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.

- The main Nushell repository can be found here. 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 page, including this book, can be found here.
- Nushell has its own line editor which has its own repository
- `nu_scripts` is a place to share scripts and modules with other users until we have some sort of package manager.
- Nana is an experimental effort to explore graphical user interface for Nushell.
- Awesome Nu contains a list of tools that work with the Nushell ecosystem: plugins, scripts, editor extension, 3rd party integrations, etc.
- Nu Showcase is a place to share works about Nushell, be it blogs, artwork or something else.
- Request for Comment (RFC) 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, 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. 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. You can also follow us on Twitter for news and updates. Finally, you can use the GitHub discussions or file GitHub issues.

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, use your favourite package manager, or build from source.

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

@code

Pre-built binaries

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

Package managers

Nu is available via several package managers:

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

For Windows:

- Winget (`winget install nushell`)
- Chocolatey (`choco install nushell`)
- Scoop (`scoop install nu`)

Cross Platform installation:

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

Build from source

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

Installing a compiler suite

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

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

Installing Rust

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

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

@code

Once you are ready, press 1 and then enter.

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

Dependencies

Debian/Ubuntu

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

@code

RHEL based distros

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

@code

macOS

Using Homebrew, you will need to install "openssl" and "cmake" using:

@code

Build using crates.io

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

@code

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

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

@code

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

@code

From there, we can build and run Nu with:

@code

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

@code

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

Default shell

Setting Nu as default shell on your terminal

Terminal	Platform	Instructions
GNOME Terminal	Linux & BSDs	Open `Edit > Preferences`. In the right-hand panel, select the `Command` tab, tick `Run a custom command instead of my shell`, and set `Custom command` to the path to Nu.
GNOME Console	Linux & BSDs	Type the command `gsettings set org.gnome.Console shell "['/usr/bin/nu']"` (replace `/usr/bin/nu` with the path to Nu). Equivalently, use dconf Editor to edit the `/org/gnome/Console/shell` key.
Kitty	Linux & BSDs	Press `Ctrl+Shift+F2` to open `kitty.conf`. Go to `shell` variable, uncomment the line and replace the `.` with the path to Nu.
Konsole	Linux & BSDs	Open `Settings > Edit Current Profile`. Set `Command` to the path to Nu.
XFCE Terminal	Linux & BSDs	Open `Edit > Preferences`. Check `Run a custom command instead of my shell`, and set `Custom command` to the path to Nu.
Terminal.app	macOS	Open `Terminal > Preferences`. Ensure you are on the `Profiles` tab, which should be the default tab. In the right-hand panel, select the `Shell` tab. Tick `Run command`, put the path to Nu in the textbox, and untick `Run inside shell`.
iTerm2	macOS	Open `iTerm > Preferences`. Select the `Profiles` tab. In the right-hand panel under `Command`, change the dropdown from `Login Shell` to `Custom Shell`, and put the path to Nu in the textbox.
Windows Terminal	Windows	Press `Ctrl+,` to open `Settings`. Go to `Add a new profile > New empty profile`. Fill in the ‘Name’ and enter path to Nu in the ‘Command line’ textbox. Go to `Startup` option and select Nu as the ‘Default profile’. Hit `Save`.

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

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

To set the login shell you can use the `chsh` 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:

@code

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

Quick Tour

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

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

@code

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

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

@code

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 use `where` to filter the contents of the `ls` table so that it only shows files over 1 kilobyte:

@code

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:

@code

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:

@code

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:

@code

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

@code

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

@code

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:

@code

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:

@code

Right now, there's just one user on the system named "sophiajt". 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 "sophiajt". 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):

@code

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>`.

@code

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

@code

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"

@code

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.

@code

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

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

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

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

If you pass a variable to a command that support globbing like this: `let f = "a[bc]d.txt"; rm $f`. It won't expand the glob pattern, only a file named `a[bc]d.txt` will be removed. Normally it's what you want, but if you want to expand the glob pattern, there are 3 ways to achieve it:

1. using spread operator along with `glob` command: `let f = "a[bc]d.txt"; rm ...(glob $f)`. This way is recommended because it's expressed most explicitly, but it doesn't work with `ls` and `du` command, for the case, you can
2. using `into glob` command: `let f = "a[bc]d.txt"; ls ($f | into glob)`. It's useful for `ls` and `du` commands.
3. annotate variable with `glob` type: `let f: glob = "a[bc]d.txt"; rm $f`. It's simple to write, but doesn't work with external command like `^rm $f`.

Changing the current directory

@code

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:

@code

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.

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:

@code

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

@code

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

@code

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:

@code

Thinking in Nu

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

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

Nushell isn't bash

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

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

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

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

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

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

Variables are immutable

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

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

```
> ls | enumerate | each { |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.

Nushell cheat sheet

Data types

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

converts string to integer

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

converts present date to provided time zone

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

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

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

converts list of strings to yaml

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

prints the table

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

selects two columns from the table and prints their values

Strings

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

prints `greetings, Alice!`

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

```
╭───┬─────────╮
│ 0 │ one     │
│ 1 │ two     │
│ 2 │ three   │
╰───┴─────────╯
```

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

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

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

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

joins the list of strings using provided delimiter

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

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

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

```
╭───┬─────────┬─────────╮
│ # │  shell  │ version │
├───┼─────────┼─────────┤
│ 0 │ Nushell │ 0.80    │
╰───┴─────────┴─────────╯
```

parses the string to columns

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

parses comma separated values (csv)

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

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

Lists

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

```
╭───┬────────╮
│ 0 │ foo    │
│ 1 │ beeze  │
│ 2 │ bar    │
│ 3 │ baz    │
╰───┴────────╯
```

inserts `beeze` value at the 2nd index of the list

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

updates 2nd value to 10

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

adds value at the beginning of the list

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

adds value at the end of the list

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

creates slice of first two values from `flowers` list

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

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

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

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

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

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

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

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

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

gives access to the 3rd item in the list

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

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

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

0	-1
1	-2

creates slice of items that satisfy provided condition

Tables

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

sorting table by size of files

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

sorting table by size of files and show first 5 entries

```
> let $a = [[first_column second_column third_column]; [foo bar snooze]]
> let $b = [[first_column second_column third_column]; [hex seeze feeze]]
> $a | append $b
```

```
╭───┬──────────────┬───────────────┬──────────────╮
│ # │ first_column │ second_column │ third_column │
├───┼──────────────┼───────────────┼──────────────┤
│ 0 │ foo          │ bar           │ snooze       │
│ 1 │ hex          │ seeze         │ feeze        │
╰───┴──────────────┴───────────────┴──────────────╯
```

concatenate two tables with same columns

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

```
╭───┬──────────────────────┬───────╮
│ # │         team         │ score │
├───┼──────────────────────┼───────┤
│ 0 │ Boston Celtics       │   311 │
│ 1 │ Golden State Warriors │   245 │
╰───┴──────────────────────┴───────╯
```

remove the last column of a table

Files & Filesystem

```
> start file.txt
```

opens a text file with the default text editor

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

saves a string to text file

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

appends a string to the end of file.txt

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

saves a record to file.json

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

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

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

runs cargo test whenever a Rust file changes

Custom Commands

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

custom command with parameter type set to string

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

custom command with default parameter set to nushell

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

> greet world --age 10
```

passing named parameter by defining flag for custom commands

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

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

```
def greet [...name: string] {
    print "hello all:"
```

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

> greet earth mars jupiter venus
```

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

Variables & Subexpressions

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

an immutable variable cannot change its value after declaration

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

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

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

declaring a mutable variable with mut key word

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

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

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

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

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

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

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

using subexpression by wrapping the expression with parentheses ()

Modules

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

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

> use greetings hello
> hello "world"
```

using inline module

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

> use greetings.nu
> $env.MYNAME
Arthur, King of the Britons
> greetings hello
hello Arthur, King of the Britons!
```

importing module from file and using its environment in current scope

```
# greetings.nu
export def hello [name: string] {
    $"hello ($name)!"
}

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

export def main [] {
    "greetings and salutations!"
}

> use greetings.nu
> greetings
greetings and salutations!
> greetings hello world
hello world!
```

using main command in module

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

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

Integers

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

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

Decimals (floats)

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

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

Strings

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

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

See Working with strings and Handling Strings for details.

Booleans

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

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

Dates

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

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

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

Durations

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

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

You can make fractional durations:

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

And you can do calculations with durations:

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

File sizes

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

The full list of filesize units are:

- `b`: bytes
- `kb`: kilobytes (aka 1000 bytes)
- `mb`: megabytes
- `gb`: gigabytes
- `tb`: terabytes

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

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

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

Ranges

A range is a way of expressing a sequence of integer or float values from start to finish. They take the form <start>..<end>. For example, the range `1..3` means the numbers 1, 2, and 3.

You can also easily create lists of characters with a form similar to ranges with the command `seq char` as well as with dates using the `seq date` command.

Specifying the step

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

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

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.

Watch out for displaying open-ended ranges like just entering `3..` into the command line. It will keep printing out numbers very quickly until you stop it with something like Ctr + c.

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-down rather than left-right.

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
```

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

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

```
╭────────┬─────────╮
│ name   │ alice   │
│ age    │ 50      │
│ hobby  │ cricket │
╰────────┴─────────╯
```

Lists

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

```
> [sam fred george]
```

```
╭───┬─────────╮
│ 0 │ sam     │
│ 1 │ fred    │
│ 2 │ george  │
╰───┴─────────╯
```

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 │
╰───┴───╯
```

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

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

```
╭───┬───╮
```

```
| 0 | 1 |
| 1 | 2 |
| 2 | 3 |
| 3 | 4 |
| 4 | 5 |
```

Tables

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

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

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

```
| # | column1 | column2 |
|---|---------|---------|
| 0 | Value1  | Value2  |
| 1 | Value3  | Value4  |
```

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

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

```
| # | name | rank |
|---|------|------|
| 0 | sam  |   10 |
| 1 | bob  |    7 |
```

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

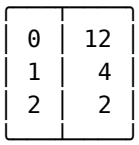
You can combine list and record data access syntax to navigate tables. When used on tables, these access chains are called "cell paths".

You can access individual rows by number to obtain records:

@code

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

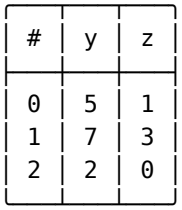
```
> [{x:12 y:5} {x:4 y:7} {x:2 y:2}].x
```



```
╭───┬────╮
│ 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:

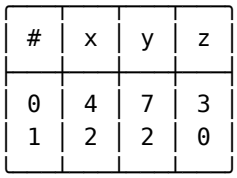
```
> [{x:0 y:5 z:1} {x:4 y:7 z:3} {x:2 y:2 z:0}] | select y z
```



```
╭───┬───┬───╮
│ # │ 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
```

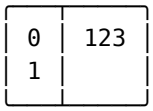


```
╭───┬───┬───┬───╮
│ # │ x │ y │ z │
├───┼───┼───┼───┤
│ 0 │ 4 │ 7 │ 3 │
│ 1 │ 2 │ 2 │ 0 │
╰───┴───┴───┴───╯
```

Optional cell paths

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

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



```
╭───┬─────╮
│ 0 │ 123 │
│ 1 │     │
╰───┴─────╯
```

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

Closures

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

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

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

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

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

Blocks

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

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

Null

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

`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
   ╰────
```

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

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

Loading data

Earlier, we saw how you can use commands like `ls`, `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:

@code

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

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

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

```
> open README.md
```

We're shown the contents of the file.

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

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):

```
{
  menus: [
    # Configuration for default nushell menus
```

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

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

One limitation of NUON currently is that it cannot represent all of the Nushell data types. Most notably, NUON does not allow the serialization of 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 list. Our next step is to see if we can split up the rows into something a little more useful. For that, we'll use the `split` 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:

@code

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>", "Sophia Turner
<547158+sophiajt@users.noreply.github.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 `http get` command. This will fetch the contents of the URL from the internet and return it:

@code

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

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.

Notes on errors when piping commands

Sometimes, it might be unclear as to why you cannot pipe to a command.

For example, PowerShell users may be used to piping the output of any internal PowerShell command directly to another, e.g.:

```
echo 1 | sleep
```

(Where for PowerShell, `echo` is an alias to `Write-Output` and `sleep` is to `Start-Sleep`.)

However, it might be surprising that for some commands, the same in Nushell errors:

```
> echo 1sec | sleep
Error: nu::parser::missing_positional

  × Missing required positional argument.
   ╭─[entry #53:1:1]
 1 │ echo 1sec | sleep
   ╰────
  help: Usage: sleep <duration> ...(rest) . Use `--help` for more information.
```

While there is no steadfast rule, Nu generally tries to copy established conventions, or do what 'feels right'. And with `sleep`, this is actually the same behaviour as Bash.

Many commands do have piped input/output however, and if it's ever unclear, you can see what you can give to a command by invoking `help <command name>`:

```
> help sleep
Delay for a specified amount of time.

Search terms: delay, wait, timer

Usage:
  > sleep <duration> ...(rest)

Flags:
  -h, --help - Display the help message for this command

Parameters:
  duration <duration>: Time to sleep.
  ...rest <duration>: Additional time.

Input/output types:
  ╭───┬──────────┬──────────╮
  │ # │  input   │ output   │
```

```
╭───┬─────────┬─────────╮
│ 0 │ nothing │ nothing │
╰───┴─────────┴─────────╯
```

In this case, sleep takes `nothing` and instead expects an argument.

So, we can supply the output of the `echo` command as an argument to it: `echo 1sec | sleep $in` or `sleep (echo 1sec)`

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.

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

Output result to external commands

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

```
> ls /usr/share/nvim/runtime/
```

```
╭────┬──────────────────────────────────┬──────┬─────────┬──────────────╮
│  # │               name               │ type │  size   │   modified   │
├────┼──────────────────────────────────┼──────┼─────────┼──────────────┤
│  0 │ /usr/share/nvim/runtime/autoload │ dir  │  4.1 KB │ 2 days ago   │
..........
..........
..........

│ 31 │ /usr/share/nvim/runtime/tools    │ dir  │  4.1 KB │ 2 days ago   │
│ 32 │ /usr/share/nvim/runtime/tutor    │ dir  │  4.1 KB │ 2 days ago   │
├────┼──────────────────────────────────┼──────┼─────────┼──────────────┤
│  # │               name               │ type │  size   │   modified   │
╰────┴──────────────────────────────────┴──────┴─────────┴──────────────╯
```

You decided to use `grep` and pipe 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
```

Command Output in Nushell

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

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

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

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

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

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

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

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

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

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

Working with strings

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

String formats at a glance

Format of string	Example	Escapes	Notes
Single-quoted string	`'[^\n]+'`	None	Cannot contain any '
Backtick string	`` `[^\n]+` ``	None	Cannot contain any backticks | | Double-quoted string |"The"| 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 `` ` `` character also exist:

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

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

Double-quoted Strings

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

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

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

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

Nushell currently supports the following escape characters:

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

Bare strings

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

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

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

```
> hello
Error: nu::shell::external_command

  × External command failed
   ╭─[entry #5:1:1]
 1 │ hello
   · ──┬──
   ·   ╰── executable was not found
   ╰────
  help: program not found
```

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

```
> true | describe
bool
> [true] | describe
list<bool>
> [trueX] | describe
list<string>
> trueX | describe
Error: nu::shell::external_command
```

```
  × External command failed
   ╭─[entry #5:1:1]
 1 │ trueX | describe
   · ──┬──
   ·   └── executable was not found
   ╰────
  help: program not found
```

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

Strings as external commands

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

```
^'C:\Program Files\exiftool.exe'

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

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

Appending and Prepending to strings

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

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

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

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

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

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

You can also use reduce:

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

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

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

String interpolation

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

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

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

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

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

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

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

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

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

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

Splitting strings

The `split row` 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 `fill` command you can add padding to a string. Padding adds characters to string until it's a certain length. For example:

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

Reversing strings

This can be done easily with the `str reverse` 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?\w+) is (?
P<adjective>\w+)'
```

#	subject	adjective
0	all data	structured

If a string is known to contain comma-separated, tab-separated or multi-space-separated data, you can use `from csv`, `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)'`

From string

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

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
# [1, 2, 10, 3, 4]
```

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

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

Removing or adding items from list

In addition to `insert` 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)
let colors = ($colors ++ "blue")
let colors = ("black" ++ $colors)
$colors # [black red yellow green purple blue]
```

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

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

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

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

And from the beginning of a list,

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

Iterating over lists

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

```
let names = [Mark Tami Amanda Jeremy]
$names | each { |it| $"Hello, ($it)!" }
# Outputs "Hello, Mark!" and three more similar lines.

$names | enumerate | each { |it| $"($it.index + 1) - ($it.item)" }
# Outputs "1 - Mark", "2 - Tami", etc.
```

The `where` 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, use the `enumerate` filter. For example:

```
let scores = [3 8 4]
$"total = ($scores | reduce { |it, acc| $acc + $it })" # total = 15

$"total = ($scores | math sum)" # easier approach, same result

$"product = ($scores | reduce --fold 1 { |it, acc| $acc * $it })" # product = 96

$scores | enumerate | reduce --fold 0 { |it, acc| $acc + $it.index * $it.item } # 0*3
+ 1*8 + 2*4 = 16
```

Accessing the list

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

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

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

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

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

The `length` 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:

```
let colors = [red green blue]
# Do any color names end with "e"?
$colors | any {|it| $it | str ends-with "e" } # true

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

let scores = [3 8 4]
# Are any scores greater than 7?
$scores | any {|it| $it > 7 } # true

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

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

```
let colors = [red green blue]
# Do all color names end with "e"?
$colors | all {|it| $it | str ends-with "e" } # false

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

let scores = [3 8 4]
# Are all scores greater than 7?
$scores | all {|it| $it > 7 } # false

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

Converting the list

The `flatten` 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 |
format date '%Y.%m.%d %H:%M')}
```

Working with tables

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

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

```
> ls
```

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

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

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 arguments provided to `select` and `get` are cell paths, a fundamental part of Nu's query language. In addition to simple queries like `get name`, you can also do things like `get name?` (will

replace missing values with `null` instead of returning an error) or `get some.deeply.nested.column` for nested data.

Changing data in a table

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

Concatenating Tables

We can concatenate tables using `append`:

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

```
╭───┬───┬───╮
│ # │ a │ b │
├───┼───┼───┤
│ 0 │ 1 │ 2 │
│ 1 │ 3 │ 4 │
╰───┴───┴───╯
```

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

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

```
╭───┬───┬───┬───╮
│ # │ a │ b │ c │
├───┼───┼───┼───┤
│ 0 │ 1 │ 2 │   │
│ 1 │ 3 │ 4 │   │
│ 2 │ 3 │   │ 4 │
╰───┴───┴───┴───╯
```

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

```
$first ++ $second ++ $third
```

```
╭───┬───┬───┬───╮
│ # │ a │ b │ c │
├───┼───┼───┼───┤
│ 0 │ 1 │ 2 │   │
│ 1 │ 3 │ 4 │   │
│ 2 │ 3 │   │ 4 │
╰───┴───┴───┴───╯
```

Merging Tables

We can use the `merge` 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   |
...
```

Rejecting/Deleting columns

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

```
> ls -l / |reject readonly num_links inode created accessed modified
```

```
╭────┬─────────┬─────────┬─────────┬────────────┬──────┬────────┬──────────╮
│  # │  name   │  type   │ target  │    mode    │ uid  │ group  │   size   │
├────┼─────────┼─────────┼─────────┼────────────┼──────┼────────┼──────────┤
│  0 │ /bin    │ symlink │ usr/bin │ rwxrwxrwx  │ root │ root   │     7 B  │
│  1 │ /boot   │ dir     │         │ rwxr-xr-x  │ root │ root   │ 1.0 KB   │
│  2 │ /dev    │ dir     │         │ rwxr-xr-x  │ root │ root   │ 4.1 KB   │
│  3 │ /etc    │ dir     │         │ rwxr-xr-x  │ root │ root   │ 3.6 KB   │
│  4 │ /home   │ dir     │         │ rwxr-xr-x  │ root │ root   │    12 B  │
│  5 │ /lib    │ symlink │ usr/lib │ rwxrwxrwx  │ root │ root   │     7 B  │
│  6 │ /lib64  │ symlink │ usr/lib │ rwxrwxrwx  │ root │ root   │     7 B  │
│  7 │ /mnt    │ dir     │         │ rwxr-xr-x  │ root │ root   │     0 B  │
...
```

Programming in Nu

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

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

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

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

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

The last three sections are aimed at organizing your code:

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

With modules, 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 environments or overriding sets of default commands with custom variants.

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

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

Custom commands

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

This is where custom commands come in.

An example definition of a custom command:

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

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

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

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

```
> greet "world"
```

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

```
___│_______
 0 │ hello
 1 │ world
___│_______
```

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 0.86.0):

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

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

Parameters with a default value

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

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

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

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

You can also combine a default value with a type requirement:

```
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 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
```

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 instead.

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

Flags

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

For example:

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

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

You can call the above using:

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

Or:

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

Or even leave the flag off altogether:

```
> greet world
```

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

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

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

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

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

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

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

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

And the definition can be either called as:

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

Or just without the switch flag:

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

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

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

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

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

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

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

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

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

Rest parameters

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

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

greet earth mars jupiter venus
```

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

Rest parameters can be used together with positional parameters:

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

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

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

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

Documenting your command

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

Taking our previous example:

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

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

```
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.

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

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

```
A greeting command that can greet the caller

Usage:
  > greet <name> {flags}

Parameters:
  <name> The name of the person to greet

Flags:
  -h, --help: Display this help message
  -a, --age <integer>: The age of the person
```

Pipeline Output

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

```
> ls | get name
```

Let's move `ls` 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 replacement of command calls (both external and internal commands). This allows you to create a shorthand name for a longer command, including its default arguments.

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

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

We can now call this alias:

```
> ll
```

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

```
> ll -a
```

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

List all loaded aliases

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

Persisting

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

```
$env.config = {
    # main configuration
}

alias ll = ls -l
```

```
# some other config and script loading
```

Piping in aliases

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

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

See more in the custom commands section of this book.

Or a more idiomatic example with nushell internal commands

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

displaying all listed files and folders in a grid.

Replacing existing commands using aliases

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

How to back up a command like `ls`:

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

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

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

```
def ls [] { ls }; ls    # Do *NOT* do this! This will throw a recursion error

#output:
#Error: nu::shell::recursion_limit_reached
#
#  × Recursion limit (50) reached
#     ╭─[C:\Users\zolodev\AppData\Roaming\nushell\config.nu:807:1]
# 807 │
# 808 │ def ls [] { ls }; ls
#     ·             ──┬─
#     ·               ╰── This called itself too many times
#     ╰────
```

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

```
# An escape hatch to have access to the original ls command
alias core-ls = ls

# Call the built-in ls command with a path parameter
def old-ls [path] {
  core-ls $path | sort-by type name -i
}
```

```
# Shadow the ls command so that you always have the sort type you want
def ls [path?] {
  if $path == null {
    old-ls .
  } else {
    old-ls $path
  }
}
```

Operators

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

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

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

Order of Operations

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

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

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

Types

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

```
> "spam" - 1
Error: nu::parser::unsupported_operation (link)

  × Types mismatched for operation.
   ╭─[entry #49:1:1]
 1 │ "spam" - 1
   · ──────┬── ┬ ┬
   ·       │   │ └── int
   ·       │   └── doesn't support these values.
   ·       └── string
   ╰────
  help: Change string or int to be the right types and try again.
```

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

Regular Expression / string-contains Operators

The `=~` and `!~` operators provide a convenient way to evaluate regular expressions. 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.

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)
```

Spread operator

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

There are three places you can use the spread operator:

- In list literals
- In record literals
- In command calls

In list literals

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

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

```
| 8 | ...Nemo                          |
└───┴──────────────────────────────────┘
```

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

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

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

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

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

```
> [ ... [] ]
╭───┬──────────────────╮
│ 0 │ ...              │
│ 1 │ [list 0 items]   │
╰───┴──────────────────╯
```

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

In record literals

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

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

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

```
> {
    ...$config,
    users: [alice bob],
    ...{ url: example.com },
    ...(sys | get mem)
  }
╭─────────────┬──────────────────╮
│ path        │ /tmp             │
│ limit       │ 5                │
│             │ ╭───┬─────────╮  │
│ users       │ │ 0 │ alice   │  │
│             │ │ 1 │ bob     │  │
│             │ ╰───┴─────────╯  │
│ url         │ example.com      │
│ total       │ 8.3 GB           │
│ free        │ 2.6 GB           │
│ used        │ 5.7 GB           │
│ available   │ 2.6 GB           │
```

```
| swap total | 2.1 GB        |
| swap free  | 18.0 MB       |
| swap used  | 2.1 GB        |
└────────────┴───────────────┘
```

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

In command calls

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Variables and Subexpressions

There are two types of evaluation expressions in Nushell: variables and subexpressions. You know that you're looking at an evaluation expression because it begins with a dollar sign (`$`). This indicates that when Nushell gets the value in this position, it will need to run an evaluation step to process

the expression and then use the resulting value. Both evaluation expression forms support a simple form and a 'path' form for working with more complex data.

Variables

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

Types of Variables

Immutable Variables

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

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

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

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

Mutable Variables

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

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

There are a couple of assignment operators used with mutable variables

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

Note

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

More on Mutability

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

```
# naive method to count number of elements in a list
mut x = 0

[1 2 3] | each { $x += 1 }   # error: $x is captured in a closure
```

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

Constant Variables

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

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

Variable Names

Variable names in Nushell come with a few restrictions as to what characters they can contain. In particular, they cannot contain these characters:

```
.  [  (  {  +  -  *  ^  /  =  !  <  >  &  |
```

It is common for some scripts to declare variables that start with `$`. This is allowed, and it is equivalent to the `$` not being there at all.

```
> let $var = 42
# identical to `let var = 42`
```

Variable Paths

A variable path works by reaching inside of the contents of a variable, navigating columns inside of it, to reach a final value. Let's say instead of `4`, we had assigned a table value:

```
> let my_value = [[name]; [testuser]]
```

We can use a variable path to evaluate the variable `$my_value` and get the value from the `name` column in a single step:

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

Sometimes, we don't really know the contents of a variable. Accessing values as shown above can result in errors if the path used does not exist. To more robustly handle this, we can use the question mark operator to return `null` in case the path does not exist, instead of an error, then we would write custom logic to handle the `null`.

For example, here, if row `0` does not exist on `name`, then `null` is returned. Without the question mark operator, an error would have been raised instead

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

The question mark operator can be used to 'guard' any path

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

Subexpressions

You can always evaluate a subexpression and use its result by wrapping the expression with parentheses `()`. Note that previous versions of Nushell (prior to 0.32) used `$()`.

The parentheses contain a pipeline that will run to completion, and the resulting value will then be used. For example, `(ls)` would run the `ls` command and give back the resulting table and `(git branch --show-current)` runs the external git command and returns a string with the name of the current branch. You can also use parentheses to run math expressions like `(2 + 3)`.

Subexpressions can also be pipelines and not just single commands. If we wanted to get a table of files larger than ten kilobytes, we could use a subexpression to run a pipeline and assign its result to a variable:

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

#	name	type	size	modified
0	Cargo.lock	File	155.3 KB	17 hours ago
1	README.md	File	15.9 KB	17 hours ago

Subexpressions and paths

Subexpressions also support paths. For example, let's say we wanted to get a list of the filenames in the current directory. One way to do this is to use a pipeline:

```
> 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 could 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 the short-hand syntax to work, the column name must appear on the left-hand side of the operation (like `size` in `size > 10kb`).

Control Flow

Nushell provides several commands that help determine how different groups of code are executed. In programming languages this functionality is often referred to as *control flow*.

One thing to note is that all of the commands discussed on this page use blocks. This means you can mutate environmental variables and other mutable variables in them.

Already covered

Below we cover some commands related to control flow, but before we get to them, it's worthwhile to note there are several pieces of functionality and concepts that have already been covered in other sections that are also related to control flow or that can be used in the same situations. These include:

- Pipes on the pipelines page.
- Closures on the types of data page.
- Iteration commands on the working with lists page. Such as:
  - `each`
  - `where`
  - `reduce`

Choice (Conditionals)

The following commands execute code based on some given condition.

The choice/conditional commands are expressions so they return values, unlike the other commands on this page. This means the following works.

```
> 'foo' | if $in == 'foo' { 1 } else { 0 } | $in + 2
3
```

`if`

`if` evaluates branching blocks of code based on the results of one or more conditions similar to the "if" functionality in other programming languages. For example:

```
> if $x > 0 { 'positive' }
```

Returns `'positive'` when the condition is `true` (`$x` is greater than zero) and `null` when the condition is `false` (`$x` is less than or equal to zero).

We can add an `else` branch to the `if` after the first block which executes and returns the resulting value from the `else` block when the condition is `false`. For example:

```
> if $x > 0 { 'positive' } else { 'non-positive' }
```

This time it returns `'positive'` when the condition is `true` (`$x` is greater than zero) and `'non-positive'` when the condition is `false` (`$x` is less than or equal to zero).

We can also chain multiple `if`s together like the following:

```
> if $x > 0 { 'positive' } else if $x == 0 { 'zero' } else { "negative" }
```

When the first condition is `true` (`$x` is greater than zero) it will return `'positive'`, when the first condition is `false` and the next condition is `true` (`$x` equals zero) it will return `'zero'`, otherwise it will return `'negative'` (when `$x` is less than zero).

`match`

`match` executes one of several conditional branches based on the value given to match. You can also do some pattern matching to unpack values in composite types like lists and records.

Basic usage of `match` can conditionally run different code like a "switch" statement common in other languages. `match` checks if the value after the word `match` is equal to the value at the start of each branch before the `=>` and if it does, it executes the code after that branch's `=>`.

```
> match 3 {
    1 => 'one',
    2 => {
        let w = 'w'
        't' + $w + 'o'
    },
    3 => 'three',
    4 => 'four'
}
three
```

The branches can either return a single value or, as shown in the second branch, can return the results of a block.

Catch all branch

You can also have a catch all condition for when the given value doesn't match any of the other conditions by having a branch whose matching value is `_`.

```
> let foo = match 7 {
    1 => 'one',
    2 => 'two',
    3 => 'three',
    _ => 'other number'
}
> $foo
other number
```

(Reminder, `match` is an expression which is why we can assign the result to `$foo` here).

Pattern Matching

You can "unpack" values from types like lists and records with pattern matching. You can then assign variables to the parts you want to unpack and use them in the matched expressions.

```
> let foo = { name: 'bar', count: 7 }
> match $foo {
    { name: 'bar', count: $it } => ($it + 3),
    { name: _, count: $it } => ($it + 7),
    _ => 1
}
10
```

The `_` in the second branch means it matches any record with field `name` and `count`, not just ones where `name` is `'bar'`.

Guards

You can also add an additional condition to each branch called a "guard" to determine if the branch should be matched. To do so, after the matched pattern put `if` and then the condition before the `=>`.

```
> let foo = { name: 'bar', count: 7 }
> match $foo {
    { name: 'bar', count: $it } if $it < 5 => ($it + 3),
    { name: 'bar', count: $it } if $it >= 5 => ($it + 7),
    _ => 1
}
14
```

You can find more details about `match` in the pattern matching cookbook page.

Loops

The loop commands allow you to repeat a block of code multiple times.

Loops and other iterating commands

The functionality of the loop commands is similar to commands that apply a closure over elements in a list or table like `each` or `where` and many times you can accomplish the same thing with either. For example:

```
> mut result = []
> for $it in [1 2 3] { $result = ($result | append ($it + 1)) }
> $result
```

```
╭───┬───╮
│ 0 │ 2 │
│ 1 │ 3 │
│ 2 │ 4 │
╰───┴───╯
```

```
> [1 2 3] | each { $in + 1 }
```

```
╭───┬───╮
│ 0 │ 2 │
│ 1 │ 3 │
│ 2 │ 4 │
╰───┴───╯
```

While it may be tempting to use loops if you're familiar with them in other languages, it is considered more in the Nushell-style (idiomatic) to use commands that apply closures when you can solve a problem either way. The reason for this is because of a pretty big downside with using loops.

Loop disadvantages

The biggest downside of loops is that they are statements, unlike `each` which is an expression. Expressions, like `each` always result in some output value, however statements do not.

This means that they don't work well with immutable variables and using immutable variables is considered a more Nushell-style. Without a mutable variable declared beforehand in the example in the previous section, it would be impossible to use `for` to get the list of numbers with incremented numbers, or any value at all.

Statements also don't work in Nushell pipelines which require some output. In fact Nushell will give an error if you try:

```
> [1 2 3] | for x in $in { $x + 1 } | $in ++ [5 6 7]
Error: nu::parser::unexpected_keyword

  × Statement used in pipeline.
   ╭─[entry #5:1:1]
 1 │ [1 2 3] | for x in $in { $x + 1 } | $in ++ [5 6 7]
   ·           ─┬─
   ·            ╰── not allowed in pipeline
   ╰────
  help: 'for' keyword is not allowed in pipeline. Use 'for' by itself, outside of a
pipeline.
```

Because Nushell is very pipeline oriented, this means using expression commands like `each` is typically more natural than loop statements.

Loop advantages

If loops have such a big disadvantage, why do they exist? Well, one reason is that closures, like `each` uses, can't modify mutable variables in the surrounding environment. If you try to modify a mutable variable in a closure you will get an error:

```
> mut foo = []
> [1 2 3] | each { $foo = ($foo | append ($in + 1)) }
Error: nu::parser::expected_keyword

  × Capture of mutable variable.
   ╭─[entry #8:1:1]
 1 │ [1 2 3] | each { $foo = ($foo | append ($in + 1)) }
   ·                  ──┬─
   ·                    ╰── capture of mutable variable
   ╰────
```

If you modify an environmental variable in a closure, you can, but it will only modify it within the scope of the closure, leaving it unchanged everywhere else. Loops, however, use blocks which means they can modify a regular mutable variable or an environmental variable within the larger scope.

```
> mut result = []
> for $it in [1 2 3] { $result = ($result | append ($it + 1)) }
> $result
╭───┬───╮
│ 0 │ 2 │
│ 1 │ 3 │
│ 2 │ 4 │
╰───┴───╯
```

`for`

`for` loops over a range or collection like a list or a table.

```
> for x in [1 2 3] { $x * $x | print }
1
4
9
```

Expression command alternatives

- `each`
- `par-each`
- `where`/`filter`
- `reduce`

`while`

`while` loops the same block of code until the given condition is `false`.

```
> mut x = 0; while $x < 10 { $x = $x + 1 }; $x
10
```

Expression command alternatives

The "until" and other "while" commands

- `take until`
- `take while`
- `skip until`
- `skip while`

`loop`

`loop` loops a block infinitely. You can use `break` (as described in the next section) to limit how many times it loops. It can also be handy for continuously running scripts, like an interactive prompt.

```
> mut x = 0; loop { if $x > 10 { break }; $x = $x + 1 }; $x
11
```

`break`

`break` will stop executing the code in a loop and resume execution after the loop. Effectively "break"ing out of the loop.

```
> for x in 1..10 { if $x > 3 { break }; print $x }
1
2
3
```

`continue`

`continue` will stop execution of the current loop, skipping the rest of the code in the loop, and will go to the next loop. If the loop would normally end, like if `for` has iterated through all the given elements, or if `while`'s condition is now false, it won't loop again and execution will continue after the loop block.

```
> mut x = -1; while $x <= 6 { $x = $x + 1; if $x mod 3 == 0 { continue }; print $x }
1
2
4
5
7
```

Errors

`error make`

`error make` creates an error that stops execution of the code and any code that called it, until either it is handled by a `try` block, or it ends the script and outputs the error message. This functionality is the same as "exceptions" in other languages.

```
> print 'printed'; error make { msg: 'Some error info' }; print 'unprinted'
printed
Error:   × Some error info
   ╭─[entry #9:1:1]
 1 │ print 'printed'; error make { msg: 'Some error info' }; print 'unprinted'
   ·                  ─────┬─────
   ·                       └── originates from here
   ╰────
```

The record passed to it provides some information to the code that catches it or the resulting error message.

You can find more information about `error make` and error concepts on the Creating your own errors page.

`try`

`try` will catch errors created anywhere in the `try`'s code block and resume execution of the code after the block.

```
> try { error make { msg: 'Some error info' }}; print 'Resuming'
Resuming
```

This includes catching built in errors.

```
> try { 1 / 0 }; print 'Resuming'
Resuming
```

The resulting value will be `nothing` if an error occurs and the returned value of the block if an error did not occur.

If you include a `catch` block after the `try` block, it will execute the code in the `catch` block if an error occurred in the `try` block.

```
> try { 1 / 0 } catch { 'An error happened!' } | $in ++ ' And now I am resuming.'
An error happened! And now I am resuming.
```

It will not execute the `catch` block if an error did not occur.

Other

`return`

`return` Ends a closure or command early where it is called, without running the rest of the command/closure, and returns the given value. Not often necessary since the last value in a closure or command is also returned, but it can sometimes be convenient.

```
def 'positive-check' [it] {
    if $it > 0 {
        return 'positive'
    };

    'non-positive'
}
```

```
> positive-check 3
positive

> positive-check (-3)
non-positive

> let positive_check = {|it| if $it > 0 { return 'positive' }; 'non-positive' }

> do $positive_check 3
positive

> do $positive_check (-3)
non-positive
```

Scripts

In Nushell, you can write and run scripts in the Nushell language. To run a script, you can pass it as an argument to the `nu` commandline application:

```
> nu myscript.nu
```

This will run the script to completion in a new instance of Nu. You can also run scripts inside the *current* instance of Nu using `source`:

```
> 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
```

Argument Type Interpretation

By default, arguments provided to a script are interpreted with the type `Type::Any`, implying that they are not constrained to a specific data type and can be dynamically interpreted as fitting any of the available data types during script execution.

In the previous example, `main [x: int]` denotes that the argument x should possess an integer data type. However, if arguments are not explicitly typed, they will be parsed according to their apparent data type.

For example:

```
# implicit_type.nu
def main [x] {
  $"Hello ($x | describe) ($x)"
}

# explicit_type.nu
def main [x: string] {
  $"Hello ($x | describe) ($x)"
}
```

```
> nu implicit_type.nu +1
Hello int 1

> nu explicit_type.nu +1
Hello string +1
```

Subcommands

A script can have multiple sub-commands like `run`, `build`, etc. which allows to execute a specific main sub-function. The important part is to expose them correctly with `def main [] {}`. See more details in the Custom Command section.

For example:

```
# myscript.nu
def "main run" [] {
    print "running"
}

def "main build" [] {
    print "building"
}

# important for the command to be exposed to the outside
def main [] {}
```

```
> nu myscript.nu build
building
> nu myscript.nu run
running
```

Shebangs (`#!`)

On Linux and macOS you can optionally use a shebang to tell the OS that a file should be interpreted by Nu. For example, with the following in a file named `myscript`:

```
#!/usr/bin/env nu
"Hello World!"
```

```
> ./myscript
Hello World!
```

For script to have access to standard input, `nu` should be invoked with `--stdin` flag:

```
#!/usr/bin/env -S nu --stdin
echo $"stdin: ($in)"
```

```
> echo "Hello World!" | ./myscript
stdin: Hello World!
```

Modules

Similar to many other programming languages, Nushell also has modules to organize your code. Each module is a "bag" containing a bunch of definitions (typically commands) that you can export (take out of the bag) and use in your current scope. Since Nushell is also a shell, modules allow you to modify environment variables when importing them.

Quick Overview

There are three ways to define a module in Nushell:

1. "inline"
   - `module spam { ... }`
2. from a file
   - using a .nu file as a module
3. from a directory
   - directory must contain a `mod.nu` file

In Nushell, creating a module and importing definitions from a module are two different actions. The former is done using the `module` keyword. The latter using the `use` keyword. You can think of `module` as "wrapping definitions into a bag" and `use` as "opening a bag and taking definitions from it". In most cases, calling `use` will create the module implicitly, so you typically don't need to use `module` that much.

You can define the following things inside a module:

- Commands* (`def`)
- Aliases (`alias`)
- Known externals* (`extern`)
- Submodules (`module`)
- Imported symbols from other modules (`use`)
- Environment setup (`export-env`)

Only definitions marked with `export` are possible to access from outside of the module ("take out of the bag"). Definitions not marked with `export` are allowed but are visible only inside the module (you could call them private). (*`export-env` is special and does not require `export`.*)

**These definitions can also be named `main` (see below).*

"Inline" modules

The simplest (and probably least useful) way to define a module is an "inline" module can be defined like this:

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

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

use greetings hello
```

```
hello "world"
```

You can paste the code into a file and run it with nu, or type into the REPL.

First, we create a module (put `hello` and `hi` commands into a "bag" called `greetings`), then we import the `hello` command from the module (find a "bag" called `greetings` and take `hello` command from it) with `use`.

Modules from files

A .nu file can be a module. Just take the contents of the module block from the example above and save it to a file `greetings.nu`. The module name is automatically inferred as the stem of the file ("greetings").

```
# greetings.nu

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

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

then

```
> use greetings.nu hello

> hello
```

The result should similar as in the previous section.

> **Note** that the `use greetings.nu hello` call here first implicitly creates the `greetings` module, then takes `hello` from it. You could also write it as `module greetings.nu`, `use greetings hello`. Using `module` can be useful if you're not interested in any definitions from the module but want to, e.g., re-export the module (`export module greetings.nu`).

Modules from directories

Finally, a directory can be imported as a module. The only condition is that it needs to contain a `mod.nu` file (even empty, which is not particularly useful, however). The `mod.nu` file defines the root module. Any submodules (`export module`) or re-exports (`export use`) must be declared inside the `mod.nu` file. We could write our `greetings` module like this:

In the following examples, `/` is used at the end to denote that we're importing a directory but it is not required.

```
# greetings/mod.nu

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

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

then

```
> use greetings/ hello

> hello
```

The name of the module follows the same rule as module created from a file: Stem of the directory name, i.e., the directory name, is used as the module name. Again, you could do this as a two-step action using `module` and `use` separately, as explained in the previous section.

You can define `main` command inside `mod.nu` to create a command named after the module directory.

Import Pattern

Anything after the `use` keyword forms an **import pattern** which controls how the definitions are imported. The import pattern has the following structure `use head members...` where `head` defines the module (name of an existing module, file, or directory). The members are optional and specify what exactly should be imported from the module.

Using our `greetings` example:

```
use greetings
```

imports all symbols prefixed with the `greetings` namespace (can call `greetings hello` and `greetings hi`).

```
use greetings hello
```

will import the `hello` command directly without any prefix.

```
use greetings [hello, hi]
```

imports multiple definitions<> directly without any prefix.

```
use greetings *
```

will import all names directly without any prefix.

`main`

Exporting a command called `main` from a module defines a command named as the module. Let's extend our `greetings` example:

```
# greetings.nu

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

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

export def main [] {
    "greetings and salutations!"
}
```

then

```
> use greetings.nu

> greetings
greetings and salutations!
```

```
> greetings hello world
hello world!
```

The `main` is exported only when

- no import pattern members are used (`use greetings.nu`)
- glob member is used (`use greetings.nu *`)

Importing definitions selectively (`use greetings.nu hello` or `use greetings.nu [hello hi]`) does not define the `greetings` command from `main`. You can, however, selectively import `main` using `use greetings main` (or `[main]`) which defines *only* the `greetings` command without pulling in `hello` or `hi`.

Apart from commands (`def`, `def --env`), known externals (`extern`) can also be named `main`.

Submodules and subcommands

Submodules are modules inside modules. They are automatically created when you call `use` on a directory: Any .nu files inside the directory are implicitly added as submodules of the main module. There are two more ways to add a submodule to a module:

1. Using `export module`
2. Using `export use`

The difference is that `export module some-module` *only* adds the module as a submodule, while `export use some-module` *also* re-exports the submodule's definitions. Since definitions of submodules are available when importing from a module, `export use some-module` is typically redundant, unless you want to re-export its definitions without the namespace prefix.

> **Note** `module` without `export` defines only a local module, it does not export a submodule.

Let's illustrate this with an example. Assume three files:

```
# greetings.nu

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

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

export def main [] {
    "greetings and salutations!"
}
```

```
# animals.nu

export def dog [] {
    "haf"
}

export def cat [] {
    "meow"
}
```

```
# voice.nu
```

```
export use greetings.nu *

export module animals.nu
```

Then:

```
> use voice.nu

> voice animals dog
haf

> voice animals cat
meow

> voice hello world
hello world

> voice hi there
hi there!

> voice greetings
greetings and salutations!
```

As you can see, defining the submodule structure also shapes the command line API. In Nushell, namespaces directly folds into subcommands. This is true for all definitions: aliases, commands, known externals, modules.

Environment Variables

Modules can also define an environment using `export-env`:

```
# greetings.nu

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

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

When `use` is evaluated, it 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!
```

You can put a complex code defining your environment without polluting the namespace of the module, for example:

```
    def tmp [] { "tmp" }
    def other [] { "other" }

    let len = (tmp | str length)
```

```
    load-env {
        OTHER_ENV: (other)
        TMP_LEN: $len
    }
}
```

Only `$env.TMP_LEN` and `$env.OTHER_ENV` are preserved after evaluating the `export-env` module.

Caveats

Like any programming language, Nushell is also a product of a tradeoff and there are limitations to our module system.

`export-env` runs only when the `use` call is *evaluated*

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

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

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

and then use it

```
> use purpose.nu

> purpose greeting_purpose
```

However, this won't work, because the code inside the module is not *evaluated*, only *parsed* (only the `export-env` block is evaluated when you call `use purpose.nu`). To export the environment of `greetings.nu`, you need to add it to the `export-env` module:

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

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

then

```
> use purpose.nu

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

Calling `use purpose.nu` ran the `export-env` block inside `purpose.nu` which in turn ran `use greetings.nu` which in turn ran the `export-env` block inside `greetings.nu`, preserving the environment changes.

Module file / command cannot be named after parent module

- Module directory cannot contain .nu file named after the directory (`spam/spam.nu`)
  - Consider a `spam` directory containing both `spam.nu` and `mod.nu`, calling `use spam *` would create an ambiguous situation where the `spam` module would be defined twice.
- Module cannot contain file named after the module
  - Same case as above: Module `spam` containing both `main` and `spam` commands would create an ambiguous situation when exported as `use spam *`.

Examples

This section describes some useful patterns using modules.

Local Definitions

Anything defined in a module without the `export` keyword will work only in the module's scope.

```
# greetings.nu

use tools/utils.nu generate-prefix  # visible only locally (we assume the file
exists)

export def hello [name: string] {
    greetings-helper "hello" "world"
}

export def hi [where: string] {
    greetings-helper "hi" "there"
}

def greetings-helper [greeting: string, subject: string] {
    $"(generate-prefix)($greeting) ($subject)!"
}
```

then

```
> use greetings.nu *


> hello "world"
hello world!

> hi "there"
hi there!

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

> generate-prefix  # fails because the command is imported only locally inside the
module
```

Dumping files into directory

A common pattern in traditional shells is dumping and auto-sourcing files from a directory (for example, loading custom completions). In Nushell, doing this directly is currently not possible, but directory modules can still be used.

Here we'll create a simple completion module with a submodule dedicated to some Git completions:

1. Create the completion directory `mkdir ($nu.default-config-dir | path join completions)`
2. Create an empty `mod.nu` for it `touch ($nu.default-config-dir | path join completions mod.nu)`

3. Put the following snippet in `git.nu` under the `completions` directory

```
export extern main [
    --version(-v)
    -C: string
    # ... etc.
]

export extern add [
    --verbose(-v)
    --dry-run(-n)
    # ... etc.
]

export extern checkout [
    branch: string@complete-git-branch
]

def complete-git-branch [] {
    # ... code to list git branches
}
```

4. Add `export module git.nu` to `mod.nu`
5. Add the parent of the `completions` directory to your NU_LIB_DIRS inside `env.nu`

```
$env.NU_LIB_DIRS = [
    ...
    $nu.default-config-dir
]
```

6. import the completions to Nushell in your `config.nu` `use completions *` Now you've set up a directory where you can put your completion files and you should have some Git completions the next time you start Nushell

> **Note** This will use the file name (in our example `git` from `git.nu`) as the module name. This means some completions might not work if the definition has the base command in it's name. For example, if you defined our known externals in our `git.nu` as `export extern 'git push' []`, etc. and followed the rest of the steps, you would get subcommands like `git git push`, etc. You would need to call `use completions git *` to get the desired subcommands. For this reason, using `main` as outlined in the step above is the preferred way to define subcommands.

Setting environment + aliases (conda style)

`def --env` commands, `export-env` block and aliases can be used to dynamically manipulate "virtual environments" (a concept well known from Python).

We use it in our official virtualenv integration https://github.com/pypa/virtualenv/blob/main/src/virtualenv/activation/nushell/activate.nu

Another example could be our unofficial Conda module: https://github.com/nushell/nu_scripts/blob/f86a060c10f132407694e9ba0f536bfe3ee51efc/modules/virtual_environments/conda.nu

> **Warning** Work In Progress

Hiding

Any custom command or alias, imported from a module or not, can be "hidden", restoring the previous definition. We do this with the `hide` 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

> **Note** `hide` is not a supported keyword at the root of a module (unlike `def` etc.)

Overlays

Overlays act as "layers" of definitions (custom commands, aliases, environment variables) that can be activated and deactivated on demand. They resemble virtual environments found in some languages, such as Python.

Note: To understand overlays, make sure to check Modules first as overlays build on top of modules.

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.

The module can be created by any of the three methods described in Modules:

- "inline" modules (used in this example)
- file
- directory

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 evaluated the `export-env` block the same way as `use` command would (see Modules chapter).

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`:

```
(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` without an argument which will remove the last active overlay.

Overlays Are Recordable

Any new definition (command, alias, environment variable) is recorded into the last active overlay:

```
(zero)> overlay use spam

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

Now, the `eggs` command belongs to the `spam` overlay. If we remove the overlay, we can't call it anymore:

```
(spam)> overlay hide spam

(zero)> eggs
Error: Can't run executable...
```

But we can bring it back!

```
(zero)> overlay use spam

(spam)> eggs
eggs
```

Overlays remember what you add to them and store that information even if you remove them. This can let you repeatedly swap between different contexts.

Sometimes, after adding an overlay, you might not want custom definitions to be added into it. The solution can be to create a new empty overlay that would be used just for recording the custom changes:

```
(zero)> overlay use spam

(spam)> module scratchpad { }

(spam)> overlay use scratchpad

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

The `eggs` command is added into `scratchpad` while keeping `spam` intact.

To make it less verbose, you can use the `overlay new` 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 { $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 can be given explicit signatures; take `str stats` as an example, the signature is like this:

```
def "str stats" []: string -> record { }
```

The type names between the `:` and the opening curly brace `{` describe the command's input/output pipeline types. The input type for a given pipeline, in this case `string`, is given before the `->`; and the output type `record` is given after `->`. There can be multiple input/output types. If there are multiple input/output types, they can be placed within brackets and separated with commas, as in `str join`:

```
def "str join" [separator?: string]: [list -> string, string -> string] { }
```

It says that the `str join` command takes an optional `string` type argument, and an input pipeline of either a `list` (implying `list<any>`) with output type of `string`, or a single `string`, again with output type of `string`.

Some commands don't accept or require data through the input pipeline, thus the input type will be `<nothing>`. The same is true for the output type if the command returns `null` (e.g. `rm` or `hide`):

```
def hide [module: string, members?]: nothing -> nothing { }
```

Testing your Nushell code

To ensure that your code works as expected, you can use the testing module.

Quick start

Download the testing module and save it as `testing.nu` in the folder with your project.

Have a file, called `test_math.nu`:

```
use std assert
```

```
#[test]
def test_addition [] {
    assert equal (1 + 2) 3
}

#[test]
#[ignore]
def test_skip [] {
    # this won't be run
}

#[test]
def test_failing [] {
    assert false "This is just for testing"
}
```

Run the tests:

```
❯ use testing.nu run-tests
❯ run-tests
INF|2023-04-12T10:42:29.099|Running tests in test_math
Error:
  × This is just for testing
    ╭─[C:\wip\test_math.nu:13:1]
 13 │ def test_failing [] {
 14 │     assert false "This is just for testing"
    ·             ──────┬──────
    ·                   ╰── It is not true.
 15 │ }
    ╰────


WRN|2023-04-12T10:42:31.086|Test case test_skip is skipped
Error:
  × some tests did not pass (see complete errors above):
  │
  │       test_math test_addition
  │     × test_math test_failing
  │     s test_math test_skip
  │
```

Assert commands

The foundation for every assertion is the `std assert` command. If the condition is not true, it makes an error. For example:

```
❯ use std assert
❯ std assert (1 == 2)
Error:
  × Assertion failed.
   ╭─[entry #13:1:1]
 1 │ std assert (1 == 2)
   ·             ───────┬──
   ·                    ╰── It is not true.
   ╰────
```

Optionally, a message can be set to show the intention of the assert command, what went wrong or what was expected:

```
❯ std assert ($a == 19) $"The lockout code is wrong, received: ($a)"
Error:
  × The lockout code is wrong, received: 13
   ╭─[entry #25:1:1]
 1 │ std assert ($a == 19) $"The lockout code is wrong, received: ($a)"
   ·             ────┬────
   ·                 ╰── It is not true.
   ╰────
```

There are many assert commands, which behave exactly as the base one with the proper operator. The additional value for them is the ability for better error messages.

For example this is not so helpful without additional message:

```
❯ std assert ($b | str contains $a)
Error:
  × Assertion failed.
   ╭─[entry #35:1:1]
 1 │ assert ($b | str contains $a)
   ·              ──────┬──────
   ·                    ╰── It is not true.
   ╰────
```

While with using `assert str contains`:

```
❯ std assert str contains $b $a
Error:
  × Assertion failed.
   ╭─[entry #34:1:1]
 1 │ assert str contains $b $a
   ·                     ──┬──
   ·                       ╰── 'haystack' does not contain 'a needle'.
   ╰────
```

In general for base `assert` command it is encouraged to always provide the additional message to show what went wrong. If you cannot use any built-in assert command, you can create a custom one with passing the label for `error make` for the `assert` command:

```
def "assert even" [number: int] {
    std assert ($number mod 2 == 0) --error-label {
        start: (metadata $number).span.start,
        end: (metadata $number).span.end,
        text: $"($number) is not an even number",
    }
}
```

Then you'll have your detailed custom error message:

```
❯ let $a = 13
❯ assert even $a
Error:
  × Assertion failed.
   ╭─[entry #37:1:1]
 1 │ assert even $a
   ·             ─┬
```

```
.               └── 13 is not an even number
└────
```

Test modules & test cases

The naming convention for test modules is `test_<your_module>.nu` and `test_<test name>` for test cases.

In order for a function to be recognized as a test by the test runner it needs to be annotated with `#[test]`.

The following annotations are supported by the test runner:

- test - test case to be executed during test run
- test-skip - test case to be skipped during test run
- before-all - function to run at the beginning of test run. Returns a global context record that is piped into every test function
- before-each - function to run before every test case. Returns a per-test context record that is merged with global context and piped into test functions
- after-each - function to run after every test case. Receives the context record just like the test cases
- after-all - function to run after all test cases have been executed. Receives the global context record

The standard library itself is tested with this framework, so you can find many examples in the Nushell repository.

Setting verbosity

The `testing.nu` module uses the `log` commands from the standard library to display information, so you can set `NU_LOG_LEVEL` if you want more or less details:

```
❯ use testing.nu run-tests
❯ NU_LOG_LEVEL=DEBUG run-tests
❯ NU_LOG_LEVEL=WARNING run-tests
```

Best practices

This page is a working document collecting syntax guidelines and best practices we have discovered so far. The goal of this document is to eventually land on a canonical Nushell code style, but as for now it is still work in progress and subject to change. We welcome discussion and contributions.

Keep in mind that these guidelines are not required to be used in external repositories (not ours), you can change them in the way you want, but please be consistent and follow your rules.

All escape sequences should not be interpreted literally, unless directed to do so. In other words, treat something like `\n` like the new line character and not a literal slash followed by `n`.

Formatting

Defaults

It's recommended to assume that by default no spaces or tabs allowed, but the following rules define where they are allowed.

Basic

- **It's recommended to** put one space before and after pipe `|` symbol, commands, subcommands, their options and arguments.
- **It's recommended to** never put several consecutive spaces unless they are part of string.
- **It's recommended to** omit commas between list items.

Correct:

```
'Hello, Nushell! This is a gradient.' | ansi gradient --fgstart '0x40c9ff' --fgend
'0xe81cff'
```

Incorrect:

```
# - too many spaces after "|": 2 instead of 1
'Hello, Nushell! This is a gradient.' |  ansi gradient --fgstart '0x40c9ff' --fgend
'0xe81cff'
```

One-line format

One-line format is a format for writing all commands in one line.

It's recommended to default to this format:

1. unless you are writing scripts
2. in scripts for lists and records unless they either:
   1. more than 80 characters long
   2. contain nested lists or records
3. for pipelines less than 80 characters long not containing items should be formatted with a long format

Rules:

1. parameters:
   1. **It's recommended to** put one space after comma `,` after block or closure parameter.
   2. **It's recommended to** put one space after pipe `|` symbol denoting block or closure parameter list end.
2. block and closure bodies:
   1. **It's recommended to** put one space after opening block or closure curly brace `{` if no explicit parameters defined.
   2. **It's recommended to** put one space before closing block or closure curly brace `}`.
3. records:
   1. **It's recommended to** put one space after `:` after record key.
   2. **It's recommended to** put one space after comma `,` after key value.
4. lists:
   1. **It's recommended to** put one space after comma `,` after list value.
5. surrounding constructs:
   1. **It's recommended to** put one space before opening square `[`, curly brace `{`, or parenthesis `(` if preceding symbol is not the same.
   2. **It's recommended to** put one space after closing square `]`, curly brace `}`, or parenthesis `)` if following symbol is not the same.

Correct:

```
[[status]; [UP] [UP]] | all {|el| $el.status == UP }
[1 2 3 4] | reduce {|it, acc| $it + $acc }
[1 2 3 4] | reduce {|it acc| $it + $acc }
{x: 1, y: 2}
{x: 1 y: 2}
[1 2] | zip [3 4]
[]
(1 + 2) * 3
```

Incorrect:

```
# too many spaces before "|el|": no space is allowed
[[status]; [UP] [UP]] | all { |el| $el.status == UP }

# too many spaces before ",": no space is allowed
[1 2 3 4] | reduce {|it , acc| $it + $acc }

# too many spaces before "x": no space is allowed
{ x: 1, y: 2}

# too many spaces before "[3": one space is required
[1 2] | zip  [3 4]

# too many spaces before "]": no space is allowed
[ ]

# too many spaces before ")": no space is allowed
(1 + 2 ) * 3
```

Multi-line format

Multi-line format is a format for writing all commands in several lines. It inherits all rules from one-line format and modifies them slightly.

It's recommended to default to this format:

1. while you are writing scripts
2. in scripts for lists and records while they either:
   1. more than 80 characters long
   2. contain nested lists or records
3. for pipelines more 80 characters long

Rules:

1. general:
   1. **It's required to omit** trailing spaces.
2. block and closure bodies:
   1. **It's recommended to** put each body pipeline on a separate line.
3. records:
   1. **It's recommended to** put each record key-value pair on separate line.
4. lists:
   1. **It's recommended to** put each list item on separate line.
5. surrounding constructs:
   1. **It's recommended to** put one `\n` before opening square `[`, curly brace `{`, or parenthesis `(` if preceding symbol is not the and applying this rule produce line with a singular parenthesis.
   2. **It's recommended to** put one `\n` after closing square `]`, curly brace `}`, or parenthesis `)` if following symbol is not the same and applying this rule produce line with a singular parenthesis.

Correct:

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

[1 2 3 4] | reduce {|it, acc|
    $it + $acc
}
```

```
{x: 1, y: 2}

[
  {name: "Teresa", age: 24},
  {name: "Thomas", age: 26}
]

let selectedProfile = (for it in ($credentials | transpose name credentials) {
    echo $it.name
})
```

Incorrect:

```
# too many spaces before "|el|": no space is allowed (like in one-line format)
[[status]; [UP] [UP]] | all { |el|
    # too few "\n" before "}": one "\n" is required
    $el.status == UP}

# too many spaces before "2": one space is required (like in one-line format)
[1  2 3 4] | reduce {|it, acc|
    $it + $acc
}

{
   # too many "\n" before "x": one-line format required as no nested lists or record
exist
   x: 1,
   y: 2
}

# too few "\n" before "{": multi-line format required as there are two nested records
[{name: "Teresa", age: 24},
  {name: "Thomas", age: 26}]

let selectedProfile = (
    # too many "\n" before "foo": no "\n" is allowed
    for it in ($credentials | transpose name credentials) {
        echo $it.name
})
```

Options and parameters of custom commands

- **It's recommended to** keep count of all positional parameters less than or equal to 2, for remaining inputs use options. Assume that command can expect source and destination parameter, like `mv`: source and target file or directory.
- **It's recommended to** use positional parameters unless they can't be used due to rules listed here or technical restrictions. For instance, when there are several kinds of optional parameters (but at least one parameter should be provided) use options. Great example of this is `ansi gradient` command where at least foreground or background must be passed.
- **It's recommended to** provide both long and short options.

Documentation

- **It's recommended to** provide documentation for all exported entities (like custom commands) and their inputs (like custom command parameters and options).

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). 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 completions work also for Nushell custom commands).

Coloring and Theming in Nu goes into more detail about how to configure Nushell's appearance.

If you want to schedule some commands to run in the background, Background task in Nu provide a simple guideline for you to follow.

And finally, hooks allow you to insert fragments of Nushell code to run at certain events.

Configuration

Nushell Configuration with `env.nu` and `config.nu`

Nushell uses a configuration system that loads and runs two Nushell script files at launch time:

- `env.nu` is used to define environment variables or write files before `config.nu` starts. For example `$env.NU_LIB_DIRS` controls where Nu finds imports. Third party scripts, like prompts or mise, must already be saved to disk before `config.nu` can read them.
- `config.nu` is used to add definitions, aliases, and more to the global namespace. It can also use the environment variables and constants defined in `env.nu`. If you can't decide which file to add stuff, prefer `config.nu`.

Check where Nushell is reading these config files from by calling `$nu.env-path` and `$nu.config-path`.

```
> $nu.env-path
/Users/FirstNameLastName/Library/Application Support/nushell/env.nu
```

(You can think of the Nushell config loading sequence as executing two REPL 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` and `default config.nu`.

The default config files aren't required. If you prefer to start with an empty `env.nu` and `config.nu` then Nu applies identical defaults in internal Rust code. You can still browse the default files for default values of environment variables and a list of all configurable settings using the `config` commands:

```
> config env --default | nu-highlight | lines
> config nu --default | nu-highlight | lines
```

Control which directory Nushell reads config files from with the `XDG_CONFIG_HOME` environment variable. When you set it to an absolute path, Nushell will read config files from `$"($env.XDG_CONFIG_HOME)/nushell"`.

`XDG_CONFIG_HOME` must be set **before** starting Nushell. Do not set it in `env.nu`.

Here's an example for reading config files from `~/.config/nushell` rather than the default directory for Windows, which is `C:\Users\username\AppData\Roaming\nushell`.

```
> $env.XDG_CONFIG_HOME = "C:\Users\username\.config"
> nu
> $nu.default-config-dir
C:\Users\username\.config\nushell
```

`XDG_CONFIG_HOME` is not a Nushell-specific environment variable and should not be set to the directory that contains Nushell config files. It should be the directory *above* the `nushell` directory. If you set it to `/Users/username/dotfiles/nushell`, Nushell will look for config files in `/Users/username/dotfiles/nushell/nushell` instead. In this case, you would want to set it to `/Users/username/dotfiles`.

Configuring `$env.config`

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

```
$env.config = {
  ...
}
```

Note that setting any key overwrites its previous value. Likewise it's an error to reference any missing key. If `$env.config` already exists you can update or gracefully insert a `cell-path` at any depth using `upsert`:

```
$env.config = ($env.config | upsert <field name> <field value>)
```

By convention, this variable is defined in the `config.nu` file.

Environment

You can set environment variables for the duration of a Nushell session using the `$env.<var> = <val>` structure inside the `env.nu` file. For example:

```
$env.FOO = 'BAR'
```

(Although $env.config is an environment variable, it is still defined by convention inside config.nu.)

These are some important variables to look at for Nushell-specific settings:

- `LS_COLORS`: Sets up colors per file type in ls
- `PROMPT_COMMAND`: Code to execute for setting up the prompt (block or string)

- `PROMPT_COMMAND_RIGHT`: Code to execute for setting up the right prompt (block)
- `PROMPT_INDICATOR = "〉"`: The indicator printed after the prompt (by default ">"-like Unicode symbol)
- `PROMPT_INDICATOR_VI_INSERT = ": "`
- `PROMPT_INDICATOR_VI_NORMAL = "〉 "`
- `PROMPT_MULTILINE_INDICATOR = "::: "`

Configurations with built-in commands

The (`config nu` and `config env`) commands open their respective configurations for quick editing in your preferred text editor or IDE. Nu determines your editor from the following environment variables in order:

1. `$env.config.buffer_editor`
2. `$env.EDITOR`
3. `$env.VISUAL`

Color Config section

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

Remove Welcome Message

To remove the welcome message, you need to edit your `config.nu` by typing `config nu` in your terminal, then you go to the global configuration `$env.config` and set `show_banner` option to false, like this:

@code

Configuring Nu as a login shell

To use Nu as a login shell, you'll need to configure the `$env` variable. This sets up the environment for external programs.

To get an idea of which environment variables are set up by your current login shell, start a new shell session, then run nu in that shell.

You can then configure some `$env.<var> = <val>` that setup the same environment variables in your nu login shell. Use this command to generate some `$env.<var> = <val>` for all the environment variables:

```
$env | reject config | transpose key val | each {|r| echo $"$env.($r.key) =
'($r.val)'"} | str join (char nl)
```

This will print out `$env.<var> = <val>` lines, one for each environment variable along with its setting. You may not need all of them, for instance the `PS1` variable is bash specific.

Next, on some distros you'll also need to ensure Nu is in the /etc/shells list:

```
> cat /etc/shells
# /etc/shells: valid login shells
/bin/sh
/bin/dash
/bin/bash
/bin/rbash
/usr/bin/screen
/usr/bin/fish
/home/sophia/.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.

What about customizing interactive shells, similar to `.zshrc`? By default `config.nu` is only loaded in interactive shells, not scripts.

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:

```
alias nu-open = open
alias open = ^open
```

The `^` symbol *escapes* the Nushell `open` command, which invokes the operating system's `open` command. For more about escape and `^` see the chapter about escapes.

PATH configuration

In Nushell, the PATH environment variable (Path on Windows) is a list of paths. To append a new path to it, you can use `$env.<var> = <val>` and `append` in `env.nu`:

```
$env.PATH = ($env.PATH | split row (char esep) | append '/some/path')
```

This will append `/some/path` to the end of PATH; you can also use `prepend` 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.

To add multiple paths only if not already listed, one can add to `env.nu`:

```
$env.PATH = $env.PATH | split row (char esep)
  | append /usr/local/bin
  | append ($env.CARGO_HOME | path join bin)
  | append ($env.HOME | path join .local bin)
  | uniq # filter so the paths are unique
```

This will add `/usr/local/bin`, the `bin` directory of CARGO_HOME, the `.local/bin` of HOME to PATH. It will also remove duplicates from PATH.

There's a convenience command for updating your system path but you must first open the `std` module (in preview):

```
use std *
path add /usr/local/bin ($env.CARGO_HOME | path join bin) # etc.
```

You can optionally `--append` paths to be checked last like the ones below.

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

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

# Linux
$env.PATH = ($env.PATH | split row (char esep) | prepend '/home/linuxbrew/.linuxbrew/
bin')
```

Pyenv
Pyenv is a popular Python version manager. To add it to your Nushell PATH:

MacOS or Linux

```
# MacOS or Linux
$env.PATH = ($env.PATH | split row (char esep) | prepend $"(pyenv root)/shims")
```

Windows
Windows users need to install pyenv-win and execute the `Get-Command pyenv` command in PowerShell to get the path of `pyenv.ps1` after the installation.

The result usually looks like: `C:\Users\<your-username>\.pyenv\pyenv-win\bin\pyenv.ps1`

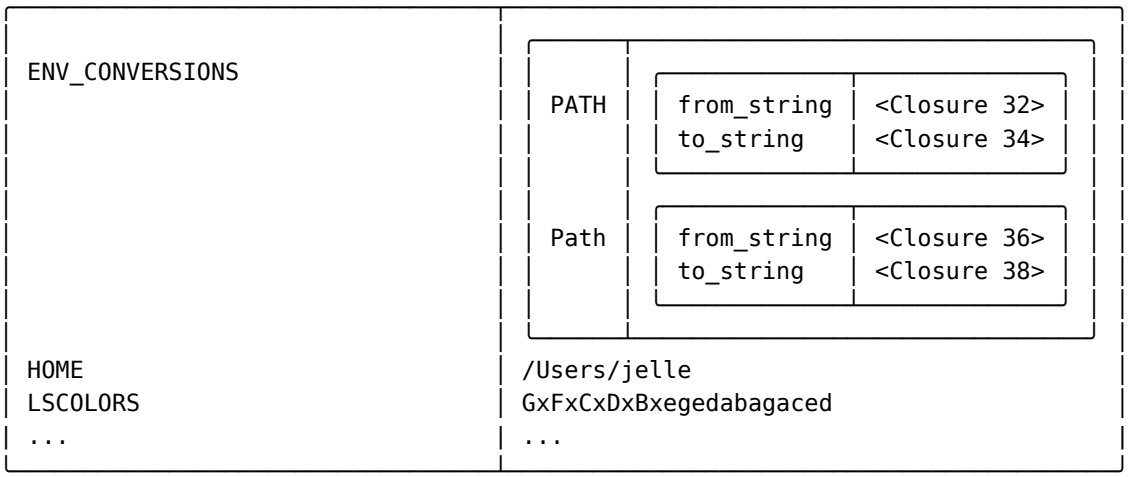
Then add the path of pyenv to your Nushell PATH:

```
# Windows
$env.Path = ($env.Path | split row (char esep) | prepend $"~/.pyenv/pyenv-win/bin/
pyenv.ps1")
```

Environment
A common task in a shell is to control the environment that external applications will use. This is often done automatically, as the environment is packaged up and given to the external application as it launches. Sometimes, though, we want to have more precise control over what environment variables an application sees.

You can see the current environment variables in the $env variable:

```
~> $env | table -e
```



In Nushell, environment variables can be any value and have any type. You can see the type of an env variable with the describe command, for example: `$env.PROMPT_COMMAND | describe`.

To send environment variables to external applications, the values will need to be converted to strings. See Environment variable conversions on how this works.

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

Setting environment variables

There are several ways to set an environment variable:

$env.VAR assignment

Using the `$env.VAR = "val"` is the most straightforward method

```
> $env.FOO = 'BAR'
```

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

```
$env.Path = ($env.Path | prepend 'C:\path\you\want\to\add')
```

Here we've prepended our folder to the existing folders in the Path, so it will have the highest priority. If you want to give it the lowest priority instead, you can use the `append` 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.

Calling a command defined with `def --env`

See Defining environment from custom commands for details.

Using module's exports

See Modules for details.

Reading environment variables

Individual environment variables are fields of a record that is stored in the `$env` variable and can be read with `$env.VARIABLE`:

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

Sometimes, you may want to access an environmental variable which might be unset. Consider using the question mark operator to avoid an error:

```
> $env.FOO | describe
Error: nu::shell::column_not_found

  × Cannot find column
   ╭─[entry #1:1:1]
 1 │ $env.FOO
   ·  ──┬── ─┬─
   ·    │    └── cannot find column 'FOO'
```

```
   .   └── value originates here
   └────
```

```
> $env.FOO? | describe
nothing

> $env.FOO? | default "BAR"
BAR
```

Alternatively, you can check for the presence of an environmental variable with `in`:

```
> $env.FOO
BAR

> if "FOO" in $env {
>     echo $env.FOO
> }
BAR
```

Scoping

When you set an environment variable, it will be available only in the current scope (the block you're in and any block inside of it).

Here is a small example to demonstrate the environment scoping:

```
> $env.FOO = "BAR"
> do {
    $env.FOO = "BAZ"
    $env.FOO == "BAZ"
}
true
> $env.FOO == "BAR"
true
```

Changing directory

Common task in a shell is to change directory with the `cd` 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).

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
$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) will preserve the environment on the caller's side:

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

> foo

> $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 includes conversion of PATH (and Path used on Windows) environment variables from a string to a list. After both `env.nu` and `config.nu` are loaded, any existing environment variable specified inside `ENV_CONVERSIONS` will be translated according to its `from_string` field into a value of any type. External tools require environment variables to be strings, therefore, any non-string environment variable needs to be converted first. The conversion of value -> string is set by the `to_string` field of `ENV_CONVERSIONS` and is done every time an external command is run.

Let's illustrate the conversions with an example. Put the following in your config.nu:

```
$env.ENV_CONVERSIONS = {
    # ... you might have Path and PATH already there, add:
    FOO : {
        from_string: { |s| $s | split row '-' }
        to_string: { |v| $v | str join '-' }
    }
}
```

Now, within a Nushell instance:

```
> with-env { FOO : 'a-b-c' } { nu }  # runs Nushell with FOO env. var. set to 'a-b-c'

> $env.FOO
  0   a
  1   b
  2   c
```

You can see the `$env.FOO` is now a list in a new Nushell instance with the updated config. You can also test the conversion manually by

```
> do $env.ENV_CONVERSIONS.FOO.from_string 'a-b-c'
```

Now, to test the conversion list -> string, run:

```
> nu -c '$env.FOO'
a-b-c
```

Because `nu` is an external program, Nushell translated the `[ a b c ]` list according to `ENV_CONVERSIONS.FOO.to_string` and passed it to the `nu` process. Running commands with `nu -c`

does not load the config file, therefore the env conversion for `FOO` is missing and it is displayed as a plain string – this way we can verify the translation was successful. You can also run this step manually by `do $env.ENV_CONVERSIONS.FOO.to_string [a b c]`

*(Important! The environment conversion string -> value happens **after** the env.nu and config.nu are evaluated. All environment variables in env.nu and config.nu are still strings unless you set them manually to some other values.)*

Removing environment variables

You can remove an environment variable only if it was set in the current scope via `hide-env`:

```
> $env.FOO = 'BAR'
...
> hide-env FOO
```

The hiding is also scoped which both allows you to remove an environment variable temporarily and prevents you from modifying a parent environment from within a child scope:

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

Stdout, Stderr, and Exit Codes

An important piece of interop between Nushell and external commands is working with the standard streams of data coming from the external.

The first of these important streams is stdout.

Stdout

Stdout is the way that most external apps will send data into the pipeline or to the screen. Data sent by an external app to its stdout is received by Nushell by default if it's part of a pipeline:

```
> external | str join
```

The above would call the external named `external` and would redirect the stdout output stream into the pipeline. With this redirection, Nushell can then pass the data to the next command in the pipeline, here `str join`.

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

Stderr

Another common stream that external applications often use to print error messages is stderr. By default, Nushell does not do any redirection of stderr, which means that by default it will print to the screen.

You can force Nushell to do a redirection by using `do { ... } | complete`. For example, if we wanted to call the external above and redirect its stderr, we would write:

```
> do { external } | complete
```

Exit code

Finally, external commands have an "exit code". These codes help give a hint to the caller whether the command ran successfully.

Nushell tracks the last exit code of the recently completed external in one of two ways. The first way is with the `LAST_EXIT_CODE` environment variable.

```
> do { external }
> $env.LAST_EXIT_CODE
```

The second uses a command called `complete`.

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 { cat unknown.txt } | complete
```

```
╭───────────┬──────────────────────────────────────────────────╮
│ stdout    │                                                  │
│ stderr    │ cat: unknown.txt: No such file or directory      │
│ exit_code │ 1                                                │
╰───────────┴──────────────────────────────────────────────────╯
```

`echo`, `print`, and `log` commands

The `echo` command is mainly for *pipes*. It returns its arguments, ignoring the piped-in value. There is usually little reason to use this over just writing the values as-is.

In contrast, the `print` command prints the given values to stdout as plain text. It can be used to write to standard error output, as well. 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.

The standard library has commands to write out messages in different logging levels. For example:

@code

The log level for output can be set with the `NU_LOG_LEVEL` environment variable:

```
NU_LOG_LEVEL=DEBUG nu std_log.nu
```

Using `out>`, `err>` to redirect stdout and stderr to files

If you want to redirect output to file, you can just type something like this:

```
cat unknown.txt out> out.log err> err.log
```

If you want to redirect both stdout and stderr to the same file, just type something like this:

```
cat unknown.txt out+err> log.log
```

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
貝
```

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 use Cmd.exe 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

repo

oh-my-posh

site

repo

If you like oh-my-posh, 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.
2. Download and install a nerd font.
3. Generate the .oh-my-posh.nu file. By default it will be generated to your home directory. You can use `--config` to specify a theme, other wise, oh-my-posh comes with a default theme.
4. Initialize oh-my-posh prompt by adding in ~/.config/nushell/config.nu(or the path output by `$nu.config-path`) to source ~/.oh-my-posh.nu.

```
# Generate the .oh-my-posh.nu file
> oh-my-posh init nu --config ~/.poshthemes/M365Princess.omp.json
```

```
# Initialize oh-my-posh.nu at shell startup by adding this line in your config.nu
file
> source ~/.oh-my-posh.nu
```

For MacOS users:

1. You can install oh-my-posh by `brew`, just following the guide here
2. Download and install a nerd font.
3. Set the PROMPT_COMMAND in the file output by `$nu.config-path`, here is a code snippet:

```
let posh_dir = (brew --prefix oh-my-posh | str trim)
let posh_theme = $'($posh_dir)/share/oh-my-posh/themes/'
# Change the theme names to: zash/space/robbyrussel/powerline/powerlevel10k_lean/
# material/half-life/lambda Or double lines theme: amro/pure/spaceship, etc.
# For more [Themes demo](https://ohmyposh.dev/docs/themes)
$env.PROMPT_COMMAND = { || oh-my-posh prompt print primary --config $'($posh_theme)/
zash.omp.json' }
# Optional
$env.PROMPT_INDICATOR = $"(ansi y)$> (ansi reset)"
```

Starship

site

repo

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.

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

The link above is the official integration of Starship and Nushell and is the simplest way to get Starship running without doing anything manual:

- Starship will create its own configuration / environment setup script
- you simply have to create it in `env.nu` and `use` it in `config.nu`

  :::

Here's an example config section for Starship:

```
$env.STARSHIP_SHELL = "nu"

def create_left_prompt [] {
    starship prompt --cmd-duration $env.CMD_DURATION_MS $'--
status=($env.LAST_EXIT_CODE)'
}

# Use nushell functions to define your right and left prompt
$env.PROMPT_COMMAND = { || create_left_prompt }
$env.PROMPT_COMMAND_RIGHT = ""

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

Now restart Nu.

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

Purs

repo

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/sophia/Source/nushell(main)> enter ../book
/home/sophia/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/sophia/Source/book(main)> enter ../music
/home/sophia/Source/music(main)> shells
```

#	active	path
0	false	/home/sophia/Source/nushell
1	false	/home/sophia/Source/book
2	true	/home/sophia/Source/music

The `shells` command shows us there are three shells: our original "nushell" source directory, "book" directory and "music" directory which is currently active.

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

```
/home/sophia/Source/music(main)> p
/home/sophia/Source/book(main)> n
/home/sophia/Source/music(main)> g 0
/home/sophia/Source/nushell(main)>
```

We can see the directory changing, but we're always able to get back to a previous directory we were working on. This allows us to work in multiple directories in the same session.

Exiting the shell

You can leave a shell you have `entered` using the `dexit` command.

You can always quit Nu, even if multiple shells are active, using `exit`.

Reedline, Nu's line editor

Nushell's line editor Reedline is a cross-platform line reader designed to be modular and flexible. The engine is in charge of controlling the command history, validations, completions, hints and screen paint.

Configuration

Editing mode

Reedline allows you to edit text using two modes: vi and emacs. If not specified, the default edit mode is emacs mode. In order to select your favorite you need to modify your config file and write down your preferred mode.

For example:

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

Default keybindings

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

Emacs and Vi Insert keybindings

Key	Event
Esc	Esc
Backspace	Backspace
End	Move to end of line
End	Complete history hint
Home	Move to line start
Ctr + c	Cancel current line
Ctr + l	Clear screen
Ctr + r	Search history
Ctr + Right	Complete history word
Ctr + Right	Move word right
Ctr + Left	Move word left
Up	Move menu up
Up	Move up
Down	Move menu down
Down	Move down
Left	Move menu left
Left	Move left
Right	History hint complete
Right	Move menu right
Right	Move right
Ctr + b	Move menu left
Ctr + b	Move left
Ctr + f	History hint complete
Ctr + f	Move menu right
Ctr + f	Move right
Ctr + p	Move menu up
Ctr + p	Move up
Ctr + n	Move menu down
Ctr + n	Move down

Vi Normal keybindings

Key	Event
Ctr + c	Cancel current line
Ctr + l	Clear screen
Up	Move menu up
Up	Move up
Down	Move menu down
Down	Move down
Left	Move menu left
Left	Move left
Right	Move menu right
Right	Move right

Besides the previous keybindings, while in Vi normal mode you can use the classic vi mode of executing actions by selecting a motion or an action. The available options for the combinations are:

Vi Normal motions

Key	motion
w	Word
d	Line end
0	Line start
$	Line end
f	Right until char
t	Right before char
F	Left until char
T	Left before char

Vi Normal actions

Key	action
d	Delete
p	Paste after
P	Paste before
h	Move left
l	Move right
j	Move down
k	Move up
w	Move word right
b	Move word left
i	Enter Vi insert at current char
a	Enter Vi insert after char
0	Move to start of line
^	Move to start of line
$	Move to end of line
u	Undo
c	Change
x	Delete char
s	History search
D	Delete to end
A	Append to end

Command history

As mentioned before, Reedline manages and stores all the commands that are edited and sent to Nushell. To configure the max number of records that Reedline should store you will need to adjust this value in your config file:

```
$env.config = {
  ...
  history: {
    ...
    max_size: 1000
    ...
  }
  ...
}
```

Customizing your prompt

Reedline prompt is also highly customizable. In order to construct your perfect prompt, you could define the next environment variables in your config file:

```
# Use nushell functions to define your right and left prompt
def create_left_prompt [] {
    let path_segment = ($env.PWD)

    $path_segment
}
```

```
def create_right_prompt [] {
    let time_segment = ([
        (date now | format date '%m/%d/%Y %r')
    ] | str join)

    $time_segment
}

$env.PROMPT_COMMAND = { create_left_prompt }
$env.PROMPT_COMMAND_RIGHT = { create_right_prompt }
```

You don't have to define the environment variables using Nushell functions. You can use simple strings to define them.

You can also customize the prompt indicator for the line editor by modifying the next env variables.

```
$env.PROMPT_INDICATOR = "〉"
$env.PROMPT_INDICATOR_VI_INSERT = ": "
$env.PROMPT_INDICATOR_VI_NORMAL = "〉"
$env.PROMPT_MULTILINE_INDICATOR = "::: "
```

The prompt indicators are environment variables that represent the state of the prompt

Keybindings

Reedline keybindings are powerful constructs that let you build chains of events that can be triggered with a specific combination of keys.

For example, let's say that you would like to map the completion menu to the `Ctrl + t` keybinding (default is `tab`). You can add the next entry to your config file.

```
  $env.config = {
    ...

    keybindings: [
      {
        name: completion_menu
        modifier: control
        keycode: char_t
        mode: emacs
        event: { send: menu name: completion_menu }
      }
    ]

    ...
  }
```

After loading this new `config.nu`, your new keybinding (`Ctrl + t`) will open the completion command.

Each keybinding requires the next elements:

- name: Unique name for your keybinding for easy reference in `$config.keybindings`
- modifier: A key modifier for the keybinding. The options are:
  - none
  - control
  - alt
  - shift
  - shift_alt

    - alt_shift
    - control_alt
    - alt_control
    - control_shift
    - shift_control
    - control_alt_shift
    - control_shift_alt
- keycode: This represent the key to be pressed
- mode: emacs, vi_insert, vi_normal (a single string or a list. e.g. [`vi_insert` `vi_normal`])
- event: The type of event that is going to be sent by the keybinding. The options are:
    - send
    - edit
    - until

All of the available modifiers, keycodes and events can be found with the command `keybindings list`

The keybindings added to `vi_insert` mode will be available when the line editor is in insert mode (when you can write text), and the keybindings marked with `vi_normal` mode will be available when in normal (when the cursor moves using h, j, k or l)

The event section of the keybinding entry is where the actions to be performed are defined. In this field you can use either a record or a list of records. Something like this

```
  ...
  event: { send: Enter }
  ...
```

or

```
  ...
  event: [
    { edit: Clear }
    { send: Enter }
  ]
  ...
```

The first keybinding example shown in this page follows the first case; a single event is sent to the engine.

The next keybinding is an example of a series of events sent to the engine. It first clears the prompt, inserts a string and then enters that value

```
  $env.config = {
    ...

    keybindings: [
    {
      name: change_dir_with_fzf
      modifier: CONTROL
      keycode: Char_t
      mode: emacs
      event:[
          { edit: Clear }
          { edit: InsertString,
            value: "cd (ls | where type == dir | each { |it| $it.name} | str join
(char nl) | fzf | decode utf-8 | str trim)"
```

```
          }
          { send: Enter }
        ]
    }

    ...
  }
```

One disadvantage of the previous keybinding is the fact that the inserted text will be processed by the validator and saved in the history, making the keybinding a bit slow and populating the command history with the same command. For that reason there is the `executehostcommand` type of event. The next example does the same as the previous one in a simpler way, sending a single event to the engine

```
  $env.config = {
    ...

    keybindings: [
    {
      name: change_dir_with_fzf
      modifier: CONTROL
      keycode: Char_y
      mode: emacs
      event: {
        send: executehostcommand,
        cmd: "cd (ls | where type == dir | each { |it| $it.name} | str join (char nl)
| fzf | decode utf-8 | str trim)"
      }
    }
  ]

    ...
  }
```

Before we continue you must have noticed that the syntax changes for edits and sends, and for that reason it is important to explain them a bit more. A `send` is all the `Reedline` events that can be processed by the engine and an `edit` are all the `EditCommands` that can be processed by the engine.

Send type

To find all the available options for `send` you can use

```
keybindings list | where type == events
```

And the syntax for `send` events is the next one

```
    ...
      event: { send: <NAME OF EVENT FROM LIST> }
    ...
```

You can write the name of the events with capital letters. The keybinding parser is case insensitive

There are two exceptions to this rule: the `Menu` and `ExecuteHostCommand`. Those two events require an extra field to be complete. The `Menu` needs the name of the menu to be activated (completion_menu or history_menu)

```
    ...
      event: {
```

```
        send: menu
        name: completion_menu
      }
    ...
```

and the `ExecuteHostCommand` requires a valid command that will be sent to the engine

```
    ...
      event: {
        send: executehostcommand
        cmd: "cd ~"
      }
    ...
```

It is worth mentioning that in the events list you will also see `Edit([])`, `Multiple([])` and `UntilFound([])`. These options are not available for the parser since they are constructed based on the keybinding definition. For example, a `Multiple([])` event is built for you when defining a list of records in the keybinding's event. An `Edit([])` event is the same as the `edit` type that was mentioned. And the `UntilFound([])` event is the same as the `until` type mentioned before.

Edit type

The `edit` type is the simplification of the `Edit([])` event. The `event` type simplifies defining complex editing events for the keybindings. To list the available options you can use the next command

```
keybindings list | where type == edits
```

The usual syntax for an `edit` is the next one

```
    ...
      event: { edit: <NAME OF EDIT FROM LIST> }
    ...
```

The syntax for the edits in the list that have a `()` changes a little bit. Since those edits require an extra value to be fully defined. For example, if we would like to insert a string where the prompt is located, then you will have to use

```
    ...
      event: {
        edit: insertstring
        value: "MY NEW STRING"
      }
    ...
```

or say you want to move right until the first `S`

```
    ...
      event: {
        edit: moverightuntil
        value: "S"
      }
    ...
```

As you can see, these two types will allow you to construct any type of keybinding that you require

Until type

To complete this keybinding tour we need to discuss the `until` type for event. As you have seen so far, you can send a single event or a list of events. And as we have seen, when a list of events is sent, each and every one of them is processed.

However, there may be cases when you want to assign different events to the same keybinding. This is especially useful with Nushell menus. For example, say you still want to activate your completion menu with `Ctrl + t` but you also want to move to the next element in the menu once it is activated using the same keybinding.

For these cases, we have the `until` keyword. The events listed inside the until event will be processed one by one with the difference that as soon as one is successful, the event processing is stopped.

The next keybinding represents this case.

```
$env.config = {
  ...

  keybindings: [
    {
      name: completion_menu
      modifier: control
      keycode: char_t
      mode: emacs
      event: {
        until: [
          { send: menu name: completion_menu }
          { send: menunext }
        ]
      }
    }
  ]

  ...
}
```

The previous keybinding will first try to open a completion menu. If the menu is not active, it will activate it and send a success signal. If the keybinding is pressed again, since there is an active menu, then the next event it will send is MenuNext, which means that it will move the selector to the next element in the menu.

As you can see the `until` keyword allows us to define two events for the same keybinding. At the moment of this writing, only the Menu events allow this type of layering. The other non menu event types will always return a success value, meaning that the `until` event will stop as soon as it reaches the command.

For example, the next keybinding will always send a `down` because that event is always successful

```
$env.config = {
  ...

  keybindings: [
    {
      name: completion_menu
      modifier: control
      keycode: char_t
      mode: emacs
      event: {
        until: [
          { send: down }
          { send: menu name: completion_menu }
```

```
          { send: menunext }
        ]
      }
    }
  ]

  ...
}
```

Removing a default keybinding

If you want to remove a certain default keybinding without replacing it with a different action, you can set `event: null`.

e.g. to disable screen clearing with `Ctrl + l` for all edit modes

```
$env.config = {
  ...

  keybindings: [
    {
      modifier: control
      keycode: char_l
      mode: [emacs, vi_normal, vi_insert]
      event: null
    }
  ]

  ...
}
```

Troubleshooting keybinding problems

Your terminal environment may not always propagate your key combinations on to nushell the way you expect it to. You can use the command `keybindings listen` to figure out if certain keypresses are actually received by nushell, and how.

Menus

Thanks to Reedline, Nushell has menus that can help you with your day to day shell scripting. Next we present the default menus that are always available when using Nushell

Help menu

The help menu is there to ease your transition into Nushell. Say you are putting together an amazing pipeline and then you forgot the internal command that would reverse a string for you. Instead of deleting your pipe, you can activate the help menu with `F1`. Once active just type keywords for the command you are looking for and the menu will show you commands that match your input. The matching is done on the name of the commands or the commands description.

To navigate the menu you can select the next element by using `tab`, you can scroll the description by pressing left or right and you can even paste into the line the available command examples.

The help menu can be configured by modifying the next parameters

```
$env.config = {
  ...

  menus = [
    ...
```

```
      {
        name: help_menu
        only_buffer_difference: true # Search is done on the text written after
activating the menu
        marker: "? "                 # Indicator that appears with the menu is active
        type: {
            layout: description      # Type of menu
            columns: 4               # Number of columns where the options are
displayed
            col_width: 20            # Optional value. If missing all the screen
width is used to calculate column width
            col_padding: 2           # Padding between columns
            selection_rows: 4        # Number of rows allowed to display found
options
            description_rows: 10     # Number of rows allowed to display command
description
        }
        style: {
            text: green                   # Text style
            selected_text: green_reverse  # Text style for selected option
            description_text: yellow      # Text style for description
        }
      }
      ...
    ]
    ...
```

Completion menu

The completion menu is a context sensitive menu that will present suggestions based on the status of the prompt. These suggestions can range from path suggestions to command alternatives. While writing a command, you can activate the menu to see available flags for an internal command. Also, if you have defined your custom completions for external commands, these will appear in the menu as well.

The completion menu by default is accessed by pressing `tab` and it can be configured by modifying these values from the config object:

```
  $env.config = {
    ...

    menus: [
      ...
      {
        name: completion_menu
        only_buffer_difference: false # Search is done on the text written after
activating the menu
        marker: "| "                  # Indicator that appears with the menu is
active
        type: {
            layout: columnar          # Type of menu
            columns: 4                # Number of columns where the options are
displayed
            col_width: 20             # Optional value. If missing all the screen
width is used to calculate column width
            col_padding: 2            # Padding between columns
        }
```

```
        style: {
            text: green                   # Text style
            selected_text: green_reverse  # Text style for selected option
            description_text: yellow      # Text style for description
        }
      }
      ...
    ]
    ...
```

By modifying these parameters you can customize the layout of your menu to your liking.

History menu

The history menu is a handy way to access the editor history. When activating the menu (default `Ctrl+r`) the command history is presented in reverse chronological order, making it extremely easy to select a previous command.

The history menu can be configured by modifying these values from the config object:

```
  $env.config = {
    ...

    menus = [
      ...
      {
        name: history_menu
        only_buffer_difference: true # Search is done on the text written after
activating the menu
        marker: "? "                 # Indicator that appears with the menu is active
        type: {
            layout: list             # Type of menu
            page_size: 10            # Number of entries that will presented when
activating the menu
        }
        style: {
            text: green                   # Text style
            selected_text: green_reverse  # Text style for selected option
            description_text: yellow      # Text style for description
        }
      }
      ...
    ]
    ...
```

When the history menu is activated, it pulls `page_size` records from the history and presents them in the menu. If there is space in the terminal, when you press `Ctrl+x` again the menu will pull the same number of records and append them to the current page. If it isn't possible to present all the pulled records, the menu will create a new page. The pages can be navigated by pressing `Ctrl+z` to go to previous page or `Ctrl+x` to go to next page.

Searching the history

To search in your history you can start typing key words for the command you are looking for. Once the menu is activated, anything that you type will be replaced by the selected command from your history. for example, say that you have already typed this

```
let a = ()
```

you can place the cursor inside the `()` and activate the menu. You can filter the history by typing key words and as soon as you select an entry, the typed words will be replaced

```
let a = (ls | where size > 10MiB)
```

Menu quick selection

Another nice feature of the menu is the ability to quick select something from it. Say you have activated your menu and it looks like this

```
>
0: ls | where size > 10MiB
1: ls | where size > 20MiB
2: ls | where size > 30MiB
3: ls | where size > 40MiB
```

Instead of pressing down to select the fourth entry, you can type `!3` and press enter. This will insert the selected text in the prompt position, saving you time scrolling down the menu.

History search and quick selection can be used together. You can activate the menu, do a quick search, and then quick select using the quick selection character.

User defined menus

In case you find that the default menus are not enough for you and you have the need to create your own menu, Nushell can help you with that.

In order to add a new menu that fulfills your needs, you can use one of the default layouts as a template. The templates available in nushell are columnar, list or description.

The columnar menu will show you data in a columnar fashion adjusting the column number based on the size of the text displayed in your columns.

The list type of menu will always display suggestions as a list, giving you the option to select values using `!` plus number combination.

The description type will give you more space to display a description for some values, together with extra information that could be inserted into the buffer.

Let's say we want to create a menu that displays all the variables created during your session, we are going to call it `vars_menu`. This menu will use a list layout (layout: list). To search for values, we want to use only the things that are written after the menu has been activated (only_buffer_difference: true).

With that in mind, the desired menu would look like this

```
$env.config = {
  ...

  menus = [
    ...
    {
      name: vars_menu
      only_buffer_difference: true
      marker: "# "
      type: {
          layout: list
          page_size: 10
      }
      style: {
```

```
            text: green
            selected_text: green_reverse
            description_text: yellow
        }
        source: { |buffer, position|
            $nu.scope.vars
            | where name =~ $buffer
            | sort-by name
            | each { |it| {value: $it.name description: $it.type} }
        }
      }
      ...
    ]
    ...
```

As you can see, the new menu is identical to the `history_menu` previously described. The only huge difference is the new field called `source`. 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

```
  $env.config = {
    ...
```

```
    keybindings: [
      {
        name: completion_menu
        modifier: control
        keycode: char_t
        mode: [vi_insert vi_normal]
        event: {
          until: [
            { send: menu name: completion_menu }
            { send: menupagenext }
          ]
        }
      }
      {
        name: history_menu
        modifier: control
        keycode: char_y
        mode: [vi_insert vi_normal]
        event: {
          until: [
            { send: menu name: history_menu }
            { send: menupagenext }
          ]
        }
      }
    ]

    ...
  }
```

Externs

Calling external commands is a fundamental part of using Nushell as a shell (and often using Nushell as a language). There's a problem, though: Nushell can't help with finding errors in the call, completions, or syntax highlighting with external commands.

This is where `extern` comes in. The `extern` keyword allows you to write a full signature for the command that lives outside of Nushell so that you get all the benefits above. If you take a look at the default config, you'll notice that there are a few extern calls in there. Here's one of them:

```
  export extern "git push" [
    remote?: string@"nu-complete git remotes",  # the name of the remote
    refspec?: string@"nu-complete git branches" # the branch / refspec
    --verbose(-v)                               # be more verbose
    --quiet(-q)                                 # be more quiet
    --repo: string                              # repository
    --all                                       # push all refs
    --mirror                                    # mirror all refs
    --delete(-d)                                # delete refs
    --tags                                      # push tags (can't be used with --all
or --mirror)
    --dry-run(-n)                               # dry run
    --porcelain                                 # machine-readable output
    --force(-f)                                 # force updates
    --force-with-lease: string                  # require old value of ref to be at
this value
    --recurse-submodules: string                # control recursive pushing of
```

```
submodules
    --thin                                  # use thin pack
    --receive-pack: string                  # receive pack program
    --exec: string                          # receive pack program
    --set-upstream(-u)                      # set upstream for git pull/status
    --progress                              # force progress reporting
    --prune                                 # prune locally removed refs
    --no-verify                             # bypass pre-push hook
    --follow-tags                           # push missing but relevant tags
    --signed: string                        # GPG sign the push
    --atomic                                # request atomic transaction on
remote side
    --push-option(-o): string               # option to transmit
    --ipv4(-4)                              # use IPv4 addresses only
    --ipv6(-6)                              # use IPv6 addresses only
  ]
```

You'll notice this gives you all the same descriptive syntax that internal commands do, letting you describe flags, short flags, positional parameters, types, and more.

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

Types and custom completions

In the above example, you'll notice some types are followed by `@` followed by the name of a command. We talk more about custom completions in their own section.

Both the type (or shape) of the argument and the custom completion tell Nushell about how to complete values for that flag or position. For example, setting a shape to `path` allows Nushell to complete the value to a filepath for you. Using the `@` with a custom completion overrides this default behavior, letting the custom completion give you full completion list.

Format specifiers

Positional parameters can be made optional with a `?` (as seen above) the remaining parameters can be matched with `...` before the parameter name, which will return a list of arguments.

```
  export extern "git add" [
    ...pathspecs: glob
    # …
  ]
```

Limitations

There are a few limitations to the current `extern` syntax. In Nushell, flags and positional arguments are very flexible: flags can precede positional arguments, flags can be mixed into positional arguments, and flags can follow positional arguments. Many external commands are not this flexible. There is not yet a way to require a particular ordering of flags and positional arguments to the style required by the external.

The second limitation is that some externals require flags to be passed using = to separate the flag and the value. In Nushell, the = is a convenient optional syntax and there's currently no way to require its use.

Custom completions

Custom completions allow you to mix together two features of Nushell: custom commands and completions. With them, you're able to create commands that handle the completions for positional

parameters and flag parameters. These custom completions work both for custom commands and known external, or `extern`, commands.

There are two parts to a custom command: the command that handles a completion and attaching this command to the type of another command using `@`.

Example custom completion

Let's look at an example:

```
> def animals [] { ["cat", "dog", "eel" ] }
> def my-command [animal: string@animals] { print $animal }
>| my-command
cat                 dog                 eel
```

In the first line, we create a custom command that will return a list of three different animals. These are the values we'd like to use in the completion. Once we've created this command, we can now use it to provide completions for other custom commands and `externs`.

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 to the custom completion. This keeps the API cleaner, while still offering all the same benefits.

This is possible because custom completion tags using `@` are locked-in as the command is first parsed.

Context aware custom completions

It is possible to pass the context to the custom completion command. This is useful in situations where it is necessary to know previous arguments or flags to generate accurate completions.

Let's apply this concept to the previous example:

```
module commands {
    def animals [] {
        ["cat", "dog", "eel" ]
    }

    def animal-names [context: string] {
        {
            cat: ["Missy", "Phoebe"]
            dog: ["Lulu", "Enzo"]
            eel: ["Eww", "Slippy"]
        } | get -i ($context | split words | last)
    }

    export def my-command [
        animal: string@animals
        name: string@animal-names
    ] {
        print $"The ($animal) is named ($name)."
    }
}
```

Here, the command `animal-names` returns the appropriate list of names. This is because `$context` is a string with where the value is the command that has been typed until now.

```
>| my-command
cat                 dog                 eel
>| my-command dog
Lulu                Enzo
>my-command dog enzo
The dog is named Enzo
```

On the second line, once we press the `<tab>` key, the argument `"my-command dog"` is passed to the `animal-names` command as context.

Custom completion and `extern`

A powerful combination is adding custom completions to known `extern` commands. These work the same way as adding a custom completion to a custom command: by creating the custom completion and then attaching it with a `@` to the type of one of the positional or flag arguments of the `extern`.

If you look closely at the examples in the default config, you'll see this:

```
export extern "git push" [
    remote?: string@"nu-complete git remotes",  # the name of the remote
    refspec?: string@"nu-complete git branches" # the branch / refspec
    ...
]
```

Custom completions will serve the same role in this example as in the previous examples. The examples above call into two different custom completions, based on the position the user is currently in.

Custom descriptions

As an alternative to returning a list of strings, a completion function can also return a list of records with a `value` and `description` field.

```
def my_commits [] {
    [
```

```
        { value: "5c2464", description: "Add .gitignore" },
        { value: "f3a377", description: "Initial commit" }
    ]
}
```

Note

with the following snippet

```
def my-command [commit: string@my_commits] {
    print $commit
}
```

be aware that, even though the completion menu will show you something like

```
>_ my-command <TAB>
5c2464  Add .gitignore
f3a377  Initial commit
```

only the value, i.e. “5c2464” or “f3a377”, will be used in the command arguments!

External completions

External completers can also be integrated, instead of relying solely on Nushell ones.

For this, set the `external_completer` field in `config.nu` to a closure which will be evaluated if no Nushell completions were found.

```
> $env.config.completions.external = {
>     enable: true
>     max_results: 100
>     completer: $completer
> }
```

You can configure the closure to run an external completer, such as carapace.

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

An external completer is a function that takes the current command as a string list, and outputs a list of records with `value` and `description` keys, like custom completion functions.

Note This closure will accept the current command as a list. For example, typing `my-command --arg1 <tab>` will receive `[my-command --arg1 " "]`.

This example will enable carapace external completions:

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

More examples of custom completers can be found in the cookbook.

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 `$env.config.table.mode` setting in `config.nu`. Here is an example:

```
> $env.config = {
    table: {
        mode: rounded
    }
}
```

Here are the current options for `$env.config.table.mode`:

- `rounded` # of course, this is the best one :)
- `basic`
- `compact`
- `compact_double`
- `light`
- `thin`
- `with_love`
- `reinforced`
- `heavy`
- `none`
- `other`

Color symbologies

- `r` - normal color red's abbreviation
- `rb` - normal color red's abbreviation with bold attribute
- `red` - normal color red
- `red_bold` - normal color red with bold attribute
- `"#ff0000"` - "#hex" format foreground color red (quotes are required)
- `{ fg: "#ff0000" bg: "#0000ff" attr: b }` - "full #hex" format foreground red in "#hex" format with a background of blue in "#hex" format with an attribute of bold abbreviated.

attributes

code	meaning
l	blink
b	bold
d	dimmed
h	hidden
i	italic
r	reverse
s	strikethrough
u	underline
n	nothing
	defaults to nothing

`normal` `colors` and `abbreviations`

code	name
g	green
gb	green_bold
gu	green_underline
gi	green_italic
gd	green_dimmed
gr	green_reverse
gbl	green_blink
gst	green_strike
lg	light_green
lgb	light_green_bold
lgu	light_green_underline
lgi	light_green_italic
lgd	light_green_dimmed
lgr	light_green_reverse
lgbl	light_green_blink
lgst	light_green_strike
r	red
rb	red_bold
ru	red_underline
ri	red_italic
rd	red_dimmed
rr	red_reverse
rbl	red_blink
rst	red_strike
lr	light_red
lrb	light_red_bold
lru	light_red_underline
lri	light_red_italic
lrd	light_red_dimmed
lrr	light_red_reverse
lrbl	light_red_blink
lrst	light_red_strike
u	blue
ub	blue_bold
uu	blue_underline
ui	blue_italic
ud	blue_dimmed
ur	blue_reverse
ubl	blue_blink
ust	blue_strike

`"#hex"` format

The “#hex” format is one way you typically see colors represented. It’s simply the `#` character followed by 6 characters. The first two are for `red`, the second two are for `green`, and the third two are for `blue`. It’s important that this string be surrounded in quotes, otherwise Nushell thinks it’s a commented out string.

Example: The primary `red` color is `"#ff0000"` or `"#FF0000"`. Upper and lower case in letters shouldn’t make a difference.

This `"#hex"` format allows us to specify 24-bit truecolor tones to different parts of Nushell.

`full "#hex"` format

The `full "#hex"` format is a take on the `"#hex"` format but allows one to specify the foreground, background, and attributes in one line.

Example: `{ fg: "#ff0000" bg: "#0000ff" attr: b }`

- foreground of red in “#hex” format
- background of blue in “#hex” format
- attribute of bold abbreviated

`Primitive values`

Primitive values are things like `int` and `string`. Primitive values and shapes can be set with a variety of color symbologies seen above.

This is the current list of primitives. Not all of these are configurable. The configurable ones are marked with *.

primitive	default color	configurable
any		
binary	Color::White.normal()	*
block	Color::White.normal()	*
bool	Color::White.normal()	*
cellpath	Color::White.normal()	*
condition		
custom		
date	Color::White.normal()	*
duration	Color::White.normal()	*
expression		
filesize	Color::White.normal()	*
float	Color::White.normal()	*
glob		
import		
int	Color::White.normal()	*
list	Color::White.normal()	*
nothing	Color::White.normal()	*
number		
operator		
path		
range	Color::White.normal()	*
record	Color::White.normal()	*
signature		
string	Color::White.normal()	*
table		
var		
vardecl		
variable		

special “primitives” (not really primitives but they exist solely for coloring)

primitive	default color	configurable
leading_trailing_space_bg	Color::Rgb(128, 128, 128))	*
header	Color::Green.bold()	*
empty	Color::Blue.normal()	*
row_index	Color::Green.bold()	*
hints	Color::DarkGray.normal()	*

Here’s a small example of changing some of these values.

```
> let config = {
    color_config: {
        separator: purple
```

```
        leading_trailing_space_bg: "#ffffff"
        header: gb
        date: wd
        filesize: c
        row_index: cb
        bool: red
        int: green
        duration: blue_bold
        range: purple
        float: red
        string: white
        nothing: red
        binary: red
        cellpath: cyan
        hints: dark_gray
    }
}
```

Here's another small example using multiple color syntaxes with some comments.

```
> let config = {
    color_config: {
        separator: "#88b719" # this sets only the foreground color like PR #486
        leading_trailing_space_bg: white # this sets only the foreground color in the
original style
        header: { # this is like PR #489
            fg: "#B01455", # note, quotes are required on the values with hex colors
            bg: "#ffb900", # note, commas are not required, it could also be all on
one line
            attr: bli # note, there are no quotes around this value. it works with or
without quotes
        }
        date: "#75507B"
        filesize: "#729fcf"
        row_index: {
            # note, that this is another way to set only the foreground, no need to
specify bg and attr
            fg: "#e50914"
        }
    }
}
```

`Shape` values

As mentioned above, `shape` is a term used to indicate the syntax coloring.

Here's the current list of flat shapes.

shape	default style	configurable
shape_block	fg(Color::Blue).bold()	*
shape_bool	fg(Color::LightCyan)	*
shape_custom	bold()	*
shape_external	fg(Color::Cyan)	*
shape_externalarg	fg(Color::Green).bold()	*
shape_filepath	fg(Color::Cyan)	*
shape_flag	fg(Color::Blue).bold()	*
shape_float	fg(Color::Purple).bold()	*
shape_garbage	fg(Color::White).on(Color::Red).bold()	*
shape_globpattern	fg(Color::Cyan).bold()	*
shape_int	fg(Color::Purple).bold()	*
shape_internalcall	fg(Color::Cyan).bold()	*
shape_list	fg(Color::Cyan).bold()	*
shape_literal	fg(Color::Blue)	*
shape_nothing	fg(Color::LightCyan)	*
shape_operator	fg(Color::Yellow)	*
shape_range	fg(Color::Yellow).bold()	*
shape_record	fg(Color::Cyan).bold()	*
shape_signature	fg(Color::Green).bold()	*
shape_string	fg(Color::Green)	*
shape_string_interpolation	fg(Color::Cyan).bold()	*
shape_table	fg(Color::Blue).bold()	*
shape_variable	fg(Color::Purple)	*

Here's a small example of how to apply color to these items. Anything not specified will receive the default color.

```
> $env.config = {
    color_config: {
        shape_garbage: { fg: "#FFFFFF" bg: "#FF0000" attr: b}
        shape_bool: green
        shape_int: { fg: "#0000ff" attr: b}
    }
}
```

`Prompt` configuration and coloring

The Nushell prompt is configurable through these environment variables and config items:

- `PROMPT_COMMAND`: Code to execute for setting up the prompt (block)
- `PROMPT_COMMAND_RIGHT`: Code to execute for setting up the *RIGHT* prompt (block) (see oh-my.nu in nu_scripts)
- `PROMPT_INDICATOR` = "〉": The indicator printed after the prompt (by default ">"-like Unicode symbol)
- `PROMPT_INDICATOR_VI_INSERT` = ":"
- `PROMPT_INDICATOR_VI_NORMAL` = "v"

- PROMPT_MULTILINE_INDICATOR = ":::"
- render_right_prompt_on_last_line: Bool value to enable or disable the right prompt to be rendered on the last line of the prompt

Example: For a simple prompt one could do this. Note that PROMPT_COMMAND requires a block whereas the others require a string.

```
> $env.PROMPT_COMMAND = { build-string (date now | format date '%m/%d/%Y %I:%M:
%S%.3f') ': ' (pwd | path basename) }
```

If you don't like the default PROMPT_INDICATOR you could change it like this.

```
> $env.PROMPT_INDICATOR = "> "
```

If you're using starship, you'll most likely want to show the right prompt on the last line of the prompt, just like zsh or fish. You could modify the config.nu file, just set render_right_prompt_on_last_line to true:

```
config {
    render_right_prompt_on_last_line = true
    ...
}
```

Coloring of the prompt is controlled by the block in PROMPT_COMMAND where you can write your own custom prompt. We've written a slightly fancy one that has git statuses located in the nu_scripts repo.

Transient prompt

If you want a different prompt displayed for previously entered commands, you can use Nushell's transient prompt feature. This can be useful if your prompt has lots of information that is unnecessary to show for previous lines (e.g. time and Git status), since you can make it so that previous lines show with a shorter prompt.

Each of the PROMPT_* variables has a corresponding TRANSIENT_PROMPT_* variable to be used for changing that segment when displaying past prompts: TRANSIENT_PROMPT_COMMAND, TRANSIENT_PROMPT_COMMAND_RIGHT, TRANSIENT_PROMPT_INDICATOR, TRANSIENT_PROMPT_INDICATOR_VI_INSERT, TRANSIENT_PROMPT_INDICATOR_VI_NORMAL, TRANSIENT_PROMPT_MULTILINE_INDICATOR. By default, the PROMPT_* variables are used for displaying past prompts.

For example, if you want to make past prompts show up without a left prompt entirely and leave only the indicator, you can use:

```
> $env.TRANSIENT_PROMPT_COMMAND = ""
```

If you want to go back to the normal left prompt, you'll have to unset TRANSIENT_PROMPT_COMMAND:

```
> hide-env TRANSIENT_PROMPT_COMMAND
```

LS_COLORS colors for the ls 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, 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 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.

```
$env.LS_COLORS = (vivid generate molokai | str trim)
```

If `LS_COLORS` is not set, nushell will default to a built-in `LS_COLORS` setting, based on 8-bit (extended) ANSI colors.

Theming

Theming combines all the coloring above. Here's a quick example of one we put together quickly to demonstrate the ability to theme. This is a spin on the `base16` themes that we see so widespread on the web.

The key to making theming work is to make sure you specify all themes and colors you're going to use in the `config.nu` file *before* you declare the `let config =` line.

```
# let's define some colors

let base00 = "#181818" # Default Background
let base01 = "#282828" # Lighter Background (Used for status bars, line number and
folding marks)
let base02 = "#383838" # Selection Background
let base03 = "#585858" # Comments, Invisibles, Line Highlighting
let base04 = "#b8b8b8" # Dark Foreground (Used for status bars)
let base05 = "#d8d8d8" # Default Foreground, Caret, Delimiters, Operators
let base06 = "#e8e8e8" # Light Foreground (Not often used)
let base07 = "#f8f8f8" # Light Background (Not often used)
let base08 = "#ab4642" # Variables, XML Tags, Markup Link Text, Markup Lists, Diff
Deleted
let base09 = "#dc9656" # Integers, Boolean, Constants, XML Attributes, Markup Link
Url
let base0a = "#f7ca88" # Classes, Markup Bold, Search Text Background
let base0b = "#a1b56c" # Strings, Inherited Class, Markup Code, Diff Inserted
let base0c = "#86c1b9" # Support, Regular Expressions, Escape Characters, Markup
Quotes
let base0d = "#7cafc2" # Functions, Methods, Attribute IDs, Headings
let base0e = "#ba8baf" # Keywords, Storage, Selector, Markup Italic, Diff Changed
let base0f = "#a16946" # Deprecated, Opening/Closing Embedded Language Tags, e.g. <?
php ?>

# we're creating a theme here that uses the colors we defined above.

let base16_theme = {
    separator: $base03
    leading_trailing_space_bg: $base04
    header: $base0b
    date: $base0e
    filesize: $base0d
    row_index: $base0c
    bool: $base08
    int: $base0b
    duration: $base08
    range: $base08
    float: $base08
```

```
    string: $base04
    nothing: $base08
    binary: $base08
    cellpath: $base08
    hints: dark_gray

    # shape_garbage: { fg: $base07 bg: $base08 attr: b} # base16 white on red
    # but i like the regular white on red for parse errors
    shape_garbage: { fg: "#FFFFFF" bg: "#FF0000" attr: b}
    shape_bool: $base0d
    shape_int: { fg: $base0e attr: b}
    shape_float: { fg: $base0e attr: b}
    shape_range: { fg: $base0a attr: b}
    shape_internalcall: { fg: $base0c attr: b}
    shape_external: $base0c
    shape_externalarg: { fg: $base0b attr: b}
    shape_literal: $base0d
    shape_operator: $base0a
    shape_signature: { fg: $base0b attr: b}
    shape_string: $base0b
    shape_filepath: $base0d
    shape_globpattern: { fg: $base0d attr: b}
    shape_variable: $base0e
    shape_flag: { fg: $base0d attr: b}
    shape_custom: {attr: b}
}

# now let's apply our regular config settings but also apply the "color_config:"
theme that we specified above.

let config = {
  filesize_metric: true
  table_mode: rounded # basic, compact, compact_double, light, thin, with_love,
rounded, reinforced, heavy, none, other
  use_ls_colors: true
  color_config: $base16_theme # <-- this is the theme
  use_grid_icons: true
  footer_mode: always #always, never, number_of_rows, auto
  animate_prompt: false
  float_precision: 2
  use_ansi_coloring: true
  filesize_format: "b" # b, kb, kib, mb, mib, gb, gib, tb, tib, pb, pib, eb, eib,
auto
  edit_mode: emacs # vi
  max_history_size: 10000
  log_level: error
}
```

if you want to go full-tilt on theming, you'll want to theme all the items I mentioned at the very beginning, including LS_COLORS, and the prompt. Good luck!

Working on light background terminal

Nushell's default config file contains a light theme definition, if you are working on a light background terminal, you can apply the light theme easily.

```
# in $nu.config-path
$env.config = {
```

```
  ...
  color_config: $dark_theme   # if you want a light theme, replace `$dark_theme` to
`$light_theme`
  ...
}
```

You can just change it to light theme by replacing `$dark_theme` to `$light_theme`

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

Accessibility

It's often desired to have the minimum amount of decorations when using a screen reader. In those cases, it's possible to disable borders and other decorations for both table and errors with the following options:

```
# in $nu.config-path
$env.config = {
  ...
  table: {
   ...
    mode: "none"
   ...
  }
  error_style: "plain"
  ...
}
```

Line editor menus (completion, history, help...)

Reedline (Nu's line editor) style is not using the `color_config` key. Instead, each menu has its own style to be configured separately. See the section dedicated to Reedline's menus configuration to learn more on this.

Hooks

Hooks allow you to run a code snippet at some predefined situations. They are only available in the interactive mode (REPL), they do not work if you run a Nushell with a script (`nu script.nu`) or commands (`nu -c "print foo"`) arguments.

Currently, we support these types of hooks:

- `pre_prompt` : Triggered before the prompt is drawn
- `pre_execution` : Triggered before the line input starts executing
- `env_change` : Triggered when an environment variable changes
- `display_output` : A block that the output is passed to (experimental).
- `command_not_found` : Triggered when a command is not found.

To make it clearer, we can break down Nushell's execution cycle. The steps to evaluate one line in the REPL mode are as follows:

1. Check for `pre_prompt` hooks and run them
2. Check for `env_change` hooks and run them

3. Display prompt and wait for user input
4. After user typed something and pressed "Enter": Check for `pre_execution` hooks and run them
5. Parse and evaluate user input
6. If a command is not found: Run the `command_not_found` hook. If it returns a string, show it.
7. If `display_output` is defined, use it to print command output
8. Return to 1.

Basic Hooks

To enable hooks, define them in your config:

```
$env.config = {
    # ...other config...

    hooks: {
        pre_prompt: { print "pre prompt hook" }
        pre_execution: { print "pre exec hook" }
        env_change: {
            PWD: {|before, after| print $"changing directory from ($before) to
($after)" }
        }
    }
}
```

Try putting the above to your config, running Nushell and moving around your filesystem. When you change a directory, the `PWD` environment variable changes and the change triggers the hook with the previous and the current values stored in `before` and `after` variables, respectively.

Instead of defining just a single hook per trigger, it is possible to define a **list of hooks** which will run in sequence:

```
$env.config = {
    ...other config...

    hooks: {
        pre_prompt: [
            { print "pre prompt hook" }
            { print "pre prompt hook2" }
        ]
        pre_execution: [
            { print "pre exec hook" }
            { print "pre exec hook2" }
        ]
        env_change: {
            PWD: [
                {|before, after| print $"changing directory from ($before) to
($after)" }
                {|before, after| print $"changing directory from ($before) to
($after) 2" }
            ]
        }
    }
}
```

Also, it might be more practical to update the existing config with new hooks, instead of defining the whole config from scratch:

```
$env.config = ($env.config | upsert hooks {
    pre_prompt: ...
    pre_execution: ...
    env_change: {
        PWD: ...
    }
})
```

Changing Environment

One feature of the hooks is that they preserve the environment. Environment variables defined inside the hook **block** will be preserved in a similar way as `def --env`. You can test it with the following example:

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

> $env.SPAM
eggs
```

The hook blocks otherwise follow the general scoping rules, i.e., commands, aliases, etc. defined within the block will be thrown away once the block ends.

Conditional Hooks

One thing you might be tempted to do is to activate an environment whenever you enter a directory:

```
$env.config = ($env.config | upsert hooks {
    env_change: {
        PWD: [
            {|before, after|
                if $after == /some/path/to/directory {
                    load-env { SPAM: eggs }
                }
            }
        ]
    }
})
```

This won't work because the environment will be active only within the `if` 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**:

```
$env.config = ($env.config | upsert hooks {
    env_change: {
        PWD: [
            {
                condition: {|before, after| $after == /some/path/to/directory }
                code: {|before, after| load-env { SPAM: eggs } }
            }
        ]
    }
})
```

When the hook triggers, it evaluates the `condition` block. If it returns `true`, the `code` block will be evaluated. If it returns `false`, nothing will happen. If it returns something else, an error will be thrown. The `condition` field can also be omitted altogether in which case the hook will always evaluate.

The `pre_prompt` and `pre_execution` hook types also support the conditional hooks but they don't accept the `before` and `after` parameters.

Hooks as Strings

So far a hook was defined as a block that preserves only the environment, but nothing else. To be able to define commands or aliases, it is possible to define the `code` field as **a string**. You can think of it as if you typed the string into the REPL and hit Enter. So, the hook from the previous section can be also written as

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

> $env.SPAM
eggs
```

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

```
$env.config = ($env.config | upsert hooks {
    env_change: {
        PWD: [
            {
                condition: {|_, after| $after == /some/path/to/directory }
                code: 'def foo [] { print "foo" }'
            }
            {
                condition: {|before, _| $before == /some/path/to/directory }
                code: 'hide foo'
            }
        ]
    }
})
```

When defining a hook as a string, the `$before` and `$after` variables are set to the previous and current environment variable value, respectively, similarly to the previous examples:

```
$env.config = ($env.config | upsert hooks {
    env_change: {
        PWD: {
            code: 'print $"changing directory from ($before) to ($after)"'
        }
    }
}
```

Examples

Adding a single hook to existing config

An example for PWD env change hook:

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

```
    if $val == null {
        $val | append {|before, after| print $"changing directory from ($before) to
($after)" }
    } else {
        [
            {|before, after| print $"changing directory from ($before) to ($after)" }
        ]
    }
})
```

Automatically activating an environment when entering a directory

This one looks for `test-env.nu` in a directory

```
$env.config = ($env.config | upsert hooks.env_change.PWD {
    [
        {
            condition: {|_, after|
                ($after == '/path/to/target/dir'
                    and ($after | path join test-env.nu | path exists))
            }
            code: "overlay use test-env.nu"
        }
        {
            condition: {|before, after|
                ('/path/to/target/dir' not-in $after
                    and '/path/to/target/dir' in $before
                    and 'test-env' in (overlay list))
            }
            code: "overlay hide test-env --keep-env [ PWD ]"
        }
    ]
})
```

Filtering or diverting command output

You can use the `display_output` hook to redirect the output of commands. You should define a block that works on all value types. The output of external commands is not filtered through `display_output`.

This hook can display the output in a separate window, perhaps as rich HTML text. Here is the basic idea of how to do that:

```
$env.config = ($env.config | upsert hooks {
    display_output: { to html --partial --no-color | save --raw /tmp/nu-output.html }
})
```

You can view the result by opening `file:///tmp/nu-output.html` in a web browser. Of course this isn't very convenient unless you use a browser that automatically reloads when the file changes. Instead of the `save` command, you would normally customize this to send the HTML output to a desired window.

Changing how output is displayed

You can change to default behavior of how output is displayed by using the `display_output` hook. Here is an example that changes the default display behavior to show a table 1 layer deep if the terminal is wide enough, or collapse otherwise:

```
$env.config = ($env.config | upsert hooks {
    display_output: {if (term size).columns >= 100 { table -ed 1 } else { table }}
})
```

`command_not_found` hook in *Arch Linux*

The following hook uses the `pkgfile` command, to find which packages commands belong to in *Arch Linux*.

```
$env.config = {
    ...other config...

    hooks: {
        ...other hooks...

        command_not_found: {
            |cmd_name| (
                try {
                    let pkgs = (pkgfile --binaries --verbose $cmd_name)
                    if ($pkgs | is-empty) {
                        return null
                    }
                    (
                        $"(ansi $env.config.color_config.shape_external)($cmd_name)
(ansi reset) " +
                        $"may be found in the following packages:\n($pkgs)"
                    )
                }
            )
        }
    }
}
```

Background tasks with Nu

Currently, Nushell doesn't have built-in background task management feature, but you can make it "support" background task with some tools, here are some examples:

1. Using a third-party task management tools, like pueue
2. Using a terminal multiplexer, like tmux or zellij

Using nu with pueue

The module borrows the power of pueue, it is possible to schedule background tasks to pueue, and manage those tasks (such as viewing logs, killing tasks, or getting the running status of all tasks, creating groups, pausing tasks etc etc)

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

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

Here is a setup example to make Nushell "support" background tasks:

1. Install pueue
2. run `pueued`, you can refer to start-the-daemon page for more information.
3. Put the task.nu file under `$env.NU_LIB_DIRS`.
4. Add a line to the `$nu.config-path` file: `use task.nu`
5. Restart Nushell.

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

Cons: It spawns a new Nushell interpreter to execute every single task, so it doesn't inherit current scope's variables, custom commands, alias definition. It only inherits environment variables whose value can be converted to a string. Therefore, if you want to use custom commands or variables, you have to `use` or `def` them within the given block.

Using nu with terminal multiplexer

You can choose and install a terminal multiplexer and use it.

It allows you to easily switch between multiple programs in one terminal, detach them (they continue to run in the background) and reconnect them to a different terminal. As a result, it is very flexible and usable.

Coming to Nu

If you are familiar with other shells or programming languages, you might find this chapter useful to get up to speed.

Coming from Bash shows how some patterns typical for Bash, or POSIX shells in general, can be mapped to Nushell. Similarly, Coming from CMD.EXE shows how built-in commands in the Windows Command Prompt can be mapped to Nushell.

Similar comparisons are made for some other shells and domain-specific languages, imperative languages, and functional languages. A separate comparison is made specifically for operators.

Coming from Bash

If you're coming from `Git Bash` on Windows, then the external commands you're used to (bash, grep, etc) will not be available in `nu` by default (unless you had explicitly made them available in the Windows Path environment variable). To make these commands available in `nu` as well, add the following line to your `config.nu` with either `append` or `prepend`.

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

Note: this table assumes Nu 0.91.0 or later.

Bash	Nu	Task
ls	ls	Lists the files in the current directory
ls <dir>	ls <dir>	Lists the files in the given directory
ls pattern*	ls pattern*	Lists files that match a given pattern
ls -la	ls --long --all or ls -la	List files with all available information, including hidden files
ls -d */	ls | where type == dir	List directories
find . -name *.rs	ls **/*.rs	Find recursively all files that match a given pattern
find . -name Makefile | xargs vim	ls **/Makefile | get name | vim ...$in	Pass values as command parameters
cd <directory>	cd <directory>	Change to the given directory
cd	cd	Change to the home directory
cd -	cd -	Change to the previous directory
mkdir <path>	mkdir <path>	Creates the given path
mkdir -p <path>	mkdir <path>	Creates the given path, creating parents as necessary
touch test.txt	touch test.txt	Create a file
> <path>	out> <path> or o> <path>	Save command output to a file
	| save <path>	Save command output to a file as structured data
>> <path>	out>> <path> or o>> <path>	Append command output to a file
	| save --append <path>	Append command output to a file as structured data
> /dev/null	| ignore	Discard command output
> /dev/null 2>&1	out+err>| ignore or o+e>| ignore	Discard command output, including stderr
command 2>&1 | less	command out+err>| less or command o+e>| less	Pipe stdout and stderr of a command into less
cmd1 | tee log.txt | cmd2	cmd1 | tee { save log.txt } | cmd2	Tee command output to a log file
cat <path>	open --raw <path>	Display the contents of the given file
	open <path>	Read a file as structured data
mv <source> <dest>	mv <source> <dest>	Move file to new location
cp <source> <dest>	cp <source> <dest>	Copy file to new location
cp -r <source> <dest>	cp -r <source> <dest>	Copy directory to a new location, recursively
rm <path>	rm <path>	Remove the given file
	rm -t <path>	Move the given file to the system trash
rm -rf <path>	rm -r <path>	Recursively removes the given path
[illegible]	[illegible]	[illegible]

Coming from CMD.EXE

This table was last updated for Nu 0.67.0.

CMD.EXE	Nu	Task
ASSOC		Displays or modifies file extension associations
BREAK		Trigger debugger breakpoint
CALL <filename.bat>	<filename.bat>	Run a batch program
	nu <filename>	Run a nu script in a fresh context
	source <filename>	Run a nu script in this context
	use <filename>	Run a nu script as a module
CD or CHDIR	$env.PWD	Get the present working directory
CD <directory>	cd <directory>	Change the current directory
CD /D <drive:directory>	cd <drive:directory>	Change the current directory
CLS	clear	Clear the screen
COLOR		Set the console default foreground/background colors
	ansi {flags} (code)	Output ANSI codes to change color
COPY <source> <destination>	cp <source> <destination>	Copy files
COPY <file1>+<file2> <destination>	[<file1>, <file2>] | each { open --raw } | str join | save --raw <destination>	Append multiple files into one
DATE /T	date now	Get the current date
DATE		Set the date
DEL <file> or ERASE <file>	rm <file>	Delete files
DIR	ls	List files in the current directory
ECHO <message>	print <message>	Print the given values to stdout
ECHO ON		Echo executed commands to stdout
ENDLOCAL	export-env	Change env in the caller
EXIT	exit	Close the prompt or script
FOR %<var> IN (<set>) DO <command>	for $<var> in <set> { <command> }	Run a command for each item in a set
FTYPE		Displays or modifies file types used in file extension associations
GOTO		Jump to a label
IF ERRORLEVEL <number> <command>	if $env.LAST_EXIT_CODE >= <number> { <command> }	Run a command if the last command returned an error code >= specified
IF <string> EQU <string> <command>	if <string> == <string> { <command> }	Run a command if strings match
[illegible] <command>	[illegible] continue <path>). ."	[illegible]

Before Nu version 0.67, Nu used to use CMD.EXE to launch external commands, which meant that the above builtins could be run as an `^external` command. As of version 0.67, however, Nu no longer uses CMD.EXE to launch externals, meaning the above builtins cannot be run from within Nu, except for `ASSOC`, `CLS`, `ECHO`, `FTYPE`, `MKLINK`, `PAUSE`, `START`, `VER`, and `VOL`, which are explicitly allowed to be interpreted by CMD if no executable by that name exists.

Nu map from other shells and domain specific languages

The idea behind this table is to help you understand how Nu builtins and plugins relate to other known shells and domain specific languages. We've tried to produce a map of relevant Nu commands and what their equivalents are in other languages. Contributions are welcome.

Note: this table assumes Nu 0.43 or later.

Nu map from imperative languages

The idea behind this table is to help you understand how Nu built-ins and plugins relate to imperative languages. We've tried to produce a map of programming-relevant Nu commands and what their equivalents are in other languages. Contributions are welcome.

Note: this table assumes Nu 0.43 or later.

Nushell	Python	Kotlin (Java)	C++	Rust
append	list.append, set.add	add	push_back, emplace_back	push, push_back
math avg	statistics.mean			
calc, = math	math operators	math operators	math operators	math operators
count	len	size, length	length	len
cp	shutil.copy			fs::copy
date	datetime.date.today	java.time.LocalDate.now		
drop	list[:−3]			
du	shutil.disk_usage			
each	for	for	for	for
exit	exit	System.exit, kotlin.system.exitProcess	exit	exit
http get	urllib.request.urlopen			
first	list[:x]	List[0], peek	vector[0], top	Vec[0]
format	format	format	format	format!
from	csv, json, sqlite3			
get	dict["key"]	Map["key"]	map["key"]	HashMap[“key”], get, entry
group-by	itertools.groupby	groupBy		group_by
headers	keys			
help	help			
insert	dict["key"] = val		map.insert({ 20, 130 })	map.insert("key", val)
is-empty	is None, is []	isEmpty	empty	is_empty
take	list[:x]			&Vec[..x]
take until	itertools.takewhile			
take while	itertools.takewhile			
kill	os.kill			
last	list[-x:]			&Vec[Vec.len()-1]
lines	split, splitlines	split	views::split	split, split_whitespace, rsplit, lines
ls	os.listdir			fs::read_dir
match	match	when		match
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	functions	map.re[illegible]

Collections.sort

Nu map from functional languages

The idea behind this table is to help you understand how Nu builtins and plugins relate to functional languages. We've tried to produce a map of relevant Nu commands and what their equivalents are in other languages. Contributions are welcome.

Note: this table assumes Nu 0.43 or later.

Nushell	Clojure	Tablecloth (Ocaml / Elm)	Haskell
append	conj, into, concat	append, (++), concat, concatMap	(++)
into binary	Integer/toHexString		showHex
count	count	length, size	length, size
date	java.time.LocalDate/now		
each	map, mapv, iterate	map, forEach	map, mapM
exit	System/exit		
first	first	head	head
format	format		Text.Printf.printf
group-by	group-by		group, groupBy
help	doc		
is-empty	empty?	isEmpty	
last	last, peek, take-last	last	last
lines			lines, words, split-with
match		match (Ocaml), case (Elm)	case
nth	nth	Array.get	lookup
open	with-open		
transpose	(apply mapv vector matrix)		transpose
prepend	cons	cons, ::	::
print	println		putStrLn, print
range, 1..10	range	range	1..10, ‘a’..’f’
reduce	reduce, reduce-kv	foldr	foldr
reverse	reverse, rseq	reverse, reverseInPlace	reverse
select	select-keys		
shuffle	shuffle		
size	count		size, length
skip	rest	tail	tail
skip until	drop-while		
skip while	drop-while	dropWhile	dropWhile, dropWhileEnd
sort-by	sort, sort-by, sorted-set-by	sort, sortBy, sortWith	sort, sortBy
split row	split, split-{at,with,lines}	split, words, lines	split, words, lines
str	clojure.string functions	String functions	
str join	join	concat	intercalate
str trim	trim, triml, trimr	trim, trimLeft, trimRight	strip
sum	apply +	sum	sum
take	take, drop-last, pop	take, init	take, init
take until	take-while	takeWhile	takeWhile
take while	take-while	takeWhile	takeWhile
uniq	filter, filter-kv, select	filter, filterMap	Data.Set

Nushell operator map

The idea behind this table is to help you understand how Nu operators relate to other language operators. We've tried to produce a map of all the nushell operators and what their equivalents are in other languages. Contributions are welcome.

Note: this table assumes Nu 0.14.1 or later.

Nushell	SQL	Python	.NET LINQ (C#)	PowerShell	Bash
==	=	==	==	-eq, -is	-eq
!=	!=, <>	!=	!=	-ne, -isnot	-ne
<	<	<	<	-lt	-lt
<=	<=	<=	<=	-le	-le
>	>	>	>	-gt	-gt
>=	>=	>=	>=	-ge	-ge
=~	like	re, in, startswith	Contains, StartsWith	-like, -contains	=~
!~	not like	not in	Except	-notlike, -notcontains	! "str1" =~ "str2"
+	+	+	+	+	+
-	-	-	-	-	-
*	*	*	*	*	*
/	/	/	/	/	/
**	pow	**	Power	Pow	**
in	in	re, in, startswith	Contains, StartsWith	-In	case in
not-in	not in	not in	Except	-NotIn	
and	and	and	&&	-And, &&	-a, &&
or	or	or	||	-Or, ||	-o, ||

(Not So) Advanced

While the "Advanced" title might sound daunting and you might be tempted to skip this chapter, in fact, some of the most interesting and powerful features can be found here.

Besides the built-in commands, Nushell has a standard library.

Nushell operates on *structured data*. You could say that Nushell is a "data-first" shell and a programming language. To further explore the data-centric direction, Nushell includes a full-featured dataframe processing engine using Polars as the backend. Make sure to check the Dataframes documentation if you want to process large data efficiently directly in your shell.

Values in Nushell contain some extra metadata. This metadata can be used, for example, to create custom errors.

Thanks to Nushell's strict scoping rules, it is very easy to iterate over collections in parallel which can help you speed up long-running scripts by just typing a few characters.

You can interactively explore data with the `explore` command.

Finally, you can extend Nushell's functionality with plugins. Almost anything can be a plugin as long as it communicates with Nushell in a protocol that Nushell understands.

Standard library (preview)

The standard library is located on the Git repository. At the moment it is an alpha development stage. You can find more documentation there.

Dataframes

To use the dataframe support you need a fully-featured build with `cargo build --features dataframe`. Starting with version 0.72, dataframes are *not* included with binary releases of Nushell. See the installation instructions 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 specification, and uses Polars as the motor for performing extremely fast columnar operations.

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 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

> All examples were run on Nushell version 0.33.1. (Command names are updated to Nushell 0.78)

File information

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

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

```
❯ let df = (dfr open .\Data7602DescendingYearOrder.csv)
❯ dfr ls
```

#	name	columns	rows
0	$df	5	5429252

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

```
❯ $df | dfr first
```

#	anzsic06	Area	year	geo_count	ec_count
0	A	A100100	2000	96	130

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

```
❯ $df | dfr dtypes
```

#	column	dtype
0	anzsic06	str
1	Area	str
2	year	i64
3	geo_count	i64
4	ec_count	i64

Loading the file

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

```
❯ timeit {open .\Data7602DescendingYearOrder.csv}
30sec 479ms 614us 400ns
```

Loading the file using native Nushell functionality took 30 seconds. Not bad for loading five million records! But we can do a bit better than that.

Let's now use Pandas. We are going to use the next script to load the file:

```
import pandas as pd

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

And the benchmark for it is:

```
❯ timeit {python load.py}
2sec 91ms 872us 900ns
```

That is a great improvement, from 30 seconds to 2 seconds. Nicely done, Pandas!

Probably we can load the data a bit faster. This time we will use Nushell's `dfr open` command:

```
❯ timeit {dfr open .\Data7602DescendingYearOrder.csv}
601ms 700us 700ns
```

This time it took us 0.6 seconds. Not bad at all.

Group-by comparison

Let's do a slightly more complex operation this time. We are going to group the data by year, and add groups using the column `geo_count`.

Again, we are going to start with a Nushell native command.

If you want to run this example, be aware that the next command will use a large amount of memory. This may affect the performance of your system while this is being executed.

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

6min 30sec 622ms 312us
```

So, six minutes to perform this aggregated operation.

Let's try the same operation in pandas:

```
import pandas as pd

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

And the result from the benchmark is:

```
❯ timeit {python .\load.py}

1sec 966ms 954us 800ns
```

Not bad at all. Again, pandas managed to get it done in a fraction of the time.

To finish the comparison, let's try Nushell dataframes. We are going to put all the operations in one `nu` file, to make sure we are doing similar operations:

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

and the benchmark with dataframes is:

```
❯ timeit {source load.nu}

557ms 658us 500ns
```

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

As you can see, Nushell's `Dataframe` commands 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 `dfr open` command like this:

```
❯ let df = (dfr open test_small.csv)
```

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

The command `dfr open` can read either **csv** or **parquet** files.

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

```
❯ dfr ls
```

#	name	columns	rows
0	$df	8	10

As you can see, the command shows the created dataframes together with basic information about them.

And if you want to see a preview of the loaded dataframe you can send the dataframe variable to the stream

```
❯ $df
```

#	int_1	int_2	float_1	float_2	first	second	third	word
0	1	11	0.10	1.00	a	b	c	first
1	2	12	0.20	1.00	a	b	c	second
2	3	13	0.30	2.00	a	b	c	third
3	4	14	0.40	3.00	b	a	c	second
4	0	15	0.50	4.00	b	a	a	third
5	6	16	0.60	5.00	b	a	a	second
6	7	17	0.70	6.00	b	c	a	third
7	8	18	0.80	7.00	c	c	b	eight
8	9	19	0.90	8.00	c	c	b	ninth
9	0	10	0.00	9.00	c	c	b	ninth

With the dataframe in memory we can start doing column operations with the `DataFrame`

If you want to see all the dataframe commands that are available you can use `scope commands | where category =~ dataframe`

Basic aggregations

Let's start with basic aggregations on the dataframe. Let's sum all the columns that exist in `df` by using the `aggregate` command

```
❯ $df | dfr sum
```

#	int_1	int_2	float_1	float_2	first	second	third	word

```
╭───┬─────────┬─────────┬───────────┬───────────┬─────────┬─────────┬─────────┬─────────╮
│ 0 │      40 │     145 │      4.50 │     46.00 │         │         │         │         │
╰───┴─────────┴─────────┴───────────┴───────────┴─────────┴─────────┴─────────┴─────────╯
```

As you can see, the aggregate function computes the sum for those columns where a sum makes sense. If you want to filter out the text column, you can select the columns you want by using the `dfr select` command

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

```
╭───┬─────────┬─────────┬───────────┬───────────╮
│ # │ int_1   │ int_2   │ float_1   │ float_2   │
├───┼─────────┼─────────┼───────────┼───────────┤
│ 0 │      40 │     145 │      4.50 │     46.00 │
╰───┴─────────┴─────────┴───────────┴───────────╯
```

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

```
❯ let res = ($df | dfr sum | dfr select int_1 int_2 float_1 float_2)
```

Type `let res = ( !! )` and press enter. This will auto complete the previously executed command. Note the space between (and !!.

And now we have two dataframes stored in memory

```
❯ dfr ls
```

```
╭───┬──────┬─────────┬──────╮
│ # │ name │ columns │ rows │
├───┼──────┼─────────┼──────┤
│ 0 │ $res │       4 │    1 │
│ 1 │ $df  │       8 │   10 │
╰───┴──────┴─────────┴──────╯
```

Pretty neat, isn't it?

You can perform several aggregations on the dataframe in order to extract basic information from the dataframe and do basic data analysis on your brand new dataframe.

Joining a DataFrame

It is also possible to join two dataframes using a column as reference. We are going to join our mini dataframe with another mini dataframe. Copy these lines in another file and create the corresponding dataframe (for these examples we are going to call it `test_small_a.csv`)

```
int_1,int_2,float_1,float_2,first
9,14,0.4,3.0,a
8,13,0.3,2.0,a
7,12,0.2,1.0,a
6,11,0.1,0.0,b
```

We use the `dfr open` command to create the new variable

```
❯ let df_a = (dfr open test_small_a.csv)
```

Now, with the second dataframe loaded in memory we can join them using the column called `int_1` from the left dataframe and the column `int_1` from the right dataframe

```
❯ $df | dfr join $df_a int_1 int_1
```

```
╭───┬─────────┬─────────┬───────────┬───────────┬─────────┬─────────┬─────────┬─────────┬─────────┬─────────────╮
│ # │ int_1   │ int_2   │ float_1   │ float_2   │ first   │ second  │ third   │  word   │ int_2_x │
float_1_x │ float_2_x │ first_x │
├───┼─────────┼─────────┼───────────┼───────────┼─────────┼─────────┼─────────┼─────────┼─────────┼─────────────┤
```

```
| 0 |     6 |    16 |    0.60 |    5.00 | b     | a      | a     | second |      11 |
0.10 |      0.00 | b       |
| 1 |     7 |    17 |    0.70 |    6.00 | b     | c      | a     | third  |      12 |
0.20 |      1.00 | a       |
| 2 |     8 |    18 |    0.80 |    7.00 | c     | c      | b     | eight  |      13 |
0.30 |      2.00 | a       |
| 3 |     9 |    19 |    0.90 |    8.00 | c     | c      | b     | ninth  |      14 |
0.40 |      3.00 | a       |
```

In `Nu` when a command has multiple arguments that are expecting multiple values we use brackets `[]` to enclose those values. In the case of `dfr join` we can join on multiple columns as long as they have the same type.

For example:

```
❯ $df | dfr join $df_a [int_1 first] [int_1 first]
```

```
| # | int_1 | int_2 | float_1 | float_2 | first | second | third |  word  | int_2_x |
float_1_x | float_2_x |
| 0 |     6 |    16 |    0.60 |    5.00 | b     | a      | a     | second |      11 |
0.10 |      0.00 |
```

By default, the join command does an inner join, meaning that it will keep the rows where both dataframes share the same value. You can select a left join to keep the missing rows from the left dataframe. You can also save this result in order to use it for further operations.

DataFrame group-by

One of the most powerful operations that can be performed with a DataFrame is the `dfr group-by`. This command will allow you to perform aggregation operations based on a grouping criteria. In Nushell, a `GroupBy` is a type of object that can be stored and reused for multiple aggregations. This is quite handy, since the creation of the grouped pairs is the most expensive operation while doing group-by and there is no need to repeat it if you are planning to do multiple operations with the same group condition.

To create a `GroupBy` object you only need to use the `dfr_group-by` command

```
❯ let group = ($df | dfr group-by first)
❯ $group
```

```
| LazyGroupBy | apply aggregation to complete execution plan |
```

When printing the `GroupBy` object we can see that it is in the background a lazy operation waiting to be completed by adding an aggregation. Using the `GroupBy` we can create aggregations on a column

```
❯ $group | dfr agg (dfr col int_1 | dfr sum)
```

```
| # | first | int_1 |
| 0 | b     |    17 |
| 1 | a     |     6 |
| 2 | c     |    17 |
```

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

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

#	first	int_1	int_2	float_1	float_2
0	a	3	11	0.60	3
1	b	4	14	2.20	4
2	c	3	10	1.70	3

As you can see, the `GroupBy` object is a very powerful variable and it is worth keeping in memory while you explore your dataset.

Creating Dataframes

It is also possible to construct dataframes from basic Nushell primitives, such as integers, decimals, or strings. Let's create a small dataframe using the command `dfr into-df`.

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

For the time being, not all of Nushell primitives can be converted into a dataframe. This will change in the future, as the dataframe feature matures

We can append columns to a dataframe in order to create a new variable. As an example, let's append two columns to our mini dataframe `$a`

```
❯ let a2 = ($a | dfr with-column $a.a --name a2 | dfr with-column $a.a --name a3)
❯ $a2
```

#	a	b	a2	a3
0	1	2	1	1
1	3	4	3	3
2	5	6	5	5

Nushell's powerful piping syntax allows us to create new dataframes by taking data from other dataframes and appending it to them. Now, if you list your dataframes you will see in total four dataframes

```
❯ dfr ls
```

#	name	columns	rows
0	$a2	4	3
1	$df_a	5	4
2	$df	8	10
3	$a	2	3
4	$res	4	1

One thing that is important to mention is how the memory is being optimized while working with dataframes, and this is thanks to **Apache Arrow** and **Polars**. In a very simple representation, each column in a DataFrame is an Arrow Array, which is using several memory specifications in order to

maintain the data as packed as possible (check Arrow columnar format). 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 `dfr into-df` command. For this reason, it isn't possible to change the value of a column in a dataframe. However, you can create new columns based on data from other columns or dataframes.

Working with Series

A `Series` is the building block of a `DataFrame`. Each Series represents a column with the same data type, and we can create multiple Series of different types, such as float, int or string.

Let's start our exploration with Series by creating one using the `dfr into-df` command:

```
❯ let new = ([9 8 4] | dfr into-df)
❯ $new
```

```
╭───┬───╮
│ # │ 0 │
├───┼───┤
│ 0 │ 9 │
│ 1 │ 8 │
│ 2 │ 4 │
╰───┴───╯
```

We have created a new series from a list of integers (we could have done the same using floats or strings)

Series have their own basic operations defined, and they can be used to create other Series. Let's create a new Series by doing some arithmetic on the previously created column.

```
❯ let new_2 = ($new * 3 + 10)
❯ $new_2
```

```
╭───┬────╮
│ # │ 0  │
├───┼────┤
│ 0 │ 37 │
│ 1 │ 34 │
│ 2 │ 22 │
╰───┴────╯
```

Now we have a new Series that was constructed by doing basic operations on the previous variable.

If you want to see how many variables you have stored in memory you can use `scope variables`

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

```
❯ let new_2 = ($new_2 | dfr rename "0" memorable)
❯ $new_2
```

```
╭───┬──────────╮
│ # │ memorable │
├───┼──────────┤
│ 0 │       37 │
│ 1 │       34 │
│ 2 │       22 │
╰───┴──────────╯
```

We can also do basic operations with two Series as long as they have the same data type

```
❯ $new - $new_2
```

```
╭
```

#	sub_0_memorable
0	-28
1	-26
2	-18

And we can add them to previously defined dataframes

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

#	a	b	new_col
0	1	2	9
1	3	4	8
2	5	6	4

The Series stored in a Dataframe can also be used directly, for example, we can multiply columns `a` and `b` to create a new Series

```
❯ $new_df.a * $new_df.b
```

#	mul_a_b
0	2
1	12
2	30

and we can start piping things in order to create new columns and dataframes

```
❯ let $new_df = ($new_df | dfr with-column ($new_df.a * $new_df.b / $new_df.new_col)
--name my_sum)
❯ $new_df
```

#	a	b	new_col	my_sum
0	1	2	9	0
1	3	4	8	1
2	5	6	4	7

Nushell's piping system can help you create very interesting workflows.

Series and masks

Series have another key use in when working with `DataFrames`, and it is the fact that we can build boolean masks out of them. Let's start by creating a simple mask using the equality operator

```
❯ let mask = ($new == 8)
❯ $mask
```

#	0
0	false
1	true
2	false

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

```
❯ $new_df | dfr filter-with $mask
```

#	a	b	new_col	my_sum
0	3	4	8	1

Now we have a new dataframe with only the values where the mask was true.

The masks can also be created from Nushell lists, for example:

```
❯ let mask1 = ([true true false] | dfr into-df)
❯ $new_df | dfr filter-with $mask1
```

#	a	b	new_col	my_sum
0	1	2	9	0
1	3	4	8	1

To create complex masks, we have the `AND`

```
❯ $mask and $mask1
```

#	and_0_0
0	false
1	true
2	false

and `OR` operations

```
❯ $mask or $mask1
```

#	or_0_0
0	true
1	true
2	false

We can also create a mask by checking if some values exist in other Series. Using the first dataframe that we created we can do something like this

```
❯ let mask3 = ($df | dfr col first | dfr is-in [b c])
❯ $mask3
```

input

#	expr	value
0	column	first
1	literal	Series[list]

```
│          │ └──┴─────────┴──────────────┘
│
│ function │ IsIn
│
│ options  │ FunctionOptions { collect_groups: ApplyFlat, input_wildcard_expansion:
false, auto_explode: tru │
│          │ e, fmt_str: "", cast_to_supertypes: true, allow_rename: false,
pass_name_to_apply: false }      │
└──────────┴
```

and this new mask can be used to filter the dataframe

```
❯ $df | dfr filter-with $mask3
```

#	int_1	int_2	float_1	float_2	first	second	third	word
0	4	14	0.40	3.00	b	a	c	second
1	0	15	0.50	4.00	b	a	a	third
2	6	16	0.60	5.00	b	a	a	second
3	7	17	0.70	6.00	b	c	a	third
4	8	18	0.80	7.00	c	c	b	eight
5	9	19	0.90	8.00	c	c	b	ninth
6	0	10	0.00	9.00	c	c	b	ninth

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

This is example is not updated to recent Nushell versions.

```
❯ $df | dfr get first | dfr set new --mask ($df.first =~ a)
```

#	string
0	new
1	new
2	new
3	b
4	b
5	b
6	b
7	c
8	c
9	c

Series as indices

Series can be also used as a way of filtering a dataframe by using them as a list of indices. For example, let's say that we want to get rows 1, 4, and 6 from our original dataframe. With that in mind, we can use the next command to extract that information

```
❯ let indices = ([1 4 6] | dfr into-df)
❯  $df | dfr take $indices
```

#	int_1	int_2	float_1	float_2	first	second	third	word
0	2	12	0.20	1.00	a	b	c	second
1	0	15	0.50	4.00	b	a	a	third

2	7	17	0.70	6.00	b	c	a	third

The command `dfr take` is very handy, especially if we mix it with other commands. Let's say that we want to extract all rows for the first duplicated element for column `first`. In order to do that, we can use the command `dfr arg-unique` as shown in the next example

```
❯ let indices = ($df | dfr get first | dfr arg-unique)
❯ $df | dfr take $indices
```

#	int_1	int_2	float_1	float_2	first	second	third	word
0	1	11	0.10	1.00	a	b	c	first
1	4	14	0.40	3.00	b	a	c	second
2	8	18	0.80	7.00	c	c	b	eight

Or what if we want to create a new sorted dataframe using a column in specific. We can use the `arg-sort` to accomplish that. In the next example we can sort the dataframe by the column `word`

The same result could be accomplished using the command `sort`

```
❯ let indices = ($df | dfr get word | dfr arg-sort)
❯ $df | dfr take $indices
```

#	int_1	int_2	float_1	float_2	first	second	third	word
0	8	18	0.80	7.00	c	c	b	eight
1	1	11	0.10	1.00	a	b	c	first
2	9	19	0.90	8.00	c	c	b	ninth
3	0	10	0.00	9.00	c	c	b	ninth
4	2	12	0.20	1.00	a	b	c	second
5	4	14	0.40	3.00	b	a	c	second
6	6	16	0.60	5.00	b	a	a	second
7	3	13	0.30	2.00	a	b	c	third
8	0	15	0.50	4.00	b	a	a	third
9	7	17	0.70	6.00	b	c	a	third

And finally, we can create new Series by setting a new value in the marked indices. Have a look at the next command

```
❯ let indices = ([0 2] | dfr into-df);
❯ $df | dfr get int_1 | dfr set-with-idx 123 --indices $indices
```

#	int_1
0	123
1	2
2	123
3	4
4	0
5	6
6	7
7	8
8	9
9	0

Unique values

Another operation that can be done with `Series` is to search for unique values in a list or column. Lets use again the first dataframe we created to test these operations.

The first and most common operation that we have is `value_counts`. This command calculates a count of the unique values that exist in a Series. For example, we can use it to count how many occurrences we have in the column `first`

```
❯ $df | dfr get first | dfr value-counts
```

#	first	counts
0	b	4
1	a	3
2	c	3

As expected, the command returns a new dataframe that can be used to do more queries.

Continuing with our exploration of `Series`, the next thing that we can do is to only get the unique unique values from a series, like this

```
❯ $df | dfr get first | dfr unique
```

#	first
0	c
1	b
2	a

Or we can get a mask that we can use to filter out the rows where data is unique or duplicated. For example, we can select the rows for unique values in column `word`

```
❯ $df | dfr filter-with ($df | dfr get word | dfr is-unique)
```

#	int_1	int_2	float_1	float_2	first	second	third	word
0	1	11	0.10	1.00	a	b	c	first
1	8	18	0.80	7.00	c	c	b	eight

Or all the duplicated ones

```
❯ $df | dfr filter-with ($df | dfr get word | dfr is-duplicated)
```

#	int_1	int_2	float_1	float_2	first	second	third	word
0	2	12	0.20	1.00	a	b	c	second
1	3	13	0.30	2.00	a	b	c	third
2	4	14	0.40	3.00	b	a	c	second
3	0	15	0.50	4.00	b	a	a	third
4	6	16	0.60	5.00	b	a	a	second
5	7	17	0.70	6.00	b	c	a	third
6	9	19	0.90	8.00	c	c	b	ninth
7	0	10	0.00	9.00	c	c	b	ninth

Lazy Dataframes

Lazy dataframes are a way to query data by creating a logical plan. The advantage of this approach is that the plan never gets evaluated until you need to extract data. This way you could chain together aggregations, joins and selections and collect the data once you are happy with the selected operations.

Let's create a small example of a lazy dataframe

```
❯ let a = ([[a b]; [1 a] [2 b] [3 c] [4 d]] | dfr into-lazy)
❯ $a
```

```
╭─────────────────┬──────────────────────────────────────────────────────────────╮
│ plan            │  DF ["a", "b"]; PROJECT */2 COLUMNS; SELECTION: "None"       │
│                 │                                                              │
│ optimized_plan  │  DF ["a", "b"]; PROJECT */2 COLUMNS; SELECTION: "None"       │
│                 │                                                              │
╰─────────────────┴──────────────────────────────────────────────────────────────╯
```

As you can see, the resulting dataframe is not yet evaluated, it stays as a set of instructions that can be done on the data. If you were to collect that dataframe you would get the next result

```
❯ $a | dfr collect
```

```
╭───┬───┬───╮
│ # │ a │ b │
├───┼───┼───┤
│ 0 │ 1 │ a │
│ 1 │ 2 │ b │
│ 2 │ 3 │ c │
│ 3 │ 4 │ d │
╰───┴───┴───╯
```

as you can see, the collect command executes the plan and creates a nushell table for you.

All dataframes operations should work with eager or lazy dataframes. They are converted in the background for compatibility. However, to take advantage of lazy operations if is recommended to only use lazy operations with lazy dataframes.

To find all lazy dataframe operations you can use

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

With your lazy frame defined we can start chaining operations on it. For example this

```
❯ $a |
• dfr reverse |
• dfr with-column [
•  ((dfr col a) * 2 | dfr as double_a)
•  ((dfr col a) / 2 | dfr as half_a)
• ] | dfr collect
```

```
╭───┬───┬───┬──────────┬────────╮
│ # │ a │ b │ double_a │ half_a │
├───┼───┼───┼──────────┼────────┤
│ 0 │ 4 │ d │        8 │      2 │
│ 1 │ 3 │ c │        6 │      1 │
│ 2 │ 2 │ b │        4 │      1 │
│ 3 │ 1 │ a │        2 │      0 │
╰───┴───┴───┴──────────┴────────╯
```

You can use the line buffer editor to format your queries (`ctr + o`) easily

This query uses the lazy reverse command to invert the dataframe and the `dfr with-column` command to create new two columns using `expressions`. An `expression` is used to define an operation that is executed on the lazy frame. When put together they create the whole set of instructions used by the lazy commands to query the data. To list all the commands that generate an expression you can use

```
scope commands | where category =~ expression
```

In our previous example, we use the `dfr col` command to indicate that column `a` will be multiplied by 2 and then it will be aliased to the name `double_a`. In some cases the use of the `dfr col` command can be inferred. For example, using the `dfr select` command we can use only a string

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

or the `dfr col` command

```
> $a | dfr select (dfr col a) | dfr collect
```

Let's try something more complicated and create aggregations from a lazy dataframe

```
❯ let a = ( [[name value]; [one 1] [two 2] [one 1] [two 3]] | dfr into-lazy )
❯ $a |
• dfr group-by name |
• dfr agg [
•  (dfr col value | dfr sum | dfr as sum)
•  (dfr col value | dfr mean | dfr as mean)
• ] | dfr collect
```

#	name	sum	mean
0	two	5	2.50
1	one	2	1.00

And we could join on a lazy dataframe that hasn't being collected. Let's join the resulting group by to the original lazy frame

```
❯ let a = ( [[name value]; [one 1] [two 2] [one 1] [two 3]] | dfr into-lazy )
❯ let group = ($a
• | dfr group-by name
• | dfr agg [
•   (dfr col value | dfr sum | dfr as sum)
•   (dfr col value | dfr mean | dfr as mean)
• ])
❯ $a | dfr join $group name name | dfr collect
```

#	name	value	sum	mean
0	one	1	2	1.00
1	two	2	5	2.50
2	one	1	2	1.00
3	two	3	5	2.50

As you can see lazy frames are a powerful construct that will let you query data using a flexible syntax, resulting in blazing fast results.

Dataframe commands

So far we have seen quite a few operations that can be done using `DataFrames` 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.

This list may be outdated. To get the up-to-date command list, see Dataframe Lazyframe and Dataframe Or Lazyframe command categories.

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 dataframe data types	
filter-with	DataFrame	Filters dataframe using a mask as reference	
first	DataFrame	Creates new dataframe with first rows	first
get	DataFrame	Creates dataframe with the selected columns	get
group-by	DataFrame	Creates a groupby object that can be used for other aggregations	group-by
is-duplicated	Series	Creates mask indicating duplicated values	
is-in	Series	Checks if elements from a series are contained in right series	in
is-not-null	Series	Creates mask where value is not null	
is-null	Series	Creates mask where value is null	`<column_name> == null`
is-unique	Series	Creates mask indicating unique values	
join	DataFrame	Joins a dataframe using columns as reference	
last	DataFrame	Creates new dataframe with last rows	last
ls-df		Lists stored dataframes	
melt	DataFrame	Unpivot a DataFrame from wide to long format	
[illegible] idx	[illegible]	[illegible]	[illegible] `<column_name> <value> | upsert <column_name> { <new_value> }`

Future of Dataframes

We hope that by the end of this page you have a solid grasp of how to use the dataframe commands. As you can see they offer powerful operations that can help you process data faster and natively.

However, the future of these dataframes is still very experimental. New commands and tools that take advantage of these commands will be added as they mature.

Keep visiting this book in order to check the new things happening to dataframes and how they can help you process data faster and efficiently.

Metadata

In using Nu, you may have come across times where you felt like there was something extra going on behind the scenes. For example, let's say that you try to open a file that Nu supports only to forget and try to convert again:

```
> open Cargo.toml | from toml
error: Expected a string from pipeline
- shell:1:18
1 | open Cargo.toml | from toml
  |                   ^^^^^^^^^ requires string input
- shell:1:5
1 | open Cargo.toml | from toml
  |      ---------- object originates from here
```

The error message tells us not only that what we gave `from toml` 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:

```
> metadata (open Cargo.toml)
╭──────┬────────────────────╮
│ span │ {record 2 fields} │
╰──────┴────────────────────╯
```

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

```
> metadata (open Cargo.toml) | get span
╭───────┬────────╮
│ start │ 212970 │
│ end   │ 212987 │
╰───────┴────────╯
```

The span "start" and "end" here refer to where the underline will be in the line. If you count over 5, and then count up to 15, you'll see it lines up with the "Cargo.toml" filename. This is how the error we saw earlier knew what to underline.

Creating your own errors

Using the metadata information, you can create your own custom error messages. Error messages are built of multiple parts:

- The title of the error
- The label of error message, which includes both the text of the label and the span to underline

You can use the `error make` 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", span: $span } }
```

Together with your custom command, it might look like this:

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

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

```
> 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. Nu plugins can use either JSON or MSGPACK as their communication encoding.

Downloading and installing a plugin

Please note that plugin installation methods are still under heavy development and that the following workflow will be refined before the release of 1.0. The nupm official package manager should simplify installation in the future when it becomes ready for general use.

To install a plugin on your system, you first need to make sure that the plugin uses the same version of Nu as your system.

```
> version
```

Find plugins that have the exact same Nushell version either on crates.io, online git repositories or `awesome-nu`. You can find which version the plugin uses by checking the Cargo.toml file.

To install a plugin by name from crates.io, run:

```
> cargo install plugin_name
```

If you chose to download the git repository instead, run this when inside the cloned repository:

```
> cargo install --path .
```

This will create a binary file that can be used to register the plugin.

Keep in mind that when installing using crates.io, the binary can be saved in different locations depending on how your system is set up. A typical location is in the users's home directory under .cargo/bin.

Registering a plugin

To enable an installed 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 waits for the plugin to tell Nu which communication encoding it should use
2. Nu sends it a "Signature" message over stdin
3. The plugin responds via stdout with a message containing its signature (name, description, arguments, flags, and more)
4. Nu saves the plugin signature in the file at `$nu.plugin-path`, so registration is persisted across multiple launches

Once registered, the plugin is available as part of your set of commands:

```
> help commands | where command_type == "plugin"
```

Updating a plugin

When updating a plugin, it is important to run `register` again just as above to load the new signatures from the plugin and allow Nu to rewrite them to the plugin file (`$nu.plugin-path`).

Managing plugins

To view the list of plugins you have installed:

```
> plugin list
╭───┬─────────┬────────────┬─────────┬───────────────────────┬───────┬─────────────────────────────
│ # │  name   │ is_running │   pid   │       filename        │ shell │
commands               │
├───┼─────────┼────────────┼─────────┼───────────────────────┼───────┼─────────────────────────────
│ 0 │ gstat   │ true       │ 1389890 │ .../nu_plugin_gstat   │       │ ╭───┬───────╮
│
│   │         │            │         │                       │       │ │ 0 │ gstat │
│
│   │         │            │         │                       │       │ ╰───┴───────╯
│
│ 1 │ inc     │ false      │         │ .../nu_plugin_inc     │       │ ╭───┬─────╮
│
│   │         │            │         │                       │       │ │ 0 │ inc │
│
│   │         │            │         │                       │       │ ╰───┴─────╯
│
│ 2 │ example │ false      │         │ .../nu_plugin_example │       │
```

```
┌───┬────────────────────────┐ │
│   │         │           │        │                       │       │ │ 0 │ nu-
example-1          │ │
│   │         │           │        │                       │       │ │ 1 │ nu-
example-2          │ │
│   │         │           │        │                       │       │ │ 2 │ nu-
example-3          │ │
│   │         │           │        │                       │       │ │ 3 │ nu-
example-config     │ │
│   │         │           │        │                       │       │ │ 4 │ nu-
example-disable-gc │ │
│   │         │           │        │                       │       │
└───┴────────────────────────┘ │
└───┴─────────┴───────────┴────────┴───────────────────────┴───────┴──────────────────
```

Plugins stay running while they are in use, and are automatically stopped by default after a period of time of inactivity. This behavior is managed by the plugin garbage collector. To manually stop a plugin:

```
> plugin stop gstat
```

If we check `plugin list` again, we can see that it is no longer running:

```
> plugin list | where name == gstat | select name is_running
```

#	name	is_running
0	gstat	false

Plugin garbage collector

Nu comes with a plugin garbage collector, which automatically stops plugins that are not actively in use after a period of time (by default, 10 seconds). This behavior is fully configurable:

```
$env.config.plugin_gc = {
    # Settings for plugins not otherwise specified:
    default: {
        enabled: true # set to false to never automatically stop plugins
        stop_after: 10sec # how long to wait after the plugin is inactive before
stopping it
    }
    # Settings for specific plugins, by plugin name
    # (i.e. what you see in `plugin list`):
    plugins: {
        gstat: {
            stop_after: 1min
        }
        inc: {
            stop_after: 0sec # stop as soon as possible
        }
        example: {
            enabled: false # never stop automatically
        }
    }
}
```

For more information on exactly under what circumstances a plugin is considered to be active, see the relevant section in the contributor book.

Removing a plugin
To remove a plugin, edit the `$nu.plugin-path` file and remove all of the `register` commands referencing the plugin you want to remove, including the signature argument.

Examples
Nu's main repo contains example plugins that are useful for learning how the plugin protocol works:

- Rust
- Python

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 is a great place to ask questions!

More details
The plugin chapter in the contributor book offers more details on the intricacies of how plugins work from a software developer point of view.

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

Signature
```
> explore --head --index --reverse --peek
```

Parameters
- `--head {boolean}`: turn off column headers
- `--index`: show row indexes (by default it's not showed)
- `--reverse`: start from the last row
- `--peek`: returns a last used value, so it can be used in next pipelines

Get Started
```
ls | explore -i
```

So the main point of `explore` is `:table` (Which you see on the above screenshot).

You can interact with it via `<Left>`, `<Right>`, `<Up>`, `<Down>` *arrow keys*.

You can inspect a underlying values by entering into cursor mode. You can press either `<i>` or `<Enter>` to do so. Then using *arrow keys* you can choose a necessary cell. And you'll be able to see it's underlying structure.

You can obtain more information about the various aspects of it by `:help`.

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`.