Lesson 2: Playing with Lists / Pattern Matching / Definitions

What's It All About?

1. Lists are introduced via an interation that presents the syntax of lists, that sheds a bit of light on list types, that illustrates the use of "cons" in Haskell, and that features some of the most prominent list primitives in Haskell.
2. The pattern matching mechanism in Haskell is introduced conceptually, and illustrated computationally.
3. Some simple list processing definitions are presented.

Second Haskell Session – List Play

```
>>>  [True,False,True]
[True,False,True]
>>>  [1,2,3,4,5]
[1,2,3,4,5]

>>>  :t [True,False,True]
[True,False,True] :: [Bool]
>>>  :t [1,2,3,4,5]
[1,2,3,4,5] :: Num a => [a]

>>>  "cat"
"cat"
>>>  :t "cat"
"cat" :: [Char]
>>>  "cat" == ['c','a','t']
True

>>>  ["red","yellow","blue"]
["red","yellow","blue"]
>>>  :t ["red","yellow","blue"]
["red","yellow","blue"] :: [[Char]]

>>>  [("Desde El Alma","vals"),("Poema","tango")]
[("Desde El Alma","vals"),("Poema","tango")]
>>>  :t [("Desde El Alma","vals"),("Poema","tango")]
[("Desde El Alma","vals"),("Poema","tango")] :: [([Char], [Char])]

>>> "Racket" : ["Prolog", "Haskell", "Rust"]
["Racket","Prolog","Haskell","Rust"]
>>> :t ["Racket","Prolog","Haskell","Rust"]
["Racket","Prolog","Haskell","Rust"] :: [[Char]]

>>>  []
```

```
[]
>>>  :t []
[] :: [a]

>>>  1:[]
[1]
>>>  2:1:[]
[2,1]
>>>  3:2:1:[]
[3,2,1]
>>>  4:3:2:1:[]
[4,3,2,1]
>>>  :t 4:3:2:1:[]
4:3:2:1:[] :: Num a => [a]

>>>  "four":"three":"two":"one":[]
["four","three","two","one"]
>>>  :t "four":"three":"two":"one":[]
"four":"three":"two":"one":[] :: [[Char]]

>>>  length [2,3,5,7]
4
>>>  :t length [2,3,5,7]
length [2,3,5,7] :: Int

>>>  elem "milk" ["coffee","with","milk"]
True
>>>  elem "milk" ["coffee","with","cream"]
False

>>>  reverse [1,2,3,4,5,6,7,8,9]
[9,8,7,6,5,4,3,2,1]

>>>  head [10,20,30,40]
10
>>>  tail [10,20,30,40]
[20,30,40]
>>>  10:[20,30,40]
[10,20,30,40]

>>>  last [1,2,3,4,5]
5
>>>  init [1,2,3,4,5]
[1,2,3,4]

>>>  take 3 [1,2,3,4,5,6,7]
[1,2,3]
>>>  drop 3 [1,2,3,4,5,6,7]
[4,5,6,7]

>>> ["red", "yellow", "green", "blue"] !! 2
"green"
>>> ["red", "yellow", "green", "blue"] !! 1
"yellow"
```

```
>>> ["red", "yellow", "green", "blue"] !! 0
"red"

>>>  null []
True
>>>  null [[]]
False

>>>  sum [1,2,3,4,5]
15

>>>  maximum [1,4,7,2,5,8,1,2,1]
8

>>>  [1..5]
[1,2,3,4,5]
>>>  [1..10]
[1,2,3,4,5,6,7,8,9,10]
>>>  [10..1]
[]
>>>  [10,9..1]
[10,9,8,7,6,5,4,3,2,1]

>>>
```

Pattern Matching

"Pattern matching" is the process of:

1. Matching data to syntactic patters.
2. Deconstructing data, and establishing bindings, according to the matched patterns.

Many functions have simple, intuitive definitions when pattern matching is used in their articulation.

Second Haskell Functions – Pattern Matching

Code

```
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
--- second_functions.hs contains some functions to illustrate pattern matching

--------------------------------------------------------------------------------
```

```
-- Thing 1

lucky :: Int -> String
lucky 7 = "Lucky number seven!"
lucky x = "Sorry, pal, you are out of luck!"

lucky' :: Int -> String
lucky' x = if ( x == 7 ) then
              "Lucky number seven!"
            else
              "Sorry, pal, you are out of luck!"

--------------------------------------------------------------------------------
-- Thing 2

digit2name :: Int -> String
digit2name 1 = "one"
digit2name 2 = "two"
digit2name 3 = "three"
digit2name 4 = "four"
digit2name 5 = "five"
digit2name x = "Not in the proper range"

digit2name' :: Int -> String
digit2name' x =
   if ( x == 1 ) then "one" else
   if ( x == 2 ) then "two" else
   if ( x == 3 ) then "three" else
   if ( x == 4 ) then "four" else
   if ( x == 5 ) then "five" else
   "Not in the proper range"

--------------------------------------------------------------------------------
-- Thing 3

factorial' :: Integer -> Integer
factorial' 1 = 1
factorial' x = x * factorial' ( x - 1 )
```

Demo

```
bash-3.2$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help

Prelude> :set prompt ">>> "

>>> :load "second_functions.hs"
[1 of 1] Compiling Main             ( second_functions.hs, interpreted )
Ok, one module loaded.

>>> lucky 7
```

```
"Lucky number seven!"
>>> lucky 11
"Sorry, pal, you are out of luck!"
>>> lucky 13
"Sorry, pal, you are out of luck!"

>>> digit2name 1
"one"
>>> digit2name 5
"five"
>>> digit2name 7
"Not in the proper range"

>>> factorial' 1
1
>>> factorial' 10
3628800
>>> factorial' 30
265252859812191058636308480000000

>>> :quit
Leaving GHCi.
bash-3.2$
```

Third Haskell Functions – Simple List Processing

Demo

```
bash-3.2$ ghci
GHCi, version 8.6.3: http://www.haskell.org/ghc/  :? for help

Prelude> :set prompt ">>> "

>>> :load "third_functions.hs"
[1 of 1] Compiling Main             ( third_functions.hs, interpreted )
Ok, one module loaded.
>>> sum' []
0
>>> sum' [2, 3, 5, 7]
17
>>> sum' [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
55

>>> head' ["red", "black", "blue"]
"red"
>>> tail' ["red", "black", "blue"]
["black","blue"]
```

```
>>> halve [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
>>> halve [1,2,3,4,5,6,7,8]
([1,2,3,4],[5,6,7,8])
>>> halve [True,False,False,True]
([True,False],[False,True])
>>> halve [True]
([],[True])

>>> third ['a','b','c','d','e']
'c'
>>> third' ['a','b','c','d','e']
'c'
>>> third'' ['a','b','c','d','e']
'c'

>>> distinct [11, 22, 33, 22, 11]
False
>>> distinct [11, 22, 33, 44, 55, 66, 77]
True
>>> distinct []
True
>>> distinct [1]
True
>>> distinct [[], [1], [1,1] ]
True
>>>
```

Code

```
-------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------
--- third_functions.hs contains some simple list processing function definitions

-------------------------------------------------------------------------------------
-- Thing 1

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

-------------------------------------------------------------------------------------
-- Thing 2

head' :: [a] -> a
head' (x:_) = x

tail' :: [a] -> [a]
tail' (_:xs) = xs

-------------------------------------------------------------------------------------
```

```
-- Thing 3

halve :: [a] -> ([a],[a])
halve xs = (take n xs, drop n xs)
   where n = div (length xs) 2

-------------------------------------------------------------------------------
-- Thing 4: Write third three ways, assuming a list of at least 3 elements

-- head/tail variant
third :: [a] -> a
third xs = head ( tail ( tail xs ) )

-- direct reference
third' :: [a] -> a
third' xs = xs !! 2

-- pattern matching
third'' :: [a] -> a
third'' (_:_:x:_) = x

-------------------------------------------------------------------------------
-- Thing 5

distinct :: Eq a => [a] -> Bool
distinct [] = True
distinct (x:xs) = if ( elem x xs ) then False else distinct xs
```