Introduction to programming
Lecture 4: processing files and counting words

UNIVERSITY OF
GOTHENBURG

Richard Johansson

September 22, 2015

overview of today's lecture

- file processing
  - splitting into sentences and words
  - character encoding
- counting words with dictionaries
- sorting and maximizing
- introduction to the next assignment

example: most frequent word in GP

```
with open('gp.txt', encoding='utf-8') as f:
    table = {}
    for line in f:
        for word in line.split():
            if word in table:
                table[word] += 1
            else:
                table[word] = 1
print(max(table, key=table.get))
```

overview

file and text processing

dictionaries

sorting and maximizing

introduction to assignment 2

files: the basics

- a **file** is a piece of data that is persistently stored in a computer's storage device (e.g. a hard disk)
- in most operating systems, there are **file names** that help us access our files
- from the computer's perspective, the content of a file is just a bunch of bytes (that is, numbers between 0 and 255)
  - a file has no meaning on its own: a program needs to interpret its content
- a **text file** is a file that contains letters only: no formatting information (unlike Word, PDF, or HTML files)
- we will now see how Python can read strings from text files (non-textual data in later lectures)

opening a file for reading

- before Python can access the contents of a file, the file needs to be **opened**
- use builtin function `open` to open a file for reading

```
with open("textfile.txt") as f:
    ...
```

- `f` is a **file object**

what to do with a file object?

- basic usage: read the whole text file as a string:

```
with open("textfile.txt") as f:
    all_content = f.read()
    print("The content of the file is: %s" % all_content)
```

- when we have read all content, `read` will return an empty string if called again

reading a file line by line

- read one line from a file

```
with open("textfile.txt") as f:
    first_line = f.readline()
    print("The first line is: %s" % first_line)
```

- we can iterate line by line through a file as in a list:

```
with open("textfile.txt") as f:
    for line in f:
        print("The line is: %s" % line)
```

writing to a file

- to write to a text file, we need to open for writing (`"w"`):
- then we write a text using `print`, with an extra input specifying where the output should go:

```
with open("output.txt", "w") as f:
    print("this is the output to the file", file=f)
```

exception handling

- what happens if we try to read a file that does not exist?

```
with open("doesnotexist.txt") as f:
    content = f.read()
    print(content)
```

- an **exception** will be **raised** when something goes wrong
- we will exit whatever we were doing, and if the exception is not **handled**, the program will stop

```
try:
    with open("doesnotexist.txt") as f:
        content = f.read()
        print(content)
except IOError:
    print("I couldn't open the file!")
```

splitting into sentences and words

- for a given text file, we want to print the words one by one
- first (incorrect) solution:

```
def print_words(filename):
    with open(filename) as f:
        for sen in f:
            for word in sen.split():
                print(word)
```

splitting into sentences and words with NLTK

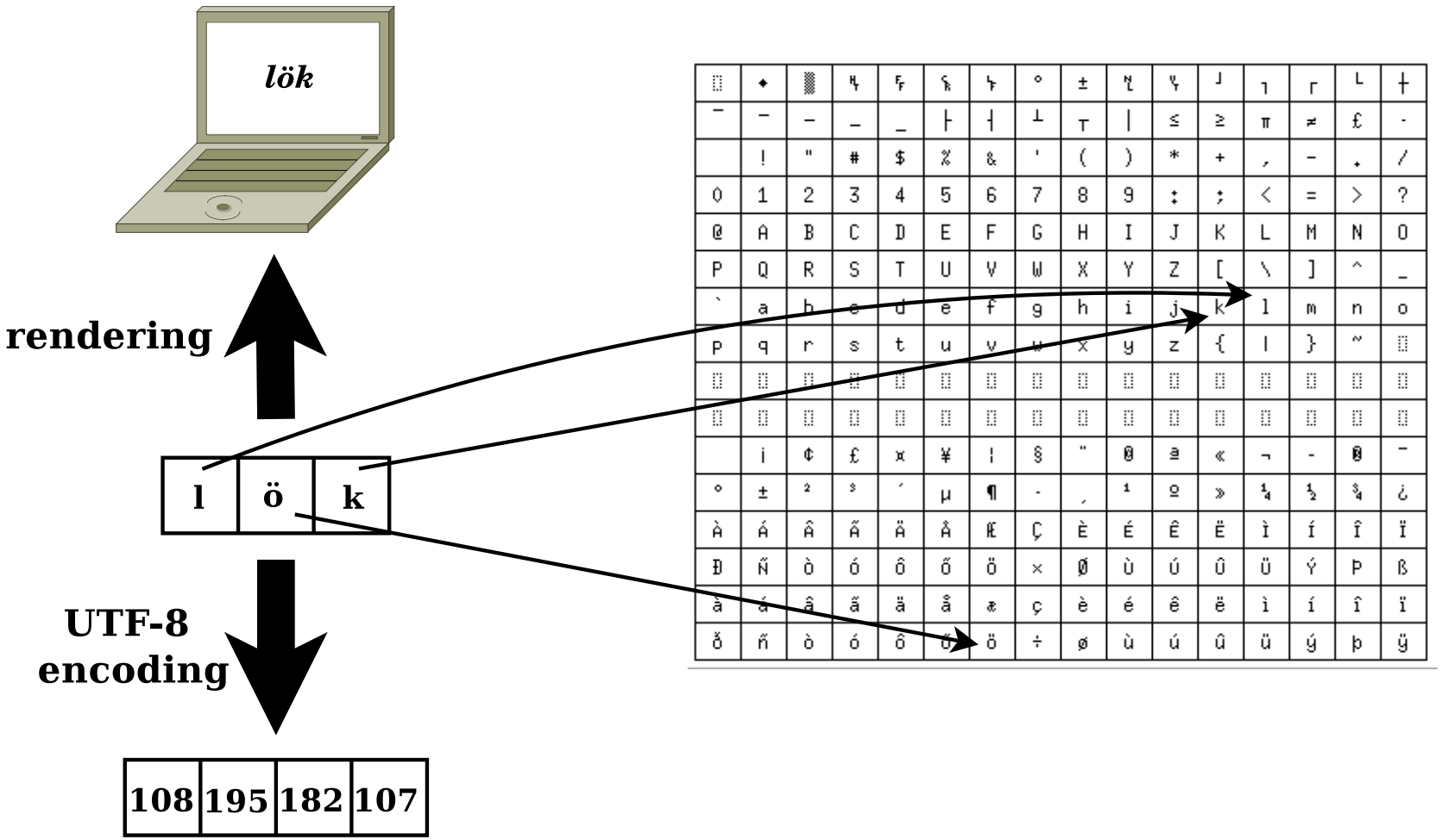
- NLTK includes sentence and word splitting functions:
- better solution:

```
from nltk.tokenize import sent_tokenize, word_tokenize
def print_words(filename):
    with open(filename) as f:
        content = f.read()
        for sen in sent_tokenize(content):
            for word in word_tokenize(sen):
                print(word)
```

going multilingual: Unicode strings

- Python uses **Unicode** strings to represent a sequence of abstract "letters"
- three levels of string processing:
  - byte encoding: what is stored in a file
  - Unicode letters: what we keep in a Python string
  - glyphs from a font: rendered on screen or page
- in Python 2, strings contained bytes; in Python 3 they contain Unicode letters
  - so in Python 3, `len('Göteborg') == 8`
  - . . . but in Python 2, `len('Göteborg') == 9` (typically)

three levels of character processing . . .



rendering . . .

- rendering may be nontrivial in some scripts:

$$\textit{kaf},\ \textit{teh},\ \textit{'alef},\ \textit{beh} \quad \rightarrow \quad \text{كتاب}$$

- even in Latin scripts, we have ligatures such as fi

taking care of the encoding

- nowadays, the **UTF-8** encoding is the most commonly used
- here's how we force `open` to use the UTF-8 encoding:

```
with open('textfile.txt', encoding='utf-8') as f:
    ...
```

- if no encoding is specified, Python uses the default encoding of your system
- on some machines, the default can be an older encoding, so you might need to specify the `encoding` when opening a file

overview

file and text processing

dictionaries

sorting and maximizing

introduction to assignment 2

dictionaries

- **dictionaries** in Python are used to store **key–value mappings**:

Richard Johansson $\rightarrow$ richard.johansson@gu.se
Ildikó Pilán $\rightarrow$ ildiko.pilan@gu.se
Simon Dobnik $\rightarrow$ simon.dobnik@ling.gu.se
Luis Nieto Piña $\rightarrow$ luis.nieto.pina@gu.se

example: looking up email addresses

- we write the dictionary using curly brackets: `{ }`
- similarly to lists, we use square brackets to access the dictionary by its key

```
# initial email dictionary
email_dict = { "Richard":"richard.johansson@svenska.gu.se",
               "Johan":"johan.roxendal@svenska.gu.se" }
# we add another name
email_dict["Simon"] = "simon.dobnik@ling.gu.se"

print(email_dict["Johan"])
```

be careful with nonexistent keys

- the dictionary will give an exception if you try to access a nonexistent key:

```
email_dict = { "Richard":"richard.johansson@svenska.gu.se",
               "Johan":"johan.roxendal@svenska.gu.se" }
# crash!
print(email_dict["Ritva"])
```

- you can test if a key is present:

```
email_dict = { "Richard":"richard.johansson@svenska.gu.se",
               "Johan":"johan.roxendal@svenska.gu.se" }
if "Ritva" in email_dict:
    print(email_dict["Ritva"])
else:
    print("not found!")
```

- alternative: `print(email_dict.get("Ritva", "not found!"))`

example: counting words

```
from nltk.tokenize import sent_tokenize, word_tokenize

def compute_word_frequencies(filename):
    frequencies = {}
    with open(filename) as f:
        content = f.read()
        for sen in sent_tokenize(content):
            for word in word_tokenize(sen):
                if word in frequencies:
                    frequencies[word] += 1
                else:
                    frequencies[word] = 1
    return frequencies

freqs = compute_word_frequencies("test.txt")
print(freqs["the"])
```

example: counting bigrams

```
import nltk

def compute_bigram_frequencies(filename):
    ...

bfreqs = compute_bigram_frequencies("test.txt")
print(bfreqs["New York"])
```

example: what's the probability of the next word?

$$P(\text{next word is York}|\text{current word is New}) = \frac{\text{count(New York)}}{\text{count(New)}}$$

```
def transition_probability(w1, w2):
    ...

print(transition_probability("New", "York"))
```

overview

file and text processing

dictionaries

sorting and maximizing

introduction to assignment 2

sorting

- sometimes we need to **sort** elements of a list (or other collection) into some order:
  - `some_list.sort()` sorts a list in place
  - `sorted(some_collection)` creates a new list and sorts it

```
the_list = [ 8, 7, 3, 6, 11 ]

print(sorted(the_list))

the_list.sort()

print(the_list)
```

detour: default input values

- we can define a **default value** for a function input:

```
def count_words(sentence, separator=" "):
    return len(sentence.split(separator))

print(count_words("this is a test sentence"))

print(count_words("this_is_another_sentence"))

print(count_words("this_is_another_sentence", "_"))
```

detour: calling a function with named inputs

- the inputs can be specified by name instead of order
- this is particularly useful when there are many inputs

```
def count_words(sentence, separator=" "):
    return len(sentence.split(separator))

print(count_words(sentence="this_is_another_sentence",
                  separator="_"))

print(count_words(separator="_",
                  sentence="this_is_another_sentence"))

print(count_words("this_is_another_sentence",
                  separator="_"))

# illegal!
#print(count_words(separator="_",
                   "this_is_another_sentence"))
```

defining your own order

- `list.sort()` and `sorted` use the **natural ordering** of the things in the list
  - that is: they use the comparison `x < y`
- sometimes you need to define your own sorting criteria as a **key** function
  - the key function returns some value by which you want to sort
  - is is specified as the input `key`
- another useful input: `reverse`

sorting example

```
def number_of_vowels(w):
    count = 0
    for c in w:
        if c in ['a', 'e', 'i', 'o', 'u' ]:
            count += 1
    return count

word_list = [ "This", "is", "a", "list", "of", "words" ]

print(sorted(word_list))
print(sorted(word_list, key=len))
print(sorted(word_list, key=number_of_vowels))
print(sorted(word_list, key=len, reverse=True))
```

this program will print:

```
['This', 'a', 'is', 'list', 'of', 'words']
['a', 'is', 'of', 'This', 'list', 'words']
['This', 'is', 'a', 'list', 'of', 'words']
['words', 'This', 'list', 'is', 'of', 'a']
```

max and min

- the max function returns the maximal element of a collection
- conversely, min returns the minimal element
- max and min both allow you to specify your own ordering with key

```
word_list = [ "This", "is", "a", "list", "of", "words" ]

print(max(word_list))
print(max(word_list, key=len))
print(max(word_list, key=number_of_vowels))
```

the most frequent word in a frequency dictionary

```
frequencies = compute_word_frequencies('some_file.txt')

print(max(frequencies, key=frequencies.get))
```

one more data type: tuples

- **tuples** are fixed-size lists that cannot be changed
  - a tuple with 2 items is called a *pair*
  - a tuple with 3 items is called a *triple*
  - a tuple with *n* items is called an *n-tuple*
- tuples are more efficient than normal lists
- they are written with round brackets: `t = (3, "xyz")`
- useful fact about tuples: they can be compared and sorted
  - will sort by first item, then by second item, ...

```
pairs1 = [ (6, "xyz"), (3, "ghi"), (5, "abc") ]
pairs2 = [ ("xyz", 6), ("ghi", 3), ("abc", 5) ]

print(sorted(pairs1))
print(sorted(pairs2))
```

back to dictionaries

- if we have a dictionary `d`, the method `d.items()` gives a collection of key–value pairs

```
email_dict = { "Richard":"richard.johansson@gu.se",
               "Ildiko":"ildiko.pilan@gu.se",
               "Simon":"simon.dobnik@ling.gu.se" }

for pair in email_dict.items():
    name = pair[0]
    email = pair[1]
    print("Name: %s, email: %s" % (name, email))
```

example: sorting alphabetically and by frequency

```
import nltk
def compute_word_frequencies(filename):
    ...
    return frequencies

def get_frequency(word_freq_pair):
    return word_freq_pair[1]

freqs = compute_word_frequencies("test.txt")
word_freq_pairs = freqs.items()
for word_freq_pair in sorted(word_freq_pairs):
    print(word_freq_pair)

for word_freq_pair in sorted(word_freq_pairs,
                             key=get_frequency,
                             reverse=True):
    print(word_freq_pair)
```

overview

file and text processing

dictionaries

sorting and maximizing

introduction to assignment 2

assignment 2: introduction

- for a given file, compute its **Flesch–Kincaid readability score**
- the file is given by a file name
- in addition, print the most “difficult” words and sentences

hint: counting syllables

- English orthography is notoriously messy
- you decide on a suitable simplification
- comment on your simplifications in your report
- just counting the vowels is not enough: e.g. “goose” does not have three syllables
- **Deadline is October 2**