Note: this is an optional assignment. You do not have to complete this, but if you do you may earn up to 40 bonus points toward your final grade.

Write a Python function that will interpret a script written in the mini-language defined below. Instead of functions, a program in this language is made up of sequential pipelines that perform basic operations on the data, similar to Awk, SQL, Splunk's query language, etc. The interpreter function will take in the name of a script file, the name of a CSV file, and the name of a pipeline to run. It should load the pipeline definitions from the script, execute the named pipeline with the data in the CSV as input, and then return an object representing the resulting table. You may choose what shape this object takes (a custom class, tuple, etc.), but make sure to describe it in your function's docstring. Additionally, your function should throw one type of exception if the input script has syntax errors, another type if the CSV data contains something other than floats, and a third type if an operation causes an exception during execution (see operation descriptions for when this happens).

You may not install extra libraries to help you, but you may use data structures and functions that come with Python (list comprehensions, map(), filter(), reduce(), etc.).

Language Semantics

This language is meant to perform a series of transformations to a *table*. A table is a matrix of floats with a name for each column. The initial input will be read from a CSV file, but you may use your own internal representation (lists, tuples, instances of your own class, etc.) in the interpreter. Individual transformations are known as *operations*, while series of transformations are known as *pipelines*.

Pipelines

Pipelines are sequences of operations. They define one or more operations to run in a row, taking the result of each one and passing it to the next one.

Operations

The basic building block of this language is an operation. There are only a two operations this language can perform, each with a specific set of inputs. All operations take in a table and result in a table.

Arithmetic

The arithmetic operation takes an operator, the names of two input columns, and the name of an output column. The operator can represent addition, subtraction, multiplication, division, or modulus. The implementation of the arithmetic operation should perform the operation represented by the operator on the two input columns of each row and store the result in the output column. The operation should throw an exception if the input columns don't exist.

Example input:

first	second
3	2
1	0

`arithmetic + first second Alice` will produce:

first	second	Alice
3	2	5
1	0	1

`arithmetic * first second Bob` will produce:

first	second	Bob
3	2	6
1	0	0

`arithmetic + first something Calvin` will result in an exception because `something` is not a column.

Project

The project operation takes the names of one or more columns and produces a table only including those columns. The operation should throw an exception if any of the input columns don't exist.

Example input:

first	second	third
3	2	1
1	0	1

`project third first` will produce:

third	first
1	3
1	1

`project screen` will result in an exception because `screen` is not a column.

Language Syntax Definition

The following defines the syntax for the language. Each symbol in a script should be separated from other symbols by at least one whitespace character, but you may ignore multiple whitespace characters.

EBNF Definition

```
identifier = letter , { letter | digit | "_" } ;
arithmetic_op = "arithmetic" , cell_op , identifier, identifier, identifier ;
cell_op = "+" | "-" | "*" | "/" | "%" ;
project_op = "project" , identifier , { identifier } ;
operation = arithmetic_op | project_op ;
pipeline_def = "pipeline" , identifier , "=" , operation , { "|" , operation } , ";";
program = pipeline_def , { pipeline_def } ;
```

Program Examples

Example 1

Goal: Turn a list of pairs into fractional values (table with columns "left" and "right")

Script: (fraction.pipes)

```
pipeline fraction = arithmetic / left right quotient | project quotient ;
```

Input: (fractions.csv)

left	right
2	3
5	7

Invocation:

```
interpret("fraction.pipes", "fractions.csv", "fraction")
```

Output:

quotient
.6666667
.7142857

Example 2

Goal: Get the difference between scheduled and actual times

Script: (timeDifference.pipes)

```
pipeline timeDifference =
  arithmetic - actualDepTime schedDepTime depTimeDiff |
  arithmetic - actualArrTime schedArrTime arrTimeDiff |
  project depTimeDiff arrTimeDiff ;
```

Input: (times.csv)

schedDepTime	actualDepTime	schedArrTime	actualArrTime
5	12	37	37
10	20	45	50
20	19	55	52

Invocation:

interpret("timeDifference.pipes", "times.csv", "timeDifference")

Output:

depTimeDiff	arrTimeDiff

7	0
10	5
-1	-3

Guidance

While the grammar describes a regular language, you may find it easier *not* to use Python's RE library. Note that with spaces surrounding everything, the pipelines components pieced together with | characters, and multiple pipelines being separated with semicolons, the Python string class's split and strip methods can handle the simple parsing required here.

Version

- Last updated 28-Feb-2022 - added guidance note

Extra Credit HW Rubric

Criteria	Ratings		Pts
Output data structure consistent across different inputs and documented in function's docstring	**5 pts Full Marks**	**0 pts No Marks**	5 pts
throws separate exceptions for invalid pipeline syntax, invalid input data, and invalid column names	**5 pts Full Marks**	**0 pts No Marks**	5 pts
coding style meets PEP8 guidelines	**5 pts Full Marks**	**0 pts No Marks**	5 pts
correctly handles pipelines made of single arithmetic operations	**10 pts Full Marks**	**0 pts No Marks**	10 pts
correctly handles pipelines made of single project operations	**5 pts Full Marks**	**0 pts No Marks**	5 pts
correctly handles pipelines of multiple operators	**10 pts Full Marks**	**0 pts No Marks**	10 pts
			Total Points: 40