

Introduction to programming using Python

Session 10

Matthieu Choplin

matthieu.choplin1@city.ac.uk

<http://moodle.city.ac.uk/>

Objectives

- Debugging
- Testing
- Introduction to Database Connection

Debugging

- What is the program supposed to do?
- Is it doing what it is expected to do?
- Why not? Investigate...

2 ways of debugging

- Naive debugging
 - Use the **print()** function, sometimes it is enough
- Smarter debugging
 - Use a debugger, i.e. **pdb** and insert a **breakpoint**
 - A breakpoint is an intentional stopping or pausing place in a program. It is also sometimes simply referred to as a pause.
 - You set it by writing the following within your program

```
import pdb; pdb.set_trace()
```

Commands for using pdb

- list (l) List 11 lines around the current line (five before and five after). Using list with a single numerical argument lists 11 lines around that line instead of the current line.
- next (n) Execute the next line in the file. This allows you to go line by line and inspect the state of the code at that point.
- continue (c) Exit out of the debugger but still execute the code.
- step into (s) to go into the execution call of an other function

To go further: <https://pymotw.com/3/pdb/>

Exercise: debug this code using a breakpoint or a print statement

```
import random

def sort_list(my_list):
    my_list = my_list.sort()
    return my_list

if __name__ == '__main__':
    # create and shuffle a list
    my_list = list(range(9))
    random.shuffle(my_list)

    # sort the list
    my_list = sort_list(my_list)
    print(my_list) # [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Testing

- You want your program to be tested automatically
- Each time you change something in your program, there is a risk to break it
- So, you should test systematically.
- Different types of testing: unit testing, integration testing, exploratory testing...

Unit testing

- Unit testing consist of testing the smallest part of your application, usually a function or a class
- To do do that in python, you can use the **unittest** library

Example (1)

We have a function `sum_two_numbers(a, b)` that takes 2 numbers in arguments and returns their sum

```
# simple_function.py  
def sum_two_numbers(a, b):  
    return a + b
```

We call it like this, and put the result in a variable:

```
sum_nb = sum_two_numbers(1, 2)  
print(sum_nb)
```

How to test it automatically?

Example (2)

```
# test_simple_functions.py
import unittest
from simple_function import sum_two_numbers

class SimpleTestClass(unittest.TestCase):

    def test_sum_two_numbers(self):
        self.assertEqual(sum_two_numbers(1, 2), 3)

if __name__ == '__main__':
    unittest.main()
```

Example (3) - Explanation

- You can download the files there [simple_function.py](#) and [test_simple_functions.py](#)
- What is important when writing a test:
 - The name of the test file must start with **test_**, so that we understand that it is a test file
 - We import the unittest library and define a class inheriting from **unittest.TestCase**
 - The tests function are within such classes, not that the name of the test method must also begin with **test_**
 - Once you have executed your use case, you need to check if the result is correct with **assertion methods** (that are actually made available by the the parent class).

Exercise: Create a test

Create a test for the `sort_list()` of the 1st exercise

1. Create an appropriate test file
2. Import the **unittest** library
3. Import the **file** where the `sort_list()` function is defined
4. Create a class inheriting from **unittest.TestCase** with an appropriate name
5. Create a function that is going to test the `sort_list()` function with an **assert** method
6. At the end of the test file, write the following:

```
if __name__ == '__main__':  
    unittest.main()
```

Rules for good unit tests

- run completely by itself, without any human input. Unit testing is about automation.
- determine by itself whether the function it is testing has passed or failed, without a human interpreting the results
- run in isolation, separate from any other test cases (even if they test the same functions). Each test case is an island

NB: Pycharm can also help you write unittest: see [here](#)

Introduction to SQL Database Connection (1)

- You will use databases when you want to structure your data using tables and fields and **persist them in memory**.
- The data stored in database will remain even if you close your program.
- You can visualize what a database is by taking a Spreadsheet where the file will be the table, the header of the rows will be the fields or columns and the rest of the rows will be the data stored in this table
- The simplest version of SQL is SQLite, we can interact with it through the module **sqlite3**

Introduction to SQL Database Connection (2)

- We need to import the module and connect to a database (that will be created if it does not exist).

```
import sqlite3  
  
connection = sqlite3.connect("test_database.db")
```

NB: the data created will be stored in the file **test_database.db** that is actually the database

Introduction to SQL Database Connection (3)

We then need a **cursor** to execute commands on the database

```
import sqlite3

connection = sqlite3.connect("test_database.db")
cursor = connection.cursor()
# We create our first TABLE People that will
# store the field FirstName, LastName and Age
cursor.execute(
    "CREATE TABLE People("
    "FirstName TEXT, "
    "LastName TEXT, "
    "AGE INT)")
```


Introduction to SQL Database Connection (4)

Imagine that the TABLE we have created is like a spreadsheet file ready to take data

It means that we can now insert data into this table with the "INSERT" command

```
cursor.execute("INSERT INTO People "  
               "VALUES ('Ron', 'Obvious', 42)")  
# we have to commit to actually  
# save the record in database  
connection.commit()
```

Introduction to SQL Database Connection (5)

When working with database, it is a good idea to use the **with** keyword to simplify your code, similar to how we used the **with** to open files

```
with sqlite3.connect("test_database.db") as connection:  
    # perform any SQL operation
```

Also, you will no longer need to use the **commit()** explicitly

Introduction to SQL Database Connection (6)

Imagine that you want to concatenate a string with a SQL command

Do not do this:

```
first_name, last_name, age = 'John', 'Doe', 21
with sqlite3.connect("test_database.db") as connection:
    cursor = connection.cursor()
    cursor.execute(
        "INSERT INTO People VALUES"
        "(" + first_name + ", " + last_name + ", " +
```

Introduction to SQL Database Connection (7)

The database is correctly updated, you can check that with the following command

```
with sqlite3.connect("test_database.db") as connection:  
    cursor = connection.cursor()  
    cursor.execute("SELECT * FROM People")  
    rows = cursor.fetchall()  
    print(rows)
```

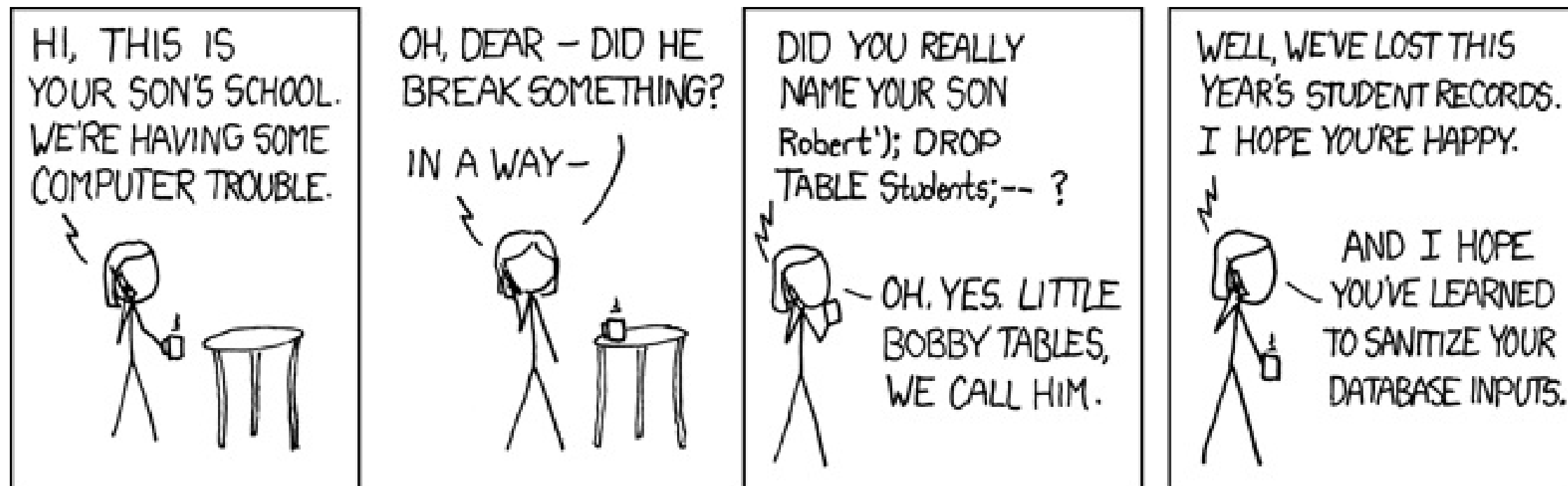
Introduction to SQL Database Connection (8)

Using the same method as 2 slides before, what happen if we try to add a user with the LastName "O'Connor"?

```
first_name, last_name, age = 'John', 'O\'Connor', 21
with sqlite3.connect("test_database.db") as connection:
    cursor = connection.cursor()
    cursor.execute(
        "INSERT INTO People VALUES"
        "(" + first_name + ", " + last_name + ", " +
```

We will get an error because the ""

Introduction to SQL Database Connection (9)



Introduction to SQL Database Connection (10)

To avoid SQL injection, use the following instead:

```
first_name, last_name, age = 'John', 'O\'Connor', 21
with sqlite3.connect("test_database.db") as connection:
    cursor = connection.cursor()
    cursor.execute(
        "INSERT INTO People VALUES"
        "(?, ?, ?)", (first_name, last_name, age))
    cursor.execute("SELECT * FROM People")
    rows = cursor.fetchall()
    print(rows)
```

Introduction to SQL Database Connection (11)

The question marks act as a placeholder for the (first_name, last_name, age) tuple; this is called a **parameterized statement**. You should always use parameterized SQL statements.

File used for the example: [test_db.py](#)

Exercise

- Populate the database with additional records
- Display the People who are older than 18 using a select command and a `cr.fetchall()`

What's next?

The best way to learn is by doing a python project

Look at the [popular python modules](#) and try to build something around them

You can also train yourself by doing some mathematical challenges on the [Project Euler](#) or the [Python Challenge](#) website.

You can host your project on Github or Bitbucket and [learn the git command](#)