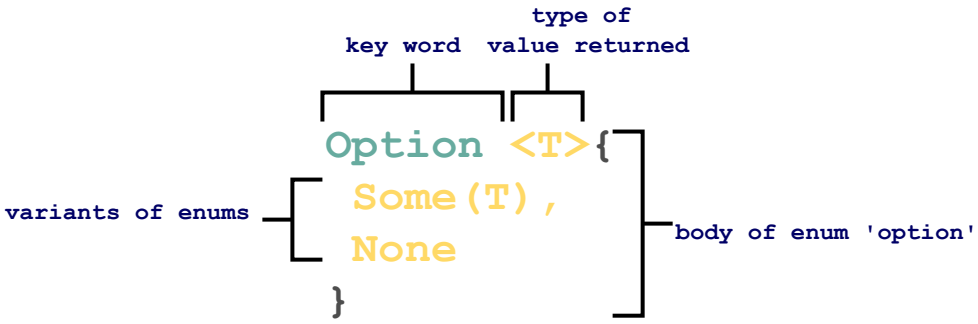
Option and Enum

This lesson will teach about a built-in enum called option.

We'll cover the following

What Is Option?

Option is a **built-in** enum in the Rust standard library. It has two variants `Some` and `None`.



Defining an enum

Variants:

- `Some(T)`, returns Some value T
- `None`, returns no value

When to Use Option?

`Options` is a good choice when:

- **The return value is none**

  Rust avoids including nulls in the language, unlike other languages. For instance, the function that returns a value may actually return nothing. So, here the Option variant `None` comes in handy.

- **The value of the variable is optional**

  The value of any variable can be set to some value or set to none.

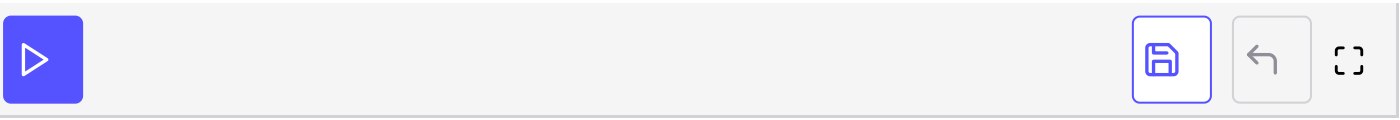
- **Out of bound exception is to be displayed**

  This is useful in the case of an array, string or a vector when an invalid index number tries to access it.

Example 1: Return Value Is None

The following example shows that if the `else` construct has no value then it can simply return `None`.

```
fn main() {
   println!("{:?}", learn_lang("Rust"));
   println!("{:?}", learn_lang("Python"));
}
fn learn_lang(my_lang:&str)-> Option<bool> {
   if my_lang == "Rust" {
      Some(true)
   } else {
      None
   }
}
```



Explanation #

- From **line 1 to 4**, `main` function is defined.

- On **line 2**, the function `learn_lang` is invoked by passing “Rust” within the function parameter.
- On **line 3**, the function `learn_lang` is invoked by passing “Python” within the function parameter.

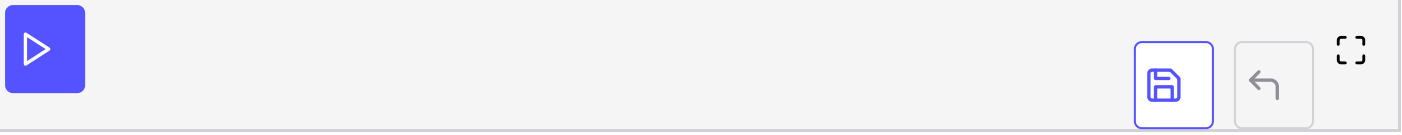
- From **line 5 to 11**, `learn_lang` function is defined. The function `learn_lang` takes a parameter `my_lang` and return an `Option<bool>`.
  - On **line 6**, an `if` condition checks if the value of `my_lang` is equal to `Rust` then it returns `Some(true)`. Note that the return type of `Option` is bool so true is passed within the `Some`.
  - On **line 7**, `else` is executed if the `if` condition evaluates to be false and returns `None`.

> **Note:** `None` does not take a parameter unlike `Some`.

Example 2: Optional Variable Value

The following example makes `level` variable of the `struct` `Course` as `Option` of type String. That means that it’s optional to set any value to it. It can be set to some value or it can be set to none.

```
//declare a struct
struct Course {
   code:i32,
   name:String,
   level: Option<String>,
}
fn main() {
   //initialize
   let course1 = Course  {
      name:String::from("Rust"),
      level:Some(String::from("beginner")),
      code:130
   };
   let course2 = Course  {
      name:String::from("Javascript"),
      level:None,
      code:122
   };
   //access
   println!("Name:{}, Level:{} ,code: {}", course1.name, course1.level.unwrap_or("Level".to_string
   println!("Name:{}, Level:{} ,code: {}", course2.name, course2.level.unwrap_or("No level defined
}
```



Explanation #

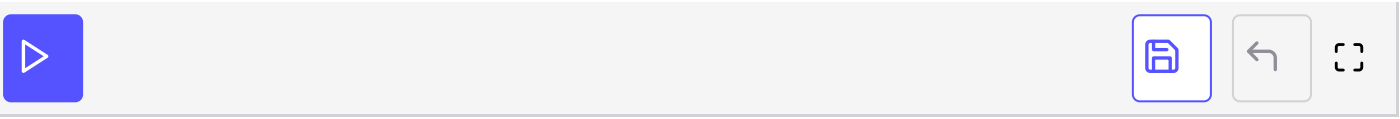
- **Line 2-5**, a `struct` `Course` has three items `code`, `name`, `level` of type `i32`, `String`, and `Option<String>` respectively.
- From **line 7 to line 22**, `main` function is defined.
  - From **line 9 to line 13**, a variable `course1` instantiates the `Course`. On **line 11**, it initializes the `level` to `Some` value. Here the value is to set to a String object, i.e., “beginner”. Note the Option has a type `String` so a value of type String can only be set to it.
  - From **line 14 to line 17**, a variable `course2` instantiates the `Course`. On **line 16**, it initializes the `level` to `None` value.
  - On **line 20**, the items of struct instance `course1` is printed using the member access operator (`.`). To print the `level` item of the `course1` instance `.unwrap_or()` built-in method is used because the `level` is of type `Option`, its values are accessed using `.unwrap_or()` with a string parameter passed to it. In this case, since the level is initialized to `Some(String::from("beginner"))` , it prints the level. But if it is set to None, it prints the string within the `.unwrap_or()` method.
  - On **line 21**, the items of struct instance `course2` is printed using the member access operator (`.`). To print the `level` item of the `course2` instance `.unwrap_or()` built-in method is used because the `level` is of type `Option`, and it is initialized with the value `None`.

Example 3: Index Out of Bound Exception

The example below uses a `match` statement that takes an index of string using `match.str.chars()` **`.nth(index_no)`** and executes the `Some` block if `index_no` is in range and `None` block otherwise.

```
fn main() {
  // define a variable
  let str = String :: from("Educative");
  // define the index value to be found
  let index = 12;
  lookup(str, index);
}
fn lookup(str: String, index: usize) {
  let matched_index = match str.chars().nth(index){
```

```
    // execute if match found print the value at specified index
     Some(c)=>c.to_string(),
     // execute if value not found
     None=>"No character at given index".to_string()
     };
  println!("{}", matched_index);
}
```



Explanation #

- From **line 1 to 7**, `main` function is defined.
    - On **line 3**, a variable `str` is initialized with value `Educative` of type `String`.
    - On **line 5**, a variable `index` is initialized with the value `12`.
    - On **line 6**, function `lookup` is invoked which takes `str` and `index` as parameters to the function.
- From **line 8 to line16**, function `loopup` is defined.
    - On **line 9**, a variable `matched_index` saves the value of `match` statement that takes `str.chars().nth(index_no)` as a condition. Here `str` and `index` are the values passed as an argument to the function. `.chars().nth(index)` finds the character at given `index`.
    - On **line 11**, `Some(c)` checks if the character is found at the given index, then it returns the character. Else,
    - On **line 13**, `None` returns a string saying that the character is not found.

`is_some()`, `is_none()` Functions

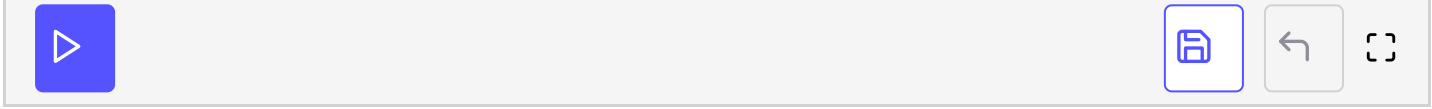
Rust provides `is_some()` and `is_none()` to identify the return type of variable of type `Option`, i.e., whether the value of type `Option` is set to `Some` or `None`.

Example 1

The following example checks whether the variable value of type `Option` is set to `Some` or `None`.

```
fn main() {
    let my_val: Option<&str> = Some("Rust Programming!");
    print(my_val); // invoke the function
```

```
}
fn print(my_val: Option<&str>){
     if my_val.is_some(){ // check if the value is equal to some value
        println!("my_val is equal to some value");
    }
    else{
        println!("my_val is equal to none");
    }
}
```



Explanation #

- The `main` function is defined from **line 1 to line 5**.
    - On **line 2**, a variable `my_var` is declared and its value is set to `Some("Rust Programming")`.
    - On **line 3**, a function `print` is invoked which takes the variable `my_var` as an argument to the function.
- On **line 6-13**, function `print` is defined.
    - The `if` construct checks if the variable `my_var` is initialized to some value using the built-in method `.is_some()`. If it is, it prints that the variable is set to some value.
    - Else, it prints it is set to none.

We need to do is to ensure that these functions return true or false. That's where `assert_eq` and `assert_ne` functions come in handy.

> **Assert Macros**
>
> - `assert_eq!(left, right)` - evaluates to true if left value is equal to that of right
> - `assert_ne!(left, right)` - evaluates to true if left value is not equal to that of right

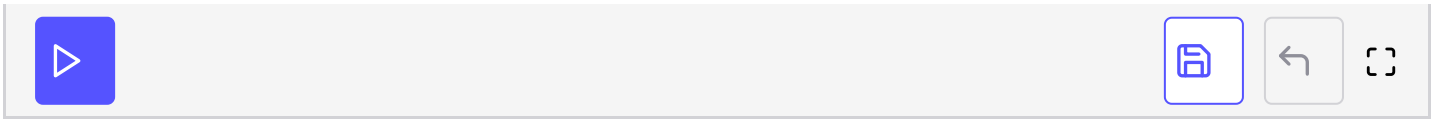
> **Output of `assert` expression?**
>
> If the assertion passes no output is displayed, and if doesn't the code gives an error saying that the assertion failed.

Example 2

The following example uses the `assert_eq!` macro to check whether the variable value of type `Option` is set to `Some` or `None`.

> **Note:** The assertion passes since the expression evaluates to true.

```
fn main() {
    let my_val: Option<&str> = Some("Rust Programming!");
    // pass since my_val is set to some value so left is true, and right is also true
    assert_eq!(my_val.is_some(), true);
    // pass since my_val is set to some value so left is false, and right is also false
    assert_eq!(my_val.is_none(), false);
}
```



Explanation

- On **line 2**, declares a variable `my_var` and sets it to `Some("Rust Programming")`.
- On **line 4**, an `assert_eq!` takes `var.is_some()` and checks if it's equal to `true`. Since `my_val` is set to some value so left `my_val.is_some()` is true and right is also set to `true`. The assertion passes.
- On **line 6**, an `assert_eq!` takes the expression `var.is_none()` and checks if it's equal to `false`.Since `my_val` is set to some value so left `my_val.is_none()` is false and right is also set to `false`. So the assertion passes.

There is often a situation when you have to display messages like "File Not found" and "Ok" on failures and successes respectively. Let's see how the built-in Result enum can help you do this in the next lesson.