

# PV248 Python

Petr Ročkai

# Disclaimer

- I am not a Python programmer
- please don't ask sneaky language-lawyer questions

# Goals

- let's learn to use Python in **practical situations**
- have a look at **existing packages** and what they can do for us
- code up some cool stuff & **have fun**

# Organisation

- I'm in India **next Monday**, Mr. Kaplan will come instead
- starting 9th of Oct, we can start at 8:30 (**let's have a vote**)

# Stuff We Could Try

- working with text, regular expressions
- using the `pdb` debugger
- plotting stuff with `bokeh` (<https://bokeh.pydata.org>)
- talking to `SQL` databases
- talking to `HTTP` servers
- being an `HTTP` server
- implementing a JSON-based `REST API`
- parsing `YAML` and/or `JSON` data
- ... (suggestions welcome)

## Some Resources

- <https://docs.python.org/3/> (obviously)
- <https://github.com/VerosK/python-pv248>
- <https://msivak.fedorapeople.org/python/>
- study materials in IS
- ...

# Part 1: Text & Regular Expressions

## Reading Input

- opening files: `open('scorelib.txt', 'r')`
- files can be iterated

```
f = open( 'scorelib.txt', 'r' )  
for line in f:  
    print line
```

# Regular Expressions

- compiling: `r = re.compile( r"Composer: (.*)" )`
- matching: `m = r.match( "Composer: Bach, J. S." )`
- extracting captures: `print m.group(1)`
  - prints `Bach, J. S.`
- substitutions: `s2 = re.sub( r"\s*$", '', s1 )`
  - strips all trailing whitespace in `s1`

# Other String Operations

- better whitespace stripping: `s2 = s1.strip()`
- splitting: `str.split(';')`

# Dictionaries

- associative arrays: map (e.g.) strings to numbers
- nice syntax: `dict = { 'foo': 1, 'bar': 3 }`
- nice & easy to work with
- can be iterated: `for k, v in dict.items()`

# Counters

- `from collections import Counter`
- like a dictionary, but the default value is `0`
- `ctr = Counter()`
- compare `ctr['baz'] += 1` with `dict`



## Exercise 1: Input

- get yourself a git/mercurial/darcs **repository**
- grab input data (**scorelib.txt**) from study materials
- read and process the text file
- use **regular expressions** to extract data
- use **dictionaries** to collect stats
- **beware!** hand-written, somewhat **irregular** data

## Exercise 1: Output

- print some interesting **statistics**
  - how many pieces by each **composer**?
  - how many pieces composed in a given century?
  - how many in the key of c minor?
- **bonus** if you are bored: searching
  - list all pieces in a given key
  - list pieces featuring a given instrument (say, bassoon)

## Exercise 1: Example Output

- Telemann, G. P.: 68
- Bach, J. S.: 79
- Bach, J. C.: 6
- ...

For centuries:

- 16th century: 10
- 17th century: 33
- 18th century: 4

# Cheat Sheet

```
for line in open('file', 'r')
dict = {}
dict[key] = value
r = re.compile(r"(.*):")
m = r.match("foo: bar")
if m is None: continue
print m.group(1)
for k, v in dict.items()
print "%d, %d" % (12, 1337)
```

read lines

an empty dictionary

set a value in a dictionary

compile a regexp

match a string

match failed, loop again

extract a capture

iterate a dictionary

print some numbers

## Part 2: Databases & SQL

# SQLite

- **lightweight** in-process **SQL** engine
- the entire database is in a **single file**
- convenient python module, **sqlite3**
- stepping stone for a “real” database

## Other Databases

- postgresql (**psycopg2**, ...)
- mysql / mariadb (**mysql-python**, **mysql-connector**, ...)
- big & expensive: Oracle (**cx\_oracle**), DB2 (**pyDB2**)

## More Resources & Stuff to Look Up

- SQL: <https://www.w3schools.com/sql/>
- <https://docs.python.org/3/library/sqlite3.html>
- Python Database API: **PEP 249**
- Object-Relational Mapping
- SQLAlchemy: constructing portable SQL
- SQL Injection

## Database Structure

- defined in `scorelib.sql` (see study materials)
- import with: `sqlite3 scorelib.dat < scorelib.sql`
- you can `rm scorelib.dat` any time to start over
- consult comments in `scorelib.sql`
- do not store duplicate rows



# Python Objects

- `class Foo`, with inheritance: `class Bar( Foo )`
- initialisation: `__init__( self, ... )`
- calling super-class methods: `super().method( param )`
- you can use `super()` to call parent's `__init__`
- object variables are created in `__init__`, not in `class`

## Python Objects (cont'd)

- don't forget `self`!
- `self.variable = 3` sets the object variable
- **different** from `variable = 3`
- set up your variables in `__init__`
- methods take `self` as an explicit argument

## Exercise 2

- create an empty `scorelib.dat` from `scorelib.sql`
- fetch `scorelib-import.py` as a starting point
- part 1: import composers & editors into the database
  - use the pre-made class `Person` for this
  - finish the implementation of its `__init__`
  - use regular expressions (cf. Exercise 1)
  - `string.split()` may come in handy

## Exercise 2 (cont'd)

- part 2: import scores
  - implement a `Score` class similar to `Person`
  - authors should be stored as a list of `Person` objects
  - also fill in the `score_author` table
  - think about how would you de-duplicate `score` rows
- part 3: the rest of the import
  - details in `scorelib.sql`
  - finish at home (you might need this later)

# SQL Cheat Sheet

- `INSERT INTO table (c1, c2) VALUES (v1, v2)`
- `SELECT (c1, c2) FROM table WHERE c1 = "foo"`

## sqlite3 Cheats

- `conn = sqlite3.connect( "scorelib.dat" )`
- `cur = conn.cursor()`
- `cur.execute( "... values (?, ?)", (foo, bar) )`
- `conn.commit()` (don't forget to do this)

## Part 3: SQL Redux & JSON

# JSON

- **structured** data format
- **atoms**: integers, strings, booleans
- **objects** (dictionaries), **arrays** (lists)
- widely used around the web &c.
- **simple** (compared to XML or YAML)

## JSON: Example

```
{  
    "composer": [ "Bach, Johann Sebastian" ],  
    "key": "g",  
    "voices": {  
        "1": "oboe",  
        "2": "bassoon"  
    }  
}
```



# JSON: Writing

- printing JSON **seems** straightforward enough
- **but**: double quotes in strings
- strings must be properly `\`-escaped during output
- also pesky commas
- keeping track of **indentation** for human readability
- better use an **existing library**: `import json`

# JSON in Python

- `json.dumps` = short for **dump to string**
- **python** dict/list/str/... data comes **in**
- a string with valid **JSON** comes **out**

## Workflow

- just convert everything to dict's and lists
- run `json.dumps` or `json.dump( data, file )`

## Python Example

```
d = {}  
d["composer"] = ["Bach, Johann Sebastian"]  
d["key"] = "g"  
d["voices"] = { 1: "oboe", 2: "bassoon" }  
json.dump( d, sys.stdout, indent=4 )
```

Beware: **keys** are always **strings** in JSON

## Exercise 3: Preliminaries

- pull data from `scorelib.dat` using SQL
- print the results as (nicely formatted) JSON
- get input from `sys.argv` (you need to `import sys`)
  - note that `sys.argv[0]` is the program name
- run as, for instance: `python search.py Bach`

## Exercise 3: Part 1

- write a script `getprint.py`
- the input is a `print number`
- the output is a `list of composers`
- you will need to use `SQL joins`
- ```
select ... from person join score_authors  
on person.id = score_author.composer ...  
where print.id = ?
```
- hint: the result of `cursor.execute` is `iterable`

## Exercise 3: Part 2

- write a script `search.py`
- the `input` is a `composer name` (substring)
- the output is a list of `all matching composers`
- along with `all their scores` in the database
- optionally also with print numbers
- ... where `person.name like "%Bach%"`

## Part 4: Plotting with Bokeh

## Preliminaries: Parsing JSON

- `import json`
- `json.load` is the counterpart to `json.dump` from last time
  - de-serialise data from an open file
  - builds lists, dictionaries, etc.
- `json.loads` corresponds to `json.dumps`



# Bokeh

- a library for plotting data in python
- not included in the default python install
- (in shell) `$ pip3 install --user bokeh`
- `from bokeh.plotting import figure, show`

## A Simple Bar Plot

```
from bokeh.plotting import figure, show
p = figure( x_range = (-1,10) )
p.vbar( x = [0, 1], top = [25, 50], width = 0.7 )
show( p )
```

# A Simple Pie Chart

```
from bokeh.plotting import figure, show
from numpy import pi
```

```
p = figure( x_range = (-10,10) )
p.wedge( x = 0, y = 0, radius = 5,
         start_angle = [ 1/4 * pi, 6/4 * pi ],
         end_angle    = [ 6/4 * pi, 1/4 * pi ],
         color = ["purple", "darkblue"] )
show( p )
```

## Creating Data Sources

```
from bokeh.models import ColumnDataSource
src = ColumnDataSource( data = {
    'start': [ 1/4 * pi, 6/4 * pi ],
    'end':   [ 6/4 * pi, 1/4 * pi ],
    'color': [ "purple", "darkblue" ],
    'label': [ "mlem", "purr" ] } )
```

Notice the **label** column – this will become the legend.

# Using Data Sources

```
from bokeh.plotting import figure
p = figure()
p.wedge( x = 0, y = 0, radius = 5,
         start_angle = 'start',
         end_angle   = 'end',
         color = 'color',
         legend = 'label',
         source = src )
```

## Exercise 4

- grab `election.json` from study materials
- part 1: `load the data` and create a `bar plot`
  - bigger parties have a `'color'` key in the JSON
  - they also have a `'short'` key for the acronym
  - set up fallbacks for both (either may be missing)

## Exercise 4 (cont'd)

- part 2: **summarise** the below-one-percent parties
  - only create a single bar for those
  - **add a legend** using the short names
- part 3: make a **pie chart** with the results
- optional: count the share of those who abstained
  - include them as a separate slice in the pie chart