File IO and Text Processing

1DV501/1DT901 - Introduction to Programming

Jonas Lundberg, office B3024

Jonas.Lundberg@lnu.se

The slides are available in Moodle

September 28, 2022

Remember to ...

- Sign up for the first Python Test on **Friday, October 7.**
- Read information in Moodle.
- Registration is mandatory. **Deadline October 2.**
- Registration is now open in Moodle (Scroll down a bit to find it.)

- Campus students
  - The test will take place at Campus Växjö or at Campus Kalmar.
  - Exact time and place for the test will be presented later on.

- Distance (Physics) students
  - Will be given an opportunity to take the Python Test remotely.
  - The test will be monitored using Zoom. You will be asked to setup a webcam (or mobile phone) in such a way that you and your computer is in clear view during the test.
  - More instructions related to the distance version of the Python Test will be presented later on in Moodle.

Today ...

- Recursion
- Tree View of the File System
- Working with Files and Directories
- File input/output (IO)
  - Working text files
  - Working with data files
- Text Processing (Extra material, Assignment 3 preparation)

Reading instructions: 8.3, 9.3
(The slides covers the extra material related to text processing.)

Recursion - An introduction

Recursion: A solution to a problem based on a smaller (but similar) problem.

Example: The sum of all integers in range 1 to n

$$S(n) = 1 + 2 + 3 + \cdots + (n-2) + (n-1) + n$$

- The sum can be computed using iteration:

```
# Computes the sum 1+2+3+...n for any n > 0
def sum(n):
    s = 0
    for i in range(1,n+1):
        s = s + i
    return s

# Program starts
p = sum(100)
print("The sum 1+2+3+4+...+100 is ", p)   # Output: 5050
```

Computing sum using smaller sums

$$S(n) = \underbrace{1 + 2 + 3 + \cdots + (n-2) + (n-1)}_{S(n-1)} + n$$

- The problem can be expressed using a smaller problem: $S(n) = S(n-1) + n$
- **Ex**: S(5) = S(4) + 5
- And moving on ...
  - S(4) = S(3) + 4
  - S(3) = S(2) + 3
  - S(2) = S(1) + 2
  - S(1) = S(0) + 1
  - S(0) = S(-1) + 0 ???

We must find a base case $\Rightarrow$ a case where it all stops!

Arithmetic Sum: Introducing a Base Case

- We need a *base case* to terminate the computation.
- We choose to set the base case to $S(1) = 1$ ($S(0) = 0$ would also work).
- The base case is expressed as a fact, not referring to any smaller problems.
- We now have a **recursive definition**:

$$S(n) = \begin{cases} 1 & n = 1 \qquad (\textit{base case}) \\ S(n-1) + n & n \geq 2 \qquad (\textit{recursive step}) \end{cases}$$

- As a recursive Python function

```python
# Recursive computation 1+2+3+...n for any n > 0
def sum_rec(n):
    if n == 1:
        return 1
    else:
        return n + sum_rec(n-1)     # A recursive call
```

Recursion in practice $\Rightarrow$ a function calls itself

Executing recursive sum

```python
# Recursive computation 1+2+3+...n for any n > 0
def sum_rec(n):
    if n == 1:
        return 1
    else:
        return n + sum_rec(n-1)
```

Executing `sum_rec(5)` $\Rightarrow$ 5 calls to `sum_rec(...)`

```
sum_rec(5)
   sum_rec(4)
      sum_rec(3)
         sum_rec(2)
            sum_rec(1)
            return 1                    // base case
         return 2 + 1                 (= 3)
      return 3 + 3                  (= 6)
   return 4 + 6                 (= 10)
return 5 + 10                 (= 15)
```

Recursion

- Compute a solution to a problem using a smaller (but similar) problem is called *recursion*.
- In general, recursion $\Rightarrow$ a method calls itself.
- In order not to be trapped in an inifinite loop, a base case (at least one) must be part of the definition.
- Everything that can be done recursively can also be done iteratively, but not always as easy.
- Recursive definitions and algorithms are common in mathematics and computer science.
- Well-known problems where recursion helps: Fibonacci numbers, Binary search trees, traversing a folder structure

Simple palindrome: A recursive definition

- A string is a *simple palindrome* if it has the same text in reverse.
- Examples: x, anna, madam, abcdefedcba, yyyyyyyy
- A palindrome can be defined as:
  1. An empty string is a palindrome
  2. A string with the length 1 is a palindrome.
  3. A string is a palindrome if the first and last characters are equal, and all characters in between is a palindrome.
- 1 and 2 are our base cases
- 3 is our recursive step

Simple palindrome: Python

```python
# str is text to be checked,
# p and q are first and last positions in text
def is_palindrome_rec(str, p, q):
    if q <= p:
        return True
    elif str[p] != str[q]:
        return False
    else:
        return is_palindrome_rec(str, p+1, q-1)

# Programs starts
s = "madam"
if is_palindrome_rec(s, 0, 4):
    print(s, " is a simple palindrome")
else:
    print(s, " is not a simple palindrome")
```

Notice: We must not only call `is_palindrome_rec` with the text to be checked, we must also provide the first and last positions in the text

Recursive helper functions

Avoid having to provide (the rather ugly) first and last positions in the text to be checked using a recursive help function

```
def is_palindrome_rec(str, p, q):
    "... see previous slide ..."

# Help function that initializes the recursive function
def is_palindrome(str):
    p = 0                 # First position
    q = len(str) - 1      # Last position
    return is_palindrome_rec(str, p, q)

# Programs starts
s = "jonas"
if is_palindrome(s):      # A better looking call
    print(s, " is a simple palindrome")
else:
    print(s, " is not a simple palindrome")
```

Hence, by introducing a help function we can avoid providing first and last positions in the text to be checked ⇒ a better looking function `is_palindrome`

Example: The Fibonacci Sequence

- In the *Fibonacci* sequence the first two numbers are 0 and 1 and the others are the sum of the two previous numbers.

  ```
  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
  ```

- **Exercise**: Write a recursive function `fib(n)` computing the n:th number in the Fibonacci sequence. For example `fib(0) = 0`, `fib(1) = 1`, and `fib(6) = 8`.

Fibonacci – Solution

- In the *Fibonacci* sequence the first two numbers are 0 and 1 and the others are the sum of the two previous numbers.

  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...`

- **Exercise**: Write a recursive function `fib(n)` computing the *n*:th number in the Fibonacci sequence. For example `fib(0) = 0`, `fib(1) = 1`, and `fib(6) = 8`.

- **Solution**:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

# Program starts
print( fib(6) )     # Output: 8
print( fib(15) )    # Output: 610
```

The First 50 Fibonacci Numbers

- Problem: Print the first 50 numbers in the Fibonacci sequence.
- **Solution:** Simple!

```
for i in range(0, 51):
    print(i, fib(i) )
```

- **Result:** The printing goes slower and slower and then dies.
- **A better solution:**

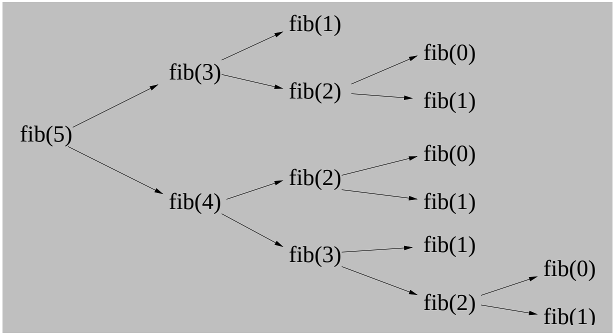
```
f0, f1  = 0, 1
for i in range(2,51):
    f = f0 + f1
    print(i, f)
    f0 = f1
    f1 = f
```

- **Result** (in less than a second)

```
2 1
3 2
...
49 7778742049
50 12586269025
```

Exponential Number of Calls

- Computing `fib(5)`



- fib(5) takes 15 calls to fib(N).
- fib(6) takes 25 calls to fib(N).
- fib(50) takes an enormous amount of calls to fib(N).
- All values between 1 and N $\Rightarrow$ the number is proportional to 2^N $\Rightarrow$ the computer crashes for $N = 50$.

Programming Example - Recursive Printing

Exercise
Write a recursive method `print_string(s)` that prints every character in a string `s` (one character per line). Add also program code that shows how the recursive function `print_string(s)` can be used.

Notice

- Implement a recursive function `print_string_rec(s,n)` that prints one character in each call and where `n` keeps track of the current printing position.
- Let `print_string(s)` be a help function that initializes the recursive call sequence $\Rightarrow$ it calls `print_string_rec(s,n)`
- How can you modify the code to make it print backwards?

Solution - Recursive Printing

```python
# Recursive function
def print_string_rec(s, n):
    if n < len(s):
        print(s[n])
        print_string_rec(s, n+1)


# Starts the recursive call sequence
def print_string(s):
    print_string_rec(s, 0)


# Program starts
print_string("Hello")    # Prints one character at the time
```

We need a recursive help function in this case to avoid an ugly extra parameter in the print function. The extra parameter is needed to keep track of which character to print in each recursive call.

By changing order of the statements `print(s[n])` and `print_string_rec(s, n+1)` in `print_string_rec(s, n)` you will print the string backwards.

Recursive functions (in general)

- A recursive function consists of:
  - One or more *base cases* where "simple" results are given explicitly.
  - One or more recursive rules (or steps) where "larger" results are expressed using "smaller" results.
- **Note**
  - We use *recursive rules* until a problem has been reduced to size where a base case can be used.
  - No base case ⇒ infinite recursion ⇒ program will crash.
- **Crash in practice**

```
# Start an infinite recursive call
def infinite(n):
    infinite(n+1)  # No base case ==> will never stop

# Program starts
infinite(0)
```

Result: The program runs for a second and crashes with a message
`RecursionError: maximum recursion depth exceeded`

Files and directories on your computer

- When programming we often need to access files and directories on our computer
- For example, read data from a file in certain directory, or output processed text to a certain file in a certain directory.
- The Python library `os` (operating system) can be used to quarry/access the file system on your computer.
- For example, to print all the Python (`.py`) files in a certain directory.
- The file system depends on your operating system (e.g. Mac or Windows) $\Rightarrow$ The examples we show today can look slightly different on different computers (operating systems)

Directory 1dv501 on my Mac

- We use my directory `1dv501` as an example in many of the following slides
- Full path (or absolute path) for directory `1dv501`:
  `/Users/jlnmsi/software/python_courses/1dv501`
- Directories and files in `1dv501`

```
1dv501                                      (directory)
   jlnmsi_assign1                           (directory)
       time.py                              (file)
       tax.py                               (file)
       shortname.py                         (file)
       ...
   jlnmsi_assign2                           (directory)
       countdigits.py                       (file)
       ...
   jlnmsi_assign3                           (directory)
       ...
   temp                                     (directory)
       mamma_mia.txt                        (file)
       ...
```

The os module

```
import os                                  # Operating system module

path = os.getcwd()                         # Get current working directory
print("Current dir:", path)

subdir = os.chdir('jlnmsi_assign1')        # Move to subdir jlnmsi_assign1
print("Moved to dir:", os.getcwd())
```

Output (fully qualified path names):

```
Current dir: /Users/jlnmsi/software/python_courses/1dv501
Moved to dir: /Users/jlnmsi/software/python_courses/1dv501/jlnmsi_assign1
```

- The os module gives support for queries related to files and directories
- `os.getcwd()` $\Rightarrow$ name of current working directory
  - The Python virtual machine's start directory in this execution
  - Not same as folder containing this program code
  - Topmost directory inside Visual Studio Code
- `os.chdir('jlnmsi_assign1')` $\Rightarrow$ change to child directory
- `os.chdir('..')` $\Rightarrow$ change to parent directory

Files and directories using module os

```
import os                      # Operating system module

path = os.getcwd()             # Get current working directory
print("Current dir:", path)    # ... /1DV501

lst = os.listdir(path)  # List files and directories in path directory
for s in lst:
    print(s)            # .DS_Store, jlnmsi_assign2, jlnmsi_assign1, .vscode

subdir = os.chdir('jlnmsi_assign1')     # Move to subdir jlnmsi_assign1
print("\nMoved to dir:", os.getcwd())  # ... /1DV501/jlnmsi_assign1
lst = os.listdir(subdir)                # List files and folders in subdir
for s in lst:
    if s.endswith(".py"):      # Print files ending with ".py"
        print(s)               # time.py, tax.py, shortname.py, quote.py, intere
```

- `os.listdir(path)` ⇒ List content (as strings) of directory path
- Directory content ⇒ files and directories
- Hidden entities (e.g. `.vscode`) have names starting with a `.` (dot)
  Hidden ⇒ not visible when traversing file system using Finder (Mac) or Explorer (Windows)

The `os.scandir(...)` Approach

The `os.listdir(...)` approach

- `os.listdir(path)` ⇒ all file and directory names in directory `path`
- Problem: all names are given as strings
  ⇒ hard to know if it is a file or a directory
- Suitable approach when you quickly wants to find the content of a given directory

The `os.scandir(...)` approach

- `os.scandir(path)` ⇒ all files and directories in `path` as `DirEntry` objects
- Each `DirEntry` object `entry` comes with two attributes:
  - `entry.name` ⇒ short local name of file or directory
  - `entry.path` ⇒ fully qualified name of file or directory

  and two methods
  - `entry.is_file()` ⇒ True if `entry` is a file
  - `entry.is_dir()` ⇒ True if `entry` is a directory
- Suitable approach for more complex problems like:
  - List all python files in a given directory
  - Find all sub-directories (transitively) of a given directory

Files and directories using `os.scandir(...)`

Previously: Using `os.listdir(...)`, Now: Using `os.scandir(...)`

```python
import os

def is_hidden(entry):   # True if entry is hidden entity
    return entry.name.startswith(".")

def print_entries(list_of_entries):
    for entry in list_of_entries:
        if entry.is_file() and not is_hidden(entry):
            print("File: ", entry.name)
        elif entry.is_dir() and not is_hidden(entry):
            print("Dir: ", entry.name, entry.path)

path = os.getcwd()
entries = os.scandir(path)  # List of entries of type DirEntry
print_entries(entries)      # See output on next slide
print()

subdir = os.chdir('jlnmsi_assign1')
entries = os.scandir(subdir)  # List of entries of type DirEntry
print_entries(entries)        # See output on next slide
```

Output from previous slide

```
Dir:  temp /Users/jlnmsi/software/python_courses/1dv501/temp
Dir:  jlnmsi_assign2 /Users/ ... /python_courses/1dv501/jlnmsi_assign2
Dir:  jlnmsi_assign1 /Users/ ... /python_courses/1dv501/jlnmsi_assign1
File:  jlnmsi_assign2.zip

File:  time.py
File:  tax.py
File:  shortname.py
File:  quote.py
File:  interest.py
File:  oddpositive.py
File:  sumofthree.py
File:  print.py
File:  randomsum.py
File:  largest.py
File:  squarecolor.py
File:  area.py
File:  fahrenheit.py
File:  change.py
```

Count subdirectories

```python
import os
# Recursive function to count subdirectories to path
def count_dirs(path):
    # print(path)
    no_dir = 1
    entries = os.scandir(path)
    for entry in entries:
        if entry.is_dir():
            no_dir += count_dirs(entry.path)  # Recursive call
    return no_dir

# Program starts
path = "/Users/jlnmsi/Documents/Teaching"
n_dirs = count_dirs(path)

print(f"Dir {path} contains {n_dirs} subdirectories")
# Output: Dir /Users/jlnmsi/Documents/Teaching contains 3620 subdirectories
```

- `count_dirs(path)` is a recursive function that visits all subdirectories
- Visits all subdirectories transitively ⇒ subdirs to subdirs to subdirs ...
- Difficult to handle without recursion

Summary: Exploring files and directories

- When programming we often need to access files and directories on our computer
- The os module give us access to the file system

-
  ```python
  path = os.getcwd()            # Get current working directory
  print("Current dir:", path)

  subdir = os.chdir('jlnmsi_assign1')  # Move to subdir
  print("Moved to dir:", os.getcwd())
  ```
- `os.scandir(path)` ⇒ all files and directories in `path` as `DirEntry` objects
- Each `DirEntry` object `entry` comes with two attributes:
  - `entry.name` ⇒ short local name of file or directory
  - `entry.path` ⇒ fully qualified name of file or directory

  and two methods
  - `entry.is_file()` ⇒ True if `entry` is a file
  - `entry.is_dir()` ⇒ True if `entry` is a directory
- `os.scandir(path)` suitable approach for more complex problems where we traverse the file system and take different actions for directories and files.

A 10 minute break?

ZZZZZZZZZZZZZZZZZZZZ

File input/output (File IO)

- File IO ⇒ File input/output ⇒ read and write data from a file
- We will handle text and numeric data
- Example of text file: `mamma_mia.txt`

```
Mamma mia, here I go again
My my, how can I resist you?
Mamma mia, does it show again
My my, just how much I've missed you?
```

- Two examples of numeric files with different formatting

  File: integers1.dat

```
23
45
73
679
2
9
...
```

File: integers2.dat

```
23; 45; 73; 679; 2; 9; ...
```

Formatting: One long row with semi-colon separated numbers

Reading text from file

```
import os

path = os.getcwd()
path += "/temp/mamma_mia.txt"
print("Reading from ",path, "\n")

file =  open(path,"r")
line_count = 0
for line in file:
    line_count += 1
    print(line)
file.close()
print("Line count: ",line_count)
```

Program output:

```
Reading from ... /1dv501/temp/mamma_mia.txt

Mamma mia, here I go again

My my, how can I resist you?

Mamma mia, does it show again

My my, just how much I've missed you?

Line count:  5
```

- `file =  open(path,"r")` ⇒ open file path for reading (`"r"`)
- `file` is here an object representing a connection to a file
- `for line in file:` ⇒ read from file line by line
- Ugly printout since `line` includes a `"\n"`

Improved file reading

```
path = ...

file =  open(path,"r")
for line in file:
    print(line.strip())
file.close()
```

Ugly print problem solved by using `print(line.strip())`
⇒ remove trailing `"\n"`

```
path = ...

file =  open(path,"r")
full_text = ""
for line in file:
    full_text += line
file.close()
print(full_text)
```

We first store entire text in a string (including linebreaks)

Reading text is easy, just remember: a) We read the text line by line, b) Lines also includes a final `"\n"`, and c) Empty lines are also included.

It is important to close the file connections (`file.close()`) once reading/writing is done. A non-closed connection might cause problems later on when you try to access a file.

Writing text to a file

```
path = ...
full_text = ...

file = open(path,"w")
file.write(full_text)
file.close()
```

Write entire text to file.
Result: Text in file has same formatting as `full_text`.

```
path = ...
lines = ["do\n","re\n","mi\n","fa\n","so\n","la\n"]

file = open(path,"w")
file.writelines(lines)
file.close()
```

We write text line by line to file.
Result: do,re,mi,fa,so,la as six separate lines.

Writing text is also easy, just remember to handle the line breaks.

Recommendations

- Always look at the content of the file you are about to read to understand how it is organized
- Always open the output file when writing to a file to inspect the result
- Start with a small subset of all data if file is large

Reading and Writing text - Summary

We use `open(...)` to make a file connection

- `open(path,"r")` ⇒ open file for reading. Program will crash is file doesn't exists (or is read protected)
- `open(path,"w")` ⇒ open file for writing. The file will be created if it doesn't exist, or replaced if it does exist.
- `open(path,"a")` ⇒ open file for appending ⇒ add new text at the end of a file. The file will be created if it doesn't exist, or appended if it does exist.
- Default is `"r"` ⇒ `open(path)` means open file for reading

We always use `file.close()` to close a file (immediately after reading completed)

`file` in `file = open(...)` is a file object. File object usage:

- `for line in file:` ⇒ read one line at the time
- `full_text = file.read()` ⇒ read entire file content
- `file.write(full_text)` ⇒ write entire text
- `file.writelines(lines)` where `lines` is a list of strings ⇒ write line by line (but not adding any linebreaks)

Safe file handling with `with-as`

```
# Safe file reading
path = ...
with open(path, "r") as file:
   for line in file:
       print( line.strip() )

# Safe file writing
path = ...
with open(path, "w") as file:
   file.write("First line to add\n")
   file.write("Last line to add\n")
```

- `with` and `as` are two Python keywords
- The `with-as` statement includes file closing and was introduced to make sure that an open file is always closed (no matter what happens)
- Although a bit cryptic, it is the recommended approach to open a file.

Reading Data Files

Two examples of numeric files with different formatting

File: integers1.dat

```
23
45
73
679
2
9
```

File: integers2.dat

```
23; 45; 73; 679; 2; 9
```

Formatting: One long row with semi-colon separated numbers

Task: For each file, implement a function `read_file(path)` returning an integer list.

Notice

- Different formatting ⇒ different versions of `read_file(path)`
- We will get data as strings ⇒ must be converted to integers

Read `integers1.dat`

`integers1.dat` formatting: One number on each line

```python
import os

def read_integers1(path):
    lst = []
    with open(path, "r") as file:
        for line in file:          # Read one line at the time
            n = int(line.strip())  # Strip and convert to integer
            lst.append(n)
    return lst


# Program starts
home = os.getcwd()
path = home + "/temp/integers1.dat"
lst1 = read_integers1(path)
print("integers1.dat: ", lst1)    # Output: [23, 45, 73, 679, 2, 9]
```

Basic idea: Read one line at the time, strip, and convert to integer

Read `integers2.dat`

`integers2.dat` formatting: One long row with semi-colon separated numbers

```
import os

def read_integers2(path):
    lst = []
    with open(path, "r") as file:
        as_string = file.read()      # Read entire file (as a long string)
        string_list = as_string.split("; ") # Split string into smaller string
        for s in string_list:    # Convert each string to an integer
            lst.append(int(s))
    return lst

# Program starts
home = os.getcwd()
path = home + "/temp/integers2.dat"
lst2 = read_integers2(path)
print("integers2.dat: ", lst2)  # Output: [23, 45, 73, 679, 2, 9]
```

Basic idea: Read entire file as a single long string, split into list of strings, and convert each string ("23") into an integer.

File IO - Recommendations

Recommendations

- Always look at the content of the file you are about to read to understand how it is organized
  - Text is often read line by line. Use `strip()` to remove line breaks (`\n`)
  - Numeric data can be organized in many different ways. For example, comma-separated or one number per row.
  - Use the `split()` function to divide rows to lists
  - File reading gives data as a string $\Rightarrow$ must be converted to numbers when dealing with numeric data
- Always open the output file when writing to a file to inspect the result
- Start with small samples and print a lot to make sure that everything works

Assignment 3 - Exercise 8 and 9

Exercises 8 and 9 is about identifying words in a large text.

You will be asked to download two large text files:

- `life_of_brian.txt` containing the script of the Monty Python movie Life of Brian. (It is a Python course after all!)
- `swedish_news_2020.txt` containing very (**very!**) many sentences taken from Swedish newspapers.

Task
Write a program `find_words.py` that can identify individual words in a text and store these words (in lower case) in a separate text file.

By a word we here mean strings containing only the English (and Swedish) letters plus "'" and "-". Hence, we consider words like "can't", "John's", and "full-time" as valid words. Furthermore, a word doesn't contain any digits, or symbols like ".", ",", "!", "?", etc.

Notice: The result of this exercise (two files containing only the words from the two text files) will be used later on in other exercises and the Mini-project.

Exercises 8 and 9 - Recommended Approach

1. A function `read_file(file_path)` that reads the file specified by `file_path` and return a list of strings containing each row in the text file.
2. A function `get_words(row)` that divides a row (a string) into words and returns a list of words (strings)
3. A function `save_words(file_path, words)` that saves all the words in the list `words` in the file specified by `file_path`

Step 2, the function `get_words(row)`, is the hard part. How do you divide a row (text) into a list of words? How do you approach the problem?

Exercises 8 and 9 - Identifying words

Input: A sentence from text as a string.
Steps to be taken

1. Convert sentence to lower case sentence
2. Replace punctuation marks (e.g. `"."`,`"!"`,`"?"`,`":"`, `...`) not part of words with `" "`
3. Split sentence into list of *raw words* using `split()`. Example raw words:

   ```
   '11', '53am', 'thursday', '13', '(january)', '2011', 'emotions',
   'fly', 'during', 'ridge', 'lane', 'development', 'hearing', 'watfor
   'town', 'hall', 'reached', 'boiling', 'point', 'when', 'x', 'i'
   'controversial', 'plans', 'for', 'a', 'back', 'garden', 'john'
   'development', 'resurfaced', 'last', 'night', '3mx3m',
   '11am', 'this', 'probably', 'doesn’t', 'come', 'as', 'a', 'm'
   ```

4. Iterate over raw words and remove non-words ⇒ our list of words
5. Example of non-words
   - Any raw word containing digits
   - The only single letter words in English are "a" and "I"

In general, look at printouts, identify non-words, and try to remove them.

Exercises 8 and 9 - Results

Result: A file containing words taken from the input text file.

- No exact word definition $\Rightarrow$ no exact word count
- We got the following word counts: 13372 for `life_of_brian`, and about 15 million for `swedish_news_2020`.
- We don't expect you to get these results exactly, but they should be about the same (say $\pm5\%$).
- Switching from `life_of_brian` to `swedish_news_2020` should be by switching input path. Not two different programs.

Suggestions

- Do not start with large text files.
- Start with text files with only (say) 20 sentences that allows manual inspection (printouts)
- Try using multiple small text samples
- Evaluate effect of, for example removing any raw word containing parentheses, by printing words that are removed $\Rightarrow$ make sure that proper words are not removed.

Assignment 3

- Assignment 3 is now available in Moodle
- Assignment 3 should be submitted using Gitlab.
  This holds for all students!
- Instructions for how to use Gitlab will be published in Moodle. Post questions in Slack if you have problems.

Notice

- Assignment 3 contains fewer but larger and more complex exercises.
- Take a small step approach where you divide the problem into several smaller problems which you solve one at the time. Test that each part works before you move on to the next one. Also, be sure to read the entire exercise text before you start any coding.