# Elixir

tutorialspoint

SIMPLY EASY LEARNING

## About the Tutorial

Elixir is a dynamic, functional language designed for building scalable and maintainable applications. It is built on top of Erlang. Elixir leverages the Erlang VM, known for running low-latency, distributed and fault-tolerant systems, while also being successfully used in web development and the embedded software domain.

## Audience

This tutorial is created for software programmers who aim to learn the fundamentals of Elixir programming language from scratch. This tutorial will give you a basic foundation to start programming in Elixir programming language.

## Prerequisites

Before proceeding with this tutorial, you should have a basic understanding of Computer Programming terminologies and exposure to any other programming language. Some familiarity with functional programming will help you in learning Elixir.

## Execute Elixir Online

For most of the examples given in this tutorial, you will find the **Try it** option. You can make use of this option to execute your Elixir programs on the go and enjoy your learning.

Try the following example using the **Try it** option available at the top right corner of the below sample code box –

```
IO.puts "Hello world"
```

## Copyright & Disclaimer

# Table of Contents

# 1. ELIXIR – OVERVIEW

Elixir is a dynamic, functional language designed for building scalable and maintainable applications. It leverages the Erlang VM, known for running low-latency, distributed and fault-tolerant systems, while also being successfully used in web development and the embedded software domain.

Elixir is a functional, dynamic language built on top of Erlang and the Erlang VM. Erlang is a language that was originally written in 1986 by Ericsson to help solve telephony problems like distribution, fault-tolerance, and concurrency. Elixir, written by José Valim, extends Erlang and provides a friendlier syntax into the Erlang VM. It does this while keeping the performance of the same level as Erlang.

## Features of Elixir

Let us now discuss a few important features of Elixir:

- **Scalability** — All Elixir code runs inside lightweight processes that are isolated and exchange information via messages.

- **Fault Tolerance** — Elixir provides supervisors which describe how to restart parts of your system when things go wrong, going back to a known initial state that is guaranteed to work. This ensures your application/platform is never down.

- **Functional Programming** — Functional programming promotes a coding style that helps developers write code that is short, fast, and maintainable.

- **Build tools** — Elixir ships with a set of development tools. Mix is one such tool that makes it easy to create projects, manage tasks, run tests, etc. It also has its own package manager - Hex.

- **Erlang Compatibility** — Elixir runs on the Erlang VM giving developers complete access to Erlang's ecosystem.

# 2. ELIXIR – ENVIRONMENT

In order to run Elixir, you need to set it up locally on your system.

To install Elixir, you will first require Erlang. On some platforms, Elixir packages come with Erlang in them.

## Installing Elixir

Let us now understand the installation of Elixir in different Operating Systems.

### Windows Setup

To install Elixir on windows, download installer from https://repo.hex.pm/elixir-websetup.exe and simply click **Next** to proceed through all steps. You will have it on your local system.

If you have any problems while installing it, you can check this page for more info.

### Mac Setup

If you have Homebrew installed, make sure that it is the latest version. For updating, use the following command:

```
brew update
```

Now, install Elixir using the command given below:

```
brew install elixir
```

### Ubuntu/Debian Setup

The steps to install Elixir in an Ubuntu/Debian setup is as follows:

Add Erlang Solutions repo:

```
wget  https://packages.erlang-solutions.com/erlang-solutions_1.0_all.deb  &&  sudo
dpkg -i erlang-solutions_1.0_all.deb

sudo apt-get update
```

Install the Erlang/OTP platform and all of its applications:

```
sudo apt-get install esl-erlang
```

Install Elixir:

```
sudo apt-get install elixir
```

### Other Linux Distros

If you have any other Linux distribution, please visit this page to set up elixir on your local system.

## Testing the Setup

To test the Elixir setup on your system, open your terminal and enter **iex** in it. It will open the interactive elixir shell like the following:

```
Erlang/OTP 19 [erts-8.0] [source-6dc93c1] [64-bit] [smp:4:4] [async-threads:10]
[hipe] [kernel-poll:false]


Interactive Elixir (1.3.1) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)>
```

Elixir is now successfully set up on your system.

# 3. ELIXIR – BASIC SYNTAX

We will start with the customary 'Hello World' program.

To start the Elixir interactive shell, enter the following command.

```
iex
```

After the shell starts, use the **IO.puts** function to "put" the string on the console output. Enter the following in your Elixir shell:

```
IO.puts "Hello world"
```

In this tutorial, we will use the Elixir script mode where we will keep the Elixir code in a file with the extension **.ex**. Let us now keep the above code in the **test.ex** file. In the succeeding step, we will execute it using **elixirc**:

```
IO.puts "Hello world"
```

Let us now try to run the above program as follows:

```
$elixirc test.ex
```

The above program generates the following result:

```
Hello World
```

Here we are calling a function **IO.puts** to generate a string to our console as output. This function can also be called the way we do in C, C++, Java, etc., providing arguments in parentheses following the function name:

```
IO.puts("Hello world")
```

## Comments

Single line comments start with a '#' symbol. There's no multi-line comment, but you can stack multiple comments. For example:

```
#This is a comment in Elixir
```

## Line Endings

There are no required line endings like ';' in Elixir. However, we can have multiple statements in the same line, using ';'. For example,

```
IO.puts("Hello"); IO.puts("World!")
```

The above program generates the following result:

```
Hello
World
```

## Identifiers

Identifiers like variables, function names are used to identify a variable, function, etc. In Elixir, you can name your identifiers starting with a lower case alphabet with numbers, underscores and upper case letters thereafter. This naming convention is commonly known as snake_case. For example, following are some valid identifiers in Elixir:

```
var1       variable_2      one_M0r3_variable
```

Please note that variables can also be named with a leading underscore. A value that is not meant to be used must be assigned to _ or to a variable starting with underscore:

```
_some_random_value = 42
```

Also elixir relies on underscores to make functions private to modules. If you name a function with a leading underscore in a module, and import that module, this function will not be imported.

There are many more intricacies related to function naming in Elixir which we will discuss in coming chapters.

## Reserved Words

Following words are reserved and cannot be used as variables, module or function names.

```
after     and     catch     do     inbits     inlist     nil     else     end
not     or     false     fn     in     rescue     true     when     xor
__MODULE__     __FILE__     __DIR__     __ENV__     __CALLER__
```

# 4. ELIXIR – DATA TYPES

For using any language, you need to understand the basic data types the language supports. In this chapter, we will discuss 7 basic data types supported by the elixir language: integers, floats, Booleans, atoms, strings, lists and tuples.

## Numerical Types

Elixir, like any other programming language, supports both integers and floats. If you open your elixir shell and input any integer or float as input, it'll return its value. For example,

```
42
```

When the above program is run, it produces the following result:

```
42
```

You can also define numbers in octal, hex and binary bases.

**OCTAL**

To define a number in octal base, prefix it with '0o'. For example, 0o52 in octal is equivalent to 42 in decimal.

**HEXADECIMAL**

To define a number in decimal base, prefix it with '0x'. For example, 0xF1 in hex is equivalent to 241 in decimal.

**BINARY**

To define a number in binary base, prefix it with '0b'. For example, 0b1101 in binary is equivalent to 13 in decimal.

Elixir supports 64bit double precision for floating point numbers. And they can also be defined using an exponentiation style. For example, 10145230000 can be written as 1.014523e10

### Atoms

Atoms are constants whose name is their value. They can be created using the color(:) symbol. For example,

```
:hello
```

### Booleans

Elixir supports **true** and **false** as Booleans. Both these values are in fact attached to atoms :true and :false respectively.

## Strings

Strings in Elixir are inserted between double quotes, and they are encoded in UTF-8. They can span multiple lines and contain interpolations. To define a string simply enter it in double quotes:

```
"Hello world"
```

To define multiline strings, we use a syntax similar to python with triple double quotes:

```
"""
Hello
World!
"""
```

We'll learn about strings, binaries and char lists (similar to strings) in depth in the strings chapter.

## Binaries

Binaries are sequences of bytes enclosed in << >> separated with a comma. For example,

```
<< 65, 68, 75>>
```

Binaries are mostly used to handle bits and bytes related data, if you have any. They can, by default, store 0 to 255 in each value. This size limit can be increased by using the size function that says how many bits it should take to store that value. For example,

```
<<65, 255, 289::size(15)>>
```

## Lists

Elixir uses square brackets to specify a list of values. Values can be of any type. For example,

```
[1, "Hello", :an_atom, true]
```

Lists come with inbuilt functions for head and tail of the list named hd and tl which return the head and tail of the list respectively. Sometimes when you create a list, it'll return a char list. This is because when elixir sees a list of printable ASCII characters, it prints it as a char list. Please note that strings and char lists are not equal. We'll discuss lists further in later chapters.

## Tuples

Elixir uses curly brackets to define tuples. Like lists, tuples can hold any value.

```
{ 1, "Hello", :an_atom, true }
```

A question arises here — why provide both **lists** and **tuples** when they both work in the same way? Well, they have different implementations.

- Lists are actually stored as linked lists, so insertions, deletions are very fast in lists.
- Tuples, on the other hand, are stored in contiguous memory block, which make accessing them faster but adds an additional cost on insertions and deletions.

# 5. ELIXIR – VARIABLES

A variable provides us with named storage that our programs can manipulate. Each variable in Elixir has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable.

## Types of Variables

Elixir supports the following basic types of variables:

### Integer

These are used for Integers. They are of size 32bit on a 32bit architecture and 64 bits on a 64-bit architecture. Integers are always signed in elixir. If an integer starts to expand in size above its limit, elixir convers it in a Big Integer which takes up memory in range 3 to n words whichever can fit it in memory.

### Floats

Floats have a 64-bit precision in elixir. They are also like integers in terms of memory. When defining a float, exponential notation can be used.

### Boolean

They can take up 2 values which is either true or false.

### Strings

Strings are utf-8 encoded in elixir. They have a strings module which provides a lot of functionality to the programmer to manipulate strings.

### Anonymous Functions/Lambdas

These are functions that can be defined and assigned to a variable, which can then be used to call this function.

### Collections

There are a lot of collection types available in Elixir. Some of them are Lists, Tuples, Maps, Binaries, etc. These will be discussed in subsequent chapters.

## Variable Declaration

A variable declaration tells the interpreter where and how much to create the storage for the variable. Elixir does not allow us to just declare a variable. A variable must be declared and assigned a value at the same time. For example, to create a variable named life and assign it a value 42, we do the following:

```
life = 42
```

This will *bind* the variable life to value 42. If we want to reassign this variable a new value, we can do this by using the same syntax as above, i.e.,

```
life = "Hello world"
```

## Variable Naming

Naming variables follow a **snake_case** convention in Elixir, i.e., all variables must start with a lowercase letter, followed by 0 or more letters(both upper and lower case), followed at the end by an optional '?' OR '!'.

Variable names can also be started with a leading underscore but that must be used only when ignoring the variable, i.e., that variable will not be used again but is needed to be assigned to something.

## Printing Variables

In the interactive shell, variables will print if you just enter the variable name. For example, if you create a variable:

```
life = 42
```

And enter 'life' in your shell, you'll get the output as:

```
42
```

But if you want to output a variable to the console (When running an external script from a file), you need to provide the variable as input to **IO.puts** function:

```
life = 42
IO.puts life
```

or

```
life = 42
IO.puts(life)
```

End of ebook preview
If you liked what you saw…
Buy it from our store @ **https://store.tutorialspoint.com**