

Functions in elixir

- In Elixir there are modules which hold functions and functions which transform data.
- Modules can be created by calling

```
defmodule Test do
```

```
end
```

- named functions are created inside your module with
def func_name(params) do end
- lets create a test function in IEx!

```
iex> defmodule Create do
...>def add(a,b) do
...>a+b
...>end
...>end
{:module, Create,
 <<70, 79, 82, 49, 0, 0, 4, 212, 66, 69, 65, 77, 65, 116, 85, 56, 0, 0, 0, 126,
    0, 0, 0, 14, 13, 69, 108, 105, 120, 105, 114, 46, 67, 114, 101, 97, 116, 101,
    8, 95, 95, 105, 110, 102, 111, 95, 95, ...>>, {:add, 2}}}
```

```
iex> Create.add(1,3)
```

```
4
```

- Elixir functions are defined by two things name and arity where the name is the name of the function and arity is the number of arguments so our add function has an arity of 2 so we would call it add/2
- In elixir there are anonymous functions which are functions without names you can create one by typing
fn _params -> _definition end
- Anonymous functions are often used when other functions take a function as a parameter

- with anonymous functions we can create our add function anonymously!

```
iex> add = fn a, b -> a+b end
```

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

```
4
```

- In elixir `&` is the capture operator which allows us to write anonymous functions

```
iex> add1 = &(&1 + &2)
```

```
iex> add1.(13,12)
```

```
25
```

using this syntax we created an anonymous function that does the same operation!

Recursive functions

- Recursive functions are functions that call themselves to perform an operation until a base case is reached.
- for example in our previous section we created a list fuction which uses tail recursion to sum up the numbers in a list

```
iex(1)> defmodule Recursion do
... (1)> def dec(0), do: 0
... (1)> def dec(n) do
... (1)> IO.inspect(n)
... (1)> Recursion.dec(n-1)
... (1)> end
... (1)> end
{:module, Recursion,
 <<70, 79, 82, 49, 0, 0, 5, 88, 66, 69, 65, 77, 65, 116, 85, 56, 0, 0, 0, 147,
 0, 0, 0, 16, 16, 69, 108, 105, 120, 105, 114, 46, 82, 101, 99, 117, 114, 115,
 105, 111, 110, 8, 95, 95, 105, 110, 102, ...>>, {:dec, 1}}
iex(2)> Recursion.dec(5)
5
4
3
2
1
0
```

In math there is a famous sequence called the fibonacci sequence where a number in the sequence is defined by the addition of the previous two numbers in the sequence for example position 2 in the sequence is $0 + 1 = 1$ so position 3 is $1 + 1 = 2$ and so on.

There is a very elegant way to duplicate this sequence using recursion (Warning this function is computational expensive!!)

open up visual studio and create a new file called fib.ex

once in the file create a module called Fib

```
defmodule fib do
```

```
end
```

In recursion it is important to have a base case so our base case will be position 0 so our first function will be called $\text{fib}(0) = 0$ and $\text{fib}(1) = 1$ so define these two functions inside our module.

```
def fib(0), do: 0
def fib(1), do: 1
```

Now how will we create the nth sequence? Simple we will use recursion! lets add $\text{fib}(n-1)$ with $\text{fib}(n-2)$

```
def fib(n), do: fib(n-1) + fib(n-2)
```

save this file and run the command **iex fib.ex** in your terminal to compile the file

Once loaded we can call our fib function with Fib.fib()

```
iex> Fib.fib(3)
```

```
2
```

```
iex> Fib.fib(10)
```

```
55
```

Lets list all of our numbers in the fib sequence from 1 to 10

```
iex> IO.inspect(Enum.map(0..10, fn i-> Fib.fib(i) end))
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

There is a lot to unpack in this iex command

IO.inspect(Enum.map(0..10, fn i-> Fib.fib(i) end)) so lets start with the IO.inspect/1

- This function takes 1 argument and shows us our argument in the terminal
- The Enum.map/2 takes two arguments a range (1..10) and a function
- inside Enum.map our second argument is an anonymous function which will simply call our fib function for the specified i value
- so all this function is doing is calling fib 10 times with the values from 1..10