Lists

Rob Hackman

Fall 2025

University of Alberta

Table of Contents

Aggregate Data

So far we've worked only with four types, which have been

- Integers (`int`)

Aggregate Data

So far we've worked only with four types, which have been

- Integers (`int`)
- Floating Points (`float`)

Aggregate Data

So far we've worked only with four types, which have been

- Integers (`int`)
- Floating Points (`float`)
- Strings (`str`)

Aggregate Data

So far we've worked only with four types, which have been

- Integers (`int`)
- Floating Points (`float`)
- Strings (`str`)
- Booleans (`bool`)

Aggregate Data

So far we've worked only with four types, which have been

- Integers (`int`)
- Floating Points (`float`)
- Strings (`str`)
- Booleans (`bool`)

We now will introduce a new data type, that will allow us to group together multiple pieces of data into one entity. When we combine multiple pieces of data into one, we call the collection type an *aggregate data type*.

Abstract Data Types

The data type we are about to introduce is an *abstract data type* (ADT).

The data type we are about to introduce is an *abstract data type* (ADT).

Abstract data types are immensely useful for computer science, they abstract away the concrete details of how data is exactly stored and instead leave the user to only concern themselves with the *behaviour* they support.

Abstract Data Types

The data type we are about to introduce is an *abstract data type* (ADT).

Abstract data types are immensely useful for computer science, they abstract away the concrete details of how data is exactly stored and instead leave the user to only concern themselves with the *behaviour* they support.

In this way, an ADT is defined by the behaviour it provides, and the functions in which the *client programmer* uses to interact with it.

The `LList` ADT

We will call our new data type an `LList` — the definition for which is provided in our `cmput274` module.

In order to use an `LList` we only need to understand the provided functions with which we can interact with the data type.

An `LList` is an aggregate data type that can store zero or more elements, and whose elements can be any type at all.

Displaying `LLists`

In order to discuss `LLists` we must be able to depict them. We will represent an `LList` as $(a_1, a_2, ..., a_n)$ where a_1 through a_n are the elements of the `LList`.

Displaying LLists

In order to discuss LLists we must be able to depict them. We will represent an LList as $(a_1, a_2, ..., a_n)$ where a_1 through a_n are the elements of the LList.

The LList that stores zero elements is denoted as () which we will call the *empty LList* or simply empty.

Table of Contents

Creating an LList

We only have one way to create a new list LList, and that is with the function empty.

```
def empty():
    '''
    empty produces an empty LList, for constructing.

    returns - An empty LList

    Examples:
      empty() -> ()
    '''
```

empty will be the start of every LList we create.

constructing an LList

New LLists are constructed from existing LLists by prepending an individual element to an existing LList. This can be done using the cons function.

```
def cons(elem, l):
  '''
  cons returns an LList constructed by prepending
       a given element to the given LList

  elem    - any, an item to be placed in an LList
  l       - an LList
  returns - an LList

  Examples:
    cons(1, empty()) -> (1)
    cons(1, cons(2, empty())) -> (1, 2)
  '''
```

A data definition for `LLists`

With `empty` defined and the procedure for constructing a new larger list from an existing list, we are able to write a recursive data definition for our `LList` data type.

An `LList` is one of:

A data definition for `LLists`

With `empty` defined and the procedure for constructing a new larger list from an existing list, we are able to write a recursive data definition for our `LList` data type.

An `LList` is one of:

- The empty list

A data definition for `LLists`

With `empty` defined and the procedure for constructing a new larger list from an existing list, we are able to write a recursive data definition for our `LList` data type.

An `LList` is one of:

- The empty list
- cons(elem, `LList`), where `elem` can be anything.

Knowledge Check: Write the function `descendList` that takes a natural number `n` and produces and `LList` that stores the numbers from `n` to zero in descending order.

Working with `LLists`

In order to work with values inside an `LList` we need a way to pull out individual values.

In order to view the value that appears first in a non-empty `LList` we can use the function `first`

The `first` function

```
def first(l):
  '''
  first returns the first element in a
       non-empty LList

  l       - a non-empty LList
  returns - the first element of l

  Examples:
    first(cons(1, cons(2, cons(3, empty())))) -> 1
    first(cons(cons(1, empty()),
                empty())) -> (1)
  '''
```

Accessing `LLists`

What if you want to access more than the first element of an `LList`?

What if you want to access more than the first element of an `LList`?

The function `rest` takes a non-empty `LList` and produces an `LList` which is everything in the provided `LList` except for its first element.

The `rest` function

```
def rest(l):
    '''
    rest returns an LList that represents all
         elements in a non-empty LList without
         its first element.

    l       - a non-empty LList
    returns - an LList

    Examples:
      rest(cons(1, cons(2, cons(3, empty()))))
        -> (2, 3)
    '''
```

Accessing elements

Knowledge Check: Assume the identifier `myL` is bound to an `LList` with ten things in it.

- Write an expression that evaluates to the second item stored in `myL`

Accessing elements

Knowledge Check: Assume the identifier `myL` is bound to an `LList` with ten things in it.

- Write an expression that evaluates to the second item stored in `myL`
- Write an expression that evaluates to the third item stored in `myL`

Accessing elements

Knowledge Check: Assume the identifier `myL` is bound to an `LList` with ten things in it.

- Write an expression that evaluates to the second item stored in `myL`
- Write an expression that evaluates to the third item stored in `myL`
- Write an expression that evaluates to the seventh item stored in `myL`

Accessing arbitrary elements

Knowledge Check: Write a function `getIth` that has an `LList` parameter and a natural number parameter `i`. The function should return the `ith` element of the given `LList`.

Note — we will use zero-indexing as we do with strings. As such, if `i` is zero it means the first element of the list should be produced, if `i` is one it means the second item of the list should be produced, etc.

You may assume that the given `LList` has at least `i+1` items within it.

One last `LList` function

Our last function for working with `LLists` is the function `isEmpty` which takes a single `LList` parameter and returns `True` if that `LList` is empty, and `False` otherwise.

With these five functions we can do anything with `LLists` that we need to do!

Table of Contents

Scheduling time

A person hires you to write them a program, they say that what they need is this:

- They will provide you with a list of pairs of times that a person leaves their house and then returns, this time will be written in strings of the form `"HH:MM"` in 24 hour fashion.

Scheduling time

A person hires you to write them a program, they say that what they need is this:

- They will provide you with a list of pairs of times that a person leaves their house and then returns, this time will be written in strings of the form `"HH:MM"` in 24 hour fashion.
  - For example, they see the person leave the house at 1:00AM and return at 8:30AM, leave again at 11:45AM and return at 2:37PM then they would provide you with a list that looked like this `("01:00", "08:30", "11:45", "14:37")`.

Scheduling time

A person hires you to write them a program, they say that what they need is this:

- They will provide you with a list of pairs of times that a person leaves their house and then returns, this time will be written in strings of the form `"HH:MM"` in 24 hour fashion.
    - For example, they see the person leave the house at 1:00AM and return at 8:30AM, leave again at 11:45AM and return at 2:37PM then they would provide you with a list that looked like this `("01:00", "08:30", "11:45", "14:37")`.
- Given this list they provide you, and a number of minutes that they need the house unoccupied so that they can set up a "surprise party" without the person coming home and catching them, they want you to return a time they can enter the house safely.

Many problems in one

This problem actually has several problems for us to solve in one. We need to first identify these sub-problems to be able to get started on the solution to our complete problem.

Many problems in one

This problem actually has several problems for us to solve in one. We need to first identify these sub-problems to be able to get started on the solution to our complete problem.

First, we consider what our problems really is at its core, to do this we think about the data we have and what we need to do with it.

Many problems in one

This problem actually has several problems for us to solve in one. We need to first identify these sub-problems to be able to get started on the solution to our complete problem.

First, we consider what our problems really is at its core, to do this we think about the data we have and what we need to do with it. What we have, is effectively a list of the form $(e_1, r_1, e_2, r_2, ..., e_n, r_n)$ where e_i indicates time i the individual leaves the house, and r_i indicates time i the individual returns home.

Many problems in one

This problem actually has several problems for us to solve in one. We need to first identify these sub-problems to be able to get started on the solution to our complete problem.

First, we consider what our problems really is at its core, to do this we think about the data we have and what we need to do with it. What we have, is effectively a list of the form $(e_1, r_1, e_2, r_2, ..., e_n, r_n)$ where e_i indicates time i the individual leaves the house, and r_i indicates time i the individual returns home.
Given this breakdown of our data, what numbers are we actually interested in?

Many problems in one

Given this breakdown of our data, what numbers are we actually interested in?

We care about the values $r_i - e_i$ as that tells us how long the individual was away for.

Translating time strings to numbers

The time we are given is formatted strings, stored in the form `"HH:MM"`. This means before we can calculate how many minutes the individual is away for, we must translate these strings into usable numbers.

Translating time strings to numbers

The time we are given is formatted strings, stored in the form `"HH:MM"`. This means before we can calculate how many minutes the individual is away for, we must translate these strings into usable numbers.

Observation: Having different units (hours and minutes) complicates things, in doing our translation perhaps we should speak in terms of only one unit?

Translating time strings to numbers

The time we are given is formatted strings, stored in the form `"HH:MM"`. This means before we can calculate how many minutes the individual is away for, we must translate these strings into usable numbers.

Observation: Having different units (hours and minutes) complicates things, in doing our translation perhaps we should speak in terms of only one unit?

Solution: Translate times into relevant minutes.

Translating from a digit-character to a digit

In Python `"6"` and `6` are very different things. How can we transform one into the other?

We need some understanding of how the characters of our strings are actually stored in our computers.

The ASCII Table

All data stored on our computer is ultimately stored in *binary*, which is to say all data is stored as a series of ones and zeros[1].

[1] Actually, we just choose to talk about them as ones and zeroes, their physical representation is typically electric charges or magnetic polarities.

The ASCII Table

All data stored on our computer is ultimately stored in *binary*, which is to say all data is stored as a series of ones and zeros[1].

We have a system for mapping ones and zeroes to the numbers we are familiar with (base-10). As such, we can map these values to numbers, we can also map numbers to characters.

[1] Actually, we just choose to talk about them as ones and zeroes, their physical representation is typically electric charges or magnetic polarities.

The ASCII Table

All data stored on our computer is ultimately stored in *binary*, which is to say all data is stored as a series of ones and zeros[1].

We have a system for mapping ones and zeroes to the numbers we are familiar with (base-10). As such, we can map these values to numbers, we can also map numbers to characters.

So, in order to store characters in our computers we have an agreed upon standard mapping between numbers and characters. One such standard is the American Standard Code for Information Interchange (ASCII)

[1] Actually, we just choose to talk about them as ones and zeroes, their physical representation is typically electric charges or magnetic polarities.

The ASCII Table

Decimal	Char	Decimal	Char	Decimal	Char	Decimal	Char	Decimal	Char
32		52	4	72	H	92	\	112	p
33	!	53	5	73	I	93	]	113	q
34	"	54	6	74	J	94	^	114	r
35	#	55	7	75	K	95	_	115	s
36	$	56	8	76	L	96	`	116	t
37	%	57	9	77	M	97	a	117	u
38	&	58	:	78	N	98	b	118	v
39	'	59	;	79	O	99	c	119	w
40	(	60	<	80	P	100	d	120	x
41	)	61	=	81	Q	101	e	121	y
42	*	62	>	82	R	102	f	122	z
43	+	63	?	83	S	103	g	123	{
44	,	64	@	84	T	104	h	124	|
45	-	65	A	85	U	105	i	125	}
46	.	66	B	86	V	106	j	126	~
47	/	67	C	87	W	107	k	127	
48	0	68	D	88	X	108	l		
49	1	69	E	89	Y	109	m		
50	2	70	F	90	Z	110	n		
51	3	71	G	91	[	111	o		

The ASCII Table

Decimal	Char	Decimal	Char	Decimal	Char	Decimal	Char	Decimal	Char
32		52	4	72	H	92	\	112	p
33	!	53	5	73	I	93	]	113	q
34	"	54	6	74	J	94	^	114	r
35	#	55	7	75	K	95	_	115	s
36	$	56	8	76	L	96	`	116	t
37	%	57	9	77	M	97	a	117	u
38	&	58	:	78	N	98	b	118	v
39	'	59	;	79	O	99	c	119	w
40	(	60	<	80	P	100	d	120	x
41	)	61	=	81	Q	101	e	121	y
42	*	62	>	82	R	102	f	122	z
43	+	63	?	83	S	103	g	123	{
44	,	64	@	84	T	104	h	124	|
45	-	65	A	85	U	105	i	125	}
46	.	66	B	86	V	106	j	126	~
47	/	67	C	87	W	107	k	127	
48	0	68	D	88	X	108	l		
49	1	69	E	89	Y	109	m		
50	2	70	F	90	Z	110	n		
51	3	71	G	91	[	111	o		

Observation: The digit characters are all located sequentially. i.e. 48 $\longrightarrow$ ‘0’, 49 $\longrightarrow$ ‘1’, ..., 57 $\longrightarrow$ ‘9’

The `ord` function

If we can access the value in the ASCII table that corresponds to a character, its *ordinal* value, then we can translate from a digit character into an integer.

The `ord` function

If we can access the value in the ASCII table that corresponds to a character, its *ordinal* value, then we can translate from a digit character into an integer.

The `ord` function in Python does exactly this. It expects a single character string, and returns the `ASCII` value of that character.

Digit to Number

We now write the function `digitToNumber` which takes a single character string that contains a digit character, and returns the `int` that is that digit.

```
def digitToNumber(c):
   '''
   digitToNumber converts a single character string that
                stores a digit into an equivalent int

   c       - a single digit character string
   returns - int

   Examples:
     digitToNumber("9") -> 9
     digitToNubmer("3") -> 3
   '''
   return ord(c) - ord("0")
```

Converting a multiple digit string into an integer

Our time strings can have numbers that are two digits. We can write a function to easily solve this.

Converting a multiple digit string into an integer

```
def strToNum(s):
  '''
  strToNum converts a two character string that
          stores only digits into an equivalent int

  s       - a two digit character string
  returns - int

  Examples:
    digitToNumber("93") -> 93
    digitToNubmer("05") -> 5
  '''
  return digitToNumber(s[0])*10 + digitToNumber(s[1])
```

Converting a time into minutes

Now we can take a time string and convert it into a number of minutes since midnight.

```
def timeToMins(s):
  '''
  timeToMins converts a time string in format HH:MM
             to an integer number of minute that
             have elapsed since 00:00

  s       - a time string
  returns - int

  Examples:
    digitToNumber("03:20") -> 200
    digitToNubmer("23:59") -> 1439
  '''
  return strToNum(s[0:2])*60 + strToNum(s[3:5])
```

Walking our list

Now that we have a function to convert our times into minutes since midnight, we can now write our final function to find when the party planner can enter the house.

So we will write the function `timeToEnter` which will return the first time string found when it is safe to enter, or `False` if there is never a duration long enough.

First, what is our base case?

```
def timeToEnter(lot, dur):
```

Given our base case of our time lists being empty, what is our answer?

```
def timeToEnter(lot, dur):
  if isEmpty(lot):
    return False
```

When our list isn't empty we look at our current exit and return time. How do we grab these values out?

```
def timeToEnter(lot, dur):
  if timeToMins(???) - timeToMins(???) >= dur:
    return ???
  return timeToEnter(???, dur)
```

If our duration is long enough, what value do we return?

```
def timeToEnter(lot, dur):
  exit = first(lot)
  enter = first(rest(lot))
  if timeToMins(enter) - timeToMins(exit) >= dur:
    return ???
  return timeToEnter(???, dur)
```

How do we step closer to the base case of our recursion?

```
def timeToEnter(lot, dur):
  exit = first(lot)
  enter = first(rest(lot))
  if timeToMins(enter) - timeToMins(exit) >= dur:
    return exit
  return timeToEnter(???, dur)
```

How do we step closer to the base case of our recursion?

```
def timeToEnter(lot, dur):
  exit = first(lot)
  enter = first(rest(lot))
  if timeToMins(enter) - timeToMins(exit) >= dur:
    return first(lot)
  return timeToEnter(rest(rest(lot)), dur)
```

Solving problems

When solving problems it will often be the case that we'll need to break it down into smaller sub-problems that we can solve with helper problems.

Solving problems

When solving problems it will often be the case that we'll need to break it down into smaller sub-problems that we can solve with helper problems.

We wrote three helper functions just to translate our time strings into a number of minutes that was usable for finding the durations that the individual was out of the house!

Solving problems

When solving problems it will often be the case that we'll need to break it down into smaller sub-problems that we can solve with helper problems.

We wrote three helper functions just to translate our time strings into a number of minutes that was usable for finding the durations that the individual was out of the house!

We could have "combine" some of those functions into one larger function. However, keeping functions small and single purpose makes them more reusable. We can always compose several small functions together, we can't as easily decompose a larger function that performs multiple tasks within.

Practice Problems

Practice Problem 1: Write a function `mean` that takes a `LList` of numbers and returns the *mean* of that list of numbers.

Practice Problem 2: Write a function `slice` that takes an `LList`, and integer start, and an integer end, and returns a new `LList` which contains the elements from the given `LList` in the range of indices $[start, end)$.