Getting Started

case, cond, and if

In this chapter, we will learn about the `case`, `cond`, and `if` control flow structures.

`case`

`case` allows us to compare a value against many patterns until we find a matching one:

```
iex> case {1, 2, 3} do
...>   {4, 5, 6} ->
...>     "This clause won't match"
...>   {1, x, 3} ->
...>     "This clause will match and bind x to 2 in this clause"
...>   _ ->
```

```
...>     "This clause would match any value"
...> end
"This clause will match and bind x to 2 in this clause"
```

If you want to pattern match against an existing variable, you need to use the `^` operator:

```
iex> x = 1
1
iex> case 10 do
...>   ^x -> "Won't match"
...>   _ -> "Will match"
...> end
"Will match"
```

Clauses also allow extra conditions to be specified via guards:

```
iex> case {1, 2, 3} do
...>   {1, x, 3} when x > 0 ->
...>     "Will match"
...>   _ ->
...>     "Would match, if guard condition were not satisfied"
...> end
"Will match"
```

The first clause above will only match when `x` is positive.

Keep in mind errors in guards do not leak but simply make the guard fail:

```
iex> hd(1)
** (ArgumentError) argument error
iex> case 1 do
...>   x when hd(x) -> "Won't match"
...>   x -> "Got #{x}"
...> end
"Got 1"
```

If none of the clauses match, an error is raised:

```
iex> case :ok do
...>   :error -> "Won't match"
...> end
** (CaseClauseError) no case clause matching: :ok
```

Consult the full documentation for guards for more information about guards, how they are used, and what expressions are allowed in them.

Note anonymous functions can also have multiple clauses and guards:

```
iex> f = fn
...>   x, y when x > 0 -> x + y
...>   x, y -> x * y
...> end
#Function<12.71889879/2 in :erl_eval.expr/5>
iex> f.(1, 3)
4
```

```
iex> f.(-1, 3)
-3
```

The number of arguments in each anonymous function clause needs to be the same, otherwise an error is raised.

```
iex> f2 = fn
...>   x, y when x > 0 -> x + y
...>   x, y, z -> x * y + z
...> end
** (CompileError) iex:1: cannot mix clauses with different arities in
anonymous functions
```

cond

`case` is useful when you need to match against different values. However, in many circumstances, we want to check different conditions and find the first one that does not evaluate to `nil` or `false`. In such cases, one may use `cond`:

```
iex> cond do
...>   2 + 2 == 5 ->
...>     "This will not be true"
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   1 + 1 == 2 ->
...>     "But this will"
```

```
...> end
"But this will"
```

This is equivalent to `else if` clauses in many imperative languages - although used less frequently in Elixir.

If all of the conditions return `nil` or `false`, an error (`CondClauseError`) is raised. For this reason, it may be necessary to add a final condition, equal to `true`, which will always match:

```
iex> cond do
...>   2 + 2 == 5 ->
...>     "This is never true"
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   true ->
...>     "This is always true (equivalent to else)"
...> end
"This is always true (equivalent to else)"
```

Finally, note `cond` considers any value besides `nil` and `false` to be true:

```
iex> cond do
...>   hd([1, 2, 3]) ->
...>     "1 is considered as true"
...> end
"1 is considered as true"
```

`if` and `unless`

Besides `case` and `cond`, Elixir also provides `if/2` and `unless/2`, which are useful when you need to check for only one condition:

```
iex> if true do
...>   "This works!"
...> end
"This works!"
iex> unless true do
...>   "This will never be seen"
...> end
nil
```

If the condition given to `if/2` returns `false` or `nil`, the body given between `do` - `end` is not executed and instead it returns `nil`. The opposite happens with `unless/2`.

They also support `else` blocks:

```
iex> if nil do
...>   "This won't be seen"
...> else
...>   "This will"
...> end
"This will"
```

This is also a good opportunity to talk about variable scoping in Elixir. If any variable is declared or

changed inside `if`, `case`, and similar constructs, the declaration and change will only be visible inside the construct. For example:

```
iex> x = 1
1
iex> if true do
...>   x = x + 1
...> end
2
iex> x
1
```

In said cases, if you want to change a value, you must return the value from the `if`:

```
iex> x = 1
1
iex> x = if true do
...>   x + 1
...> else
...>   x
...> end
2
```

> *Note: An interesting note regarding `if/2` and `unless/2` is that they are implemented as macros in the language; they aren't special language constructs as they would be in many languages. You can check the documentation and the source of `if/2` in the `Kernel` module docs. The `Kernel` module is also where operators like `+/2` and*

> *functions like `is_function/2` are defined, all automatically imported and available in your code by default.*

We have concluded the introduction to the most fundamental control-flow constructs in Elixir. Now it is time to talk about “Binaries, strings, and char lists”.