

The roc-ansi manual

THE ROC-ANSI CONTRIBUTORS

Version main

Table of Contents

1. What roc-ansi is	2
1.1. What it covers	2
1.2. What it looks like	2
1.3. What it does not do	4
1.4. Compatibility	4
1.5. Where to ask and report	4
2. Get started	5
2.1. Add the package	5
2.2. Run it	5
2.3. Where to go next	6
2.4. API reference for every version	6
3. Learn from the examples	7
3.1. Which copy to use	7
3.2. What the output looks like	7
4. Contribute	9
4.1. Set up	9
4.2. Two compilers	9
4.3. Run the checks	10
4.4. Where tests live	10
4.5. Change an example	11
4.6. Search for inputs nobody thought of	11
4.7. Keep allocations from creeping up	11
4.8. Build this manual	12
5. Maintain and release	13
5.1. What each workflow shows	13
5.2. Update the tooling	13
5.3. Make a release	14
5.4. How the site is put together	15
6. Release notes	16
6.1. roc-ansi 0.13.2	16

roc-ansi lets a Roc program colour and style what it prints, move the cursor, read key presses and draw simple tables, by writing the ANSI escape sequences a terminal understands for you.

How to use this manual

Deciding whether to use roc-ansi

Start with [What roc-ansi is](#).

Using it for the first time

Follow [Get started](#).

Looking for code to adapt

Browse [Learn from the examples](#).

Changing roc-ansi itself

Read [Contribute](#).

Releasing or looking after the repository

Read [Maintain and release](#).

Upgrading

See [Release notes](#).

Chapter 1. What roc-ansi is

A terminal changes colour, moves its cursor and reports key presses through short byte sequences called ANSI escape codes. They are easy to get wrong and unpleasant to read:

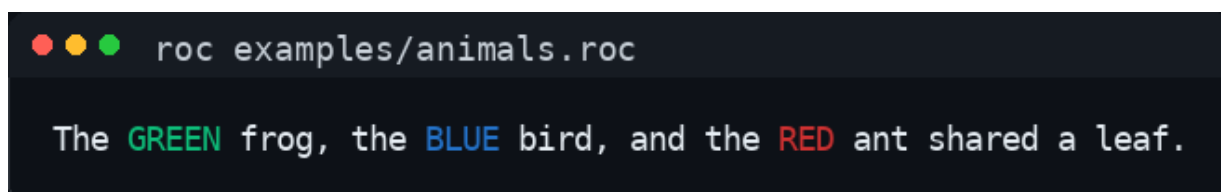
`\u001b[38;5;208m` means "orange text". `roc-ansi` is a Roc package that gives each of them a typed name, so a program says `ANSI.color("hello", { fg: C256(208), bg: Default })` and gets the right bytes.

It is a pure Roc package. It has no platform of its own and works wherever Roc does, with any platform that can write to a terminal.

1.1. What it covers

Module	What it is for
<code>ANSI</code>	The entry point: colour and style text, turn any escape into a string, parse raw key presses, and draw to a screen buffer
<code>Color</code> , <code>C16</code> , <code>C256</code> , <code>Rgb</code>	The four ways to name a colour: the terminal default, the 16 named colours, the 256-colour palette, and 24-bit RGB
<code>Style</code>	Bold, faint, italic, underline, strikethrough and inverted text
<code>Control</code>	Cursor movement, clearing the screen or a line, and scrolling
<code>Layout</code> , <code>Spacing</code>	Bordered grids and tables, padding and margins
<code>PieceTable</code>	A piece table, the structure a text editor keeps its buffer in

1.2. What it looks like



```

roc examples/colors.roc

Default color
ANSI 16 colors
Standard Red fg
Standard Green fg
Standard Blue fg
Bright Red bg
Bright Green bg
Bright Blue bg
{ fg: Bright Red, bg: Standard Black }
{ fg: Standard Green, bg: Standard Red }
ANSI 256 colors
{ fg: DarkGray, bg: Orange }
RGB colors, these are not supported in all terminals
{ fg: DarkTeal, bg: MintGreen }
{ fg: CoralPink, bg: RoyalBlue }
{ fg: ElectricPurple, bg: Tangerine }

```

```

roc examples/styles.roc

Bold On
Faint On
Italic On
Strikethrough On
Underline On
Invert On
Combination
This should not have any style

```

Each picture is the output of one of the programs in [Learn from the examples](#).

1.3. What it does not do

- It does not put the terminal into raw mode or read from standard input. Your platform does that; roc-ansi parses the bytes it hands you.
- It does not detect what a terminal supports. A program that must work everywhere should keep to the 16 named colours.

1.4. Compatibility

Roc has no stable release yet, so a roc-ansi release is built and tested with one nightly compiler, named in its release notes. The latest release is tested against the latest nightly every day, and a failure is fixed by a patch release. Use the latest release with a current [nightly](#).

1.5. Where to ask and report

- Bugs and requests: [GitHub issues](#).
- Vulnerabilities: privately, as described in [SECURITY.md](#).
- roc-ansi is open source under the [Universal Permissive License 1.0](#).

Chapter 2. Get started

This chapter takes you from nothing to a program that prints coloured text. You need a current Roc `nightly` installed as `roc`.

2.1. Add the package

A Roc application names its packages in its header, each by the URL of a release. Save this as `hello.roc`:

The latest release is 0.13.2

```
app [main!] {
  ansi: "https://github.com/lukewilliamboswell/roc-ansi/releases/download/0.13.2/7yixATpLmwqtNujvYk1sKvZHmgFBMm7B9y7amxVZAAn.tar.zst",
}

import ansi.ANSI

main! = |_args| {
  echo!(ANSI.color("hello", { fg: Standard(Green), bg: Default }))
  Ok({})
}
```

For release 0.13.2:

- [API reference](#)
- [Examples](#), each depending on this release
- [Release notes](#)

The URL ends in a hash of the package's contents. Roc checks what it downloads against that hash, so the URL can only ever mean one thing.

2.2. Run it

```
roc hello.roc
```

The word `hello` is printed in green. `ANSI.color` wrapped it in the escape codes that switch the colour

on, and a reset that switches it off again.

2.3. Where to go next

- [Learn from the examples](#) has a short program for each part of the package.
- The API reference linked above lists every function of every module.

2.4. API reference for every version

Each release keeps its own API reference, so the pages for the version you depend on never change under you.

- [main](#), which is not released yet
- [0.13.2](#)
- [0.13.1](#)
- [0.13.0](#)
- [0.12.0](#)
- [0.11.0](#)
- [0.10.0](#)
- [0.8.0](#)

Chapter 3. Learn from the examples

Each example is a short, complete program that shows one thing, with comments that walk through it. Read them top to bottom, or copy one as a starting point.

Example	What it shows
<code>animals.roc</code>	Colouring a few words inside a sentence with <code>ANSI.color</code>
<code>colors.roc</code>	Every way of naming a colour, from the most portable to the least
<code>styles.roc</code>	Bold, faint, italic and the other text styles, and how to switch them off
<code>menu.roc</code>	Combining colours and styles to draw a menu with one highlighted entry
<code>piece-table.roc</code>	Editing text with a piece table

3.1. Which copy to use

The examples exist in two places, and they are not the same.

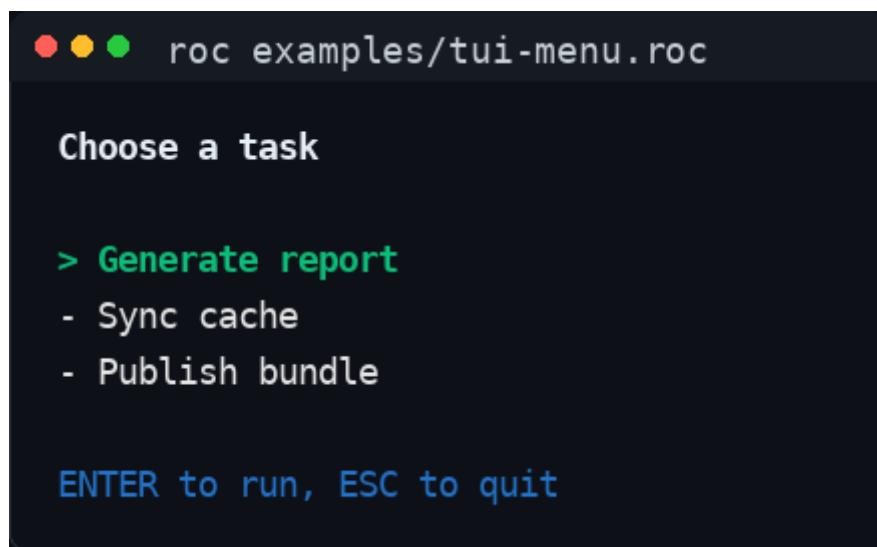
The examples for a release

Each release attaches an archive, `roc-ansi-examples-<version>.zip`, linked from [Add the package](#). Its examples depend on that release's package URL and never change. Use these to start a project.

The examples in the repository

These depend on the package beside them, `ansi: "../package/main.roc"`, so they always match the unreleased code on `main`. They may use something no release has yet. Use these when you are changing `roc-ansi`.

3.2. What the output looks like



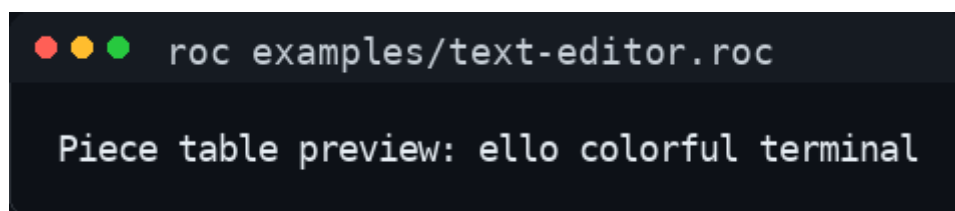
```
roc examples/tui-menu.roc

Choose a task

> Generate report
- Sync cache
- Publish bundle

ENTER to run, ESC to quit
```

Figure 1. menu.roc



```
roc examples/text-editor.roc

Piece table preview: ello colorful terminal
```

Figure 2. piece-table.roc

The archive also holds `spec.json`, which records exactly what each example printed when the release was made.

Chapter 4. Contribute

This chapter is for someone changing roc-ansi: what to install, how to run the checks a pull request must pass, and where each kind of test lives.

Use [issues](#) for bugs and design discussion, and pull requests for changes. Say what behaviour changes and how you checked it. A bug fix or a new behaviour comes with a test.

4.1. Set up

You need three things:

- [Nix](#), with flakes enabled.
- The `blueprint` command from [roc-blueprint](#), at the release named by the platform URL at the top of `Blueprint.roc`. Download the binary for your system from that release and check it against the release’s `blueprint.sha256`, or run `nix profile install github:lukewilliamboswell/roc-blueprint/<version>`.
- A current Roc [nightly](#) on your `PATH` as `roc`.

4.2. Two compilers

Command	Chosen by	Used for
<code>roc</code>	You: the latest nightly	Checking, testing, bundling and running the package and its examples
<code>roc-stable</code>	The release named in <code>Blueprint.roc</code>	Running the scripts under <code>scripts/</code>

The package is always tested with the newest compiler, because that is what its users install. `roc-stable` exists so the repository’s own tooling keeps working on the day a nightly breaks something. It never checks or builds the package, and it only exists inside `blueprint run` and `blueprint shell`.

Nothing pins a compiler for the package or the examples. Each script names `roc-stable`’s release in its ``roc` header field, so that running one with a different compiler says why it misbehaves.

4.3. Run the checks

```
blueprint run check    # everything a pull request must pass
blueprint shell        # or enter the environment and run scripts directly
blueprint tasks        # list every task
```

`blueprint run check` runs `scripts/check_all.roc`, which:

1. Fails if a compiler pin appears outside `scripts/`, or a script's pin differs from `Blueprint.roc`.
2. Tests, checks and formats the scripts with `roc-stable`.
3. Formats, checks and tests the package with `roc`.
4. Builds every example against the working tree, runs it, and compares its output with `tests/spec.json`.
5. Builds each fuzz target and replays its committed corpus.
6. Counts allocations and compares them with `tests/allocations.json`.
7. Bundles the package, builds the examples archive, and runs the archived examples, unchanged, against that bundle.
8. Builds the API documentation and its archive.

A pull request can fail because a new nightly arrived, with no fault in the change. Compare with the latest run of the nightly workflow and with `main` before treating a failure as yours.

4.4. Where tests live

Kind	Where	How to add one
Unit tests	<code>expect</code> beside the code in <code>package/</code>	Write an <code>expect</code> in the module. <code>roc test package/main.roc</code> runs them all.
Example output	<code>tests/spec.json</code>	Add a case to the example's entry, then record its output; see Change an example .
Properties	<code>fuzz/<name>/main.roc</code>	A <code>roc-fuzz</code> target with a small <code>corpus/</code> ; see Search for inputs nobody thought of .
Allocations	<code>tests/allocations.json</code>	Add a case and its ceiling; see Keep allocations from creeping up .
Script logic	<code>expect</code> in the modules under <code>scripts/src/</code>	Keep effects out of the module, and add it to the list in <code>check_scripts!</code> .

4.5. Change an example

An example's exact output is recorded, so a change to what it prints must be recorded too:

```
blueprint shell
scripts/test_examples.roc -- --update
```

That rewrites `tests/spec.json` from what the examples print now. Review the diff like any other change. An example that cannot run for a reason outside this repository is skipped in the specification with a `reason` and an `issue` link; a skip without both is rejected.

Examples are for users of the package. Keep each one short, about one thing, and commented for someone reading it with no other context. A test belongs in `package/` or `tests/`, not in `examples/`.

4.6. Search for inputs nobody thought of

`fuzz/piece-table` states a rule that must hold for every document and index, and `roc-fuzz` searches for an input that breaks it. Every pull request only builds the target and replays its committed corpus. To search:

```
roc build --fuzz fuzz/piece-table/main.roc --output=piece-table
./piece-table run fuzz/piece-table/corpus --time=60
```

When it finds a failure, fix the bug and keep the minimised input as an ordinary `expect` in the package.

4.7. Keep allocations from creeping up

A change can make an in-place update copy a whole list on every call, and every test still passes. `tests/allocations.json` records the most allocations each listed case may make, and the check fails above that.

- A count below its ceiling passes, and the check says what to lower the ceiling to. Lower it in the same pull request, so the gain cannot be lost.
- Raising a ceiling needs a reason in the pull request. It is not a way to make a red build green.

The counts come from `tests/allocations/main.roc`, built on the `roc-fuzz` platform because its host counts allocations. Nothing is fuzzed there.

4.8. Build this manual

The manual is the AsciiDoc under `docs/`, one chapter per file, rendered by the shared [build-docs](#) action. Every pull request builds it, and the site can be downloaded from the `Documentation` workflow run. To build it yourself you need Docker and a checkout of `roc-automation`:

```
blueprint run install-snippet
python3 ../roc-automation/actions/build-docs/build_docs.py --workspace . --pdf-filename roc-ansi.pdf
blueprint run serve
```

The first command writes the parts of the manual that depend on what has been released into `docs/generated/`, which needs `gh` to be authenticated. The second renders the site and the PDF into `.docs-out/site`. The last adds the API reference for `main` and serves the site on a free port, or the one named by `SITE_PORT`; with `FETCH_RELEASE_DOCS=1` it also restores the API reference of every release.

Chapter 5. Maintain and release

This chapter is for a maintainer: what each workflow shows, how the tooling is updated, and how a release is made. The repository follows the [roc-automation manual](#), which gives the reasons behind each practice.

5.1. What each workflow shows

Workflow	Runs on	What it shows
<code>ci.yml</code>	Pull requests, pushes to <code>main</code>	The working tree passes <code>blueprint run check</code> on the latest nightly
<code>release.yml</code>	Manual dispatch; pull requests as a dry run	The exact bundle, examples archive and documentation proposed for publication work
<code>nightly.yml</code>	Daily	The latest release's examples archive still works on the latest nightly
<code>docs.yml</code>	Pull requests, pushes to <code>main</code>	The manual and site build; <code>main</code> deploys them

The package is pure Roc, so every job runs on Linux only.

`nightly.yml` keeps one tracking issue: opened on the first failure, updated by later failures, and closed by the next pass. A failure there means the latest release needs a patch release. A published archive is never edited. A run that fails before any example ran, such as a failed download, opens no issue.

To re-run a past combination, dispatch `nightly.yml` with a `nightly_tag`, a `release_tag`, or both. Left empty, they mean the latest nightly and the latest full release.

5.2. Update the tooling

Nothing updates the tooling automatically. Each of these is a pull request.

Refresh the lock

Run `blueprint update` and commit `Blueprint.lock`.

Change the tooling compiler

Change the tag in `Command("roc-stable", ...)` in `Blueprint.roc` and the `roc` field of every script to the same nightly. The check fails until they agree. Run `blueprint update roc` if the lock predates that nightly, and fix any script the new compiler breaks.

Upgrade Blueprint

Change the platform URL at the top of `Blueprint.roc`, and the `version` and `sha256-*` inputs the workflows pass to `setup-blueprint`, from the same `roc-blueprint` release.

Update a script dependency

Change its URL in every script header and under `RocPackages` in `Blueprint.roc`, then run `blueprint update`.

5.3. Make a release

1. Choose the version. `roc bump` compares the package with the latest release and says how far the version must move:

```
roc bump --old <latest release bundle URL> package/main.roc
```

2. Write the notes in `docs/releases/<version>.adoc`, add them to the top of `docs/releases.adoc`, and merge that pull request. Write them for someone deciding whether and how to upgrade. Use `{package-url}` wherever the release's bundle URL belongs.
3. Dispatch the **Release** workflow from `main` with that version.

The workflow then:

1. Rejects a version that already exists, one with no notes, and one that `roc bump --expect` says understates the change.
2. Runs `blueprint run check`.
3. Builds the bundle, the examples archive, the documentation archive and this manual as a PDF, and renders the notes with the real bundle URL and the versions of `roc` and `roc-stable` it used. That record replaces a header pin as the statement of what the release was built and tested with.
4. Tests the examples archive, unchanged, against the bundle.
5. Publishes. A separate job, which checks out nothing and runs no code from the repository, creates a draft release at the tested commit, uploads every asset, downloads each back and

compares it with the tested file, and only then publishes.

6. Rebuilds and deploys the site, unless the version is a release candidate.

A release holds:

Asset	Purpose
<code><hash>.tar.zst</code>	The bundle users depend on. Its name is its content hash.
<code>roc-ansi-examples-<version>.zip</code>	The examples, depending on this bundle, with the output they printed
<code>roc-ansi-docs-<version>.zip</code>	The API reference for this version, which the site restores
<code>roc-ansi-<version>.pdf</code>	This manual

No follow-up pull request is needed, because nothing on `main` names its own release. A published release is never changed: if one is wrong, publish a new version.

On a pull request the workflow builds and tests the same things under the version `0.0.1-dry-run`, renders the newest notes, and publishes nothing.

A release candidate, such as `0.14.0-rc1`, goes through the same workflow. It is marked as a prerelease, is never the latest release, and needs its own notes file.

5.4. How the site is put together

The site is this manual, rendered by the shared `build-docs` action, with the API reference added beside it by `blueprint run site`:

- `main/` holds fresh API documentation for the working tree.
- Each release's API documentation sits under its version, such as `0.13.1/`, restored from the archive that release attached. Its address keeps working without its HTML being stored in Git, and generated documentation is never committed.

The manual leads with the latest release: its first program, its API reference and its examples archive. Those parts are written by `blueprint run install-snippet` before the manual is rendered, because a bundle URL holds a hash nobody can write by hand.

`docs.yml` builds the site on every pull request and deploys it from `main`. `release.yml` rebuilds and deploys it after publishing, since a release changes which one is the latest.

Chapter 6. Release notes

Each release has its notes in `docs/releases/`, newest first. Releases before 0.13.2 have theirs on the [releases page](#).

6.1. roc-ansi 0.13.2

This release changes nothing in the package's API. It is the first whose examples archive records what each example prints, so the daily compatibility check compares output as well as building and running.

6.1.1. Use this release

```
app [main!] {  
  ansi: "{package-url}",  
}
```

6.1.2. Examples

- `text-editor.roc` is now `piece-table.roc` and `tui-menu.roc` is now `menu.roc`. Each prints one frame, and its name now says so.
- Every example carries comments that walk through it.
- The archive `roc-ansi-examples-0.13.2.zip` holds `spec.json`, the exact output each example printed when this release was made.

6.1.3. Documentation

- The project now has a manual, published as a site and attached to each release as a PDF.
- The documentation archive is `roc-ansi-docs-0.13.2.zip`. Earlier releases attached a `.tar.gz`.