Python advanced class

Module 7, The filesystem, file formats, and database access

- Pathlib
- JSON, CSV, TOML og YAML
- SQLite

Pathlib

Pathlib and the Path object

- Pathlib is the modern way to work with directories, file patterns, and paths
- Pathlib is fully object oriented, and provides clean ways of iterating, testing, and convert to and from strings
- A path to a file or directory is representedby the `Path` object:

In [1]:
```
from pathlib import Path
curdir = Path('.').absolute()
parentdir = curdir.parent
childdir = curdir / '01_examples'
print(curdir, parentdir, childdir)
```

```
/home/tabac/work/python_v /home/tabac/work /home/tabac/work/pyth
on_v/01_examples
```

Working with directories

- Pathlib makes easy to work with directories and directory trees
- `for f in curdir.iterdir(): ...` makes it easy to iterate through a directory
- `curdir.glob('*.py')` finds all files that matches a wildcard string in directory
- `curdir.rglob('*.py')` finds files recursively decending into subdirs

In [2]:
```
for f in curdir.iterdir():
    if f.is_dir() and 'examples' in f.name:
        print(f.name, end=' ')
print()
subdir = curdir / '01_examples'
python_files = [f.name for f in subdir.glob('*.py')]
print(python_files)
```

```
05_examples 07_examples 02_examples 04_examples 03_examples 06_e
xamples 08_examples 01_examples
['findargp.py', 'argparse_ex.py', 'findpython.py', 'findany.py']
```

Accessing file properties

- A file or directory has a number of properties that can be examined:
    - Type: `.is_dir()`, `.is_file()`, `.is_symlink()`
    - Owner and group: `.owner()`, `.group()`
    - Operating system dependent data: `.stat()`
    - Name without path: `.name`, extension: `.suffix`

In [3]:
```
f = curdir / 'exercises.md'
print(f"Is a dir: {f.is_dir()}, Owner: {f.owner()}, Permissions: {f.sta
```

```
Is a dir: False, Owner: tabac, Permissions: 100755
```

Opening files

- Except for the name `.open(...)` works the same as the builtin function

In [4]:
```
f = curdir / 'exercises.md'
with f.open('r', encoding='utf-8') as exer_f:
    line = exer_f.readline().rstrip()
    print(line)
```

```
 # Chapter 1
```

Moving and deleting files

- Pathlib also allows for creating and moving directories and files:
    - `.mkdir()` to create directory
    - `.rename('newname')` to move a file
    - `.unlink()` to remove a file

In [5]:
```
tempdir = curdir / 'tempdir'
if not tempdir.exists():
    tempdir.mkdir()
tempfile = tempdir / 'tempfile.txt'
tempfile.write_text("This is the content")
print(tempfile.read_text())
tempfile.unlink()
tempdir.rmdir()
print(tempdir.exists())
```

```
This is the content
False
```

JSON, CSV, TOML og YAML

Reading and writing JSON data

- The `json` module provides functions to read and write JSON files through `load()` and `dump()`
- JSON can't represent `set` type, and it can't differentiate between `list` and `tuple`
- If `dump()` has an `indent=` argument the output will be human readable

In [6]:
```
import json

data1 = [[1, 2, 3], {'Fred': 56, 'John': 49}, (True, False, None)]
with open("data.json", "w", encoding="utf-8") as jsonfile:
    json.dump(data1, jsonfile, indent=4)
with open("data.json", "r", encoding="utf-8") as jsonfile:
    data2 = json.load(jsonfile)
print(data2)
```

```
[[1, 2, 3], {'Fred': 56, 'John': 49}, [True, False, None]]
```

Reading and writing CSV data

- CSV is an apparently simple dataformat, but it is not that easy to parse
- The `csv` module takes care of quoting and multiline fields

In [7]:
```
import csv
data1 = [('Fred', 56, 'New York'), ('John', 49, 'London'), ('Peter', 71
with open('data.csv', 'w', encoding='utf-8', newline='') as csvfile:
    wr = csv.writer(csvfile, dialect=csv.excel, quoting=csv.QUOTE_STRIN
    wr.writerow( ('Name', 'Age', 'Hometown') )
    wr.writerows(data1)
with open('data.csv', 'r', encoding='utf-8', newline='') as csvfile:
    rd = csv.reader(csvfile, dialect=csv.excel)
    for row in rd:
        print(row)
```

```
['Name', 'Age', 'Hometown']
['Fred', '56', 'New York']
['John', '49', 'London']
['Peter', '71', 'Paris']
```

Pickle, TOML, YAML

- The `dump()` and `load()` from JSON is common among other formats too
- The `pickle` module provides a Python-specific but fast and inclusive datadump format
- The `tomllib` module in the standard library provides a `load()` function. Writing needs the "Tomli-W" package installed
- The third-party 'pyyaml' package provides a `yaml` module also with a `dump()` and `load()` function

In [8]:
```
import pickle

data1 = [[1, 2, 3], {'Fred': 56, 'John': 49}, (True, False, None)]
with open("data.pickle", "wb") as picklefile:
    pickle.dump(data1, picklefile)
with open("data.pickle", "rb") as picklefile:
    data2 = pickle.load(picklefile)
print(data2)
```

```
[[1, 2, 3], {'Fred': 56, 'John': 49}, (True, False, None)]
```

SQLite

What is SQLite

- SQLite3 is a simple implementation of a datastorage with SQL interface
- SQLite3 implements most of SQL language, but has no GRANT (for obvious reasons)
- It stores its data in a single file
- But all techniques in the `sqlite3` module is identical to MS SQL Server, Oracle, PostgreSQL

Connecting and the cursor

- Before doing any operation, we must connect (open) to the database (which is just a path to a file)
- After obtaining the connection, we need a cursor to perform queries with the database

In [9]:
```
import sqlite3

conn = sqlite3.connect('mydata.sqlite3')
cursor = conn.cursor()
```

Executing SQL statements

- With the cursor we can perform queries with the `cursor.execute(sqltext)` method

In [10]:
```
sqltext1 = 'DROP TABLE IF EXISTS mydata'
sqltext2 = 'CREATE TABLE IF NOT EXISTS mydata (name VARCHAR(40), age IN
cursor.execute(sqltext1)
cursor.execute(sqltext2)
```

Out[10]:
```
<sqlite3.Cursor at 0x7e61348c8a40>
```

Executing SQL with placeholders

- Because SQL injection is always a risk, data should be processed through placeholders
- A placeholder is a questionmark in the SQL query to be filled
- Placeholder data is provided with the `cursor.execute()` method

In [11]:
```
sqltext3 = 'INSERT INTO mydata (name, age, hometown) VALUES(?, ?, ?)'
data1 = [('Fred', 56, 'New York'), ('John', 49, 'London'), ('Peter', 71
for row in data1:
    cursor.execute(sqltext3, row)
```

Reading data from SELECT statements

- Queries that has data return, 'SELECT', can use the cursor as an iterable

In [12]:
```
sqltext4 = 'SELECT name, age, hometown FROM mydata ORDER BY hometown'
cursor.execute(sqltext4)
for row in cursor:
    print(row)
```

```
('John', 49, 'London')
('Fred', 56, 'New York')
('Peter', 71, 'Paris')
```

Commit or Rollback

- Commit and rollback is issued with the connection object
- After the database operations are finish the cursor and the connection should be closed

In [13]:
```
conn.commit()
cursor.close()
cursor.close()
```