Data Processing: Formats and Tools

a topic in

DM565 – Formal Languages and Data Processing

Kim Skak Larsen

Department of Mathematics and Computer Science (IMADA)
University of Southern Denmark (SDU)

kslarsen@imada.sdu.dk

September, 2023

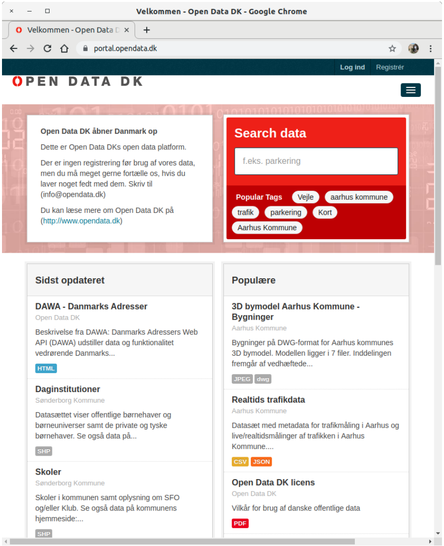
Motivation

There is so much data and information that we can use to understand the world better and to create useful applications; see the next page for an example of data collections.

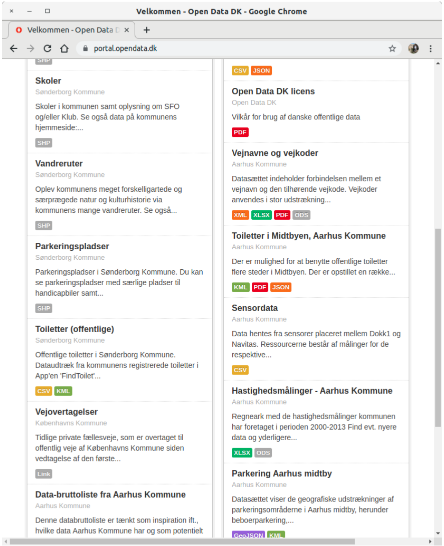
To exploit all of this data efficiently, we need *tools* for examining data quickly (prototyping) before we make our choices and design an application (possibly in a standard programming language).

To decide on which tools to use and how, we need some understanding of the *formats* in which data is stored.

Motivation



Velkommen - Open Data DK - Google Chrome
Velkommen - Open Data D
portal.opendata.dk
Log ind
Registrér
OPEN DATA DK
Open Data DK åbner Danmark op
Dette er Open Data DKs open data platform.
Der er ingen registrering før brug af vores data, men du må meget gerne fortælle os, hvis du laver noget fedt med dem. Skriv til (info@opendata.dk)
Du kan læse mere om Open Data DK på (http://www.opendata.dk)
Search data
f.eks. parkering
Popular Tags
Vejle
aarhus kommune
trafik
parkering
Kort
Aarhus Kommune
Sidst opdateret
DAWA - Danmarks Adresser
Open Data DK
Beskrivelse fra DAWA: Danmarks Adressers Web API (DAWA) udstiller data og funktionalitet vedrørende Danmarks...
HTML
Daginstitutioner
Sønderborg Kommune
Datasættet viser offentlige børnehaver og børneuniverser samt de private og tyske børnehaver. Se også data på...
SHP
Skoler
Sønderborg Kommune
Skoler i kommunen samt oplysning om SFO og/eller Klub. Se også data på kommunens hjemmeside:...
SHP
Populære
3D bymodel Aarhus Kommune - Bygninger
Aarhus Kommune
Bygninger på DWG-format for Aarhus kommunes 3D bymodel. Modellen ligger i 7 filer. Inddelingen fremgår af vedhæftede...
JPEG
dwg
Realtids trafikdata
Aarhus Kommune
Datasæt med metadata for trafikmåling i Aarhus og live/realtidsmålinger af trafikken i Aarhus Kommune....
CSV
JSON
Open Data DK licens
Open Data DK
Vilkår for brug af danske offentlige data
PDF



Velkommen - Open Data DK - Google Chrome
Velkommen - Open Data D
portal.opendata.dk
Skoler
Sønderborg Kommune
Skoler i kommunen samt oplysning om SFO og/eller Klub. Se også data på kommunens hjemmeside:...
SHP
Vandreruter
Sønderborg Kommune
Oplev kommunens meget forskelligartede og særprægede natur og kulturhistorie via kommunens mange vandreruter. Se også...
SHP
Parkeringspladser
Sønderborg Kommune
Parkeringspladser i Sønderborg Kommune. Du kan se parkeringspladser med særlige pladser til handicapbiler samt...
SHP
Toiletter (offentlige)
Sønderborg Kommune
Offentlige toiletter i Sønderborg Kommune. Dataudtræk fra kommunens registrerede toiletter i App'en 'FindToilet'...
CSV
KML
Vejovertagelser
Københavns Kommune
Tidlige private fællesveje, som er overtaget til offentlig veje af Københavns Kommune siden vedtagelse af den første...
Link
Data-bruttoliste fra Aarhus Kommune
Aarhus Kommune
Denne databruttoliste er tænkt som inspiration ift., hvilke data Aarhus Kommune har og som potentielt
CSV
JSON
Open Data DK licens
Open Data DK
Vilkår for brug af danske offentlige data
PDF
Vejnavne og vejkoder
Aarhus Kommune
Datasættet indeholder forbindelsen mellem et vejnavn og den tilhørende vejkode. Vejkoder anvendes i stor udstrækning...
XML
XLSX
PDF
ODS
Toiletter i Midtbyen, Aarhus Kommune
Aarhus Kommune
Der er mulighed for at benytte offentlige toiletter flere steder i Midtbyen. Der er opstillet en række...
KML
PDF
JSON
Sensordata
Aarhus Kommune
Data hentes fra sensorer placeret mellem Dokk1 og Navitas. Ressourcerne består af målinger for de respektive...
CSV
Hastighedsmålinger - Aarhus Kommune
Aarhus Kommune
Regneark med de hastighedsmålinger kommunen har foretaget i perioden 2000-2013 Find evt. nyere data og yderligere...
XLSX
ODS
Parkering Aarhus midtby
Aarhus Kommune
Datasættet viser de geografiske udstrækninger af parkeringsområderne i Aarhus midtby, herunder beboerparkering,...
GeoJSON
KML

Data Formats

There is an abundance of formats:

HTML, JSON, XML, CVS, XHTML, TSV, TXT, DOC, MOV, TIFF, DOCX, PDF, XLS, XLSX, PDF, TEX, PNG, GIF, MPG, SQL, JPG, RTF, MARKDOWN, KML, ...

To organize this, we divide them into three rough categories.

Data Formats in Rough Categories

Tabular Data
sql, csv, tsv, (xls, etc.), ...

Parentheses Structures
xml, json, markdown, (html), ...

Others
mpg, jpg, pdf, doc, ...

In the “others” category, data extraction is problematic, either nearly impossible or requiring specialized tools such as image analysis or similar.

We focus on the first two...

Tabular Data Formats

These are data formats implementing a list of records (tuples, rows) such as

Animal	Cuteness
Giant Panda	1.0
Sea Otter	0.95
Meerkat	0.9
Rabbit	0.8
Red Panda	0.8
Leopard	0.7
Clown Fish	0.4
Python	0.1
Rat	0.07
Tarantula	0.00001

Tabular Data Formats

Tabular data can be encoded as text files using a designated separator character between fields and newline between two records.

`tsv` – tab-separated values – is one such standard, where `\t` (tab) is the separator character.

`csv` – comma-separated values – is another standard, where the separator character is a comma. Sometimes this term is used broadly for the general idea, and one can specify the separator character (to be tab, for instance).

These formats may come with a header record, i.e., a first line specifying the names of the different fields.

`sql` tables, `xlsx` documents, or similar may contain tabular data, but often with more complex additional information. However, the tabular information itself can often be exported to a `csv` file, for instance.

Thus, tabular information can often be processed by line-based tools.

CSV-Like Formats

No absolute standard, but special characters such as comma (if that is the separator) and newline must be quoted or escaped.

Likely “rules” to check data for:

- Header record, possibly optional.
- Same number of comma-separated fields in each record.
- What should be escaped and how? For instance, fields with commas and newlines should be quoted and then a quote should be doubled.

Be aware regarding the following:

- Spaces are probably part of a field.
- Is an empty line white-space or an empty record?

CSV-Like Formats: Example

The one record

Animal	Cuteness
Giant, "The Cutie", Panda	1.0

should likely be represented as

```
"Giant, ""The Cutie"", Panda",1.0
```

CSV-Like Formats: Resources

https://frictionlessdata.io/specs/csv-dialect/

https://docs.python.org/3/library/csv.html

CSV-Like Formats: Python CSV Module

```
> cat example.csv
Giant Panda,1.0
Sea Otter,0.95
Meerkat,0.9
Rabbit,0.8
Red Panda,0.8
Clown Fish,0.4
Python,0.1
Tarantula,0.00001
>
```

CSV-Like Formats: Python CSV Module

```
> cat readWriteCSV.py
import csv
exampleFile = open('example.csv', 'r')
exampleReader = csv.reader(exampleFile)
exampleData = list(exampleReader)
exampleFile.close()
print(exampleData)
outputFile = open('output.csv', 'w')
outputWriter = csv.writer(outputFile)
for record in exampleData:
    outputWriter.writerow(record)
outputWriter.writerow(['Leopard', '0.7'])
outputFile.close()
> python3 readWriteCSV.py
```

CSV-Like Formats: Python CSV Module

prints

```
[['Giant Panda', '1.0'], ['Sea Otter', '0.95'], ['Meerkat', '0.9'], ['Rabbit',
'0.8'], ['Red Panda', '0.8'], ['Clown Fish', '0.4'], ['Python', '0.1'], ['Tara
ntula', '0.00001']]
```

and output.csv contains

```
> cat output.csv
Giant Panda,1.0
Sea Otter,0.95
Meerkat,0.9
Rabbit,0.8
Red Panda,0.8
Clown Fish,0.4
Python,0.1
Tarantula,0.00001
Leopard,0.7
>
```

CSV-Like Formats: Python CSV Module

SDU

Can specify various things such as

- `delimiter`
- `lineterminator`
- ...

CSV-Like Formats

SDU

One can convert back and forth between CSV formats and many other formats. Most spreadsheets and database management systems support the format.

Some editors support the format such that one can get a better editing experience, e.g., getting a column-based layout.

Ex: emacs has modes for operating on TSV or CSV files.

Data Transformation

Recall the natural steps in a data transformation process:

- data discovery
- data mapping
- code generation
- code execution
- data review

Parts of the process are repeated if the data review is not completely successful.

Command-Line Tools

It is what developers use... Among many other tools, of course.

"Native" in Linux and macOS.

Possible in Windows via Windows Subsystem for Linux (WSL 2).

You probably saw a very brief introduction to command-line tools first year.

Everything we do with command-line tools could be done using Java or Python, so why bother?

- Fast and easy data discovery
- Fast, easy, and incremental code generation (prototyping)
- Highly efficient code execution on large datasets

How to Learn

Try all the commands on small examples, do the exercises, check the man-pages.
Long-term learning: *Decide never to do anything repetitive again!*

Command-Line Tools for Data Discovery

Pitfalls

Character encoding: ascii, UTF-8, ISO-8859, ...

Line separator: Unix style newline (LF) or MS-DOS-style (CRLF).
LF is `\n` (ascii 10), CR is `\r` (ascii 13).

Tools

`wc` – print newline, word, and byte counts for each file argument.

`file` – determine file type of file argument.

`recode` – convert between character sets, e.g., `recode l1..u8`.

`od` – octal dump, i.e., actually see the bytes, e.g.,
`od -tcuC` – show byte value in decimal and ascii character if printable.

Command-Line Tools for Data Discovery

SDU

Examples

`myfile` contains the one line “blåbærgrød”; originally encoded in Latin1, but then we recode to UTF-8.

```
> file myfile
myfile: ISO-8859 text
> wc myfile
 1  1 11 myfile
> od -tcuC myfile
0000000   b   l 345   b 346   r   g   r 370   d  \n
         98 108 229  98 230 114 103 114 248 100  10
0000013
> recode l1..u8 myfile
> file myfile
myfile: UTF-8 Unicode text
> wc myfile
 1  1 14 myfile
> od -tcuC myfile
0000000   b   l 303 245   b 303 246   r   g   r 303 270   d  \n
         98 108 195 165  98 195 166 114 103 114 195 184 100  10
0000016
>
```

Command-Line Tools for Code Generation

Example tools include

- grep
- sed
- gawk (GNU awk)
- sort
- uniq
- tr
- cut
- paste
- join
- head/tail

The first three use regular expressions, as do editors, programming languages, etc.

Regular Expressions in Practice

Major differences between regular expressions in practice and regular expressions from formal languages textbooks:

- Alphabets are large (in the hundreds); not just $\{0, 1\}$ or $\{a, b\}$.
- Some symbols in the alphabet are not printable characters.
- Operators of regular expressions are characters, and are *also* in the alphabet.

These issues create problems that we discuss now.

Regular Expressions in Practice

We introduce short-hands such as (examples depend on the tool)

- **.** matches any one character different from **\n**.
- **[a-z]** matches any one character in the given range.
- **^** matches the empty string, but only at the beginning of a line.

We "make" the most popular non-printable characters representable, such as **\n**, **\t**, **\r**, ...

We *escape* either the operators or the characters with the same representation as the operators, e.g.,

- We have seen examples where union (or) is written \|.
- We have seen examples where the parentheses in (.*) *groups* the regular expression .*, so then the parenthesis character must be represented by \(. And of course backslash must be backslashed!

The choice of what to escape is tool-dependent.

Command-Line Tools

The tools can do more than we show; sometimes *much* more.

Basic and often sufficient information can be found via the man-pages, e.g., **`man grep`**.

Many of the tools have online manuals or tutorials available and books can be purchased.

Command-Line Tool: grep

global **r**egular **e**xpression **p**rint

How to Learn

The word file is great for testing, because it has lots of data, but it is still relatively easy to to determine if a correct answer has been found. It is also always good to make small examples.

At `https://gist.github.com/WChargin/8927565`, you can find a classic file of words, present on some systems as **`/usr/share/dict/words`**.

Primary Functionality

`grep` reports lines containing a substring *matching* the regular expression, i.e., a substring in the language of the regular expression.

It takes a file as input or reads input from **`stdin`**:

```
> grep '42' filename
> cat filename | grep '42'
```

Command-Line Tool: grep

Before we discuss the regular expressions used in **`grep`**, we discuss one option commonly used, namely **`-E`**.

When using the option, basically all characters that have special meaning are unescaped; when using **`grep`** *without* that option, the following are interpreted as regular characters and must be escaped to be interpreted as operators:

`?`, **`+`**, **`|`**, **`(`**, **`)`**, **`{`**, **`}`**

Unless one is matching these special characters, regular expressions are often more readable when using **`grep -E`**.

Command-Line Tool: grep

SDU

Examples

In the file

```
()
ac
```

- `grep '()'` – the first line is a match
- `grep '(a|b)c'` – no matching lines
- `grep '\(a\|b\)c'` – the second line is a match

On the other hand,

- `grep -E '()'` – both lines match (the parenthesis are just grouping so the regular expression is equivalent to ε which is a substring of any line)
- `grep -E '(a|b)c'` – the second line is a match
- `grep -E '\(a\|b\)c'` – no matching lines

Command-Line Tool: grep

We now go through the operators of the regular expressions used by **grep -E**.

The simplest expressions are characters in the alphabet. If they are special characters, they must be escaped (as on the previous slide). The same goes for non-printable characters such as tab or newline.

The standard operators union and Kleene star are represented by `|` and `*`. Writing two regular expressions next to each other indicates concatenation.

Example

grep -E 'Kim Skak\tLarsen|Tobias Klink\tLehn' will match lines where one of your educators appear in a representation where first names and last names are separated by a tab.

Command-Line Tool: grep

^ matches ε, but only first on a line. Similarly, **$** matches ε, but only last on a line.

. matches any single character different from newline.

Example

Words starting and ending with a "k" can be found by

```
grep -E '^k.*k$' /usr/share/dict/words
```

Command-Line Tool: grep

Character groups can be defined using square brackets. If ^ is placed immediately following the opening square bracket, the defined character set is the complement relative to the alphabet. Hyphens can be used to indicate ranges.

Examples

`[aeiouy]` matches any single vowel and `[^aeiouy]` matches any single character that is not a vowel.

`[a-zA-Z0-9_]` matches any single character normally allowed in an identifier in a programming language.

^ is only special when appearing first: `[^^]` matches any character which is *not* ^.

Some predefined character ranges exist such as, for instance,

`[:alnum:]` for alpha-numeric

`[:punct:]` for punctuation

`[:digit:]` for digits

They only define the range, so `[a-zA-Z0-9]` could also be written `[[:alnum:]]`.

Command-Line Tool: grep

Commonly used repetition constructs in addition to `*` has been added (`r` is some regular expression and `n` and `m` are natural numbers):

`r+` – one or more repetitions of `r` (same as `rr*`)

`r?` – zero or one `r` (same as ε|`r`)

`r{n,m}` – matches if `r` can be matched at least `n` and at most `m` times.

Abbreviations

`{n}` means `{n,n}`

`{n,}` means `{n,`∞`}`

`{,m}` means `{0,m}`

Example

The line

```
prerequisite
```

gives a match with `grep -E '(re){2}'`. We also get a match with `grep -E '(re){1}'` and `grep -E '(re){0}'` (which is equivalent to ε), but not with `grep -E '(re){3}'`.

Command-Line Tool: grep

Every time parenthesis are used, also when we just use them to resolve a precedence issue, they define a *group* that we can refer to later using *backreferencing*.

The expression `\1` matches the exact string that the first group matched. Nine groups can be handled, allowing us to write one positive digit after the backslash.

Examples

The words "bonbon" and "mama" (alone on a line) are a match to `grep -E '^(.*)\1$'`.

The words 'abracadabra'" and "hotshots" are a match to `grep -E '^(....).*\1$'`.

`grep -E '^(.)(.)(.)\3\2\1$'` finds palindromes of length 6.

`grep -E '(.)(.)(.).\3\2\1'` finds lines *containing* a palindrome of length 7, such as "in**terpret**er".

Command-Line Tool: grep

More Examples

US dollar amounts can be captured by `\$[0-9]+(\.[0-9]{2})?`

US times can be captured by `([1-9]|1[012]):[0-5][0-9] (am|pm)`

Lines starting and ending with the same word can be captured by
`^([[:alpha:]]+) .* \1$`

Lines with two separate parentheses can be captured by the following:

Using **grep -E**:

```
grep -E '(\(.*\).*){2,}'
```

Using **grep** without the **-E** option:

```
grep '\((.*).*\)\{2,\}'
```

Command-Line Tool: grep

SDU

`grep` has lots of options that can be useful in different situations, e.g.,

- **`-n`** print the line number when finding matching lines in a file
- **`-v`** negate matches, i.e., print lines *not* matching the regular expression
- **`-i`** ignore case