Elixir Cheat Sheet

elixir-lang.org v1.2.4

Command Line:

```
elixir /your.exs
iex -S mix

iex --help
iex --sname foo
iex -v
```

Mix

```
mix new # new project
mix help cmd

#Example `mix help run`
mix --help
mix help cmd
```

iEX

```
#iex:break! — back to prompt
c "filename.exs" —compile
r Module —reload
h function_name —help
v [n] — session history
i DefinedModule # shows docs
(Ctrl+G) User switch command.
```

Operators

Comparison

```
=== !== and or not
==  !=  &&  || !
>  >=  <  <=
```

Math

```
+ - * /  returns Float
div rem  returns Integer
```

Concatenation

```
“awesome ” <> “elixir”
[0] ++ [1,2,3]
[”a”, “b”, “c”] -- [”c”]
in e.g. 4 in [1,2,4,10]
```

More info at “http://elixir-lang.org/getting-started/basic-operators.html”

Built in Type Checks

```
is_atom
is_binary
is_bitstring
is_boolean
is_float
is_function
is_integer
is_list
is_map
is_nil
is_number
is_pid
is_port
is_reference
is_tuple
```

Introduced in Elixir 1.2 — with

```
mylist = [”one”, :two, 3]

result =
with {:ok, value}  <- foo.(mylist),
     {:ok, true}   <- baz.(value),
do: bar.(value)
```

If the results of foo and baz both match.

bar is executed.

```
result =
with {:ok, value} <- foo(mylist),
     {:ok, true}  <- baz(value),
else
  {:match, pattern} -> execute.(pattern)
end
```

Update to with coming in 1.3

Processes

after clause can be given in case the message was not received after set time:t

```
spawn(myfunc)
spawn(Mod, :func, [args])
spawn_link(myfunc)
```

```
receive do
  pattern [gaurd] -> foo()
  _ -> bar()
after
  5000 -> :error
end
```

Anonymous Functions

Call with func_name “.” then ()

```
fn(param) when is_integer(param) ->
  1 + 1
end

myfunc = fn
  param when is_atom(param) ->
              foo_func.(param)
  param when is_string(param) ->
              baz_func.(param)
end
myfunc.(”binary”) # Call function
transform_list = fn
  list when length(list) < 100 ->
         foo.(list)
  list when length(list) > 100 ->
         bar.(list)
end
[1,2,3,4,5] |> transform_list.()
```

Gaurd Clauses

Used in Module funcs. & Anonymous funcs.

```
when param > 10
when param != :literal
when length(param) > 10_000
when is_atom(foo) and is_list(bar)
```

Can be used in combination with Built in type checks

```
abs(num)
bit_size(bits)
byte_size(bits)
div(num,num)
elem(tuple, n)
float(term)
hd(list)
length(list)
map_size(map)
node()
node(pid|ref|port)
rem(num,num)
round(num)
self()
size(tuple|bits)
tl(list)
trunc(num)
tuple_size(tuple)
<> and ++ (left side literal)
```

Types

Additional Types listed Online

Integer
1234 0xcafe 0177
0b100 10_000

Float
1.0 3.1415 6.02e23

Atom
:foo :foo@baz :”string”

Tuple {1,2,:ok,”xy”}

Range 0..100 foo..bar

List
[1,2,3] [head|tail]

Char lists ‘abc’
“““ here doc ”””

Truth
true, false, nil

Keyword List
[foo: "ABC", baz: 123]

Pattern Matching / Control Flow

```
iex> {a, b, c} = {:hello, "world", 42}
       a is now :hello, b is now “world”, c is now 42
iex> [head|tail] = [1,3,100]
       head is now 1 tail is [3,100]
File.read(”path/to/file.txt”)
|> case do      Result of File.read is matched in case statement.
  {:ok, binary}    -> do_something_with(binary)
  {:error, posix} -> handle_error(posix)
end
```

```
if expr do
  foo
else
  bar
end
```

```
cond do
  foo -> expr
  bar -> expr
  true -> expr
end
```

```
case value do
  foo [gaurds] -> expr
  bar [optional] -> expr
  res -> expr(res)
end
```

cond Raises an error if all conditions evaluate to nil or false.

Pipelines |>

```
foo = expr
|> first_func
|> second_func(var)
```

Transformed result of first_func/1 is passed to second_func/2 result is assigned to foo

This is the same as:

```
second_func( first_func(expr), var)
```

Comprehensions

A comprehension is made of three parts: generators, filters and collectables.

```
                < Generator >
iex> for n <- [1, 2, 3, 4], do: n * n
# => [1, 4, 9, 16]
iex> multiple_of_3 = fn(n) -> rem(n, 3) == 0 end

iex> for n <- [%{one: 3}, %{one: 10}, %{one: 9}],
     multiple_of_3.(n.one), <- Filter
     do: ["tag_#{n.one}", n.one * n.one] <- Collectable
# => [["tag_3", 9], ["tag_9", 81]]
```

Sigils

Delimiters: { }, [ ], ( ), <>, or non-word char

```
~S[ string ]  #=> “ string ” (no interpolation)
  baz = “Sparky”
~s[ #{baz} ]  #=> “Sparky”   (with interpolation)
~C[ string ]  #=> ‘ string ’ character list
~c[ #{baz} ]  #=> ‘Sparky’   character list
~R[regexp]    #=> /regexp/
~r[^#{baz}]   #=> /^Sparky/  w/interpolation
~W(white space delim) #=>["white", "space", "delim"]
~w(one two #{baz})  #=> ["one", "two", "Sparky"]
~w(one two #{dog})a #=> [:one, :two, :Sparky]
```

#ARTD - ALWAYS READ THE DOCUMENTATION

Elixir Cheat Sheet

elixir-lang.org v1.2.4

Kernel.SpecialForms

{args} Creates a tuple
% Creates a struct
%{} Creates a map
&(expr) Captures or creates an anonymous function
left . right Defines a remote-call or alias
left :: right Used by types and bitstrings to specify types
<<args>> Defines a new bitstring
left = right Matches the value on the right against the pattern on the left
^var Accesses an already bound variable in match clauses.
__CALLER__ Returns the current calling environment as a Macro.Env struct
__DIR__ Returns the current directory as a binary
__ENV__ Returns the current environment information as a Macro.Env struct
__MODULE__ Returns the current module name as an atom or nil otherwise
__aliases__(args) Internal special form to hold aliases information
__block__(args) Internal special form for block expressions
alias(module, opts) alias is used to setup aliases, often useful with modules names.
case(condition, clauses) Matches the given expression against the given clauses
cond(clauses) Evaluates the expression corresponding to the first clause that evaluates to a truthy value
fn [clauses] end # Defines an anonymous function
for(args) Comprehensions allow you to quickly build a data structure from an enumerable or a bitstring
import(module, opts) Imports functions and macros from other modules.
quote(opts, block) Gets the representation of any expression
receive(args) Checks if there is a message matching the given clauses in the current process mailbox
require(module, opts) Requires a given module to be compiled and loaded.
super(args) Calls the overriden function when overriding it with Kernel.defoverridable/1
try(args) Evaluates the given expressions and handle any error, exit or throw that may have happened
unquote(expr) Unquotes the given expression from inside a macro
unquote_splicing(expr) Unquotes the given list expanding its arguments. Similar to unquote
with(args) Used to combine matching clauses

Modules

Attributes Predifined :: See Docs.

@after_compile, @before_compile, @behaviour, @compile, @doc, @file, @moduledoc, @on_definition, @on_load, @vsn, @external_resource, @dialyzer, @type, @typep, @opaque, @spec, @callback, @macrocallback

Documentation

This can also be written in Markdown.
When documenting a function, argument names are inferred by the compiler.

Macros: Named Functions

def defines a function with the given name and body

defp defines a private function

defdelegate defines a function that delegates to another module.

defstruct defines a struct (A struct is a tagged map).

|> Macros: Code Reuse

```
alias ParentMod.MyModule, as: MyMod
# use within module as MyMod.func

import Parent.MyModule, only: [myfun: 0, myfunc: 2]
# only: expects keyword list of [public_func: arity]
# except: [public_func: arity] also available.

require MyModuleWithMacros, as: MmWM
# specified Module will be loaded and compiled.

use MyModuleWithMacros
# Uses the given module in the current context.
```

Protocols

A protocol specifies an API that should be defined by its implementations.

```
defprotocol Movement do
  def fix_gear?(data) # Note missing “do”
end
```

Macros: Meta Programming

defmacro defines a macro

defmacrop defines a private macro..

```
defmodule PedalSystem do
  defmacro pedal(params) do
     quote do: unquote(params)
  end
end
```

```
Defining a Module ->
Using SpecialForms require ->
Using module attributes ->
using macros ->
```

```
defmodule Bicycle do
  require PedalSystem
  @vsn 1.0 #Pre defined module attribute - SEE DOCS.
  defstruct [ wheels: 2, pedals: nil, engine: false ]
end
```

```
defmodule BikeTypeCheck do
  @moduledoc """
  Provides documentation for the current module.
  """
  # Comments begin with hash tag.
  def check(%Bicycle{wheels: count, pedals: val}=bike) do
    return = with {:ok, 2} <- count_wheels(count),
                  {:ok, true} <- has_pedals(val),
              do: pedal_away(bike)
   keep_passing_msg = fn
     {:error, :motorbike} -> handle_motor_bike(bike)
     {:error, :tricycle}  -> handle_tricycle(bike)
     answer -> IO.inspect(answer)
   end
   return |> keep_passing_msg.()
  end

  @doc “Provides documentation for the function or macro.”
  defp count_wheels(2), do: {:ok, 2}
  defp count_wheels(3), do: {:error, :tricycle}
  defp count_wheels(_), do: {:error, :not_bike}
  defp safe(expr \\ :false), do: {:ok, expr)
end                          ^ setting default params

# implementation of protocol
defimpl Movement, for: Bicycle do
   def fix_gear?(foo) do: [ your implementation ]
end
```

Function heads with different params & guards are allowed.

A Macro must be defined and available before elixir compiles other modules using it.

Debugging

#Start iex -S mix #Exit iex> respawn

```
defmodule TriCycle do
  require IEx

def foo(_) do
  ...
  IEx.pry    <- starts pry session when function called
  ...
end
```

respawn Respawns the current shell by starting a new shell process.

#ARTD - ALWAYS READ THE DOCUMENTATION

#ARTD - ALWAYS READ THE DOCUMENTATION

Put together by @orionengleton | tuplelife.com