just click through the standard library reference and explore the modules and their functions. You can also view the Haskell source code for each module. Reading the source code of some modules is a really good way to learn Haskell and get a solid feel for it.

To search for functions or to find out where they're located, use Hoogle. It's a really awesome Haskell search engine, you can search by name, module name or even type signature.

Data.List

The **`Data.List`** module is all about lists, obviously. It provides some very useful functions for dealing with them. We've already met some of its functions (like **`map`** and **`filter`**) because the **`Prelude`** module exports some functions from **`Data.List`** for convenience. You don't have to import **`Data.List`** via a qualified import because it doesn't clash with any **`Prelude`** names except for those that **`Prelude`** already steals from **`Data.List`**. Let's take a look at some of the functions that we haven't met before.

`intersperse` takes an element and a list and then puts that element in between each pair of elements in the list. Here's a demonstration:

```
ghci> intersperse '.' "MONKEY"
"M.O.N.K.E.Y"
ghci> intersperse 0 [1,2,3,4,5,6]
[1,0,2,0,3,0,4,0,5,0,6]
```

`intercalate` takes a list of lists and a list. It then inserts that list in between all those lists and then flattens the result.

```
ghci> intercalate " " ["hey","there","guys"]
"hey there guys"
ghci> intercalate [0,0,0] [[1,2,3],[4,5,6],[7,8,9]]
[1,2,3,0,0,0,4,5,6,0,0,0,7,8,9]
```

`transpose` transposes a list of lists. If you look at a list of lists as a 2D matrix, the columns become the rows and vice versa.

```
ghci> transpose [[1,2,3],[4,5,6],[7,8,9]]
[[1,4,7],[2,5,8],[3,6,9]]
ghci> transpose ["hey","there","guys"]
["htg","ehu","yey","rs","e"]
```

Say we have the polynomials $3x^2 + 5x + 9$, $10x^3 + 9$ and $8x^3 + 5x^2 + x - 1$ and we want to add them together. We can use the lists **`[0,3,5,9]`**, **`[10,0,0,9]`** and **`[8,5,1,-1]`** to represent them in Haskell. Now, to add them, all we have to do is this:

```
ghci> map sum $ transpose [[0,3,5,9],[10,0,0,9],[8,5,1,-1]]
[18,8,6,17]
```

When we transpose these three lists, the third powers are then in the first row, the second powers in the second one and so on. Mapping **sum** to that produces our desired result.

`foldl'` and `foldl1'` are stricter versions of their respective lazy incarnations. When using lazy folds on really big lists, you might often get a stack overflow error. The culprit for that is that due to the lazy nature of the folds, the accumulator value isn't actually updated as the folding happens. What actually happens is that the accumulator kind of makes a promise that it will compute its value when asked to actually produce the result (also called a thunk). That happens for every intermediate accumulator and all those thunks overflow your stack. The strict folds aren't lazy buggers and actually compute the intermediate values as they go along instead of filling up your stack with thunks. So if you ever get stack overflow errors when doing lazy folds, try switching to their strict versions.

`concat` flattens a list of lists into just a list of elements.

```
ghci> concat ["foo","bar","car"]
"foobarcar"
ghci> concat [[3,4,5],[2,3,4],[2,1,1]]
[3,4,5,2,3,4,2,1,1]
```

It will just remove one level of nesting. So if you want to completely flatten **`[[[2,3],[3,4,5],[2]],[[2,3],[3,4]]]`**, which is a list of lists of lists, you have to concatenate it twice.

Doing `concatMap` is the same as first mapping a function to a list and then concatenating the list with **`concat`**.

```
ghci> concatMap (replicate 4) [1..3]
[1,1,1,1,2,2,2,2,3,3,3,3]
```

`and` takes a list of boolean values and returns **`True`** only if all the values in the list are **`True`**.

```
ghci> and $ map (>4) [5,6,7,8]
True
ghci> and $ map (==4) [4,4,4,3,4]
False
```

`or` is like **`and`**, only it returns **`True`** if any of the boolean values in a list is **`True`**.

```
ghci> or $ map (==4) [2,3,4,5,6,1]
True
ghci> or $ map (>4) [1,2,3]
False
```

`any` and `all` take a predicate and then check if any or all the elements in a list satisfy the predicate, respectively. Usually we use these two functions instead of mapping over a list and then doing **`and`** or **`or`**.

```
ghci> any (==4) [2,3,5,6,1,4]
True
ghci> all (>4) [6,9,10]
True
ghci> all (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
False
ghci> any (`elem` ['A'..'Z']) "HEYGUYSwhatsup"
True
```

`iterate` takes a function and a starting value. It applies the function to the starting value, then it applies that function to the result, then it applies the function to that result again, etc. It returns all the results in the form of an infinite list.

```
ghci> take 10 $ iterate (*2) 1
[1,2,4,8,16,32,64,128,256,512]
ghci> take 3 $ iterate (++ "haha") "haha"
["haha","hahahaha","hahahahahaha"]
```

`splitAt` takes a number and a list. It then splits the list at that many elements, returning the resulting two lists in a tuple.

```
ghci> splitAt 3 "heyman"
("hey","man")
ghci> splitAt 100 "heyman"
("heyman","")
ghci> splitAt (-3) "heyman"
("","heyman")
ghci> let (a,b) = splitAt 3 "foobar" in b ++ a
"barfoo"
```

`takeWhile` is a really useful little function. It takes elements from a list while the predicate holds and then when an element is encountered that doesn't satisfy the predicate, it's cut off. It turns out this is very useful.

```
ghci> takeWhile (>3) [6,5,4,3,2,1,2,3,4,5,4,3,2,1]
[6,5,4]
ghci> takeWhile (/=' ') "This is a sentence"
"This"
```

Say we wanted to know the sum of all third powers that are under 10,000. We can't map **`(^3)`** to **`[1..]`**, apply a filter and then try to sum that up because filtering an infinite list never finishes. You may know that all the elements here are ascending but Haskell doesn't. That's why we can do this:

```
ghci> sum $ takeWhile (<10000) $ map (^3) [1..]
53361
```

We apply `(^3)` to an infinite list and then once an element that's over 10,000 is encountered, the list is cut off. Now we can sum it up easily.

`dropWhile` is similar, only it drops all the elements while the predicate is true. Once predicate equates to `False`, it returns the rest of the list. An extremely useful and lovely function!

```
ghci> dropWhile (/=' ') "This is a sentence"
" is a sentence"
ghci> dropWhile (<3) [1,2,2,2,3,4,5,4,3,2,1]
[3,4,5,4,3,2,1]
```

We're given a list that represents the value of a stock by date. The list is made of tuples whose first component is the stock value, the second is the year, the third is the month and the fourth is the date. We want to know when the stock value first exceeded one thousand dollars!

```
ghci> let stock = [(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)]
ghci> head (dropWhile (\(val,y,m,d) -> val < 1000) stock)
(1001.4,2008,9,4)
```

`span` is kind of like `takeWhile`, only it returns a pair of lists. The first list contains everything the resulting list from `takeWhile` would contain if it were called with the same predicate and the same list. The second list contains the part of the list that would have been dropped.

```
ghci> let (fw, rest) = span (/=' ') "This is a sentence" in "First word:" ++ fw ++ ", the rest
"First word: This, the rest: is a sentence"
```

Whereas `span` spans the list while the predicate is true, `break` breaks it when the predicate is first true. Doing `break p` is the equivalent of doing `span (not . p)`.

```
ghci> break (==4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
ghci> span (/=4) [1,2,3,4,5,6,7]
([1,2,3],[4,5,6,7])
```

When using `break`, the second list in the result will start with the first element that satisfies the predicate.

`sort` simply sorts a list. The type of the elements in the list has to be part of the `Ord` typeclass, because if the elements of a list can't be put in some kind of order, then the list can't be sorted.

```
ghci> sort [8,5,3,2,1,6,4,2]
```

```
[1,2,2,3,4,5,6,8]
ghci> sort "This will be sorted soon"
"    Tbdeehiillnooorssstw"
```

`group` takes a list and groups adjacent elements into sublists if they are equal.

```
ghci> group [1,1,1,1,2,2,2,2,3,3,2,2,2,5,6,7]
[[1,1,1,1],[2,2,2,2],[3,3],[2,2,2],[5],[6],[7]]
```

If we sort a list before grouping it, we can find out how many times each element appears in the list.

```
ghci> 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]
[(1,4),(2,7),(3,2),(5,1),(6,1),(7,1)]
```

`inits` and `tails` are like **`init`** and **`tail`**, only they recursively apply that to a list until there's nothing left. Observe.

```
ghci> inits "w00t"
["","w","w0","w00","w00t"]
ghci> tails "w00t"
["w00t","00t","0t","t",""]
ghci> let w = "w00t" in zip (inits w) (tails w)
[("","w00t"),("w","00t"),("w0","0t"),("w00","t"),("w00t","")]
```

Let's use a fold to implement searching a list for a sublist.

```
search :: (Eq a) => [a] -> [a] -> Bool
search needle haystack =
    let nlen = length needle
    in  foldl (\acc x -
> if take nlen x == needle then True else acc) False (tails haystack)
```

First we call **`tails`** with the list in which we're searching. Then we go over each tail and see if it starts with what we're looking for.

With that, we actually just made a function that behaves like `isInfixOf`. **`isInfixOf`** searches for a sublist within a list and returns **`True`** if the sublist we're looking for is somewhere inside the target list.

```
ghci> "cat" `isInfixOf` "im a cat burglar"
True
ghci> "Cat" `isInfixOf` "im a cat burglar"
False
ghci> "cats" `isInfixOf` "im a cat burglar"
False
```

`isPrefixOf` and `isSuffixOf` search for a sublist at the beginning and at the end of a list, respectively.

```
ghci> "hey" `isPrefixOf` "hey there!"
True
ghci> "hey" `isPrefixOf` "oh hey there!"
False
ghci> "there!" `isSuffixOf` "oh hey there!"
True
ghci> "there!" `isSuffixOf` "oh hey there"
False
```

`elem` and `notElem` check if an element is or isn't inside a list.

`partition` takes a list and a predicate and returns a pair of lists. The first list in the result contains all the elements that satisfy the predicate, the second contains all the ones that don't.

```
ghci> partition (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOBMORGAN","sidneyeddy")
ghci> partition (>3) [1,3,5,6,3,2,1,0,3,7]
([5,6,7],[1,3,3,2,1,0,3])
```

It's important to understand how this is different from **`span`** and **`break`**:

```
ghci> span (`elem` ['A'..'Z']) "BOBsidneyMORGANeddy"
("BOB","sidneyMORGANeddy")
```

While **`span`** and **`break`** are done once they encounter the first element that doesn't and does satisfy the predicate, **`partition`** goes through the whole list and splits it up according to the predicate.

`find` takes a list and a predicate and returns the first element that satisfies the predicate. But it returns that element wrapped in a **`Maybe`** value. We'll be covering algebraic data types more in depth in the next chapter but for now, this is what you need to know: a **`Maybe`** value can either be **`Just something`** or **`Nothing`**. Much like a list can be either an empty list or a list with some elements, a **`Maybe`** value can be either no elements or a single element. And like the type of a list of, say, integers is **`[Int]`**, the type of maybe having an integer is **`Maybe Int`**. Anyway, let's take our **`find`** function for a spin.

```
ghci> find (>4) [1,2,3,4,5,6]
Just 5
ghci> find (>9) [1,2,3,4,5,6]
Nothing
ghci> :t find
find :: (a -> Bool) -> [a] -> Maybe a
```

Notice the type of **`find`**. Its result is **`Maybe a`**. That's kind of like having the type of **`[a]`**, only a value of the type **`Maybe`** can

contain either no elements or one element, whereas a list can contain no elements, one element or several elements.

Remember when we were searching for the first time our stock went over $1000. We did `head (dropWhile (\(val,y,m,d) -> val < 1000) stock)`. Remember that `head` is not really safe. What would happen if our stock never went over $1000? Our application of `dropWhile` would return an empty list and getting the head of an empty list would result in an error. However, if we rewrote that as `find (\(val,y,m,d) -> val > 1000) stock`, we'd be much safer. If our stock never went over $1000 (so if no element satisfied the predicate), we'd get back a `Nothing`. But there was a valid answer in that list, we'd get, say, `Just (1001.4,2008,9,4)`.

`elemIndex` is kind of like `elem`, only it doesn't return a boolean value. It maybe returns the index of the element we're looking for. If that element isn't in our list, it returns a `Nothing`.

```
ghci> :t elemIndex
elemIndex :: (Eq a) => a -> [a] -> Maybe Int
ghci> 4 `elemIndex` [1,2,3,4,5,6]
Just 3
ghci> 10 `elemIndex` [1,2,3,4,5,6]
Nothing
```

`elemIndices` is like `elemIndex`, only it returns a list of indices, in case the element we're looking for crops up in our list several times. Because we're using a list to represent the indices, we don't need a `Maybe` type, because failure can be represented as the empty list, which is very much synonymous to `Nothing`.

```
ghci> ' ' `elemIndices` "Where are the spaces?"
[5,9,13]
```

`findIndex` is like find, but it maybe returns the index of the first element that satisfies the predicate. `findIndices` returns the indices of all elements that satisfy the predicate in the form of a list.

```
ghci> findIndex (==4) [5,3,2,1,6,4]
Just 5
ghci> findIndex (==7) [5,3,2,1,6,4]
Nothing
ghci> findIndices (`elem` ['A'..'Z']) "Where Are The Caps?"
[0,6,10,14]
```

We already covered `zip` and `zipWith`. We noted that they zip together two lists, either in a tuple or with a binary function (meaning such a function that takes two parameters). But what if we want to zip together three lists? Or zip three lists with a function that takes three parameters? Well, for that, we have `zip3`, `zip4`, etc. and `zipWith3`, `zipWith4`, etc. These variants go up to 7. While this may look like a hack, it works out pretty fine, because there aren't many times when you want to zip 8 lists together. There's also a very clever way for zipping infinite numbers of lists, but we're not advanced enough to cover that just yet.

```
ghci> zipWith3 (\x y z -> x + y + z) [1,2,3] [4,5,2,2] [2,2,3]
[7,9,8]
ghci> zip4 [2,3,3] [2,2,2] [5,5,3] [2,2,2]
[(2,2,5,2),(3,2,5,2),(3,2,3,2)]
```

Just like with normal zipping, lists that are longer than the shortest list that's being zipped are cut down to size.

`lines` is a useful function when dealing with files or input from somewhere. It takes a string and returns every line of that string in a separate list.

```
ghci> lines "first line\nsecond line\nthird line"
["first line","second line","third line"]
```

`'\n'` is the character for a unix newline. Backslashes have special meaning in Haskell strings and characters.

`unlines` is the inverse function of **`lines`**. It takes a list of strings and joins them together using a **`'\n'`**.

```
ghci> unlines ["first line", "second line", "third line"]
"first line\nsecond line\nthird line\n"
```

`words` and `unwords` are for splitting a line of text into words or joining a list of words into a text. Very useful.

```
ghci> words "hey these are the words in this sentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> words "hey these           are    the words in this\nsentence"
["hey","these","are","the","words","in","this","sentence"]
ghci> unwords ["hey","there","mate"]
"hey there mate"
```

We've already mentioned `nub`. It takes a list and weeds out the duplicate elements, returning a list whose every element is a unique snowflake! The function does have a kind of strange name. It turns out that "nub" means a small lump or essential part of something. In my opinion, they should use real words for function names instead of old-people words.

```
ghci> nub [1,2,3,4,3,2,1,2,3,4,3,2,1]
[1,2,3,4]
ghci> nub "Lots of words and stuff"
"Lots fwrdanu"
```

`delete` takes an element and a list and deletes the first occurence of that element in the list.

```
ghci> delete 'h' "hey there ghang!"
"ey there ghang!"
ghci> delete 'h' . delete 'h' $ "hey there ghang!"
```

```
"ey tere ghang!"
ghci> delete 'h' . delete 'h' . delete 'h' $ "hey there ghang!"
"ey tere gang!"
```

`\\` is the list difference function. It acts like a set difference, basically. For every element in the right-hand list, it removes a matching element in the left one.

```
ghci> [1..10] \\ [2,5,9]
[1,3,4,6,7,8,10]
ghci> "Im a big baby" \\ "big"
"Im a  baby"
```

Doing **`[1..10] \\ [2,5,9]`** is like doing **`delete 2 . delete 5 . delete 9 $ [1..10]`**.

`union` also acts like a function on sets. It returns the union of two lists. It pretty much goes over every element in the second list and appends it to the first one if it isn't already in yet. Watch out though, duplicates are removed from the second list!

```
ghci> "hey man" `union` "man what's up"
"hey manwt'sup"
ghci> [1..7] `union` [5..10]
[1,2,3,4,5,6,7,8,9,10]
```

`intersect` works like set intersection. It returns only the elements that are found in both lists.

```
ghci> [1..7] `intersect` [5..10]
[5,6,7]
```

`insert` takes an element and a list of elements that can be sorted and inserts it into the last position where it's still less than or equal to the next element. In other words, **`insert`** will start at the beginning of the list and then keep going until it finds an element that's equal to or greater than the element that we're inserting and it will insert it just before the element.

```
ghci> insert 4 [3,5,1,2,8,2]
[3,4,5,1,2,8,2]
ghci> insert 4 [1,3,4,4,1]
[1,3,4,4,4,1]
```

The **`4`** is inserted right after the **`3`** and before the **`5`** in the first example and in between the **`3`** and **`4`** in the second example.

If we use **`insert`** to insert into a sorted list, the resulting list will be kept sorted.

```
ghci> insert 4 [1,2,3,5,6,7]
[1,2,3,4,5,6,7]
ghci> insert 'g' $ ['a'..'f'] ++ ['h'..'z']
```

```
"abcdefghijklmnopqrstuvwxyz"
ghci> insert 3 [1,2,4,3,2,1]
[1,2,3,4,3,2,1]
```

What **length**, **take**, **drop**, **splitAt**, **!!** and **replicate** have in common is that they take an **Int** as one of their parameters (or return an **Int**), even though they could be more generic and usable if they just took any type that's part of the **Integral** or **Num** typeclasses (depending on the functions). They do that for historical reasons. However, fixing that would probably break a lot of existing code. That's why **Data.List** has their more generic equivalents, named genericLength, genericTake, genericDrop, genericSplitAt, genericIndex and genericReplicate. For instance, **length** has a type signature of **length :: [a] -> Int**. If we try to get the average of a list of numbers by doing **let xs = [1..6] in sum xs / length xs**, we get a type error, because you can't use **/** with an **Int**. **genericLength**, on the other hand, has a type signature of **genericLength :: (Num a) => [b] -> a**. Because a **Num** can act like a floating point number, getting the average by doing **let xs = [1..6] in sum xs / genericLength xs** works out just fine.

The **nub**, **delete**, **union**, **intersect** and **group** functions all have their more general counterparts called nubBy, deleteBy, unionBy, intersectBy and groupBy. The difference between them is that the first set of functions use **==** to test for equality, whereas the *By* ones also take an equality function and then compare them by using that equality function. **group** is the same as **groupBy (==)**.

For instance, say we have a list that describes the value of a function for every second. We want to segment it into sublists based on when the value was below zero and when it went above. If we just did a normal **group**, it would just group the equal adjacent values together. But what we want is to group them by whether they are negative or not. That's where **groupBy** comes in! The equality function supplied to the *By* functions should take two elements of the same type and return **True** if it considers them equal by its standards.

```
ghci> let values = [-4.3, -2.4, -1.2, 0.4, 2.3, 5.9, 10.5, 29.1, 5.3, -2.4, -14.5, 2.9, 2.3]
ghci> groupBy (\x y -> (x > 0) == (y > 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

From this, we clearly see which sections are positive and which are negative. The equality function supplied takes two elements and then returns **True** only if they're both negative or if they're both positive. This equality function can also be written as **\x y -> (x > 0) && (y > 0) || (x <= 0) && (y <= 0)**, although I think the first way is more readable. An even clearer way to write equality functions for the *By* functions is if you import the on function from **Data.Function**. **on** is defined like this:

```
on :: (b -> b -> c) -> (a -> b) -> a -> a -> c
f `on` g = \x y -> f (g x) (g y)
```

So doing **(==) `on` (> 0)** returns an equality function that looks like **\x y -> (x > 0) == (y > 0)**. **on** is used a lot with the *By* functions because with it, we can do:

```
ghci> groupBy ((==) `on` (> 0)) values
[[-4.3,-2.4,-1.2],[0.4,2.3,5.9,10.5,29.1,5.3],[-2.4,-14.5],[2.9,2.3]]
```

Very readable indeed! You can read it out loud: Group this by equality on whether the elements are greater than zero.

Similarly, the `sort`, `insert`, `maximum` and `minimum` also have their more general equivalents. Functions like `groupBy` take a function that determines when two elements are equal. `sortBy`, `insertBy`, `maximumBy` and `minimumBy` take a function that determine if one element is greater, smaller or equal to the other. The type signature of `sortBy` is `sortBy :: (a -> a -> Ordering) -> [a] -> [a]`. If you remember from before, the `Ordering` type can have a value of `LT`, `EQ` or `GT`. `sort` is the equivalent of `sortBy compare`, because compare just takes two elements whose type is in the `Ord` typeclass and returns their ordering relationship.

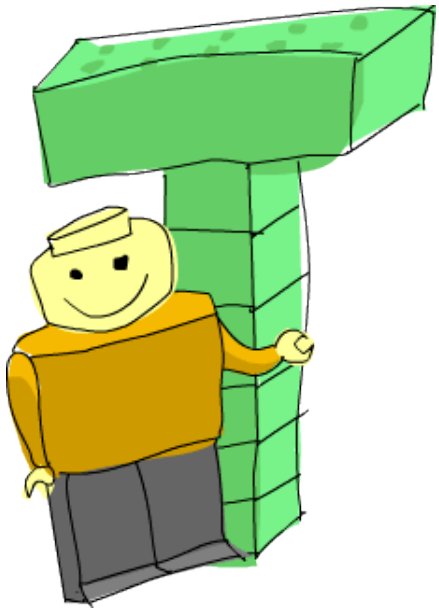
Lists can be compared, but when they are, they are compared lexicographically. What if we have a list of lists and we want to sort it not based on the inner lists' contents but on their lengths? Well, as you've probably guessed, we'll use the `sortBy` function.

```
ghci> let xs = [[5,4,5,4,4],[1,2,3],[3,5,4,3],[],[2],[2,2]]
ghci> sortBy (compare `on` length) xs
[[],[2],[2,2],[1,2,3],[3,5,4,3],[5,4,5,4,4]]
```

Awesome! `compare `on` length` ... man, that reads almost like real English! If you're not sure how exactly the `on` works here, `compare `on` length` is the equivalent of `\x y -> length x `compare` length y`. When you're dealing with *By* functions that take an equality function, you usually do `(==) `on` something` and when you're dealing with *By* functions that take an ordering function, you usually do `compare `on` something`.

Data.Char

The `Data.Char` module does what its name suggests. It exports functions that deal with characters. It's also helpful when filtering and mapping over strings because they're just lists of characters.



`Data.Char` exports a bunch of predicates over characters. That is, functions that take a character and tell us whether some assumption about it is true or false. Here's what they are:

`isControl` checks whether a character is a control character.

`isSpace` checks whether a character is a white-space characters. That includes spaces, tab characters, newlines, etc.

`isLower` checks whether a character is lower-cased.