The type `[T]` denotes an unbounded sequence of values of type `T`

Suppose you have a list

```
oops = [1, 2, 'c']
```

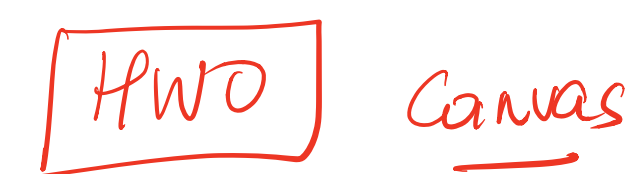
There is no `T` that we can use

- As last element is not `Int`
- First two elements are not `Char` !

Result: Mysterious Type Error!

Constructing Lists

There are two ways to construct lists



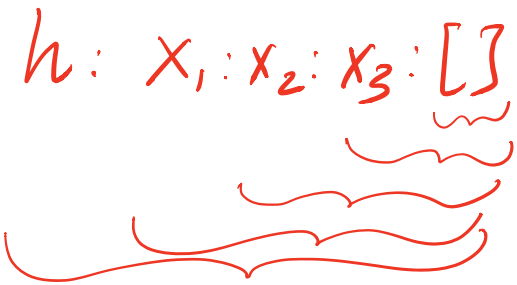
```
[]     -- creates an empty list
h:t    -- creates a list with "head" 'h' and "tail" t
```

For example

```
>>> 3 : []
[3]

>>> 2 : (3 : [])
[2, 3]

>>> 1 : (2 : (3 : []))
[1, 2, 3]
```



Cons Operator : is Right Associative

`x1 : x2 : x3 : x4 : t` means `x1 : (x2 : (x3 : (x4 : t)))`

So we can just avoid the parentheses.

Syntactic Sugar

Haskell lets you write `[x1, x2, x3, x4]` instead of `x1 : x2 : x3 : x4 : []`

Functions Producing Lists

Lets write a function `copy3` that

- takes an input `x` and
- returns a list with *three* copies of `x`

```
copy3 :: ???
copy3 x = ???
```

When you are done, you should see the following

```
>>> copy3 5
[5, 5, 5]

>>> copy3 "cat"
["cat", "cat", "cat"]
```

PRACTICE: Clone

Write a function `clone` such that `clone n x` returns a list with `n` copies of `x`.

```
clone :: ???
clone n x = ???
```

When you are done you should see the following behavior

```
>>> clone 0 "cat"
[]

>>> clone 1 "cat"
["cat"]

>>> clone 2 "cat"
["cat", "cat"]

>>> clone 3 "cat"
["cat", "cat", "cat"]

>>> clone 3 100
[100, 100, 100]
```

How does `clone` *execute?*

(Substituting equals-by-equals!)

```
clone 3 100
  =*> ???
```

clone 3 "cat"

=> "cat" : (clone 2 "cat")

=> "cat" : ("cat" : clone 1 "cat")

"cat" : "cat" : "cat" : clone 0

||

[]

EXERCISE: Range

Write a function `range` such that `range i j` returns the list of values `[i, i+1, ..., j]`

```
range :: ???
range i j = ???
```

When we are done you should get the behavior

```
>>> range 3 3
[]

>>> range 2 3
[2]

>>> range 1 3
[1, 2]

>>> range 0 3
[1, 2, 3]
```

Functions Consuming Lists

So far: how to *produce* lists.

Next how to *consume* lists!

Example

Lets write a function `firstElem` such that `firstElem xs` returns the *first* element `xs` if it is a non-empty list, and `0` otherwise.

```
firstElem :: [Int] -> Int
firstElem xs = ???
```

When you are done you should see the following behavior:

```
>>> firstElem []
0

>>> firstElem [10, 20, 30]
10

>>> firstElem [5, 6, 7, 8]
5
```