```haskell
-- MODULES --

-- LOADING MODULES --

-- When importing modules, import them before degining any functions and each on thier own line.
import Data.List
import qualified Data.Map as Map
import Data.Function (on)
import Data.Char
import qualified Data.Set as Set
--import Geometry
import qualified Geometry.Sphere as Sphere
import qualified Geometry.Cuboid as Cuboid
import qualified Geometry.Cube as Cube
-- Data.List contains useful functions for dealing with lists.
-- One function in the Data.list moduel is the nub function, which takes a list and returns a list that has any duplicate elements removed.
-- Function to find the length of a list excluding duplicate elements.
numUniques :: (Eq a) => [a] -> Int
numUniques = length . nub
-- You can also add modules into your current GHCI session with :m + <module name> <module name> <module name>, etc.
-- If you only need a few functions from a module, you can say
-- ::: import <module name> (function, function, function)
-- So if we only needed the nub and sort functions, we could do 'import Data.List (nub, sort)'
-- We can also EXCLUDE certain functions when loading a module by using 'hiding'
-- So if we didn't want to have the nub function because it's a part of another module, we could import Data.List like
-- ::: 'import Data.List hiding (nub)'

-- Another way to import functions from a certian module that may interfere with the functions from another module is to use 'qualified'
-- Using 'qualified' means that we NEED to call a function like <module name>.<function name>, so like 'Data.List.nub' instead of just 'nub'
-- We'd import the Data.List module like
-- ::: 'import qualified Data.Map'
-- We can simplify the name so we don't have to type Data.List every time by using 'as' in our imports. So we can say like 'D.nub', which would
-- be the same as 'Data.List.nub.' We do this by importing like
-- ::: 'import qualified Data.List as D'

-- DATA.LIST --
-- Data.List is all about lists, and it's got functions like map and filter in it. We don't need to qualify the import becuase it's got no
-- functions that clash with Prelude.
-- intersperse takes an element and a list and puts that element in between each element of the list
inter :: String
inter = intersperse '.' "MONKEY"

-- intercalate takes a list and a list of lists, then puts the list in between each of the lists in the list of lists and then flattens it all
interCalc :: [Int]
interCalc = intercalate [0,0,0] [[1,2,3], [4,5,6], [7,8,9]]

-- transpose takes a list of lists and then "turns" them into a 2D matrix and returns the columns as lists of lists
transposeExample :: [[Int]]
transposeExample = transpose [[1,2,3],[4,5,6],[7,8,9]]
-- If we had the polynomials 3x^2 + 5x + 9, 10x^3 + 9 and 8x^3 + 5x^2 + x - 1, and we wanted to sum them all up, we can use transpose
transposePoly :: [Int]
transposePoly = map sum $ transpose [[0,3,5,9], [10,0,0,9], [8,5,1,-1]]

-- concat flstens a list of lists into a list of elements
concatTest :: [Char]
concatTest = concat ["hello", " ","there"]

-- concatMap is the same thing as mapping a function over a list and then flattening that list.
concatMapTest :: [Int]
concatMapTest = concatMap (replicate 4) [1..3]

-- 'and' takes a list of boolean values and returns true only if all the elements of the list are True.
andTest :: Bool
andTest = and $ map (<4) [1..3]
-- This will return False because there is an element that isn't less than three.
andTestF :: Bool
andTestF = and $ map (<3) [1..3]

-- 'or' is like 'and', but it returns True if ANY of the values in the list are True.
orTest :: Bool
orTest = or $ map (<3) [2..50]

-- 'any' takes a predicate and a list and returns True if any of the values in the list satisfy the predicate.
anyTest :: Bool
anyTest = any (`elem` ['A'..'Z']) "HEYGUYSwhatsup"

-- 'all' takes a predicate and a list and returns True only if ALL of the values in the list satisfy the predicate
allTest :: Bool
allTest = all (`elem` ['A'..'Z']) "HEYGUYSwhatsup"

-- iterate takes a function and a starting value and iterates that function over the starting values. Returns an infinite list.
iterateTest :: [Int]
iterateTest = take 10 $ iterate (*2) 1

-- splitAt takes a number and a list and it splits the list at the position that the number represents
splitAtTest :: ([Char], [Char])
splitAtTest = splitAt 5 "hellothere"

-- takeWhile takes a predicate and a list and takes values from the list while the predicate is true. When it's no longer true, the function stops
takeWhile' :: [Int]
takeWhile' = takeWhile (>3) [6,5,4,3,2,1,3,2,5,7,8,2,6,4]
-- Function to determine the sum of all cubes that are less than 10,000.
sumCubes :: Int
sumCubes = sum $ takeWhile (<10000) $ map (^3) [1..]

-- dropWhile is like takeWhile, but it drops the elements from a list while the predicate is true.
dropWhile' :: [Int]
dropWhile' = dropWhile (<3) [1..10]
dropWhile'' :: [Char]
dropWhile'' = dropWhile (/=' ') "Hello there."

-- Example that shows how we can use dropWhile in order to determine when an event occours given a few data points.
stocks :: (Double, Int, Int, Int)
```

```haskell
stocks = head (dropWhile (\(val,y,m,d) -> val < 1000) [(994.4,2008,9,1),(995.2,2008,9,2),(999.2,2008,9,3),(1001.4,2008,9,4),(998.3,2008,9,5)])

-- span is like takeWhile, but it returns a double of lists. One is what takeWhile would return, and the other is what it would leave behind.
spanExample :: [Char]
spanExample =
        let
                (fw, rest) = span (/=' ') "This is a sentence."
        in
                "First word: " ++ fw ++ ". Rest: " ++ rest

-- break will break a list in two and return a double is lists when the predicate is first met somewhat like span.
breakOn :: ([Int], [Int])
breakOn = break (==4) [1..10]

-- sort just sorts a list. The elements of the list must be of the Ord typeclass because they need to be able to be ordered.
sort' :: [Int]
sort' = sort [1,5,1,3,6,1,88,2,6,4,23]

-- group takes a list and returns a list of lists. Each of the sublists contains the same value. So [1,1,2,3,3,3] -> [[1,1],[2],[3,3,3]]
-- however, the elements must be adjecent to each other.
group' :: [[Char]]
group' = group "Hello there. I am a list."
-- if we sort a list before we group it, we can see how many of each element are in a list
howMany :: [(Int, Int)]
howMany = map (\l@(x:xs) -> (x,length l)) . group . sort $ [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]

-- inits is like init, but it applies recursively to the list that it's given. Returns a list of lists.
initsTest :: [[Char]]
initsTest = inits "this is a list"

-- tails is just like inits, but it works backwards. Starts with the whole list and then gradually reduces the length.
tailsTest :: [[Char]]
tailsTest = tails "this is a list"

-- function to search a list for a sublist
listSearch :: (Eq a) => [a] -> [a] -> Bool
listSearch needle haystack =
        let nlength = length needle
        in foldl (\acc x -> if take nlength x == needle then True else acc) False $ tails haystack

-- isInfixOf does the exact same thing as listSearch.
isInfixOfTest :: Bool
isInfixOfTest = isInfixOf "cat" "there's a cat in here"

-- isPrefixOf checks to see if your provided list is at the beginning of the list you're searching through
isPrefixOfTest :: Bool
isPrefixOfTest = isPrefixOf "hey" "oh hey there!"

-- isSuffixOf does the same thing as isPrefixOf, but it checks to see if the list you're searching through ENDS with your value
isSuffixOfTest :: Bool
isSuffixOfTest = isSuffixOf "there!" "oh hey there!"

-- partition takes a predicate and a list and returns a double of lists. One list contains elements that satisfy the predicate and the other
-- contains the rest of the elements. It is important to remember that it's different from span or break in that it goes through the
-- entire list, while span and break both end when the predicate is no longer met.
partitionTest :: ([Int], [Int])
partitionTest = partition (<6) [1..10]

-- find takes a predicate and a list and returns the first element that satisfies it (a Maybe value.) That is, either Nothing or Just _, where _
-- is the element you searched for.
findTest :: Maybe Char
findTest = find (=='a') "this is a cat"

-- Rework the stocks function from earlier so that in case the stocks never go over 1,000, we don't get an arror for calling head on
-- an empty list.
stocks' :: Maybe (Double, Int, Int, Int)
stocks' = find (\(val, y, m, d) -> val > 1000) $ [(994.4,2008,9,1),(995.2,2008,9,2),(999.2,2008,9,3),(1001.4,2008,9,4),(998.3,2008,9,5)]

-- 'elem' and 'notElem' check if an element is a part of a list.

-- elemIndex is like elem, but it returns the index of the first element in a list that we're looking for.
elemIndexTest :: Maybe Int
elemIndexTest = elemIndex 4 [1,2,3,4,5,6]

-- elemIndicies is like elemIndex, but it returns a list of indices. Or it can return an empty list if there is nothing there.
elemIndicesTest :: [Int]
elemIndicesTest = elemIndices 5 [1,2,3,4,5,1,7,2,6,5,2,7,3,5]

-- findIndex takes a predicate and a list and returns a Maybe value upon the first occourence of a match
findIndexTest :: Maybe Int
findIndexTest = findIndex (`elem` [1..3]) [7,5,7,4,3,8,54,2,1]

-- findIndices takes a list and a predicate and returns all indexes of elements that match.
findIndicesTest :: [Int]
findIndicesTest = findIndices (`elem` [1..3]) [8,3,7,1,4,7,2,7,0,4,2,4,6,1,7,3]

-- zip3 and zipWith3 are like zip and zipWith, but they zip 3 lists together. zip_/zipWith_ go all the way up to 7.
zip3' :: [(Int, Int, Int)]
zip3' = zip3 [1,2,3] [4,5,6] [7,8,9]
zipWith3' :: [Int]
zipWith3' = zipWith3 (\x y z -> x + y + z) [1,2,3] [4,5,2,2] [2,2,3]

-- 'lines' is a useful function because it breaks up all new lines into new elements in a list
lines' :: [String]
lines' = lines "hello\nthere\nI\nam\na\nlist"

-- unlines is the opposite of lines. It takes a list of strings and joins them together using \n
unlines' :: String
unlines' = unlines lines'

-- words is for splitting up a string of words into a list of words. Breaks on a space.
words' :: [String]
words' = words "hello there I am a string."
```

```haskell
-- unwords is the exact opposite of words. Takes a list of strings and joins them on a space.
unwords' :: String
unwords' = unwords words'

-- delete takes an element and a list and deletes the first occurence of that element in the list
delete' :: String
delete' = delete 'h' "hey there ghang!"
delete'' :: String
delete'' = delete 'h' . delete 'h' . delete 'h' $ "hey there ghang!"

-- '\\' is the list difference function. It first takes a list and then a list of elements to remove from that list. If you have a list
-- [1,2,2,3,4,5] and a secondary list [1,2,3], then it'll return [2,4,5] because there was only one 2 in the list of items to remove.
removeSet :: [Int]
removeSet = [1..10] \\ [1,5,6,6]
-- doing this is like doing: delete 1 . delete 5 . delete 6 . delete 6 $ [1..10]

-- union is a function that joins lists together, but removes any duplicates. Duplicates are removed from the second list.
union' :: [Int]
union' = union [1..7] [5..10]

-- intersect returns a list of elements that are in both lists provided.
intersect' :: [Char]
intersect' = intersect "hey there man." "what's up dude?"
intersect'' :: [Int]
intersect'' = intersect [1,2,3,4,5,5,5] [4,5,6,7,8]

-- insert takes an element and a list of elements. It will go through the list until it finds an element that is greater than the element
-- you want to insert. Once it finds an element greater, it will insert your element right before it. If you do this on a sorted list, it will
-- keep the list sorted.
insert' :: String
insert' = insert 'g' $ ['a'..'f'] ++ ['h'..'z']
insert'' :: [Int]
insert'' = insert 4 [1,2,6,2,7,4,10,2,4,2,3,6,2,3,6,2,3,6]

-- IMPORTANT --
-- We have our length, take, drop, splitAt, !!, and replicate functions, but they can cause some problems because they either take or return
-- an Int value. Say you wanted to find the average value of a [Int]. You could do something like 'let xs = [1..6] in sum xs / length xs' in
-- order to get that, but Haskell would yell at you since you can't divide with an Int value. You can use genericLength (or genaricANYTHING)
-- instead of length,  because genericLength returns a Num value, which can represent a floating number instead of an Int. Doing
-- 'let xs = [1..6] in sum xs / genericLength xs' would give us a real result.

-- We also have our nub, delete, union, intersect, and group functions that take Eq values. We can call nubBy, deleteBy, unionBy, intersectBy,
-- and groupBy. These functions take a function that returns a boolean value to determine equality.

-- Then we have functions like sortBy, insertBy, maximumBy, and minimumBy that take a function that returns an ORDERING. So GT, LT or EQ.
-- For example, sort is the same as sortBy compare because you're sorting by what element is larger.
-- Function to allow us to sort a list of lsts by the length of the sublist.
sortList :: [[Int]] -> [[Int]]
sortList xs = sortBy (compare `on` length) xs


-- DATA.CHAR --
-- Data.Char contains a bunch of functions that will give you information about a character that you give them.
-- Some of these functions consist of isControl, isSpace, isLower, isUpper, isAlpha, isAlphaNum, isPrint, isDigit, isOctDigit, isHexDigit,
-- etc, etc. (http://learnyouahaskell.com/modules#data-char) ALL of these predicates have the type signature of 'Char -> Bool'
-- Almost all the time, you'll use these functions to filter strings.
-- Say we want to see if a username is all alphanumeric for an account.
isAlphaNumTest :: Bool
isAlphaNumTest = all isAlphaNum "bobby823"
isAlphaNumTest' :: Bool
isAlphaNumTest' = all isAlphaNum "this is a name"

-- Let's use isSpace to emulate words from Data.List
words'' :: [String]
words'' = filter (not . any isSpace) . groupBy ((==) `on` isSpace) $ "hello there I am a list"

-- Data.Char also has a function that returns a data type of GeneralCategory. These General Categories are things like Space, UppercaseLetter,
-- MathSymbol, etc.
generalCategoryTest :: [GeneralCategory]
generalCategoryTest = map generalCategory " h19*/1>|?"
-- The GeneralCategory typeclass is a part of the Eq typeclass, so you can do things like 'generalCategory 'a' == LowercaseLetter'
-- topUpper converts a character to its upper case form. Spaces, numbers, and the like will remain unchanged.
-- toLower converts a character to lower case.
-- toTitle converts a character to its title case (which is upper case in most cases)
-- digitToInt converts a character to an integer. To succeed, the character must be in '0'..'9', 'a'..'f', or 'A'..'F'
digitToInt' :: [Int]
digitToInt' = map digitToInt "FF85AB"

-- intToDigit is the opposite of digitToInt in that it takes an int and it converts it to the digit form.

-- chr and ord convert numbers to characters and characters to numbers.
chrOrd :: Int
chrOrd = ord . chr $ 1
mapOrd :: [Int]
mapOrd = map ord "abcdefgh"

-- Encoding by shifting ca character by a certain number of ascii numbers
encode :: Int -> String -> String
encode shift msg =
                        let
                                ords = map ord msg
                                shifted = map (+shift) ords
                        in map chr shifted
-- We could also write it in one line like this
encode' :: Int -> String -> String
encode' shift msg = map (chr . (+shift) . ord) msg

-- Now let's make a decode function
decode :: Int -> String -> String
decode shift msg = encode (negate shift) msg

-- DATA.MAP --
-- Association lists are used to store key-value pairs where ordering doesn't really matter. We could use association lists to store phone
-- numbers and names. We don't care about the order they're stored. We just want to be able to access the right value for the key.
```

```haskell
-- Function for calling the more-important findKey function
associateTest :: String -> String
associateTest key = let phones = [("betty","555-2938")
                                               ,("bonnie","452-2928")
                                               ,("patsy","493-2928")
                                               ,("lucille","205-2928")
                                               ,("wendy","939-8282")
                                               ,("penny","853-2492")]
                        in findKey key phones

-- Function to find the value of something given the key
findKey :: (Eq a) => a -> [(a,v)] -> v
findKey key xs = snd . head . filter (\(k,v) -> key == k) $ xs

-- But what if our key doesn't exist in a list? We would crash the program. So let's use Maybe values
findKey' :: (Eq k) => k -> [(k,v)] -> Maybe v
findKey' _ [] = Nothing
findKey' key ((k,v):xs) = if key == k
                                                                then Just v
                                                                else findKey' key xs

-- Same as the other function above, but this one uses folds.
findKey'' :: (Eq a) => a -> [(a,v)] -> Maybe v
findKey'' key = foldl (\acc (k,v) -> if key == k then Just v else acc) Nothing

-- These were all examples of the lookup function from Data.Map.
-- The Data.Map module offers association lists that are much faster beause they are impimented internally with trees, and it also offers a lot
-- of useful function. From now on, I'll say we're working with maps instead of association lists
-- Imported qualified Data.Map at the top.
-- We'll always want to use Data.Map for when we have key-value associations unless the keys and values aren't a part of the Ord typeclass.

-- Map.fromList takes an association list and returns a map with the same associations. If there are duplicate keys, it discards the ones
-- that are closer to the beginning.
fromList' :: Map.Map Int Int
fromList' = Map.fromList [(1,2),(3,4),(3,2),(5,5)]
-- When we were doing maps with just regular lists, we only had to make the keys and values Eq, but with association lists, we need to make the
-- types Ord because that's the type of the fromList function.

-- empty just returns an empty map.
empty' :: Map.Map Int Int
empty' = Map.empty

-- insert takes a key, value, and map ad adds the key and value to the map.
insertMap :: Map.Map Int Int
insertMap = Map.insert 50 100 Map.empty
insertMap' :: Map.Map Int Int
insertMap' = Map.insert 50 100 $ Map.insert 3 10 $ Map.insert 60 8 $ Map.empty

-- We can impliment out own fromList using folds and Map.insert.
fromList'' :: (Ord k) => [(k,v)] -> Map.Map k v
fromList'' = foldr (\(k,v) acc -> Map.insert k v acc) Map.empty

-- null checks if a map is empty.
nullTest :: Bool
nullTest = Map.null Map.empty

-- size reports the size of a map.
sizeTest :: Int
sizeTest = Map.size $ Map.fromList [(2,4),(3,3),(4,2),(5,4),(6,4)]

-- singleton takes a key and a value and returns a map that has only one mapping
singletonTest :: Map.Map Int Int
singletonTest = Map.singleton 3 9

-- lookup takes a key and returns a Maybe value.
lookupTest :: Maybe Int
lookupTest = Map.lookup 1 $ Map.fromList [(1,2), (3,4), (5,6)]

-- member takes a key and a map and checks to see if they key is in the map.
memberTest :: Bool
memberTest = Map.member 3 $ Map.fromList [(1,2), (3,4), (5,6)]

-- map works like the default map, but it only maps over the values in the maps.
mapTest :: Map.Map Int Int
mapTest = Map.map (*100) $ Map.fromList [(1,2), (3,4), (5,6)]
-- filter works like the default filter, but it filters by the values
filterTest :: Map.Map Int Int
filterTest = Map.filter (`elem` [1,3,4,6]) $ Map.fromList [(1,2), (3,4), (5,6)]

-- toList is the inverse of fromList. Takes a map and returns a list.
toListTest :: [(Int, Int)]
toListTest = Map.toList $ Map.fromList [(1,2), (3,4), (5,6)]

-- keys returns a list of keys
keysTest :: [Int]
keysTest = Map.keys $ Map.fromList [(1,2), (3,4), (5,6)]

-- elems returns a list of elements
elemsTest :: [Int]
elemsTest = Map.elems $ Map.fromList [(1,2), (3,4), (5,6)]

-- What if we have a few numbers for one person like this?
         -- phoneBook =
     --   [("betty","555-2938")
     --   ,("betty","342-2492")
     --   ,("bonnie","452-2928")
     --   ,("patsy","493-2928")
     --   ,("patsy","943-2929")
     --   ,("patsy","827-9162")
     --   ,("lucille","205-2928")
     --   ,("wendy","939-8282")
     --   ,("penny","853-2492")
     --   ,("penny","555-2111")
```

```haskell
    --   ]
-- If we use fromList, we'll lose some of the numbers in there. So we can use Map.fromListWith instead.
phoneBookToMap :: (Ord k) => [(k,String)] -> Map.Map k String
phoneBookToMap xs = Map.fromListWith (\number1 number2 -> number1 ++ ", " ++ number2) xs
-- This takes all the duplicate values and puts them into one value.
phoneBookToMap' :: (Ord k) => [(k, String)] -> Map.Map k [String]
phoneBookToMap' xs = Map.fromListWith (++) $ map (\(k,v) -> (k,[v])) xs

-- If we have an association list of numbers, we can get it to take the max or the min.
maxNums :: (Ord k, Ord v) => [(k,v)] -> Map.Map k v
maxNums xs = Map.fromListWith max xs

-- InsertWith is like frommListWith, so if there is a duplicate key, the function you give insertWith decides what happens with the value.
insertWithTest :: (Ord k) => k -> String -> [(k,String)] -> Map.Map k String
insertWithTest k v xs = Map.insertWith (\val2 val1 -> val1 ++ " " ++ val2) k v $ Map.fromList xs


-- DATA.SET --
-- A lot of the functions in Data.Set clash with those in Prelude and Data.Map, so it's imported as Set at the top of this file.
-- What if we want to see what elements are used in two lists?
-- fromList generates a set of unique values.
usedInBoth :: (Set.Set Char -> Set.Set Char -> a) -> a
usedInBoth x =
                        let
                                text1 = "I just had an anime dream. Anime... Reality... Are they so different?"
                                text2 = "The old man left his garbage can out and now his trash is all over my lawn!"
                                set1 = Set.fromList text1
                                set2 = Set.fromList text2
                        in
                                x set1 set2
-- We can use the difference function to tell us which values are in one but not the other. (feed to usedInBoth)
-- We can use the union function to see what characters are used by both sentences. (feed to usedInBoth)

-- The null, size, member, empty, singleton, insert, and delete functions all work as you would expect them to from Data.Map.

-- Set.isSubsetOf will tell us if a set exists within a set that we provide.
subset :: [Char] -> [Char] -> Bool
subset x s = Set.fromList x `Set.isSubsetOf` Set.fromList s

-- Set.isProperSubsetOf will check to see if a set is part of a set, but is not the whole set. So "hello" "hello" returns False, but
-- "hello" "hello there" returns True
properSubset :: [Char] -> [Char] -> Bool
properSubset x s = Set.fromList x `Set.isProperSubsetOf` Set.fromList s

-- We can also filter a set.
filterSet :: [Int] -> Set.Set Int
filterSet x = Set.filter odd $ Set.fromList x

-- We can then also map a function over a set.
mapSet :: [Int] -> Set.Set Int
mapSet x = Set.map (+1) $ Set.fromList x

-- It's much faster to nub a list by using Sets because they are automatically converted to a list of unique characters with Set.fromList
setNub :: [Char] -> [Char]
setNub xs = Set.toList $ Set.fromList xs
-- setNub is faster on longer lists, but it doesn't preserve the ordering of the list, unfortunately.


-- MAKING OUR OWN MODULES --
-- Created a file called Geometry.hs
-- Each module can have its own submodules. For example, I've created the Sphere, Cuboid, and Cube submodules to Geometry, each in their
-- own files.

-- IF WE WANT TO HAVE THE MODULES IN A SUBFOLDER, WE HAVE TO DO 'import [SUBFOLDER].[MODULE]' and we have to name the module "[SUBFOLDER].[MODULE]"
go :: Float
go = Sphere.volume 3.264

go' :: Float
go' = Cuboid.area 5 9 3

go'' :: Float
go'' = Cube.volume 9
```