

1. The tasks described in this worksheet are part of the formative assessment. They serve the purpose to prepare you for the examination. We will discuss the solutions during the next **interactive session** after they are handed out – while they fit to the lecture of the week they are handed out, they might be discussed in two weeks time due to the bi-weekly exercise schedule.
2. Make sure to plan your time for the whole sheet carefully. The complete exercise should represent approximately three hours of independent study. The time limit indicates how much time you should spend on each task, and not how much time you may actually need; it is important that you engage with the material and not that you complete all tasks perfectly. Feel free to collaborate and team up.
3. The exercises are designed to challenge you and train you further as guided self-study. The time limit might be too ambitious for you; you may team up with colleagues. It is not an issue as long as you manage to at least partially resolve each task within the time budget. If you (and your team) are struggling, reach out for help in Teams! You may also share your thoughts via the Studip Forum.
4. We recommend that you create a (private) Git repository (see <https://gitlab.gwdg.de>) where you store your findings and outcomes while processing the exercises. This portfolio of work can be useful in the future.

Contents

Task 1: MapReduce: Processing of Text Data (Python) (90 min)	1
Task 2: Pipe Diagrams (30 min)	1
Task 3: Data-Flow Programming (Python) (120 min)	2

Task 1: MapReduce: Processing of Text Data (Python) (90 min)

First complete the task from the previous worksheet with the same name. Make sure to read the hints given on the previous sheet for using hadoop.

If you managed to create the Python surrogate last week, the translation to a Hadoop's MapReduce Streaming Program and measuring it shouldn't take long, but it takes time to set up Hadoop using the scripts in the repository. Take your time to check its usage.

Task 2: Pipe Diagrams (30 min)

Assume you have a CSV file with the schema:

```
1 StudentID,StudentName,StudentEmail,ModuleCodes
```

Such a file could store, for each student, the attendance of all modules and details.

An example line could look like this:

```
1 4711,Max Musterman,max.musterman@uni-goettingen.de,['HPDA', 'MODULE2']
```

Consider the following Python program that processes `file.csv` using a data-flow programming model¹:

```
1 d = dataflow.read(file.csv)
2 # Analyse what this could compute based on the definition of the operators below
3
4 flat = d.map(lambda t: (t[0], eval(t[3])))
5 bd = flat.filter(lambda t: HPDA in t[1])
6 bd.write(out.csv)
7
8 # Analyse what this could compute based on the definition of the operators
9 fm = flat.flatmap(lambda t: [[t[0], x] for x in t[1]]) # list comprehension
10 z = fm.group(lambda t: t[1])
11 r = z.reduce(lambda t1: len(t1))
12 print(r)
13
14 # We can also use functions
15 # Calculate cross-product, exclude identity
16 def joinFunc(x, y):
17     if x != y:
18         return [x, y]
19     else:
20         return None
21
22 e = fm.join(fm, joinFunc)
23 print(e)
```

Sketch a pipe diagram for the program using examples.

Portfolio (directory: 5/pipe-diagram)

5/pipe-diagram/example.pdf The pipe diagram visualizing the program

Task 3: Data-Flow Programming (Python) (120 min)

In this exercise, you will implement a simplified data-flow programming model in Python². Note that the prescribed duration is optimistic here - a full solution from sketch can be done in 30 minutes! However, if you are not experienced with Python, you may need much longer. Nevertheless, thinking about the pseudocode should be doable for everyone during this time!

The language could support the following operators:

- `read(file)` – Provide the content of a (trivial) CSV file as a list of tuples.
- `write(file)` – Write the output into the file.
- `filter(function(t))` – Keep the list of tuples for which `function(t)` returns true.
- `map(function(t))` – Transform one tuple for each input tuple.
- `flatmap(function(t))` – Transform each input tuple into a list with arbitrary number of tuples.
- `group(function(t))` – Group together all tuples with the same string (returned by function). The result is a list of tuples containing the list of names and all tuples.

¹The operators are properly defined in the next task.

²You can also use a different language, but we advise to use Python.

Example:

```
1 d = [ [name1, 1, 2],
2       [name2, 3, 4],
3       [name1, 5, 6]]
4 r = d.group(lambda x: x[0])
```

Should return for r:

```
1 r = [[name1, [[name1, 1, 2], [name1, 5, 6]],
2       [name2, [[name2, 3, 4]]] ]
```

- `reduce(function(t))` – Apply the reduction function to each group (or all data if no groups are used) returning a single tuple. The function is called for each group with the list of elements as a single argument.
- `join(y, function(tx,ty))` – Join any tuple in itself with any tuple from y by calling `function(tx,ty)` on each pair of tuples.

These operators should be chainable with the functional programming paradigm. An example program using your class could be the one from the task before. Python already has good concepts and functions for data transformations, but you should implement them yourself to understand them better.

Hints

- Use the `csv` package to read CSV files
- Implement a class keeping the list of tuples internally. Each operator should return a new class instance while updating the changed data. Note that there are several implementations possible (particularly for `join`) which require a user of your library to write more or less complicated code.
- Implement the `__str__()` method for the class to allow convenient printing of the tuples.
- The `eval()` function interprets text provided as Python code. It is dangerous to use in general but can be used for demonstration purposes.

Portfolio (directory: 5/functional)

5/functional/data-flow.py Your (commented) data-flow implementation with a few examples