Syntax in functions.

1. Pattern matching → consists of specifying patterns to which some data should conform and then checking to see if it does.

Deconstructing the data according to these patterns.

↓

when defining functions, we can define separate fn. bodies for different patterns

```
sayMe :: (Integral a) => a -> String
sayMe 1 = "One!"
sayMe 2 = "Two!"
sayMe 3 = "Three!"
sayMe 4 = "Four!"
sayMe 5 = "Five!"
sayMe x = "Not between 1 and 5"
```

↓

factorial fn:

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n-1)
```

→ recursion

↓

important concept in Haskell.

↓

to remember: when making patterns, we should always include a catch-all pattern so that our program doesn't crash if we get some unexpected o/p.

↓

pattern matching can also be applied on tuples. for example take two tuples of two size and then adding their parts separately to return a tuple.

↓

addVectors :: (Num a) => (a, a) -> (a, a) -> (a, a)

addVectors (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

↓

writing a fst, and analogy for 3 elements vectors.

first :: (a, b, c) -> a

first (x, _, _) = x

second :: (a, b, c) -> b

second (_, y, _) = y

third :: (a, b, c) -> c

third (_, _, z) = z

↓

pattern matching on list comprehensions.

```
ghci> let xs = [(1,3), (4,3), (2,4), (5,3), (5,6), (3,1)]
ghci> [a+b | (a,b) <- xs]
[4,7,6,8,11,4]
```

↓

lists themselves can be used in pattern matching.
we can match with the empty list [] or any pattern that involves ':' and empty list.

↓

But since [1, 2, 3] is syntatic sugar for 1 : 2 : 3 : [],
we can use the former pattern as well.

↓

A pattern like x : xs binds the head of the list to x

and tail to xs.

↓

used a lot, especially for recursive functions.

but patterns that have ':' in them only match against lists of length 1 or more!.

↓

wanting to bind the first three elements to variables

`a : b : c : zs` → will only match for lists with more than 3 or equal to 3 elements.

→ own implementation of head function

```
head' :: [a] -> a
head' [] = error "Can't call head on an empty list, dummy!"
head' (x : _) = x
```

(need to surround with ( ))

(takes a string and generates a runtime error → causes the program to crash.)

→ function to report some of the first elements of a list

```
tell :: (Show a) => [a] -> String
tell [] = "The list is empty"
tell (x : []) = "List has only one element" ++ show x
tell (x : y : []) = "List has only two elements" ++ show x
```

```
tell (x:_) = "list has many elements" ++ show x ++ show y
```

could have been written as [x, y]

→ implementing our own length function

```
length' :: (Num b) => a -> b
length' [] = 0
length' (_,xs) = 1 + length' xs
```

} again using recursion.

→ implementing our own sum function

```
sum' :: (Num a) => [a] -> a
sum' [] = 0
sum' (x:xs) = x + sum' xs
```

→ patterns is a concept in Haskell wherein we can keep a reference to the whole argument which was being deconstructed due to the pattern matching.

↓

Done by placing @ in front of a pattern. for instance the pattern xs@(x:y:xs)

↓

```
capital :: String -> String
capital "" = "empty string"
capital all@(x:xs) = "The first letter of" ++ all ++ " is " ++ [x]
```

can't be used for pattern matching