

# Using and Making Modules

Stephen A. Edwards

Columbia University

Fall 2021



## Using Modules

Import every name from a module:

```
import Data.List  
  
numUniques :: Eq a => [a] -> Int  
numUniques = length . nub
```

In GHCi,

```
Prelude> :m + Data.List  
Prelude Data.List> :m + Data.Map  
Prelude Data.List Data.Map> :set prompt "ghci> "  
ghci> -- under control
```

```
Prelude> :m + Data.List Data.Map  
Prelude Data.List Data.Map> -- Multiple ones
```

## Import Variants

```
import Data.List (nub, sort)           -- Only nub and sort
import Data.List hiding (nub, sort)  -- All but nub and sort
import qualified Data.List           -- Data.List.nub, etc.
import qualified Data.List as L      -- L.nub, L.sort, etc.
```

```
Prelude> :m + Data.List
```

```
Prelude Data.List> intersperse '*' "MASH"  
"M*A*S*H"
```

```
Prelude Data.List> intercalate ", " ["Foo","Bar","Baz"]  
"Foo, Bar, Baz"
```

```
Prelude Data.List> transpose [[1,2,3],[4,5,6],[7,8,9]]  
[[1,4,7],[2,5,8],[3,6,9]]
```

```
Prelude Data.List> concat ["PFP ", "is ", "fun"]  
"PFP is fun"
```

```
Prelude Data.List> concatMap (replicate 3) [1..3]  
[1,1,1,2,2,2,3,3,3]
```

```
Prelude Data.List> and [True, False, True]
False
-- and = foldr (&&) True
Prelude Data.List> and [True, True]
True

Prelude Data.List> or [True, False, True]
True
-- or = foldl (||) False

Prelude Data.List> any (==4) [1..5]
True
-- any p = or . map p

Prelude Data.List> all (>4) [5..10]
True
-- all p = and . map p
Prelude Data.List> all (<=4) [5..10]
False
```

```
Prelude Data.List> take 5 $ iterate (*2) 1  
[1,2,4,8,16]
```

```
Prelude Data.List> splitAt 3 "pfprocks"  
("pfp","rocks")
```

```
Prelude Data.List> takeWhile (<10) [1..]  
[1,2,3,4,5,6,7,8,9] -- Prefix of list
```

```
Prelude Data.List> dropWhile (<5) [1..10]  
[5,6,7,8,9,10] -- Suffix of list
```

```
Prelude Data.List> span (<5) [1..10]  
([1,2,3,4],[5,6,7,8,9,10]) -- Prefix/suffix split
```

```
Prelude Data.List> sort [8,5,3,2,1,6,4,2]
[1,2,2,3,4,5,6,8]
```

```
Prelude Data.List> group [1,1,1,2,2,1,1,1,1,5,5,4,3,3]
[[1,1,1],[2,2],[1,1,1,1],[5,5],[4],[3,3]]
```

```
Prelude Data.List> maxRun = maximum . map length . group
Prelude Data.List> maxRun [1,1,1,2,2,1,1,1,1,5,5,4,3,3]
4
```

```
Prelude Data.List> inits "whoa!"
["","w","wh","who","whoa","whoa!"]
```

```
Prelude Data.List> tails "whoa!"
["whoa!","hoa!","oa!","a!","!",""]
```

```
Prelude Data.List> let s = "whoa" in zip (inits s) (tails s)
[("", "whoa"), ("w", "hoa"), ("wh", "oa"), ("who", "a"), ("whoa", "")]
```

## Searching Lists

```
isPrefixOf      :: Eq a => [a] -> [a] -> Bool
isPrefixOf []   _      = True
isPrefixOf _    []     = False
isPrefixOf (x:xs) (y:ys) = x == y && isPrefixOf xs ys
```

```
Prelude Data.List> "PFP" `isPrefixOf` "PFP Rocks!"
True
Prelude Data.List> "PFP" `isPrefixOf` "PHP Rocks!"
False
Prelude Data.List> :set prompt "> "
> search needle haystack = any (isPrefixOf needle) (tails haystack)
> search "fun" "PFP is fun, dontcha know"
True
> search "fun" "Columbia"
False
```

Data.List calls it `isInfixOf` instead of `search`. There is also `isSuffixOf`



## Partition and Quicksort Revisited

```
Prelude Data.List> msg = "He Is Daring, Dumb, and Educated, Nancy"  
Prelude Data.List> partition (`elem` ['A'..'Z']) msg  
("HIDDEN", "e s aring, umb, and ducated, ancy")
```

```
import Data.List ( partition )
```

```
quicksort :: Ord a => [a] -> [a]
```

```
quicksort [] = []
```

```
quicksort (p:xs) = quicksort prefix ++ [p] ++ quicksort suffix  
                  where (prefix,suffix) = partition (<p) xs
```

```
*Main> :l quicksort3
```

```
[1 of 1] Compiling Main ( quicksort3.hs, interpreted )
```

```
Ok, one module loaded.
```

```
*Main> quicksort "the quick brown fox jumps over the lazy dog"
```

```
"      abcdeefghhijklmnooopqrrsttuuvwxyz"
```

## Lists as Text

```
Prelude> lines "first\nsecond\nthird\nfourth"  
["first","second","third","fourth"]
```

```
Prelude> unlines ["one","two","three"]  
"one\ntwo\nthree\n"
```

```
Prelude> words "The Quick Brown Fox Jumps"  
["The","Quick","Brown","Fox","Jumps"]
```

```
Prelude> unwords ["My","gosh","it's","full","of","stars"]  
"My gosh it's full of stars"
```

## Lists as Sets: Assumes Unique But Unordered

```
Prelude Data.List> nub [1,3,2,4,3,2,1,2,3,4,3,2,1]
[1,3,2,4]           -- Duplicates removed, unordered

Prelude Data.List> nub "the quick brown fox jumps over the lazy dog"
"the quickbrownfxjmpsvlazydg"

Prelude Data.List> delete 'e' "Stephen"
"Stphen"           -- Delete the first matching element

Prelude Data.List> ([1..10] ++ [1..3]) \\ [2,5,9]
[1,3,4,6,7,8,10,1,2,3] -- List difference: delete first matching

Prelude Data.List> "the quick brown fox" `union` ['a'..'z']
"the quick brown foxadgjlmpsvyz"

Prelude Data.List> "the quick brown fox" `intersect` ['a'..'m']
"heickbf"

Prelude Data.List> insert 'p' "almost"
"almopst" -- To last position where it's <=; maintains sorted order
```

```
genericLength    :: Num i => [a] -> i
genericTake      :: Integral i => i -> [a] -> [a]
genericDrop      :: Integral i => i -> [a] -> [a]
genericSplitAt   :: Integral i => i -> [a] -> ([a], [a])
genericIndex     :: Integral i => [a] -> i -> a
genericReplicate :: Integral i => i -> a -> [a]
```

```
nubBy            :: (a -> a -> Bool) -> [a] -> [a]
deleteBy         :: (a -> a -> Bool) -> a -> [a] -> [a]
deleteFirstsBy  :: (a -> a -> Bool) -> [a] -> [a] -> [a]
unionBy          :: (a -> a -> Bool) -> [a] -> [a] -> [a]
intersectBy     :: (a -> a -> Bool) -> [a] -> [a] -> [a]
groupBy         :: (a -> a -> Bool) -> [a] -> [[a]]
```

```
sortBy          :: (a -> a -> Ordering) -> [a] -> [a]
insertBy       :: (a -> a -> Ordering) -> a -> [a] -> [a]
maximumBy     :: Foldable t => (a -> a -> Ordering) -> t a -> a
minimumBy     :: Foldable t => (a -> a -> Ordering) -> t a -> a
```

## Data.Char: Character Type Predicates

```
isAscii,      isLatin1,      isControl,  
isAsciiUpper, isAsciiLower,  
isPrint,      isSpace,        isUpper,  
isLower,      isAlpha,          isDigit,  
isOctDigit,   isHexDigit,       isAlphaNum,  
isPunctuation, isSymbol           :: Char -> Bool
```

```
Prelude Data.Char> all isHexDigit "18deadBEEF"  
True  
Prelude Data.Char> all isHexDigit "gosh"  
False  
Prelude Data.Char> map generalCategory " \t\nA9?|"  
[Space,Control,Control,UppercaseLetter,DecimalNumber,  
OtherPunctuation,MathSymbol]
```

## Data.Char: Conversion Functions

```
Prelude Data.Char> map toUpper "the quick brown fox"  
"THE QUICK BROWN FOX"
```

```
Prelude Data.Char> map toLower "THE QUICK Brown FoX"  
"the quick brown fox"
```

```
Prelude Data.Char> map digitToInt "09afBC"  
[0,9,10,15,11,12]    -- Hex digits allowed
```

```
Prelude Data.Char> map intToDigit [4,2,10,15]  
"42af"              -- Inverse of digitToInt
```

```
Prelude Data.Char> map ord "!ABab"  
[32,33,65,66,97,98] -- ASCII/Unicode values
```

```
Prelude Data.Char> map chr [71,101,116,32,66,101,110,116]  
"Get Bent"         -- Inverse of ord
```

## Association Lists: Slow, Straightforward

```
phoneBook =  
  [("Jenny", "867-5309")  
  , ("Morris", "777-9311")  
  , ("Alessia", "273-8255")  
  , ("Tina", "606-0842")  
  , ("Alicia", "489-4608")  
  , ("Glenn", "736-5000")  
  ]  
find :: Eq k => k -> [(k, v)] -> v  
find k = snd . head . filter ((==k) . fst)
```

```
*Main> find "Alicia" phoneBook  
"489-4608"      -- Alicia is one of the keys  
*Main> find "Jenny" phoneBook  
"867-5309"  
*Main> find "Marty" phoneBook  
*** Exception: Prelude.head: empty list
```

```
Prelude> import qualified Data.Map as Map

Prelude Map> :t Map.fromList
Map.fromList :: Ord k => [(k, a)] -> Map.Map k a -- Ordered keys

Prelude Map> Map.fromList [("Jenny","837-5306"),("Alicia","489-4608")]
fromList [("Alicia","489-4608"),("Jenny","837-5306")]

Prelude Map> Map.empty
fromList [] -- The empty map

Prelude Map> Map.insert "Alicia" "489-4608" Map.empty
fromList [("Alicia","489-4608")] -- Add a pair

Prelude Map> fromList' = foldr \(k,v) m -> Map.insert k v m) Map.empty
Prelude Map> Map.null Map.empty
True -- Is the map empty?

Prelude Map> Map.null $ Map.fromList [(1,1)]
False

Prelude Map> Map.size $ Map.fromList [(1,1),(2,3)]
2 -- Number of pairs
```



```
Prelude Map> Map.singleton "Jenny" "867-5309"  
fromList [("Jenny","867-5309")]  
Prelude Map> Map.insert 1 "one" $ Map.singleton 0 "zero"  
fromList [(0,"zero"),(1,"one")]
```

```
*Main Map> phoneMap = Map.fromList phoneBook  
*Main Map> Map.lookup "Jenny" phoneMap  
Just "867-5309"  
*Main Map> Map.lookup "Freddy" phoneMap  
Nothing  
*Main Map> Map.member "Alicia" phoneMap  
True
```

```
Prelude Map> Map.map (*10) $ Map.fromList [(2,1),(3,5),(1,8)]  
fromList [(1,80),(2,10),(3,50)] -- Applied to values
```

```
Prelude Map> Map.filter odd $ Map.fromList [(x,x+3) | x <- [0..8]]  
fromList [(0,3),(2,5),(4,7),(6,9),(8,11)] -- Filter values
```

```
*Main Map> phoneMap = Map.fromList phoneBook
```

```
*Main Map> Map.keys phoneMap
```

```
["Alessia","Alicia","Jenny","Morris","Tina"]
```

```
*Main Map> Map.elems phoneMap
```

```
["273-8255","489-4608","867-5309","777-9311","606-0842"]
```

```
*Main Map> Map.toList phoneMap
```

```
[("Alessia","273-8255"),("Alicia","489-4608"), -- Sorted  
 ("Jenny","867-5309"),("Morris","777-9311"),  
 ("Tina","606-0842")]
```

```
Prelude Map> :set +m
```

```
Prelude Map> let dups = [(1,1),(1,20),(2,5),(1,300),(3,8),(3,80)]
```

```
Prelude Map| in Map.fromListWith (+) dups
```

```
fromList [(1,321),(2,5),(3,88)] -- Duplicate key's values added
```

```
Prelude> import qualified Data.Set as Set
Prelude Set> :t Set.fromList
Set.fromList :: Ord a => [a] -> Set.Set a

Prelude Set> set1 = Set.fromList "the quick brown fox jumps over"
Prelude Set> set2 = Set.fromList "pack my box with five dozen"
Prelude Set> set1
fromList " bcefhijklmnopqrstuvw"      -- Unique, sorted
Prelude Set> set2
fromList " abcdefhikmnoptvwxyz"        -- Unique, sorted
Prelude Set> Set.union set1 set2
fromList " abcdefhijklmnopqrstvwxyz"   -- in set1 or set2
Prelude Set> Set.intersection set1 set2
fromList " bcefhikmnoptvw"             -- in set1 and set2
Prelude Set> Set.difference set1 set2
fromList "jqrsu"                        -- in set1 but not set2
Prelude Set> Set.difference set2 set1
fromList "adyz"                          -- in set2 but not set1
```

```
Prelude Set> Set.null Set.empty
True
Prelude Set> Set.null $ Set.fromList [3,4,5,5,4,3]
False
Prelude Set> Set.size $ Set.fromList [3,4,5,5,4,3]
3
Prelude Set> Set.singleton 42
fromList [42]
Prelude Set> Set.insert 2 $ Set.insert 4 $ Set.singleton 1
fromList [1,2,4]
Prelude Set> Set.delete 7 $ Set.fromList [1..10]
fromList [1,2,3,4,5,6,8,9,10]
Prelude Set> 5 `Set.member` Set.fromList [1..10]
True
Prelude Set> 0 `Set.member` Set.fromList [1..10]
False
```

```
Prelude Set> :set prompt "> "
```

```
> Set.fromList [2..4] `Set.isSubsetOf` Set.fromList [0..10]  
True
```

```
> Set.fromList [2..4] `Set.isSubsetOf` Set.fromList [2..4]  
True
```

```
> Set.fromList [2..4] `Set.isProperSubsetOf` Set.fromList [2..4]  
False
```

```
> Set.fromList [2..4] `Set.isSubsetOf` Set.fromList [0..3]  
False
```

```
> Set.map (2^) $ Set.fromList [1..5]  
fromList [2,4,8,16,32]
```

```
> Set.filter odd $ Set.fromList [0..10]  
fromList [1,3,5,7,9]
```

## Writing a Module: Geometry.hs

```
module Geometry
( sphereVolume      -- Exported names
, cubeVolume
) where

sphereVolume :: Float -> Float
sphereVolume radius = (4.0 / 3.0) * pi * (radius ^ 3)

cubeVolume :: Float -> Float
cubeVolume side = cuboidVolume side side side

cuboidVolume :: Float -> Float -> Float -> Float
cuboidVolume a b c = rectangleArea a b * c

rectangleArea :: Float -> Float -> Float      -- Internal only
rectangleArea a b = a * b
```

## Using the Geometry Package

```
Prelude> :l Geometry
[1 of 1] Compiling Geometry          ( Geometry.hs, interpreted )
Ok, one module loaded.
*Geometry> :show modules
Geometry          ( Geometry.hs, interpreted )
*Geometry> :reload
Ok, one module loaded.

*Geometry> sphereVolume 10.0
4188.7905
*Geometry> cubeVolume 2
8.0
```

## Breaking up Modules

Create

Geom/Sphere.hs

Geom/Cube.hs

Geom/Cuboid.hs

```
Prelude> :l Geom.Sphere Geom.Cube
[1 of 3] Compiling Geom.Cuboid      ( Geom/Cuboid.hs, interpreted )
[2 of 3] Compiling Geom.Cube        ( Geom/Cube.hs, interpreted )
[3 of 3] Compiling Geom.Sphere      ( Geom/Sphere.hs, interpreted )
Ok, three modules loaded.
*Geom.Sphere> Geom.Cube.volume 2.0
8.0
```



## Geom/Sphere.hs

```
module Geom.Sphere
( volume
, area
) where

volume :: Float -> Float
volume radius = (4.0 / 3.0) * pi * (radius ^ 3)

area :: Float -> Float
area radius = 4 * pi * (radius ^ 2)
```

## Geom/Cuboid.hs

```
module Geom.Cuboid
( volume
, area
) where

volume :: Float -> Float -> Float -> Float
volume a b c = rectangleArea a b * c

area :: Float -> Float -> Float -> Float
area a b c = rectangleArea a b * 2 +
              rectangleArea a c * 2 +
              rectangleArea c b * 2

rectangleArea :: Float -> Float -> Float
rectangleArea a b = a * b
```

## Geom/Cube.hs

```
module Geom.Cube
( volume
, area
) where

import qualified Geom.Cuboid as Cuboid

volume :: Float -> Float
volume side = Cuboid.volume side side side

area :: Float -> Float
area side = Cuboid.area side side side
```