Introduction to Elixir

IEx - Elixir’s Interactive Shell

$ iex

Erlang/OTP 19 [erts-8.3] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false]

Interactive Elixir (1.4.4) - press Ctrl+C to exit (type h() ENTER for help)

iex(1)>

IEx - Elixir’s Interactive Shell

iex(1)> h()

- h/1 - prints help for the given module, function or macro
- i/1 - prints information about the data type of any given term
- v/0 - retrieves the last value from the history
- v/1 - retrieves the nth value from the history

Integers

iex(1)> 123

123

iex(2)> 0

0

iex(3)> -1_234

-1234

iex(4)> 1234567890123456789012345678901234567890

1234567890123456789012345678901234567890

Integers

```
iex(1)> 0xFFFF

65535

iex(2)> 0o0017

15

iex(3)> 0b1010

10
```

Integers

iex(1)> i(123)

Term

  123

Data type

  Integer

Reference modules

  Integer

Integers

```
iex(1)> 123

123

iex(2)> v()

123

iex(3)> v(-2)

123

iex(4)> v(1)

123
```

Floats

```
iex(1)> 1.0

1.0

iex(2)> 3.141592653589793

3.141592653589793

iex(3)> 0.3141592653589793e1

3.141592653589793

iex(4)> 1.234_567e-89

1.234567e-89
```

Addition

iex(1)> 1 + 1

2

iex(2)> 1 + 1.0

2.0

iex(3)> 1.0 + 1.0

2.0

iex(4)> h Kernel.+/2

Subtraction

```
iex(1)> 1 - 1

0

iex(2)> 1 - 1.0

0.0

iex(3)> 1.0 - 1.0

0.0

iex(4)> h Kernel.-/2
```

Multiplication

```
iex(1)> 1 * 1

1

iex(2)> 1 * 1.0

1.0

iex(3)> 1.0 * 1.0

1.0

iex(4)> h Kernel.*/2
```

Division

```
iex(1)> 1 / 1

1.0

iex(2)> 1 / 1.0

1.0

iex(3)> 1.0 / 1.0

1.0

iex(4)> h Kernel.//2
```

Parenthesis

```
iex(1)> 1 + 2 * 3

7

iex(2)> (1 + 2) * 3

9

iex(3)> 1 + (2 * 3)

7
```

Integer Division

```
iex(1)> div(1,  1)

1

iex(2)> div(1, 1.0)

** (ArithmeticError) bad argument in arithmetic expression

    :erlang.div(1, 1.0)

iex(2)> h Kernel.div/2
```

Problem

How many days has it been since 0 / 0 / 0 given that today is 7 / 6 / 2017?

- Today is the 158th day of 2017
- There are 365 days in a year
- Every 4 years has an extra day
- Except every 100 years does not have an extra
- Every 400 years has an extra day

Problem

How many days has it been since 0 / 0 / 0 given that today is 7 / 6 / 2017?

- Today is the 158th day of 2017
- 2017 is the 2018th year
- There are 365 days in a year
- Every 4 years has an extra day
- Every 100 years does not have an extra day
- Every 400 years has an extra day

```
iex(1)> 158 + (365 * 2017) + div(2016, 4) - div(2016, 100) + div(2016, 400)

736852
```

Problem

How many days has it been since 0 / 0 / 0 given that today is 7 / 6 / 2017?

- Today is the 158th day of 2017
- 2017 is the 2018th year
- There are 365 days in a year
- Every 4 years has an extra day
- Every 100 years does not have an extra day
- Every 400 years has an extra day

iex(1)> 158 + (365 * 2017) + (2016 / 4) - (2016 / 100) + (2016 / 400)

736851.88

Expressions continue onto new lines

```
iex(1)> 158 +

...(1)> (365 * 2017) +

...(1)> div(2016, 4) -

...(1)> div(2016, 100) +

...(1)> div(2016, 400)

736502
```

Comments

```
iex(1)> 158 +                  # days in 2017
...(1)> (365 * 2017) +         # days before 2017
...(1)> div(2016, 4) -         # extra day every 4 years
...(1)> div(2016, 100) +       # but not every 100 years
...(1)> div(2016, 400)         # extra day every 400 years
736502
```

IEx break

iex(1)> 158 +

...(1)> #iex:break

** (TokenMissingError) iex:1: incomplete expression

Binding

```
iex(1)> a = 123

123

iex(2)> b = a + 1

124
```

Pattern Matching

```
iex(1)> a = 123

123

iex(2)> b = a + 1

124

iex(3)> 123 = a

123

iex(4)> 123 = b

** (MatchError) no match of right hand side value: 124
```

Rebinding

```
iex(1)> a = 123

123

iex(2)> b = a + 1

124

iex(3)> a = 1

1

iex(4)> b

124
```

Pattern Matching

```
iex(1)> a = 123

123

iex(2)> b = a + 1

124

iex(3)> ^a = a

123

iex(4)> ^b = 123

** (MatchError) no match of right hand side value: 123
```

Anonymous Functions

```
iex(1)> fun = fn -> 123 end

#Function<20.118419387/0 in :erl_eval.expr/5>

iex(2)> fun.()

123

iex(3)> ident = fn x -> x end

#Function<6.118419387/1 in :erl_eval.expr/5>

iex(4)> ident.(123)

123
```

Anonymous Functions

```
iex(1)> fun = fn -> 123 end

#Function<20.118419387/0 in :erl_eval.expr/5>

iex(2)> fun()

** (CompileError) iex:2: undefined function fun/0

iex(2)> fun.(123)

** (BadArityError) #Function<20.118419387/0 in :erl_eval.expr/5> with arity 0
called with 1 argument (123)
```

Anonymous Functions

```
iex(1)> sum = fn x, y -> x + y end

#Function<12.118419387/2 in :erl_eval.expr/5>

iex(2)> sum.(1, 2)

3
```

Problem

Write an anonymous function to calculate the total cost of items when including VAT at a rate of 25%.

For example:

- Laptop KR10,000.00 -> KR12,500.00
- Milk KR8.00 -> KR10.00
- Book KR248.00 -> KR310.00

Problem

Write an anonymous function to calculate the total cost of items when including VAT at a rate of 25%.

For example:

- Laptop KR10,000.00 -> KR12,500.00
- Milk KR8.00 -> KR10.00
- Book KR248.00 -> KR310.00

```
iex(1)> add_vat = fn x -> x * 1.25 end      # using floats for money :(

iex(2)> add_vat.(248)

310.0
```

Atoms

Constants where the name is also the value.

iex(1)> :hello

:hello

iex(2)> :world?

:world?

iex(3)> :Bang!

:Bang!

Atoms

Constants where the name is also the value.

```
iex(1)> :my_name@FOO123

:my_name@FOO123

iex(2)> :*

:*

iex(3)> :"text with spaces"

:"text with spaces"
```

Pattern Matching With Anonymous Functions

```
iex(1)> to_atom = fn
...(1)>     1 -> :one
...(1)>     2 -> :two
...(1)> end
#Function<6.118419387/1 in :erl_eval.expr/5>
iex(2)> to_atom.(1)
:one
```

Pattern Matching With Anonymous Functions

```
iex(1)> to_atom = fn
...(1)>     1 -> :one
...(1)>     2 -> :two
...(1)> end
#Function<6.118419387/1 in :erl_eval.expr/5>
iex(2)> to_atom.(2)
:two
```

Pattern Matching With Anonymous

```
iex(1)> to_atom = fn

...(1)>     1 -> :one

...(1)>     2 -> :two

...(1)> end

#Function<6.118419387/1 in :erl_eval.expr/5>

iex(2)> to_atom.(3)

** (FunctionClauseError) no function clause matching in

:erl_eval."-inside-an-interpreted-fun-"/1
```

Pattern Matching With Anonymous Functions

```
iex(1)> to_atom = fn
...(1)>     0 -> :zero
...(1)>     _ -> :non_zero
...(1)> end
#Function<6.118419387/1 in :erl_eval.expr/5>
iex(2)> to_atom.(1)
:non_zero
```

Pattern Matching With Anonymous Functions

```
iex(1)> op = fn
...(1)>     :add, x, y -> x + y
...(1)>     :subtract, x, y -> x - y
...(1)> end
#Function<6.118419387/1 in :erl_eval.expr/5>
iex(2)> op.(:add, 1, 2)
3
```

Problem

Write an anonymous function to calculate the total cost of items when including VAT at a rate of 6% for a book, 12% for food and 25% for other goods.

For example:

- Laptop KR10,000.00 -> KR12,500.00
- Food KR8.00 -> KR8.96
- Book KR248.00 -> KR262.88

Problem

Write an anonymous function to calculate the total cost of items when including VAT at a rate of 6% for a book, 12% for food and 25% for other goods.

```
iex(1)> add_vat = fn

...(1)>     :book, x -> x * 1.06

...(1)>     :food, x -> x * 1.12

...(1)>     _, x -> x * 1.25

...(1)> end

iex(2)> 262.88 = add_vat.(:book, 248)
```

Booleans

The booleans values are `true` and `false`.

iex(1)> 1 == 1

true

iex(2)> 1 == 2

false

iex(3)> 1 != 2

true

Booleans

The booleans values are `true` and `false`.

iex(1)> 1 >= 1

true

iex(2)> 1.0 < 1.0

false

iex(3)> 1 <= 1.0

true

Booleans

The booleans values are `true` and `false`.

iex(1)> 1 === 1

true

iex(2)> 1.0 !== 1.0

false

iex(3)> 1 === 1.0

false

Booleans

The booleans values are `true` and `false`.

iex(1)> true and false

false

iex(2)> true or false

true

iex(3)> not true

false

Booleans

The booleans values are `true` and `false`, and are actually atoms.

iex(1)> :true

true

iex(2)> :false

false

Guards

Additional requirements to match a clause:

```
iex(1)> abs = fn
...(1)>     x when x >= 0 -> x
...(1)>     x when x < 0 -> -x
...(1)> end
#Function<6.118419387/1 in :erl_eval.expr/5>
iex(2)> abs.(1)
1
```

Guards

Additional requirements to match a clause:

```
iex(1)> abs = fn
...(1)>     x when x >= 0 -> x
...(1)>     x when x < 0 -> -x
...(1)> end
#Function<6.118419387/1 in :erl_eval.expr/5>
iex(2)> abs.(-2)
2
```

Guards

Additional requirements to match a clause:

```
iex(1)> abs = fn
...(1)>     x when x >= 0 -> x
...(1)>     x when x < 0 -> -x
...(1)> end
#Function<6.118419387/1 in :erl_eval.expr/5>
iex(2)> abs.(:hello)
:hello
```

Guards

Additional requirements to match a clause:

```
iex(1)> assert_number = fn

...(1)>     x when is_integer(x) or is_float(x) -> x

...(1)> end

#Function<6.118419387/1 in :erl_eval.expr/5>

iex(2)> assert_number.(:hello)

** (FunctionClauseError) no function clause matching in
:erl_eval."-inside-an-interpreted-fun-"/1
```

Problem

Write an anonymous function that determines whether a number is positive, negative or zero. For example:

- 1 -> :positive
- 0 -> :zero
- -1 -> :negative

Problem

Write an anonymous function that determines whether a number is positive, negative or zero. For example:

```
iex(1)> classify = fn
...(1)>     x when x > 0 -> :positive
...(1)>     x when x < 0 -> :negative
...(1)>     x when x == 0 -> :zero
...(1)> end
iex(2)> :zero = classify.(0.0)
```

case

Match on multiple possible values.

```
iex(1)> x = 1

iex(2)> case x do

...(2)>     nil -> :oops

...(2)>     int when is_integer(int) -> :integer

...(2)> end

:integer
```

case

Match on multiple possible values.

```
iex(1)> x = 1

iex(2)> case x do

...(2)>     nil -> :oops

...(2)>     x when is_integer(x) -> :integer

...(2)> end

:integer
```

case

Match on multiple possible values.

```
iex(1)> x = 1

iex(2)> case x do

...(2)>     nil -> :oops

...(2)>     ^x -> :one

...(2)>     int when is_integer(int) -> :integer

...(2)> end

:one
```

Problem

Write an anonymous function using case to calculate the total cost of items when including VAT at a rate of 6% for a book, 12% for food and 25% for other goods.

For example:

- Laptop KR10,000.00 -> KR12,500.00
- Food KR8.00 -> KR8.96
- Book KR248.00 -> KR262.88

Problem

```
iex(1)> add_vat = fn item, price ->
...(1)>     case item do
...(1)>             :book -> price * 1.06
...(1)>             :food -> price * 1.12
...(1)>             _ -> price * 1.25
...(1)>     end
...(1)> end
iex(2)> 8.96 = add_vat.(:food, 8)
```

nil

For lack of a value.

iex(1)> nil

nil

iex(2)> is_atom(nil)

true

iex(3)> is_nil(nil)

true

Truthy and Falsy

`false` and `nil` are falsy, everything else is truthy.

```
iex(1)> 1 && 2

2

iex(2)> true || 2

true

iex(3)> nil || 1

1
```

cond

Find the first matching condition.

```
iex(1)> cond do
...(1)>     nil -> :never_truthy
...(1)>     1 -> :truthy
...(1)>     true -> :never_runs
...(1)> end
:always_truthy
```

if

Check a single condition.

```
iex(1)> x = 1

iex(2)> if x == 1 do

...(2)>     :one

...(2)> else

...(2)>     :not_one

...(2)> end

:one
```

if

Check a single condition.

```
iex(1)> x = 2

iex(2)> if x == 1 do

...(2)>     :one

...(2)> end

nil
```

unless

Check a single condition.

```
iex(1)> x = 2

iex(2)> unless x == 1 do

...(2)>     :not_one

...(2)> end

:not_one
```

Problem

Write an anonymous function using cond to calculate the total cost of items when including VAT at a rate of 6% for a book, 12% for food and 25% for other goods.

For example:

- Laptop KR10,000.00 -> KR12,500.00
- Food KR8.00 -> KR8.96
- Book KR248.00 -> KR262.88

Problem

```
iex(1)> add_vat = fn item, price ->
...(1)>     cond do
...(1)>         item == :book -> price * 1.06
...(1)>         item == :food -> price * 1.12
...(1)>         true -> price * 1.25
...(1)>     end
...(1)> end
iex(2)> 1.25e4 = add_vat.(:laptop, 10_000)
```

Modules

Name and group functions.

```
iex(1)> defmodule Hello do
...(1)>     def hello() do
...(1)>         :hello_world
...(1)>     end
...(1)> end
{:module, Hello, … }
iex(2)> :hello_world = Hello.hello()
```

Modules

```
$ cat hello.ex

defmodule Hello do
  def hello() do
    :hello_world
  end
end

iex(1)> c("hello.ex")

[Hello]
```

Modules

```
defmodule Math do
    def add(x, y) do
        x + y
    end
end
```

```
iex(1)> Math.add(1, 2)
3
```

Functions

```
def classify(int) when is_integer(int) do
    :integer
end

def classify(atom) when is_atom(atom) do
    :atom
end
```

Functions

```
def classify(nil), do: nil

def classify(int) when is_integer(int) do
    :integer
end

def classify(atom) when is_atom(atom) do
    :atom
end
```

Problem

Write a module called VAT with an add_vat/2 function to calculate the total cost of items when including VAT at a rate of 6% for a book, 12% for food and 25% for other goods.

For example:

- Laptop KR10,000.00 -> KR12,500.00
- Food KR8.00 -> KR8.96
- Book KR248.00 -> KR262.88

Problem

```
defmodule VAT do

  def add_vat(:book, price), do: price * 1.06

  def add_vat(:food, price), do: price * 1.12

  def add_vat(_, price), do: price * 1.25

end

iex(1)> [VAT] = c("vat.ex")

iex(2)> 5.0e4 = VAT.add_vat(:party, 40_000)
```

Strings

UTF-8 encoded strings, surrounded by double quotes.

iex(1)> "hello"

"hello"

iex(2)> "Fizz" <> "Buzz"

"FizzBuzz"

iex(3)> "José Valim"

"José Valim"

Strings

UTF-8 encoded strings, surrounded by double quotes.

```
iex(1)> String.starts_with?("hello world", "hello")

true

iex(2)> String.reverse("Fizz" <> "Buzz")

"zzuBzziF"

iex(2)> String.upcase("José Valim")

"JOSÉ VALIM"
```

Binaries

A sequence of bytes, that is possibly a string.

```
iex(1)> <<"hello">>

"hello"

iex(2)> <<0xFF, 0x12>>

<<255, 18>>

iex(3)> <<2017::signed-little-integer-64>>

<<225, 7, 0, 0, 0, 0, 0, 0>>
```

Binaries

A sequence of bytes, that is possibly a string.

```
iex(1)> <<5.0::float>>

<<64, 20, 0, 0, 0, 0, 0, 0>>

iex(2)> <<float :: float>> = <<64, 20, 0, 0, 0, 0, 0, 0>>

<<64, 20, 0, 0, 0, 0, 0, 0>>

iex(3)> float

5.0
```

Binaries

A sequence of bytes, that is possibly a string.

```
iex(1)> <<5, "hello">>

<<5, 104, 101, 108, 108, 111>>

iex(2)> <<len, message :: binary-size(len)>> = v(1)

<<5, 104, 101, 108, 108, 111>>

iex(3)> message

"hello"
```

Problem

Decode the following binary:

<<5, 0, 105, 32, 97, 116, 101, 64, 32, 0, 0, 99, 97, 107, 101, 115>>

- The first 2 bytes are a little endian signed integer
- The next bytes are a string with length defined by the first 2 bytes
- The next 4 bytes are a float
- The remainder is a string

Problem

iex(1)> data = <<5, 0, 105, 32, 97, 116, 101, 64, 32, 0, 0, 99, 97, 107, 101, 115>>

iex(2)> <<len::little-unsigned-16, string1::binary-size(len), float::float-32, string2::binary>> = <<5, 0, 105, 32, 97, 116, 101, 64, 32, 0, 0, 99, 97, 107, 101, 115>>

iex(3)> "i ate" = string1

iex(4)> 2.5 = float

iex(5)> "cakes" = string2

Lists

Linked list: ordered set of elements where each element points to successor.

```
iex(1)> [1, 2, 3]

[1, 2, 3]

iex(2)> [1 | [2, 3]]

[1, 2, 3]

iex(3)> [1 | [2 | [3 | []]]]

[1, 2, 3]
```

Lists

Linked list: ordered set of elements where each element points to successor.

```
iex(1)> [a, b, c] = [1, 2, 3]

iex(2)> [^a | tail] = [1, 2, 3]

iex(3)> [2, 3] = tail

iex(4)> [^b, ^c | []] = tail

iex(5)> [^a | [ ^b | [^c]]] = [1, 2, 3]
```

Recursion

Divide work into smaller parts that can be solved by the same function.

```
defmodule Recursion do

    def len([]), do: 0

    def len([_ | tail]), do: 1 + len(tail)

end
```

Tail Call Optimisation

A tail, or last, call causes the calling function to remove itself from the call stack.

```
defmodule TailCall do
    def multi_duplicate(string, a, b) do
        n = a * b
        String.duplicate(string, n) # tail call removes multi_duplicate/3 from stack
    end
end
```

Tail Recursion

Combine recursion and tail calls.

```
defmodule TailRecursion do

    def len(list), do: len(list, 0)

    defp len([], acc), do: acc                    # defp is private to the module

    defp len([_ | tail], acc), do: len(tail, acc+1)

end
```

Problem

Write a module called Sum with a sum/1 function to calculate the sum of a list of integers.

For example:

- [] -> 0
- [1, 2] -> 3
- [4, 9, 17] -> 30

Problem

Write a module called Sum with a sum/1 function to calculate the sum of a list of integers.

```
defmodule Sum do

    def sum(list), do: sum(list, 0)

    defp sum([], acc), do: acc

    defp sum([int | tail], acc), do: sum(tail, acc+int)

end
```

Problem

Write a module called Factorial with a factorial/1 function to calculate the factorial of an integer.

For example:

- 0 -> 1
- 1 -> 1
- 2 -> 2 * 1 -> 2
- 3 -> 3 * 2 * 1 -> 6
- 4 -> 4 * 3 * 2 * 1 -> 24

Problem

Write a module called Factorial with a factorial/1 function to calculate the factorial of an integer.

```
defmodule Factorial do

    def factorial(n) when is_integer(n) and n >= 0, do: factorial(n, 1)

    defp factorial(0, acc), do: acc

    defp factorial(n, acc), do: factorial(n-1, n * acc)

end
```

Tuple

Fixed size ordered set of elements.

```
iex(1)> {1, 2, 3}

{1, 2, 3}

iex(2)> {:ok, :hello}

{:ok, :hello}

iex(3)> {}

{}
```

Tuple

Fixed size ordered set of elements.

```
iex(1)> put_elem({1, 2, 3}, 0, :put)

{:put, 2, 3}

iex(2)> elem({:ok, :hello}, 1)

:hello

iex(3)> Tuple.to_list({})

[]
```

Keywords

Key-value structure using lists of 2-tuples with atom keys.

iex(1)> [key: :value, another_key: :and_value]

[key: :value, another_key: :and_value]

iex(2)> [{:key, :value}, {:another_key, :and_value}]

[key: :value, another_key: :and_value]

iex(3)> [key: :value, key: :duplicate]

[key: :value, key: :duplicate]

Keywords

Key-value structure using lists of 2-tuples with atom keys.

```
iex(1)> Keyword.put([key: :value],  another_key: :and_value)
[another_key: :and_value, key: :value]
iex(2)> Keyword.get([{:key, :value},  {:another_key, :and_value}], :key)
:value
iex(3)> Keyword.pop([key: :value, key: :duplicate, another_key: :and_value], :key)
{:value, [another_key: :and_value]}
```

Maps

Unordered collection of key-value pairs where only one value per key.

```
iex(1)> %{key: :value, another_key: :and_value}

%{another_key: :and_value, key: :value}

iex(2)> %{"key" => :value, "another_key" => "and_value"}

%{"another_key" => "and_value", "key" => :value}

iex(3)> %{"key" => :value, "key" => :dup}

%{"key" => :dup}
```

Maps

Collection of key-value pairs where only one value per key.

```
iex(1)> map = %{key: :value}

%{key: :value}

iex(2)> %{map | key: :new_value}    # update existing key

%{key: :new_value}

iex(3)> %{map | "another_key" => :another_value}      # does not allow insert

** (KeyError) key "another_key" not found in: %{key: :value}
```

Maps

Collection of key-value pairs where only one value per key.

iex(1)> %{key: value} = %{key: :value, another_key: :and_value}

%{key: :value, another_key: :and_value}

iex(2)> value

:value

iex(3) = %{key: :value, another_key: :and_value}

%{key: :value, another_key: :and_value}

Maps

Collection of key-value pairs where only one value per key.

```
iex(1)> Map.fetch(%{key: :value, another_key: :and_value}, :key)

{:ok, :value}

iex(2)> Map.put(%{key: :value}, :another_key, :and_value)

%{key: :value, another_key: :and_value}

iex(3)> Map.delete(%{key: :value}, :key)

%{}
```

Problem

Write a module called Fetcher with a fetch/3 function that takes a map as first argument, a two keys as the remaining arguments, and returns {:ok, value1, value2} if both are present, otherwise :error.

For example:

- Fetcher.fetch(%{}, :foo, :bar) -> :error
- Fetcher.fetch(%{foo: :buzz}, :foo, :bar) -> :error
- Fetcher.fetch(%{foo: :buzz, :bar: :quux}, :foo, :bar) -> {:ok, :buzz, :quux}

Problem

```
defmodule Fetcher do
    def fetch(map, key1, key2) do
        case map do
            %{^key1 => val1, ^key2 => val2} -> {:ok, val1, val2}
            %{} -> :error
        end
    end
end
```

with

Handle multiple pattern matches sequentially without nesting.

```
iex(1)> with {:ok, min} <- Keyword.fetch([min: 10], :min),
...(1)>          {i, _} when i >= min <- Integer.parse("20") do
...(1)>            {:ok, i}
...(1)> else
...(1)>     _ -> :error
...(1)> end
{:ok, 20}
```

with

Handle multiple pattern matches sequentially without nesting.

```
iex(1)> with {:ok, min} <- Keyword.fetch([min: 10], :min),
...(1)>           {i, _} when i >= min <- Integer.parse("1") do
...(1)>      {:ok, i}
...(1)> else
...(1)>      _ -> :error
...(1)> end
:error
```

Problem

Write a module called Fetcher with a fetch/3 function that takes a map as first argument, a two keys as the remaining arguments, and returns {:ok, value1, value2} if both are present, otherwise :error. Use with!

For example:

- Fetcher.fetch(%{}, :foo, :bar) -> :error
- Fetcher.fetch(%{foo: :buzz}, :foo, :bar) -> :error
- Fetcher.fetch(%{foo: :buzz, :bar: :quux}, :foo, :bar) -> {:ok, :buzz, :quux}

Problem

```
defmodule Fetcher do
    def fetch(map, key1, key2) do
        with {:ok, val1} <- Map.fetch(map, key1),
             {:ok, val2} <- Map.fetch(map, key2) do
             {:ok, val1, val2}
        end
    end
end
```

Enum

Module to transform, filter, sort, group and fetch items from lists, maps and other enumerables. Tuples, atoms and binaries are not enumerable.

```
iex(1)> Enum.map([1, 2, 3], fn x -> x + 3 end)

[4, 5, 6]

iex(2)> Enum.reduce([1, 2, 3], 0, fn x, acc -> x + acc end)

6

iex(3)> Enum.filter([1, 2, 3], fn rem(x, 2) == 1 end)

[1, 3]
```

Enum

Module to transform, filter, sort, group and fetch items from lists, maps and other enumerables. Tuples, atoms and binaries are not enumerable.

```
iex(1)> Enum.map({1, 2, 3}, fn x -> x + 3 end)

** (Protocol.UndefinedError) protocol Enumerable not implemented for {1, 2, 3}

iex(2)> Enum.reduce(:atom, 0, fn x, acc -> x + acc end)

** (Protocol.UndefinedError) protocol Enumerable not implemented for :atom

iex(3)> Enum.filter(<<1, 2, 3>>, fn rem(x, 2) == 1 end)

** (Protocol.UndefinedError) protocol Enumerable not implemented for <<1, 2, 3>>
```

Pipe

Takes output from left expressions and uses as input as first argument to right.

```
iex(3)> %{foo: 1, bar: 2} |> Enum.map(fn {k, v} -> 2 * v end)  |> Enum.sum()
6
iex(2)> "FizzBuzz" |> String.reverse() |> String.to_atom()
:zzuBzziF
iex(3)> 1 |> Kernel.*(2) |> Kernel.-(3) |> Kernel.+(4)
3     # ( ( 1 * 2 ) - 3 ) + 4
```

Stream

Lazy operations on, or to generate, enumerables. The transformation etc are delayed until an Enum function call is made.

iex(1)> Stream.map([1,2,3], fn(x) -> x * -2 end)

#Stream<[enum: [1, 2, 3], funs: [#Function<47.122079345/1 in Stream.map/2>]]>

iex(2)> Stream.filter(v(-1), fn(x) -> x < -2 end)

#Stream<[enum: [1, 2, 3], funs: [...]>

iex(3)> Enum.sort(v(-1))

[-6, -4]

Capture Syntax

Short hand anonymous function syntax for partial application.

```
iex(1)> &(&1 - 2)

#Function<6.118419387/1 in :erl_eval.expr/5>

iex(2)> Enum.map([1,2,3], &(&1 + 1))

[2, 3, 4]

iex(3)> &(&1 - &2)

&:erlang.-/2
```

Capture Syntax

Short hand anonymous function syntax for partial application.

```
iex(1)> &Kernel.-/2

&:erlang.-/2

iex(2)> &Map.put(%{key: :value}, &1, &2)

#Function<12.118419387/2 in :erl_eval.expr/5>

iex(3)> &put_elem(&1, 2, :hello)

#Function<6.118419387/1 in :erl_eval.expr/5>
```

Problem

Write a module called Shopping with a calculate/1 function to calculate the total cost of a list of items and a list of just individual items without price or quantity.

- Input is in form [{item, price, quantity}]:
- Include VAT at a rate of 6% for a book, 12% for food and 25% for other goods
- Books have a 10% sales discount
- The return is {[item, ...], total}

Problem

```
defmodule Shopping do

    def calculate(list) do

        list

            |> Stream.map(&sales_discount/1)

            |> Stream.map(&add_vat/1)

            |> Stream.flat_map(&duplicate/1)

            |> Enum.map_reduce(0, &calc_price/2)

    end
```

Problem

```
    defp sales_discount({:book, price}), do: {:book, price * 0.9}

    defp sales_discount(item), do: item

    defp add_vat({:book, price, count}), do: {:book, price * 1.06, count}

    defp add_vat({:food, price, count}), do: {:food, price * 1.12, count}

    defp add_vat({item, price, count}), do: {item, price * 1.25, count}

    defp duplicate({item, price, count}), do: List.duplicate({item, price}, count)

    defp calc_price({item, price}, total), do: {item, total + price}

end
```

Ranges

An enumerable that generates integers between two integer values, exclusive.

iex(1)> 1..5

1..5

iex(2)> Enum.to_list(1..5)

[1, 2, 3, 4, 5]

iex(3)> 1..-5

1..-5

MapSet

A set of unique values with an opaque structure that uses a map underneath. Also an enumerable.

iex(1)> MapSet.new([1, 2, 3])

#MapSet<[1, 2, 3]>

iex(2)> Enum.to_list(v(-1))

[1, 2, 3]

iex(3)> MapSet.union(v(1), MapSet.new([4]))

#MapSet<[1, 2, 3, 4]>

for

Comprehensions provide syntactic sugar over common mapping and filtering Enum operations.

```
iex(1)> for int <- 1..5, do: -int

[-1, -2, -3, -4, -5]

iex(2)> for x <- 1..2, y <- 3..4, do: {x, y}

[{1, 3}, {1, 4}, {2, 3}, {2, 4}]

iex(3)> for x <- 1..2, y <- x..3, do: y

[1, 2, 3, 2, 3]
```

for

Comprehensions provide syntactic sugar over common mapping and filtering Enum operations.

```
iex(1)> for int <- 1..5, rem(int, 2) == 0, do: -int

[-2, -4]

iex(2)> for x <- 1..2, y <- 3..4, not (x == 1 and y== 3), do: {x, y}

[{1, 4}, {2, 3}, {2, 4}]

iex(3)> for x <- 1..2, y <- x..3, :truthy, do: y

[1, 2, 3, 2, 3]
```

Problem

Using a for comprehension find all possible pairs of the multiples of 3 between 1 and 15 and the MapSet of :apple, :pear and :orange.

Problem

Using a for comprehension find all possible pairs of the multiplies of 3 between 1 and 15 and the MapSet of :apple, :pear and :orange.

iex(1) > for x <- 1..15, rem(x, 3) == 0, fruit <- MapSet.new([:apple, :pear, :orange]), do: {x, fruit}

[{3, :apple}, {3, :orange}, {3, :pear}, {6, :apple}, {6, :orange}, {6, :pear},

{9, :apple}, {9, :orange}, {9, :pear}, {12, :apple}, {12, :orange},

{12, :pear}, {15, :apple}, {15, :orange}, {15, :pear}]

Protocols

Modules to handle polymorphism.

```
defprotocol Size do
    def size(term)
end

defimpl Size, for: Tuple do
    def size(tuple), do: tuple_size(tuple)
end
```

Protocols

Modules to handle polymorphism.

```
defprotocol Size do
    def size(term)
end

defimpl Size, for: Map do
    def size(map), do: map_size(map)
end
```

Protocols

Modules to handle polymorphism.

```
iex(1)> c "size.ex"

[Size.Map, Size.Tuple, Size]

iex(2)> Size.size({1, 2, 3})

3

iex(3)> Size.size(%{key: :value})

1
```

Structs

Maps with fixed atom keys and compile time validation of keys.

```
defmodule MyStruct do
    defstruct [:list, :extra]
    def new(), do: %MyStruct{list: []}
    def grow(%MyStruct{list: list} = struct) do
        %MyStruct{struct | list: [:padding | list]}
    end
end
```

Structs

Maps with fixed atom keys and compile time validation of keys.

```
defmodule MyStruct do
    defstruct [:list, :extra]
    def new(), do: %MyStruct{list: []}
    def grow(%MyStruct{list: list} = struct) do
        %MyStruct{struct | list: [:padding | list]}
    end
end
```

Structs

Maps with fixed atom keys and compile time validation of keys.

```
iex(1)> MyStruct.new()

%MyStruct{extra: nil, list: []}

iex(2)> MyStruct.grow(v(-1))

%MyStruct{extra: nil, list: [:padding]}

iex(3)> MyStruct.grow(v(-1))

%MyStruct{extra: nil, list: [:padding, :padding]}
```

Structs

Maps with fixed atom keys and compile time validation of keys.

```
defimpl Size, for: MyStruct do
    def size(%MyStruct{list: list}), do: length(list)
end
```

```
iex(1)> Size.size(MyStruct.new())
0
```

Problem

Write the Empty protocol with a single callback empty?/1 that returns true if the term is empty and false otherwise. Implement for maps, lists and MapSet.

Problem

Write the Empty protocol with a single callback empty?/1 that returns true if the term is empty and false otherwise. Implement for maps, lists and MapSet.

```
defprotocol Empty do

  def empty?(term)

end

defimpl Empty, for: Map do

  def empty?(map), do: map_size(map) == 0

end
```

Problem

Write the Empty protocol with a single callback empty?/1 that returns true if the term is empty and false otherwise. Implement for maps, lists and MapSet.

```
defprotocol Empty do

  def empty?(term)

end

defimpl Empty, for: List do

  def empty?(list), do: list == []

end
```

Problem

Write the Empty protocol with a single callback empty?/1 that returns true if the term is empty and false otherwise. Implement for maps, lists and MapSet.

```
defprotocol Empty do

  def empty?(term)

end

defimpl Empty, for: MapSet do

  def empty?(map_set), do: MapSet.size(map_set) == 0

end
```

Problem

```
iex(1)> Empty.empty?([])

true

iex(2)> Empty.empty?(%{foo: :bar})

false

iex(3)> Empty.empty?(MapSet.new([1,2,3]))

false
```

Mix

Build tool for Elixir.

```
$ mix help

mix cmd              # Executes the given command

mix compile          # Compiles source files

mix deps.get         # Gets all out of date dependencies

mix new              # Creates a new Elixir project

mix test             # Runs a project's tests

iex -S mix           # Starts IEx and runs the default task
```

Mix

Build tool for Elixir.

$ mix new first_project

* creating README.md

* creating .gitignore

* creating mix.exs

* creating config

* creating config/config.exs

* creating lib

* creating lib/first_project.ex

* creating test

* creating test/test_helper.exs

* creating test/first_project_test.exs

Your Mix project was created successfully.

ExUnit

```
$ cat test/first_project_test.exs

defmodule FirstProjectTest do
        use ExUnit.Case
        doctest FirstProject
        test "the truth" do
                assert 1 + 1 == 2
        end
end
```

Problem

Create a mix project called `hotel` with `mix new hotel` and write a module to help eliminated hotels from a list of hotels. Reject hotels that are not interesting to stay at because they are both more expensive and further away than another hotel.

- defstruct [:price, :distance, :name]
- Add tests in test/hotel_test.exs
- Tip: See Enum.sort_by/2