Scripting Languages

Python data serialization

- data persistence
- pickle module
- exercise

Data persistence

Ability to outlive the process that created it:

- database → for large software projects
  - most flexible data storage/retrieval
  - available modules for database interfacing
  - dependence on external software (configuration/maintenance)
  - significant programming effort (not for simple tasks)
- plain text files → for data exchange/manipulation
  - most universal/exchangeable data format
  - simple implementation for plain data structures
  - requires custom parsing procedures
  - slow for big data
- data serialization → efficient Python software
  - applicable for any kind of data
  - simplest implementation (no parsing)
  - dedicated serialization modules
  - language-specific data format

`pickle` module

- pickle uses files for data storage
  - standard file operations:

    `file = open("filename", 'wb')` or **`open("filename", 'rb')`**

    ```
    ...
    file.close()
    ```

- data serialization/de-serialization
  - functions: `dump()` and `load()`:

    ```
    pickle.dump(data, file)
    data = pickle.load(file)
    ```

- protocols for pickling:
  - efficient binary format by default (write binary - **wb**, read binary – **rb**)
  - old Python 2, text-based, protocols are also supported

Serialization example

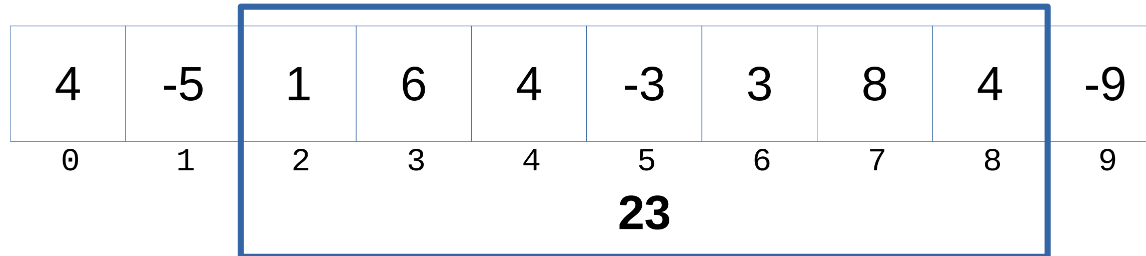
```
import pickle

data = {1: "Hello", 'x': [1,2,{0:0}], (1,'a'):0 }

f = open("datafile.pkl",'wb')
pickle.dump(data, f)
f.close()
```

```
import pickle

f = open("datafile.pkl",'rb')
data = pickle.load(f)
f.close()

print(data)
```

```
{1: 'Hello', (1, 'a'): 0, 'x': [1, 2, {0: 0}]}
```

Exercise

Find the contiguous subsequence with maximum sum

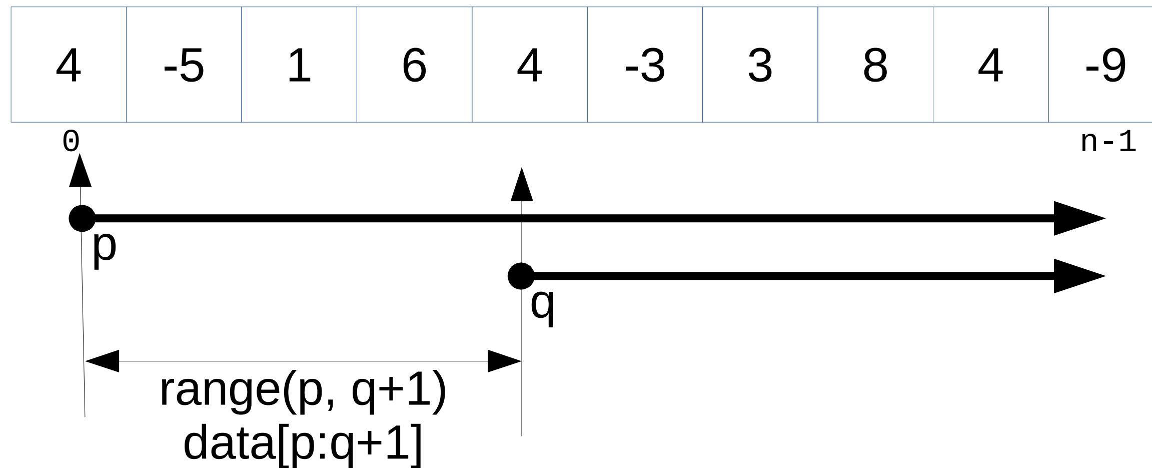


- Read data (as lists) from pickle-files:
  - "substr_1e1.pkl" … "substr_1e5.pkl"
  - 10, 100, 1000, 10000, 1000000 numbers

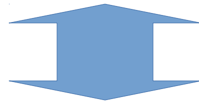
Program structure

```
import pickle
import time

datafile = open("substr_1e1.pkl", 'rb')
data = pickle.load(datafile)
datafile.close()

n = len(data)
maxsum = 0

start = time.time()

here goes your code
...

t = time.time()-start
print(n, t, maxsum)
```

Brute-force solution




```
for p in range(0, n):
    for q in range(p, n):
        s = 0
        for i in range(p, q+1):
            s += data[i]
        maxsum = max(maxsum, s)
```



```
for p in range(0, n):
    for q in range(p, n):
        maxsum = max(maxsum, sum(data[p:q+1]))
```

To Do

- Find out the approximate relation between execution time (t) and number of elements (n)
  - computational complexity: O(?)
- Can you write a better algorithm, faster than brute-force?