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')`	read lines
`dict = {}`	an empty dictionary
`dict[key] = value`	set a value in a dictionary
`r = re.compile(r"(.*):")`	compile a regexp
`m = r.match("foo: bar")`	match a string
`if m is None: continue`	match failed, loop again
`print m.group(1)`	extract a capture
`for k, v in dict.items()`	iterate a dictionary
`print "%d, %d" % (12, 1337)`	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

```python
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

Part 5: Serving HTTP

Hyper-Text Transfer Protocol

- originally a simple text-based, stateless protocol
- however
  - SSL/TLS, cryptography (https)
  - pipelining (somewhat stateful)
  - cookies (somewhat stateful in a different way)
- typically between client (browser) and a front-end server
- but also as a back-end protocol (web server to app server)

Request Anatomy

- request type (see below)
- header (text-based, like e-mail)
- content

Request Types

- `GET` – asks the server to send a resource
- `HEAD` – like `GET` but only send back headers
- `POST` – send data to the server

Python and HTTP

- both client and server functionality
  - `import http.client`
  - `import http.server`
- TLS/SSL wrappers are also available
  - `import ssl`
- synchronous by default
- async available (next time)

Serving Requests

- derive from `BaseHTTPRequestHandler`
- implement a `do_GET` method
- this gets called whenever the client does a `GET`
- also available: `do_HEAD`, `do_POST`, etc.
- pass the class (not an instance) to `HTTPServer`

Serving Requests (cont'd)

- `HTTPServer` creates a new instance of your `Handler`
- the `BaseHTTPRequestHandler` machinery runs
- it calls your `do_GET` etc. method
- request data is available in instance variables
  - `self.path`, `self.headers`

Talking to the Client

- HTTP responses start with a response code
  - `self.send_response( 200, 'OK' )`
- the headers follow (set at least `Content-Type`)
  - `self.send_header( 'Connection', 'close' )`
- headers and the content need to be separated
  - `self.end_headers()`
- finally, send the content by writing to `self.wfile`

Sending Content

- `self.wfile` is an open file
- it has a `write()` method which you can use
- sockets only accept byte sequences, not `str`
- use the `bytes` built-in function to convert `str` to `bytes`

Exercise 5

- implement a simple HTTP server
  - listen on a high port (e.g. 8000)
  - point your browser to `http://localhost:8000/`
- part 1: serve some static text (or HTML)
- part 2: get & print back some data from the URL
  - e.g. when serving `http://localhost:8000/file.txt`
  - return "you asked for `file.txt`"

Reminder: `https://docs.python.org/3/library`

Part 6: A Simple Web Interface

Parsing URLs

- `import urllib.parse`
- `urlparse` – parses the URL
  - e.g. `url = "/foo?bar=1"`
  - `urlparse( url ).query` gives `bar=1`
- `parse_qs` – parses the query string
  - loads up the key-value pairs into a `dict`

Building Strings

- `str()` – turns an object into a string
- the `%` operator: `printf`-style formatting
  - `"<li>%s</li>" % (item,)`
- the `+` operator (concatenation)
- triple-quote literals for long/multiline strings
  - `template = """<html>... %s ...</html>"""`
  - `html = template % (...)`

Exercise 6

- respond to `GET /result?q=search+term&f=json`
- the functionality is the same as `search.py` in Exercise 3
- part 1: return JSON for the result
  - only if `f=json` is in the query string
  - you can test with `curl` or `wget`
- part 2: format the output as HTML
  - only if `f=html` is in the query string
- part 3: generate a simple web form on `GET /`
  - submit goes to the above (with `f=html`)

Part 7: Basic Linear Algebra with NumPy

Computing in Python

- Python code is generally slow
- and not very memory-efficient either
- hence not suitable for number crunching

- NumPy provides compact array data type
- along with a whole lot of C code to do math
  - uses LAPACK in the backend
- much faster than doing math in pure Python

What can NumPy do for you?

- matrices and linear algebra
  - multiplication, inversion
  - eigenvalues and eigenvectors
  - linear equation solver
- standard math function toolkit
  - trigonometric & hyperbolic
  - exponentials and logarithms
  - and so on…
  - handles complex numbers

What can NumPy do for you? (cont'd)

- discrete Fourier transform
  - useful in signal processing
- a statistical toolkit
  - averages, medians
  - variance, standard deviation
  - histograms
- polynomials
  - basic algebra and calculus
  - least square fitting
- random sampling and distributions

Plotting

- bokeh integrates with NumPy
- and so does matplotlib

Why Plots?

- especially useful for measurement data
- and corresponding regressions

Documentation

- as usual, read online docs for the package
  - `https://numpy.org/`
- also: `help()` in the interactive interpreter
  - use as: `from numpy import linalg`
  - then type `help(linalg.det)`

Entering Data

- most data is stored in `numpy.array`
- can be constructed from from a `list`
  - a list of list for 2D arrays
- or directly loaded from / stored to a file
  - binary: `numpy.load`, `numpy.save`
  - text: `numpy.loadtxt`, `numpy.savetxt`

Exercise 7

- part 1: basic use
  - load a matrix from a text file
  - compute the determinant and inverse
  - both are available in `numpy.linalg`
- part 2: equations
  - load the coefficients like in part 1
  - use `linalg.solve`
  - make sure you understand the meaning

Exercise 7 (cont'd)

- part 3: nice equations
  - parse a human-readable system of equations
  - variables are letters, coefficients are numbers
  - only + and − are allowed
  - print the solution (using variable names)

```
2 x + 3 y = 5
  x -   y = 0
solution: x = 1, y = 1
```

Part 8: Animations with Pygame

Pygame and SDL

- SDL = Simple DirectMedia Layer
- an easy-to-use library for interactive media
  - 2D graphics and animation
  - audio output
  - input handling
  - can be combined with OpenGL
- not a fancy engine for building complicated games

Pygame Components

- `from pygame import display, draw, time, event`
  - display – putting your graphics on the screen
  - draw – simple 2D shapes (lines, circles, etc.)
  - time – clocks and frame rate
  - event – reacting to user inputs
- there's a lot more, but we won't need it today
  - see `https://pygame.org` for docs

Setting up Graphics

- `screen = display.set_mode( [800, 600] )`
  - screen is a surface that you can draw on
  - double buffered by default
- `display.flip()` swaps the 2 buffers
- everything else is done via the surface (`screen`)

Drawing Things

- `screen.fill(colour)` clears the surface
  - `screen` always refers to the off-screen buffer
- `draw.line`, `draw.circle`, etc.
  - the first parameter is the surface
  - will be just `screen` in our case
  - then colour and geometry parameters

Example

```python
screen = display.set_mode([800, 600])
screen.fill([0, 0, 0])
colour = [200, 100, 100]
center = [400, 300]
radius = 50
draw.circle(screen, colour, center, radius, 1)
display.flip()
time.wait(2000) # milliseconds
```

Animation and Time

- repeat forever
  - draw a picture in the off-screen buffer
  - wait until the next frame should be shown
  - flip buffers
- use a timer for the wait
  - `clock = time.Clock()`
  - `clock.tick(60)`
  - this caps the frame rate at 60 frames per second

Events

- keyboard, mouse, joysticks and so on
- events are queued by the library
- `ev = event.poll()` gives you the next event
  - `ev.type` is `pygame.NOEVENT` if the queue is empty
  - `pygame.KEYDOWN` tells you a key was pressed

Exercise 8: Raindrops / Bubbles

- do a screen-saver-like animation at 60 fps
- draw expanding circles on the screen
  - put them in random locations
  - expansion at 1 pixel per frame works nicely
- remove circles when they get too big
- add new circles as old ones disappear
  - 1 new circle per frame works
- quit when the user hits a key

Part 9: Testing and Debugging

The Python Debugger

- run as `python -m pdb program.py`
- there's a built-in `help` command
- `next` steps through the program
- `break` to set a breakpoint
- `cont` to run until end or a breakpoint

Writing Unit Tests

- `from unittest import TestCase`
- derive your test class from `TestCase`
- put test code into methods named `test_*`
- run with `python -m unittest program.py`
  - add `-v` for more verbose output

Unit Test Example

```
from unittest import TestCase

class TestArith(TestCase):
    def test_add(self):
        self.assertEqual(1, 4 - 3)
    def test_leq(self):
        self.assertTrue(3 <= 2 * 3)
```

Exercise 9: Preliminaries

- suppose an n-dimensional space
- $n + 1$ points determine an n-simplex
  - 3 points determine a 2D triangle
  - 4 points determine a 3D tetrahedron
- pick one of the points
  - subtract it from the others to get n vectors
  - put the vectors into columns of an $n \times n$ matrix
- volume of an n-parallelotope is the determinant
  - divide by $n!$ to get n-simplex volume

Exercise 9: Part 1

- start with `def volume(*args): pass`
- write unit tests for this `volume` function
- check with a few obvious examples
  - it's enough to check in 2D and 3D
  - do at least 4–5 distinct cases
- also check behaviour for invalid inputs
  - mismatched number of points vs dimension
  - extraneous components in points

Exercise 9: Part 2

- actually implement the `volume` function
  - take variable number of parameters
  - for `def f(*args)`, `args` is a tuple
- compute n-simplex volume
  - only makes sense for 3 or more points
  - use `numpy` to do the math
- check that the unit tests pass