Introduction to programming with Python

Session 10

Objectives

- Introduction to Database Connection
- Debugging
- Using a virtual environment
- Testing
- Understand the purpose of using Version Control Systems

Introduction to SQL Database Connection (1)

- Databases are used to store data in a structured format. Data stored in a database will remain stored on the hard drive even if you quit the application.
- A simplified view of what a database is, is to compare it to a spreadsheet.
  - The spreadsheet will be the table.
  - The first row where you declare headers of rows will be the fields (or columns)
  - The data in each row will represent a record of data stored in the table

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 databases, 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 to create an 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)
```

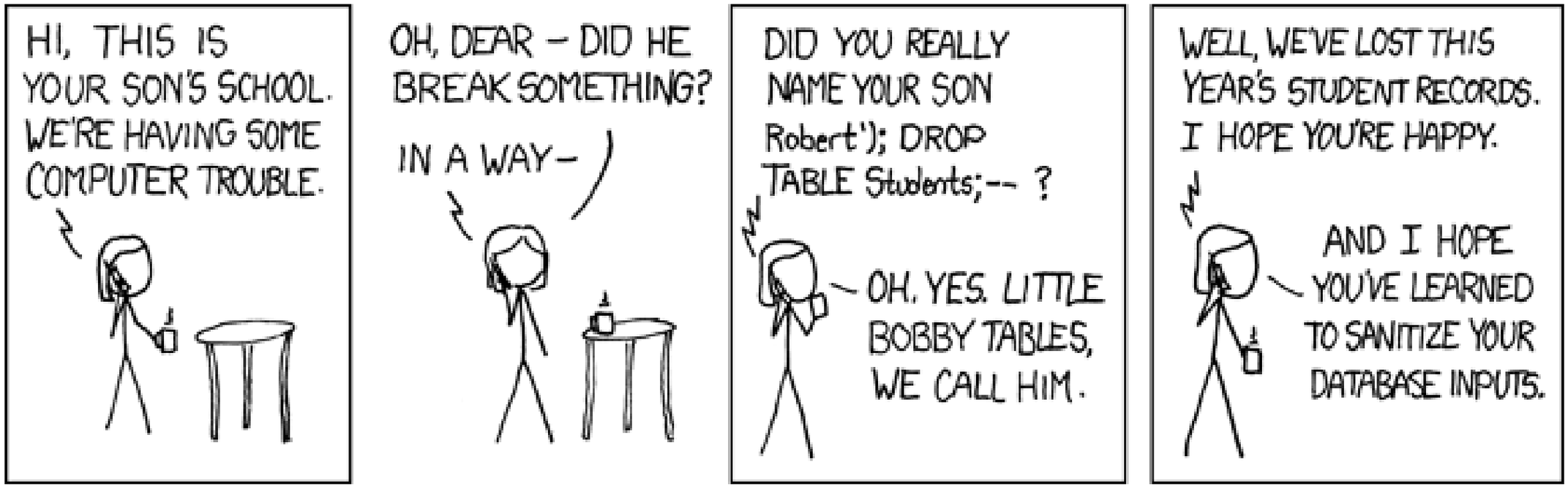
Introduction to SQL Database Connection (8)

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 placeholders for the (first_name, last_name, age) tuple; this is called a **parameterised statement**. You should always used parameterised 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()

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]
```

Using a virtual environment

- Some applications use a complete dedicated machine to be installed and run
- But you may want to run different python versions with different libraries on the same machine
- From Python3.6, you can use:

```
python3 -m venv /path/to/new/virtual/environment
```

Using a virtual environment: Note

- Before Python3.6, you could use **pyvenv**:

```
pyvenv /path/to/new/virtual/environment
```

- Before pyvenv, we would use an external library called **virtualenv**. If you are working with python2, this is what you should use
- More on virtual environment: https://docs.python.org/3.7/library/venv.html

Virtualenv for Python2 programs

- If not installed, install it with pip

```
pip install virtualenv
```

- Create the virtual environment

```
virtualenv -p /path/to/python2.7 venv
```

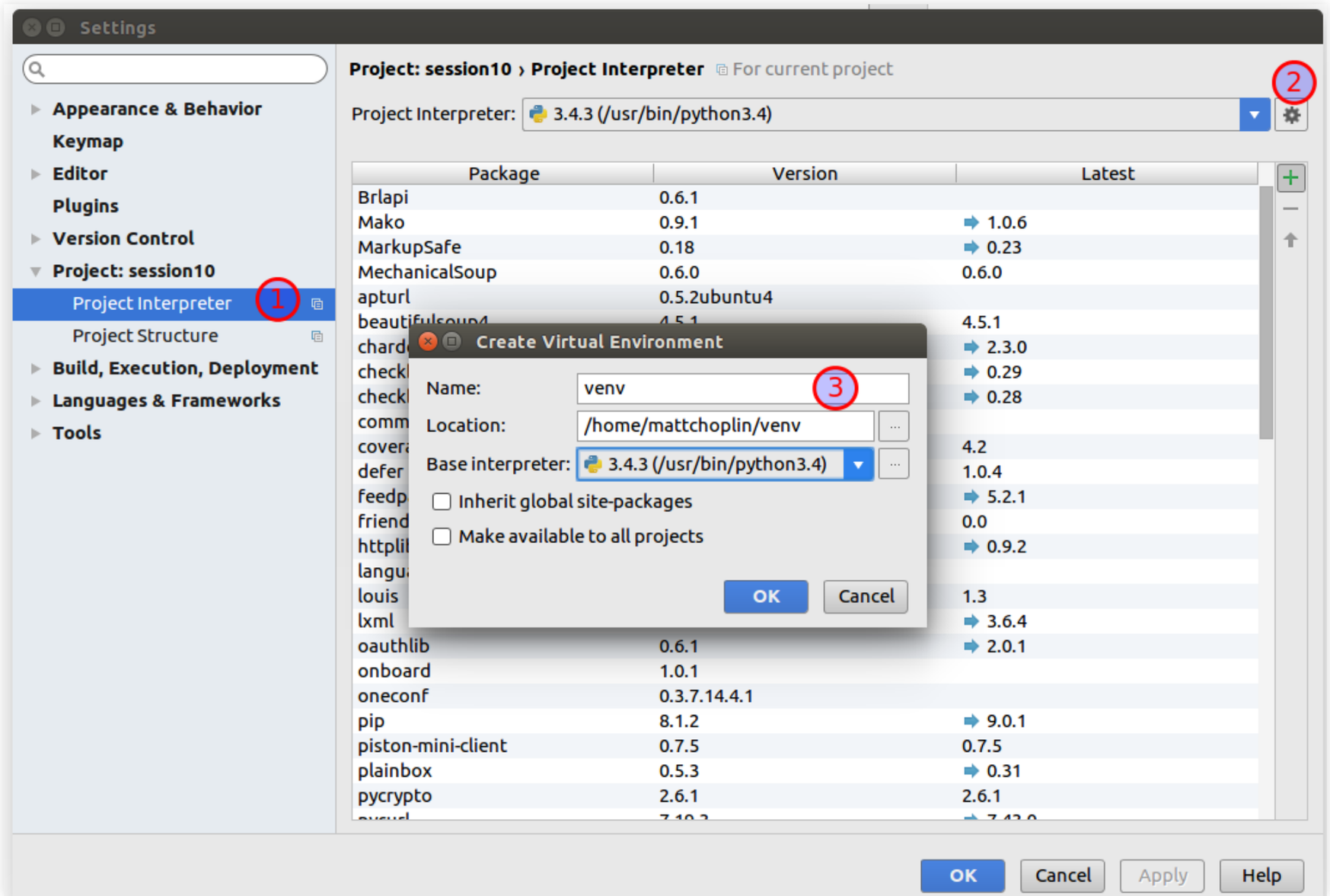
- Activate it

```
source venv/bin/activate
```

- Install the required libraries

```
pip install openpyxl
```

Virtualenv in Pycharm



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 consists of testing the smallest part of your application, usually a function or a class
- To 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 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 test functions are within classes of this type. Note 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 parent class).

Exercise: Create a test

Create a test for the sort_list() function of the first 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()
```

The setUp() method

Method called to prepare the test fixture. This is called immediately before calling the test method. The default implementation does nothing.

The tearDown() method

Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception. The default implementation does nothing.

The assert methods

Method	Checks that	New in
`assertEqual(a, b)`	`a == b`	
`assertNotEqual(a, b)`	`a != b`	
`assertTrue(x)`	`bool(x) is True`	
`assertFalse(x)`	`bool(x) is False`	
`assertIs(a, b)`	`a is b`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

- Many others: https://docs.python.org/3.6/library/unittest.html

Solution

session_10_debugging_solution.py

test_session_10_debugging_solution.py

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

Measuring your code coverage

You can use the module coverage

- Use **coverage run** to run your program and gather data:

```
$ coverage run my_program.py arg1 arg2
```

- Use **coverage report** to report on the results:

```
$ coverage report -m
```

- Also possible in Pycharm

Version Control System, why

- Keep track of changes happening in your project
- Experiment things and make changes with confidence, and revert when needed.
- Work in a team with files and directory structure that are consistent for all team members to allow communication
- Understand who made a change and why it happened

Version Control System, how

- Different tools exist but the most common used today is **Git** that you can use with Github.com or BitBucket.com
- Git is a decentralized VCS, as opposed to SVN and CVS (the previous generation of VCS)
- Example of a branching strategy: the Feature Branch Workflow

To go further on using a VCS

- See the different workflow strategies here: https://www.atlassian.com/git/tutorials/comparing-workflows/
- Learn Git

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 Git