

C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes










Topic 5: Control flow, Enum VS Stream, comprehensions

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

The copyright to this original work is held by Alex Ufkes. Students registered in course C/CPS 506 can use this material for the purposes of this course but no other use is permitted, and there can be no sale or transfer or use of the work for any other purpose without explicit permission of Alex Ufkes.

Course Administration (CCPS)

        Alexander Ufkes 

Content Grades Assessment ▾ Communication ▾ Resources ▾ Classlist Course Admin

- Elixir labs due on Feb 27

This Week

More Advanced Elixir:

- Enum & Stream
- Control flow, keyword lists
- List comprehensions

Let's Get Started!

Enum

Enum

A set of algorithms for enumeration over enumerables!

Enum applies functions to lists in various ways. We will see a few:

Enum.all?

Entire collection must evaluate to true for a given condition

Enum.any?

Any value in the collection must evaluate true

Enum.map

Apply a function to every element in the collection

More: <https://elixirschool.com/en/lessons/basics/enum/>

Enum.all?

Entire collection must evaluate to true for a given condition

The image shows a Notepad++ window with the following Elixir code:

```
1 vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2
3 IO.puts Enum.all?(vals, fn(n) -> n < 5 end)
4 IO.puts Enum.all?(vals, fn(n) -> n >= 0 end)
5 IO.puts Enum.all?(vals, fn(n) -> n < 18 end)
6
7
```

Annotations in the image:

- A green box labeled "Pass list as first arg" has an arrow pointing to the `vals` argument in line 3.
- A blue box labeled "Anon function as second arg" has a bracket pointing to the anonymous function `fn(n) -> n < 5 end` in line 3.

The Command Prompt window shows the output of running the code:

```
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
false
true
true
C:\Users\aufke\Desktop\ElixirScripts>
```


Enum.all? We can do it!

```
1 defmodule MyEnum do
2   def all(list, f) do all(list, f, true) end
3   def all([], _, res) do res end
4   def all([h|t], f, res) do all(t, f, f.(h) and res) end
5 end
6
7 vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
8 f1 = fn(a) -> a < 5 end
9 f2 = fn(a) -> a >= 0 end
10 f3 = fn(a) -> a < 18 end
11
12 IO.puts MyEnum.all(vals, f1)
13 IO.puts MyEnum.all(vals, f2)
14 IO.puts MyEnum.all(vals, f3)
15
```

Tail recursive!

- **and** the function result with the running Boolean result.
- If we hit an element for which **f.(h)** is false, the entire running Boolean becomes false.


Enum.all? We can do it!

```
1 defmodule MyEnum do
2   def all(list, f) do all(list, f, true) end
3   def all([], _, res) do res end
4   def all([h|t], f, res) do all(t, f, f.(h) and res) end
5 end
6
7 vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
8 f1 = fn(a) -> a < 5 end
9 f2 = fn(a) -> a >= 0 end
10 f3 = fn(a) -> a < 18 end
11
12 IO.puts MyEnum.all(vals, f1)
13 IO.puts MyEnum.all(vals, f2)
14 IO.puts MyEnum.all(vals, f3)
15
```

Windows PowerShell

```
PS D:\GoogleDrive\Teaching - Ryerson\CCPS 506\_S_20
false
true
true
PS D:\GoogleDrive\Teaching - Ryerson\CCPS 506\_S_20
```

Enum.all? Short Circuit?

```
defmodule MyEnum do
  def all(list, f) do all(list, f, true) end
   defp all(_, _, false) do false end
  defp all([], _, res) do res end
  defp all([h|t], f, res) do all(t, f, f.(h) and res) end
end
```

- No need to continue if res becomes false.
- **false** and anything = **false**
- Can also make tail recursive functions private

Enum.any?

Any value in collection must evaluate to true for a given condition

```
C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
simple.exs Modules.ex
1 vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2
3 IO.puts Enum.any?(vals, fn(n) -> n == 2.2 end)
4 IO.puts Enum.any?(vals, fn(n) -> n > 5 end)
5 IO.puts Enum.any?(vals, fn(n) -> n < 18 end)
6
7
```

```
Command Prompt
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
false
true
true
C:\Users\aufke\Desktop\ElixirScripts>
```

Enum.any? We can do it!

```
1 defmodule MyEnum do
2   def all(list, f) do all(list, f, true) end
3   def all([], _, res) do res end
4   def all([h|t], f, res) do all(t, f, f.(h) and res) end
5
6   def any(list, f) do any(list, f, false) end
7   def any([], _, res) do res end
8   def any([h|t], f, res) do any(t, f, f.(h) or res) end
9 end
10
11 vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
12 f1 = fn(a) -> a < 0 end
13 f2 = fn(a) -> a >= 0 end
14 f3 = fn(a) -> a > 18 end
15
16 IO.puts MyEnum.any(vals, f1)
17 IO.puts MyEnum.any(vals, f2)
18 IO.puts MyEnum.any(vals, f3)
```

Very similar to MyEnum.all

- Initialize res to false
- Any true value from function f will turn result true.
- We are **ORing** instead of **ANDing**

Enum.any? We can do it!

```
1 defmodule MyEnum do
2   def all(list, f) do all(list, f, true) end
3   def all([], _, res) do res end
4   def all([h|t], f, res) do all(t, f, f.(h) and res) end
5
6   def any(list, f) do any(list, f, false) end
7   def any([], _, res) do res end
8   def any([h|t], f, res) do any(t, f, f.(h) or res) end
9 end
10
11 vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
12 f1 = fn(a) -> a < 0 end
13 f2 = fn(a) -> a >= 0 end
14 f3 = fn(a) -> a > 18 end
15
16 IO.puts MyEnum.any(vals, f1)
17 IO.puts MyEnum.any(vals, f2)
18 IO.puts MyEnum.any(vals, f3)
```

Windows PowerShell

```
PS D:\GoogleDrive\Teaching - Ryerson\CCPS 506\_S_201
false
true
false
PS D:\GoogleDrive\Teaching - Ryerson\CCPS 506\_S_201
```

Enum.any? Short Circuit?

```
defmodule MyEnum do
  def all(list, f) do all(list, f, false) end
  defp all(_, _, true) do true end
  defp all([], _, res) do res end
  defp all([h|t], f, res) do all(t, f, f.(h) or res) end
end
```

Enum.map

Very useful! Apply a function to every element

```
Erlang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10]
Interactive Elixir (1.6.5) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
iex(2)> Enum.map(vals, fn(n) -> n+1 end)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
iex(3)> Enum.map(vals, fn(n) -> n*2 end)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
iex(4)> Enum.map(vals, fn(n) -> n*n end)
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
iex(5)> █
```


Enum.map: We can do it!

```
4 def all([h|t], f, res) do all(t, f, f.(h) and res) end
5
6 def any(list, f) do any(list, f, false) end
7 def any([], _, res) do res end
8 def any([h|t], f, res) do any(t, f, f.(h) or res) end
9
10 def map(list, f) do map(list, f, []) end
11 def map([], _, res) do res end
12 def map([h|t], f, res) do map(t, f, res ++ [f.(h)]) end
13 end
14
15 vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
16 f1 = fn(a) -> a + 1 end
17 f2 = fn(a) -> a * 2 end
18 f3 = fn(a) -> a * a end
19
20 IO.inspect MyEnum.map(vals, f1), charlists: :as_lists
21 IO.inspect MyEnum.map(vals, f2), charlists: :as_lists
22 IO.inspect MyEnum.map(vals, f3), charlists: :as_lists
```

- Result initialized as an empty list
- Concatenate [f.(h)] to the running result list

Enum.map: We can do it!

```
4 def all([h|t], f, res) do all(t, f, f.(h) and res) end
5
6 def any(list, f) do any(list, f, false) end
7 def any([], _, res) do res end
8 def any([h|t], f, res) do any(t, f, f.(h) or res) end
9
10 def map(list, f) do map(list, f, []) end
11 def map([], _, res) do res end
12 def map([h|t], f, res) do map(t, f, res ++ [f.(h)]) end
13 end
14
15 vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
16 f1 = fn(a) -> a + 1 end
17 f2 = fn(a) -> a * 2 end
18 f3 = fn(a) -> a * a end
19
20 IO.inspect MyEnum.map(vals, f1), charl
21 IO.inspect MyEnum.map(vals, f2), charl
22 IO.inspect MyEnum.map(vals, f3), charl
```

Windows PowerShell

```
PS D:\GoogleDrive\Teaching - Ryerson\CCPS 506\_S_2018\
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
PS D:\GoogleDrive\Teaching - Ryerson\CCPS 506\_S_2018\
```

Enum.map: We can do it!

```
4 def all([h|t], f, res) do all(t, f, f.(h) and res) end
5
6 def any(list, f) do any(list, f, false) end
7 def any([], _, res) do res end
8 def any([h|t], f, res) do any(t, f, f.(h) or res) end
9
10 def map(list, f) do map(list, f, []) end
11 def map([], _, res) do res end
12 def map([h|t], f, res) do map(t, f, res ++ [f.(h)]) end
13 end
14
15 vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
16 f1 = fn(a) -> a + 1 end
17 f2 = fn(a) -> a * 2 end
18 f3 = fn(a) -> a * a end
19
20 IO.inspect MyEnum.map(vals, f1), charlists: :as_lists
21 IO.inspect MyEnum.map(vals, f2), charlists: :as_lists
22 IO.inspect MyEnum.map(vals, f3), charlists: :as_lists
```

Enum.reduce

Distill collection to single value based on some function

```
C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
simple.exs Modules.ex
1 vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2
3 IO.puts Enum.reduce(vals, fn(n, acc) -> n+acc end)
4 IO.puts Enum.reduce(vals, fn(n, acc) -> n*acc end)
5 IO.puts Enum.reduce(vals, fn(n, acc) -> n-acc end)
6 IO.puts Enum.reduce(vals, fn(n, acc) -> acc-n end)
7
8
```

```
Command Prompt
C:\Users\aufke\Desktop\ElixirScripts
45
0
5
-45
C:\Users\aufke\Desktop\ElixirScripts
```

- **acc** is the running value
- By default, initialized to first element in list

Enum.reduce

Distill collection to single value based on some function

```
C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
simple.exs Modules.ex
1 vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
2
3 IO.puts Enum.reduce(vals, fn(n, acc) -> n+acc end)
4 IO.puts Enum.reduce(vals, fn(n, acc) -> n*acc end)
5 IO.puts Enum.reduce(vals, fn(n, acc) -> n-acc end)
6 IO.puts Enum.reduce(vals, fn(n, acc) -> acc-n end)
7
8
```

```
Command Prompt
C:\Users\aufke\Desktop\ElixirScripts
45
0
5
-45
C:\Users\aufke\Desktop\ElixirScripts
```

$$(9 - (8 - (7 - (6 - (5 - (4 - (3 - (2 - (1 - acc)...) VS (... (acc - 1) - 2) - 3) - 4) - 5) - 6) - 7) - 8) - 9)$$

Enum.reduce: We can do it!

```
10 def map(list, f) do map(list, f, []) end
11 def map([], _, res) do res end
12 def map([h|t], f, res) do map(t, f, res ++ [f.(h)]) end
13
14 def reduce(list, f) do reduce((tl list), f, hd list) end
15 def reduce([], _, res) do res end
16 def reduce([h|t], f, res) do reduce(t, f, f.(h, res)) end
17 end
18
19 vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
20 f1 = fn(n, acc) -> n+acc end
21 f2 = fn(n, acc) -> n*acc end
22 f3 = fn(n, acc) -> n-acc end
23 f4 = fn(n, acc) -> acc-n end
24
25 IO.puts MyEnum.reduce(vals, f1)
26 IO.puts MyEnum.reduce(vals, f2)
27 IO.puts MyEnum.reduce(vals, f3)
28 IO.puts MyEnum.reduce(vals, f4)
```

- Result initialized as head of list
- Pass head of list and current result into f

Enum.reduce: We can do it!

```
10 def map(list, f) do map(list, f, []) end
11 def map([], _, res) do res end
12 def map([h|t], f, res) do map(t, f, res ++ [f.(h)]) end
13
14 def reduce(list, f) do reduce((tl list), f, hd list) end
15 def reduce([], _, res) do res end
16 def reduce([h|t], f, res) do reduce(t, f, f.(h, res)) end
17 end
18
19 vals = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
20 f1 = fn(n, acc) -> n+acc end
21 f2 = fn(n, acc) -> n*acc end
22 f3 = fn(n, acc) -> n-acc end
23 f4 = fn(n, acc) -> acc-n end
24
25 IO.puts MyEnum.reduce(vals, f1)
26 IO.puts MyEnum.reduce(vals, f2)
27 IO.puts MyEnum.reduce(vals, f3)
28 IO.puts MyEnum.reduce(vals, f4)
```

Windows PowerShell

```
PS D:\GoogleDrive\Teaching - Ryerson\CCPS 506\_S_2018\I
45
0
5
-45
PS D:\GoogleDrive\Teaching - Ryerson\CCPS 506\_S_2018\I
```

- **acc** is the running value
- By default, initialized to first element in list

We can add an optional 3rd argument to initialize **acc**:

```
iex> Enum.reduce([1, 2, 3], 10, fn(x, acc) -> x+acc end)  
16
```

```
iex> Enum.reduce([1, 2, 3], fn(x, acc) -> x+acc end)  
6
```

10 + 1 + 2 + 3

VS

1 + 2 + 3

Stream

Streams

Like Enum, but Streams are *lazy*!

- Enum functions are strict/eager.
- The result of an Enum is the list that results from applying it:

```
iex> list = [1, 2, 3, 4, 5]
```

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

```
iex> Enum.map(list, &(&1 + 1))
```

```
[2, 3, 4, 5, 6]
```

Streams

Like Enum, but Streams are lazy!

- Stream and Enum share many functions.
- What is the result of evaluating Stream.map?

```
iex> list = [1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
iex> Stream.map(list, &(&1 + 1))
#Stream<[
  enum: [1, 2, 3, 4, 5],
  funs: [#Function<48.103564624/1 in Stream.map/2>]
]>
```

```
iex> list = [1, 2, 3, 4, 5]
      [1, 2, 3, 4, 5]
iex> Stream.map(list, &(&1 + 1))
#Stream<[
  enum: [1, 2, 3, 4, 5],
  funs: [#Function<48.103564624/1 in Stream.map/2>]
]>
```

- That's not a list! Stream is its own type.
- Think of a stream as a *recipe* for producing the transformed list.
- Here, our stream is a recipe for adding 1 to every element.
- We haven't actually done the cooking!
- Why is this useful?

Streams

Consider the following script:

```
list = [1, 2, 3, 4, 5]
r1 = Enum.map(list, &(&1 + 1)) |>
      Enum.map(&(&1 * 3)) |>
      Enum.map(&(&1 / 2))
```

An aside: Pipe is useful here!

- Output list from Enum piped into next call as 1st arg.
- Thus, subsequent Enum calls only have 1 arg.

How many new lists are created when we evaluate this?

One for each Enum call! ***Very inefficient.***

Streams

```
list = [1, 2, 3, 4, 5]
r1 = Stream.map(list, &(&1 + 1)) |>
      Stream.map(&(&1 * 3)) |>
      Stream.map(&(&1 / 2))
```

- **r1** is a recipe for a new list
- At this point, no new list(s) have been created!

```
list = [1, 2, 3, 4, 5]
r1 = Stream.map(list, &(&1 + 1)) |>
      Stream.map(&(&1 * 3)) |>
      Enum.map(&(&1 / 2))
```

- If we finish with an Enum call, the stream is applied.
- Only one new list created

```
C:\_cps506\elixir\StreamTest.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
StreamTest.exs
1 list = [1, 2, 3, 4, 5]
2 r1 = Stream.map(list, &(&1 + 1)) |>
3     Stream.map(&(&1 * 3)) |>
4     Enum.map(&(&1 / 2))
5 IO.inspect r1
6
7
length: 135 lines: 7 Ln: 7 Col: 1 Sel
```

```
Windows PowerShell
PS C:\_cps506\elixir> elixir StreamTest.exs
[3.0, 4.5, 6.0, 7.5, 9.0]
PS C:\_cps506\elixir>
```

Apply the Stream?

Can also use `Enum.to_list`

```
list = [1, 2, 3, 4, 5]
r1 = Stream.map(list, &(&1 + 1)) |>
      Stream.map(&(&1 * 3)) |>
      Stream.map(&(&1 / 2))
Enum.to_list(r1)
```



```
C:\_cps506\elixir\StreamTest.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
StreamTest.exs
1 list = [1, 2, 3, 4, 5]
2 r1 = Stream.map(list, &(&1 + 1)) |>
3     Stream.map(&(&1 * 3)) |>
4     Stream.map(&(&1 / 2))
5 IO.inspect (Enum.to_list r1)
6
7
length: 152 lines: 7 Ln: 7 Col: 1 Sel: 0|0 Windows (CR LF)
```

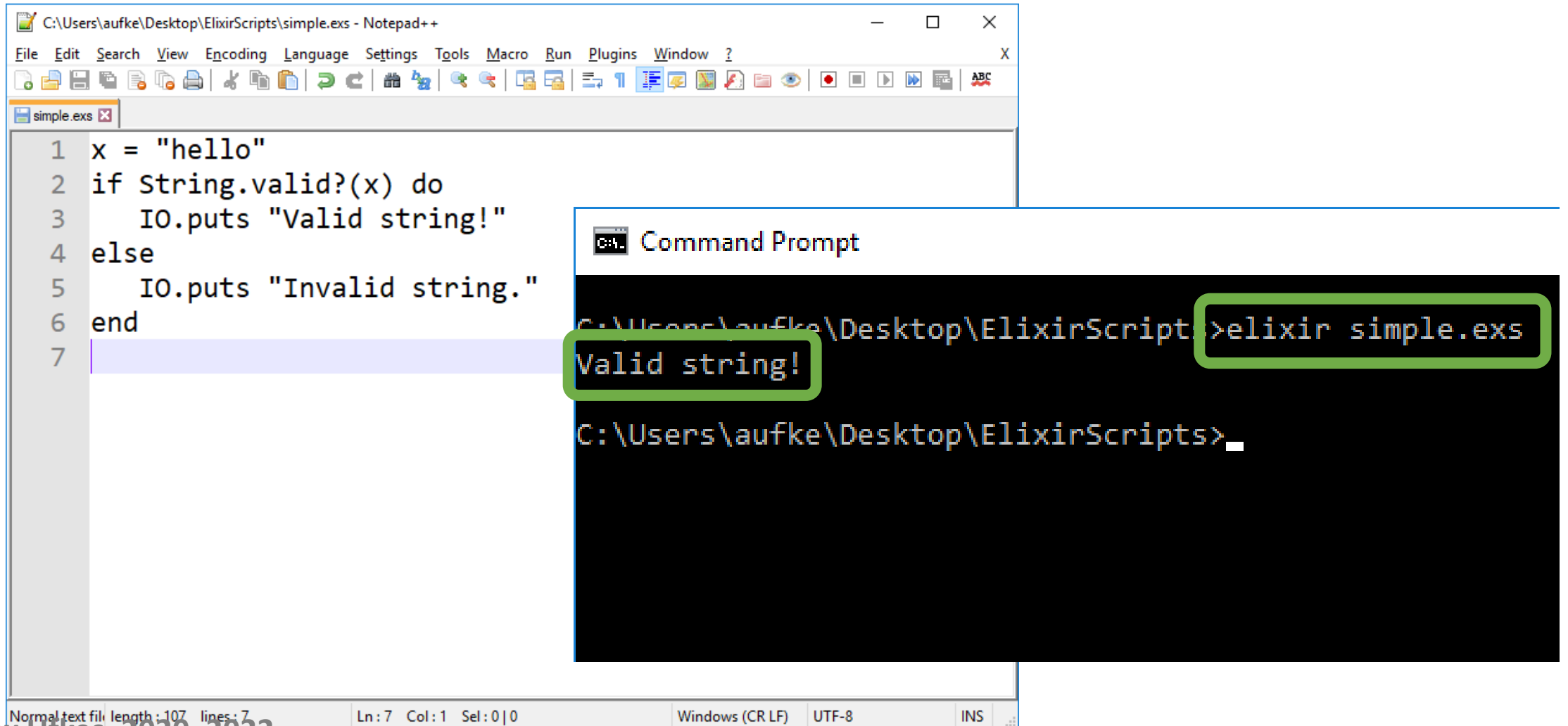
```
Windows PowerShell
PS C:\_cps506\elixir> elixir StreamTest.exs
[3.0, 4.5, 6.0, 7.5, 9.0]
PS C:\_cps506\elixir> _
```

Lots more: <https://hexdocs.pm/elixir/Stream.html>

Control “Structures”

Implemented using function calls and pattern matching

Selection: `if/else`



The image shows a Notepad++ window with the following Elixir code:

```
1 x = "hello"  
2 if String.valid?(x) do  
3   IO.puts "Valid string!"  
4 else  
5   IO.puts "Invalid string."  
6 end  
7
```

Below the code, a Command Prompt window is open, showing the execution of the code:

```
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs  
Valid string!  
C:\Users\aufke\Desktop\ElixirScripts>
```

Green boxes highlight the command `elixir simple.exs` and the output `Valid string!`.

At the bottom of the Notepad++ window, the status bar shows: Normal text file, length: 107, lines: 7, Ln: 7 Col: 1 Sel: 0|0, Windows (CR LF), UTF-8, INS.

C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?

simple.exs

```
1 x = 3.14159
2 if String.valid?(x) do
3   IO.puts "Valid string!"
4 else
5   IO.puts "Invalid string."
6 end
7
```

Normal text file | length : 107 | lines : 7 | Ln : 7 | Col : 1

Command Prompt

```
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
Valid string!
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
Invalid string.
C:\Users\aufke\Desktop\ElixirScripts>
```

```
C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
simple.exs
1 if "48" do
2   IO.puts true
3 else
4   IO.puts false
5 end
6
```

length: 59 lines: 6 Ln: 6 Col: 1 Sel: 0|0

As long as this expressions evaluates to true or false

"48"

Why?

```
Command Prompt
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
true
C:\Users\aufke\Desktop\ElixirScripts>
```

Boolean Expressions

Boolean:

true, false



`&&, ||, !`

With these operators:

- *non-false and non-nil are **true**.*
- *nil and false are **false**.*
- *0 is considered true!*

```
iex> "gh" && false  
false
```

```
iex> "gh" || false  
"gh"
```

Except...

- *The result isn't true or false*
- *It's the **value that decided the result** of true or false*

*What we actually get is the **value that determined the truthiness of the expression***

“No loop/if-else/case constructs”

In Elixir, we have several control structures that are implemented as macros. They are not actually constructs of the programming language.

Their implementation exists in the Elixir Kernel module.

They allow us to write if/else-style constructs in a familiar way. However, these are function calls behind the scenes.

```
Erlang
File Edit Options View Help
Erlang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10]
Interactive Elixir (1.6.5) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> h if/2 ←
* defmacro if(condition, clauses)

Provides an `if/2` macro.

This macro expects the first argument to be a condition and the second
argument to be a keyword list?.

## one-liner examples

    if(foo, do: bar)

In the example above, `bar` will be returned if `foo` evaluates to
`true` (i.e., it is neither `false` nor `nil`). Otherwise, `nil` will be
returned.

An `else` option can be given to specify the opposite:

    if(foo, do: bar, else: baz)
```


do/end VS Keyword List


```
if 1 < 2 do  
  "Hello"  
end
```

Is the same as:

```
if 1 < 2, do: "Hello"
```

Is the same as:

```
if(1 < 2, do: "Hello")
```



This form is a syntactic convenience allowed by Elixir to make the language more accessible.

do/end VS Keyword List

```
if 1 < 2 do
  "Hello"
else
  "World"
end
```

Is the same as:

```
if 1 < 2, do: "Hello", else: "World"
```

Is the same as:

```
if(1 < 2, do: "Hello", else: "World")
```

```
iex> if 1 < 2, do: "Hello", else: "World"
"Hello"
```

```
iex> if(1 < 2, do: "Hello", else: "World")
"Hello"
```

do/end VS Keyword List

```
if 1 < 2 do
  "Hello"
else
  "World"
end
```

Is the same as:

```
if(1 < 2, do: "Hello", else: "World")
```

Is the same as:

```
if(1<2, [{:do, "Hello"}, {:else, "World"}])
```

```
if(1<2, [{:do, "Hello"}, {:else, "World"}])
```



do/end VS Keyword List

```
if 1 < 2 do
  "Hello"
else
  "World"
end
```

Is the same as:

```
if 1 < 2, do: "Hello", else: "World"
```

Is the same as:

```
if(1<2, [{:do, "Hello"}, {:else, "World"}])
```

```
iex> if 1 < 2, do: "Hello", else: "World"
"Hello"
```

```
iex> if(1 < 2, [{:do, "Hello"}, {:else, "World"}])
"Hello"
```

Can be any expression!

```
iex> if(1 < 2, [{:do, IO.puts "Hello"}, {:else, "World"}])  
Hello  
:ok  
iex>
```

unless

```
unless is_integer("hello") do
  "Not an Int"
end
```

```
iex> unless(is_integer("hello"), do: "Not an Int")
"Not an Int"
```

```
iex> unless(is_integer("hello"), [{:do, "Not an Int"}])
"Not an Int"
```

unless: With an else

```
unless is_integer(0b10101) do
  "Not an Int"
else
  "An Int"
end
```

"An Int"

case

We can never get here!

if and **unless** can't handle pattern matching gracefully:

- Matching returns the right-hand side...
- *UNLESS* no match is found, then it yields a MatchError.
- If a match is found we'd be OK – [1, 2, 3] is true (non-nil, non-false)

```
C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugin
simple.exs ElixirScriptList.txt
1 unless [1, x, 2] = [1, 2, 3] do
2   IO.puts x
3 else
4   IO.puts x
5 end
6
7
length: 78 lines: 7 Ln: 7 Col: 1 Sel: 0|0
```

```
Command Prompt
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
warning: no clause will ever match
simple.exs:1

** (MatchError) no match of right hand side value: [1, 2, 3]
simple.exs:1: (file)
(elixir) lib/code.ex:677: Code.require_file/2

C:\Users\aufke\Desktop\ElixirScripts>
```

case

Match this tuple successively with each case:

```
tup = { :ok, "Hello World" }  
case tup do  
  → { :ok, result } -> result  
  → { :error } -> "Uh oh!"  
  → _ -> "Catch all"  
end
```

Pattern match!

```
{:ok, result} = {:ok, "Hello World"}
```

```
{:error} = {:ok, "Hello World"}
```

```
_ = {:ok, "Hello World"}
```

Without a catch-all, we'd get an error if no match was found.

```
*C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
simple.exs x ElixirScriptList.txt x
1 case {:ok, "Hello World"} do
2   {:ok, result} -> IO.puts result
3   {:error} -> IO.puts "Uh oh!"
4   _ -> IO.puts "Catch all"
5 end
6
7
length: 137 lines: Ln: 7 Col: 1 Sel: 0|0
```

```
Command Prompt
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
Hello World
C:\Users\aufke\Desktop\ElixirScripts>
```

```
C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
simple.exs ElixirScriptList.txt
1 case {:error} do
2   {:ok, result} -> IO.puts result
3   {:error} -> IO.puts "Uh oh!"
4   _ -> IO.puts "Catch all"
5 end
6
7
length: 125 lines: Ln: 7 Col: 1 Sel: 0|0
```

```
Command Prompt
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
Uh oh!
C:\Users\aufke\Desktop\ElixirScripts>
```

```
C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
simple.exs ElixirScriptList.txt
1 case {:error, "Blah"} do
2   {:ok, result} -> IO.puts result
3   {:error} -> IO.puts "Uh oh!"
4   _ -> IO.puts "Catch all"
5 end
6
7
```

length: 133 lines: Ln: 7 Col: 1 Sel: 0|0

```
Command Prompt
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
Catch all
C:\Users\aufke\Desktop\ElixirScripts>
```

```
C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
simple.exs x ElixirScriptList.txt x
1 case {:error, "Blah"} do
2   {:ok, result} -> IO.puts result
3   {:error} -> IO.puts "Uh oh!"
4   #_ -> IO.puts "Catch all"
5 end
6
```

Comment out
catch all case

```
Command Prompt
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
warning: no clause will ever match
simple.exs:1

** (CaseClauseError) no case clause matching: {:error, "Blah"}

simple.exs:1: (file)
(elixir) lib/code.ex:677: Code.require_file/2

C:\Users\aufke\Desktop\ElixirScripts>
```

case: Matching Variables

```
pi = 3.14
```

```
IO.puts pi ← What prints?
```

```
case "apple pie" do
```

```
→ pi -> IO.puts "Tasty " <> pi
```

```
_ -> IO.puts "#{pi} is not tasty"
```

```
end
```

Attempts to match: pi = "apple pie"

- What's the problem here?

```
C:\_cps506\elixir\test.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
test.exs
1 pi = 3.14
2 IO.puts pi
3
4 case "apple pie" do
5   pi -> IO.puts "Tasty " <> pi
6   _ -> IO.puts "#{pi} is not tasty"
7 end
8
9
length: 131 |ir Ln: 9 Col: 1 Sel: 0|0 Windows (CR LF) UTF-8 INS
```

```
Windows PowerShell
PS C:\_cps506\elixir> elixir test.exs
3.14
Tasty apple pie
PS C:\_cps506\elixir> _
```

```
C:\_cps506\elixir\test.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
test.exs
1 pi = 3.14
2 IO.puts pi
3
4 case "apple pie" do
5   ^pi -> IO.puts "Tasty " <> pi
6   _ -> IO.puts "#{pi} is not tasty"
7 end
8
9
length: 132 |ir Ln: 9 Col: 1 Sel: 0|0 Windows (CR LF) UTF-8 INS
```

Pin pi using ^

```
Windows PowerShell
PS C:\_cps506\elixir> elixir test.exs
3.14
3.14 is not tasty
PS C:\_cps506\elixir>
```


Guard Clauses

```
C:\_cps506\elixir\test.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
test.exs
1 tup = {1, 2, 3}
2 case tup do
3     {1, x, 3} when x < 0 ->
4         IO.puts "Match #{x}"
5     _ ->
6         IO.puts "No match!"
7 end
8
```

```
Windows PowerShell
PS C:\_cps506\elixir> elixir test.exs
No match!
PS C:\_cps506\elixir>
```

Place a condition on the match:

- In this case, match is only successful if $x < 0$

Guard reference: <https://hexdocs.pm/elixir/master/guards.html>

Guard Clauses

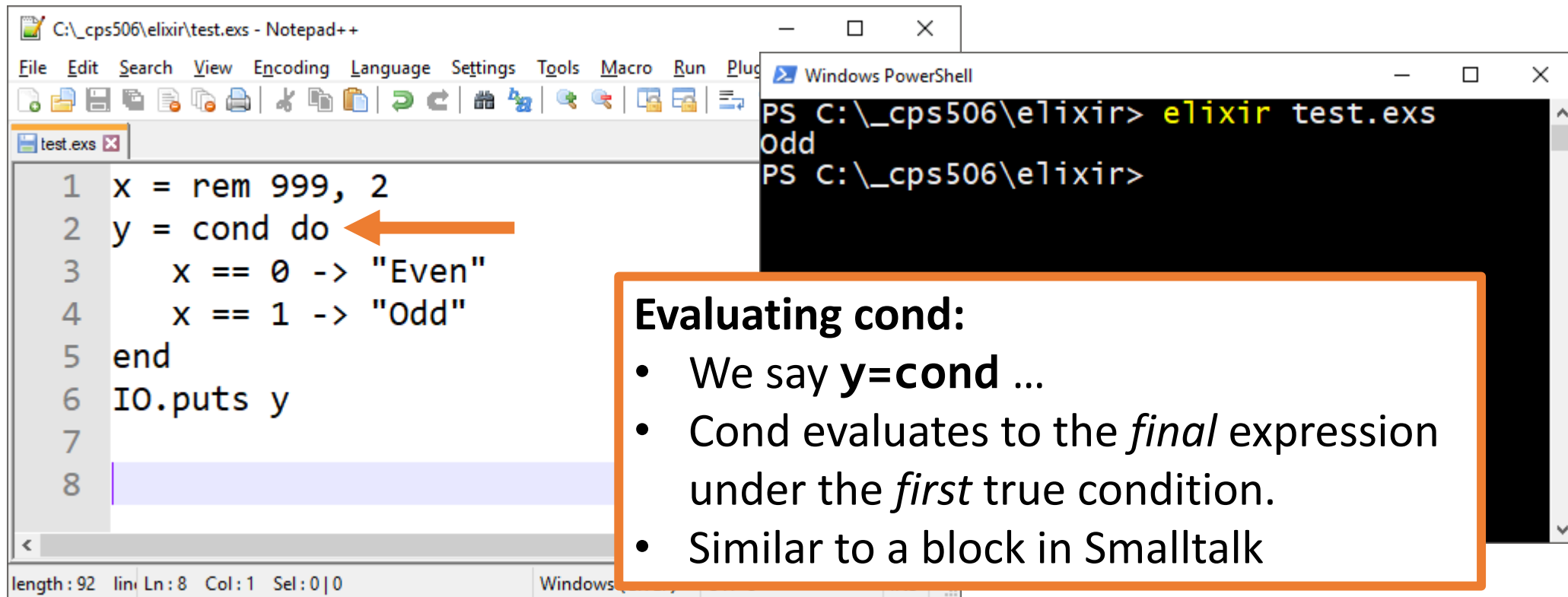
```
C:\cps506\elixir\test.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
test.exs
1 tup = {1, -2, 3}
2 case tup do
3   {1, x, 3} when x < 0 ->
4     IO.puts "Match #{x}"
5   _ ->
6     IO.puts "No match!"
7 end
8
```

```
Windows PowerShell
PS C:\cps506\elixir> elixir test.exs
Match -2
PS C:\cps506\elixir>
```

Guard reference: <https://hexdocs.pm/elixir/master/guards.html>

cond

case is for pattern matching, **cond** is for conditions:



The image shows a Notepad++ window with the following Elixir code in `test.exs`:

```
1 x = rem 999, 2
2 y = cond do ←
3   x == 0 -> "Even"
4   x == 1 -> "Odd"
5 end
6 IO.puts y
7
8
```

An orange arrow points to the `do` keyword in line 2. To the right, a Windows PowerShell window shows the command `elixir test.exs` being executed, with the output `Odd`.

Evaluating cond:

- We say **y=cond** ...
- Cond evaluates to the *final* expression under the *first* true condition.
- Similar to a block in Smalltalk

```
C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ? X
simple.exs ElixirScriptList.txt
1 cond do
2   2 + 2 == 5 ->
3     IO.puts "This will not be true"
4   2 * 2 == 3 ->
5     IO.puts "Nor this"
6   1 + 1 == 7 ->
7     IO.puts "Oops"
8 end
9
```

length: 162 |ir Ln: 9 Col: 1 Sel: 0|0

```
Command Prompt
C:\Users\aufke\Desktop\ElixirScripts>elixir simple.exs
** (CondClauseError) no cond clause evaluated to a true value
(file)
(elixir) lib/code.ex:677: Code.require_file/2

C:\Users\aufke\Desktop\ElixirScripts>_
```

cond: Always have a catch-all

```
C:\Users\aufke\Desktop\ElixirScripts\simple.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window
simple.exs ElixirScriptList.txt
1 cond do
2     2 + 2 == 5 ->
3         IO.puts "This will not be true"
4     2 * 2 == 3 ->
5         IO.puts "Nor this"
6     1 + 1 == 7 ->
7         IO.puts "Oons"
8     true ->
9         IO.puts "Catch-all"
10 end
11
```

```
Command Prompt
C:\Users\aufke\Desktop\ElixirScripts>elixir simple
Catch-all
C:\Users\aufke\Desktop\ElixirScripts>
```

List Comprehensions

for :

- Generating
- Filtering
- Operating

Produces a list when it's done!

Not the same as an imperative-style for loop!

Not for general purpose iteration.

List Comprehensions

Very much like comprehensions in Python:

Three parts:

- Generator
- Filter
- Collector

Comprehensions can be used to do things that we could otherwise do with Enum or recursive functions

List Comprehensions

```
iex> Enum.map([1, 2, 3, 4], &(&1*&1))  
[1, 4, 9, 16]
```

```
iex> for n <- [1, 2, 3, 4] do: n*n  
[1, 4, 9, 16]
```

Generator: Any enumerable

- In this case, a plain old list

List Comprehensions

```
iex> Enum.map([1, 2, 3, 4], &(&1*&1))  
[1, 4, 9, 16]
```

```
iex> for n <- [1, 2, 3, 4], do: n*n  
[1, 4, 9, 16]
```

```
iex> for n <- 1..4, do: n*n  
[1, 4, 9, 16]
```

- Used to produce list [1, 2, 3, 4]
- Can generate large lists this way
- Note: **1..4** is **NOT** itself a list!
- It is a *Range*

```
Erlang
File Edit Options View Help
ErLang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [ds:8:8:10] [async-threads:10]
Interactive Elixir (1.6.5) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> 1..4
1..4
iex(2)> IO.inspect 1..4
1..4
1..4
iex(3)> Enum.to_list(1..4)
[1, 2, 3, 4]
iex(4)> hd 1..4
** (ArgumentError) argument error
:erlang.hd(1..4)
iex(4)> tl 1..4
** (ArgumentError) argument error
:erlang.tl(1..4)
iex(4)> Enum.map(1..4, &(&1+1))
[2, 3, 4, 5]
iex(5)> █
```

List Comprehensions

```
iex> for n <- 1..4, do: n*n  
[1, 4, 9, 16]
```

- List comprehensions produce lists
- Generators like the above are lazy (Range)
- Operate on elements one at a time, discarding previous.
- That is, at no point do we produce the complete list [1, 2, 3, 4] in memory.


<https://hexdocs.pm/elixir/Range.html>

List Comprehensions: Pattern Matching

```
iex> vals = [good: 1, good: 2, bad: 3, good: 4]
```



Keyword list!



```
iex> vals = [{:good, 1}, {:good, 2}, {:bad, 3}, {:good, 4}]  
[good: 1, good: 2, bad: 3, good: 4]
```

List Comprehensions: Pattern Matching

```
iex> vals = [good: 1, good: 2, bad: 3, good: 4]  
          [good: 1, good: 2, bad: 3, good: 4]
```

```
iex> for {:good, n} <- vals, do: n*n  
[1, 4, 16]
```

- Pattern matching is powerful
- We can also filter in a Boolean fashion

List Comprehensions: Filtering

```
iex> fun = &(rem(&1, 3) == 0)
#Function<6.99386804/1 in :erl_eval.expr/5>
iex> for n <- 1..20, fun.(n), do: n
[3, 6, 9, 12, 15, 18]
```

Filter is optional

- Include it after generator if desired
- Only elements that evaluate to true when filtered will make it to the **do:** block

List Comprehensions: Filtering & Matching

```
iex> list = [a: 1, b: "2", a: 3.0, a: "4.0", b: {5}, a: ["6.0"]]  
[a: 1, b: "2", a: 3.0, a: "4.0", b: {5}, a: ["6.0"]]
```

```
iex> for {:a, n} <- list, is_number(n), do: n  
[1, 3.0]
```

Nothing semantically new here

- Anything we can do with comprehensions we can do with Enum or our own functions.
- It might require more syntax, but we can do it.
- Comprehensions can be used to create concise code

List Comprehensions: In 2D?

```
iex> for i <- [ :a, :b, :c ], j <- [1, 2], do: {i, j}
[a: 1, a: 2, b: 1, b: 2, c: 1, c: 2]
```

We get a keyword list containing combinations of all elements from both generators

Elixir Processes (In Brief):

Elixir is built on a process model. Recall:

- *Elixir code runs inside lightweight threads of execution.*
 - *Isolated, exchange information via message passing.*
- *Not uncommon to have hundreds of thousands of processes running concurrently in same VM.*
 - *Note: These are NOT operating system processes!*
 - *Extremely lightweight in terms of CPU and memory*
 - *A process need not be an expensive resource*

Elixir Processes

```
Erlang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [ds:8:8:10] [as^
Interactive Elixir (1.6.5) - press ctrl+C to exit (type h())
iex(1)> self()
#PID<0.83.0>
iex(2)> Process.alive?(self())
true
iex(3)>
```

Playing with processes:

- **self()** Returns PID of current process.
 - In this case, it's the PID of our interactive shell session
- **Process.alive?()** tests if a process is currently active.
- We can spawn functions as processes!

Elixir Processes

```
Erlang/OTP 20 [erts-9.2] [64-bit] [smp:8:8] [ds:8:8:10] [as^
Interactive Elixir (1.6.5) - press ctrl+C to exit (type h())
iex(1)> self()
#PID<0.83.0>
iex(2)> Process.alive?(self())
true
iex(3)> pid = spawn fn -> 1 + 2 end
#PID<0.87.0>
iex(4)> Process.alive?(pid)
false
iex(5)>
```



- **spawn** takes a function as an argument and returns its PID once spawned.
- Function executes when spawned

Process is not active, the function is not currently executing

Elixir Processes: Send & Receive

```
iex> send(self(), {:Hello, "World"})  
{:Hello, "World"}
```

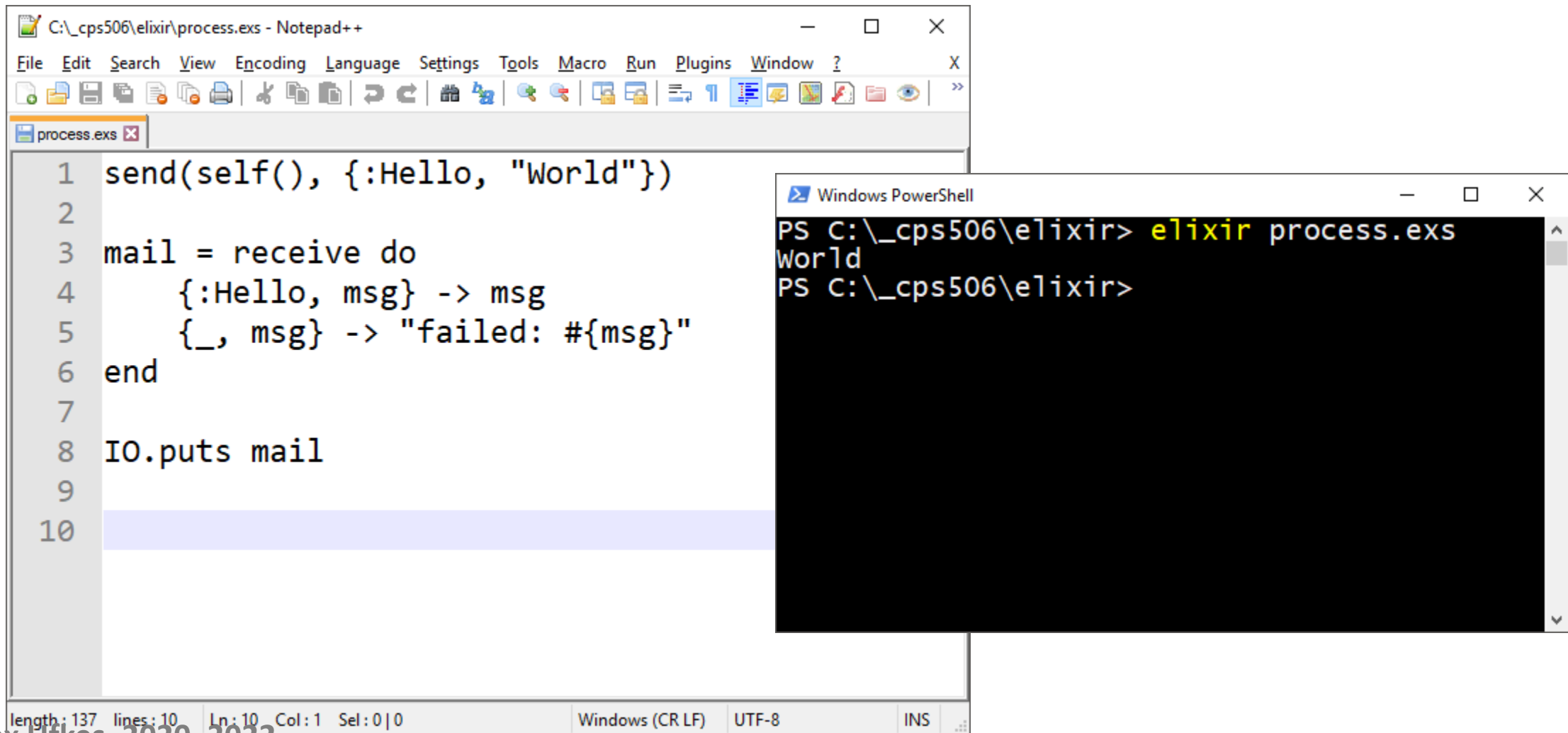
- **send/2** can be used to send a message to a process (by PID)
- This message goes into a mailbox and can be received using the **receive/1** function (or using its macro syntax form, as we will)
- When invoking receive, it will go through the messages in the mailbox and attempt to match the messages with the provided patterns

Elixir Processes: Send & Receive

```
iex> send(self(), {:Hello, "World"})
      {:Hello, "World"}
iex> receive do
...>   {:Hello, msg} -> msg
...>   {:World, msg} -> "won't match"
...> end
      "World"
iex>
```

- Once the message is received, it is consumed!
- We can't receive the same message twice.
- Subsequent receive calls will be *blocking*

Elixir Processes: Send & Receive

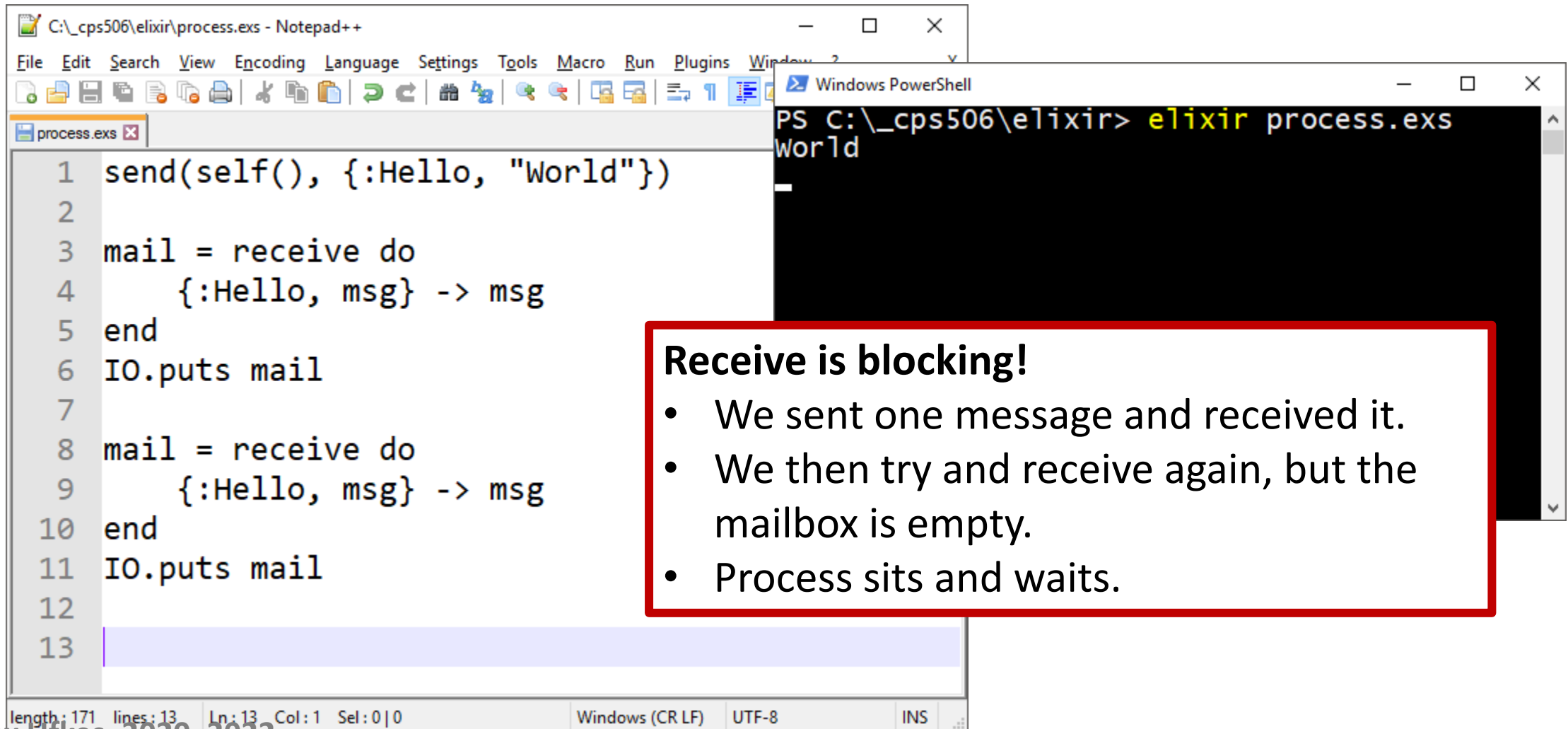


The image shows two overlapping windows. The top window is Notepad++ with a file named 'process.exs'. The code in the file is:

```
1 send(self(), {:Hello, "World"})
2
3 mail = receive do
4   {:Hello, msg} -> msg
5   _, msg -> "failed: #{msg}"
6 end
7
8 IO.puts mail
9
10
```

The bottom window is Windows PowerShell. It shows the command `elixir process.exs` being executed, which outputs `World`. The prompt then returns to `PS C:_cps506\elixir>`.

Elixir Processes: Send & Receive



The image shows a Notepad++ window with the following Elixir code in a file named process.exs:

```
1 send(self(), {:Hello, "World"})
2
3 mail = receive do
4   {:Hello, msg} -> msg
5 end
6 IO.puts mail
7
8 mail = receive do
9   {:Hello, msg} -> msg
10 end
11 IO.puts mail
12
13
```

Overlaid on the right is a Windows PowerShell window with the following command and output:

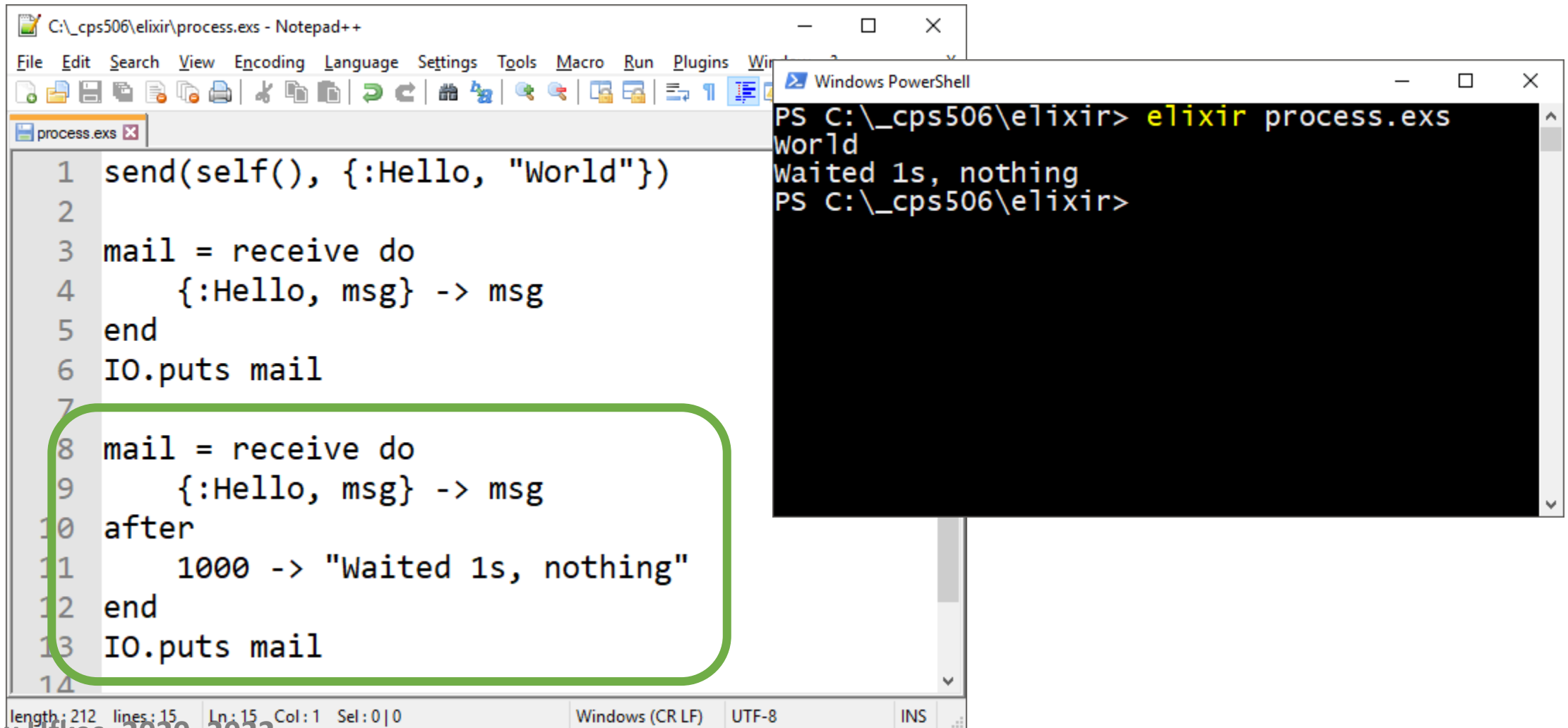
```
PS C:\_cps506\elixir> elixir process.exs
World
```

A red-bordered box highlights the following text:

Receive is blocking!

- We sent one message and received it.
- We then try and receive again, but the mailbox is empty.
- Process sits and waits.

Elixir Processes: Send & Receive



```
C:\_cps506\elixir\process.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Win
process.exs
1 send(self(), {:Hello, "World"})
2
3 mail = receive do
4   {:Hello, msg} -> msg
5 end
6 IO.puts mail
7
8 mail = receive do
9   {:Hello, msg} -> msg
10 after
11   1000 -> "Waited 1s, nothing"
12 end
13 IO.puts mail
14

length: 212 lines: 15 Ln: 15 Col: 1 Sel: 0|0 Windows (CR LF) UTF-8 INS

Windows PowerShell
PS C:\_cps506\elixir> elixir process.exs
World
Waited 1s, nothing
PS C:\_cps506\elixir>
```


Elixir Processes: Send & Receive

```
C:\cps506\elixir\process.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
process.exs x
1 defmodule Mail do
2   def get() do
3     receive do
4       :Hello -> IO.puts "Hello"
5       :World -> IO.puts "World"
6       _ -> IO.puts "Huh?"
7     end
8   end
9 end
10
11 pid = spawn(&Mail.get/0)
12 send(pid, :world)
13 pid = spawn(&Mail.get/0)
14 send(pid, :Hello)
15 pid = spawn(&Mail.get/0)
16 send(pid, 97)
length: 335 lines: 20 Ln: 19 Col: 1 Sel: 0|0 Window
```

- Spawning a function as a process executes that function.
- A blocking receive can be used to wait for messages.
- Once the function receives a message, it will pattern match.
- Receive only blocks once! We must spawn the function three times.

```
Windows PowerShell
PS C:\cps506\elixir> elixir process.exs
World
Huh?
Hello
World
Hello
Huh?
PS C:\cps506\elixir>
```

- Different order?
- Execution is interleaved.
- Up to scheduler.

Elixir Processes: Send & Receive

```
*C:\_cps506\elixir\process.exs - Notepad++
File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
process.exs
1 defmodule Mail do
2   def get() do
3     receive do
4       :Hello -> IO.puts "Hello"
5       :World -> IO.puts "World"
6       _ -> IO.puts "Huh?"
7     end
8   end
9 end
10
11 pid1 = spawn(&Mail.get/0)
12 pid2 = spawn(&Mail.get/0)
13 pid3 = spawn(&Mail.get/0)
14 send(pid1, :World)
15 send(pid2, :Hello)
16 send(pid3, 97)
length: 341 lines: 20 Ln: 18 Col: 1 Sel: 0|0 Window
```

- Spawn all three, send each a message.
- Which child process gets chosen to execute is up to the scheduler.

```
Windows PowerShell
PS C:\_cps506\elixir> elixir process.exs
Hello
World
Huh?
PS C:\_cps506\elixir>
```

Elixir Processes

- This has been a taste. There's lots more.
- Elixir is famous for powerful concurrent processing.
- Processes can be used to emulate the object message passing model in languages like Smalltalk.
- If you understand a bit about concurrency from 209 or 590, check it out.

<https://elixir-lang.org/getting-started/processes.html>



elixir

Fin.

Functional Programming & Elixir

We saw:

Functions: First-class entities

- Create and pass anonymous functions as arguments
- Return anonymous functions as values

Immutable data: Variables are bound and matched using =

- Collections are not modified.
- Enum.map returns a *new* collection

Recursion: Repetition accomplished with tail-recursion.

- Enum functions work this way behind the scenes

Functional Programming & Elixir

Control flow is not built into the language as syntax constructs

- Selection and branching are implemented as functions
- Operate using keyword lists and pattern matching

```
if 1 < 2 do
  "Hello"
else
  "World"
end
```

Is the same as:

```
if(1 < 2, do: "Hello", else: "World")
```

Is the same as:

```
if(1<2, [{:do, "Hello"}, {:else, "World"}])
```

Elixir Syntax

- Dynamically typed
 - Type inferred at run-time
 - Need not explicitly specify type upon declaration
- Provides syntax conveniences to make it more intuitive to programmers accustomed to imperative languages
- Interactive shell provides help/search functionality

<https://media.pragprog.com/titles/elixir/ElixirCheat.pdf>

Elixir Syntax

Reserved words

- true, false, nil
 - Used as atoms
- when, and, or, not, in
 - Used as operators
- fn
 - Used for anonymous function definitions
- do, end, catch, rescue, after, else
 - Used in do/end blocks

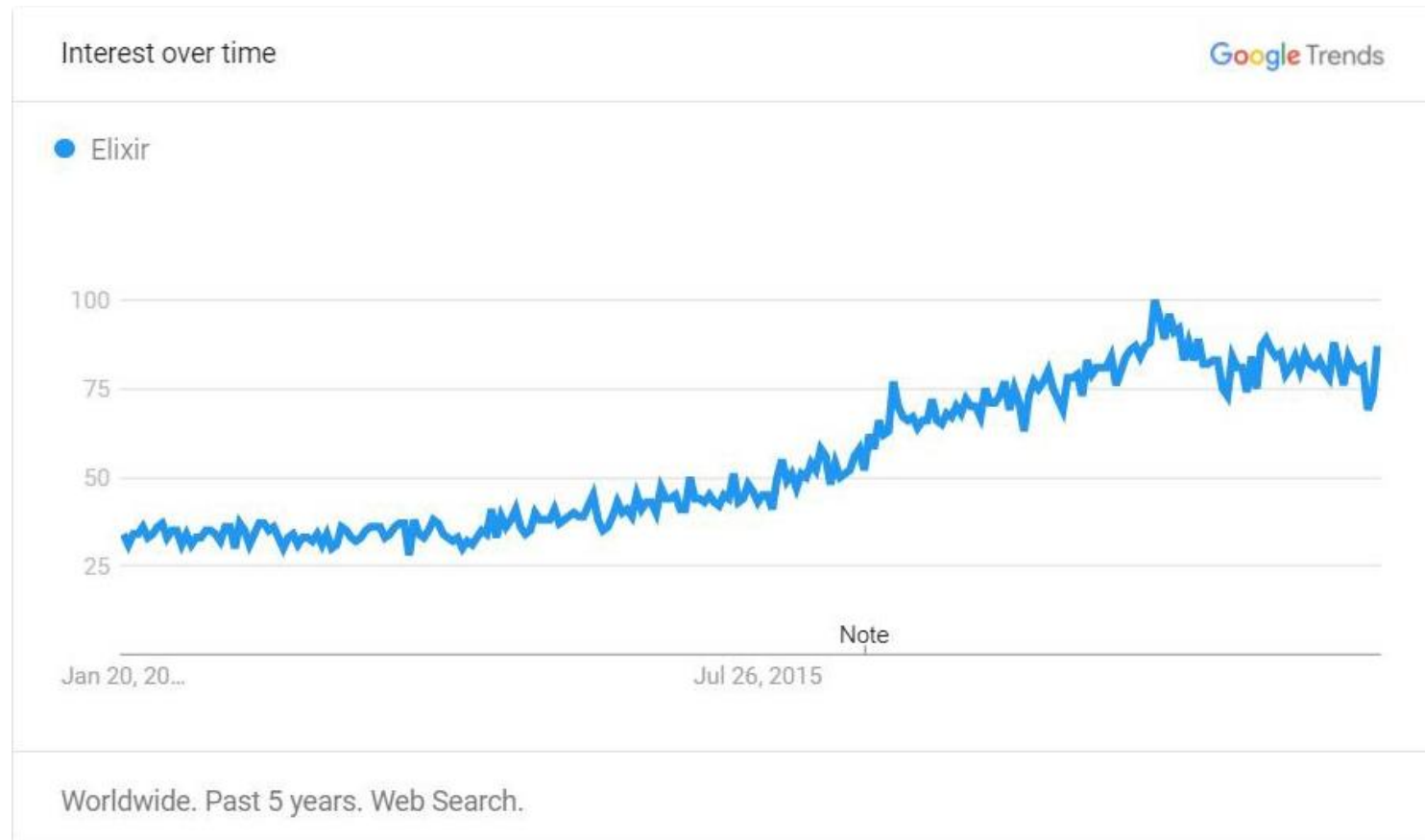
<https://github.com/elixir-lang/elixir/blob/master/lib/elixir/pages/Syntax%20Reference.md>

Further Reading

<https://elixir-lang.org/getting-started/introduction.html>

<https://elixirschool.com/en/lessons/basics/basics/>

Elixir Popularity



Elixir Popularity

<https://techbeacon.com/5-emerging-programming-languages-bright-future>

This list also includes Rust!

In Summary

More Advanced Elixir:

- Control flow, keyword lists
- Enum VS Stream
- List comprehensions
- Elixir processes
- Elixir sendoff

