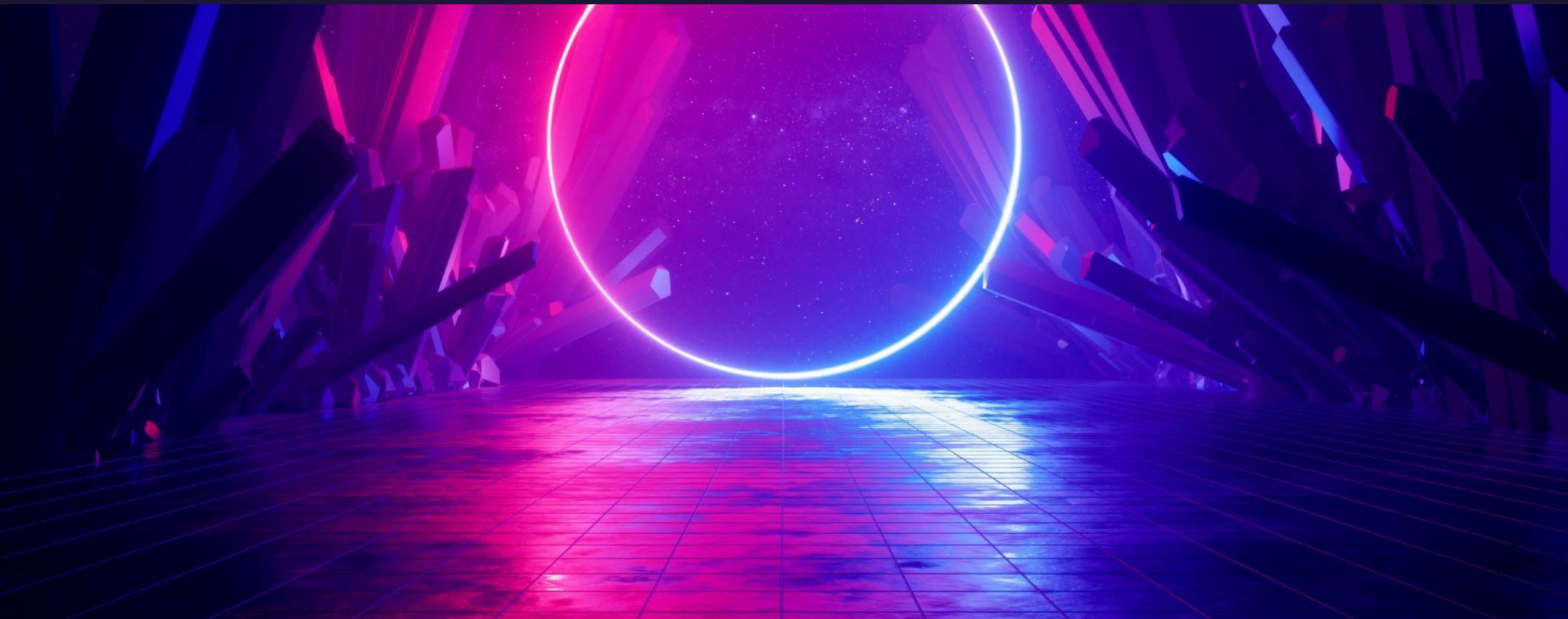
Programowanie w Pythonie

Łukasz Mioduszewski, UKSW 2022

sqlite3, template, threading, collections, logging



Czemu warto zamykać pliki

- Python może naraz otworzyć tylko około 10 000 plików
- Jeśli coś zepsuje się w trakcie pisania, plik może zostać uszkodzony
- Otwarcie pliku (w zależności od OS) blokuje go dla innych procesów, może być niemożliwy do usunięcia albo dostać niewłaściwe uprawnienia
- CPython ma garbage collector który m.in. zamyka nieużywane pliki, ale nie każda implementacja Pythona tak ma

Biblioteka sqlite3 - wprowadzenie

- Biblioteka sqlite3 w prosty sposób pracuje z bazami SQLite – przykład z filmami:

```
import sqlite3
con = sqlite3.connect("tutorial.db") # połączenie z bazą na dysku
cur = con.cursor() # tworzy kursor dla danego połączenia
cur.execute("CREATE TABLE movie(title, year, score)") # wykonuje polecenia SQL
res = cur.execute("SELECT name FROM sqlite_master")
res.fetchone() # zwraca pole z 1 rekordu z tabeli, w tym przypadku z sqlite_master
('movie',)          # zwraca w postaci tupli
res = cur.execute("SELECT name FROM sqlite_master WHERE name='spam'")
res.fetchone() is None # albo None jeśli żadnego takiego rekordu nie ma
True
```

Biblioteka sqlite3 - wprowadzenie

- Biblioteka sqlite3 w prosty sposób pracuje z bazami SQLite – przykład z filmami:

```
import sqlite3
con = sqlite3.connect("tutorial.db") # połączenie z bazą na dysku
cur = con.cursor() # tworzy kursor dla danego połączenia
cur.execute("CREATE TABLE movie(title, year, score)") # wykonuje polecenia SQL
cur.execute("""
    INSERT INTO movie VALUES
        ('Monty Python and the Holy Grail', 1975, 8.2),
        ('And Now for Something Completely Different', 1971, 7.5)
""")  # to jeszcze nie zapisuje nic do pliku tutorial.db
con.commit() # dopiero commit zapisuje do pliku
```

Biblioteka sqlite3 - wprowadzenie

- Biblioteka sqlite3 w prosty sposób pracuje z bazami SQLite – przykład z filmami:

```
# ... ciąg dalszy
res = cur.execute("SELECT score FROM movie")
res.fetchall() # fetchall zwraca pola ze wszystkichj rekordów w postaci listy tupli
[(8.2,), (7.5,)] # oceny dwóch filmów które są w bazie
data = [
    ("Monty Python Live at the Hollywood Bowl", 1982, 7.9),
    ("Monty Python's The Meaning of Life", 1983, 7.5),
    ("Monty Python's Life of Brian", 1979, 8.0),
] # tworzę listę tupli, można je przekształcić na rekordy
cur.executemany("INSERT INTO movie VALUES(?, ?, ?)", data) # placeholdery
con.commit()  # Remember to commit the transaction after executing INSERT
```

Biblioteka sqlite3 - wprowadzenie

- Biblioteka sqlite3 w prosty sposób pracuje z bazami SQLite – przykład z filmami:

```
# ... ciąg dalszy
for row in cur.execute("SELECT year, title FROM movie ORDER BY year"):
    print(row) # to co zwraca
(1971, 'And Now for Something Completely Different')
(1975, 'Monty Python and the Holy Grail')
(1979, "Monty Python's Life of Brian")
(1982, 'Monty Python Live at the Hollywood Bowl')
(1983, "Monty Python's The Meaning of Life")
```

Biblioteka sqlite3 - wprowadzenie

- Biblioteka sqlite3 w prosty sposób pracuje z bazami SQLite – przykład z filmami:

```
# ... ciąg dalszy
con.close() # możemy zamknąć połączenie i otworzyć nowe
new_con = sqlite3.connect("tutorial.db")
new_cur = new_con.cursor()
res = new_cur.execute("SELECT title, year FROM movie ORDER BY score DESC")
title, year = res.fetchone()
print(f'The highest scoring Monty Python movie is {title!r}, released in {year}')
The highest scoring Monty Python movie is 'Monty Python and the Holy Grail',
released in 1975
```

Placeholdery

- W zadaniu z wyrażeniami regularnymi dane od użytkownika można było włączyć do stringa przez zwykłą konkatenację, np. "\w{" + str(x) + ",}"
- Przy pracy z bazami danych nie robimy tak, ze względu na SQLinjection (formatowane stringi też odpodają)

```
# Never do this -- insecure!
symbol = 'RHAT'
cur.execute("SELECT * FROM stock WHERE symbol = '%s'" % symbol)
```

- Zamiast tego stosuje się placeholdery

Placeholdery

- Funkcja execute(sql, parameters=(), /) ma jeden obowiązkowy parametr (pojedyncze polecenie sql) i jeden nieobowiązkowy: parametry
- Funkcja executemany(sql, parameters, /) ma 2 obowiązkowe argumenty i wywołuje polecenie sql **wiele razy** dla każdego elementu z **listy** parameters
- Jeśli używamy ? jako placeholderów, parameters (albo element z listy parameters) musi być sekwencją, której długość odpowiada liczbie ?
- Jeśli placeholdery mają nazwy, argument parameters może być słownikiem
- / oznacza koniec argumentów wyłącznie pozycyjnych (wykład 2, slajd 20)

Placeholdery - przykład

```
con = sqlite3.connect(":memory:")
cur = con.execute("CREATE TABLE lang(name, first_appeared)")

# This is the qmark style:
cur.execute("INSERT INTO lang VALUES(?, ?)", ("C", 1972))

# The qmark style used with executemany():
lang_list = [
    ("Fortran", 1957),
    ("Python", 1991),
    ("Go", 2009),
]
cur.executemany("INSERT INTO lang VALUES(?, ?)", lang_list)

# And this is the named style:
cur.execute("SELECT * FROM lang WHERE first_appeared = :year", {"year": 1972})
print(cur.fetchall())
```

Funkcja connect

- sqlite3.connect(database, timeout=5.0, detect_types=0, isolation_level='DEFERRED', check_same_thread=True, factory=sqlite3.Connection, cached_statements=128, uri=False)
- database to albo ścieżka do bazy, albo ":memory:" dla bazy danych w RAMie
- konwersja typów niewspieranych przez SQLite do Pythona jest domyślnie wyłączona
- isolation_level mówi o tym kiedy są wykonywane transakcje (w chwili commit())
- domyślnie tylko dany wątek używa połączenia, aby nie zniszczyć bazy
- poprzez URI można przekazywać parametry do bazy danych

Różne funkcje biblioteki sqlite3

- complete_statement(statement) sprawdza czy polecenie jest poprawne
- register_adapter(type, adapter, /) używa funkcji adapter do przekształcania zmiennych na typ który SQLite rozumie

Python type	SQLite type
`None`	`NULL`
`int`	`INTEGER`
`float`	`REAL`
`str`	`TEXT`
`bytes`	`BLOB`

SQLite type	Python type
`NULL`	`None`
`INTEGER`	`int`
`REAL`	`float`
`TEXT`	depends on `text_factory`, `str` by default
`BLOB`	`bytes`

Różne funkcje w klasie connection()

- rollback() wraca do stanu sprzed (odwrotnie do commit() które zatwierdza)
- executescript(sql_script, /) zwraca nowy kursor na którym od razu został wywołany skrypt sql_script
- create_function(name, narg, func, *, deterministic=False) tworzy funkcję:

```
>>> import hashlib
>>> def md5sum(t):
...     return hashlib.md5(t).hexdigest()
>>> con = sqlite3.connect(":memory:")
>>> con.create_function("md5", 1, md5sum)
>>> for row in con.execute("SELECT md5(?)", (b"foo",)):
...     print(row)
('acbd18db4cc2f85cedef654fccc4a4d8',)
```

Różne funkcje w klasie connection()

- Nie trzeba jawnie tworzyć kursorów (ale pamiętajmy o con.close()):

```
# Create and fill the table.
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(name, first_appeared)")
data = [
    ("C++", 1985),
    ("Objective-C", 1984),
]
con.executemany("INSERT INTO lang(name, first_appeared) VALUES(?, ?)", data)

# Print the table contents
for row in con.execute("SELECT name, first_appeared FROM lang"):
    print(row)

print("I just deleted", con.execute("DELETE FROM lang").rowcount, "rows")
```

Różne funkcje w klasie cursor()

- fetchone() zwraca pola z jednego rekordu w postaci tupli (kolejne wywołania zwrócą pola z kolejnych rekordów), fetchmany(size=cursor.arraysize) zwraca pola z **kolejnych** size rekordów, fetchall() zwraca ze wszystkich **pozostałych**
- close() zamyka kursor na stałe
- Na następnym slajdzie trudny przykład konwersji…

```
import sqlite3
import datetime

con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES|sqlite3.PARSE_COLNAMES)
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print(today, "=>", row[0], type(row[0]))
print(now, "=>", row[1], type(row[1]))

cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"')
row = cur.fetchone()
print("current_date", row[0], type(row[0]))
print("current_timestamp", row[1], type(row[1]))

con.close()
```

Używanie connection razem z with i try

```
con = sqlite3.connect(":memory:")
con.execute("CREATE TABLE lang(id INTEGER PRIMARY KEY, name VARCHAR UNIQUE)")

# Successful, con.commit() is called automatically afterwards
with con:
    con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))

# con.rollback() is called after the with block finishes with an exception,
# the exception is still raised and must be caught
try:
    with con:
        con.execute("INSERT INTO lang(name) VALUES(?)", ("Python",))
except sqlite3.IntegrityError:
    print("couldn't add Python twice")

# Connection object used as context manager only commits or rollbacks transactions,
# so the connection object should be closed manually
con.close()
```

Uwagi o sqlite

- Dowolnie dużo procesów może odczytywać bazę
- Jeśli jakiś proces zapisuje do bazy, jest ona blokowana na czas zapisu
- Można tego uniknąć używając Write-Ahead Logging: https://www.sqlite.org/wal.html

Jeszcze więcej formatowania stringów

- from string import Template pozwala tworzyć własne formaty (wciąż podatne na SQLinjection), w których placeholderem jest ${nazwa}, $$ tworzy znak '$'

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

- safe_substitute pozwala pracować na niekompletnych danych

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Jeszcze więcej formatowania stringów

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
...
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f

>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Wielowątkowość

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')
```

Biblioteka collections – kolejka dequeue

- dequeue to double-ended queue, można dodawać i usuwać elementy z obu stron
- deque([iterable[, maxlen]]) tworzy nową kolejkę z opcjonalnego obiektu iterable (wywołując dla każdego obiektu z iterable metodę append(kolejnyelement)) Jeśli maxlen jest użyte, kolejka ma maksymalną długośc maxlen (elementy nadmiarowe będą usuwane od najstarszych)

```
def tail(filename, n=10):
    'Return the last n lines of a file'
    with open(filename) as f:
        return deque(f, n)
```

Metody kolejki dequeue

- count(x) liczy ile elementów ma wartość x, index(x) zwraca pierwszy indeks x
- extend(iterable) i extendleft(iterable) dodają do kolejki elementy z iterable odpowiednio z prawej i lewej strony
- append(x) i appendleft(x) dodają x, a pop() i popleft() dodają lub usuwają element odpowiednio z prawej lub z lewej
- reverse() odwraca kolejność kolejki, rotate(n=1) przesuwa n elementów z prawego końca na lewy początek (albo odwrotnie, jeśli n jest ujemne)
- remove(value) usuwa pierwsze wystąpienie value, clear() czyści kolejkę

Przykład użycia kolejki dequeue

```
def moving_average(iterable, n=3):
    # moving_average([40, 30, 50, 46, 39, 44]) --> 40.0 42.0 45.0 43.0
    # https://en.wikipedia.org/wiki/Moving_average
    it = iter(iterable)
    d = deque(itertools.islice(it, n-1))
    d.appendleft(0)
    s = sum(d)
    for elem in it:
        s += elem - d.popleft()
        d.append(elem)
        yield s / n
```

Biblioteka collections

- Counter to taki słownik który łatwo liczy ile czego w nim jest

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
...     cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})

>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
 ('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

Biblioteka logging

- Domyślnie komunikaty wysyłane są do sys.stderr, ale można do pliku

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

- Zostanie wypisane to:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

Bibliografia

- https://realpython.com/why-close-file-python/#:~:text=You've%20learned%20why%20it's,handles%20or%20experiencing%20corrupted%20data