Product Types, Sum Types, AKA Tagged Unions, or we've been cheated all these years

One of Rust's great features not really seen outside of functional languages (unless you count union. and you absolutely shouldn't) is the Sum Type. To review watered down set theory as it applies to types, *product types* are those types that combine other types using 'and', and *sum types* combine other types using 'or'.

Product types are meat and potatoes types that you are already used to. Lists, tuples, hashmaps, etc. Yawn.

Sum types, however are more exotic, and a lot more powerful. In Rust, the type which falls under this category is *enum*. Rust's enum is unlike the enum that you are probably used to in c or c++, and have complained about the lack of in python. Because Rust's enum is a full blown sum type. We will go over all of its great features, but first, lets step back and look at how you create an enum. You do so using the *enum* keyword:

```
pub enum Level {
    Show,
    Seq,
    Shot
}
```

Instantiating a variant is rather simple as well:

```
let level = Level::Shot;
assert_eq!(level,Level::Shot);
```

Ok, well there is nothing particularly exotic there, eh? Kind of run of the mill. But wait, there's more. Because each variant is not limited to a simple name. Each variant may be a struct or a tuplestruct as well.

```
pub enum Level {
    Show(String),
    Seq{show: String, seq: String},
    Shot{pub show: String, pub seq: String, pub shot: String}
}

let show = Level::Show(stringify!(DEV01));
let lvl = Level::Shot{show: stringify!(DEV01), seq: stringify!(RD), shot:
stringify(0001)};
```

So the first problem as you might guess is accessing said data. While you could do what I have above and make the fields public, that is not how its done. Instead, we rely on destructuring.

if let

The first form that we will look at is the `if let` form, and it looks like this:

```
if let Level::Show(show) = level {
    println!("show: {}", show);
} else {
    println!("not a show");
}
```

Destructuring

This is destructuring, not unlike python's tuple destructuring, except that this can be arbitrarily complex. Now you might be wondering about ownership, at this point. The result of a destructure is dependent on the type of the data (owned vs non) and the use of modifiers.

You may add a couple of them - `ref` and `mut`.

```
let mut lvl = Level::Show("dev01".into());

if let Level::Show(ref mut show) = lvl {
    *show = show.to_uppercase();
}
println!("the value of lvl is now: {:?}", lvl);
```

Destructuring allows you to extract owned content from containers, as well as reference components.

match

The most powerful tool for destructuring uses the `match` keyword. Match is most similar to `switch` in c, although with the addition of pattern matching and the lack of fall through

```
let level = Level::Shot{show: "dev01".into(), seq: "rd".into(),
"0001".into()};

match &level {
    // branches
    Level::Show(show) => println!("show: {}", show),
    // elision
    Level::Seq { show, seq } => println!("show {}, seq: {}", show, seq),
    Level::Shot { show, seq, shot } => println!(
        "the show: {} has a sequence: {} with a shot: {}",
        show, seq, shot
    ),
}
```

A couple of things to note:

- by default, matches must be exhaustive. You cannot leave off a pattern. The compiler will bark at you.
- You can use '_' to match all remaining patterns in you want to.
- Ownership is dictated by the ownership of the variable that you are matching against. If it is owned, you can extract owned components. Otherwise, you will be dealing with references
- you can explicitly note a reference using the `ref` keyword
- you can explicitly make the ref mutable by using the `mut ref` keywords, provided the variable is mutable to begin with.
- match is an expression. Each arm returns the value of the last expression in the branch. You can add parens to group multiple expressions together if you want. If the last statement in the branch ends in a semi-colon, then the return value is the unit `()`.
- you can nest match expressions
- you can assign the return value to a variable (and often will).

```
let mut level = Level::Shot {
        show: "dev01".into(),
        seq: "rd".into(),
        shot: "0001".into(),
    };

    match level {
        // branches
        Level::Show(ref mut show) => {
            println!("show: {}", show);
            *show = show.to_uppercase();
        }
        // elision
        Level::Seq { show, seq } => println!("show {}, seq: {}", show,
seq),
        Level::Shot {
            ref mut show,
            ref mut seq,
            ref mut shot,
        } => {
            *show = show.to_uppercase();
            *seq = seq.to_uppercase();
            println!(
                "the show: {} has a sequence: {} with a shot: {}",
                show, seq, shot
            );
        }
    }
```

Match Guards

You can refine the match arms using guards. For example

```
let id = 3;
match id {
```

```
    1 | 2 | 3 => println!("small"),
    4 | 5 | 6 => println!("med"),
    _ => println!("large") ,
}
```

enums may have methods, just like structs.

Destructuring is very useful, but that is only half the power of these enums. The other killer feature is that enums may define methods and associated functions just like structs.

```
impl Level {
    /// print the content
    pub fn printme(&self) {
        match self {
            Self::Show(show) => println!("show: {}", show),
            Self::Seq { show, seq } => println!("show {}, seq: {}", show,
seq),
            Self::Shot { show, seq, shot } => println!(
                "the show: {} has a sequence: {} with a shot: {}",
                show, seq, shot
            ),
        }
    }

    pub fn show(&self) -> String {
        match self {
            Self::Show(show) =>show.clone(),
            Self::Seq { show,.. } => show.clone(),
            Self::Shot { show, .. } => show.clone(),
        }
    }
}
```

standard enums

The standard library defines a couple of very important enums that we will look at now

Option

The first is `Option`. Rust, as I have noted, has no `null`. That much maligned million dollar mistake is not part of Rust. But we need to model the notion of an optional value somehow. So, we do it with an enum. If you are Haskell savvy, you might be saying that is just a `Maybe` monad, and you would be right.

Here is how Option is defined:

```
pub enum Option<T> {
    Some(T),
```

```
    None
}
```

An Option is either Some of something, or None. Option shows off Rust's functional roots by providing the standard functor method `map`. Map lets you operate on the contained data without extracting it, which is a nice convenience.

```
pub fn map<U, F>(self, f: F) -> Option<U>
where
    F: FnOnce(T) -> U;
```

Map takes a function that can transform the contents of one type to another, returning a new Option.

```
impl Level {
    pub fn to_uppercase(&self) -> Self {
        match self {
            Self::Show(show) => Self::Show(show.to_uppercase()),
            Self::Seq{show,seq} => Self::Seq{show: show.to_uppercase(),
seq: seq.to_uppercase()},
            Self::Shot{show, seq, shot} => Self::Shot{
                show: show.to_uppercase(),
                seq: seq.to_uppercase(),
                shot: shot.to_uppercase()
            }
        }
    }
}

let  level = Some(Level::Shot {
        show: "dev01".into(),
        seq: "rd".into(),
        shot: "0001".into(),
    });

let new_level = level.map(|x| x.to_uppercase());
println!("its uppercase {:?}", new_level);
let level = None;
let still_none = level.map(|x| x.to_uppercase());
```

Rust also lets you `unwrap` the option, extracting the data, if you are certain that it is not None. However, be forewarned, rust will `panic` if you are wrong....

```
let foo: Option<String> = Some("test".into());
let bar = foo.unwrap();
// however
let foo: Option<String> = None;
foo.unwrap(); //kaboom
```

There are other safer convenience functions like `unwrap_or`

```
pub fn unwrap_or(self, default: T) -> T
```

and `unwrap_or_else`

```
pub fn unwrap_or_else<F>(self, f: F) -> T where
    F: FnOnce() -> T,
```

(Notice that the former takes a default value which is greedily evaluated, and the latter takes a function or closure which is lazily evaluated.)

It also has some interesting type manipulation that will come in handy. One of those which you will find yourself needing sooner or later is `as_ref`, which transforms an `&Option<T>` into an `Option<&T>`. Pretty handy if you are passing an option around by reference and you want to crack it open and get a reference to its guts.

In fact, there are so many interesting methods, that you should really just consult the documentation. Its good practice for reading function signatures... Option Docs

Result

The second enum that you will see a lot is Result. Whereas `Option` models an optional value, `Result` models a value that may either represent a success or a failure. This resembles Haskell's `Either` monad. The signature is as follows:

```
enum Result<T,E> {
    Ok(T),
    Err(E),
}
```

This is a big part of Rust's error handling story. Not all mind you, as Rust provides an Error trait that we will look at later. But for now, lets examine Result a bit more. Result is used to return from functions that expect to encounter recoverable errors. Because `Result` has two unconstrained generic parameters, you can use any types you fancy.

```
fn bad_game(guess: u8) -> Result<&'static str,&'static str> {
    if guess == 1 {
        Ok("you did it")
    } else {
        Err("you guessed wrong. and that is an error")
    }
}
```

Of course I really phoned it in here. We don't use String for error types as it has no semantic information. One thing to note is that Rust expects you to use results. That is, you should be checking to see what state they are in (Ok or Err) and handling them appropriately.