What will YOU use programming for?
Data Analysis

- Scrutinizing large data sets
  - meter readings, usage statistics, connection data
- Coming up with hypotheses
- Verifying the hypotheses
Our goal

- Simple tools for simple data analysis
- Rehearse with small examples
Real world data

Problem

- Where to get it?
- Often sensitive personal information
- May be possible to re-engineer identities from anonymized data
- Example: network logs of the university

Solution for the course

- Use publicly available data
First application
Text analysis
First application

Statistical analysis on public texts

- Obtain a public domain text
  - Gutenberg project
  - Wikipedia (very large)
  - public corpora (e.g.,

- Possible tasks
  - Which language?
  - Which genre?
  - Which author?
### Which Language?

- Every language has a characteristic letter frequency
- also digraphs and trigrams may be analyzed
## On the tasks

### Which Language?

- Every language has a characteristic letter frequency
- also digraphs and trigrams may be analyzed

### Which genre / author?

- Analyze usage patterns of common words
- [https://en.wikipedia.org/wiki/Most_common_words_in_English](https://en.wikipedia.org/wiki/Most_common_words_in_English)
On letter frequency and cryptanalysis

Background: substitution cipher

- plain text and cipher text (after encryption) are drawn from the same set of symbols
- a (monoalphabetic) substitution cipher is a one-to-one mapping between symbols
- particularly simple example: Caesar’s cipher, which rotates letters by 13 (how would you decrypt?)
**Example: Caesar’s cipher**

Caesar’s substitution

<table>
<thead>
<tr>
<th>symbols</th>
<th>abcdefghijklmnopqrstuvwxyz</th>
</tr>
</thead>
<tbody>
<tr>
<td>substitutes</td>
<td>nopqrstuvwxyzabcdefghijklm</td>
</tr>
</tbody>
</table>
Example: Caesar’s cipher

### Caesar’s substitution

<table>
<thead>
<tr>
<th>symbols</th>
<th>abcdefghijklmnopqrstuvwxyz</th>
</tr>
</thead>
<tbody>
<tr>
<td>substitutes</td>
<td>nopqrstuvwxyzabcdefghijklm</td>
</tr>
</tbody>
</table>

### Application

<table>
<thead>
<tr>
<th>plain text</th>
<th>we had goldfish and they circled around</th>
</tr>
</thead>
<tbody>
<tr>
<td>cipher text</td>
<td>jr unq tbyqsvfu naq gurl pvepyrq nebhaq</td>
</tr>
</tbody>
</table>
On letter frequency and cryptanalysis

Breaking a substitution cipher

- Assumptions:
  - language is known
  - cipher text is sufficiently long
- Analyze letter frequency
- Match with letter frequency table for the language
- Compute inverse substitution
Which substitution is the best match?

To assess different substitutions, you need to compute the distance to the language’s letter frequency.

The standard distance function to minimize computes the squareroot of the squares of the differences:

$$d(\bar{x}, \bar{y}) = \sqrt{\sum_i (x_i - y_i)^2}$$
Distance in Python

**Code**

```python
def distance(xs, ys):
    s = 0
    for x, y in zip(xs, ys):
        s += (x - y) * (x - y)
    return math.sqrt(s)
```

**Explanation**

- `zip(xs, ys)` creates a list of pairs of corresponding entries of lists `xs` and `ys`.

- `for x, y in sequence` loops over the entries in `sequence`, which must be pairs, and binds `x` and `y` to the first and second component of each pair, respectively.
Intermezzo

Useful Python IO idioms
Reading a file naively

```
# prepare to 'r'ead from file 'filename'
f = open('filename', 'r')
s = f.read()
# process s = content of file
f.close()
```

- Reads all of a file named “filename” into the string `s`
- Then work with `s`
Reading a file naively

```python
# prepare to 'r'ead from file 'filename'
f = open('filename', 'r')
s = f.read()
# process s = content of file
f.close()
```

- Reads all of a file named “filename” into the string `s`
- Then work with `s`
- Problems:
  - This will consume a lot of memory if the file is big
  - It’s easy to forget to close the file
  - No error handling
More robust file handling

Reading a file (recommended)

```python
with open ('filename', 'r') as f:
    for line in f:
        # process f line-by-line
        # line is a string
```
More robust file handling

Reading a file (recommended)

```python
with open ('filename', 'r') as f:
    for line in f:
        # process f line-by-line
        # line is a string
```

Advantages

- No memory issues as file is read line-by-line
- Automatic close when leaving `with`
- (Hidden) error handling if there is a problem with the file
More robust file handling

Reading a file (recommended)

```python
with open ('filename', 'r') as f:
    for line in f:
        # process f line-by-line
        # line is a string
```

Advantages

- No memory issues as file is read line-by-line
- Automatic close when leaving `with`
- (Hidden) error handling if there is a problem with the file

Disadvantage

Have to deal with file contents one line at a time
Example: the word count utility

```python
# wc counts lines, words, and characters in a file
def exe(name):
    # initialization
    lcount = 0  # line count
    wcount = 0  # word count
    ccount = 0  # character count
    with open(name, 'r') as f:
        for line in f:
            # process one line
            lcount += 1
            ccount += len(line)
            for words in line.split():
                wcount += 1

    return (lcount, wcount, ccount)
```