Documentation tests

`rustdoc` supports executing your documentation examples as tests. This makes sure that examples within your documentation are up to date and working.

The basic idea is this:

```
/// # Examples
///
/// ```
/// let x = 5;
/// ```
# fn f() {}
```

The triple backticks start and end code blocks. If this were in a file named `foo.rs`, running `rustdoc --test foo.rs` will extract this example, and then run it as a test.

Please note that by default, if no language is set for the block code, rustdoc assumes it is Rust code. So the following:

````
```rust
let x = 5;
```
````

is strictly equivalent to:

````
```
let x = 5;
```
````

There's some subtlety though! Read on for more details.

Passing or failing a doctest

Like regular unit tests, regular doctests are considered to "pass" if they compile and run without panicking. So if you want to demonstrate that some computation gives a certain result, the `assert!` family of macros works the same as other Rust code:

```
let foo = "foo";
assert_eq!(foo, "foo");
```

This way, if the computation ever returns something different, the code panics and the doctest fails.

Pre-processing examples

In the example above, you'll note something strange: there's no `main` function! Forcing you to write `main` for every example, no matter how small, adds friction

and clutters the output. So `rustdoc` processes your examples slightly before running them. Here's the full algorithm `rustdoc` uses to preprocess examples:

1. Some common `allow` attributes are inserted, including `unused_variables`, `unused_assignments`, `unused_mut`, `unused_attributes`, and `dead_code`. Small examples often trigger these lints.
2. Any attributes specified with `#![doc(test(attr(...)))]` are added.
3. Any leading `#![foo]` attributes are left intact as crate attributes.
4. If the example does not contain `extern crate`, and `#![doc(test(no_crate_inject))]` was not specified, then `extern crate <mycrate>;` is inserted (note the lack of `#[macro_use]`).
5. Finally, if the example does not contain `fn main`, the remainder of the text is wrapped in `fn main() { your_code }`.

For more about that caveat in rule 4, see "Documenting Macros" below.

Hiding portions of the example

Sometimes, you need some setup code, or other things that would distract from your example, but are important to make the tests work. Consider an example block that looks like this:

```
/// ```
/// /// Some documentation.
/// # fn foo() {} // this function will be hidden
/// println!("Hello, World!");
/// ```
# fn f() {}
```

It will render like this:

```
/// Some documentation.
# fn foo() {}
println!("Hello, World!");
```

Yes, that's right: you can add lines that start with `#`, and they will be hidden from the output, but will be used when compiling your code. You can use this to your advantage. In this case, documentation comments need to apply to some kind of function, so if I want to show you just a documentation comment, I need to add a little function definition below it. At the same time, it's only there to satisfy the compiler, so hiding it makes the example more clear. You can use this technique to explain longer examples in detail, while still preserving the testability of your documentation.

For example, imagine that we wanted to document this code:

```
let x = 5;
let y = 6;
println!("{}", x + y);
```

We might want the documentation to end up looking like this:

> First, we set `x` to five:
>
> ```
> let x = 5;
> # let y = 6;
> # println!("{}", x + y);
> ```
>
> Next, we set `y` to six:
>
> ```
> # let x = 5;
> let y = 6;
> # println!("{}", x + y);
> ```
>
> Finally, we print the sum of `x` and `y`:
>
> ```
> # let x = 5;
> # let y = 6;
> println!("{}", x + y);
> ```

To keep each code block testable, we want the whole program in each block, but we don't want the reader to see every line every time. Here's what we put in our source code:

````
First, we set `x` to five:

```
let x = 5;
# let y = 6;
# println!("{}", x + y);
```

Next, we set `y` to six:

```
# let x = 5;
let y = 6;
# println!("{}", x + y);
```

Finally, we print the sum of `x` and `y`:

```
# let x = 5;
# let y = 6;
println!("{}", x + y);
```
````

By repeating all parts of the example, you can ensure that your example still compiles, while only showing the parts that are relevant to that part of your

explanation.

The `#`-hiding of lines can be prevented by using two consecutive hashes `##`. This only needs to be done with the first `#` which would've otherwise caused hiding. If we have a string literal like the following, which has a line that starts with a `#`:

```
let s = "foo
## bar # baz";
```

We can document it by escaping the initial `#`:

```
/// let s = "foo
/// ## bar # baz";
```

Using ? in doc tests

When writing an example, it is rarely useful to include a complete error handling, as it would add significant amounts of boilerplate code. Instead, you may want the following:

```
/// ```
/// use std::io;
/// let mut input = String::new();
/// io::stdin().read_line(&mut input)?;
/// ```
# fn f() {}
```

The problem is that `?` returns a `Result<T, E>` and test functions don't return anything, so this will give a mismatched types error.

You can get around this limitation by manually adding a `main` that returns `Result<T, E>`, because `Result<T, E>` implements the `Termination` trait:

```
/// A doc test using ?
///
/// ```
/// use std::io;
///
/// fn main() -> io::Result<()> {
///     let mut input = String::new();
///     io::stdin().read_line(&mut input)?;
///     Ok(())
/// }
/// ```
# fn f() {}
```

Together with the `#` from the section above, you arrive at a solution that appears to the reader as the initial idea but works with doc tests:

```
/// ```
/// use std::io;
```

```
/// # fn main() -> io::Result<()> {
/// let mut input = String::new();
/// io::stdin().read_line(&mut input)?;
/// # Ok(())
/// # }
/// ```
# fn f() {}
```

As of version 1.34.0, one can also omit the `fn main()`, but you will have to disambiguate the error type:

```
/// ```
/// use std::io;
/// let mut input = String::new();
/// io::stdin().read_line(&mut input)?;
/// # Ok::<(), io::Error>(())
/// ```
# fn f() {}
```

This is an unfortunate consequence of the `?` operator adding an implicit conversion, so type inference fails because the type is not unique. Please note that you must write the `(())` in one sequence without intermediate whitespace so that `rustdoc` understands you want an implicit `Result`-returning function.

Showing warnings in doctests

You can show warnings in doctests by running `rustdoc --test --test-args=--show-output` (or, if you're using cargo, `cargo test --doc -- --show-output`). By default, this will still hide `unused` warnings, since so many examples use private functions; you can add `#![warn(unused)]` to the top of your example if you want to see unused variables or dead code warnings. You can also use `#![doc(test(attr(warn(unused))))]` in the crate root to enable warnings globally.

Documenting macros

Here's an example of documenting a macro:

```
/// Panic with a given message unless an expression evaluates to true.
///
/// # Examples
///
/// ```
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(1 + 1 == 2, "Math is broken.");
/// # }
/// ```
```

```
///
/// ```should_panic
/// # #[macro_use] extern crate foo;
/// # fn main() {
/// panic_unless!(true == false, "I'm broken.");
/// # }
/// ```
#[macro_export]
macro_rules! panic_unless {
    ($condition:expr, $($rest:expr),+) => ({ if ! $condition { panic!($($rest),+); } });
}
# fn main() {}
```

You'll note three things: we need to add our own `extern crate` line, so that we can add the `#[macro_use]` attribute. Second, we'll need to add our own `main()` as well (for reasons discussed above). Finally, a judicious use of `#` to comment out those two things, so they don't show up in the output.

Attributes

Code blocks can be annotated with attributes that help `rustdoc` do the right thing when testing your code:

The `ignore` attribute tells Rust to ignore your code. This is almost never what you want as it's the most generic. Instead, consider annotating it with `text` if it's not code or using `#`s to get a working example that only shows the part you care about.

```
/// ```ignore
/// fn foo() {
/// ```
# fn foo() {}
```

`should_panic` tells `rustdoc` that the code should compile correctly but panic during execution. If the code doesn't panic, the test will fail.

```
/// ```should_panic
/// assert!(false);
/// ```
# fn foo() {}
```

The `no_run` attribute will compile your code but not run it. This is important for examples such as "Here's how to retrieve a web page," which you would want to ensure compiles, but might be run in a test environment that has no network access. This attribute can also be used to demonstrate code snippets that can cause Undefined Behavior.

```
/// ```no_run
/// loop {
```

```
///     println!("Hello, world");
/// }
/// ```
# fn foo() {}
```

`compile_fail` tells `rustdoc` that the compilation should fail. If it compiles, then the test will fail. However, please note that code failing with the current Rust release may work in a future release, as new features are added.

```
/// ```compile_fail
/// let x = 5;
/// x += 2; // shouldn't compile!
/// ```
# fn foo() {}
```

`edition2015`, `edition2018` and `edition2021` tell `rustdoc` that the code sample should be compiled using the respective edition of Rust.

```
/// Only runs on the 2018 edition.
///
/// ```edition2018
/// let result: Result<i32, ParseIntError> = try {
///     "1".parse::<i32>()?
///         + "2".parse::<i32>()?
///         + "3".parse::<i32>()?
/// };
/// ```
# fn foo() {}
```

Syntax reference

The *exact* syntax for code blocks, including the edge cases, can be found in the Fenced Code Blocks section of the CommonMark specification.

Rustdoc also accepts *indented* code blocks as an alternative to fenced code blocks: instead of surrounding your code with three backticks, you can indent each line by four or more spaces.

```
    let foo = "foo";
    assert_eq!(foo, "foo");
```

These, too, are documented in the CommonMark specification, in the Indented Code Blocks section.

However, it's preferable to use fenced code blocks over indented code blocks. Not only are fenced code blocks considered more idiomatic for Rust code, but there is no way to use attributes such as `ignore` or `should_panic` with indented code blocks.

Include items only when collecting doctests

Rustdoc's documentation tests can do some things that regular unit tests can't, so it can sometimes be useful to extend your doctests with samples that wouldn't otherwise need to be in documentation. To this end, Rustdoc allows you to have certain items only appear when it's collecting doctests, so you can utilize doctest functionality without forcing the test to appear in docs, or to find an arbitrary private item to include it on.

When compiling a crate for use in doctests (with `--test` option), `rustdoc` will set `#[cfg(doctest)]`. Note that they will still link against only the public items of your crate; if you need to test private items, you need to write a unit test.

In this example, we're adding doctests that we know won't compile, to verify that our struct can only take in valid data:

```
/// We have a struct here. Remember it doesn't accept negative numbers!
pub struct MyStruct(pub usize);

/// ```compile_fail
/// let x = my_crate::MyStruct(-5);
/// ```
#[cfg(doctest)]
pub struct MyStructOnlyTakesUsize;
```

Note that the struct `MyStructOnlyTakesUsize` here isn't actually part of your public crate API. The use of `#[cfg(doctest)]` makes sure that this struct only exists while `rustdoc` is collecting doctests. This means that its doctest is executed when `--test` is passed to rustdoc, but is hidden from the public documentation.

Another possible use of `#[cfg(doctest)]` is to test doctests that are included in your README file without including it in your main documentation. For example, you could write this into your `lib.rs` to test your README as part of your doctests:

```
#[doc = include_str!("../README.md")]
#[cfg(doctest)]
pub struct ReadmeDoctests;
```

This will include your README as documentation on the hidden struct `ReadmeDoctests`, which will then be tested alongside the rest of your doctests.