```
-- Haskell is a functional programming language
-- Everything is immutable so once a value is set it is set forever
-- Functions can be passed as a parameter to other functions
-- Recursion is used often
-- Haskell has no for, while, or technically variables, but it does have
-- constants
-- Haskell is lazy in that it doesn't execute more then is needed and instead
-- just checks for errors

-- Best Free Haskell Book
-- http://learnyouahaskell.com/chapters

-- Type ghci to open it up in your terminal
-- Load script with :l haskelltut
-- :quit exits the GHCi

-- Import a module
import Data.List
import System.IO

{-
Beginning of multiline comment
-}

-- ---------- DATA TYPES ----------
-- Haskell uses type inference meaning it decides on the data type based on the -- value stored in it
-- Haskell is statically typed and can't switch type after compiling
-- Values can't be changed (Immutable)
-- You can use :t in the terminal to get the data type (:t value)
-- Int : Whole number -2^63 - 2^63
-- :: Int defines that maxInt is an Int
maxInt = maxBound :: Int
minInt = minBound :: Int

-- Integer : Unbounded whole number

-- Float : Single precision floating point number
-- Double : Double precision floating point number (11 pts precision)
bigFloat = 3.99999999999 + 0.00000000005

-- Bool : True or False
-- Char : Single unicode character denoted with single quotes
-- Tuple : Can store a list made up of many data types

-- You declare the permanent value of a variable like this
always5 :: Int
always5 = 5

-- ---------- MATH ----------
-- Something crazy to start
sumOfVals = sum [1..1000]

addEx = 5 + 4
subEx = 5 - 4
multEx = 5 * 4
divEx = 5 / 4
```

```
-- mod is a prefix operator
modEx = mod 5 4

-- With back ticks we can use it as an infix operator
modEx2 = 5 `mod` 4

-- Negative numbers must be surrounded with parentheses
negNumEx = 5 + (-4)

-- If you define an Int you must use fromIntegral to use it with sqrt
-- :t sqrt shows that it returns a floating point number
num9 = 9 ::Int
sqrtOf9 = sqrt (fromIntegral num9)

-- Built in math functions
piVal = pi
ePow9 = exp 9
logOf9 = log 9
squared9 = 9 ** 2
truncateVal = truncate 9.999
roundVal = round 9.999
ceilingVal = ceiling 9.999
floorVal = floor 9.999

-- Also sin, cos, tan, asin, atan, acos, sinh, tanh, cosh, asinh, atanh, acosh

trueAndFalse = True && False
trueOrFalse = True || False
notTrue = not(True)
```

```
-- Remember you use :t in the terminal to get the data type (:t value)
-- You can also see how functions use data types with :t

-- :t (+) = Num a => a -> a -> a
-- Type a is in the type class num, we receive 2 of them and return 1

-- :t truncate = (RealFrac a, Integral b) => a -> b

-- ---------- LISTS ----------
-- Lists are singly linked and you can only add to the front of it

-- Lists store many elements of the same type
primeNumbers = [3,5,7,11]

-- Concatenate lists (Can be slow if your using a large list)
morePrimes = primeNumbers ++ [13,17,19,23,29]

-- You can use the cons operator to construct a list
favNums = 2 : 7 : 21 : 66 : []

-- You can make a list of lists
multList = [[3,5,7],[11,13,17]]

-- Quick way to add 1 value to the front of a list
morePrimes2 = 2 : morePrimes

-- Get number of elements in the list
lenPrime = length morePrimes2
```

```haskell
-- Reverse the list
revPrime = reverse morePrimes2

-- return True if list is empty
isListEmpty = null morePrimes2

-- Get the number in index 1
secondPrime = morePrimes2 !! 1

-- Gets the 1st value in a list
firstPrime = head morePrimes2

-- Gets the last value
lastPrime = last morePrimes2

-- Gets everything but the first value
primeTail = tail morePrimes2

-- Gets everything but the last value
primeInit = init morePrimes2

-- Get specified number of elements from the front of a list
first3Primes = take 3 morePrimes2

-- Return values left after removing specified values
removedPrimes = drop 3 morePrimes2

-- Check if value is in list
```

```
is7InList = 7 `elem` morePrimes2


-- Get max value

maxPrime = maximum morePrimes2


-- Get minimum value

minPrime = minimum morePrimes2


-- Sum values in list

sumPrimes = sum morePrimes2


-- Get product of values in list (Value all can evenly divide by)

newList = [2,3,5]

prodPrimes = product newList


-- Create list from 0 to 10

zeroToTen = [0..10]


-- Create list of evens by defining the step between the first 2 values

evenList = [2,4..20]


-- You can use letters as well

letterList = ['A','C'..'Z']


-- You can generate an infinite list and Haskell will only generate what you

-- need

infinPow10 = [10,20..]


-- repeat repeats a value a defined number of times
```

```
many2s = take 10 (repeat 2)

-- replicate generates a value a specified number of times
many3s = replicate 10 3

-- cycle replicates the values in a list indefinitely
cycleList = take 10 (cycle [1,2,3,4,5])

-- You could perform operations on all values in a list
-- Cycle through the list storing each value in x which is multiplied by 2 and
-- then stored in a new list
listTimes2 = [x * 2 | x <- [1..10]]

-- We can filter the results with conditions
listTimes3 = [x * 3 | x <- [1..20], x*3 <= 50]

-- Return all values that are divisible by 13 and 9
divisBy9N13 = [x | x <- [1..500], x `mod` 13 == 0, x `mod` 9 == 0]

-- Sort a list
sortedList = sort [9,1,8,3,4,7,6]

-- zipwith can combine lists using a function
sumOfLists = zipWith (+) [1,2,3,4,5] [6,7,8,9,10]

-- Filter returns a list of items that match a condition
listBiggerThen5 = filter (>5) sumOfLists

-- takeWhile returns list items until the condition is false
```

```
evensUpTo20 = takeWhile (<=20) [2,4..]

-- foldl applies the operation on each item of a list
-- foldr applies these operations from the right
multOfList = foldl (*) 1 [2,3,4,5]

-- ---------- LIST COMPREHENSION ----------

-- We can generate a list from 1 to 10 to the power of 3
pow3List = [3^n | n <- [1..10]]

-- We can filter the results to only show values divisible by 9
pow3ListDiv9 = [3^n | n <- [1..10], 3^n `mod` 9 == 0]

-- Generate a multiplication table by multiplying x * y where y has the values
-- 1 through 10 and where x does as well
multTable = [[x * y | y <- [1..10]] | x <- [1..10]]

-- ---------- TUPLES ----------
-- Stores list of multiple data types, but has a fixed size

randTuple = (1,"Random tuple")

-- A tuple pair stores 2 values
bobSmith = ("Bob Smith",52)

-- Get the first value
bobsName = fst bobSmith
```

```
-- Get the second value
bobsAge = snd bobSmith

-- zip can combine values into tuple pairs
names = ["Bob","Mary","Tom"]
addresses = ["123 Main","234 North","567 South"]

namesNAddress = zip names addresses

-- ---------- FUNCTIONS ----------
-- ghc --make haskelltut compiles your program and executes the main function

-- Functions must start with lowercase letters

-- We can define functions and values in the GHCi with let
-- let num7 = 7
-- let getTriple x = x * 3

-- getTriple num7 = 21

-- main is a function that can be called in the terminal with main
main = do
        -- Prints the string with a new line
        putStrLn "What's your name: "

        -- Gets user input and stores it in name
        -- <- Pulls the name entered from an IO action
        name <- getLine
```

```haskell
        putStrLn ("Hello " ++ name)

-- Create function addMe
-- x is a parameter and the operation follows the equals sign
-- The data type passed in will work if it makes sense
-- Every function must return something
-- A function name can't begin with a capital letter
-- A function that doesn't receive parameters is called a definition or name

-- You can define a type declaration for functions
-- funcName :: param1 -> param2 -> returnType
addMe :: Int -> Int -> Int

-- funcName param1 param2 = operations (Returned Value)
-- Execute with : addMe 4 5
addMe x y = x + y

-- Without type declaration you can add floats as well
sumMe x y = x + y

-- You can also add tuples : addTuples (1,2) (3,4) = (4,6)
addTuples :: (Int, Int) -> (Int, Int) -> (Int, Int)
addTuples (x, y) (x2, y2) = (x + x2, y + y2)

-- You can perform different actions based on values
whatAge :: Int -> String
whatAge 16 = "You can drive"
whatAge 18 = "You can vote"
whatAge 21 = "You're an adult"
```

```
-- The default
whatAge x = "Nothing Important"

-- Define that we expect an Int in and out
factorial :: Int -> Int

-- If 0 return a 1 (Recursive Function)
factorial 0 = 1
factorial n = n * factorial (n - 1)

-- 3 * factorial (2) : 6
-- 2 * factorial (1) : 2
-- 1 * factorial (0) : 1

-- You could also use product to calculate factorial
productFactorial n = product [1..n]

-- We can use guards that provide different actions based on conditions
isOdd :: Int -> Bool
isOdd n
        -- if the modulus using 2 equals 0 return False
        | n `mod` 2 == 0 = False

        -- Else return True
        | otherwise = True

-- This could be shortened to
isEven n = n `mod` 2 == 0
```

```
-- Use guards to define the school to output
whatGrade :: Int -> String
whatGrade age
        | (age >= 5) && (age <= 6) = "Kindergarten"
        | (age > 6) && (age <= 10) = "Elementary School"
        | (age > 10) && (age <= 14) = "Middle School"
        | (age > 14) && (age <= 18) = "High School"
        | otherwise = "Go to college"

-- The where clause keeps us from having to repeat a calculation
batAvgRating :: Double -> Double -> String
batAvgRating hits atBats
        | avg <= 0.200 = "Terrible Batting Average"
        | avg <= 0.250 = "Average Player"
        | avg <= 0.280 = "Your doing pretty good"
        | otherwise = "You're a Superstar"
        where avg = hits / atBats

-- You can access list items by separating letters with : or get everything but
-- the first item with xs
getListItems :: [Int] -> String
getListItems [] = "Your list is empty"
getListItems (x:[]) = "Your list contains " ++ show x
getListItems (x:y:[]) = "Your list contains " ++ show x ++ " and " ++ show y
getListItems (x:xs) = "The first item is " ++ show x ++ " and the rest are "
        ++ show xs

-- We can also get values with an As pattern
```

```
getFirstItem :: String -> String
getFirstItem [] = "Empty String"
getFirstItem all@(x:xs) = "The first letter in " ++ all ++ " is "
        ++ [x]

-- ---------- HIGHER ORDER FUNCTIONS ----------
-- Passing of functions as if they are variables

times4 :: Int -> Int
times4 x = x * 4

-- map applies a function to every item in the list
listTimes4 = map times4 [1,2,3,4,5]

-- Let's make map
multBy4 :: [Int] -> [Int]
multBy4 [] = []

-- Takes the 1st value off the list x, multiplies it by 4 and stores it in the
-- new list
-- xs is then passed back into multBy4 until there is nothing left of the list -- to process (Recursion)
multBy4 (x:xs) = times4 x : multBy4 xs

-- Check if strings are equal with recursion
areStringsEq :: [Char] -> [Char] -> Bool
areStringsEq [] [] = True
areStringsEq (x:xs) (y:ys) = x == y && areStringsEq xs ys
areStringsEq _ _ = False
```

```
-- PASSING A FUNCTION INTO A FUNCTION
-- (Int -> Int) says we expect a function that receives an Int and returns an
-- Int
doMult :: (Int -> Int) -> Int

-- We receive the function and pass 3 into it
doMult func = func 3

-- We pass in the function that multiplies by 4
num3Times4 = doMult times4

-- RETURNING A FUNCTION FROM A FUNCTION
getAddFunc :: Int -> (Int -> Int)

-- We can pass in the values to the function
getAddFunc x y = x + y

-- We could also get a function that adds 3 for example
adds3 = getAddFunc 3

fourPlus3 = adds3 4

-- We could use this function with map as well
threePlusList = map adds3 [1,2,3,4,5]

-- ---------- LAMBDA ----------
-- How we create functions without a name
-- \ represents lambda then you have the arguments -> and result
```

```
dbl1To10 = map (\x -> x * 2) [1..10]

-- ---------- CONDITIONALS ----------

-- Comparison Operators : < > <= >= == /=
-- Logical Operators : && || not

-- Every if statement must contain an else
doubleEvenNumber y =
        if (y `mod` 2 /= 0)
                then y
                else y * 2

-- We can use case statements
getClass :: Int -> String
getClass n = case n of
        5 -> "Go to Kindergarten"
        6 -> "Go to elementary school"
        _ -> "Go some place else"

-- ---------- MODULES ----------
-- You can group functions into modules. I showed previously how to load them
-- You can create your own module by creating a file that contains all your
-- functions and then list the functions at the top like this
-- module SampFunctions (getClass, doubleEvenNumber) where
-- They can then be imported with import SampFunctions

-- ---------- ENUMERATION TYPES ----------
-- Used when you want a list of possible types
-- Provide name, a list and then Show converts into a String for printing

data BaseballPlayer = Pitcher
                                        | Catcher
                                        | Infield
                                        | Outfield
                                deriving Show

barryBonds :: BaseballPlayer -> Bool
barryBonds Outfield = True

barryInOF = print(barryBonds Outfield)

-- ---------- CUSTOM TYPES ----------
-- You can store multiple values sort of like a struct to create custom types
data Customer = Customer String String Double
        deriving Show

-- Define Customer and its values
tomSmith :: Customer
tomSmith = Customer "Tom Smith" "123 Main St" 20.50

-- Define how we'll find the right customer (By Customer) and the return value
getBalance :: Customer -> Double
getBalance (Customer _ _ b) = b

tomSmithBal = print (getBalance tomSmith)

-- We can define a type with all possible values
```

```
data RPS = Rock | Paper | Scissors

shoot :: RPS -> RPS -> String
shoot Paper Rock = "Paper Beats Rock"
shoot Rock Scissors = "Rock Beats Scissors"
shoot Scissors Paper = "Scissors Beat Paper"
shoot Scissors Rock = "Scissors Loses to Rock"
shoot Paper Scissors = "Paper Loses to Scissors"
shoot Rock Paper = "Rock Loses to Paper"
shoot _ _ = "Error"

-- We could define 2 versions of a type
-- First 2 floats are center coordinates and then radius for Circle
-- First 2 floats are for upper left hand corner and bottom right hand corner
-- for the Rectangle
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
        deriving (Show)

-- :t Circle = Float -> Float -> Float -> Shape

-- Create a function to calculate area of shapes
area :: Shape -> Float
area (Circle _ _ r) = pi * r ^ 2
area (Rectangle x y x2 y2) = (abs (x2 - x)) * (abs (y2 -y))

-- Could also be area (Rectangle x y x2 y2) = (abs $ x2 - x) * (abs $ y2 -y)
-- $ means that anything that comes after it will take precedence over anything
-- that comes before (Alternative to adding parentheses)
-- The . operator allows you to chain functions to pass output on the right to
-- the input on the left
-- sumValue = putStrLn (show (1 + 2)) becomes
sumValue = putStrLn . show $ 1 + 2

-- Get area of shapes
areaOfCircle = area (Circle 50 60 20)
areaOfRectangle = area $ Rectangle 10 10 100 100

-- ---------- TYPE CLASSES ----------
-- Num, Eq, Ord and Show are type classes
-- Type classes correspond to sets of types which have certain operations
-- defined for them.
-- Polymorphic functions, which work with multiple parameter types, define
-- the types it works with through the use of type classes
-- For example (+) works with parameters of the type Num
-- :t (+) = Num a => a -> a -> a
-- This says that for any type a, as long as a is an instance of Num, + can take
-- 2 values and return an a of type Num

-- Create an Employee and add the ability to check if they are equal
data Employee = Employee { name :: String,
                                              position :: String,
                                              idNum :: Int
                                              } deriving (Eq, Show)

samSmith = Employee {name = "Sam Smith", position = "Manager", idNum = 1000}
pamMarx = Employee {name = "Pam Marx", position = "Sales", idNum = 1001}
```

```haskell
isSamPam = samSmith == pamMarx

-- We can print out data because of show
samSmithData = show samSmith

-- Make a type instance of the typeclass Eq and Show
data ShirtSize = S | M | L

instance Eq ShirtSize where
        S == S = True
        M == M = True
        L == L = True
        _ == _ = False

instance Show ShirtSize where
        show S = "Small"
        show M = "Medium"
        show L = "Large"

-- Check if S is in the list
smallAvail = S `elem` [S, M, L]

-- Get string value for ShirtSize
theSize = show S

-- Define a custom typeclass that checks for equality
-- a represents any type that implements the function areEqual
class MyEq a where
        areEqual :: a -> a -> Bool
```

```
-- Allow Bools to check for equality using areEqual
instance MyEq ShirtSize where
        areEqual S S = True
        areEqual M M = True
        areEqual L L = True
        areEqual _ _ = False

newSize = areEqual M M

-- ---------- I/O ----------

sayHello = do
        -- Prints the string with a new line
        putStrLn "What's your name: "

        -- Gets user input and stores it in name
        name <- getLine

        -- $ is used instead of the parentheses
        putStrLn $ "Hello " ++ name

-- File IO
-- Write to a file
writeToFile = do

        -- Open the file using WriteMode
        theFile <- openFile "test.txt" WriteMode
        -- Put the text in the file
        hPutStrLn theFile ("Random line of text")

        -- Close the file
        hClose theFile

readFromFile = do

        -- Open the file using ReadMode
        theFile2 <- openFile "test.txt" ReadMode

        -- Get the contents of the file
        contents <- hGetContents theFile2
        putStr contents

        -- Close the file
        hClose theFile2

-- ---------- EXAMPLE : FIBONACCI SEQUENCE ----------

-- Calculate the Fibonacci Sequence
-- 1, 1, 2, 3, 5, 8, ...

-- 1 : 1 : says to add 2 1s to the beginning of a list
-- | for every (a, b) add them
-- <- stores a 2 value tuple in a and b
-- tail : get all list items minus the first
-- zip creates pairs using the contents from 2 lists being the lists fib and the
-- list (tail fib)
```

```
fib = 1 : 1 : [a + b | (a, b) <- zip fib (tail fib) ]


-- First time through fib = 1 and (tail fib) = 1

-- The list is now [1, 1, 2] because a: 1 + b: 1 = 2


-- The second time through fib = 1 and (tail fib) = 2

-- The list is now [1, 1, 2, 3] because a: 1 + b: 2 = 3


fib300 = fib !! 300 -- Gets the value stored in index 300 of the list


-- take 20 fib returns the first 20 Fibonacci numbers
```