

COMP3311 24T1

Database Systems

week 7 - 1



Outline



- **Announcements**
- Roadmap
- Python
- Psycopg2



Announcement 1



- **Assignment - 2**

- specifications will be released later this week
- due date: TBD (you will be given > 2 weeks)
- practicing pyscopg2 (to be learned today)
- will be using Python!

- **Do it earlier!**

- ask questions on the forum earlier
 - staffs
 - helpful peers
- setup guide is released earlier than the questions
 - try to set things up earlier



- **Use private posts to post your assignment code!**

- only staff members can view private posts.

Announcement 2



- **Quiz 4**
 - Due on this Friday (29 Mar)
 - Just do it!



Announcement 3



- **Help Session**
 - Thursday session to be shifted to 11:00 - 13:00 (G05)



Outline



- Announcements
- **Roadmap**
- Python
- Psycopg2



Roadmap



- Using PL to interact with RDBMS (postgres) - Today
- Relational design theory
- Relational algebra
- Transactions, Concurrency Control
- Future of DB, Course Review, Exam Preview, Guest Lecture ...



Outline



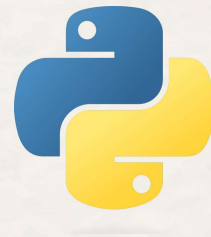
- Announcements
- Roadmap
- **Python**
- Psycopg2



Python

Python is a very popular programming language

- easy to learn/use
- with a wide range of useful libraries



We assume that you know enough Python basics.

- The primary topic is Database, not Python programming ...

If you are not overly familiar with Python...

- there will be many examples of Python code in this course
- there are many excellent tutorials online (like learnpython.org)



Why Python here?









Python is a very popular programming language



TIOBE Index for March 2024

March Headline: The gap between Python and the rest has never been that large

February has been a very quiet month for the TIOBE index. The only interesting notes are that Python is now 4.5% ahead of the rest, Scratch reentered the top 10, and Rust keeps climbing. --Paul Jansen CEO TIOBE Software

Mar 2024	Mar 2023	Change	Programming Language		Ratings	Change
1	1			Python	15.63%	+0.80%
2	2			C	11.17%	-3.56%
3	4	▲		C++	10.70%	-2.59%
4	3	▼		Java	8.95%	-4.61%
5	5			C#	7.54%	+0.37%
6	7	▲		JavaScript	3.38%	+1.21%
7	8	▲		SQL	1.92%	-0.04%
8	10	▲		Go	1.56%	+0.32%



Why Python here?

Python is easy to learn.

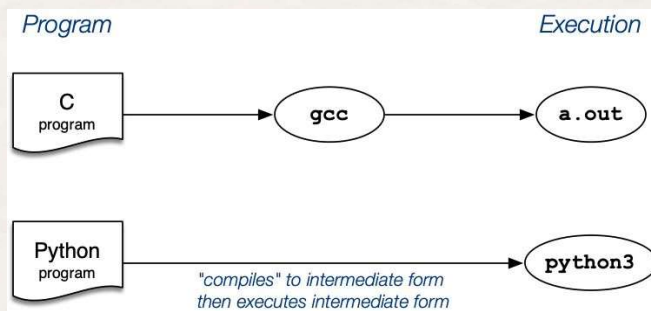
- Trust me, you won't want to use languages like Rust here in this course.

My personal view: Python is a good scripting language and a very nice language for larger projects.

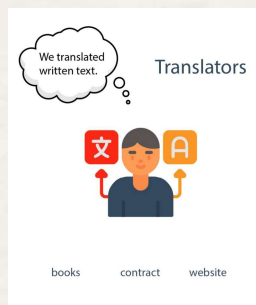


Python (interpreter vs compiler)

Python is an interpreted language.



Compiler	Interpreter
The compiler reads the entire program at once.	One statement at a time is translated.
Because it reads the code all at once, any errors (if any) are displayed at the conclusion.	Errors are displayed line by line since it reads code one line at a time.
The main benefit of compilers is their speed of execution.	It is less desirable due to interpreters' slowness in processing object code.
It translates source code to object code.	It examines the source code line by line rather than converting it to object code.
It does not necessitate the use of source code for later execution.	It necessitates source code to be executed later.
C, C++, C# etc.	Python, Ruby, Perl, SNOBOL, MATLAB, etc.



Python (execution)



Python has an interactive interface (like psql)

```
$ python3
Python 3.7.3 (default, Jul 25 2020, 13:03:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" ...
>> print("Hello, world")
Hello, world
>> quit()
$
```

Or you can run programs that are stored in files

```
$ echo 'print("Hello, world")' > hello.py
$ python3 hello.py
Hello, world
$
```



Python Basics

Like C, Python programs consist of

- expressions, statements, control structures, function definitions, imports, ...

The key characteristic: Python uses **indentation** to indicate code nesting

Python

```
if Condition:  
    Statements1  
else:  
    Statements2  
Next Statement
```

C

```
if (Condition ) {  
    Statements1  
} else {  
    Statements2  
}  
Next Statement
```



Modern IDEs can handle the conversion between tabs and spaces.



Python Basics (cont)



Comments are introduced by `#`, to end of line

Data types and constants:

- **booleans**, e.g. `True`, `False`
- **numbers**, e.g. `1`, `42`, `3.14`, `-5`
- **strings**, e.g. `"a string"`, `"string2"`, `'it\'s fun'`
- **lists**, e.g. `[1,4,9,16,25]`, `['a','b','c']`
- **tuples**, e.g. `(3,5)`, `(1,'a',3.0)`
- **dictionaries**, e.g. `{'a': 5, 'b': 98, 'c': 99}`

Assignment is via `=`, "the usual" operators are available



E

Python Operators and Expressions - example



```
name = "Giraffe"  
age = 18  
height = 2048.11 # mm
```

```
print(name + ", " + str(age) + ', ' + str(height))  
print(f"{name}, {age}, {height}")  
print(type(name))  
print(type(age))
```

```
n = 16 // 3  
print(f"3 ** 3 == {3 ** 3}")  
print(f"16 / 3 == {16 / 3}")  
print(f"16 // 3 == {n}")
```

Try it: [Python - OneCompiler - Write, run and share Python code online](#)



E

Python Function Declaration - example



```
# iterative factorial
def faci(n):
    f = 1
    for i in range(1,n):
        f = f * i
    return f
```

```
print('6! =' ,faci(6))
```

```
# recursive factorial
def facr(n):
    if n <= 1:
        return 1
    else:
        return n # Q: what to be put here?
```

```
print('5! =' ,facr(5))
```

Try it: [Python - OneCompiler - Write, run and share Python code online](#)



K

Python Basics (cont)

Python programs can import external modules (module = collection of definitions)

Packages (e.g. **psycopg2**) are collections of sub-packages and modules

```
import sys
import psycopg2
import sound.effects.echo
from sound.effects import echo
```



K

Python Basics (cont)

Python programs can take commandline arguments/options

- via **sys.argv**

```
import sys
```

```
# extracts a slice from argv
```

```
for i,arg in enumerate(sys.argv[1:]):
```

```
    print(f"{i}: {arg}")
```

Try it: online-python.com



Best practice: use the **argparse** module - <https://docs.python.org/3/library/argparse.html>



Outline

- Announcements
- Roadmap
- Python
- **Psycopg2**



K

Psycopg2

Psycopg2 is a Python module that provides

- a method to connect to PostgreSQL databases
- a collection of DB-related exceptions
- a collection of type and object constructors

In order to use Psycopg2 in a Python program

- `import psycopg2`

Need to install it first! (<https://pypi.org/project/psycopg2/>)



Better to install in a [virtual environment](#).

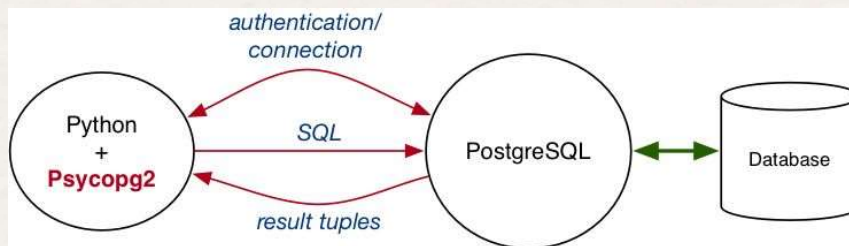
Already installed for you on vxdb2 for python3.

Although we are learning psycopg2, psycopg3 is available now. The design level differences are [here](#).



K

Psycopg2 - Workflow



Database connections



```
conn = psycopg2.connect(DB_connection_string)
```

- creates a **connection** object on a named database
- effectively starts a session with the database (cf **psql**)
- returns a **connection** object used to access DB
- if can't connect, raises an **exception**

DB connection string components

- **dbname** ... name of database
- **user** ... user name (for authentication)
- **password** ... user password (for authentication)
- **host** ... where is the server running (default=localhost)
- **port** ... which port is server listening on (default=5432)



E

Example: connecting to a database

Simple script that connects and then closes connection

```
import psycopg2

try:
    conn = psycopg2.connect("dbname=mydb")
    print(conn) # state of connection after opening
    conn.close()
    print(conn) # state of connection after closing
except Exception as e:
    print("Unable to connect to the database")
```



K

Operations on connections



```
cur = conn.cursor()
```

- set up a **handle** for performing queries/updates on database
- must create a **cursor** before performing any DB operations

```
conn.close()
```

- close the database connection **conn**

```
conn.commit()
```

- commit changes made to database since last **commit()**



many other operations ... here: <https://www.psycopg.org/docs/connection.html>

K

Database Cursors



Cursors are "pipelines" to the database

Cursor objects allow you to ...

- execute queries, perform updates, change meta-data

Cursors are created from a database **connection**

- can create **multiple** cursors from the same connection
- each cursor handles one DB operation at a time
- but cursors are not isolated (can see each others' changes)

To set up a **cursor** object called **cur** ...



```
cur = conn.cursor()
```

Operations on cursors



cur.execute(SQL_statement, Values)

- if supplied, insert values into the SQL statement
- then execute the SQL statement
- results are available via the cursor's **fetch** methods

Examples:

run a fixed query

```
cur.execute("select * from R where x = 1");
```

run a query with values inserted

```
cur.execute("select * from R where x = %s", (1,))
```

```
cur.execute("select * from R where x = %s", [1])
```

run a query stored in a variable

```
query = "select * from Students where name ilike %s"
```

```
pattern = "%mith%"
```

```
cur.execute(query, [pattern])
```



Operations on cursors (cont)



`ur.mogrify(SQL_statement, Values)`

- return the SQL statement as a string, with values inserted
- useful for checking whether **`execute()`** is doing what you want

Examples:

```
query = "select * from R where x = %s"  
print(cur.mogrify(query, [1]))
```

Produces: b'select * from R where x = 1'

```
query = "select * from R where x = %s and y = %s"  
print(cur.mogrify(query, [1,5]))
```

Produces: b'select * from R where x = 1 and y = 5'

```
query = "select * from Students where name ilike %s"  
pattern = "%mith%"  
print(cur.mogrify(query, [pattern]))
```

Produces: b'select * from Students where name ilike '%mith%'



K

Operations on cursors (cont)



```
list = cur.fetchall()
```

- gets all answers for a query and stores in a list of tuples
- can iterate through list of results using Python's **for**

Examples:

```
# table R contains (1,2), (2,1), ...
```

```
cur.execute("select * from R")
for tup in cur.fetchall():
    x,y = tup
    print(x,y) # or print(tup[0],tup[1])
```

```
# prints
```

```
1 2
2 1
...
```



Operations on cursors (cont)



```
tup = cur.fetchone()
```

- gets next result for a query and stores in a tuple
- can iterate through list of results using Python's **while**

Examples:

```
# table R contains (1,2), (2,1), ...
```

```
cur.execute("select * from R")
```

```
while True:
```

```
    t = cur.fetchone()
```

```
    if t == None:
```

```
        break
```

```
    xy = tup
```

```
    print(xy)
```

```
# prints
```

```
1 2
```

```
2 1
```

```
...
```



Operations on cursors (cont)



```
tup = cur.fetchmany(nTuples)
```

- gets next nTuples results for a query
- stores tuples in a list
- when no results left, returns empty list

Examples:

```
# table R contains (1,2), (2,1), ...
```

```
cur.execute("select * from R")
while True:
    tups = cur.fetchmany(3)
    if tups == []:
        break
    for tup in tups:
        x,y = tup
        print(x,y)
```

```
# prints
```

```
1 2
2 1
...
```



Python vs PostgreSQL data types



Strings:

- in Python: written with `"..."` or `'...'`, including `\x`
- converted to SQL strings e.g. `"O'Reilly"` → `'O"Reilly'`
- Python supports `""" """` multi-line strings (useful for SQL queries)

Tuples:

- in Python: contain multiple heterogeneous values (cf. C **struct**)
- similar to PostgreSQL **composite (tuple)** types
- written as: `(val1, val2, ..., valn)` (note that `(val1)` is not a tuple)
- examples: `(1,2,3)`, `(1,"John",3.14)`, `(1,)`



K

Calling PostgreSQL functions



Two ways to call PostgreSQL functions

- use **cursor.execute()**
- e.g., `cursor.execute("select * from yourfunction(5)")`

- use **cursor.callproc(functionname, values)** method
- parameters supplied as a list of values/vars
- e.g., `cursor.execute("yourfunction", [5])`



K

Some Psycopg2 Tricks



`cur.execute(SQL Statement)`

- clearly the SQL statement can be **SELECT**
- can also be **UPDATE** or **DELETE**
- can also be a meta-data statement, e.g.
 - **CREATE TABLE, DROP TABLE, CREATE VIEW, ...**

`cur.fetchmany(#tuples)`

- gets a list of the next **#tuples** tuples
- could replace PLpgSQL **LIMIT** in some contexts



K

Pitfalls of using Python+SQL



! For better performance, try to reduce the number of queries.

Example:

```
cur.execute("select x.y from R")
for tup in cur.fetchall():
    q = "select * from S where id=%s"
    cur.execute(q, [tup[0]])
    for t in cur.fetchall():
        ... process t ...
```

vs

```
qry = """
select *
from R join S on (R.x = S.id)
"""
cur.execute(qry)
for tup in cur.fetchall():
    ... process tup ...
```



Thank you!