the hyperbolic tangent function.

Signature

`> math arctanh`

Examples

Get the arctanh of 1

```
> 1 | math arctanh
```

math avg

version: 0.74.0

usage:

Returns the average of a list of numbers

Signature

`> math avg`

Examples

Compute the average of a list of numbers

```
> [-50 100.0 25] | math avg
```

math ceil

version: 0.74.0

usage:

Returns the ceil of a number (smallest integer greater than or equal to that number)

Signature

`> math ceil`

Examples

Apply the ceil function to a list of numbers

```
> [1.5 2.3 -3.1] | math ceil
```

math cos

version: 0.74.0

usage:

Returns the cosine of the number.

Signature

`> math cos --degrees`

Parameters

- `--degrees`: Use degrees instead of radians

Examples

Apply the cosine to pi

```
> math pi | math cos
```

Apply the cosine to a list of angles in degrees

```
> [0 90 180 270 360] | math cos -d
```

math cosh

version: 0.74.0

usage:

Returns the hyperbolic cosine of the number.

Signature

```
> math cosh
```

Examples

Apply the hyperpolic cosine to 1

```
> 1 | math cosh
```

math e

version: 0.74.0

usage:

Returns the mathematical constant e (exp(1)/'1 | math exp').

Signature

```
> math e
```

Examples

Get the first three decimal digits of e

```
> math e | math round --precision 3
```

math eval

version: 0.74.0

usage:

Deprecated command

Signature

```
> math eval
```

math floor

version: 0.74.0

usage:

Returns the floor of a number (largest integer less than or equal to that number)

Signature

`> math floor`

Examples

Apply the floor function to a list of numbers

```
> [1.5 2.3 -3.1] | math floor
```

math ln

version: 0.74.0

usage:

Returns the natural logarithm. Base: (math e)

Signature

`> math ln`

Examples

Get the natural logarithm of e

```
> math e | math ln
```

math log

version: 0.74.0

usage:

Returns the logarithm for an arbitrary base.

Signature

`> math log (base)`

Parameters

- `base`: Base for which the logarithm should be computed

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: 0.74.0

usage:

Returns the maximum of a list of numbers, or of columns in a table

Signature

`> math max`

Examples

Find the maximum of 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
```

math median

version: 0.74.0

usage:

Computes the median of a list of numbers

Signature

```
> math median
```

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
```

math min

version: 0.74.0

usage:

Finds the minimum within a list of numbers or tables

Signature

```
> math min
```

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
```

math mode

version: 0.74.0

usage:

Returns the most frequent element(s) from a list of numbers or tables

Signature

`> math mode`

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 pi

version: 0.74.0

usage:

Returns the mathematical constant .

Signature

> math pi

Examples

Get the first two decimal digits of

```
> math pi | math round --precision 2
```

math product

version: 0.74.0

usage:

Returns the product of a list of numbers or the products of each column of a table

Signature

> math product

Examples

Compute the product of a list of numbers

```
> [2 3 3 4] | math product
```

math round

version: 0.74.0

usage:

Returns the input number rounded to the specified precision

Signature

> math round --precision

Parameters

- `--precision {number}`: digits of precision

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 -p 2
```

math sin

version: 0.74.0

usage:

Returns the sine of the number.

Signature

`> math sin --degrees`

Parameters

- `--degrees`: Use degrees instead of radians

Examples

Apply the sine to /2

```
> (math pi) / 2 | math sin
```

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: 0.74.0

usage:

Returns the hyperbolic sine of the number.

Signature

`> math sinh`

Examples

Apply the hyperpolic sine to 1

```
> 1 | math sinh
```

math sqrt

version: 0.74.0

usage:

Returns the square root of the input number

Signature

`> math sqrt`

Examples

Compute the square root of each number in a list

```
> [9 16] | math sqrt
```

math stddev

version: 0.74.0

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)

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 -s
```

math sum

version: 0.74.0

usage:

Returns the sum of a list of numbers or of each column in a table

Signature

`> math sum`

Examples

Sum a list of numbers

```
> [1 2 3] | math sum
```

Get the disk usage for the current directory

```
> ls | get size | math sum
```

math tan

version: 0.74.0

usage:

Returns the tangent of the number.

Signature

`> math tan --degrees`

Parameters

- `--degrees`: Use degrees instead of radians

Examples

Apply the tangent to pi/4

```
> (math pi) / 4 | math tan
```

Apply the tangent to a list of angles in degrees

```
> [-45 0 45] | math tan -d
```

math tanh

version: 0.74.0

usage:

Returns the hyperbolic tangent of the number.

Signature

```
> math tanh
```

Examples

Apply the hyperpolic tangent to 10*pi

```
> (math pi) * 10 | math tanh
```

math tau

version: 0.74.0

usage:

Returns the mathematical constant .

Signature

```
> math tau
```

Examples

Compare   and

```
> (math tau) / 2
```

math variance

version: 0.74.0

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)

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 -s
```

max

version: 0.74.0

usage:

Creates a max expression

Signature

> max

Examples

Max aggregation for a group-by

```
> [[a b]; [one 2] [one 4] [two 1]]
    | into df
    | group-by a
    | agg (col b | max)
```

max

version: 0.74.0

usage:

Aggregates columns to their max value

Signature

```
> max
```

Examples

Max value from columns in a dataframe

```
> [[a b]; [6 2] [1 4] [4 1]] | into df | max
```

mean

version: 0.74.0

usage:

Creates a mean expression for an aggregation

Signature

```
> mean
```

Examples

Mean aggregation for a group-by

```
> [[a b]; [one 2] [one 4] [two 1]]
    | into df
    | group-by a
    | agg (col b | mean)
```

mean

version: 0.74.0

usage:

Aggregates columns to their mean value

Signature

> mean

Examples

Mean value from columns in a dataframe

```
> [[a b]; [6 2] [4 2] [2 2]] | into df | mean
```

median

version: 0.74.0

usage:

Creates a median expression for an aggregation

Signature

> median

Examples

Median aggregation for a group-by

```
> [[a b]; [one 2] [one 4] [two 1]]
    | into df
    | group-by a
    | agg (col b | median)
```

median

version: 0.74.0

usage:

Aggregates columns to their median value

Signature

```
> median
```

Examples

Median value from columns in a dataframe

```
> [[a b]; [6 2] [4 2] [2 2]] | into df | median
```

melt

version: 0.74.0

usage:

Unpivot a DataFrame from wide to long format

Signature

```
> melt
```

Examples

melt dataframe

```
> [[a b c d]; [x 1 4 a] [y 2 5 b] [z 3 6 c]] | into df
| melt -c [b c] -v [a d]
```

merge

version: 0.74.0

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

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.
```

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*'}]
```

metadata

version: 0.74.0

usage:

Get the metadata for items in the stream

Signature

> metadata (expression)

Parameters

- `expression`: the expression you want metadata for

Examples

Get the metadata of a variable

```
> let a = 42; metadata $a
```

Get the metadata of the input

```
> ls | metadata
```

min

version: 0.74.0

usage:

Creates a min expression

Signature

> min

Examples

Min aggregation for a group-by

```
> [[a b]; [one 2] [one 4] [two 1]]
    | into df
    | group-by a
```

```
    | agg (col b | min)
```

min

version: 0.74.0

usage:

Aggregates columns to their min value

Signature

`> min`

Examples

Min value from columns in a dataframe

```
> [[a b]; [6 2] [1 4] [4 1]] | into df | min
```

mkdir

version: 0.74.0

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).

Examples

Make a directory named foo

```
> mkdir foo
```

Make multiple directories and show the paths created

```
> mkdir -v foo/bar foo2
```

module

version: 0.74.0

usage:

Define a custom module

Signature

`> module (module_name) (block)`

Parameters

- `module_name`: module name
- `block`: body of the module

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

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 { let-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 [] { let-env FOO_BAR
= "BAZ" } }; use foo bar; bar; $env.FOO_BAR
```

move

version: 0.74.0

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

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: 0.74.0

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

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

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 }
```

mv

version: 0.74.0

usage:

Move files or directories.

Signature

`> mv (source) (destination) --verbose --force --interactive`

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

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
```

n-unique

version: 0.74.0

usage:

Counts unique values

Signature

`> n-unique`

Examples

Counts unique values

```
> [1 1 2 2 3 3 4] | into df | n-unique
```

n-unique

version: 0.74.0

usage:

creates a n-unique expression

Signature

`> n-unique`

Examples

Creates a is n-unique expression from a column

```
> col a | n-unique
```

n

version: 0.74.0

usage:

Switch to the next shell.

Signature

`> n`

Examples

Make two directories and enter new shells for them, use `n` to jump to the next shell

```
> mkdir foo bar; enter foo; enter ../bar; n
```

Run `n` several times and note the changes of current directory

```
> n
```

nu-check

version: 0.74.0

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

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 -d script.nu
```

Parse an external stream as script by showing error message

```
> open foo.nu | nu-check -d script.nu
```

Parse an internal stream as module by showing error message

```
> open module.nu | lines | nu-check -d --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 -ad
```

nu-highlight

version: 0.74.0

usage:

Syntax highlight the input string.

Signature

`> nu-highlight`

Examples

Describe the type of a string

```
> 'let x = 3' | nu-highlight
```

open-df

version: 0.74.0

usage:

Opens csv, json, arrow, or parquet file to create dataframe

Signature

`> open-df`

Examples

Takes a file name and creates a dataframe

```
> open test.csv
```

open

version: 0.74.0

usage:

Load a file into a cell, converting to table if possible (avoid by appending '--raw').

Signature

```
> open (filename) --raw
```

Parameters

- `filename`: the filename to use
- `--raw`: open file as raw binary

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
```

otherwise

version: 0.74.0

usage:

completes a when expression

Signature

```
> otherwise
```

Examples

Create a when conditions

```
> when ((col a) > 2) 4 | otherwise 5
```

Create a when conditions

```
> when ((col a) > 2) 4 | when ((col a) < 0) 6 | otherwise
0
```

Create a new column for the dataframe

```
> [[a b]; [6 2] [1 4] [4 1]]
    | into lazy
    | with-column (
        when ((col a) > 2) 4 | otherwise 5 | as c
      )
    | with-column (
        when ((col a) > 5) 10 | when ((col a) < 2) 6 | otherwise
0 | as d
      )
    | collect
```

overlay

version: 0.74.0

usage:

Commands for manipulating overlays.

Signature

> overlay

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.
```

overlay hide

version: 0.74.0

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

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

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 { let-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
```

overlay list

version: 0.74.0

usage:

List all active overlays

Signature

```
> overlay list
```

Notes

> The overlays are listed in the order they were activated.

Examples

Get the last activated overlay

```
> module spam { export def foo [] { "foo" } }
    overlay use spam
    overlay list | last
```

overlay new

version: 0.74.0

usage:

Create an empty overlay

Signature

```
> overlay new (name)
```

Parameters

- `name`: Name of the overlay

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
```

Examples

Create an empty overlay

```
> overlay new spam
```

overlay use

version: 0.74.0

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.

Notes

```
This command is a parser keyword. For details, check:
```

> https://www.nushell.sh/book/thinking_in_nu.html

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 --prefix spam
    spam foo
```

Create an overlay from a file

```
> 'export-env { let-env FOO = "foo" }' | save spam.nu
    overlay use spam.nu
    $env.FOO
```

p

version: 0.74.0

usage:

Switch to the previous shell.

Signature

```
> p
```

Examples

Make two directories and enter new shells for them, use `p` to jump to the previous shell

```
> mkdir foo bar; enter foo; enter ../bar; p
```

Run `p` several times and note the changes of current directory

```
> p
```

par-each

version: 0.74.0

usage:

Run a closure on each row of the input list in parallel, creating a new list with the results.

Signature

`> par-each (closure) --numbered`

Parameters

- `closure`: the closure to run
- `--numbered`: iterate with an index (deprecated; use a two-parameter closure instead)

Examples

Multiplies each number. Note that the list will become arbitrarily disordered.

```
> [1 2 3] | par-each { 2 * $in }
```

Iterate over each element, print the matching value and its index

```
> [1 2 3] | par-each -n { |it| if $it.item == 2 { $"found
2 at ($it.index)!"} }
```

parse

version: 0.74.0

usage:

Parse columns from string data using a simple pattern.

Signature

`> parse (pattern) --regex`

Parameters

- `pattern`: the pattern to match. Eg) "{foo}: {bar}"
- `--regex`: use full regex syntax for patterns

Examples

Parse a string into two named columns

```
> "hi there" | parse "{foo} {bar}"
```

Parse a string using regex pattern

```
> "hi there" | parse -r '(?P<foo>\w+) (?P<bar>\w+)'
```

Parse a string using fancy-regex named capture group pattern

```
> "foo bar." | parse -r '\s*(?<name>\w+)(?=\.)'
```

Parse a string using fancy-regex capture group pattern

```
> "foo! bar." | parse -r '(\w+)(?=\.)|(\w+)(?=!)'
```

Parse a string using fancy-regex look behind pattern

```
> " @another(foo bar)   " | parse -r '\s*(?<=[() ])(@\w+)
(\([^)]*\))?\s*'
```

Parse a string using fancy-regex look ahead atomic group pattern

```
> "abcd" | parse -r '^a(bc(?=d)|b)cd$'
```

path

version: 0.74.0

usage:

Explore and manipulate paths.

Signature

> path

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.
```

path basename

version: 0.74.0

usage:

Get the final component of a path

Signature

> path basename --columns --replace

Parameters

- `--columns {table}`: For a record or table input, convert strings in the given columns to their basename
- `--replace {string}`: Return original path with basename replaced by this string

Examples

Get basename of a path

```
> '/home/joe/test.txt' | path basename
```

Get basename of a path by column

```
> [[name];[/home/joe]] | path basename -c [ name ]
```

Replace basename of a path

```
> '/home/joe/test.txt' | path basename -r 'spam.png'
```

path dirname

version: 0.74.0

usage:

Get the parent directory of a path

Signature

```
> path dirname --columns --replace --num-levels
```

Parameters

- `--columns {table}`: For a record or table input, convert strings at the given columns to their dirname
- `--replace {string}`: Return original path with dirname replaced by this string
- `--num-levels {int}`: Number of directories to walk up

Examples

Get dirname of a path

```
> '/home/joe/code/test.txt' | path dirname
```

Get dirname of a path in a column

```
> ls ('.' | path expand) | path dirname -c [ name ]
```

Walk up two levels

```
> '/home/joe/code/test.txt' | path dirname -n 2
```

Replace the part that would be returned with a custom path

```
> '/home/joe/code/test.txt' | path dirname -n 2 -r /home/
viking
```

path exists

version: 0.74.0

usage:

Check whether a path exists

Signature

`> path exists --columns`

Parameters

- `--columns {table}`: For a record or table input, check strings at the given columns, and replace with result

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`.
```

Examples

Check if a file exists

```
> '/home/joe/todo.txt' | path exists
```

Check if a file exists in a column

```
> ls | path exists -c [ name ]
```

path expand

version: 0.74.0

usage:

Try to expand a path to its absolute form

Signature

`> path expand --strict --no-symlink --columns`

Parameters

- `--strict`: Throw an error if the path could not be expanded
- `--no-symlink`: Do not resolve symbolic links
- `--columns {table}`: For a record or table input, expand strings at the given columns

Examples

Expand an absolute path

```
> '/home/joe/foo/../bar' | path expand
```

Expand a path in a column

```
> ls | path expand -c [ name ]
```

Expand a relative path

```
> 'foo/../bar' | path expand
```

path join

version: 0.74.0

usage:

Join a structured path or a list of path parts.

Signature

`> path join ...rest --columns`

Parameters

- `...rest`: Path to append to the input
- `--columns {table}`: For a record or table input, join strings at the given columns

Notes

```
Optionally, append an additional path to the result. It
is designed to accept
the output of 'path parse' and 'path split' subcommands.
```

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
```

Append a filename to a path inside a column

```
> ls | path join spam.txt -c [ name ]
```

Join a list of parts into a path

```
> [ '/' 'home' 'viking' 'spam.txt' ] | path join
```

Join a structured path into a path

```
> [[ parent stem extension ]; [ '/home/viking' 'spam' 'txt'
]] | path join
```

path parse

version: 0.74.0

usage:

Convert a path into structured data.

Signature

```
> path parse --columns --extension
```

Parameters

- `--columns {table}`: For a record or table input, convert strings at the given columns
- `--extension {string}`: Manually supply the extension (without the dot)

Notes

```
Each path is split into a table with 'parent', 'stem' and
'extension' fields.
On Windows, an extra 'prefix' column is added.
```

Examples

Parse a path

```
> '/home/viking/spam.txt' | path parse
```

Replace a complex extension

```
> '/home/viking/spam.tar.gz' | path parse -e tar.gz | upsert
extension { 'txt' }
```

Ignore the extension

```
> '/etc/conf.d' | path parse -e ''
```

Parse all paths under the 'name' column

```
> ls | path parse -c [ name ]
```

path relative-to

version: 0.74.0

usage:

Express a path as relative to another path.

Signature

`> path relative-to (path) --columns`

Parameters

- `path`: Parent shared with the input path
- `--columns {table}`: For a record or table input, convert strings at the given columns

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.
```

Examples

Find a relative path from two absolute paths

```
> '/home/viking' | path relative-to '/home'
```

Find a relative path from two absolute paths in a column

```
> ls ~ | path relative-to ~ -c [ name ]
```

Find a relative path from two relative paths

```
> 'eggs/bacon/sausage/spam' | path relative-to 'eggs/bacon/
sausage'
```

path split

version: 0.74.0

usage:

Split a path into a list based on the system's path separator.

Signature

```
> path split --columns
```

Parameters

- `--columns {table}`: For a record or table input, split strings at the given columns

Examples

Split a path into parts

```
> '/home/viking/spam.txt' | path split
```

Split all paths under the 'name' column

```
> ls ('.' | path expand) | path split -c [ name ]
```

path type

version: 0.74.0

usage:

Get the type of the object a path refers to (e.g., file, dir, symlink)

Signature

```
> path type --columns
```

Parameters

- `--columns {table}`: For a record or table input, check strings at the given columns, and replace with result

Notes

```
This checks the file system to confirm the path's object
type.
If nothing is found, an empty string will be returned.
```

Examples

Show type of a filepath

```
> '.' | path type
```

Show type of a filepath in a column

```
> ls | path type -c [ name ]
```

port

version: 0.74.0

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)

Examples

get a free port between 3121 and 4000

```
> port 3121 4000
```

get a free port from system

```
> port
```

post

version: 0.74.0

usage:

Post a body to a URL.

Signature

`> post (path) (body) --user --password --content-type --content-length --headers --raw --insecure`

Parameters

- `path`: the URL to post to
- `body`: 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
- `--content-length {any}`: the length of the content being posted
- `--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

Notes

```
Performs HTTP POST operation.
```

Examples

Post content to url.com

```
> post url.com 'body'
```

Post content to url.com, with username and password

```
> post -u myuser -p mypass url.com 'body'
```

Post content to url.com, with custom header

```
> post -H [my-header-key my-header-value] url.com
```

Post content to url.com with a json body

```
> post -t application/json url.com { field: value }
```

prepend

version: 0.74.0

usage:

Prepend any number of rows to a table.

Signature

`> prepend (row)`

Parameters

- `row`: the row, list, or table to prepend

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.
```

Examples

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]
```

print

version: 0.74.0

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

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.
```

Examples

Print 'hello world'

```
> print "hello world"
```

Print the sum of 2 and 3

```
> print (2 + 3)
```

ps

version: 0.74.0

usage:

View information about system processes.

Signature

`> ps --long`

Parameters

- `--long`: list all available columns for each entry

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'
```

quantile

version: 0.74.0

usage:

Aggregates the columns to the selected quantile

Signature

`> quantile`

Examples

Quantile aggregation for a group-by

```
> [[a b]; [one 2] [one 4] [two 1]]
    | into df
    | group-by a
    | agg (col b | quantile 0.5)
```

quantile

version: 0.74.0

usage:

Aggregates the columns to the selected quantile

Signature

```
> quantile
```

Examples

quantile value from columns in a dataframe

```
> [[a b]; [6 2] [1 4] [4 1]] | into df | quantile 0.5
```

query

version: 0.74.0

usage:

Show all the query commands

Signature

```
> query
```

query db

version: 0.74.0

usage:

Query a database using SQL.

Signature

```
> query db (SQL)
```

Parameters

- `SQL`: SQL to execute against the database

Examples

Execute SQL against a SQLite database

```
> open foo.db | query db "SELECT * FROM Bar"
```

query df

version: 0.74.0

usage:

Query dataframe using SQL. Note: The dataframe is always named 'df' in your query's from clause.

Signature

```
> query df
```

Examples

Query dataframe using SQL

```
> [[a b]; [1 2] [3 4]] | into df | query df 'select a from
df'
```

query json

version: 0.74.0

usage:

execute json query on json file (open --raw <file> | query json 'query string')

Signature

```
> query json
```

query web

version: 0.74.0

usage:

execute selector query on html/web

Signature

```
> query web
```

query xml

version: 0.74.0

usage:

execute xpath query on xml

Signature

```
> query xml
```

random

version: 0.74.0

usage:

Generate a random value.

Signature

```
> random
```

Notes

> ```
> You must use one of the following subcommands. Using this
> command as-is will only produce this help message.
> ```

random bool

version: 0.74.0

usage:

Generate a random boolean value

Signature

`> random bool --bias`

Parameters

- `--bias {number}`: Adjusts the probability of a "true" outcome

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: 0.74.0

usage:

Generate random chars

Signature

`> random chars --length`

Parameters

- `--length {int}`: Number of chars

Examples

Generate random chars

```
> random chars
```

Generate random chars with specified length

```
> random chars -l 20
```

random decimal

version: 0.74.0

usage:

Generate a random decimal within a range [min..max]

Signature

`> random decimal (range)`

Parameters

- `range`: Range of values

Examples

Generate a default decimal value between 0 and 1

```
> random decimal
```

Generate a random decimal less than or equal to 500

```
> random decimal ..500
```

Generate a random decimal greater than or equal to 100000

```
> random decimal 100000..
```

Generate a random decimal between 1.0 and 1.1

```
> random decimal 1.0..1.1
```

random dice

version: 0.74.0

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

Examples

Roll 1 dice with 6 sides each

```
> random dice
```

Roll 10 dice with 12 sides each

```
> random dice -d 10 -s 12
```

random integer

version: 0.74.0

usage:

Generate a random integer [min..max]

Signature

`> random integer (range)`

Parameters

- `range`: Range of values

Examples

Generate an unconstrained random integer

```
> random integer
```

Generate a random integer less than or equal to 500

```
> random integer ..500
```

Generate a random integer greater than or equal to 100000

```
> random integer 100000..
```

Generate a random integer between 1 and 10

```
> random integer 1..10
```

random uuid

version: 0.74.0

usage:

Generate a random uuid4 string

Signature

```
> random uuid
```

Examples

Generate a random uuid4 string

```
> random uuid
```

range

version: 0.74.0

usage:

Return only the selected rows.

Signature

```
> range (rows)
```

Parameters

- `rows`: range of rows to return: Eg) 4..7 (=> from 4 to 7)

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: 0.74.0

usage:

Aggregate a list to a single value using an accumulator closure.

Signature

```
> reduce (closure) --fold --numbered
```

Parameters

- `closure`: reducing function
- `--fold {any}`: reduce with initial value
- `--numbered`: iterate with an index (deprecated; use a 3-parameter closure instead)

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 ] | reduce {|it, acc, ind| $acc + $it + $ind
}
```

Sum values with a starting value (fold)

```
> [ 1 2 3 4 ] | reduce -f 10 {|it, acc| $acc + $it }
```

Replace selected characters in a string with 'X'

```
> [ i o t ] | reduce -f "Arthur, King of the Britons" {|it,
acc| $acc | str replace -a $it "X" }
```

Add ascending numbers to each of the filenames, and join with semi-colons.

```
> ['foo.gz', 'bar.gz', 'baz.gz'] | reduce -f '' {|str all
ind| $"($all)(if $ind != 0 {'; '})($ind + 1)-($str)" }
```

register

version: 0.74.0

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)

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Register `nu_plugin_query` plugin from ~/.cargo/bin/ dir

```
> register ~/.cargo/bin/nu_plugin_query
```

Register `nu_plugin_query` plugin from `nu -c`(plugin will be available in that nu session only)

```
> let plugin = ((which nu).path.0 | path dirname | path
join 'nu_plugin_query'); nu -c $'register ($plugin); version'
```

reject

version: 0.74.0

usage:

Remove the given columns from the table. To remove rows, use 'drop'.

Signature

```
> reject ...rest
```

Parameters

- `...rest`: the names of columns to remove from the 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 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
```

rename

version: 0.74.0

usage:

Rename a dataframe column

Signature

```
> rename
```

Examples

Renames a series

```
> [5 6 7 8] | into df | rename '0' new_name
```

Renames a dataframe column

```
> [[a b]; [1 2] [3 4]] | into df | rename a a_new
```

Renames two dataframe columns

```
> [[a b]; [1 2] [3 4]] | into df | rename [a b] [a_new
b_new]
```

rename

version: 0.74.0

usage:

Creates a new table with columns renamed.

Signature

```
> rename ...rest --column
```

Parameters

- `...rest`: the new names for the columns
- `--column {list<string>}`: column name to be changed

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 -c [a ham]
```

Rename the fields of a record

```
> {a: 1 b: 2} | rename x y
```

replace-all

version: 0.74.0

usage:

Replace all (sub)strings by a regex pattern

Signature

`> replace-all`

Examples

Replaces string

```
> [abac abac abac] | into df | replace-all -p a -r A
```

replace

version: 0.74.0

usage:

Replace the leftmost (sub)string by a regex pattern

Signature

`> replace`

Examples

Replaces string

```
> [abc abc abc] | into df | replace -p ab -r AB
```

return

version: 0.74.0

usage:

Return early from a function

Signature

`> return (return_value)`

Parameters

- `return_value`: optional value to return

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Return early

```
> def foo [] { return }
```

reverse

version: 0.74.0

usage:

Reverses the input list or table.

Signature

```
> reverse
```

Examples

Reverse a list

```
> [0,1,2,3] | reverse
```

Reverse a table

```
> [{a: 1} {a: 2}] | reverse
```

reverse

version: 0.74.0

usage:

Reverses the LazyFrame

Signature

> reverse

Examples

Reverses the dataframe

```
> [[a b]; [6 2] [4 2] [2 2]] | into df | reverse
```

rm

version: 0.74.0

usage:

Remove files and directories.

Signature

> rm (filename) ...rest --trash --permanent --recursive --force --verbose --interactive --interactive-once

Parameters

- `filename`: the path of the file you want to remove
- `...rest`: additional file path(s) to remove
- `--trash`: move to the platform's trash instead of permanently deleting
- `--permanent`: delete permanently, ignoring the 'always_trash' config option
- `--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

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: 0.74.0

usage:

Rolling commands for tables

Signature

```
> roll
```

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

roll down

version: 0.74.0

usage:

Roll table rows down

Signature

`> roll down --by`

Parameters

- `--by {int}`: Number of rows to roll

Examples

Rolls rows down of a table

```
> [[a b]; [1 2] [3 4] [5 6]] | roll down
```

roll left

version: 0.74.0

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

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: 0.74.0

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

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: 0.74.0

usage:

Roll table rows up

Signature

`> roll up --by`

Parameters

- `--by {int}`: Number of rows to roll

Examples

Rolls rows up

```
> [[a b]; [1 2] [3 4] [5 6]] | roll up
```

rolling

version: 0.74.0

usage:

Rolling calculation for a series

Signature

`> rolling`

Examples

Rolling sum for a series

```
> [1 2 3 4 5] | into df | rolling sum 2 | drop-nulls
```

Rolling max for a series

```
> [1 2 3 4 5] | into df | rolling max 2 | drop-nulls
```

rotate

version: 0.74.0

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

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: 0.74.0

usage:

Runs external command

Signature

`> run-external (command) ...rest --redirect-stdout --redirect-stderr --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
- `--trim-end-newline`: trimming end newlines

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
```

sample

version: 0.74.0

usage:

Create sample dataframe

Signature

```
> sample
```

Examples

Sample rows from dataframe

```
> [[a b]; [1 2] [3 4]] | into df | sample -n 1
```

Shows sample row using fraction and replace

```
> [[a b]; [1 2] [3 4] [5 6]] | into df | sample -f 0.5
-e
```

save

version: 0.74.0

usage:

Save a file.

Signature

`> save (filename) --stderr --raw --append --force`

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

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: 0.74.0

usage:

Show the schema of a SQLite database.

Signature

`> schema`

Examples

Show the schema of a SQLite database

```
> open foo.db | schema
```

select

version: 0.74.0

usage:

Down-select table to only these columns.

Signature

`> select ...rest --ignore-errors`

Parameters

- `...rest`: the columns to select from the table
- `--ignore-errors`: when an error occurs, instead of erroring out, suppress the error message

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 name and size columns

```
> ls | select name size
```

select

version: 0.74.0

usage:

Selects columns from lazyframe

Signature

`> select`

Examples

Select a column from the dataframe

```
> [[a b]; [6 2] [4 2] [2 2]] | into df | select a
```

seq

version: 0.74.0

usage:

Output sequences of numbers.

Signature

> seq ...rest

Parameters

- `...rest`: sequence values

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 '|'
```

seq char

version: 0.74.0

usage:

Print a sequence of ASCII characters

Signature

> seq char (start) (end)

Parameters

- `start`: start of character sequence (inclusive)
- `end`: end of character sequence (inclusive)

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: 0.74.0

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

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 -r
```

print the previous 10 days starting today in MM/DD/YYYY format with newline separator

```
> seq date --days 10 -o '%m/%d/%Y' -r
```

print the first 10 days in January, 2020

```
> seq date -b '2020-01-01' -e '2020-01-10'
```

print every fifth day between January 1st 2020 and January 31st 2020

```
> seq date -b '2020-01-01' -e '2020-01-31' -n 5
```

set-with-idx

version: 0.74.0

usage:

Sets value in the given index

Signature

`> set-with-idx`

Examples

Set value in selected rows from series

```
> let series = ([4 1 5 2 4 3] | into df);
    let indices = ([0 2] | into df);
    $series | set-with-idx 6 -i $indices
```

set

version: 0.74.0

usage:

Sets value where given mask is true

Signature

```
> set
```

Examples

Shifts the values by a given period

```
> let s = ([1 2 2 3 3] | into df | shift 2);
    let mask = ($s | is-null);
    $s | set 0 --mask $mask
```

shape

version: 0.74.0

usage:

Shows column and row size for a dataframe

Signature

```
> shape
```

Examples

Shows row and column shape

```
> [[a b]; [1 2] [3 4]] | into df | shape
```

shells

version: 0.74.0

usage:

Lists all open shells.

Signature

```
> shells
```

Examples

Enter a new shell at parent path '..' and show all opened shells

```
> enter ..; shells
```

Show currently active shell

```
> shells | where active == true
```

shift

version: 0.74.0

usage:

Shifts the values by a given period

Signature

```
> shift
```

Examples

Shifts the values by a given period

```
> [1 2 2 3 3] | into df | shift 2 | drop-nulls
```

shuffle

version: 0.74.0

usage:

Shuffle rows randomly.

Signature

`> shuffle`

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
```

size

version: 0.74.0

usage:

Gather word count statistics on the text.

Signature

`> size`

Examples

Count the number of words in a string

```
> "There are seven words in this sentence" | size
```

Counts unicode characters

```
> '    ' | size
```

Counts Unicode characters correctly in a string

```
> "Amélie Amelie" | size
```

skip

version: 0.74.0

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

Notes

```
To skip specific numbered rows, try 'drop nth'. To skip
specific named columns, try 'reject'.
```

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
```

skip until

version: 0.74.0

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

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: 0.74.0

usage:

Skip elements of the input while a predicate is true.

Signature

```
> skip while (predicate)
```

Parameters

- `predicate`: the predicate that skipped element must match

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: 0.74.0

usage:

Delay for a specified amount of time.

Signature

`> sleep (duration) ...rest`

Parameters

- `duration`: time to sleep
- `...rest`: additional time

Examples

Sleep for 1sec

```
> sleep 1sec
```

slice

version: 0.74.0

usage:

Creates new dataframe from a slice of rows

Signature

`> slice`

Examples

Create new dataframe from a slice of the rows

```
> [[a b]; [1 2] [3 4]] | into df | slice 0 1
```

sort-by

version: 0.74.0

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, ...)

Examples

Sort files by modified date

```
> ls | sort-by modified
```

Sort files by name (case-insensitive)

```
> ls | sort-by name -i
```

Sort a table by a column (reversed order)

```
> [[fruit count]; [apple 9] [pear 3] [orange 7]] | sort-
by fruit -r
```

sort-by

version: 0.74.0

usage:

sorts a lazy dataframe based on expression(s)

Signature

`> sort-by`

Examples

Sort dataframe by one column

```
> [[a b]; [6 2] [1 4] [4 1]] | into df | sort-by a
```

Sort column using two columns

```
> [[a b]; [6 2] [1 1] [1 4] [2 4]] | into df | sort-by
[a b] -r [false true]
```

sort

version: 0.74.0

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, ...)

Examples

sort the list by increasing value

```
> [2 0 1] | sort
```

sort the list by decreasing value

```
> [2 0 1] | sort -r
```

sort a list of strings

```
> [betty amy sarah] | sort
```

sort a list of strings in reverse

```
> [betty amy sarah] | sort -r
```

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: 0.74.0

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

Examples

Sources the environment from foo.nu in the current context

```
> source-env foo.nu
```

source

version: 0.74.0

usage:

Runs a script file in the current context.

Signature

`> source (filename)`

Parameters

- `filename`: the filepath to the script file to source

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

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
```

split-by

version: 0.74.0

usage:

Create a new table splitted.

Signature

`> split-by (splitter)`

Parameters

- `splitter`: the splitter value to use

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: 0.74.0

usage:

Split contents across desired subcommand (like row, column) via the separator.

Signature

```
> split
```

Notes

> You must use one of the following subcommands. Using this command as-is will only produce this help message.

split chars

version: 0.74.0

usage:

Split a string into a list of characters

Signature

```
> split chars
```

Examples

Split the string into a list of characters

```
> 'hello' | split chars
```

split column

version: 0.74.0

usage:

Split a string into multiple columns using a separator

Signature

> split column (separator) ...rest --collapse-empty

Parameters

- `separator`: the character or string that denotes what separates columns
- `...rest`: column names to give the new columns
- `--collapse-empty`: remove empty columns

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 -c ''
```

Split a list of strings into a table

```
> ['a-b' 'c-d'] | split column -
```

split list

version: 0.74.0

usage:

Split a list into multiple lists using a separator

Signature

> split list (separator)

Parameters

- `separator`: the value that denotes what separates the list

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 row

version: 0.74.0

usage:

Split a string into multiple rows using a separator

Signature

`> split row (separator) --number`

Parameters

- `separator`: the character that denotes what separates rows
- `--number {int}`: Split into maximum number of items

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 words

version: 0.74.0

usage:

Split a string's words into separate rows

Signature

`> split words --min-word-length`

Parameters

- `--min-word-length {int}`: The minimum word length

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 -l 3
```

A real-world example of splitting words

```
> fetch https://www.gutenberg.org/files/11/11-0.txt | str
downcase | split words -l 2 | uniq -c | sort-by count -
-reverse | first 10
```

start

version: 0.74.0

usage:

Open a folder or file in the default application or viewer.

Signature

```
> start (filepath)
```

Parameters

- `filepath`: the filepath to open

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
```

std

version: 0.74.0

usage:

Creates a std expression for an aggregation

Signature

```
> std
```

Examples

Std aggregation for a group-by

```
> [[a b]; [one 2] [one 2] [two 1] [two 1]]
    | into df
    | group-by a
    | agg (col b | std)
```

std

version: 0.74.0

usage:

Aggregates columns to their std value

Signature

```
> std
```

Examples

Std value from columns in a dataframe

```
> [[a b]; [6 2] [4 2] [2 2]] | into df | std
```

str-lengths

version: 0.74.0

usage:

Get lengths of all strings

Signature

```
> str-lengths
```

Examples

Returns string lengths

```
> [a ab abc] | into df | str-lengths
```

str-slice

version: 0.74.0

usage:

Slices the string from the start position until the selected length

Signature

```
> str-slice
```

Examples

Creates slices from the strings

```
> [abcded abc321 abc123] | into df | str-slice 1 -l 2
```

str

version: 0.74.0

usage:

Various commands for working with string data

Signature

`> str`

Notes

> You must use one of the following subcommands. Using this command as-is will only produce this help message.

str camel-case

version: 0.74.0

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

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: 0.74.0

usage:

Capitalize first letter of text

Signature

`> str capitalize ...rest`

Parameters

- `...rest`: For a data structure input, convert strings at the given cell paths

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 collect

version: 0.74.0

usage:

'str collect' is deprecated. Please use 'str join' instead.

Signature

`> str collect (separator)`

Parameters

- `separator`: optional separator to use when creating string

Examples

Create a string from input

```
> ['nu', 'shell'] | str collect
```

Create a string from input with a separator

```
> ['nu', 'shell'] | str collect '-'
```

str contains

version: 0.74.0

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

Examples

Check if input contains string

```
> 'my_library.rb' | str contains '.rb'
```

Check if input contains string case insensitive

```
> 'my_library.rb' | str contains -i '.RB'
```

Check if input contains string in a table

```
>  [[ColA ColB]; [test 100]] | str contains 'e' ColA
```

Check if input contains string in a table

```
>  [[ColA ColB]; [test 100]] | str contains -i '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 -n o
```

str distance

version: 0.74.0

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

Examples

get the edit distance between two strings

```
> 'nushell' | str distance 'nutshell'
```

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: 0.74.0

usage:

Make text lowercase

Signature

`> str downcase ...rest`

Parameters

- `...rest`: For a data structure input, convert strings at the given cell paths

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: 0.74.0

usage:

Check if an input ends with a string

Signature

```
> str ends-with (string) ...rest
```

Parameters

- `string`: the string to match
- `...rest`: For a data structure input, check strings at the given cell paths, and replace with result

Examples

Checks if string ends with '.rb'

```
> 'my_library.rb' | str ends-with '.rb'
```

Checks if string ends with '.txt'

```
> 'my_library.rb' | str ends-with '.txt'
```

str find-replace

version: 0.74.0

usage:

Deprecated command

Signature

```
> str find-replace
```

str index-of

version: 0.74.0

usage:

Returns start index of first occurrence of string in input, or -1 if no match

Signature

```
> str index-of (string) ...rest --range --end
```

Parameters

- `string`: the string to find index of
- `...rest`: For a data structure input, search strings at the given cell paths, and replace with result
- `--range {any}`: optional start and/or end index
- `--end`: search from the end of the input

Examples

Returns index of string in input

```
> 'my_library.rb' | str index-of '.rb'
```

Returns index of string in input with start index

```
> '.rb.rb' | str index-of '.rb' -r '1,'
```

Returns index of string in input with end index

```
> '123456' | str index-of '6' -r ',4'
```

Returns index of string in input with start and end index

```
> '123456' | str index-of '3' -r '1,4'
```

Alternatively you can use this form

```
> '123456' | str index-of '3' -r [1 4]
```

Returns index of string in input

```
> '/this/is/some/path/file.txt' | str index-of '/' -e
```

str join

version: 0.74.0

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

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: 0.74.0

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

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: 0.74.0

usage:

Output the length of any strings in the pipeline

Signature

`> str length ...rest`

Parameters

- `...rest`: For a data structure input, replace strings at the given cell paths with their length

Examples

Return the lengths of multiple strings

```
> 'hello' | str length
```

Return the lengths of multiple strings

```
> ['hi' 'there'] | str length
```

str lpad

version: 0.74.0

usage:

Left-pad a string to a specific length

Signature

`> str lpad ...rest --length --character`

Parameters

- `...rest`: For a data structure input, pad strings at the given cell paths
- `--length {int}`: length to pad to
- `--character {string}`: character to pad with

Examples

Left-pad a string with asterisks until it's 10 characters wide

```
> 'nushell' | str lpad -l 10 -c '*'
```

Left-pad a string with zeroes until it's 10 character wide

```
> '123' | str lpad -l 10 -c '0'
```

Use lpad to truncate a string to its last three characters

```
> '123456789' | str lpad -l 3 -c '0'
```

Use lpad to pad Unicode

```
> ' ' | str lpad -l 10 -c ' '
```

str pascal-case

version: 0.74.0

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

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: 0.74.0

usage:

Find and replace text

Signature

`> str replace (find) (replace) ...rest --all --no-expand --string`

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
- `--string`: match the pattern as a substring of the input, instead of a regular expression

Examples

Find and replace contents with capture group

```
> 'my_library.rb' | str replace '(.+).rb' '$1.nu'
```

Find and replace all occurrences of find string

```
> 'abc abc abc' | str replace -a 'b' 'z'
```

Find and replace all occurrences of find string in table

```
> [[ColA ColB ColC]; [abc abc ads]] | str replace -a 'b'
'z' ColA ColC
```

Find and replace contents without using the replace parameter as a regular expression

```
> 'dogs_$1_cats' | str replace '\$1' '$2' -n
```

Find and replace the first occurrence using string replacement *not* regular expressions

```
> 'c:\some\cool\path' | str replace 'c:\some\cool' '~'
-s
```

Find and replace all occurrences using string replacement *not* regular expressions

```
> 'abc abc abc' | str replace -a 'b' 'z' -s
```

Find and replace with fancy-regex

```
> 'a successful b' | str replace '\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

```
> 'GHIKK-9+*' | str replace '[*[:xdigit:]+]' 'z'
```

str reverse

version: 0.74.0

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

Examples

Reverse a single string

```
> 'Nushell' | str reverse
```

Reverse multiple strings in a list

```
> ['Nushell' 'is' 'cool'] | str reverse
```

str rpad

version: 0.74.0

usage:

Right-pad a string to a specific length

Signature

> str rpad ...rest --length --character

Parameters

- `...rest`: For a data structure input, pad strings at the given cell paths
- `--length {int}`: length to pad to
- `--character {string}`: character to pad with

Examples

Right-pad a string with asterisks until it's 10 characters wide

```
> 'nushell' | str rpad -l 10 -c '*'
```

Right-pad a string with zeroes until it's 10 characters wide

```
> '123' | str rpad -l 10 -c '0'
```

Use rpad to truncate a string to its first three characters

```
> '123456789' | str rpad -l 3 -c '0'
```

Use rpad to pad Unicode

```
> ' ' | str rpad -l 10 -c ' '
```

str screaming-snake-case

version: 0.74.0

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

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: 0.74.0

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

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: 0.74.0

usage:

Check if an input starts with a string

Signature

`> str starts-with (string) ...rest`

Parameters

- `string`: the string to match
- `...rest`: For a data structure input, check strings at the given cell paths, and replace with result

Examples

Checks if input string starts with ‘my’

```
> 'my_library.rb' | str starts-with 'my'
```

Checks if input string starts with ‘my’

```
> 'Cargo.toml' | str starts-with 'Car'
```

Checks if input string starts with ‘my’

```
> 'Cargo.toml' | str starts-with '.toml'
```

str substring

version: 0.74.0

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`

Parameters

- `range`: the indexes to substring [start end]
- `...rest`: For a data structure input, turn strings at the given cell paths into substrings

Examples

Get a substring “nushell” from the text “good nushell” using a range

```
>  'good nushell' | str substring 5..12
```

Alternately, you can pass in a list

```
> 'good nushell' | str substring [5 12]
```

Or a simple comma-separated string

```
> 'good nushell' | str substring '5,12'
```

Drop the last `n` characters from the string

```
> 'good nushell' | str substring ',-5'
```

Get the remaining characters from a starting index

```
> 'good nushell' | str substring '5,'
```

Get the characters from the beginning until ending index

```
> 'good nushell' | str substring ',7'
```

str title-case

version: 0.74.0

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

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 to-datetime

version: 0.74.0

usage:

Deprecated command

Signature

```
> str to-datetime
```

str to-decimal

version: 0.74.0

usage:

Deprecated command

Signature

```
> str to-decimal
```

str to-int

version: 0.74.0

usage:

Deprecated command

Signature

`> str to-int`

str trim

version: 0.74.0

usage:

Trim whitespace or specific character

Signature

`> str trim ...rest --char --left --right --all --both --format`

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 (default: whitespace)
- `--right`: trims characters only from the end of the string (default: whitespace)
- `--all`: trims all characters from both sides of the string *and* in the middle (default: whitespace)
- `--both`: trims all characters from left and right side of the string (default: whitespace)
- `--format`: trims spaces replacing multiple characters with singles in the middle (default: whitespace)

Examples

Trim whitespace

```
> 'Nu shell ' | str trim
```

Trim a specific character

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

Trim all characters

```
> ' Nu   shell ' | str trim -a
```

Trim whitespace from the beginning of string

```
> ' Nu shell ' | str trim -l
```

Trim a specific character

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

Trim whitespace from the end of string

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

Trim a specific character

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

str upcase

version: 0.74.0

usage:

Make text uppercase

Signature

```
> str upcase ...rest
```

Parameters

- `...rest`: For a data structure input, convert strings at the given cell paths

Examples

Upcase contents

```
> 'nu' | str upcase
```

strftime

version: 0.74.0

usage:

Formats date based on string rule

Signature

```
> strftime
```

Examples

Formats date

```
> let dt = ('2020-08-04T16:39:18+00:00' | into datetime
-z 'UTC');
    let df = ([$dt $dt] | into df);
    $df | strftime "%Y/%m/%d"
```

sum

version: 0.74.0

usage:

Creates a sum expression for an aggregation

Signature

```
> sum
```

Examples

Sum aggregation for a group-by

```
> [[a b]; [one 2] [one 4] [two 1]]
    | into df
    | group-by a
    | agg (col b | sum)
```

sum

version: 0.74.0

usage:

Aggregates columns to their sum value

Signature

```
> sum
```

Examples

Sums all columns in a dataframe

```
> [[a b]; [6 2] [1 4] [4 1]] | into df | sum
```

summary

version: 0.74.0

usage:

For a dataframe, produces descriptive statistics (summary statistics) for its numeric columns.

Signature

```
> summary
```

Examples

list dataframe descriptives

```
> [[a b]; [1 1] [1 1]] | into df | summary
```

sys

version: 0.74.0

usage:

View information about the system.

Signature

```
> sys
```

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: 0.74.0

usage:

Render the table.

Signature

```
> table --start-number --list --width --expand --expand-deep --flatten --flatten-separator --collapse
```

Parameters

- `--start-number {int}`: row number to start viewing from
- `--list`: list available table modes/themes
- `--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
- `--flatten`: Flatten simple arrays
- `--flatten-separator {string}`: sets a separator when 'flatten' used
- `--collapse`: expand the table structure in colapse mode. Be aware collapse mode currently doesn't support width control

Notes

```
If the table contains a column called 'index', this column
is used as the table index instead of the usual continuous
```

```
index
```

Examples

List the files in current directory, with indexes starting from 1.

```
> ls | table -n 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
```

take

version: 0.74.0

usage:

Creates new dataframe using the given indices

Signature

```
> take
```

Examples

Takes selected rows from dataframe

```
> let df = ([[a b]; [4 1] [5 2] [4 3]] | into df);
    let indices = ([0 2] | into df);
```

```
    $df | take $indices
```

Takes selected rows from series

```
> let series = ([4 1 5 2 4 3] | into df);
    let indices = ([0 2] | into df);
    $series | take $indices
```

take

version: 0.74.0

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

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
```

take until

version: 0.74.0

usage:

Take elements of the input until a predicate is true.

Signature

```
> take until (predicate)
```

Parameters

- `predicate`: the predicate that element(s) must not match

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: 0.74.0

usage:

Take elements of the input while a predicate is true.

Signature

`> take while (predicate)`

Parameters

- `predicate`: the predicate that element(s) must match

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: 0.74.0

usage:

Returns a record containing the number of columns (width) and rows (height) of the terminal

Signature

`> term size`

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
```

to

version: 0.74.0

usage:

Translate structured data to a format

Signature

`> to`

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

to arrow

version: 0.74.0

usage:

Saves dataframe to arrow file

Signature

```
> to arrow
```

Examples

Saves dataframe to arrow file

```
> [[a b]; [1 2] [3 4]] | into df | to arrow test.arrow
```

to csv

version: 0.74.0

usage:

Saves dataframe to csv file

Signature

```
> to csv
```

Examples

Saves dataframe to csv file

```
> [[a b]; [1 2] [3 4]] | into df | to csv test.csv
```

Saves dataframe to csv file using other delimiter

```
> [[a b]; [1 2] [3 4]] | into df | to csv test.csv -d '|'
```

to csv

version: 0.74.0

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

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 -s ';'
```

to html

version: 0.74.0

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

Notes

```
Screenshots of the themes can be browsed here: https://
github.com/mbadolato/iTerm2-Color-Schemes
```

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
```

to json

version: 0.74.0

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

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 -i 4
```

Outputs an unformatted JSON string representing the contents of this table

```
> [1 2 3] | to json -r
```

to md

version: 0.74.0

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

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: 0.74.0

usage:

Converts table data into Nuon (Nushell Object Notation) text.

Signature

```
> to nuon
```

Examples

Outputs a nuon string representing the contents of this list

```
> [1 2 3] | to nuon
```

to parquet

version: 0.74.0

usage:

Saves dataframe to parquet file

Signature

`> to parquet`

Examples

Saves dataframe to parquet file

```
> [[a b]; [1 2] [3 4]] | into df | to parquet test.parquet
```

to text

version: 0.74.0

usage:

Converts data into simple text.

Signature

`> to text`

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: 0.74.0

usage:

Convert record into .toml text

Signature

`> to toml`

Examples

Outputs an TOML string representing the contents of this record

```
> {foo: 1 bar: 'qwe'} | to toml
```

to tsv

version: 0.74.0

usage:

Convert table into .tsv text

Signature

`> to tsv --noheaders`

Parameters

- `--noheaders`: do not output the column names as the first row

Examples

Outputs an TSV string representing the contents of this table

```
> [[foo bar]; [1 2]] | to tsv
```

to url

version: 0.74.0

usage:

Convert record or table into URL-encoded text

Signature

```
> to url
```

Examples

Outputs a URL string representing the contents of this record

```
> { mode:normal userid:31415 } | to url
```

Outputs a URL string representing the contents of this 1-row table

```
> [[foo bar]; ["1" "2"]] | to url
```

to xml

version: 0.74.0

usage:

Convert table into .xml text

Signature

```
> to xml --pretty
```

Parameters

- `--pretty {int}`: Formats the XML text with the provided indentation setting

Examples

Outputs an XML string representing the contents of this table

```
> { "note": { "children": [{ "remember": {"attributes"
: {}, "children": [Event]}}], "attributes": {} } } | to
xml
```

Optionally, formats the text with a custom indentation setting

```
> { "note": { "children": [{ "remember": {"attributes"
: {}, "children": [Event]}}], "attributes": {} } } | to
xml -p 3
```

to yaml

version: 0.74.0

usage:

Convert table into .yaml/.yml text

Signature

`> to yaml`

Examples

Outputs an YAML string representing the contents of this table

```
> [[foo bar]; ["1" "2"]] | to yaml
```

touch

version: 0.74.0

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

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: 0.74.0

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

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 -i val
```

Transfer back to record with -d flag

```
> {c1: 1, c2: 2} | transpose | transpose -i -r -d
```

try

version: 0.74.0

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

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Try to run a missing command

```
> try { asdfasdf }
```

Try to run a missing command

```
> try { asdfasdf } catch { echo 'missing' }
```

tutor

version: 0.74.0

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

Examples

Begin the tutorial

```
> tutor begin
```

Search a tutorial by phrase

```
> tutor -f "$in"
```

uniq-by

version: 0.74.0

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

Examples

Get rows from table filtered by column uniqueness

```
> [[fruit count]; [apple 9] [apple 2] [pear 3] [orange
7]] | uniq-by fruit
```

uniq

version: 0.74.0

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

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 -u
```

Ignore differences in case when comparing input values

```
> ['hello' 'goodbye' 'Hello'] | uniq -i
```

Return a table containing the distinct input values together with their counts

```
> [1 2 2] | uniq -c
```

unique

version: 0.74.0

usage:

Returns unique values from a dataframe

Signature

`> unique`

Examples

Returns unique values from a series

```
> [2 2 2 2 2] | into df | unique
```

Creates a is unique expression from a column

```
> col a | unique
```

update

version: 0.74.0

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

Examples

Update a column value

```
> {'name': 'nu', 'stars': 5} | update name 'Nushell'
```

Use in closure form for more involved updating logic

```
> [[count fruit]; [1 'apple']] | update count {|row index|
($row.fruit | str length) + $index }
```

Alter each value in the 'authors' column to use a single string instead of a list

```
> [[project, authors]; ['nu', ['Andrés', 'JT', 'Yehuda']]]
| update authors {|row| $row.authors | str join ','}
```

update cells

version: 0.74.0

usage:

Update the table cells.

Signature

`> update cells (closure) --columns`

Parameters

- `closure`: the closure to run an update for each cell
- `--columns {table}`: list of columns to update

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
        }
}
```

uppercase

version: 0.74.0

usage:

Uppercase the strings in the column

Signature

`> uppercase`

Examples

Modifies strings to uppercase

```
> [Abc aBc abC] | into df | uppercase
```

upsert

version: 0.74.0

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

Examples

Update a record's value

```
> {'name': 'nu', 'stars': 5} | upsert name 'Nushell'
```

Insert a new entry into a single record

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

Use in closure form for more involved updating logic

```
> [[count fruit]; [1 'apple']] | upsert count {|row index|
($row.fruit | str length) + $index }
```

Upsert an int into a list, updating an existing value based on the index

```
> [1 2 3] | upsert 0 2
```

Upsert an int into a list, inserting a new value based on the index

```
> [1 2 3] | upsert 3 4
```

url

version: 0.74.0

usage:

Various commands for working with URLs

Signature

```
> url
```

Notes

```
You must use one of the following subcommands. Using this
command as-is will only produce this help message.
```

url encode

version: 0.74.0

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 `/`, `.`, `:`

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 anphanumeric chars with all flag

```
> 'https://example.com/foo bar' | url encode --all
```

url parse

version: 0.74.0

usage:

Parses a url

Signature

`> url parse ...rest`

Parameters

- `...rest`: optionally operate by cell path

Examples

Parses a url

```
> 'http://user123:pass567@www.example.com:8081/foo/bar?param1=section&p
| url parse
```

use

version: 0.74.0

usage:

Use definitions from a module

Signature

`> use (module) (members)`

Parameters

- `module`: Module or module file
- `members`: Which members of the module to import

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

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 [] { let-env FOO_BAR
= "BAZ" } }; use foo bar; bar; $env.FOO_BAR
```

value-counts

version: 0.74.0

usage:

Returns a dataframe with the counts for unique values in series

Signature

```
> value-counts
```

Examples

Calculates value counts

```
> [5 5 5 5 6 6] | into df | value-counts
```

values

version: 0.74.0

usage:

Given a record or table, produce a list of its columns' values.

Signature

```
> values
```

Notes

```
This is a counterpart to `columns`, which produces a list
of columns' names.
```

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
```

var

version: 0.74.0

usage:

Create a var expression for an aggregation

Signature

`> var`

Examples

Var aggregation for a group-by

```
> [[a b]; [one 2] [one 2] [two 1] [two 1]]
    | into df
    | group-by a
    | agg (col b | var)
```

var

version: 0.74.0

usage:

Aggregates columns to their var value

Signature

```
> var
```

Examples

Var value from columns in a dataframe

```
> [[a b]; [6 2] [4 2] [2 2]] | into df | var
```

version

version: 0.74.0

usage:

Display Nu version.

Signature

```
> version
```

Examples

Display Nu version

```
> version
```

view-source

version: 0.74.0

usage:

View a block, module, or a definition

Signature

```
> view-source (item)
```

Parameters

- `item`: name or block to view

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 [] { let-env BAR = 'BAZ' }; view-source foo
```

View the source of a module

```
> module mod-foo { export-env { let-env FOO_ENV = 'BAZ'
} }; view-source mod-foo
```

View the source of an alias

```
> alias hello = echo hi; view-source hello
```

watch

version: 0.74.0

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)

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 }
```

when

version: 0.74.0

usage:

Creates and modifies a when expression

Signature

```
> when
```

Examples

Create a when conditions

```
> when ((col a) > 2) 4
```

Create a when conditions

```
> when ((col a) > 2) 4 | when ((col a) < 0) 6
```

Create a new column for the dataframe

```
> [[a b]; [6 2] [1 4] [4 1]]
    | into lazy
    | with-column (
        when ((col a) > 2) 4 | otherwise 5 | as c
      )
    | with-column (
        when ((col a) > 5) 10 | when ((col a) < 2) 6 | otherwise
0 | as d
      )
    | collect
```

where

version: 0.74.0

usage:

Filter values based on a row condition.

Signature

```
> where (row_condition) --closure
```

Parameters

- `row_condition`: Filter condition
- `--closure {closure(any, int)}`: use with a closure instead (deprecated: use 'filter' command instead)

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.
```

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
```

which

version: 0.74.0

usage:

Finds a program file, alias or custom command.

Signature

`> which (application) ...rest --all`

Parameters

- `application`: application
- `...rest`: additional applications
- `--all`: list all executables

Examples

Find if the ‘myapp’ application is available

```
> which myapp
```

while

version: 0.74.0

usage:

Conditionally run a block in a loop.

Signature

```
> while (cond) (block)
```

Parameters

- `cond`: condition to check
- `block`: block to loop if check succeeds

Notes

```
This command is a parser keyword. For details, check:
  https://www.nushell.sh/book/thinking_in_nu.html
```

Examples

Loop while a condition is true

```
> mut x = 0; while $x < 10 { $x = $x + 1 }
```

window

version: 0.74.0

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

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 --remainder
```

with-column

version: 0.74.0

usage:

Adds a series to the dataframe

Signature

`> with-column`

Examples

Adds a series to the dataframe

```
> [[a b]; [1 2] [3 4]]
    | into df
    | with-column ([5 6] | into df) --name c
```

Adds a series to the dataframe

```
> [[a b]; [1 2] [3 4]]
    | into lazy
    | with-column [
      ((col a) * 2 | as "c")
      ((col a) * 3 | as "d")
     ]
    | collect
```

with-env

version: 0.74.0

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

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 row(e.g. `open x.json` or `from json`)

```
> '{"X":"Y","W":"Z"}'|from json|with-env $in { [$env.X
$env.W] }
```

wrap

version: 0.74.0

usage:

Wrap the value into a column.

Signature

`> wrap (name)`

Parameters

- `name`: the name of the column

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: 0.74.0

usage:

Combine a stream with the input

Signature

```
> zip (other)
```

Parameters

- `other`: the other input

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 }
```