Enums and Pattern Matching

Defining Enums

- Enums allow us to enumerate over a list of variables
- Example:

  ```
  enum IpAddrKind{
      V4,
      V6,
  }

  let four = IpAddrKind::V4;
  ```

  - This just defines an ENUM but it doesn't really do anything or contain anything useful
  - If you were to print this, (you would need to do a debug print) but you would see that it just prints out `V4`.

- To give it some more information, you can make it into a struct

  - Example:

    ```
    enum IpAddrKind{
        V4,
        V6,
    }

    struct IpAddr{
        kind: IpAddrKind,
        address: String,
    }

    let four = IpAddr{IpAddrKind::V4, String::from("ip address v4")};
    ```

- A better way of doing this is by storing values directly into the enum using tuple structure

  - Example:

    ```
    enum IpAddrKind{
        V4(u8,u8,u8,u8),
        V6(String),
    }

    let four = IpAddrKind::V4(String::from("ip address v6 address"));
    ```

Defining Functions for Enum

- Just like Structs, enums can have their own **associated functions**
- Example:

```
enum IpAddrKind{
    V4(u8,u8,u8,u8),
    V6(String),
}
impl IpAddrKind{
    fn new_v6(address: String) -> IpAddrKind{
        let kind = IpAddrKind::V6(String::from(address));
        kind
    }
}
```

Option Enum

- Allows for matching of None and handle the case where there is no value for a type
- Base Example:'

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

let some_number: Option<i32> = Some(1);
let absent_number: Option<i32> = None;
```

  - Here the enum Option allows for two types, one is any type and the other is an empty value.
  - One thing to note is we don't need to define the type if we give `Some()`, but if we use `None` then we need to define the type. This is like an uninitialized variable.

- This is such a useful aspect that the enum is there by default.
- Let's look at some use cases
- Example:

~~~
let x: i8 = 3;
let y: Option<i8> = Some(5);
let sum = x+y; //error
let sum = x + y.unwrap(); //returns a value if not None
let sum = x + y.unwrap_or(0); //returns a default value is needed
```rust
~~~

```
*  Here we can see that x and y are not the same type and so cannot be
treated as such
*  Although Y has the option to be of that type, it also has the option to
not be of that type
*  The way to extract the value is from `unwrap()` which returns a value if
there is `Some()` value.
*  The `unwrap_or()` function is used if you are not sure whether there is a
value, then you can return a default value in the case the pattern matches
with None.
```

Pattern Matching

- It is similar to functional programming language pattern matching
- Uses `match <Enum>{}` this allows you to give all enums fields particular value
- Using match needs to be exhaustive, as in there must be an option for enum fields
- Example:

```
#[derive(Debug)]
enum UsState{
    NewYork,
    California,
    Alaska,
}
#[derive(Debug)]
enum Coin{
    Penny,
    Nickel,
    Dime(UsState),
    Quarter,
}


fn value_in_cents(coin : Coin) -> u8 {
    match coin{
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime(state) => {
            println!("Dime from {:?}",state);
            10
        },
        _ => 0
    }
}
```

  - Here we can see that we match each value of coin
  - We can also see that we can bind the value of the Enum from Dime into a variable

- Lastly we can see that we can make a complex match statement for Dime.
- `_` This means for anything else that doesn't match the top, even Nones, then return a default value.

- Let's see how we can use pattern matching with Option Enum

  - Example: ```rust fn plus_one(x: Option) -> Option{ match x{ None => None, Some(i) => Some(i+1), } }

    ```
    let x = Some(5);
    println!("x -> {}, x+1 -> {}",x.unwrap(), plus_one(x).unwrap());
    ```
    ```

    - Here we can see that there is a pattern matching with i32 Options. And it can be of only two types, None or Some.
    - If the option is None then it remains as None
    - If the option has some value, then we return `Some(i+1)`

- **IF LET** syntax

  - It is in between `match` and `if` statements
  - It doesn't have to be exhaustive but it is performing a match of sort
  - Example: ```rust if let Some(3) = some_value{ println!("three"); }

    ```
        *It is read a bit backwards,
        * You read it like this: "if let some_value == Some(3) then print
    three"
        *You are matching some_value to the pattern Some(3)
        * All other patterns are ignored in this case
    ```
    ```