Elixir for Beginners

§

Made with Obsidian

Type blog | Category computer-science | Technologies Elixir, VS Code, PowerShell 7

Website Post Link

Elixir is a compiled, dynamically-typed, general-purpose, functional programming language developed by Brazilian software developer José Valim, first released in 2012. It runs on top of the BEAM virtual machine, which is also used to implement the Erlang programming language. Thus Elixir inherits many of Erlang's abstractions and methods.

Elixir is especially good for building highly concurrent and fault-tolerant applications; concurrent applications are programs designed to execute multiple tasks or activities simultaneously, often in a parallel or overlapping fashion.

In this Blog Article, we'll review some historical context around Elixir, its main advantages, and some of the most popular use cases. We will then proceed to install the language along with a useful VS Code extension and discuss how to create projects using Mix. We'll write our first Elixir module while discussing syntax, data types, variable definitions, operators, some of the most relevant methods for interacting with different types of variables, flow control statements using conditional constructs and cases, iterators & recursion, function definitions, additional module definitions, and finally, compiling and running our project using Mix.

Finally, we'll include some next steps for those interested in becoming more well-versed in this superb language.

We'll be using Elixir scripts which can be found in the Blog Article Repo.

§

Table of Contents

§

Why learn Elixir?

What does this language bring to the table in an ocean of programming languages? Well, let us start by noting that not only does Elixir consistently rank as one of the most loved programming languages in the yearly Stack Overflow Developer Survey (*getting second place, just below Rust, in last year's survey*), but Phoenix, a web framework written in Elixir, ranked as the most loved framework in the same survey last year, effectively obliterating every JavaScript framework by more than ten percentual points.

But why are developers so happy with Elixir?

1. Distributed systems and concurrency

Distributed software is becoming normalized as new languages implement new ways to deal with concurrency in a safe manner. This is important because some years ago, a single CPU might have been enough, but today, that's most definitely not the case, even with efficient multi-threading and low-level management.

In Elixir: The Documentary, creator José Valim explains that traditional languages are sometimes bottlenecked in terms of performance and scalability because they were not designed to deploy distributed programs.

Elixir tries to solve this problem by allowing the building of scalable, fault-tolerant, and distributed applications. It achieves this through its support for concurrency and parallelism, fault-tolerance mechanisms, and distributed computing capabilities.

In this same mini-documentary, Stefan Kellner, CEO of Qixxit, mentions a critical aspect of Elixir; a system involving thousands of active simultaneous connections can be deployed using Elixir and will result in a highly resilient implementation.

This is important since systems will only get more complex and have more moving parts as we expand; we will need battle horses that can process multiple connections quickly and infallibly.

2. Functional programming

When talking about programming paradigms, we have four relevant options:

- **Imperative:** Describes the sequence of steps a program must take to accomplish a task.
- **Procedural:** Organizes code into procedures (*also known as functions or subroutines*) that can be called to perform specific tasks.
- **Object-oriented:** Models programs as a collection of objects that interact with each other. Objects are instances of classes that encapsulate data (*attributes*) and behaviors (*methods*).
- **Functional:** Computation is based on the evaluation of mathematical functions and the avoidance of mutable states and side effects. Functional programming promotes immutability, recursion, and higher-order functions.
- **Multi-paradigm:** A combination of two or more paradigms.

If we learned to program using Python, chances are, we have only been exposed to Object-Oriented programming; OOP requires a specific way of thinking about writing code, which sticks with us the more we write in a language using OOP.

Conversely, Elixir is a functional language, meaning it uses functional programming. This paradigm is entirely different from OOP to the very bones; thus, it requires a different way of thinking when solving problems.

Shifting from OOP to functional programming is not easy, but there are several advantages:

- Many high-performing concurrent languages like Scala, Haskell, and Erlang use functional programming. Learning a new language of the same nature is much easier if we are already familiar with the paradigm.
- It provides a different way to think about approaching and solving problems because, in the end, programming is just that: solving problems.

3. Scalability and performance

Elixir has lightweight processes, preemptive scheduling, and a garbage collection mechanism that can handle millions of processes with low latency. It was designed to handle large interconnected systems from the ground up and scale them virtually limitlessly.

4. Inheritance from a titan

Erlang is a language developed by Ericsson, originally designed for telecom switching. Its main emphasis is on building extremely robust and fault-tolerant distributed applications that can quickly adapt to changing requirements. Erlang has built-in concurrency support so that massive requests can be serviced simultaneously, and single software errors can be easily contained.

Elixir is built based on Erlang and runs on the same VM called BEAM. This makes a statement; Elixir will handle even the most demanding tasks. Additionally, Elixir inherits Erlang's OTP by default, a set of libraries and design principles providing middle-ware to develop concurrent systems.

5. Syntax

Elixir's syntax is based on Ruby. It's highly readable and expressive while having all the functional programming advantages. Elixir has practical syntactical elements such as the pipe operator `|>` used to chain functions, as well as pattern matching that allows us to destructure data structures and match them against specific patterns.

Overall, Elixir's syntax is easy to learn, write, read, and debug.

6. Adoption

Elixir is an emerging language, meaning its adoption is not up there with more popular languages. However, companies that do use Elixir tackle very specific & complex problems with it:

- **Discord:** A popular voice and text chat application for gamers using Elixir and Erlang to power their real-time communication infrastructure, infallibly handling massive amounts of concurrent users.
- **Pinterest**: A social media platform that allows users to discover and save ideas for their projects and interests. They implemented Elixir to scale up their real-time notification system.
- **PepsiCo:** A multinational food and beverage corporation using Elixir for their entire supply chain management system.
- **WhatsApp:** A popular messaging app using Elixir and Erlang for their real-time communication infrastructure.
- **Motorola:** A multinational telecommunications company using Elixir for handling massive volumes of video data in their video security and analytics platform.

- **Financial Times:** A British daily business newspaper using the Absinthe framework in Elixir and Elixir's meta-programming ability to create DSLs (*Domain Specific Languages*).
- **Toyota Connected:** A mobility & transportation subsidiary of Toyota Motor Corporation, using Elixir in their Mobility Service Platform (*MSPF*), which connects their cars, allowing them to send real-time events.
- **Phoenix framework:** A web framework for the Elixir programming language designed to help developers build high-performance, scalable web applications.

Most applications are centered around high-traffic, concurrent, complex systems that require solid performance and fault tolerance even in the most demanding production environments.

7. Community

Even though Elixir is not as popular as other languages, it has an active community filled with enthusiasts and experts constantly providing information on the latest and greatest.

Also, it's open source. Adding to the fact that the language is still niche, we can focus on learning Elixir and even submitting new packages to Hex or making pull requests to submit new source code.

Now that we're hopefully convinced that Elixir is a great language to learn and has huge potential, we can start by making some preparations.

8. Package repository

Elixir uses Hex as its package manager. All Hex packages are available on the official Hex packages page. As of the writing of this article, there are 16k+ packages available, with over 9bn total downloads.

There are packages tailored for everything: from JSON parsing to SSL verification to mime-type handling to documentation generation, and of course, packages tailored for concurrent systems design such as GenServer, Task, Flow, and Broadway.

Another great thing about Hex is that it supports private packages for organizations via a modest paid subscription. We can publish private packages to Hex.pm that only our organization members can access and download. With our organization, we get a repository namespace on Hex.pm so that our private packages will not conflict with packages in the global public repository.

Overall, Elixir's Hex is fully-featured and supported by thousands of developers across the globe.

§

What to expect

Elixir is relatively easy to learn, especially when coming from a functional programming context. However, if we're dealing with a more complex system, grasping and implementing concepts such as concurrency and parallelism, processes, macros, and some functional programming aspects, can be challenging.

In this segment, we'll focus on learning the fundamentals while introducing some general concepts around functional programming.

Another thing to mention is that we'll be using Mix for project creation, compilation, and running purposes. This will save us some time since most of the boilerplate components of a typical Elixir project will be handled for us.

§

Installation

For this segment, we will need to install four main components:

- The Erlang programming language.
- The Elixir programming language.
- Visual Studio Code.
- Elixir VS Code extension.

We will also install some packages, which will come later when we get to the dependencies section.

The installation and setup will be focused on the Windows operating system. Still, a similar process can be used for macOS or Unix-like platforms.

1. Elixir

We will first install the latest stable release of the Elixir programming language. We can head to the official Elixir website downloads page for Windows. We will click on Download the installer. Once we have it, we'll run the executable and follow the steps below:

- If we have Microsoft Defender enabled, it will probably tell us that the installation is not secure. We will omit this warning and initialize the installer.
- We will select the latest release (*they're sorted from newest to oldest*), which at the time of this article, should be `1.14.2`.
- We will then select the version for the type of CPU architecture we have (*32-bit or 64-bit*). If we're working on a recent computer model, we may have to install the 64-bit version. If unsure, we can always check our system architecture.
- The Elixir installer will automatically install the Erlang language, so we don't have to worry about that. Once we're prompted with the Erlang installation, we will select:
  - `Erlang`
  - `Associations`
  - `Erlang Documentation`
- Once the installation concludes, we will be asked if we want to add Erlang, Elixir, and `escripts` to `PATH`. We will select yes to all (*this is important because if we don't select this, our system will be unable to find the required executables for each language*).

We can verify our Elixir installation by opening a new PowerShell prompt and typing the following:

Code

```
elixir --version
```

If all goes well, we should get something like this, depending on the version we installed:

Output

```
Erlang/OTP 25 [erts-13.0.4] [source] [64-bit] [smp:20:20] [ds:20:20:10] [async-threads:1]
[jit:ns]

Elixir 1.14.4 (compiled with Erlang/OTP 25)
```

2. VS Code

If we don't yet have VS Code installed, we can get it from the official downloads page. We need to select the Windows 8, 10, 11 executable and wait for it to download. When the installation is complete, we can verify by opening the Visual Studio Code application directly from the Windows start menu. A detailed configuration guide for VS Code is out of the scope of this article but can be consulted on the VS Code official documentation site.

3. Elixir VS Code extension

Once we have Erlang, Elixir, and VS Code installed, we will proceed to install the Elixir VS Code Extension:

1. Open VS code and head to the Extensions menu in the left panel. We can also open the Extensions menu by using the shortcut `Ctrl` + `Shift` + `X` or by opening the command palette by typing `F1` and searching for *Extensions: Install Extensions*.
2. We will search for `ElixirLS: Elixir support and debugger`, maintained by *ElixirLS*, install it, and enable it. We can also get the extension using this link.

We mentioned that Elixir is a compiled language, so it'll be best to have a shell prompt at hand.

Now that everything's in place, we're ready to start configuring our working environment.

§

Creating a project

Elixir projects can be created using Mix. Mix is a handy tool that manages a good portion of the boilerplate code required for kickstarting our first project and creating the required files to compile it and run it in a breeze. Mix also introduces handy tools for testing our application and managing its dependencies. There are other methods for compiling and running Elixir files, such as `iex` or directly with `elixir`, but we won't be covering those here.

To create a new project called `project_1`, we can open a new PowerShell session and include the following:

Code

```
mix new project_1
```

Output

```
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating Mix.exs
* creating lib
* creating lib/project1.ex
* creating test
* creating test/test_helper.exs
* creating test/project1_test.exs

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd project_1
    mix test

Run "mix help" for more commands.
```

This command will create several files and folders:

- `project_1`: The main folder for our project.
  - `.elixir_ls`: A folder containing configuration and cache files used by ElixirLS to provide advanced code-editing features, such as autocompletion, code navigation, and code formatting. This folder is created by the Elixir extension we're using, in this case, ElixirLS for VS Code.
  - `lib/project1.ex`: Our project's main source file.
  - `test/project1_test.exs`: Contains an example test and serves as a starting point for writing our own tests for our application.
  - `test/test_helper.exs`: The script invoked when testing our program; Elixir uses the ExUnit testing framework, which comes built-in with the language, to write and run program tests.
  - `.formatter.exs`: A configuration file used by the Elixir code formatter. The code formatter is a tool that automatically formats Elixir code according to a set of predefined rules, making it easier to read and maintain. The `.formatter.exs` file allows us to customize the code formatter's behavior by specifying options such as line length, indentation, and syntax preferences.
  - `.gitignore`: If we're including a GitHub repository along with our project, a `.gitignore` file serves as a way to let git know which files and folders to ignore in our commits. If we have an intended repository, we can leave this file. Else, we can delete it. Mix populates the project's `.gitignore` file with some helpful entries we are not supposed to include in a GitHub repository.
  - `Mix.exs`: The project's configuration file. It's used to define the project's dependencies, version, and other metadata and to configure various settings related to the build process, such as compilers and output directories.
  - `README.md`: Yet another GitHub-related file. A `README.md` file is typically used in repositories to provide information about our project. As with `.gitignore`, Mix will populate this file with useful information. We can delete this file if we're not working with a GitHub repo.

As we may have noted, Elixir source files have the `.ex` file extension. We can open the `lib/project1.ex` in VS Code and start writing some code.

Before we start writing Elixir code, we must consider that indentation is not part of Elixir's syntax, i.e., we don't need correct indentation for an Elixir project to compile. That said, proper indentation is always a good idea when we have cases such as this since it improves readability considerably.

1. The main source file

Upon creating our Mix project, our main file, `lib/project1.ex`, was also created and populated. It should look something like such:

Code

```elixir
defmodule Project1 do
  @moduledoc """
  Documentation for `Project1`.
  """

  @doc """
  Hello world.

  ## Examples

      iex> Project1.hello()
      :world

  """
  def hello do
    :world
  end
end
```

The first line denotes a module declaration. This is achieved by using `defmodule projectname do end`, the main module of our project.

Below the module declaration, docstrings using triple double quotes `"""` are included:

- The `@moduledoc` attribute is used to add documentation to the module.
- The `@doc` attribute is used before a function to provide documentation.

Finally, we have a function declaration. By default, the name of our function is `hello`, although we probably want to change that to something more transcendental. We will also take some time to write simple module and function docstrings:

Code

```elixir
defmodule Project1 do
  @moduledoc """
  This is a module containing functions to explore the Elixir language.
  """

  @doc """
  Print hello world.
  """
  def main do
    IO.puts("Hello World")
  end
end
```

Much better, right? So now we would like to execute this project and make it print the intended message to `stdout`.

One thing to note is that, although we're defining our first function as `main`, this is just for clarity and convention purposes since Elixir does not require us to define a `main` function as with other languages.

The idea is to define a `main` function for each module, responsible for calling all the other functions in our module.

2. Managing dependencies

Dependencies in Elixir are external libraries or packages used in a project to provide additional functionality. Elixir uses the Hex package manager to manage dependencies.

The complete set of packages currently available in Hex can be consulted here.

As we have already discussed, we can include dependencies for our project in the `project_1/mix.exs` file.

2.1 The mix.exs file

If we open our `mix.exs` file, we can see that it contains the following:

Code

```elixir
defmodule Project1.MixProject do
  use Mix.Project

  def project do
    [
      app: :project_1,
      version: "0.1.0",
      elixir: "~> 1.14",
      start_permanent: Mix.env() == :prod,
      deps: deps()
    ]
  end

  def application do
    [
      extra_applications: [:logger]
    ]
  end

  defp deps do
    [
      # {:dep_from_hexpm, "~> 0.3.0"},
      # {:dep_from_git, git: "https://github.com/elixir-lang/my_dep.git", tag: "0.1.0"}
    ]
  end
end
```

We're interested in the `deps` section; we can enclose dependencies using Hex or by directly including a GitHub Repository URL.

We can include a package from Hex using the following syntax:

Code

```
defp deps do
  [
    {:jason, "~> 1.4.0"},
  ]
end
```

If we try to compile our project as is, we will not be able to do so yet; we also need to install Hex if we include dependencies for the first time (*we'll review compilation in more detail later on. For now, we need to know that `mix run -e "Project1.main()` will compile and run the main function of our Project1 module").

Code

```
mix run -e "Project1.main()"
```

Output

```
Could not find Hex, which is needed to build dependency :jason
Shall I install Hex? (if running non-interactively, use "mix local.hex --force") [Yn]
```

We will then select `Y`. Once we have Hex installed, we will execute the following command:

Code

```
mix deps.get
```

This command is used to fetch and install dependencies for an Elixir project. It's typically used after modifying the project's `Mix.exs` file to add or update dependencies.

If everything goes right, we should end with the following output:

Output

```
Resolving Hex dependencies...
Resolution completed in 0.018s
New:
  jason 1.4.0
* Getting jason (Hex package)
```

We can see that a new directory for our newly installed package was created in `project_1/deps/jason`. This folder will contain a `lib` directory with all the modules; if the package were written in Elixir, modules would be defined as `modulename.ex` files. We will also typically get the following files:

- A `README.md` file containing the package's documentation.
- A `LICENSE` file containing the package's license and distribution agreements.
- A `mix.exs` file containing the dependencies and configuration for the package.
- A `CHANGELOG.md` containing a chronological set of enhancements, fixes and security improvements made over time.

We can include a dependency in our project by using the `require` or `import` directives:

Code

```
defmodule Project1 do
  @moduledoc """
  This is a module containing functions to explore the Elixir language.
  """
  # Import Jason package
  import Jason

  @doc """
  Print hello world.
  """
  def main do
    IO.puts("Hello World")
  end
end
```

If we then compile our project again, the `jason` package will also be compiled:

Code

```
mix run -e "Project1.main()"
```

Output

```
==> jason
Compiling 10 files (.ex)
Generated jason app
==> project_1
Hello World
```

Each package run and compilation is denoted by the thick arrow `==>`.

This process is only executed once since, upon the first compilation, Erlang BEAM binaries will be built inside the `project_1/_build/dev/lib/jason/ebin` directory.

3. Compiling & running

In order to run an Elixir project, we will need to compile it first. The good news is that if we're using Mix, we don't have to worry about that since it's automatically handled when running.

We can create a new PowerShell instance, head to our project's main directory, and include the following:

Code

```
mix run -e "Project1.main()"
```

Mix first compiles the `Project1` module and then runs `myfun1()`.

Output

```
Compiling 1 file (.ex)
Generated project_1 app
Hello World
```

And voila, we've written, compiled, and run our first Elixir project.

Upon compiling and running, the compiler will create a new folder called `_build`. This folder will at first contain one subfolder called `dev`. This subfolder contains the development build artifacts. This includes compiled Elixir code, compiled Erlang code, and compiled NIFs (*Native Implemented Functions*) if our project uses them.

§

Commenting

Elixir provides three main ways to add comments and docstrings to a project:

- Single-line comments: Can be introduced using the hash `#` symbol.
- Module documentation: Can be introduced using `@moduledoc`. We'll discuss it further when we get to the modules section.
- Function documentation: Can be introduced using `@doc`. We'll discuss it further when we get to the functions section.

1. Single-line comment

We can introduce a single-line comment using the following syntax:

Code

```
def main do
  # This is a single-line comment
end
```

§

Variables

As we have seen, we can define a variable using the following syntax:

Code

```
def main do
  # Define an integer
  myint = 13

  # Define a floating-point number
  myfloat = 3.1416

  # Define a string
  mystring = "Hello World"

  # Define a boolean
  mybool = true
end
```

It's important to remember that declared variables must be used eventually; else, the compiler will let us build our application but return a warning upon compilation. If we're not using a variable we declared, we can prefix it with an underscore `_` to denote a placeholder variable.

Also, we don't need to define a variable's data type in Elixir since it's dynamically typed.

All variables in Elixir are immutable, meaning once a variable is defined in memory, it cannot change its state. This does not mean we cannot rebind a variable; we can throw away a variable reference to a given type and rebind it to a new reference.

Let us explain this with an example:

Code

```
def main do
  # Variable rebinding
  myvar1 = 3
  IO.puts("Original variable: #{myvar1}")
  myvar1 = myvar1 + 3
  IO.puts("New variable with same label: #{myvar1}")
end
```

Output

```
Original variable: 3
New variable with same label: 6
```

Let us explain in detail:

- We first declare a variable `myvar1` and assign it an integer value `3`.
- At compile time, the actual value is saved in memory at a given address, while the variable name or label is set as a reference (*points to*) to this address. The binding is created in the current process's environment or scope.
- When we perform an operation that appears to "*modify*" the value, Elixir does not change the original value in memory. Instead, it creates a new value resulting from the operation and stores it in a different memory location. The original value remains unchanged.
- If we bind a variable to a new value, the variable now points to the memory address of the new value. The old value is still in memory but may become eligible for garbage collection if no other variables reference it.

- Since Elixir runs on the Erlang Virtual Machine, it takes advantage of Erlang's lightweight concurrency and preemptive scheduling. Each process has its own isolated memory space, and variables are only accessible within the process they are defined in. This means that data sharing between processes must be done via message-passing, further reinforcing immutability.

Unfortunately, there is no direct way to prove immutability, as Elixir is a high-level programming language. Usually, these languages do not have readily available methods to interact with hardware, in this case, getting memory addresses and comparing the address of an original value to the address of a new value with the same variable name (*which in theory should be different*).

§

Printing

There are three main methods we can use to print to `stdout`:

- `IO.puts`: Takes a string as its argument and outputs that string to `stdout` followed by a newline character.
- `IO.write`: Writes data to an output device, such as `stdout` or a file, without including a new line character as in `IO.puts`. It takes two arguments: the data to be written and the output device to write to. The default output device is `stdout`.
- `IO.inspect`: Takes a value as its argument and outputs the internal representation of that value (*an Elixir expression*) to `stdout`.

1. IO.puts

Elixir's syntax is similar to Ruby's; one of the similarities is that we don't require using parenthesis `()` in statements. This would provide us with two different ways to print to `stdout`:

Code

```
def main do
  # Define an integer
  myint = 13

  # Print with parenthesis
  IO.puts(myint)

  # Print without parenthesis
  IO.puts myint
end
```

Output

```
13
13
```

Note that we cannot print multiple variables inside one print statement by simply using a comma `,`, but we need to use either string interpolation or string concatenation.

1.1 Using string interpolation

We can also use string interpolation to print a string and a variable. For this, we enclose our variable in curly brackets prepended by a hash sign `#{}` inside our print statement:

Code

```
def main do
  # Define an integer
  myint = 13

  # Print with string interpolation
  IO.puts("The number is: #{myint}")
end
```

Output

```
The number is: 13
```

For multiple variables, we can simply use the following syntax:

Code

```
def main do
  myint1 = 13
  myint2 = 14
  IO.puts("Numbers: #{myint1}, #{myint2}")
end
```

Output

```
Numbers: 13, 14
```

1.2. Using string concatenation

We can also print multiple values using one statement with string interpolation. String interpolation requires us to have the same types, `string`, for all values to print to `stdout`.

If we have different data types, we must cast them when printing them:

Code

```
def main do
  # Define an integer
  myint = 13

  # Print with string concatenation
  IO.puts("The number is: " <> Integer.to_string(myint))
end
```

Output

```
The number is: 13
```

For multiple variables, we can simply use the following syntax:

Code

```
def main do
  myint1 = 13
  myint2 = 14
  IO.puts("Numbers: " <> Integer.to_string(myint1) <> ", " <> Integer.to_string(myint2))
end
```

Output

```
Numbers: 13, 14
```

2. IO.write

This method has properties similar to `IO.puts`, in that it expects a string as `input` and also accepts string interpolations as part of its syntaxis:

Code

```
def main do
  # Define atoms
  myvar1 = :moon
  myvar2 = :sun

  # Write without new line characters
  IO.write("Under the light of the #{myvar1}")
  IO.write("Without the heat of the #{myvar2}")
end
```

Output

```
Under the light of the moonWithout the heat of the sun
```

However, a new line character can be added at the end of each statement, which would effectively provide a behavior equal to `IO.puts`:

Code

```
def main do
  # Define atoms
  myvar1 = :moon
  myvar2 = :sun

  # Write including new line characters
  IO.write("Under the light of the #{myvar1}\n")
  IO.write("Without the heat of the #{myvar2}\n")
end
```

Output

```
Under the light of the moon
Without the heat of the sun
```

In this segment, we will only cover IO.write as a printing to `stdout` method.

3. IO.inspect

This method is suited specifically when we're trying to visualize the internal representation of a given value exactly as it was originally written.

One great example would be if we have a `list` that we would like to print to `stdout` while maintaining its structure:

Code

```
def main do
  mylist = [1, 2, 3, 4, 5]

  # Inspect
  IO.inspect(mylist, label: "Inspect")

  # Puts
  IO.puts("Puts: #{mylist}")
end
```

Output

```
Inspect: [1, 2, 3, 4, 5]
Puts: ☺☻♥♦♣
```

As we can see from the second output, calling `IO.puts` on our list will result in gibberish. This is because the `IO.puts` function expects a `string` as an argument, not a `list`. When we pass a list to `IO.puts`, Elixir will try to convert the list to a string, but the result may not be what we expect.

Another important thing to note is that we used the `label` option for `IO.inspect`. In this method, we cannot use string interpolation `#{}`; thus, we cannot append additional strings as we did with `IO.puts`. We need to use the `label` property to achieve this, which will behave the same as with `IO.puts`.

It may happen that when using `IO.inspect`, the printed internal representation does not match the actual definition (*types might get interpreted differently, e.g., printing chars instead of integers inside a list*). If we do not get what we were looking for, we can explicitly define the type we're defining inside our sequence:

Code

```
def main do
  mylist = [1, 2, 3, 4]
  IO.inspect(mylist, label: "Inspect explicitly defining types inside sequence", charlists:
:as_lists)
end
```

Output

```
Inspect explicitly defining types inside sequence: [1, 2, 3, 4]
```

If we still want to use `IO.puts` to print a sequence type such as a `list`, we can include an alternative method, `inspect`, inside the string interpolation statement:

Code

```
def main do
  # Puts with inspect method inside string interpolation
  mylist = [1, 2, 3, 4, 5]
  IO.puts("Puts with inspect method: #{inspect mylist}")
end
```

Output

```
Puts with inspect method: [1, 2, 3, 4, 5]
```

§

Data types

Elixir has multiple data types we can make use of. As we venture into sequence data types, it's worth pointing out that Elixir uses 0-based indexing, whereas Erlang uses 1-based indexing. This will be important as we move on to sequence data types.

We have eight main native data types:

- Integers
- Floats
- Strings
- Atoms
- Lists
- Tuples

- Ranges
- Maps

1. Integers

Integers do not have a limit on their size. They can be defined by simply assigning the value to a variable.

We can perform all of the expected arithmetic operations on integer values:

Code

```
def main do
  # Define ints
  myint1 = 14
  myint2 = 7
  IO.puts("Ints: #{myint1}, #{myint2}")

  # Arithmetic operations
  IO.puts("Addition: #{myint1 + myint2}")
  IO.puts("Subtraction: #{myint1 - myint2}")
  IO.puts("Product: #{myint1 * myint2}")
  IO.puts("Division returning int or float: #{myint1 / myint2}")
  IO.puts("Division returning int: #{div(myint1, 8)}")
  IO.puts("Remanent: #{rem(myint1, myint2)}")
  IO.puts("Exponentiation: #{myint1**2}")
end
```

Output

```
Addition: 21
Subtraction: 7
Product: 98
Division returning int or float: 2.0
Division returning int: 1
Remanent: 0
Exponentiation: 196
```

Note that the `div` operator returns a truncated integer value, meaning the result is not rounded; only the integer part is taken.

We can also perform logical comparisons between integer values. We'll discuss this in detail when we get to logical operators.

2. Floats

Floats are accurate to 16 decimal places. They can be defined either using decimal or scientific notation:

Code

```
def main do
  # Define float
  myfloat1 = 1200.0
  IO.puts("Float: #{myfloat1}")

  # Define float using scientific notation
  myfloat2 = 12.0e2
  IO.puts("Float Scientific: #{myfloat2}")
end
```

Output

```
Float: 1.2e3
Float Scientific: 1.2e3
```

And can also be operated on using the same arithmetic and logical operators we used above for integer values.

3. Strings

We have already defined some strings in previous sections. It's worth pointing out that strings are defined using double quotes `""`.

We can define a string and perform some useful operations:

Code

```elixir
def main do
  # Declare some strings
  mystr1 = "Elixir"
  mystr2 = "is"
  mystr3 = "awesome"

  # Get length of string
  IO.puts("Len: #{String.length(mystring)}")

  # Concatenate strings
  mystr4 = mystr1 <> " " <> mystr2 <> " " <> mystr3 <> "."
  IO.puts("Concatenated: #{mystr4}")

  # Convert to uppercase
  mystr5 = String.upcase(mystr4)
  IO.puts("Upper: #{mystr5}")

  # Go back to lowercase (downcase)
  IO.puts("Lower: #{String.downcase(mystr5)}")

  # Capitalize first characters
  IO.puts("Capitalize: #{String.capitalize(mystr5)}")

  # Check if strings are equal (value and datatype ===)
  IO.puts("Strings are equal?...#{mystr4 === mystr5}")

  # Check if string contains another string
  targetstring = "awesome"
  IO.puts("String contains #{targetstring}?: #{String.contains?(mystr4, targetstring)}")

  # Return first character
  IO.puts("First character: #{String.first(mystr5)}")

  # Index a character inside a string
  IO.puts("Second character: #{String.at(mystr5, 1)}")

  # Get substring using slice (from index 0 to 6)
  IO.puts("String slice: #{String.slice(mystr5, 0, 6)}")

  # Reverse string
  IO.puts("Reverse string: #{String.reverse(mystr5)}")
end
```

Output

```
Len: 5
Concatenated: Elixir is awesome.
Upper: ELIXIR IS AWESOME.
Lower: elixir is awesome.
Capitalize: Elixir is awesome.
Strings are equal?...false
String contains awesome?: true
First character: E
Second character: L
String slice: ELIXIR
Reverse string: .EMOSEWA SI RIXILE
```

4. Atoms

In Elixir, an atom is a constant whose value is its own name. An atom is written by starting with a colon `:` followed by a sequence of characters. For example, `:hello`, `:world`, and `:Mexico` are all atoms.

Atoms might seem an unusual type; only some languages provide equivalents:

- **Erlang:** Atoms are also a fundamental data type in Erlang, and they're called the same.
- **Lisp:** Atoms are a fundamental data type in Lisp and its dialects, such as Scheme and Clojure.
- **Prolog:** Atoms are used to represent constant values in Prolog, which is a logic programming language.
- **Ruby:** Atoms are also called **symbols** in Ruby, and they are used to represent static, immutable values such as keys in hashes.
- **Python:** Python has a similar concept to atoms called **string interning**, where identical string literals are reused in memory.

Atoms are useful when we're trying to represent constant values that are used throughout a system, e.g.:

- **Naming keys in maps:** Atoms can be used to name keys in maps, providing a more descriptive and readable way to access data in a map. For example, we might use the atom `:name` to represent the name key in a user record.
- **Representing states or statuses:** Atoms can represent states or statuses in a system, such as `:running` or `:stopped`. This can make it easier to reason about a system's behavior and write code that responds to specific states or events.
- **Defining configuration values:** Atoms can also be used to define configuration values in a system, such as `:debug` or `:log_level`. This allows us to easily configure a system's behavior without changing code.
- **Naming functions or modules:** Atoms can also be used to name functions or modules in Elixir, providing a more descriptive and readable way to refer to these constructs in code.

```
def main do
  # Declare atom without space
  myatom1 = :Hello

  # Declare atom with space
  myatom2 = :"Mexico City"

  # Print atoms
  IO.puts("#{myatom1}, #{myatom2}")

  # Check types
  IO.puts("#{is_atom(myatom1)}, #{is_atom(myatom1)}")
end
```

Output

```
Hello, Mexico City
true, true
```

Let us look at an example where we're trying to represent algebraic operations in a user-defined function, `arithmeticCalculator`:

Code

```
def main do
  operation = :add
  num1 = 7
  num2 = 14
  result = arithmeticCalculator(operation, num1, num2)
  IO.puts("The result of #{operation} #{num1} and #{num2} is #{result}")
end

def arithmeticCalculator(operation, num1, num2) do
  case operation do
    :add -> num1 + num2
    :subtract -> num1 - num2
    :multiply -> num1 * num2
    :divide -> num1 / num2
  _ -> {:error, "Invalid operation"}
  end
end
```

Output

```
The result of add 7 and 14 is 21
```

In this example, we use atoms to represent different arithmetic operations in a simple calculator. The `arithmeticCalculator` function takes the desired operation and two numbers as arguments and returns the result of the operation. If an invalid operation is provided, it returns an error tuple with the `"Invalid operation"` message.

5. Lists

Elixir provides many functions and operators for working with lists, often used to represent data sequences and functional programming patterns. A `list` is an ordered collection of elements represented using square brackets `[]` with commas `,` as value separators. Lists are immutable, meaning their contents cannot be changed once created.

Lists can contain elements of any type, including other lists and nested data structures. However, each individual list can only contain elements of a single type at a time. This is because Elixir lists are implemented as singly linked lists, where each element points to the next element.

We can define a `list` using the following syntax:

Code

```elixir
def main do
  # Define a list
  mylist = [1, 2, 3, 4, 5]
  IO.inspect(mylist, label: "A list of integer values")
end
```

Output

```
A list of integer values: [1, 2, 3, 4, 5]
```

Lists have multiple methods we can use to interact with its values and structure:

Code

```elixir
def main do
  # Define a list
  mylist = [1, 2, 3, 4, 5]
  IO.inspect(mylist, label: "A list of integer values")

  # Concatenating lists

  # Define two lists of atoms
  mylist1 = [:jupyter, :neptune, :mars]
  mylist2 = [:earth, :sun, :moon]

  # Concatenate the two lists
  mylist3 = mylist1 ++ mylist2
  IO.inspect(mylist3, label: "A concatenated list")

  # Insert elements using List.insert
  mylist3 = List.insert_at(mylist3, 6, :saturn)
  IO.inspect(mylist3, label: "Inserted atom at index 6 using insert_at")

  # Subtract items from list
  mylist4 = mylist3 -- mylist1
  IO.inspect(mylist4, label: "Subtracting lists")

  # Verify if item is on list
  IO.puts("Verify if item is in list: #{:neptune in mylist4}")

  # Get head (index 0) and tail (index 1 to -1)
  [head | tail] = mylist4
  IO.puts("Head and tail of list: #{inspect head}, #{inspect tail}")

  # Remove items from list using value
  IO.puts("Remove item using value: #{List.delete(mylist4, :saturn)}")

  # Remove items from list using index
  IO.puts("Remove item using index: #{List.delete_at(mylist4, 0)}")

  # Get first item from list
  IO.puts("Get first item using 'first': #{inspect List.first(mylist4)}")

  # Get last item from list
  IO.puts("Get last item using 'last': #{inspect List.last(mylist4)}")
end
```

Output

```
A list of integer values: [1, 2, 3, 4, 5]
A concatenated list: [:jupyter, :neptune, :mars, :earth, :sun, :moon]
Inserted atom at index 6 using insert_at: [:jupyter, :neptune, :mars, :earth, :sun, :moon,
:saturn]
Subtracting lists: [:earth, :sun, :moon, :saturn]
Verify if item is in list: false
Head and tail of list: :earth, [:sun, :moon, :saturn]
Remove item using value: [:earth, :sun, :moon]
Remove item using index: [:sun, :moon, :saturn]
Get first item using 'first': :earth
Get last item using 'last': :saturn
```

A detail worth mentioning is that if we try to remove a value that does not exist (*either by index or label*), the compiler will not return an error.

We can define a list containing multiple `key` : `value` tuples, for example, a list of atoms:

Code

```
def main do
  mylist = [home: :earth, closest: :mercury, farthest: :neptune]
  IO.puts("Key - value pairs in list: #{inspect mylist}")
end
```

Output

```
Key - value pairs in list: [home: :earth, closest: :mercury, farthest: :neptune]
```

We can also include different data types as values if we want to.

Additionally, we can also iterate over lists using multiple methods, which we'll discuss in more detail when we get to iterators.

6. Tuples

Tuples in Elixir are ordered collections of values that can contain one or more different data types. They are not meant to hold multiple values or iterate over them; when we access an element in a tuple by index, Elixir needs to traverse the entire tuple to find the element, which can be slow and inefficient for large tuples. In contrast, arrays and lists allow us to access elements by index in constant time, making them much faster for iterative operations.

Unlike other languages, we use curly brackets `{}` to define a tuple in Elixir:

Code

```
def main do
  # Define a tuple
  mytuple = {1, 2, 3.0, :mars}
  IO.inspect(mytuple, label: "A tuple containing different types")
end
```

Output

```
A tuple containing different types: {1, 2, 3.0, :mars}
```

Tuples have multiple methods we can use to interact with its values and structure:

Code

```elixir
def main do
  # Define a tuple
  mytuple = {1, 2, 3.0, :mars}

  # Append value
  mytuple = Tuple.append(mytuple, "hello")
  IO.inspect(mytuple, label: "A tuple with appended value")

  # Index tuple
  IO.puts("Indexing on 2: #{elem(mytuple, 2)}")

  # Get tuple size
  IO.puts("Tuple size: #{tuple_size(mytuple)}")

  # Remove item at index
  mytuple = Tuple.delete_at(mytuple, 0)
  IO.inspect(mytuple, label: "A tuple with first value removed")

  # Insert item at index
  mytuple = Tuple.insert_at(mytuple, 1, :jupyter)
  IO.inspect(mytuple, label: "A tuple with atom inserted at index 1")

  # Create a tuple with repeating values (duplicate neptune seven times)
  mytuple2 = Tuple.duplicate(:neptune, 7)
  IO.inspect(mytuple2, label: "A tuple with repeating atoms")

  # Pattern matching with tuples (assign multiple variables in a single line)
  {x, y, z} = {13, 21, 3}
  IO.puts("Multiple variable assignment: #{x}, #{y}, #{z}")
end
```

Output

```
A tuple with appended value: {1, 2, 3.0, :mars, "hello"}
Indexing on 2: 3.0
Tuple size: 5
A tuple with first value removed: {2, 3.0, :mars, "hello"}
A tuple with atom inserted at index 1: {2, :jupyter, 3.0, :mars, "hello"}
A tuple with repeating atoms: {:neptune, :neptune, :neptune, :neptune, :neptune, :neptune,
:neptune}
Multiple variable assignment: 13, 21, 3
```

7. Ranges

A `range` is a data type representing an interval of values. A range can also be used as an input to various iterators, such as `Enum.each/2`, `Enum.map/2`, `Enum.filter/2`, and `Enum.reduce/3`, which we'll review in more depth as we discuss iterators.

Ranges represent a sequence of zero, one, or many ascending or descending integers with a common difference called a step. They are always inclusive, and may have a custom step defined.

A simple range can be defined as follows:

Code

```
def main do
  myrange = 1..100
end
```

We can perform multiple operations on a `range` using `Enum`:

Code

```
def main do
  # Declare a range
  myrange = 1..20

  # Perform operations on a range

  # Enumerate the elements in the range
  enumerated_elements = Enum.to_list(myrange)
  IO.inspect(enumerated_elements)

  # Check if a value is within the range
  is_in_range = Enum.member?(myrange, 5)
  IO.inspect(is_in_range)

  # Sum the elements in the range
  sum_of_elements = Enum.sum(myrange)
  IO.inspect(sum_of_elements)

  # Get the range length
  range_length = Enum.count(myrange)
  IO.inspect(range_length)
end
```

Output

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
true
210
20
```

8. Maps

A `map` is a data structure that allows us to associate keys with values. Maps are created using the `%{}` syntax, where a comma separates each `key` - `value` pair.

Maps in Elixir are similar to dictionaries in Python in that both data structures allow us to associate keys with values. They are both unordered collections of key-value pairs. However, there are also some key differences:

- **Syntax:** Maps in Elixir are created using the `%{}` syntax, while dictionaries in Python are created using the `{}` syntax. Also, `key` - `value` assignments in Elixir are done using the `=>` operator, while in Python, this is achieved using the colon `:` operator.
- **Mutability:** Maps in Elixir are immutable, which means that once a map is created, it cannot be modified. In contrast, dictionaries in Python are mutable, meaning we can add, remove, and modify items in a dictionary after it has been created.
- **Key types:** In Elixir, keys in a `map` can be any term, including atoms, strings, and integers. In Python, keys in a dictionary can be any hashable object, such as strings, integers, and tuples.
- **Error handling:** Accessing a key that does not exist in a map in Elixir will raise a `KeyError` exception in Python. In Elixir, we can use the `Map.get/3` function to safely access a key and return a default value if the key does not exist.

We can define a `map` using the following syntax:

Code

```elixir
def main do
  # Define a map using strings
  mymap = %{"Charles" => "Dickens",
            "Oscar" => "Wilde",
            "Jane" => "Austen",
            "Marcel" => "Proust"}

  IO.puts("A map of strings: #{inspect mymap}")
end
```

Output

```
A map of strings: %{"Charles" => "Dickens", "Jane" => "Austen", "Marcel" => "Proust", "Oscar" =>
"Wilde"}
```

We can also use atoms instead of strings:

Code

```elixir
def main do
  # Define a map using atoms
  mymap2 = %{charles: "Dickens",
             oscar: "Wilde",
             jane: "Austen",
             marcel: "Proust"}

  IO.puts("A map of atoms: #{inspect mymap2}")
end
```

Output

```
A map of atoms: %{Charles: "Dickens", Jane: "Austen", Marcel: "Proust", Oscar: "Wilde"}
```

Maps have multiple methods we can use to interact with its keys, values, and structure:

Code

```
def main do
  # Methods for map of strings

  # Index by key using get
  IO.puts("Index by key using get (Charles): #{Map.get(mymap1, "Charles")}")

  # Index by key using square brackets
  IO.puts("Index by key using square brackets (Charles): #{mymap1["Charles"]}")

  # Insert element using put_new
  mymap3 = Map.put_new(mymap1, "Virginia", "Woolf")
  IO.puts("Insert element using put_new (Virginia => Woolf): #{inspect mymap3}")

  # Methods for map of atoms

  # Index by key using dot . (Cannot use this method if we have upper-case keys)
  IO.puts("Index by key using dot . (Charles): #{inspect mymap2.charles}")
end
```

Output

```
Index by key using get (Charles): Dickens
Index by key using square brackets (Charles): Dickens
Insert element using put_new (Virginia => Woolf): %{"Charles" => "Dickens", "Jane" => "Austen",
"Marcel" => "Proust", "Oscar" => "Wilde", "Virginia" => "Woolf"}
Index by key using dot . (Charles): "Dickens"
```

9. Checking types

There's no direct way to get a variable type in Elixir/Erlang. However, we sometimes want to know the type of a variable to act accordingly. We can check if a variable is of a given type by using the `is_type` method, where `type` must be substituted with the appropriate type name:

Code

```elixir
def main do
  # Declare variables
  myint = 20
  myfloat = 20.13
  mystring = "Hello"
  mybool = true

  # Check if type is int
  IO.puts("Is #{myint} an integer number?: #{is_integer(myint)}")

  # Check if type is float
  IO.puts("Is #{myfloat} a floating-point number?: #{is_float(myfloat)}")

  # Check if type is string
  IO.puts("Is #{mystring} a string?: #{is_bitstring(mystring)}")

  # Check if type is bool
  IO.puts("Is #{mybool} a boolean?: #{is_boolean(mybool)}")
end
```

Output

```
Is 20 an integer number?: true
Is 20.13 a floating-point number?: true
Is Hello a string?: true
Is true a boolean?: true
```

§

Operators

Elixir provides comparison, arithmetic, and logical operators for any type.

1. Comparison operators

Elixir provides all the comparison operators we would expect from any language, plus some additional operators referring to data type comparisons:

Operator	Description
`===`	Value and data type are equal to
`==`	Value is equal to
`!==`	Value and data type are not equal to
`!=`	Value is not equal to
`>`	Value is greater than
`<`	Value is less than
`>=`	Value is greater than or equal to
`<=`	Value is less than or equal to

Table 1: Basic Comparison Operators

We can make some comparisons:

Code

```elixir
def main do
  # Define int and float
  myint = 7
  myfloat = 7.0

  # Compare values
  IO.puts("===: #{myint === myfloat}")
  IO.puts("==: #{myint == myfloat}")
  IO.puts("!==: #{myint !== myfloat}")
  IO.puts("!=: #{myint != myfloat}")
  IO.puts(">: #{myint > myfloat}")
  IO.puts("<: #{myint < myfloat}")
  IO.puts(">=: #{myint >= myfloat}")
  IO.puts("<=: #{myint <= myfloat}")
end
```

Output

```
===: false
==: true
!==: true
!=: false
>: false
<: false
>=: true
<=: true
```

2. Arithmetic operators

Elixir brings in the usual algebraic operators we would expect from any language:

Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Product
`/`	Division (returns result as `float`)
`div`	Division (returns truncated result as `int`)
`rem`	Reminder
`**`	Exponentiation

TABLE 2: BASIC ARITHMETIC OPERATORS

Additionally, Elixir also has special methods for data types such as `lists`:

Operator	Description
`++`	Concatenate two lists
`--`	Subtract items from lists

TABLE 3: ARITHMETIC OPERATORS FOR LISTS

3. Logical operators

As we mentioned, Elixir provides a boolean type, `bool`. Logical operators evaluate two conditions and output a boolean value depending on the comparison.

There are three main logical operators we can use:

Operator	Description
`and`	Condition A and B are true
`or`	Condition A or B are true
`not`	Invert the boolean value

TABLE 4: LOGICAL OPERATORS

We can make logical comparisons by using the following syntax:

CODE

```
def main do
  # Define integers
  myint1 = 14
  myint2 = 15
  myint3 = 29

  # Compare conditions
  IO.puts("Less than: #{(myint1 <= myint2) and (myint1 <= myint3)}")
  IO.puts("Greater than (or): #{(myint2 >= myint1) or (myint2 >= myint3)}")
  IO.puts("Greater than (or, not): #{(myint2 >= myint1) and not(myint2 >= myint3)}")
end
```

Output

```
Less than: true
Greater than (or): true
Greater than (or, not): true
```

Naturally, logical comparisons also apply to boolean values directly:

Code

```
def main do
  # Compare boolean values
  mybool1 = true
  mybool2 = false

  # Compare conditions
  IO.puts("True and True: #{mybool1 and mybool1}")
  IO.puts("True and False: #{mybool1 and mybool2}")
  IO.puts("True or True: #{mybool1 or mybool1}")
  IO.puts("True or False: #{mybool1 or mybool2}")
  IO.puts("Not True: #{not(mybool1)}")
  IO.puts("Not False: #{not(mybool2)}")
end
```

Output

```
True and True: true
True and False: false
True or True: true
True or False: true
Not True: false
Not False: true
```

§

Control flow

As in other languages, control flow in Elixir can be achieved using conditional constructs. Elixir provides three main ways to do so:

- `if, else` construct: The most common method for evaluating one condition.
- `if, unless`: The most common method for evaluating one condition with a fallback to evaluating an alternative condition if the first condition is `false`.
- `case` construct: The most common method for evaluating multiple conditions (*cases*) based on pattern matching.
- `cond` construct: The most common method for evaluating multiple conditions until one is true based on boolean expressions.

This might seem a bit confusing at first; let's work it out with examples to better understand the differences and use cases between the three.

1. Using if, else

`if` is used to execute a block of code if a boolean expression is true, and `else` is used to execute a block of code if the boolean expression under `if` is false. This construct works specifically for testing only one condition.

We can define an `if else` test as follows:

Code

```elixir
def main do
  # Define some variables
  myvar1 = 14
  myvar2 = 196

  # If else
  if :math.sqrt(myvar2) == myvar1 do
    IO.puts("#{myvar1} is the square root of #{myvar2}.")
  else
    IO.puts("#{myvar1} is not the square root of #{myvar2}.")
  end
end
```

Output

```
14 is the square root of 196.
```

2. Using unless, else

Conversely, we can use `unless` to execute a code block if a boolean expression is false, as with `if else`, this construct specifically tests one main condition.

We can define an `unless else` test as follows:

Code

```elixir
def main do
  # Define some variables
  myvar1 = 14
  myvar2 = 197

  # Unless else
  unless (:math.sqrt(myvar2) == myvar1) and (rem(myvar2, 2) == 0) do
    IO.puts("Either #{myvar1} is not square root of #{myvar2} or #{myvar2} is an odd number.")
  else
    IO.puts("#{myvar1} is the square root of #{myvar2} and #{myvar2} is an even number.")
  end
end
```

Output

```
Either 14 is not square root of 197 or 197 is an odd number.
```

Here, we used the intersection of two conditions to evaluate to one final boolean value. Both conditions had to evaluate to `true`; else, it evaluates to `false`.

3. Using cond

`cond` evaluates a series of boolean expressions in order and executes the code block corresponding to the first `true` expression. Each expression in a `cond` statement must evaluate to either `true` or `false` and can include any valid Elixir expression.

A `cond` construct is equivalent to `else if` clauses in many imperative languages like Python.

We can continue with our previous example and define a `cond` test as follows:

Code

```
def main do
  # Define some variables
  myvar1 = 14
  myvar2 = 196

  cond do
    # First case
    (:math.sqrt(myvar2) == myvar1) and not(rem(myvar2, 2) == 0) -> IO.puts("Case 1: #{myvar1} is
the square root of #{myvar2}.")

    # Second case
    rem(myvar2, 2) == 0 and not(:math.sqrt(myvar2) == myvar1) -> IO.puts("Case 2: #{myvar2} is an
even number.")

    # Third dcase
    (:math.sqrt(myvar2) == myvar1) and (rem(myvar2, 2) == 0) -> IO.puts("Case 3: #{myvar1} is the
square root of #{myvar2} and #{myvar2} is an even number.")

    # Fourth case
    not(rem(myvar2, 2) == 0) and not(:math.sqrt(myvar2) == myvar1) -> IO.puts("Case 4: #{myvar1}
is not the square root of #{myvar2}, and #{myvar2} is an odd number.")

    # Default case
    true -> IO.puts("Case 5: There was a problem since none of the conditions were met.")
  end
end
```

Output

```
Case 3: 14 is the square root of 196 and 196 is an even number.
```

What we're doing here is defining 4 cases where:

- We declare the left side of the case as our condition.
- We insert a right arrow `->` to denote the output if a given condition is met.

- We declare the right side of the case as our output.

Additionally:

- All cases are exclusive.
- The set of cases covers all the possible options by making use of `and` and `not` (*they are exhaustive*) (*we could also change equality signs for inequality signs and play a little bit with our statements, but the point is made*).
- We set a default statement that will always evaluate to `true`. This statement should always be at the end of our set of cases (*order matters*).

Below are some important things to remember when using `cond`:

- As mentioned above, the first condition to evaluate to `true` will be returned, so we need to specify the order of our conditions wisely, or else make sure we build them appropriately using the corresponding logical constructs (*`and`*, *`or`*, *`not`*). If we're working with multiple upper and lower limits (`>=`, `<=`), we need to define those wisely and in the correct order too.
- Although we have a default statement that will always evaluate to true, we need to make sure we're being exhaustive with our conditions. This is always good practice and will make our life easier when debugging.

4. Using case

Cases behave like `C` switches, Rust `match` statements, and Bash `case` statements. `case` is used to match a value or expression against a set of patterns and execute a block of code for the first pattern that matches.

Each pattern in a `case` expression can include guards (*additional conditions that must be true for the pattern to match*) and variables bound to parts of the matched value.

We can define a `case` test as follows:

Code

```
def main do
  # Define a target variable
  myvar1 = 14

  # Define cases
  case myvar1 do
    14 -> IO.puts("Number is 14")
    15 -> IO.puts("Number is 15")
    16 -> IO.puts("Number is 16")
    _ -> IO.puts("No match among options for number")
  end
end
```

Output

```
Number is 14
```

Whenever we're specifying cases, it's always good practice to specify a default case in the event that no other is matched. We do this by using the underscore `_` symbol.

Of course, this is a very simple case, but we might want to implement a collection of cases when, for example, we're expecting a user input or a known return value from a given process; there are countless applications for using `case`.

§

Pipes

For those already familiar with piping in `bash` or `zsh`, pipes are convenient syntactical constructs for chaining together functions in a way that makes the code more readable and easier to understand. Pipes allow us to pass the output of one function as the input to another function without the need for nested function calls or temporary variables. This results in a more efficient use of resources and generally more performant code.

Although `bash` is mostly defined as a procedural language, pipes are common in functional languages; they go well with the functional programming philosophy.

We can pipe in Elixir by using the `|>` operator:

Code

```
def main do
  # Define a string
  mystring = "awesome are pipes"
  IO.puts("Original String: #{mystring}")

  # Transform a string using pipes
  mystring2 =
    mystring
    |> String.split()
    |> Enum.reverse()
    |> Enum.join(" ")
    |> String.capitalize()

  IO.puts("Transformed String: #{mystring2}")
end
```

Output

```
Original String: awesome are pipes
Transformed String: Pipes are awesome
```

§

Iterators

In Elixir, we can iterate over structures using multiple methods, although following functional programming best practices, there are three main approaches we can use:

- Enumeration of collections
- Recursion
- Higher-order functions

Elixir also has constructs like `for` and `while` loops, which can be used for more complex looping scenarios. However, functional programming paradigms emphasize recursion, enumeration of collections, and higher-order functions over loops, as they provide a more expressive and composable way to work with data.

Let us look at some examples:

1. Using Enum

`Enum` is a module that provides a set of functions for working with enumerable collections, which are collections that can be iterated over.

The `Enum` module includes functions for common operations on collections, such as filtering, mapping, reducing, and sorting. The `Enum` module functions are designed to work with any collection that implements the Enumerable protocol, including lists, tuples, maps, ranges, and streams.

We can use `Enum` with a `list` as follows:

Code

```elixir
def main do
  # Define list of integer values
  mylist = [10, 20, 30, 40]

  # Enumerate items using Enum
  Enum.each mylist, fn item ->
    IO.puts(item)
  end
end
```

Output

```
10
20
30
40
```

2. Using recursion

As we mentioned earlier, functional languages encourage the use of recursion as a fundamental technique for solving problems.

Recursion is the process of solving a problem by breaking it down into smaller sub-problems of the same type and then solving each sub-problem recursively. The recursion stops when a base case is reached; a sub-problem that can be solved directly without further recursion.

There are several reasons why functional languages heavily use recursion:

- Functional languages strongly focus on immutability and avoiding side effects, making recursion a natural choice for iterating over data structures instead of using loops with mutable variables.
- Recursion is a declarative way of expressing algorithms and is often more concise and easier to understand than imperative code using loops and mutable variables.
- Recursion can be optimized by compilers and interpreters to make it as efficient as iterative solutions, and sometimes even more so. This is because recursion allows for tail call optimization, eliminating the

overhead of maintaining a call stack.

Recursion can sometimes be tricky and requires a different way of thinking about the problem we're trying to solve.

Let us try to exemplify using a diagram:

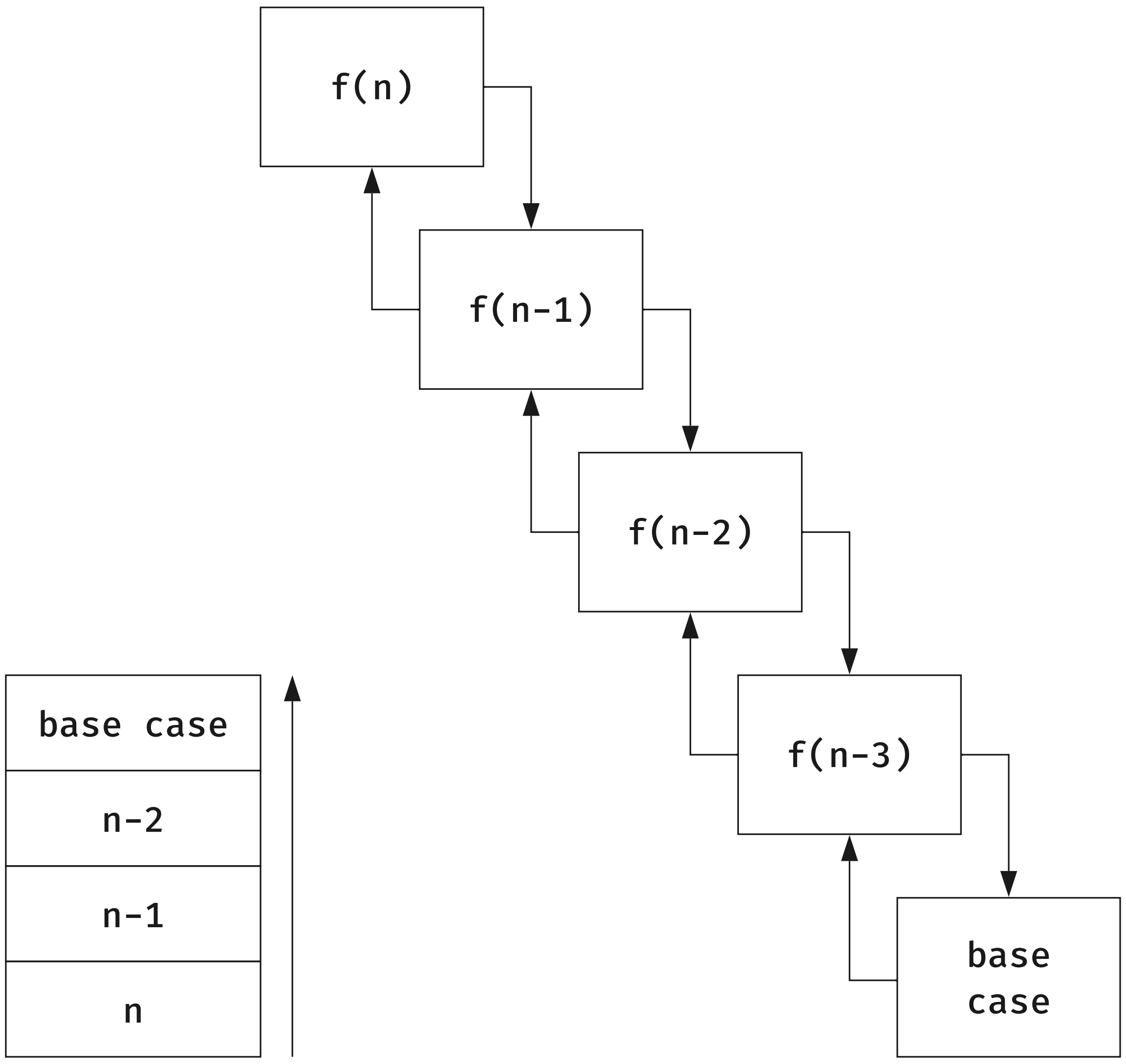

Figure 1: A Generic Recursive Process

- **We start by defining a base case:** The simplest version of the problem that can be solved without recursion. This serves as the stopping condition for the recursive calls.
- **We then define the recursive case:** The more complex version of the problem that can be broken down into smaller sub-problems. In the recursive case, the function calls itself with a smaller input until the base case is reached.
- **We iteratively reduce our problem to the base case:** When the base case is reached, the function returns a result, propagating up the call stack to the original function call.

One particular application of recursion is iteration. We can iterate over a list by using the following syntax:

Code

```
def main do
  # Define a list of words
  mylist = ["This", "is", "a", "list", "of", "strings"]

  # Call recursive function
  myfun(mylist)
end

# Recursive function
def myfun([head|tail]) do
  IO.puts("#{head}")

  # Calls itself with tail as reduced case
  myfun(tail)
end

# Base case (empty list)
def myfun([]), do: nil
```

Output

```
This
is
a
list
of
strings
```

Let us explain in more detail what's happening:

- We define a function that accepts a non-empty `list` and gets its `head` and `tail`.
- We define a base case list that accepts an empty `list` using the same function name as above.
- We define a `list` of strings `mylist`
- We call `myfun` with `mylist` as the argument.
- If the list is not empty, the `head` and `tail` are separated.
- The `head` is printed to `stdout`, and the `tail` is fed as our new argument.
- If and when our `tail` is empty, we cannot call the first `myfun` anymore since it's expecting a non-empty list, so the second definition of `myfun` is called, which returns `nil` (*empty value*).

In this example, we printed our words to `stdout`, but in a more complex implementation, we can do all sorts of things with the `head` that's being outputted.

§

Functions

In Elixir, a function is a self-contained unit of code that performs a specific task. Functions are defined using the `def` keyword, followed by the function name, any arguments that the function takes, and the body of the function enclosed in `do` and `end` statements, which contain the code to be executed when the function is called.

Function names are typically defined in snake_case notation (*all lowercase letters with words separated by underscores*). However, if we want to use a different syntax, we can do so, but we must not capitalize the first letter of the name since this is reserved for module definitions.

Additionally, we can define a function documentation string (*equivalent to a docstring in Python*) using the `@doc` keyword before our function definition:

Code

```
@doc """
Print hello world.
"""
def main do
  IO.puts("Hello World")
end
```

In Elixir, we have two main types of functions we can work with:

- Named functions
- Anonymous functions

1. Named functions

A named function is a function that has a defined name and can be called by that name from anywhere in the program.

Named functions can be of the following type:

- Without arguments
- With arguments
- With default arguments

In Elixir, there is no return statement as in other languages; we need to structure our code so the last statement executed is the return value.

1.1 Without arguments

A function without arguments does not accept arguments when calling it. We already defined a named function without arguments when we included `main` in our program.

The syntax is as follows:

Code

```
def main do
  # Call fun with return to stdout
  mynamedfun1()

  # Call fun with return to variable (assignment)
  myvalue2 = mynamedfun2()
  IO.puts("Value for z is: #{myvalue2}")
end

# Function with return to stdout and no arguments
def mynamedfun1 do
  x = 7
  y = 2
  IO.puts("Values for x and y are: #{x}, #{y}")
end

# Function with return to variable and no arguments
def mynamedfun2 do
  x = 7
  y = 2
  z = x * y
end
```

Output

```
Values for x and y are: 7, 2
Value for z is: 14
```

In both cases, the last statement will be returned:

- In the first case, to `stdout`.
- In the second case, to a variable.

1.2 With arguments

We can also define a named function that accepts arguments. If they are not included in the function call, we will get an error:

Code

```
def main do
  # Call fun with arguments
  myvalue3 = mynamedfun3(7, 2)
  IO.puts("Value for z is: #{myvalue3}")
end

# Function with return to variable and arguments
def mynamedfun3(x, y) do
  z = x * y
end
```

Output

```
Value for z is: 14
```

1.3 With default arguments

If we want to define default values for our arguments, we can do it so that the following will occur:

- **No arguments provided:** The two arguments are taken from defaults.
- **One argument provided:** The one(s) not provided is/are taken from defaults.
- **All arguments provided:** None is taken from defaults.

Code

```
def main do
  # Call fun with default arguments
  myvalue4 = mynamedfun4()
  IO.puts("Value for z is: #{myvalue4}")
end

# Function with return to variable and default arguments
def mynamedfun4(x \\ 7, y \\ 2) do
  z = x * y
end
```

Output

```
Value for z is: 14
```

2. Anonymous functions

In Elixir, anonymous functions are functions that do not have a name and are defined using the `fn` and `end` keywords. Anonymous functions can be assigned to variables, passed as arguments to other functions, and returned as results from functions.

2.1 A simple anonymous function

We can define an anonymous function using the following syntax:

Code

```
def main do
  # Define anonymous function that exponentiates a given value
  myfun = fn (x, y) -> x**y end

  # Define variables
  x = 7
  y = 2

  # Call anonymous function
  squared_result = myfun.(x, y)

  # Print result
  IO.puts("#{x} to the power of #{y}: #{squared_result}")
end
```

Output

```
7 to the power of 2: 49
```

2.2 Using shorthand notation

Anonymous functions can also be defined using a shorthand notation:

Code

```
def main do
  # Define the same anonymous function using shorthand notation
  myfun = &(&1**&2)

  # Call anonymous function
  squared_result = myfun.(x, y)

  # Print result
  IO.puts("#{x} to the power of #{y}: #{squared_result}")
end
```

Output

```
7 to the power of 2: 49
```

As we can see, the calling is the same for both methods, but the function definition is done differently.

2.3 Using a multi-clause function

If we're not sure about the number of parameters we will receive when calling our function, we can define a set of cases that will help us act according to the number of parameters we get:

Code

```
def main do
  # Define a function where we do not know the number of expected parameters
  myfun = fn
    # If parameters are x and y, exponentiate
    {x, y} -> x**y

    # If parameters are x, y, and z, calculate product
    {x, y, z} -> x * y * z

    # If parameters are none, return nil
    {} -> nil
  end

  x = 7
  y = 2
  z = 3

  res_1 = myfun.({x, y})
  res_2 = myfun.({x, y, z})
  res_3 = myfun.({})
end
```

Output

```
Results (cases 1, 2, 3): 49, 42,
```

Let us explain in detail:

- We define a function `myfun` where we don't necessarily know the exact number of parameters that will be used to call it.
- We set 3 cases:
  - **Case 1:** Two parameters, `x` and `y`, are used: Exponentiate.
  - **Case 2:** Three parameters, `x`, `y`, and `z` are used: Product.
  - **Case 3:** No parameter is used: `nil` (*similar to* `null` *in other languages; a value that represents the absence of a value*)
- We then call our function testing all 3 cases by enclosing our parameters in curly brackets `{}` inside the function call.

§

Modules

Modules are used to organize and group related functions and data. Modules provide a namespace for the functions and data they contain, preventing naming conflicts with other parts of our code. They also help structure our applications and make them easier to maintain, understand, and test.

As in many other languages, Modules in Elixir are typically defined using CamelCase (*the first letter of each word is capitalized*). This recommendation can or cannot be followed. However, we do need to at least capitalize our module name.

Additionally, we can define a module documentation string (*equivalent to a docstring in Python*)

We can define a module with its `moduledoc` using the following syntax:

Code

```
defmodule Project1 do
  @moduledoc """
  This is a module containing functions to explore the Elixir language.
  """
  # Define function inside module
  def main do
    IO.puts("Hello World")
  end
end
```

Here, we have defined a module named `Project1` with a function inside of it called `myfun`.

We have seen that upon compilation using `mix`, we can define the module and function to compile:

Code

```
mix run -e "Project1.main()"
```

Output

```
Hello World
```

However, we can also define another module with another `main` function calling a function defined in our `Project1` module and compile it so that our module is now `Module1` instead of `Project1` but calls methods defined inside `Project1`:

Code

```
# Define Project1 Module
defmodule Project1 do
  def myfun do
    IO.puts("Printing from within Project1 Module")
  end
end

# Define Module2 module inside same script
defmodule Module2 do
  def main do
    Project1.myfun()
  end
end
```

We now compile our new module:

Code

```
mix run -e "Module2.main()"
```

Output

```
Printing from within Project1 Module
```

§

Next steps

Elixir is all about learning to think in functional programming terms. It brings all the aspects that make functional programming great while at the same time providing an easy and expressive syntax.

It then makes sense to learn the pillars of functional thinking as the first step:

- **Focusing on functions:** In functional programming, functions are the building blocks of programs. We can think in terms of functions that take inputs and produce outputs. Functions should be pure, meaning they don't have side effects and always produce the same output given the same input.
- **Using immutable data structures:** In functional programming, data structures are usually immutable, meaning they can't be changed once created. When we need to modify a data structure, we create a new version of the structure that reflects the change. This can take some time to get used to, but it leads to more predictable and less error-prone code.
- **Avoiding mutable state:** In functional programming, a mutable state is usually avoided in favor of pure functions and immutable data structures. If we need to maintain a state, we can use techniques like recursion or passing state as function arguments and return values.
- **Using higher-order functions:** In functional programming, functions can take other functions as arguments and return functions as results. This allows us to write more generic and reusable code that can be customized with different functions.
- **Using recursion:** This one is trickier and takes some time to get used to; if we come from an OOP context, the more obvious approach is to simply declare a `for` loop and nest other `loops` to provide the appropriate level of depth as per required. We didn't mention `for` loops in this segment for a reason; recursive approaches usually provide increased simplicity, generality, conciseness, and, most important of all, efficiency.
- **Understanding functional programming concepts:** Functional programming is not a religion but a set of tools to make programs more computationally efficient. To truly think in functional programming terms, it's important to understand the underlying concepts and principles of functional programming and why they matter in the first place.

There are a lot of free resources for learning more advanced Elixir concepts. Below we will find a collection of books, Courses, YouTube Channels, and articles:

Disclaimer: None of the resources below are sponsored. All the material was selected by myself.

First-stops

- **The Elixir Language, Official Page**: Contains a summary of what Elixir can do, along with multiple redirections to amazing learning resources.
- **The Elixir Learning page**: An index of the official recommended learning resources.
- **Elixir GitHub Repository**: A great place where we can check the latest releases, issues & bug reports and also get to know the maintainers that make Elixir possible.
- **Elixir Hex**: Hex is the package manager for Erlang ecosystems containing 16k+ packages at the time of writing of this article. We can visualize them by Most Downloaded, New Packages, and Recently Updated.

Relevant bibliography:

- **Elixir in Action, Manning Publications**: A tutorial book that teaches Elixir and Erlang from the ground up to advanced concepts. The most recommended book among senior Elixir programmers.
- **Designing Elixir Systems with OTP, The Pragmatic Bookshelf**: A great resource teaching how not just to write Elixir code, but actually think in Elixir. Aimed at teaching how to write highly scalable, self-healing, fault-tolerant software with layers.
- **Concurrent Data Processing in Elixir**: Oriented towards teaching how to write fast, resilient, concurrent applications using OTP, GenStage, Flow, and Broadway.

Online Courses:

- **Elixir-School**: An open and community-driven effort inspired by Twitter's Scala School, entirely for free. The site's content consists of peer-reviewed lessons on various Elixir topics that range in difficulty. The lessons are currently available in over ten languages to help make programming Elixir more accessible to non-English speakers.
- **Elixir/OTP Course, The Pragmatic Studio**: A hands-on, step-by-step 6-hour video teaching how to build a concurrent, fault-tolerant application from scratch. Also, The Pragmatic Studio offers Purchasing Power Parity Pricing. How awesome is that?
- **Elixir Course, grox.io**: A full program containing an e-book, eight videos, dozens of exercises, and two full test-first projects.
- **OTP Course, grox.io**: A module focused at teaching OTP for first-time learners.

YouTube Channels:

- **Coding Tech, Why Elixir Matters**: A nice conference imparted by Osa Gaius, explaining theory, history & relevance behind functional programming, as well as the importance of Elixir today.
- **Elixir Tutorial, Derek Banas**: A must first stop for beginners.
- **Elixir & Phoenix Playlist, Tensor Programming**: A great series providing five segments on Elixir, and nine segments on Phoenix, including multiple hands-on examples.

Community:

- **The Elixir Forum**: The place where Elixir programmers support each other and share ideas.
- **Dev.to, Elixir**: A great forum belonging to dev.to, exclusive for Elixir and OTP.
- **Erlang Solutions, Hub**: A great hub hosting conferences, publishing reports, and exploring Elixir & Erlang success stories throughout the globe.

Interactive notebook environments:

- **Fly.io**: Specialized in providing interfaces for multiple frameworks and languages. Supports the Phoenix framework, where we can write Elixir applications.
- **Livebook.dev**: Supports Elixir programming through Mix.

§

Conclusions

In this segment, we introduced Elixir's context and main advantages. We then installed the language along with a VS Code extension, created our first project using Mix, included a dependency using Mix & Hex, and compiled it to bytecode that can be run on the Erlang Virtual Machine (*VM*). We then talked about syntax, data types, variables, operators, methods, conditional constructs, iterators using `Enum` and recursion, different function types, and modules.

We concluded by providing some next steps for those interested in learning more about this robust, fault-tolerant functional language.

References

- Elixir: The Documentary
- Stack Overflow, Developer Survey 2022
- Erlang Solutions, Which companies are using Elixir, and why?
- How To Install Elixir, Derek Banas
- Elixir Tutorial, Derek Banas
- Elixir Documentation
- Elixir Documentation, Mix
- Elixir Documentation, Standard Library
- MIT, Concurrency
- Computing University of Colorado Boulder, Parallel Programming
- Elixir Documentation, Atom
- Elixir Documentation, Enum

Copyright