Elixir Cheat Sheet

A comprehensive cheat sheet for Elixir, covering syntax, data types, control flow, modules, and more, with examples.

Getting Started & Basic Syntax

Basic Structure

Elixir is a functional, concurrent language built on the Erlang VM.

- `.ex` - Elixir source code files.
- `.exs` - Elixir script files.

Running Elixir Code

```
elixir myfile.exs
```

Hello, World!

```
defmodule Greeter do
  def greet(name) do
    message = "Hello, " <> name <> "!"
    IO.puts message
  end
end

Greeter.greet("world")
```

Output

```
Hello, world!
```

Variables

```
age = 23
name = "Elixir"
```

Variables are immutable; you cannot change the value of an existing variable.

```
age = 23
age = 24 # this will produce a warning
as it is rebind
```

Data Types

Primitive Types

`nil`	Nil/null value.
`true / false`	Boolean values.
`?a`	Integer (ASCII) value.
`23`	Integer value.
`3.14`	Float value.
`'hello'`	Charlist (list of characters).
`<<2, 3>>`	Binary data.
`"hello"`	Binary string.
`:`	Atom (a constant with name).

Collections

`[a, b]`	List (ordered collection).
`{a, b}`	Tuple (ordered, fixed-size collection).
`%{a: "hello"}`	Map (key-value pairs).
`%MyStruct{a: "hello"}`	Struct (extension of map, with predefined keys).

Functions

`fn -> ... end`

Anonymous function (lambda).

Type Checks

`is_atom/1`, `is_bitstring/1`, `is_boolean/1`, `is_function/1`, `is_integer/1`, `is_float/1`, etc.

Functions to check the type of a value.

Control Flow & Operators

Control Flow

If

```
if condition do
  # code
else
  # code
end
```

Case

```
case value do
  pattern1 ->
    # code
  pattern2 ->
    # code
  _ -> # Default case
    # code
end
```

Cond

```
cond do
  condition1 ->
    # code
  condition2 ->
    # code
  true -> # Default condition
    # code
end
```

With

```
with {:ok, result1} <- function1(),
     {:ok, result2} <-
function2(result1) do
  # Success code
else
  {:error, reason} ->
    # Error handling
end
```

Error Handling

```
try do
  # code that might raise an error
rescue
  exception ->
    # Handle the exception
after
  # code that always runs
end
```

Operators

`left != right`	Not equal.
`left !== right`	Strict not equal(no type conversion).
`left ++ right`	Concatenate lists.
`left <> right`	Concatenate string/binary.
`left =~ right`	Regexp match.

Modules and Functions

Modules

```
defmodule MyModule do
  def my_function(arg) do
    # Function implementation
  end
end
```

Accessing a Function:

```
MyModule.my_function(value)
```

Importing and Aliasing

`require Module` - Compiles a module.

`import Module` - Compiles and allows use without the `Module.` prefix.

`use Module` - Compiles and runs `Module.__using__/1`.

`alias Module, as: Alias` - Creates an alias for a module.

Example:

```
import String
String.length("hello")
```

can be written as

```
import String
length("hello")
```

String Functions

```
import String

str = "hello"
str |> length()        # → 5
str |> codepoints()    # → ["h", "e",
"l", "l", "o"]
str |> slice(2..-1)    # → "llo"
str |> split(" ")      # → ["hello"]
str |> capitalize()    # → "Hello"
```

Anonymous Functions (Lambdas)

```
square = fn n -> n * n end
square.(20) # 400
```

Shorthand Syntax

```
square = &(&1 * &1)
square.(20)

square = &Math.square/1 # Capturing
existing function
```

Function Heads and Pattern Matching

```
def join(a, b \\ nil) # b has default
value nil
def join(a, b) when is_nil(b) do: a #
function head with a guard
def join(a, b) do: a <> b # normal
function head
```

Advanced Features

Structs

```
defmodule User do
  defstruct name: "", age: nil
end

%User{name: "John", age: 20}

%User{}.__struct__ # → User
```

Protocols

```
defprotocol Blank do
  @doc "Returns true if data is
considered blank/empty"
  def blank?(data)
end

defimpl Blank, for: List do
  def blank?([]), do: true
  def blank?(_), do: false
end

Blank.blank?([]) # → true
```

`Any` - Implementing for any type.
`@derive Protocol` - Deriving implementations.

Comprehensions

For Loops

```
for n <- [1, 2, 3, 4], do: n * n # [1,
4, 9, 16]
for n <- 1..4, do: n * n # same result
```

Conditions

```
for n <- 1..10, rem(n, 2) == 0, do: n #
even numbers
```

Into

```
for {key, val} <- %{a: 10, b: 20}, into:
%{}, do: {key, val * val}
# → %{a: 100, b: 400}
```

Metaprogramming

`__MODULE__` - Current module.

`__MODULE__.__info__` - Module information.

`@before_compile Module` - Code to run before compilation.

`@after_compile Module` - Code to run after compilation.

`@on_definition {__MODULE__, :on_def}` - Callback when a function is defined.

Regular Expressions

```
exp = ~r/hello/
exp = ~r/hello/i # case insensitive
"hello world" =~ exp # returns position
or false
```

Sigils

`~r/regexp/` - Regular expression.

`~w(list of strings)` - List of strings.

`~s|string with #{interpolation}|` - String with interpolation.

`~S|string without interpolation|` - String without interpolation.

`~c(charlist)` - Charlist.