CS 61A
Summer 2025

SQL, Final Review
Discussion 12: August 7, 2025

Final Review

Lists

The two most common mutation operations for lists are item assignment and the `append` method.

```
>>> s = [1, 3, 4]
>>> t = s  # A second name for the same list
>>> t[0] = 2  # this changes the first element of the list to 2, affecting both s and t
>>> s
[2, 3, 4]
>>> s.append(5)  # this adds 5 to the end of the list, affecting both s and t
>>> t
[2, 3, 4, 5]
```

There are many other list mutation methods:

- `append(elem)`: Add `elem` to the end of the list. Return `None`.
- `extend(s)`: Add all elements of iterable `s` to the end of the list. Return `None`.
- `insert(i, elem)`: Insert `elem` at index `i`. If `i` is greater than or equal to the length of the list, then `elem` is inserted at the end. This does not replace any existing elements, but only adds the new element `elem`. Return `None`.
- `remove(elem)`: Remove the first occurrence of `elem` in list. Return `None`. Errors if `elem` is not in the list.
- `pop(i)`: Remove and return the element at index `i`.
- `pop()`: Remove and return the last element.

Q1: Word Rope

Definition: A *rope* in Python is a list containing only one-letter strings except for the last element, which may either be a one-letter string or a rope.

Implement `word_rope`, a Python function that takes a non-empty string `s` containing only letters and spaces that does not start or end with a space. It returns a *rope* containing the letters of `s` in which each word is in a separate list.

Important: You may not use slicing or the `split`, `find`, or `index` methods of a string. Solve the problem using list operations.

Reminder: `s[-1]` evaluates to the last element of a sequence `s`.

```
def word_rope(s):
    """Return a rope of the words in string s.

    >>> word_rope('the ref was wrong')
    ['t', 'h', 'e', ['r', 'e', 'f', ['w', 'a', 's', ['w', 'r', 'o', 'n', 'g']]]]
    """
    assert s and s[0] != ' ' and s[-1] != ' '
    result = []
    word = result
    for x in s:
        if x == ' ':
            word.append([])
            word = word[-1]
        else:
            word.append(x)
    return result
```

Linked Lists

A linked list is a `Link` object or `Link.empty`.

You can mutate a `Link` object `s` in two ways: - Change the first element with `s.first = ...` - Change the rest of the elements with `s.rest = ...`

You can make a new `Link` object by calling `Link`: - `Link(4)` makes a linked list of length 1 containing 4. - `Link(4, s)` makes a linked list that starts with 4 followed by the elements of linked list `s`.

```
class Link:
    """A linked list is either a Link object or Link.empty

    >>> s = Link(3, Link(4, Link(5)))
    >>> s.rest
    Link(4, Link(5))
    >>> s.rest.rest.rest is Link.empty
    True
    >>> s.rest.first * 2
    8
    >>> print(s)
    <3 4 5>
    """
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```

Q2: Linear Sublists

Definition: A *sublist* of linked list `s` is a linked list of some of the elements of `s` in order. For example, `<3 6 2 5 1 7>` has sublists `<3 2 1>` and `<6 2 7>` but not `<5 6 7>`. A *linear sublist* of a linked list of numbers `s` is a sublist in which the difference between adjacent numbers is always the same. For example `<2 4 6 8>` is a linear sublist of `<1 2 3 4 6 9 1 8 5>` because the difference between each pair of adjacent elements is 2.

Implement `linear` which takes a linked list of numbers `s` (either a `Link` instance or `Link.empty`). It returns the longest linear sublist of `s`. If two linear sublists are tied for the longest, return either one.

```
def linear(s):
    """Return the longest linear sublist of a linked list s.

    >>> s = Link(9, Link(4, Link(6, Link(7, Link(8, Link(10))))))
    >>> linear(s)
    Link(4, Link(6, Link(8, Link(10))))
    >>> linear(Link(4, Link(5, s)))
    Link(4, Link(5, Link(6, Link(7, Link(8)))))
    >>> linear(Link(4, Link(5, Link(4, Link(7, Link(3, Link(2, Link(8))))))))
    Link(5, Link(4, Link(3, Link(2))))
    """
    def complete(first, rest):
        "The longest linear sublist of Link(first, rest) with difference d."
        if rest is Link.empty:
            return Link(first, rest)
        elif rest.first - first == d:
            return Link(first, complete(rest.first, rest.rest))
        else:
            return complete(first, rest.rest)
    if s is Link.empty:
        return s
    longest = Link(s.first) # The longest linear sublist found so far
    while s is not Link.empty:
        t = s.rest
        while t is not Link.empty:
            d = t.first - s.first
            candidate = Link(s.first, complete(t.first, t.rest))
            if length(candidate) > length(longest):
                longest = candidate
            t = t.rest
        s = s.rest
    return longest

def length(s):
    if s is Link.empty:
        return 0
    else:
        return 1 + length(s.rest)
```

Scheme

> As you read through this section, it may be difficult to understand the differences between the various representations of Scheme containers. We recommend that you use our online Scheme interpreter to see the box-and-pointer diagrams of pairs and lists that you're having a hard time visualizing! (Use the command `(autodraw)` to toggle the automatic drawing of diagrams.)

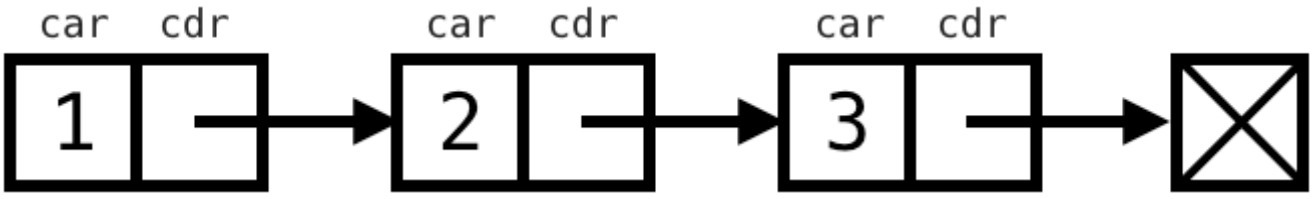
Lists

Scheme lists are very similar to the linked lists we've been working with in Python. Just like how a linked list is constructed of a series of `Link` objects, a Scheme list is constructed with a series of pairs, which are created with the constructor `cons`.

Scheme lists require that the `cdr` is either another list or `nil`, an empty list. A list is displayed in the interpreter as a sequence of values (similar to the `__str__` representation of a `Link` object). For example,

```
scm> (cons 1 (cons 2 (cons 3 nil)))
(1 2 3)
```

Here, we've ensured that the second argument of each `cons` expression is another `cons` expression or `nil`.



list

We can retrieve values from our list with the `car` and `cdr` procedures, which now work similarly to the Python `Link`'s `first` and `rest` attributes. (Curious about where these weird names come from? Check out their etymology.)

```
scm> (define a (cons 1 (cons 2 (cons 3 nil))))  ; Assign the list to the name a
a
scm> a
(1 2 3)
scm> (car a)
1
scm> (cdr a)
(2 3)
scm> (car (cdr (cdr a)))
3
```

If you do not pass in a pair or nil as the second argument to `cons`, it will error:

```
scm> (cons 1 2)
Error
```

`list` Procedure

There are a few other ways to create lists. The `list` procedure takes in an arbitrary number of arguments and constructs a list with the values of these arguments:

```
scm> (list 1 2 3)
(1 2 3)
scm> (list 1 (list 2 3) 4)
(1 (2 3) 4)
scm> (list (cons 1 (cons 2 nil)) 3 4)
((1 2) 3 4)
```

Note that all of the operands in this expression are evaluated before being put into the resulting list.

Quote Form

We can also use the quote form to create a list, which will construct the exact list that is given. Unlike with the `list` procedure, the argument to `'` is *not* evaluated.

```
scm> '(1 2 3)
(1 2 3)
scm> '(cons 1 2)           ; Argument to quote is not evaluated
(cons 1 2)
scm> '(1 (2 3 4))
(1 (2 3 4))
```

Built-In Procedures for Lists

There are a few other built-in procedures in Scheme that are used for lists. Try them out in the interpreter!

```
scm> (null? nil)               ; Checks if a value is the empty list
True
scm> (append '(1 2 3) '(4 5 6)) ; Concatenates two lists
(1 2 3 4 5 6)
scm> (length '(1 2 3 4 5))     ; Returns the number of elements in a list
5
```

Q3: Increasing Rope

Definition: A *rope* in Scheme is a non-empty list containing only numbers except for the last element, which may either be a number or a rope.

Implement `up`, a Scheme procedure that takes a positive integer `n`. It returns a rope containing the digits of `n` that is the shortest rope in which each pair of adjacent numbers in the same list are in increasing order.

Reminder: the `quotient` procedure performs floor division, like `//` in Python. The `remainder` procedure is like `%` in Python.

```
(define (up n)
    (define (helper n result)
        (if (zero? n) result
            (helper
                (quotient n 10)
                (let ((first (remainder n 10)))
                    (if (< first (car result))
                        (cons first result)
                        (list first result))
                    ))))
    (helper
      (quotient n 10)
      (list (remainder n 10))
    ))

(expect (up 314152667899) (3 (1 4 (1 5 (2 6 (6 7 8 9 (9)))))))
```

SQL

A `SELECT` statement describes an output table based on input rows. To write one: 1. Describe the **input rows** using `FROM` and `WHERE` clauses. 2. **Group** those rows and determine which groups should appear as output rows using `GROUP BY` and `HAVING` clauses. 3. Format and order the **output rows** and columns using `SELECT` and `ORDER BY` clauses.

`SELECT` *(Step 3)* `FROM` *(Step 1)* `WHERE` *(Step 1)* `GROUP BY` *(Step 2)* `HAVING` *(Step 2)* `ORDER BY` *(Step 3)*;

Step 1 may involve joining tables (using commas) to form input rows that consist of two or more rows from existing tables.

The `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY` clauses are optional.

Visualizing SQL

The CS61A SQL Web Interpreter is a great tool for visualizing and debugging SQL statements!

To get started, visit code.cs61a.org and hit `Start SQL interpreter` on the launch screen.

Most tables used in assignments are already available for use, so let's try to execute a `SELECT` statement:

In addition to displaying a visual representation of the output table, the "Step-by-step" button lets us step through the SQL execution and visualize every transformation that takes place. For our example, clicking on the next arrow will produce the following visuals, demonstrating exactly how SQL is grouping our rows to form the final output!

After you finish your Thanksgiving dinner, you realize that you still need to buy gifts for all your loved ones over the holidays. However, you also want to spend as little money as possible (you're not cheap, just looking for a great sale).

This question utilizes the following tables:

products

```
CREATE TABLE products AS
  SELECT 'phone' AS category, 'uPhone' AS name, 99.99 AS MSRP, 4.5 AS rating UNION
  SELECT 'phone'            , 'rPhone'        , 79.99        , 3             UNION
  SELECT 'phone'            , 'qPhone'        , 89.99        , 4             UNION
  SELECT 'games'            , 'GameStation'   , 299.99       , 3             UNION
  SELECT 'games'            , 'QBox'          , 399.99       , 3.5           UNION
  SELECT 'computer'         , 'iBook'         , 112.99       , 4             UNION
  SELECT 'computer'         , 'wBook'         , 114.29       , 4.4           UNION
  SELECT 'computer'         , 'kBook'         , 99.99        , 3.8                ;
```

inventory

```
CREATE TABLE inventory AS
  SELECT 'Hallmart' AS store, 'uPhone' AS item, 99.99 AS price UNION
  SELECT 'Targive'          , 'uPhone'        , 100.99         UNION
  SELECT 'RestBuy'          , 'uPhone'        , 89.99          UNION

  SELECT 'Hallmart'         , 'rPhone'        , 69.99          UNION
  SELECT 'Targive'          , 'rPhone'        , 79.99          UNION
  SELECT 'RestBuy'          , 'rPhone'        , 75.99          UNION

  SELECT 'Hallmart'         , 'qPhone'        , 85.99          UNION
  SELECT 'Targive'          , 'qPhone'        , 88.98          UNION
  SELECT 'RestBuy'          , 'qPhone'        , 87.98          UNION

  SELECT 'Hallmart'         , 'GameStation'   , 298.98         UNION
  SELECT 'Targive'          , 'GameStation'   , 300.98         UNION
  SELECT 'RestBuy'          , 'GameStation'   , 310.99         UNION

  SELECT 'Hallmart'         , 'QBox'          , 399.99         UNION
  SELECT 'Targive'          , 'QBox'          , 390.98         UNION
  SELECT 'RestBuy'          , 'QBox'          , 410.98         UNION

  SELECT 'Hallmart'         , 'iBook'         , 111.99         UNION
  SELECT 'Targive'          , 'iBook'         , 110.99         UNION
  SELECT 'RestBuy'          , 'iBook'         , 112.99         UNION

  SELECT 'Hallmart'         , 'wBook'         , 117.29         UNION
  SELECT 'Targive'          , 'wBook'         , 119.29         UNION
  SELECT 'RestBuy'          , 'wBook'         , 114.29         UNION

  SELECT 'Hallmart'         , 'kBook'         , 95.99          UNION
  SELECT 'Targive'          , 'kBook'         , 96.99          UNION
  SELECT 'RestBuy'          , 'kBook'         , 94.99               ;
```

stores

```
CREATE TABLE stores AS
  SELECT 'Hallmart' AS store, '50 Lawton Way' AS address, 25 AS Mbs UNION
  SELECT 'Targive'          , '2 Red Circle Way'        , 40        UNION
  SELECT 'RestBuy'          , '1 Kiosk Ave'             , 30            ;
```

Q4: Price Check

Let's start off by surveying our options. Using the `products` table, write a query that creates a table `average_prices` that lists categories and the average price of items in the category (using MSRP as the price). Finally, sort the resulting rows by highest to lowest average price.

You should get the following output:

category	average_price
games	350.0
computer	109.0
phone	90.0

> Due to floating point errors, you may get average price values that are slightly off, such as 349.99 instead of 350.

```
CREATE TABLE average_prices AS
  SELECT category, AVG(msrp) AS average_price -- alternatively: SELECT category, SUM(msrp
    ) / COUNT(*) AS average_price
  FROM products
  GROUP BY category
  ORDER BY average_price DESC;
```

Q5: Lowest Prices

Now, you want to figure out which stores sell each item in products for the lowest price. Write a SQL query that uses the `inventory` table to create a table `lowest_prices` that lists items, the stores that sells that item for the lowest price, and the price that the store sells that item for. Finally, sort the resulting rows alphabetically by item name.

You should expect the following output:

store	item	lowest_price
Hallmart	GameStation	298.98
Targive	QBox	390.98
Targive	iBook	110.99
RestBuy	kBook	94.99
Hallmart	qPhone	85.99
Hallmart	rPhone	69.99
RestBuy	uPhone	89.99
RestBuy	wBook	114.29

> In other variants of SQL such as PostgreSQL, you are not allowed to include columns in the `SELECT` clause if they are not included in the `GROUP BY` clause or an aggregation function in the `SELECT` clause. However, SQLite allows this and this variant is what we use in this class.

```
CREATE TABLE lowest_prices AS
  SELECT store, item, MIN(price) AS lowest_price
  FROM inventory
  GROUP BY item
  ORDER BY item;
```

Q6: Shopping List

You want to make a shopping list by choosing the item that is the best deal possible for every category. For example, for the "phone" category, the uPhone is the best deal because the MSRP price of a uPhone divided by its ratings yields the lowest cost. That means that uPhones cost the lowest money per rating point out of all of the phones. Note that the item with the lowest MSRP price may not necessarily be the best deal.

Write a query to create a table `shopping_list` that lists the items that you want to buy from each category.

After you've figured out which item you want to buy for each category, add another column that lists the store that sells that item for the lowest price.

Finally, sort the resulting table alphabetically by item name.

> Hint: What table have you already defined that gives you the store that sells a given item for the lowest price? How can you use that to determine which item in each category is the best deal?

You should expect the following output:

item	store
GameStation	Hallmart
uPhone	RestBuy
wBook	RestBuy

> **Note 1:** In other variants of SQL such as PostgreSQL, you are not allowed to include columns in the `SELECT` clause if they are not included in the `GROUP BY` clause or an aggregation function in the `SELECT` clause. However, SQLite allows this and this variant is what we use in this class.
>
> **Note 2:** In other variants of SQL such as PostgreSQL, you cannot provide a numeric type as the predicate of the `HAVING` clause (the staff solution uses `HAVING MIN(...)` which isn't allowed since `MIN` returns a number but `HAVING` expects a boolean). However, SQLite allows this and this variant is what we use in this class.

```
CREATE TABLE shopping_list AS
  SELECT p.name, l.store
  FROM products AS p, lowest_prices AS l
  WHERE l.item = p.name
  GROUP BY p.category
  HAVING MIN(p.msrp / p.rating)
  ORDER BY p.name;

  -- Alternate solution using implicit inner join:
  -- SELECT p.name, l.store
  -- FROM products AS p, lowest_prices AS l
  -- WHERE l.item = p.name
  -- GROUP BY p.category
  -- HAVING MIN(p.msrp / p.rating)
  -- ORDER BY p.name;
```

Q7: Bandwidth

Using the Mbs (`megabits`) column from the `stores` table, write a query to calculate the total amount of bandwidth needed to get everything in your shopping list (assume the `shopping_list` table from the previous question is already defined).

You should expect the following output:

total_bandwidth
85

```
CREATE TABLE bandwidth AS
  SELECT SUM(s.mbs) AS total_bandwidth
  FROM stores AS s, shopping_list AS sl
  WHERE s.store = sl.store;

  -- Alternate solution using explicit inner join:
  -- SELECT SUM(s.mbs) AS total_bandwidth
  -- FROM stores AS s
  -- JOIN shopping_list AS sl
  -- ON s.store = sl.store;
```