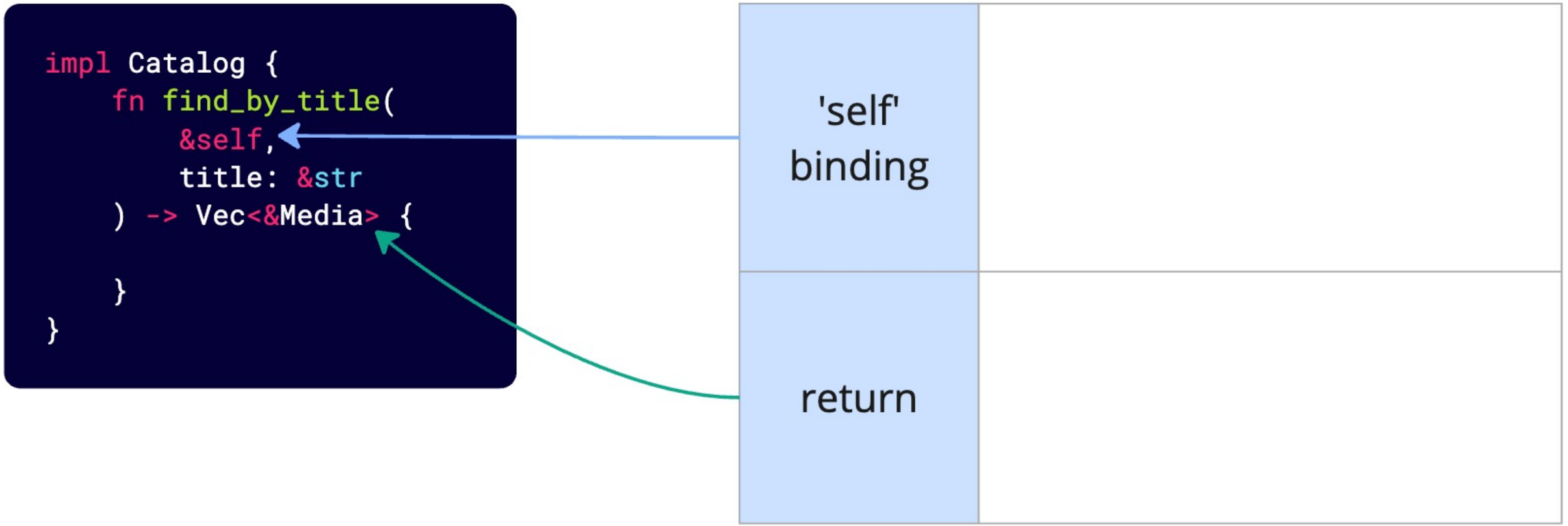
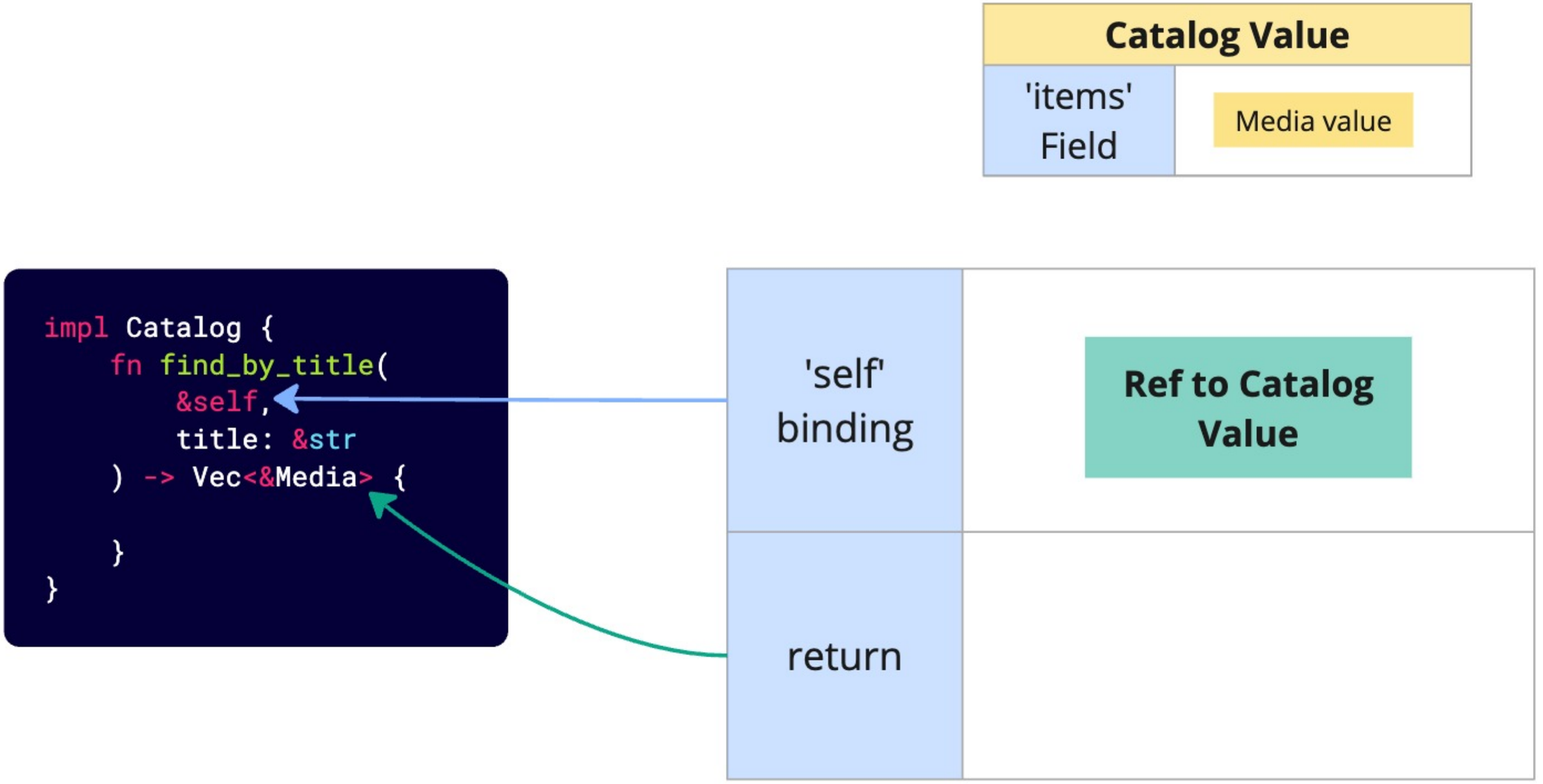
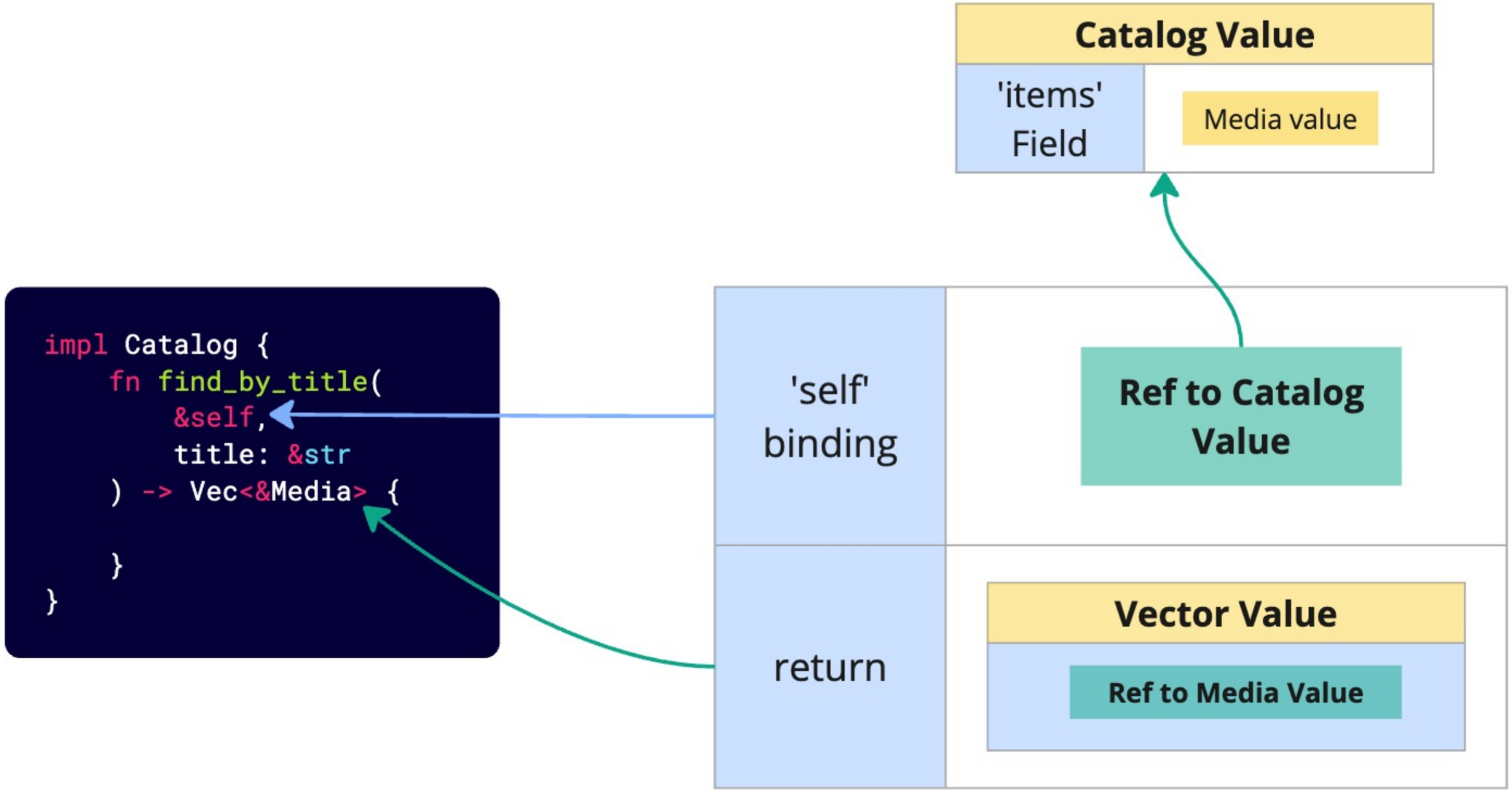
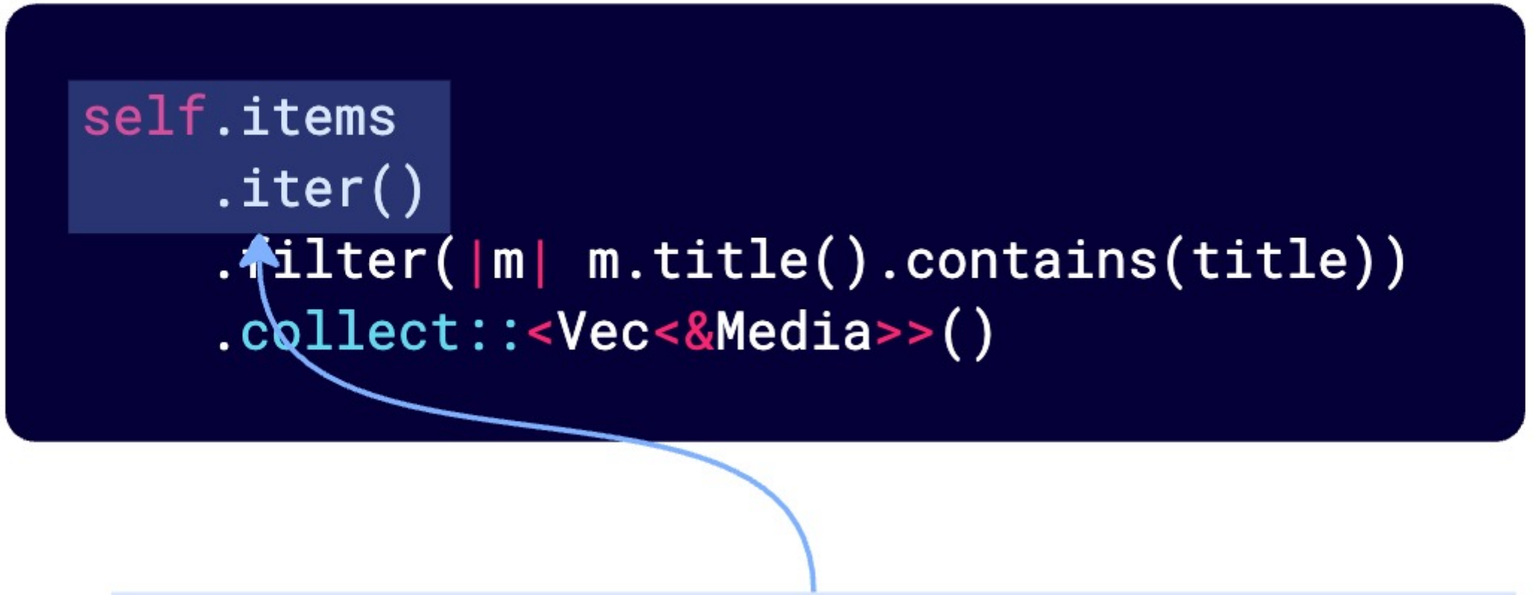

impl Catalog {
fn find_by_title(
&self,
title: &str
) -> Vec<&Media> {
}
}
'self'
binding
return


Catalog Value
'items' Field
Media value
impl Catalog {
fn find_by_title(
&self,
title: &str
) -> Vec<&Media> {
}
}
'self' binding
Ref to Catalog Value
return

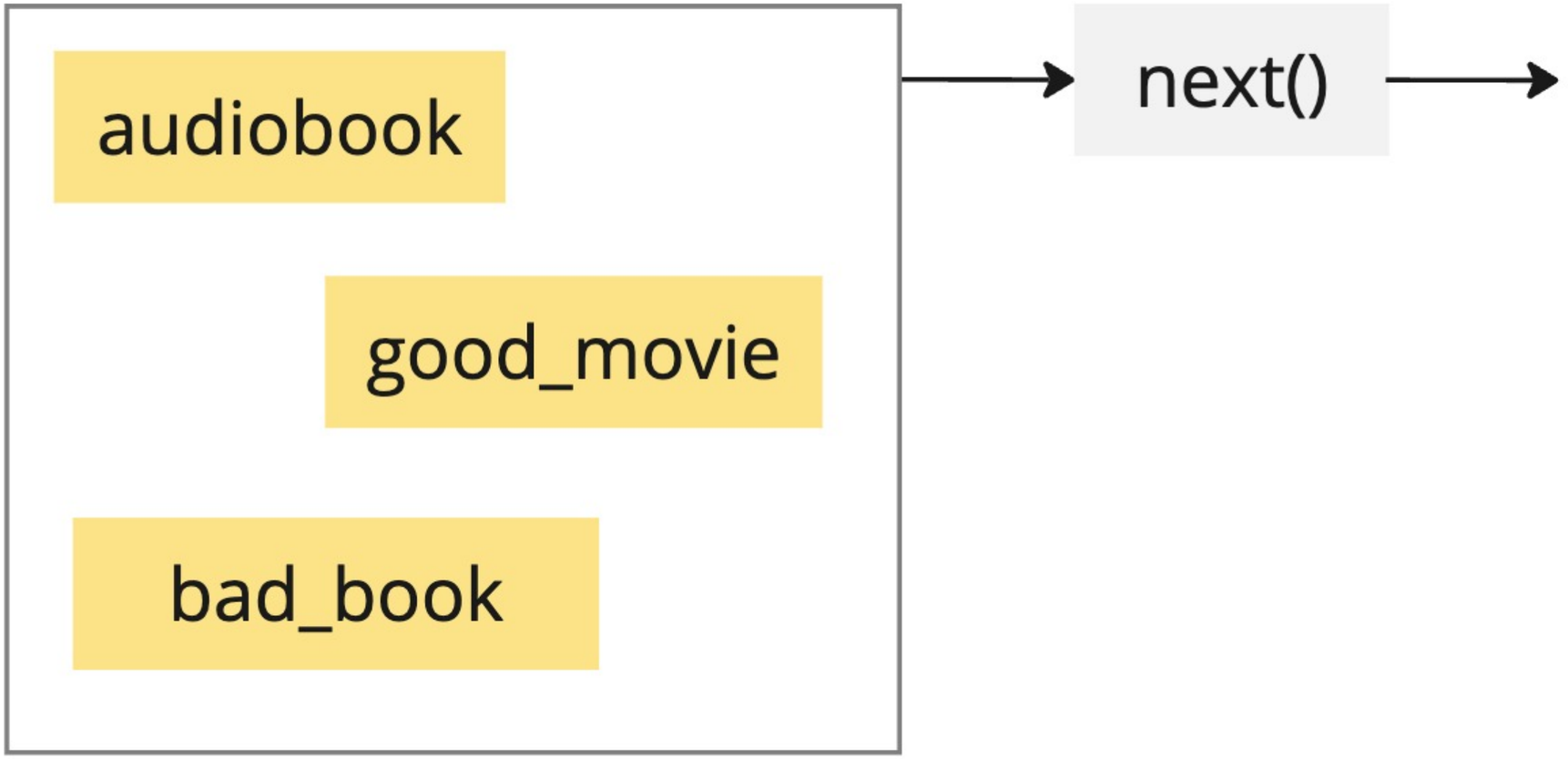

Catalog Value
'items' Field
Media value
impl Catalog {
fn find_by_title(
&self,
title: &str
) -> Vec<&Media> {
}
}
'self' binding
Ref to Catalog Value
return
Vector Value
Ref to Media Value



Gives us an ***iterator***

Iterators are the #1 tool we have for working with collections of data

Iterator

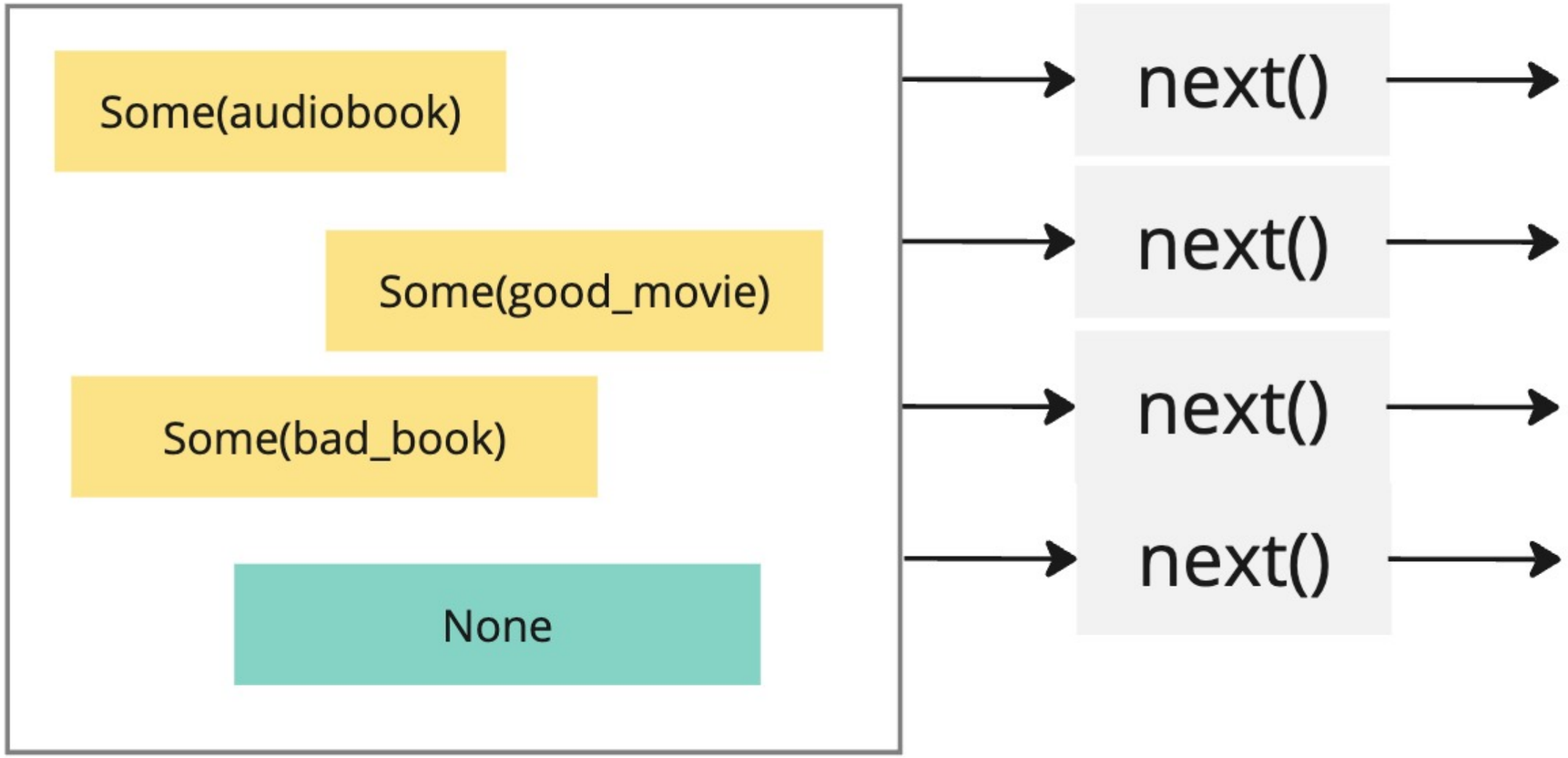


Option, Some, None

Rust doesn't have the concept of 'null' or 'nil'

Still have to deal with 'no value' somehow...

Iterator


Some(audiobook)
Some(good_movie)
Some(bad_book)
None
next()
next()
next()
next()

```rust
enum Media {
    Book { title: String, author: String },
    Movie { title: String, director: String },
    Audiobook { title: String },
}

impl Media {
    fn description(&self) -> String {
        match self {
             Media::Book { title, author } => {
                format!("Book: {} {}", title, author)
            }
            Media::Movie { title, director } => {
                format!("Movie: {} {}", title, director)
            }
            Media::Audiobook { title } => {
                format!("Audiobook: {}", title)
            }
        }
    }
}
```

Enums define some values that are similar but distinctly different

To work with enum values, we have to use a 'match' or pattern matching ('if let')

Working with the values is *exhaustive* - we have to handle every case

```rust
enum Media {
    Book(String),
    Movie,
    Audiobook,
}

fn main() {
    let book = Media::Book(String::from("Bad Book"));
    let movie = Media::Movie;
    let audiobook = Media::Audiobook;

}
```

Enums don't have to have fields

Can be single values, or no value at all

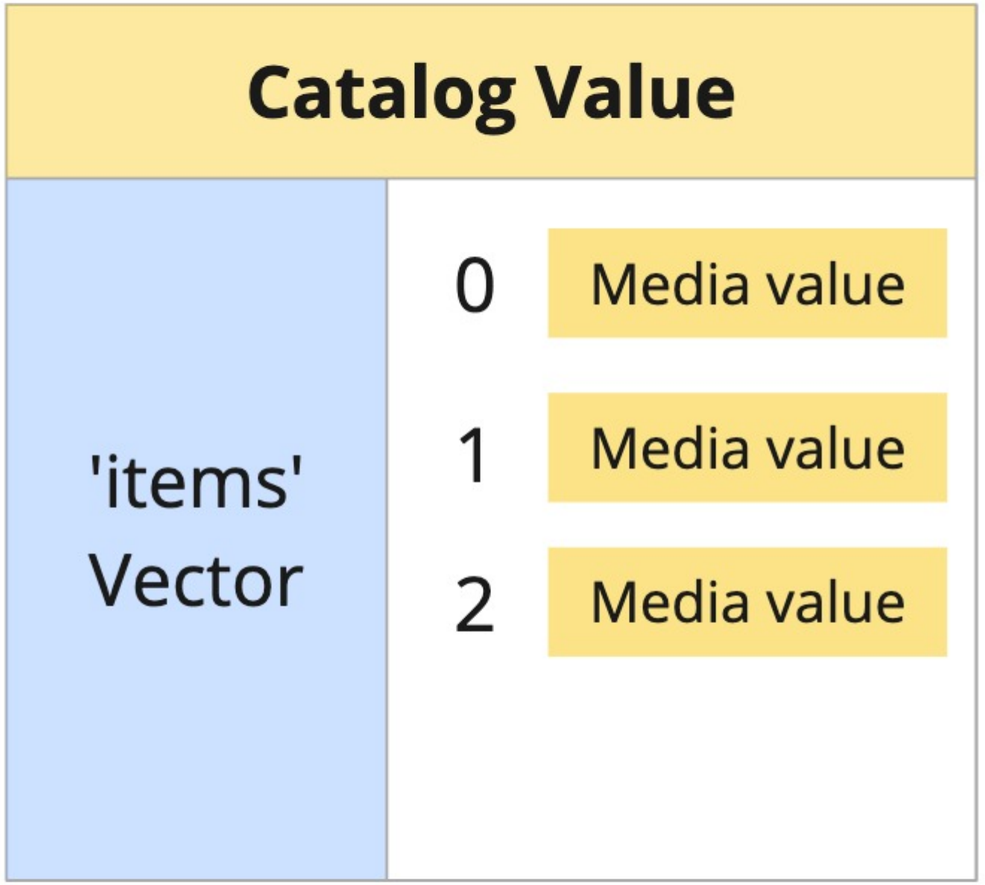
```
enum Option<T> {
    Some(T),
    None
}
```

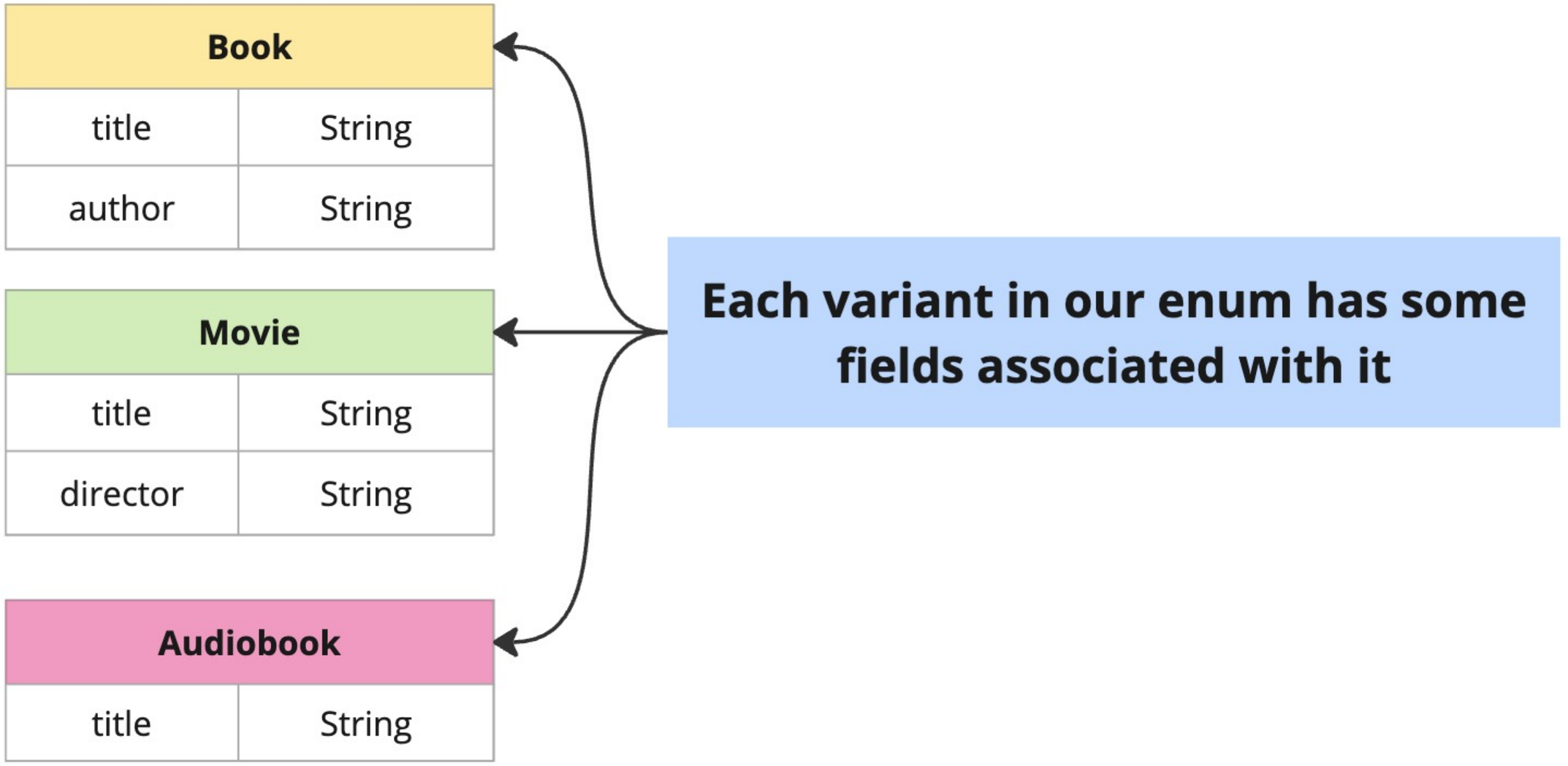
'Option' is a built-in enum

Two possible values - 'Some' indicates a value exists, 'None' indicates nothing

Matching must be exhaustive

Matching must be exhaustive


Catalog Value
'items'
Vector
0
Media value
1
Media value
2
Media value


Book
title
String
author
String
Movie
title
String
director
String
Audiobook
title
String
Each variant in our enum has some fields associated with it

Book	
title	String
author	String

Movie	
title	String
director	String

Audiobook	
title	String

Podcast
u32 *to represent episode #*

Placeholder

Variants can have unlabeled fields

```rust
fn get_element(&self, index: usize)
    -> &Media {
    if self.items.len() > index {
        // Good, we have something to
            return
        &self.items[index]
    } else {
        // Bad, we have nothing to
            return
    }
}
```

Rust has no concept of null, nil, or undefined

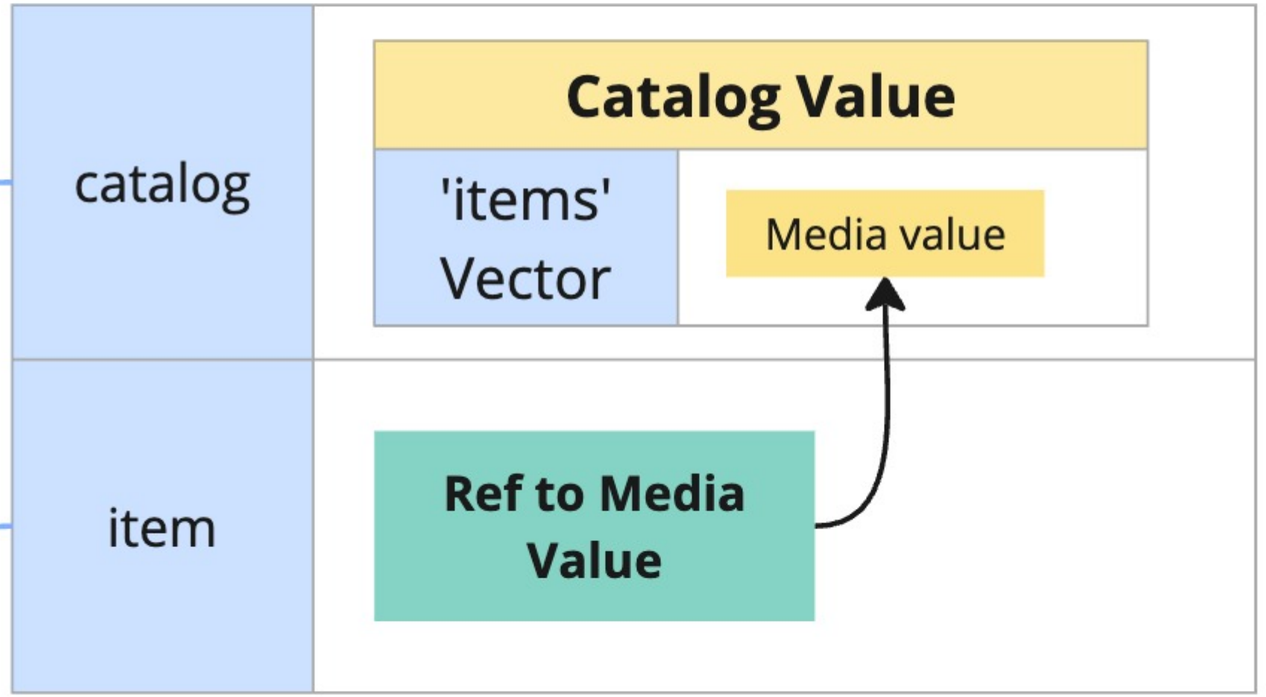
A function can't return something in some cases, but nothing in others

Common workaround is to use an enum

There's a built-in enum to handle this, we'll make our own for right now then replace it with the built-in one

```rust
fn main() {
    let catalog = Catalog::new();

    let item = catalog.get_by_index(10);
}
```



Rust has no concept of null, nil, or undefined

Common workaround is to use an enum

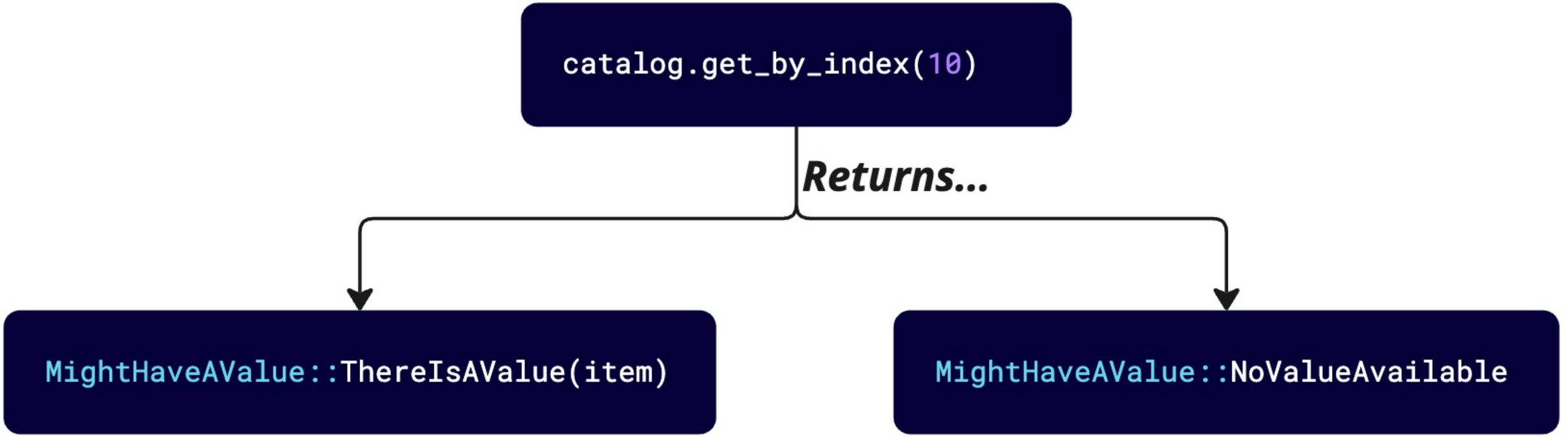
Allowing null/nil/undefined values is a huge source of bugs

Rust doesn't have **null, nil, undefined**

Rust makes you handle null/nil scenarios by using an **enum**

```rust
impl Media {
    fn description(&self) -> String {
        match self {
            Media::Book { title, author } => {
                format!("Book: {} {}", title, author)
            }
            Media::Movie { title, director } => {
                format!("Movie: {} {}", title, director)
            }
            Media::Audiobook { title } => {
                format!("Audiobook: {}", title)
            }
            Media::Podcast(id) => {
                format!("Podcast: {}", id)
            }
            Media::Placeholder => {
                format!("Placeholder")
            }
        }
    }
}
```

When working with an enum, we have to *exhaustively* handle every variant



We can use a 'match' statement or pattern matching (the 'if let' thing) to figure out which we have

```
enum Option {
    Some(value),
    None
}
```

Rust doesn't have null, nil, or undefined

Instead, we get a built-in enum called 'Option'

Has two variants - 'Some' and 'None'

If you want to work with Option you have to use pattern matching (the 'if let' thing) or a match statement

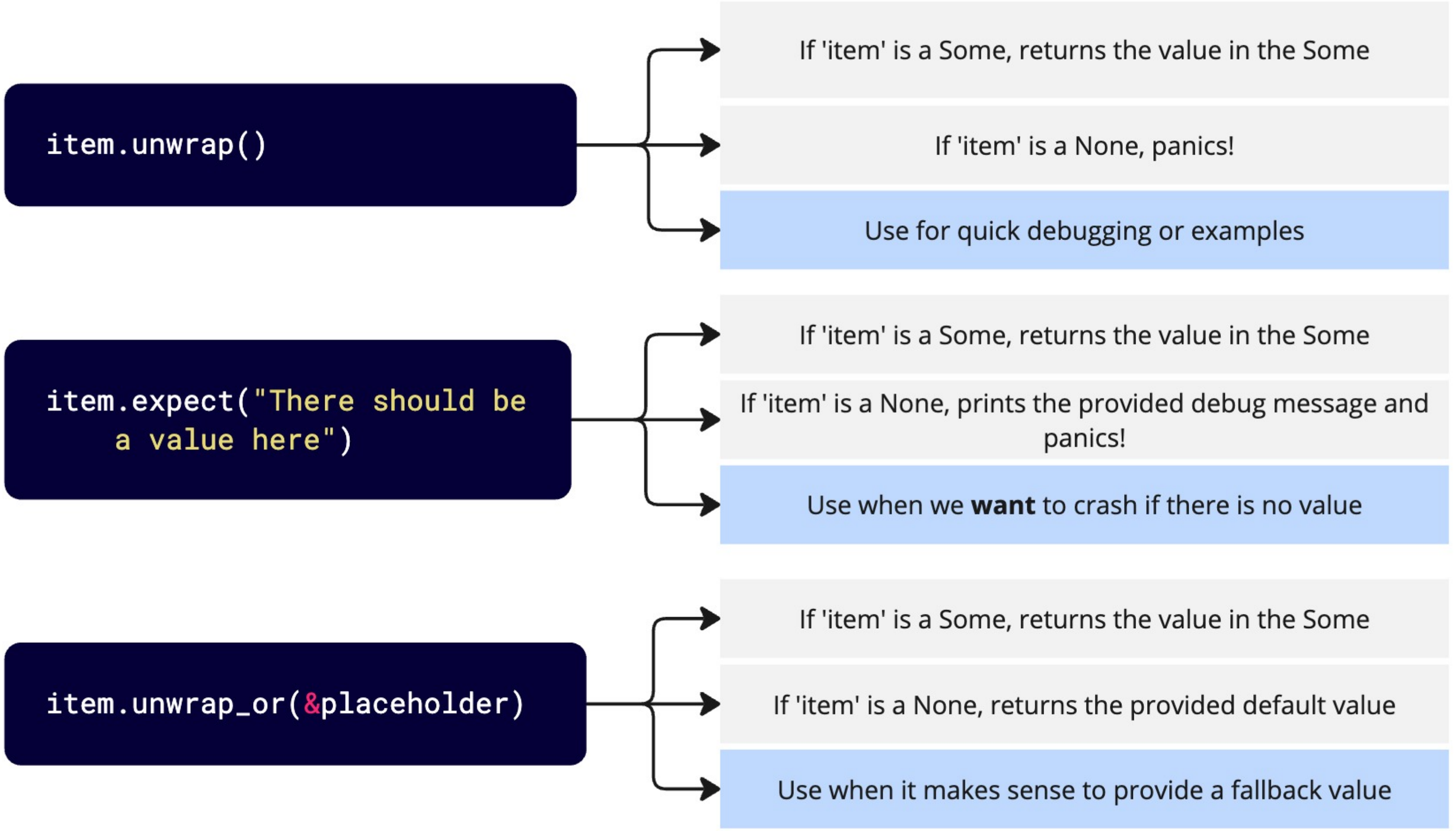
Forces you to handle the case in which you have a value and the case in which you don't

```rust
match catalog.get_by_index(9999) {
    Some(value) => {
        println!("Item: {:#?}", value);
    }
    None => {
        println!("No value here!");
    }
}
```

Using a 'match' is the ideal way to figure out if we have Some or None

There are some other ways to figure out if we have 'Some' or 'None'

They're more compact, but have big downsides (program will crash)


item.unwrap()
If 'item' is a Some, returns the value in the Some
If 'item' is a None, panics!
Use for quick debugging or examples
item.expect("There should be a value here")
If 'item' is a Some, returns the value in the Some
If 'item' is a None, prints the provided debug message and panics!
Use when we **want** to crash if there is no value
item.unwrap_or(&placeholder)
If 'item' is a Some, returns the value in the Some
If 'item' is a None, returns the provided default value
Use when it makes sense to provide a fallback value