

Elixir Style reference card

Source Code Layout

- Use two **spaces** per indentation level. No hard tabs.
- Use one expression per line, don't use semicolon ; to separate statements and expressions.
- Use spaces around default arguments \\ definition.
- Add underscores to large numeric literals to improve their readability, e.g. num = 1_000_000
- Avoid trailing whitespaces.
- End each file with a newline.

Spaces in code

Use spaces around binary operators, after commas ,, colons : and semicolons ;. Do not put spaces around matched pairs like brackets [], braces {}, etc. Whitespace might be (mostly) irrelevant to the Elixir compiler, but its proper use is the key to writing easily readable code.

```
sum = 1 + 2
[first | rest] = 'three'
{a1, a2} = {2, 3}
Enum.join(["one", <<"two">>, sum])
```

No spaces in code

No spaces after unary operators and inside range literals, the only exception is the not operator.

```
angle = -45
^result = Float.parse("42.01")
2 in 1..5
not File.exists?(path)
```

bitstring segment options

Do not put spaces around segment options definition in bitstrings.

```
# Bad
<<102 :: unsigned-big-integer, rest :: binary>>
<<102::unsigned - big - integer, rest::binary>>
# Good
<<102::unsigned-big-integer, rest::binary>>
```

Guard clauses

Indent when guard clauses on the same level as the function/macro signature in the definition they're part of. Do this only if you cannot fit the when guard on the same line as the definition.

```
def format_error({exception, stacktrace})
  when is_list(stacktrace) and stacktrace != [] do
# ...
end
```

```
defmacro dgettext(domain, msgid, msgid_plural, count)
  when is_binary(msgid) and is_binary(msgid_plural) do
# ...
end
```

Quotes around atoms

When using atom literals that need to be quoted because they contain characters that are invalid in atoms (such as : "foo-bar"), use double quotes around the atom name:

<i>Bad</i>	<i>Good</i>
: 'foo-bar'	: "foo-bar"
: atom number #{index}'	: atom number #{index}"

Multi-line expression assignment

When assigning the result of a multi-line expression, do not preserve alignment of its parts.

```
# Bad
{found, not_found} = Enum.map(files, &Path.expand(&1, path))
                       |> Enum.partition(&File.exists?/1)
```

```
prefix = case base do
  :binary -> "0b"
  :octal  -> "0o"
  :hex    -> "0x"
end
# Good
{found, not_found} =
  Enum.map(files, &Path.expand(&1, path))
  |> Enum.partition(&File.exists?/1)
```

```
prefix = case base do
  :binary -> "0b"
  :octal  -> "0o"
  :hex    -> "0x"
end
```

Trailing comma

When dealing with lists, maps, structs, or tuples whose elements span over multiple lines and are on separate lines with regard to the enclosing brackets, it's advised to use a trailing comma even for the last element.

```
[
  :foo,
  :bar,
  :baz,
]
```

Expression group alignment

Avoid aligning expression groups:

<i>Bad</i>	<i>Good</i>
module = env.module	module = env.module
arity = length(args)	arity = length(args)
def inspect(false), do: "false"	def inspect(false), do: "false"
def inspect(true), do: "true"	def inspect(true), do: "true"
def inspect(nil), do: "nil"	def inspect(nil), do: "nil"

The same non-alignment rule applies to <- and -> clauses as well.

Syntax

Function parentheses

Always use parentheses around def arguments, don't omit them even when a function has no arguments.

<i>Bad</i>	<i>Good</i>
def main arg1, arg2 do	def main(arg1, arg2) do
#...	#...
end	end
def main do	def main() do
#...	#...
end	end

Pipeline operator

Favor the pipeline operator |> to chain function calls together.

<i>Bad</i>	<i>Good</i>
String.downcase(String.strip(input))	input > String.strip > String.downcase
String.strip(input) > String.downcase	String.strip(input) > String.downcase

Use a single level of indentation for multi-line pipelines.

```
String.strip(input)
|> String.downcase
|> String.slice(1, 3)
```

Needless pipeline operator

Avoid needless pipelines like the plague.

<i>Bad</i>	<i>Good</i>
result = input > String.strip	result = String.strip(input)

Parentheses for local zero-arity functions

Parentheses are a must for **local** zero-arity function calls and definitions.

<i>Bad</i>	<i>Good</i>
pid = self	pid = self()
def new, do: %MapSet{}	def new(), do: %MapSet{}
	config = IEx.Config.new

The same should be done for **remote** zero-arity function calls:

<i>Bad</i>	<i>Good</i>
Mix.env	Mix.env()

This rule also applies to one-arity function calls (both local and remote) in pipelines:

<i>Bad</i>	<i>Good</i>
input	input
> String.strip	> String.strip()
> decode	> decode()

Anonymous function parentheses

Never wrap the arguments of anonymous functions in parentheses.

<i>Bad</i>	<i>Good</i>
Agent.get(pid, fn(state) -> state end)	Agent.get(pid, fn state -> state end)
Enum.reduce(numbers, fn(number, acc) ->	Enum.reduce(numbers, fn number, acc ->
acc + number	acc + number
end)	end)

Binary operators at the end of line

When writing a multi-line expression, keep binary operators at the end of each line. The only exception is the |> operator (which goes at the beginning of the line).

<i>Bad</i>	<i>Good</i>
"No matching message.\n"	"No matching message.\n" <>
<> "Process mailbox:\n"	"Process mailbox:\n" <>
<> mailbox	mailbox

with indentation

Use the indentation shown below for the with special form:

```
with {year, ""} <- Integer.parse(year),
     {month, ""} <- Integer.parse(month),
     {day, ""} <- Integer.parse(day) do
  new(year, month, day)
else
  - ->
  - {:error, :invalid_format}
end
```

Always use the indentation above if there's an else option. If there isn't, the following indentation works as well:

```
with {:ok, date} <- Calendar.ISO.date(year, month, day),
     {:ok, time} <- Time.new(hour, minute, second, microsecond),
     do: new(date, time)
```

for indentation

```
for {alias, _module} <- aliases_from_env(server),
    [name] = Module.split(alias),
    starts_with?(name, hint),
    into: [] do
  %{:kind: :module, type: :alias, name: name}
end
```

If the body of the do block is short, the following indentation works as well:

```
for partition <- @..(partitions - 1),
    pair <- safe_lookup(registry, partition, key),
    into: [],
    do: pair
```

Never use unless with else

Rewrite these with the positive case first.

<i>Bad</i>	<i>Good</i>
<pre>unless Enum.empty?(coll) do :ok else :error end</pre>	<pre>if Enum.empty?(coll) do :error else :ok end</pre>

No nil-else

Omit else option in if and unless clauses if it returns nil.

```
# Bad
if byte_size(data) > 0, do: data, else: nil
```

```
# Good
if byte_size(data) > 0, do: data
```

true in cond

If you have an always-matching clause in the cond special form, use true as its condition.

<i>Bad</i>	<i>Good</i>
<pre>cond do char in ?0..?9 -> char - ?0 char in ?A..?Z -> char - ?A + 10 :other -> char - ?a + 10 end</pre>	<pre>cond do char in ?0..?9 -> char - ?0 char in ?A..?Z -> char - ?A + 10 true -> char - ?a + 10 end</pre>

Boolean Operators

Never use ||, && and ! for strictly boolean checks. Use these operators only if any of the arguments are non-boolean.

<i>Bad</i>	<i>Good</i>
<pre>is_atom(name) && name != nil is_binary(task) is_atom(task)</pre>	<pre>is_atom(name) and name != nil is_binary(task) or is_atom(task) line && line != 0 file "sample.exs"</pre>

Patterns matching binaries

Favor the binary concatenation operator <> over bitstring syntax for patterns matching binaries.

<i>Bad</i>	<i>Good</i>
<pre><<"http://", _rest::bytes>> = input <<first::utf8, rest::bytes>> = input</pre>	<pre>"http://" <> _rest = input <<first::utf8>> <> rest = input</pre>

Use uppercase in definition of hex literals

<i>Bad</i>	<i>Good</i>
<pre><<0xef, 0xbb, 0xbf>></pre>	<pre><<0xEF, 0xBB, 0xBF>></pre>

Naming

- Use snake_case for naming directories and files, e.g. lib/my_app/task_server.ex.
- Avoid using one-letter variable names.
- Use snake_case for atoms, functions, variables and module attributes.

<i>Bad</i>	<i>Good</i>
<pre>:no_match" :error :badReturn</pre>	<pre>:no_match :error :bad_return</pre>
<pre>fileName = "sample.txt"</pre>	<pre>file_name = "sample.txt"</pre>
<pre>@_VERSION "0.0.1"</pre>	<pre>@version "0.0.1"</pre>
<pre>def readFile(path) do #... end</pre>	<pre>def read_file(path) do #... end</pre>

Use CamelCase for module names

<i>Bad</i>	<i>Good</i>
<pre>defmodule :appStack do #... end defmodule App_Stack do #... end defmodule Appstack do #... end</pre>	<pre>defmodule AppStack do #... end defmodule Appstack do #... end</pre>

Predicate function names

The names of predicate functions (a function that return a boolean value) should have a trailing question mark ? rather than a leading has_ or similar.

```
def leap?(year) do
  #...
end
```

Always use a leading is_ when naming guard-safe predicate macros.

```
defmacro is_date(month, day) do
  #...
end
```

Comments

- Write self-documenting code and ignore the rest of this section. **Seriously!**
- Use one space between the leading # character of the comment and the text of the comment.
- Avoid superfluous comments. e.g. String.first(input) # Get first grapheme

Modules

Module Layout

Use a consistent structure when calling use/import/alias/require: call them in this order and group multiple calls to each of them.

```
use GenServer

import Bitwise
import Kernel, except: [length: 1]
```

```
alias Mix.Utils
alias MapSet, as: Set
```

```
require Logger
```

Use __MODULE__ to reference current module

```
# Bad
:ets.new(Kernel.LexicalTracker, [:named_table])
GenServer.start_link(Module.LocalsTracker, nil, [])
```

```
# Good
:ets.new(__MODULE__, [:named_table])
GenServer.start_link(__MODULE__, nil, [])
```

Regular expressions are the last resort

Pattern matching and String module are things to start with.

```
# Bad
Regex.run(~r/#{\d{2}}{\d{2}}{\d{2}}/, color)
Regex.match?(~r/(email|password)/, input)
```

```
# Good
<<?#, p1::2-bytes, p2::2-bytes, p3::2-bytes>> = color
String.contains?(input, ["email", "password"])
```

Use non-capturing RegEx

...when you don't use the captured result.

```
~r/(?post|zip )code: (\d+)/
```

Caret and dollar RegEx

Be careful with ^ and \$ as they match start and end of the line respectively. If you want to match the whole string use: \A and \z (not to be confused with \Z which is the equivalent of \n?\z).

Structs

defstruct with default fields

When calling defstruct/1, don't explicitly specify nil for fields that default to nil.

```
# Bad
defstruct first_name: nil, last_name: nil, admin?: false
```

```
# Good
defstruct [:first_name, :last_name, admin?: false]
```

Exceptions

Make exception names end with a trailing Error

<i>Bad</i>	<i>Good</i>
<pre>BadResponse ResponseException</pre>	<pre>ResponseError</pre>

Use non-capitalized error messages

...when raising exceptions, with no trailing punctuation.

```
# Bad
raise ArgumentError, "Malformed payload."
```

```
# Good
raise ArgumentError, "malformed payload"
```

There is one exception to the rule - always capitalize Mix error messages. Mix.raise "Could not find dependency"

Typespecs

Never use parens on zero-arity types

```
# Bad
@spec start_link(module(), term(), Keyword.t()) :: on_start()
```

```
# Good
@spec start_link(module, term, Keyword.t) :: on_start
```

ExUnit

ExUnit assertion side

When asserting (or refuting) something with comparison operators (such as ==, <, >=, and similar), put the expression being tested on the left-hand side of the operator and the value you're testing against on the right-hand side.

```
# Bad
assert "hello" == Atom.to_string("hello")
```

```
# Good
assert Atom.to_string("hello") == "hello"
```

When using the match operator =, put the pattern on the left-hand side (as it won't work otherwise).

```
assert {:error, _reason} = File.stat("./non_existent_file")
```

This reference card is an adaptation of the "Elixir Style Guide", it was created by Milton Mazzarri and is licensed under the CC BY 4.0 license. <http://github.com/milmazz/elixir-style-refcard/> You can find the original "Elixir Style Guide" by Aleksei Magusev here: <https://github.com/lexmag/elixir-style-guide>