QCOMPPW Intro: Programming Warm-up

Paul-Emile Morgades

Based on:
Qiskit textbook & numpy documentation

Practical Outline: During this practical we will quickly get familiar with Python and Qiskit

Contents

1 Introduction

To properly simulate quantum behaviours using a classical computer and Python 3, we are going to need two libraries. The first one is **Numpy**, providing numerical computing tools like linear algebra. The second one, **Qiskit** [kiss-kit], is an open source SDK for working with quantum computers at the level of pulses, circuits and algorithms.
Do not worry, this practical aims to be short and efficient. The main goal is to code simple functions, which are a pretext to packages installation. Please be thorough, and do not hesitate to ask for help.
Each practical is graded, and this one is a good opportunity to start properly while quantum computing might seem very straightforward. Of course, cheating and code sharing are not permitted. This rule applies to the upcoming sessions as well.

1.1 IDE

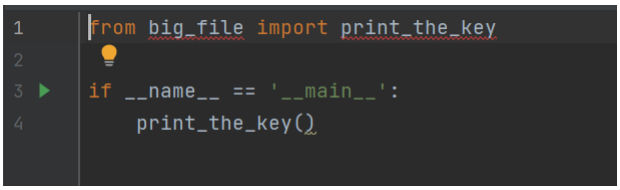
We recommend using Pycharm, it allows to easily download the package needed.

1.1.1 Installation

You can download Pycharm **here**. Both the professional and the community edition are good.

1.2 Submission

Your code will need to be **submitted using github**, on a repository from this **assignment**. The only files that matter to us are the given Python files named exercise_*.py, and these are the only files that you should 'git ad
Every practical has a deadline. Today's one is on **Sunday the 21st of november at 23:59**. Only the very last commit is taken into account.
The grade is determined by the script **run_test_suite.py**. To run it through Pycharm, you have to open it and click on the green arrow:



It will output something like this:

```
Your mark is:  0
Your proof of work is: 0,9967204840567662
```

You will have to put the proof of work in this **Excel file**. Your proof of work represent your mark, so update it every time you have a better mark. You also have to push your corresponding work on GitHub.

1.3 Given files

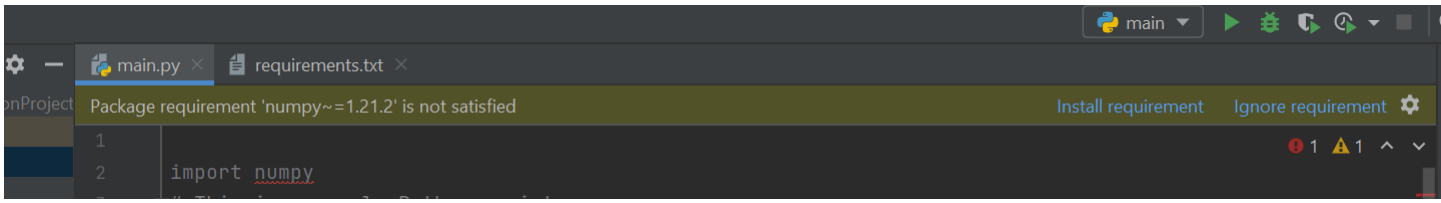
You can download the given files to complete all along the next exercises and the test suite: **here**. You have to download all the file.

1.4 Download Python's packages

Open your GitHub repository with Pycharm.

Then download the given file and put them in your repository.

Wait a little bit and click on "install requirement":



1.5 Imports

Through these exercises, you will be asked to use **numpy** and, at the end, **qiskit**. They are the only allowed imports.

```
import numpy as np
from qiskit import *
```

1.6 Documentation

numpy: https://numpy.org/doc/1.19/
qiskit: https://qiskit.org/documentation/

2 Numpy

Please mind 2.1. Numpy is one of the top-used Python library, especially among data scientists and computer engineers. These following exercise aim to show you the power of this tool. Even if some useful tips are given to you each time, do not hesitate to take a glimpse at the documentation.

All along these exercise, tricky cases (empty list, None, equal number of values...) won't be tested.

2.1 Exercise 1: Kronecker Product

2.1.1 Without loops (using numpy)

This function should return the Kronecker product between two matrices. **Using loops is... forbidden**. As always, think about numpy functions, that is what this practical is about. If you don't remember what the Kronecker product is, below is a quick recap. Further details are given is the textbook.
Given A a matrix $m * n$ and B a matrix $p * q$. The Kronecker product symbolized $A \otimes B$, of size $mp * nq$, is defined as such:

$$A \otimes B = \begin{pmatrix} a_{11}B & \cdots & a_{1n}B \\ \vdots & \ddots & \vdots \\ a_{m1}B & \cdots & a_{mn}B \end{pmatrix}$$

```
m1 = [5]
m2 = [7]
result = kroneckerProduct(m1, m2)
reference = [35]
assert np.array_equal(result, reference)

m1 = [[2]]
m2 = [[4, 5],
      [6, 7]]
result = kroneckerProduct(m1, m2)
reference = [[8, 10],
             [12, 14]]
assert np.array_equal(result, reference)

m1 = [[1, 2]]
m2 = [[4, 5],
      [6, 7]]
result = kroneckerProduct(m1, m2)
reference = [[4, 5, 8, 10],
             [6, 7, 12, 14]]
assert np.array_equal(result, reference)
```

2.2 Exercise 2: Iterative kronecker product

To do so, you will have to program the function **computeMatrix**. This function takes into input a matrix :2x2 **baseMatrix**, an integers : **nbQbit**, and an integer **fQbit**. This function output:

$$I_2^{\otimes(nbQbit-fQbit-1)} \otimes matrix \otimes I_2^{\otimes fQbit}$$

. Let's consider

$$I^{\otimes 3} = I \otimes I \otimes I$$

and

$$I^{\otimes 0} = \begin{bmatrix}1\end{bmatrix}$$

The function must pass this test:

```
m = [[5]]
    arr = [[5]]
    assert (np.array_equal(computeMatrix(m, 1, 0), arr))

m = [[5]]
    arr = [[5.0, 0],
    [0, 5.0]]
    assert (np.array_equal(computeMatrix(m, 2, 0), arr))

m = [[0,1],
     [1,0]]
arr = [[0., 1., 0., 0.],
 [1., 0., 0., 0.],
 [0., 0., 0., 1.],
 [0., 0., 1., 0.]]
assert(np.array_equal(computeMatrix(m,2,0), arr))
m = [[1,2],[3,4]]
arr = [[1., 0., 2., 0.],
 [0., 1., 0., 2.],
 [3., 0., 4., 0.],
 [0., 3., 0., 4.]]
assert(np.array_equal(computeMatrix(m,2,1), arr))

m = [[1,2],[3,4]]
arr = np.kron(np.kron(np.identity(2),m), np.identity(2))
assert(np.array_equal(computeMatrix(m,3,1), arr))
```

One way to implement it is to:

- create an array of **nbQbit** I_2 matrixes ([I_2,I_2,...,I_2])
- replace the **fQbit**th matrix by **baseMatrix**
- make a Kronecker product out of the values of the array. In the reverse orders !!!!!

Think to check the function **numpy.identity**.
Try to debug by looking at your algorithm's execution rather than looking at the output matrix.

3 Qiskit

Please mind 3.1. Qiskit accelerates the development of quantum applications by providing the complete set of tools needed for interacting with quantum systems and simulators. It is one of the preferred choices for quantum computing in Python.

3.1 Introduction to Qiskit

This exercice will be pretty straightforward. We do not want you to get scared by quantum computing, and we will pretty much give you a hint for every single line of your code. So do not worry, and let's dive into it.

Please mind 3.2. Here are shortcuts link for this exercise:

1. *Single qubits gates*
2. And if you get the curiosity: *here is how to create more complex circuits*

To get more familiar about the Python library, here is a code sample for the **NOT** function. Try to understand it and, why not, run it.

```
from qiskit import *

def NOT(input):

    q = QuantumRegister(1) # a qubit in which to encode and manipulate the input
    c = ClassicalRegister(1) # a bit to store the output
    qc = QuantumCircuit(q, c) # this is where the quantum program goes

    # We encode '0' as the qubit state |0>, and '1' as |1>
    # Since the qubit is initially |0>, we don't need to do anything for an input of
    '0'
    # For an input of '1', we do an x to rotate the |0> to>|1>
    if input==1:
        qc.x( q[0] )

    # Now we've encoded the input, we can do a NOT on it using x
    qc.x( q[0] )

    # We extract the |0>/|1> output of the qubit and encode it in the bit c[0]
    qc.measure( q[0], c[0] )

    # We'll run the program on a simulator
    backend = Aer.get_backend('qasm_simulator')
    # Since the output will be deterministic, we can use just a single shot to get it
    job = execute(qc,backend,shots=1)
    output = next(iter(job.result().get_counts()))

    return output
```

3.2 Exercice 3: XOR

Your goal is to fill out the **XOR** function code sample.
Remember that, if $\{|0\rangle, |1\rangle\}$ are the only allowed input values for both qubits, then the TARGET output of the CNOT gate corresponds to the result of a classical XOR gate.
Here Qiskit's CNOT gate's documentation.

Before		After	
Control	**Target**	**Control**	**Target**
$\|0\rangle$	$\|0\rangle$	$\|0\rangle$	$\|0\rangle$
$\|0\rangle$	$\|1\rangle$	$\|0\rangle$	$\|1\rangle$
$\|1\rangle$	$\|0\rangle$	$\|1\rangle$	$\|1\rangle$
$\|1\rangle$	$\|1\rangle$	$\|1\rangle$	$\|0\rangle$

Figure 1: CNOT truth table