CS 61A Summer 2022

Regular Expressions, SQL

Discussion 11: August 2, 2022 Solutions

Regular Expressions

Regular expressions are a way to describe sets of strings that meet certain criteria, and are incredibly useful for pattern matching.

The simplest regular expression is one that matches a sequence of characters, like `aardvark` to match any "aardvark" substrings in a string.

However, you typically want to look for more interesting patterns. We recommend using an online tool like regexr.com or regex101.com for trying out patterns, since you'll get instant feedback on the match results.

Character classes

A character class makes it possible to search for any one of a set of characters. You can specify the set or use pre-defined sets.

Class	Description
`[abc]`	Matches a, b, or c
`[a-z]`	Matches any character between a and z
`[^A-Z]`	Matches any character that is not between A and Z.
`\w`	Matches any "word" character. Equivalent to `[A-Za-z0-9_]`.
`\d`	Matches any digit. Equivalent to `[0-9]`.
`[0-9]`	Matches a single digit in the range 0 - 9. Equivalent to `\d`.
`\s`	Matches any whitespace character (spaces, tabs, line breaks).
`.`	Matches any character besides new line.

Character classes can be combined, like in `[a-zA-Z0-9]`.

Combining patterns

There are multiple ways to combine patterns together in regular expressions.

Combo	Description
`AB`	A match for A followed immediately by one for B. Example: `x[.,]y` matches "x.y" or "x,y".
A|B	Matches either A or B. Example: \d+|Inf matches either a sequence containing 1 or more digits **or** "Inf".

A pattern can be followed by one of these quantifiers to specify how many instances of the pattern can occur.

Symbol	Description
`*`	0 or more occurrences of the preceding pattern. Example: `[a-z]*` matches any sequence of lower-case letters or the empty string.
`+`	1 or more occurrences of the preceding pattern. Example: `\d+` matches any non-empty sequence of digits.
`?`	0 or 1 occurrences of the preceding pattern. Example: `[-+]?` matches an optional sign.
`{1,3}`	Matches the specified quantity of the preceding pattern. `{1,3}` will match from 1 to 3 instances. `{3}` will match exactly 3 instances. `{3,}` will match 3 or more instances. Example: `\d{5,6}` matches either 5 or 6 digit numbers.

Groups

Parentheses are used similarly as in arithmetic expressions, to create groups. For example, `(Mahna)+` matches strings with 1 or more "Mahna", like "MahnaMahna". Without the parentheses, `Mahna+` would match strings with "Mahn" followed by 1 or more "a" characters, like "Mahnaaaa".

Anchors

- `^`: Matches the beginning of a string. Example: `^(I|You)` matches I or You at the start of a string.
- `$`: Normally matches the empty string at the end of a string or just before a newline at the end of a string. Example: `(\.edu|\.org|\.com)$` matches .edu, .org, or .com at the end of a string.
- `\b`: Matches a "word boundary", the beginning or end of a word. Example: `s\b` matches s characters at the end of words.

Special characters

The following special characters are used above to denote types of patterns:

```
\ / ( ) [ ] { } + * ? | $ ^ .
```

That means if you actually want to match one of those characters, you have to *escape* it using a backslash. For example, `\(1\+3\)` matches "(1 + 3)".

Using regular expressions in Python

Many programming languages have built-in functions for matching strings to regular expressions. We'll use the Python re module in 61A, but you can also use similar functionality in SQL, JavaScript, Excel, shell scripting, etc.

The search method searches for a pattern anywhere in a string:

```
re.search(r"(Mahna)+", "Mahna Mahna Ba Dee Bedebe")
```

That method returns back a match object, which is considered truth-y in Python and can be inspected to find the matching strings. If no match is found, returns `None`.

For more details, please consult the re module documentation or the re tutorial.

Q1: Email Domain Validator

Create a regular expression that makes sure a given string `email` is a valid email address and that its domain name is in the provided list of `domains`.

An email address is valid if it contains letters, number, or underscores, followed by an @ symbol, then a domain.

All domains will have a 3 letter extension following the period.

> **Hint**: For this problem, you will have to make a regex pattern based on the elements in the `domains` parameter. A for loop can help with that.
>
> **Extra**: There is a particularly elegant solution that utilizes join and replace instead of a for loop.
>
> **Note:** The skeleton code is just a suggestion; feel free to use your own structure if you prefer.

```
import re
def email_validator(email, domains):
    """
    >>> email_validator("oski@berkeley.edu", ["berkeley.edu", "gmail
    .com"])
    True
    >>> email_validator("oski@gmail.com", ["berkeley.edu", "gmail.
    com"])
    True
    >>> email_validator("oski@berkeley.com", ["berkeley.edu", "gmail
    .com"])
    False
    >>> email_validator("oski@berkeley.edu", ["yahoo.com"])
    False
    >>> email_validator("xX123_iii_OSKI_iii_123Xx@berkeley.edu", ["
    berkeley.edu", "gmail.com"])
    True
    >>> email_validator("oski@oski@berkeley.edu", ["berkeley.edu", "
    gmail.com"])
    False
    >>> email_validator("oski@berkeleysedu", ["berkeley.edu", "gmail
    .com"])
    False
    """
    pattern = r"^\w+@("
    for domain in domains:
        if domain == domains[-1]:
            pattern += domain[:-4] + r"\." + domain[-3:] + r")$"
        else:
            pattern += domain[:-4] + r"\." + domain[-3:] + r"|"
    return bool(re.search(pattern, email))
    # Alternate, elegant solution
    domains_list = "|".join([domain.replace(".", "\.") for domain in
     domains])
    return bool(re.search(rf"^\w+@({domains_list})$", email))
```

Q2: Basic URL Validation

In this problem, we will write a regular expression which matches a URL. URLs look like the following:

URL

For example, in the link `https://cs61a.org/resources/#regular-expressions`, we would have:

- Scheme: `https://`
- Domain Name: `cs61a.org`
- Path to the file: `/resources`
- Anchor: `/#regular-expressions`

The port and parameters are not present in this example and you will not be required to match them for this problem.

You can reference this documentation from MDN if you're curious about the various parts of a URL.

For this problem, a valid **domain name** consists of two "words" separated by a single period. Recall that a "word" can consist of letters, numbers, and underscores. The second "word" should be exactly 3 characters long and represents the domain's extension. In the case of the above example, "cs61a" and "org" are the two "words" that are joined by a period.

For a URL to be "valid," it must contain a valid domain name and will optionally have a scheme, path, and anchor. (Note: In this problem, "scheme" does not refer to the programming language.)

A valid **scheme** will either be `http://` or `https://`.

A valid **path** starts with a slash and then must be a valid path to a file or directory. A path to a directory should look something like `path/to/directory`, while a path to a file might look something like `/composingprograms.html` (note the period followed by the extension). Paths should not end with a slash or have more than one period – `/composing.programs.html/` is not a valid path. Any non-slash and non-period character in a path should be a letter, number, or underscore.

A valid **anchor** starts with `/#`. While they are more complicated, for this problem assume that valid anchors will then be followed by letters, numbers, hyphens, or underscores.

> **Hint**: You can use \ to escape special characters in regex.

```
import re
def match_url(text):
    """
    >>> match_url("https://cs61a.org/resources/#regular-expressions
    ")
    True
    >>> match_url("https://pythontutor.com/composingprograms.html")
    True
    >>> match_url("https://pythontutor.com/should/not.match.this")
    False
    >>> match_url("https://link.com/nor.this/")
    False
    >>> match_url("http://insecure.net")
    True
    >>> match_url("htp://domain.org")
    False
    """
    scheme = r"^(https?:\/\/)?"
    domain = r"\w+\.\w{3}"
    path = r"(\/\w+)*(\.\w+)?"
    anchor = r"(\/#[\w-]+)?$"
    full_string = scheme + domain + path + anchor
    return bool(re.match(full_string, text))
```

SQL

SQL is an example of a declarative programming language. Statements do not describe computations directly, but instead describe the desired result of some computation. It is the role of the query interpreter of the database system to plan and perform a computational process to produce such a result.

> For this discussion, you can test out your code at sql.cs61a.org. The records table should already be loaded in.

Select Statements

We can use a **SELECT** statement to create tables. The following statement creates a table with a single row, with columns named "first" and "last":

```
sqlite> SELECT "Ben" AS first, "Bitdiddle" AS last;
Ben|Bitdiddle
```

Given two tables with the same number of columns, we can combine their rows into a larger table with UNION:

```
sqlite> SELECT "Ben" AS first, "Bitdiddle" AS last UNION
...> SELECT "Louis", "Reasoner";
Ben|Bitdiddle
Louis|Reasoner
```

We can SELECT specific values from an existing table using a FROM clause. This query creates a table with two columns, with a row for each row in the records table:

```
sqlite> SELECT name, division FROM records;
Alyssa P Hacker|Computer
...
Robert Cratchet|Accounting
```

The special syntax SELECT * will select all columns from a table. It's an easy way to print the contents of a table.

```
sqlite> SELECT * FROM records;
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle
...
Robert Cratchet|Accounting|Scrivener|18000|Eben Scrooge
```

We can choose which columns to show in the first part of the SELECT, we can filter out rows using a WHERE clause, and sort the resulting rows with an ORDER BY clause. In general the syntax is:

```
SELECT [columns] FROM [tables]
WHERE [condition] ORDER BY [criteria];
```

For instance, the following statement lists all information about employees with the "Programmer" title.

```
sqlite> SELECT * FROM records WHERE title = "Programmer";
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle
Cy D Fect|Computer|Programmer|35000|Ben Bitdiddle
```

The following statement lists the names and salaries of each employee under the accounting division, sorted in descending order by their salaries.

```
sqlite> SELECT name, salary FROM records
...> WHERE division = "Accounting" ORDER BY salary desc;
Eben Scrooge|75000
Robert Cratchet|18000
```

Note that all valid SQL statements must be terminated by a semicolon (;). Additionally, you can split up your statement over many lines and add as much whitespace as you want, much like Scheme. But keep in mind that having consistent indentation and line breaking does make your code a lot more readable to others (and your future self)!

SQL Queries

For the following questions, you will be referring to the **records** table:

Name	Division	Title	Salary	Supervisor
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
...	...	...	...	...
Robert Cratchet	Accounting	Scrivener	18000	Eben Scrooge

To see the full table, you can go to sql.cs61a.org and enter `SELECT * FROM records;` . However, it is not necessary to see the full table in order to complete the questions.

Q3: Oliver Employees

Write a query that outputs the names of employees that Oliver Warbucks directly supervises.

```
SELECT name FROM records WHERE supervisor = "Oliver Warbucks";
```

Q4: Self Supervisor

Write a query that outputs all information about employees that supervise themselves.

```
SELECT * FROM records WHERE name = supervisor;
```

Q5: Rich Employees

Write a query that outputs the names of all employees with salary greater than 50,000 in alphabetical order.

```
SELECT name FROM records WHERE salary > 50000 ORDER BY name;
```

Joins

Suppose we have another table **meetings** which records the divisional meetings.

Division	**Day**	**Time**
Accounting	Monday	9am
Computer	Wednesday	4pm
Administration	Monday	11am
Administration	Wednesday	4pm

Data are combined by joining multiple tables together into one, a fundamental operation in database systems. There are many methods of joining, all closely related, but we will focus on just one method (the inner join) in this class.

When tables are joined, the resulting table contains a new row for each combination of rows in the input tables. If two tables are joined and the left table has m rows and the right table has n rows, then the joined table will have _m*n_ rows. Joins are expressed in SQL by separating table names by commas in the **FROM** clause of a **SELECT** statement.

```
sqlite> SELECT name, day FROM records, meetings;
Ben Bitdiddle | Monday
Ben Bitdiddle | Wednesday
...
Alyssa P Hacker | Monday
...
```

Tables may have overlapping column names, and so we need a method for disambiguating column names by table. A table may also be joined with itself, and so we need a method for disambiguating tables. To do so, SQL allows us to give aliases to tables within a FROM clause using the keyword AS and to refer to a column within a particular table using a dot expression. In the example below we find the name and title of Louis Reasoner's supervisor.

```
sqlite> SELECT b.name, b.title FROM records AS a, records AS b
...> WHERE a.name = "Louis Reasoner" AND
...> a.supervisor = b.name;
Alyssa P Hacker | Programmer
```

Join Questions

For the following questions, you will be referring to the **records** and **meetings** tables:

records

Name	Division	Title	Salary	Supervisor
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
...	...	...	...	...
Robert Cratchet	Accounting	Scrivener	18000	Eben Scrooge

meetings

Division	**Day**	**Time**
Accounting	Monday	9am
Computer	Wednesday	4pm
Administration	Monday	11am
Administration	Wednesday	4pm

Q6: Oliver Employee Meetings

Write a query that outputs the meeting days and times of all employees directly supervised by Oliver Warbucks.

```
SELECT m.day, m.time FROM records AS r, meetings AS m WHERE r.
    division = m.division
AND r.supervisor = "Oliver Warbucks";
```

Q7: Different Division

Write a query that outputs the names of employees whose supervisor is in a different division.

```
SELECT e.name FROM records AS e, records AS s WHERE e.supervisor = s
    .name AND e.division != s.division;
```

Q8: Middle Manager

A middle manager is a person who is both supervising someone and is supervised by someone different. Write a query that outputs the names of all middle managers.

```
SELECT b.name FROM records AS a, records AS b WHERE a.supervisor = b
    .name AND b.supervisor != b.name;
```

Aggregation

So far, we have joined and manipulated individual rows using SELECT statements. But we can also perform aggregation operations over multiple rows with the same SELECT statements.

We can use the MAX, MIN, COUNT, and SUM functions to retrieve more information from our initial tables. If we wanted to find the name and salary of the employee who makes the most money, we might say

```
sqlite> SELECT name, MAX(salary) FROM records;
Oliver Warbucks|150000
```

Using the special **COUNT(*)** syntax, we can count the number of rows in our table to see the number of employees at the company.

```
sqlite> SELECT COUNT(*) from RECORDS;
9
```

These commands can be performed on specific sets of rows in our table by using the GROUP BY [column name] clause. This clause takes all of the rows that have the same value in column name and groups them together.

We can find the minimum salary earned in each division of the company.

```
sqlite> SELECT division, MIN(salary) FROM records GROUP BY division;
Computer|25000
Administration|25000
Accounting|18000
```

These groupings can be additionally filtered by the HAVING clause. In contrast to the WHERE clause, which filters out rows, the HAVING clause filters out entire groups. To find all titles that are held by more than one person, we say

```
sqlite> SELECT title FROM records GROUP BY title HAVING COUNT(*) >
    1;
Programmer
```

Aggregation Questions

For the following questions, you will be referring to the **records** and **meetings** tables:

records

Name	Division	Title	Salary	Supervisor
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
...	...	...	...	...
Robert Cratchet	Accounting	Scrivener	18000	Eben Scrooge

meetings

Division	**Day**	**Time**
Accounting	Monday	9am
Computer	Wednesday	4pm
Administration	Monday	11am
Administration	Wednesday	4pm

Q9: Supervisor Sum Salary

Write a query that outputs each supervisor and the sum of salaries of all the employees they supervise.

```
SELECT supervisor, SUM(salary) FROM records GROUP BY supervivor;
```

Q10: Num Meetings

Write a query that outputs the days of the week for which fewer than 5 employees have a meeting. You may assume no department has more than one meeting on a given day.

```
SELECT m.day FROM records AS e, meetings AS m WHERE e.division = m.
    division GROUP BY m.day HAVING COUNT(*) < 5;
```

Q11: Rich Pairs

Write a query that outputs all divisions for which there is more than one employee, and all pairs of employees within that division that have a combined salary less than 100,000.

```
SELECT e1.division FROM records AS e1, records AS e2 WHERE e1.name
    != e2.name AND e1.division = e2.division
GROUP BY e1.division HAVING MAX(e1.salary + e2.salary) < 100000;
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.