REGULAR EXPRESSIONS, SQL

CS61A SMALL GROUP TUTORING

August 3-5, 2022

1 Regular Expressions

Regular expressions help you match "patterns" to strings. We use the `re` module. For example, the `re.search()` function returns a Match object, which tells you if a certain "pattern" can be found within a given string.

```
>>> import re
>>> bool(re.search(r"hello", "Why hello!"))
True
>>> bool(re.search(r"bye", "Why hello!"))
False
```

We use the raw string syntax, r"regex",to specify a pattern because it stops any backslash characters from being interpreted as special characters such as newlines.

```
>>> print("my\nstring") # \n is interpreted as newline
my
string
>>> print(r"my\nstring")
my\nstring
```

There are many components to a pattern.

Character Classes: Character classes are one of the ways we can specify what characters your pattern matches. For each example, we match characters from the string, "Hello World! Today is 10/15/2021". Boxed characters are matched.

Character class	Description	Example
`[oa]`	A singular o or singular a	"Hell o W o rld! T o d a y is 10/15/2021"
`[0-9]`, `\d`	Any digit	"Hello World! Today is 1 0 / 1 5 / 2 0 2 1"
`[^a-z]`	Anything except a lowercase letter	" H ello W orld ! T oday is 1 0 / 1 5 / 2 0 2 1 "
`\s`	Any whitespace	"Hello World! Today is 10/15/2021"
`[a-zA-z0-9_]`, `\w`	Any letter, digit, or underscore	Hello World ! Today is 10 / 15 / 2021
`.`	Anything except newline	Hello World! Today is 10/15/2021
`\b`	Word boundary (this is an anchor class!)	Hello World! Today is 10 / 15 / 2021

We can combine character classes in the following two ways.

- `[a-z][A-Z]` matches a lowercase letter AND then an uppercase letter
- `([a-z]|[A-Z])` matches a lowercase letter OR an uppercase letter like `[a-zA-Z]` does. `|` splits the entire pattern to match either the left-hand side or the right-hand side.

There are also two special characters: ^(used outside of a character class) and $. These are called anchors and match the beginning and ends of strings respectively.

Quantifiers: Quantifiers tell you how many of something you will match.

- `[a-z]+` matches one or more lowercase letters
- `[0-9]*` matches zero or more digits
- `[A-Z]?` matches zero or one uppercase letters
- `a{1,3}` matches 1, 2, or 3 a's
- `a{2,}` matches 2 or more a's
- `a{,2}` matches 0, 1, or 2 a's
- `a{2}` matches exactly 2 a's

Python Functions: The Python `re` module has a lot of functions for matching patterns.

```
import re
ptn, stn = r"...", "..."     # ptn = pattern, stn = string
# note: regex patterns use raw strings!
re.search(ptn, stn)          # does `string` contain `pattern`?
re.fullmatch(ptn, stn)       # does all of `string` follow `
   pattern`?
re.match(ptn, stn)           # does `string` start with `pattern`?
re.findall(ptn, stn)         # list of all matches of `pattern
re.sub(ptn, "<nope>", stn)   # replace `pattern` with `"<nope>"`
```

Groups: Groups allow you to group character classes.

```
>>> re.findall(r"ab{1,3}", "abab abb abbb aaab aabb")
['ab', 'ab', 'abb', 'abbb', 'ab', 'abb']
>>> re.findall(r"(ab){1,3}", "abab abb abbb aaab aabb")
['ab', 'ab', 'ab', 'ab', 'ab']
```

1. We are given a sentence, and we want to find all the words that end with ing! Given a string sentence, we want to extract all words that end with ing.

```
import re
def extract_ing(sentence):
    """
    Given a string sentence, finds all words that end
       with "ing". For the purpose of this function,
       words can only have word characters.
    >>> extract_ing("Extracting single word") # single
       does not end with "ing"
    ['Extracting']
    >>> extract_ing("cool wording!")
    ['wording']
    >>> extract_ing("thising, ising,exciting...")
    ['thising', 'ising', 'exciting']
    >>> extract_ing("sad-dening")
    ['dening']
    """
    return re.findall(r"_________", sentence)
```

2. We are given an input string `eq_str` and we want to determine whether it is of the form $mx + b$ where x can be any alphabetic character and m and b are integers or decimals. `linear_functions` returns True if `eq_str` has the correct form for a linear function and False otherwise.

```
import re
def linear_functions(eq_str):
    """
    Given a string, returns whether or not it has the form
       'mx+b'.
    >>> linear_functions("1x+0")
    True
    >>> linear_functions("100y+44")
    True
    >>> linear_functions("99.9z+.23")
    True
    >>> linear_functions("55t")
    True
    >>> linear_functions("x+3")
    True
    >>> linear_functions("10b+")
    False
    >>> linear_functions("+43")
    False
    """
    return bool(re.search(r"__________", eq_str))
```

3. We are given a string password, and we want to determine whether it is a valid password or not. A valid password must only have alphanumeric characters, along with at least one symbol (\?#$%&ˆ_).

```
import re
def validate_password(pwd):
    """
    Given a string pwd, returns whether the password is valid
       or not
    >>> validate_password("thisisapassword") # no symbol
    False
    >>> validate_password("?validPassword")
    True
    >>> validate_password("!")
    True
    >>> validate_password("whoo_hoo")
    True
    """
    return bool(re.search(r"__________", eq_str))
```

2 SQL

SQL (Standardized Query Language) is a declarative programming language that allows us to store, access, and manipulate data stored in databases. Each database contains tables, which can store many rows of data that all share the same properties (columns).

Creating Tables

To create a table, we can use the `CREATE TABLE` operation. For example, if we want to make a table with 2 columns 'name' and 'number' and fill it with 3 rows of data, we could do the following:

```
CREATE TABLE numbers AS
    SELECT "Papa John's Pizza" AS name, 5108457272 AS number
       UNION
    SELECT "UCPD", 5106426760 UNION
    SELECT "Foothill Mailroom", 5106429703;
```

Filtering

We can then filter data using queries which have the following general structure:

```
SELECT col1, col2, ... FROM table WHERE conditions ORDER BY
   column [DESC] LIMIT num;
```

1. `SELECT` chooses specific columns to include in the output. Column names can be changed using the AS operation (for example, `SELECT number AS phone` would rename the number column to 'phone'.)
2. `FROM` chooses which table(s) to select data from. If multiple tables are included, then they are joined together such that every possible combination of rows are outputted. The number of rows in the resulting table will be the multiple of all rows in the original tables (table_a_rows table_b_rows table_c_rows ...). The same table can also be joined to itself if aliasing is used (e.g. `SELECT * FROM numbers AS a, numbers AS b`).
3. `WHERE` restricts which rows appear in the output. Valid conditions include less than/greater than/equal to (<, >, =), AND/OR, and not equal (!=, <>). All comparisons involving aggregations must go in the HAVING clause instead of WHERE.
4. `ORDER BY` sorts rows using the values of the specified column (smallest to largest if numbers, alphabetical order if strings). If the DESC keyword is included, then rows will be sorted from largest to smallest.
5. `LIMIT` restricts the maxiumum number of rows in the output table. For example, `ORDER BY name LIMIT 10` would only get the first 10 names in alphabetical order.

Aggregation

In addition to filtering results, we can also use SQL to perform aggregation. Doing so combines data in a specified manner to get information such as the sum of all numbers in the group.

We can perform aggregation and filtering at the same time using a query with the following structure:

```
SELECT col1, col2, ... FROM table WHERE conditions GROUP BY
   column HAVING conditions ORDER BY column [DESC] LIMIT num;
```

1. `GROUP BY` aggregates the table by combining all rows with the same value into one group. Properties of this group can then be accessed using COUNT, MIN, MAX, SUM, etc.
2. `HAVING` is similar to WHERE, except it performs filtering on aggregate functions rather than the columns themselves. For example, if we want all groups that have less than 5 entries in them, we could filter using `HAVING COUNT(*) < 5`.

CS 61A wants to start a fish hatchery, and we need your help to analyze the data we've collected for the fish populations! Running a hatchery is expensive – we'd like to make some money on the side by selling some seafood (only older fish of course) to make delicious sushi.

The table `fish` contains a subset of the data that has been collected. The SQL column names are listed in brackets.
Table name: `fish`*

Species [species]	**Population [pop]**	**Breeding Rate [rate]**	**$/piece [price]**	**# of pieces per fish [pieces]**
Salmon	500	3.3	4	30
Eel	100	1.3	4	15
Yellowtail	700	2.0	3	30
Tuna	600	1.1	3	20

*(This was made with fake data, do not actually sell fish at these rates)

1. Write a query to find the three most populated fish species.

2. Write a query to find the total number of fish in the ocean. Additionally, include the number of species we summed. Your output should have the number of species and the total population.

3. Profit is good, but more profit is better. Write a query to select the species that yields the most number of pieces for each price. Your output should include the species, price, and pieces.

Business is good, but a bunch of competition has sprung up! Through some cunning corporate espionage, we have determined one such competitor's selling prices. The table `competitor` contains the competitor's `price` for each `species`.

Species [species]	**$/piece [price]**
Salmon	2
Eel	3.4
Yellowtail	3.2
Tuna	2.6

1. Write a query that returns, for each species, the difference between our hatchery's revenue versus the competitor's revenue for one whole fish.

For the following two questions, you have access to two tables.

Grades, which contains three columns: `day`, `class`, and `score`. Each row represents the `score` you got on a midterm for some `class` that you took on some `day`.

Outfits, which contains two columns: `day` and `color`. Each row represents the `color` of the shirt you wore on some `day`. Assume you have a row for each possible day.

Table name: `grades`

Day	Class	Score
10/31	Music 70	88
9/20	Math 1A	72

Table name: `outfits`

Day	Color
11/5	Blue
9/13	Red
10/31	Orange

1. You want to find out which classes you need to prepare for the most by determining how many points you have so far. However, you only want to do so for classes where you did relatively poorly.

   Write a query that will output the sum of your midterm scores for each class along with the corresponding class, but only for classes in which you scored less than 80 points on at least one midterm. List the output from highest to lowest total score.

2. Instead of actually studying for your finals, you decide it would be the best use of your time to determine what your ”lucky shirt” is. Suppose you’re pretty happy with your exam scores this semester, so you define your lucky shirt as the shirt you wore to the most exams.

   Write a query that will output the color of your lucky shirt and how many times you wore it.

Past Exam Questions

- Regular Expressions
  - Sp22 Final Q14, Q15
  - Sp21 Practice Final Q7
- SQL (without aggregation)
  - without aggregation
    - Su21 Final Q4a
    - Fa18 Final Q7a, Q7c
  - Fa19 Final Q10
  - Sp19 Final Q9