`async fn`s are not yet supported in traits in Rust.

Erroneous code example:

```
trait T {
    // Neither case is currently supported.
    async fn foo() {}
    async fn bar(&self) {}
}
```

`async fn`s return an `impl Future`, making the following two examples equivalent:

```
edition2018,ignore (example-of-desugaring-equivalence) async fn
foo() -> User {     unimplemented!() } // The async fn above gets
desugared as follows: fn foo(&self) -> impl Future<Output = User>
+ '_ {     unimplemented!() }
```

But when it comes to supporting this in traits, there are a few implementation issues. One of them is returning `impl Trait` in traits is not supported, as it would require Generic Associated Types to be supported:

“‘edition2018,ignore (example-of-desugaring-equivalence) impl MyDatabase { async fn get_user(&self) -> User { unimplemented!() } }

impl MyDatabase { fn get_user(&self) -> impl Future<Output = User> + ’_ { unimplemented!() } } “‘

Until these issues are resolved, you can use the `async-trait` crate, allowing you to use `async fn` in traits by desugaring to “boxed futures” (`Pin<Box<dyn Future + Send + 'async>>`).

Note that using these trait methods will result in a heap allocation per-function-call. This is not a significant cost for the vast majority of applications, but should be considered when deciding whether to use this functionality in the public API of a low-level function that is expected to be called millions of times a second.

You might be interested in visiting the async book for further information.