Elixir Cheatsheet

A concise reference for Elixir syntax, data structures, and common functions.

Basics & Syntax

Basic Syntax

Variable Assignment	`variable_name = value` Elixir is immutable, so variables can only be bound once.
Atoms	Atoms are constants whose value is their name. `:atom_name`

Modules	`defmodule MyModule do`
	`  def hello(name) do`
	`  IO.puts "Hello, #{name}!"`
	`  end`
	`end`
	`MyModule.hello("World")`

Anonymous Functions	`fn`
	`  (x, y) -> x + y`
	`end`
	Can be assigned to variables: `add = fn (x, y) -> x + y end`

Comments	`# This is a comment`
Pipe Operator	`value |> function1() |> function2()` Chains function calls, passing the result of the previous function as the first argument to the next.

Data Types

Integers	`123, -456`
Floats	`3.14, -0.01`
Booleans	`true, false`
Strings	`"Hello, world!"`
Lists	`[1, 2, 3]`
Tuples	`{ :ok, "value" }`

Data Structures

Lists

Creating Lists	`[1, 2, 3]`
List Concatenation	`[1, 2] ++ [3, 4] #=> [1, 2, 3, 4]`
List Subtraction	`[1, 2, 3] -- [2] #=> [1, 3]`
Head and Tail	`[head | tail] = [1, 2, 3]` `# head = 1, tail = [2, 3]`
Accessing Elements	Lists are not designed for random access. Use `Enum` module for list operations.

Tuples

Creating Tuples	`{ :ok, "result" }`
Accessing Elements	`elem({:ok, "value"}, 1) #=> "value"`
Tuple Size	`tuple_size({:ok, "value"}) #=> 2`
Use Cases	Often used to return multiple values from a function, especially for error handling.

Maps

Creating Maps	`%{ key => value, "key" => value }` `%{:name => "John", :age => 30}`
Accessing Values	`map[:key] #=> value` `map.key  #=> value (when key is an atom)`
Updating Values	`Map.put(map, :key, new_value)` `Map.replace(map, :key, new_value)`
Adding Values	`Map.put(map, :new_key, value)`
Removing Values	`Map.delete(map, :key)`

Control Flow

Conditional Statements

`if` Statement	`if condition do # Code to execute if condition is true else # Code to execute if condition is false end`
`unless` Statement	`unless condition do # Code to execute if condition is false else # Code to execute if condition is true end`
`cond` Statement	`cond do condition1 -> # Code to execute if condition1 is true condition2 -> # Code to execute if condition2 is true true -> # Default case end`

Case Statement

Basic Usage	`case value do pattern1 -> # Code to execute if value matches pattern1 pattern2 -> # Code to execute if value matches pattern2 _ -> # Default case end`
Pattern Matching	`case {:ok, result} do {:ok, value} -> IO.puts "Success: #{value}" {:error, reason} -> IO.puts "Error: #{reason}" end`
Guards	`case age do age when age >= 18 -> IO.puts "Adult" age when age < 18 -> IO.puts "Minor" end`

Enum Module

`Enum.map/2`	Applies a function to each element in a collection and returns a new collection with the results. `Enum.map([1, 2, 3], fn x -> x * 2 end) #=> [2, 4, 6]`
`Enum.filter/2`	Filters elements from a collection based on a given function. `Enum.filter([1, 2, 3, 4], fn x -> rem(x, 2) == 0 end) #=> [2, 4]`
`Enum.reduce/3`	Reduces a collection to a single value by applying a function cumulatively. `Enum.reduce([1, 2, 3], 0, fn x, acc -> x + acc end) #=> 6`
`Enum.each/2`	Iterates over a collection and applies a function to each element (for side effects). `Enum.each([1, 2, 3], fn x -> IO.puts(x) end)`

Concurrency & OTP

Processes

Spawning Processes	`spawn(fn -> # Process logic end)` Creates a new lightweight process.
Sending Messages	`send(pid, message)` Sends a message to a process identified by its PID.
Receiving Messages	`receive do pattern1 -> # Handle message matching pattern1 pattern2 -> # Handle message matching pattern2 end`

GenServer

Defining a GenServer	`defmodule MyServer do use GenServer # Define init, handle_call, handle_cast, handle_info, terminate end`
Starting a GenServer	`GenServer.start_link(MyServer, initial_state, options)`
`handle_call/3`	Handles synchronous requests. `{:reply, reply, new_state}`
`handle_cast/2`	Handles asynchronous requests. `{:noreply, new_state}`
`handle_info/2`	Handles other messages. `{:noreply, new_state}`

Supervisors

Defining a Supervisor

```
defmodule MySupervisor do
  use Supervisor

  def start_link(args) do

Supervisor.start_link(__M
ODULE__, args, strategy:
:one_for_one)
  end

  def init(_args) do
    children = [
      worker(MyWorker,
[])
    ]
    {:ok, children}
  end
end
```

Supervision Strategies

- `:one_for_one`: Restarts only the failing child.
- `:one_for_all`: Restarts all children when one fails.
- `:rest_for_one`: Restarts the failing child and all children started after it.