-- 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 than 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

-- 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

-- than 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

listBiggerThan5 = 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