S1B2_PythonTutorial

February 20, 2022

1 Introducing our Tools

- Python primer
- First steps in Qiskit

1.1 Python Tutorial

- Very brief tutorial into Python
- We work with Jupyter Notebooks and use Anconda to manage our packages (convenient, similar across different OS)
- Python 3.x

[1]:
```python
# Hello world example

print("Hello World!")
```

```
Hello World!
```

[2]:
```python
2+2
```

[2]: 4

[3]:
```python
"Hello World!"
```

[3]: 'Hello World!'

[5]:
```python
# integer
s1=3;

type(s1)
```

[5]: int

[6]:
```python
# string
s2="Hello";

type(s2)
```

[6]: str

[7]:
```
# float
s3=3.1415;

type(s3)
```

[7]: float

[8]:
```
type(s1+s3)
```

[8]: float

[10]:
```
# list
l=[1,2,3]

# tuple
t=(1,2,3)

# set
s=set((1,2,3))

# dictionary
d={1:"one",2:"two","3":"three"}
```

[11]:
```
d
```

[11]: {1: 'one', 2: 'two', '3': 'three'}

[12]:
```
d[1]
```

[12]: 'one'

[13]:
```
l[2]
```

[13]: 3

[14]:
```
l[2]=10
```

[15]:
```
l
```

[15]: [1, 2, 10]

[16]:
```
# formatting a string
s = "And the winner is %s with %d points" % ("Jan",10)
```

[17]:
```
s
```

[17]: 'And the winner is Jan with 10 points'

[18]:
```
# if statement

a = 2;
if a>3:
    print("true")
else:
    print("false")
```

```
false
```

[20]:
```
# loop
for i in range(10):
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

[21]:
```
# function
def sumAB(a,b):
    return a+b
```

[22]:
```
sumAB(1,2)
```

[22]: 3

[25]:
```
def function(*argv, **kwarg):
    print("argv -- variable length arguments: %s" % str(argv))
    print("kwarg -- keyed arguments: %s" % str(kwarg))
```

[26]:
```
function(1,2,3,four=4,five=5)
```

```
argv -- variable length arguments: (1, 2, 3)
kwarg -- keyed arguments: {'four': 4, 'five': 5}
```

[27]:
```
function(6)
```

```
argv -- variable length arguments: (6,)
kwarg -- keyed arguments: {}
```

[28]:
```python
# function with more outputs
def function2(a):
    return 2*a,0.5*a
```

[29]:
```python
function2(2)
```

[29]: (4, 1.0)

[30]:
```python
output1, output2=function2(2)
output2
```

[30]: 1.0

[31]:
```python
# Class
class dummyClass(object):
    v = ""

    def __init__(self,inputString):
        self.v = inputString

    def append(self, append2v):
        return self.v+str(append2v)
```

[32]:
```python
c=dummyClass("My string")
```

[33]:
```python
c.v
```

[33]: 'My string'

[34]:
```python
c.append(" ... ")
```

[34]: 'My string … '

[30]:
```python
# lambda function -- example apply function on a list
list(map(lambda x: x+x,[1,2,3,4]))
```

[30]: [2, 4, 6, 8]

2 NumPy

Let us add few bits and pieces of the numPy library for the high-performance numerical computations.

- Fast operations with arrays
- C/C++/Fortran integration underneath
- Linear algebra, random numbers, Fourier transform…

numPy is used in scikit-learn and thus we familiarise with it as part of the Python tutorial.

[35]:
```
import numpy as np
```

[36]:
```
np.array([1,2,3])
```

[36]:
```
array([1, 2, 3])
```

[37]:
```
np.array([1,2,3],dtype=complex)
```

[37]:
```
array([1.+0.j, 2.+0.j, 3.+0.j])
```

2.1 Basic operations

[38]:
```
npArray=np.array([[1,2],[3,4],[5,6]])
```

[39]:
```
npArray.ndim
```

[39]:
```
2
```

[40]:
```
npArray.shape
```

[40]:
```
(3, 2)
```

[41]:
```
npArray.size
```

[41]:
```
6
```

[42]:
```
npArray.dtype
```

[42]:
```
dtype('int64')
```

[43]:
```
npArray.itemsize # of each element in bytes
```

[43]:
```
8
```

[44]:
```
np.zeros([3,3])
```

[44]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

[45]:
```
np.ones(3)
```

[45]:
```
array([1., 1., 1.])
```

[46]:
```
np.empty([3,3])
```

[46]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

[47]:
```python
np.random.random((1))
```

[47]:
```
array([0.57137326])
```

[49]:
```python
np.random.random((1,5))
```

[49]:
```
array([[0.42891732, 0.63999045, 0.47397617, 0.75932328, 0.45034846]])
```

[50]:
```python
np.random.randint(3, size=(3,3))
```

[50]:
```
array([[0, 2, 0],
       [0, 1, 1],
       [1, 1, 2]])
```

[51]:
```python
np.eye(3)
```

[51]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

[52]:
```python
np.linspace(0,3,num=5)
```

[52]:
```
array([0.  , 0.75, 1.5 , 2.25, 3.  ])
```

[53]:
```python
np.arange(0,3,step=0.25)
```

[53]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  , 2.25, 2.5 ,
       2.75])
```

[54]:
```python
np.arange(12).reshape(2,6)
```

[54]:
```
array([[ 0,  1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10, 11]])
```

[55]:
```python
a1=np.random.randint(3, size=(3,3));
a2=np.arange(9).reshape(3,3);
```

[56]:
```python
a1
```

[56]:
```
array([[1, 0, 0],
       [1, 2, 0],
       [2, 1, 1]])
```

[57]:
```
a2
```

[57]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

[58]:
```
a1,a2
```

[58]:
```
(array([[1, 0, 0],
        [1, 2, 0],
        [2, 1, 1]]), array([[0, 1, 2],
        [3, 4, 5],
        [6, 7, 8]]))
```

[59]:
```
a1+a2
```

[59]:
```
array([[1, 1, 2],
       [4, 6, 5],
       [8, 8, 9]])
```

[60]:
```
a1-a2
```

[60]:
```
array([[ 1, -1, -2],
       [-2, -2, -5],
       [-4, -6, -7]])
```

[61]:
```
a1*2
```

[61]:
```
array([[2, 0, 0],
       [2, 4, 0],
       [4, 2, 2]])
```

[62]:
```
a1**2
```

[62]:
```
array([[1, 0, 0],
       [1, 4, 0],
       [4, 1, 1]])
```

[63]:
```
a1>3
```

[63]:
```
array([[False, False, False],
       [False, False, False],
       [False, False, False]])
```

[64]:
```
a1+=2
```

[65]:
```
a1
```

[65]:
```
array([[3, 2, 2],
       [3, 4, 2],
       [4, 3, 3]])
```

[66]:
```
# matrix product
np.dot(a1,a2)
```

[66]:
```
array([[18, 25, 32],
       [24, 33, 42],
       [27, 37, 47]])
```

[67]:
```
# elementw-wise
a1*a2
```

[67]:
```
array([[ 0,  2,  4],
       [ 9, 16, 10],
       [24, 21, 24]])
```

[ ]:
```

```

3 Qiskit

Our objective is to use qiskit actively. Easy way to install the Qiskit is to use Anaconda (install the most recent version, all libraries). Otherwise follow https://qiskit.org for details.

Qiskit is a Sofgtware Development Toolkit for working with quantum computers on the low-level (pulses, circuits...). It is open-source and we can execute code on IBM Quantum Computers.

[3]:
```
# Import numpy
import numpy as np

# Import basic object from the Qiskit
from qiskit import QuantumCircuit
```

[4]:
```
# Check the version of the qiskit you use
import qiskit.tools.jupyter
%qiskit_version_table
%qiskit_copyright
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

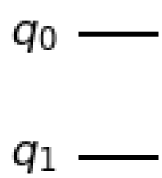
The Qiskit library allows us to build the quantum circuits, where we can perform operations on quantum bits and thus implement the quantum algorithms.

[5]:
```
# Create quantum circuit
circuit = QuantumCircuit(2)
```

[6]:
```
# Draw it
%matplotlib inline

circuit.draw('mpl')
```

[6]:

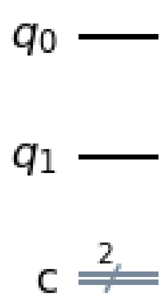


We have created simple object, which is composed of two qubits. We have not done anything with the object yet -- we have set the workbench and we can start adding elements to it.

The circuit with some gates…

[9]:
```
# Create quantum circuit -- 2 quibits and 2 classical bits
circuit = QuantumCircuit(2, 2)
circuit.draw('mpl')
```

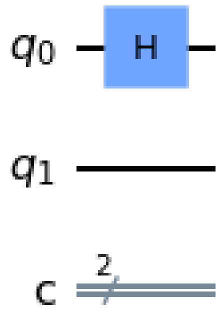
[9]:



[10]:
```
# Add an H gate to qubit 0
circuit.h(0)
```

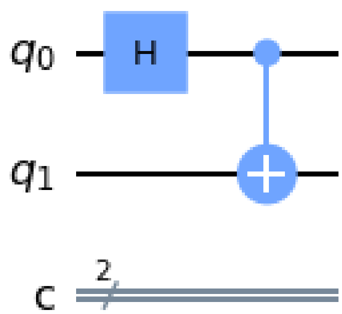
```
circuit.draw('mpl')
```

[10]:



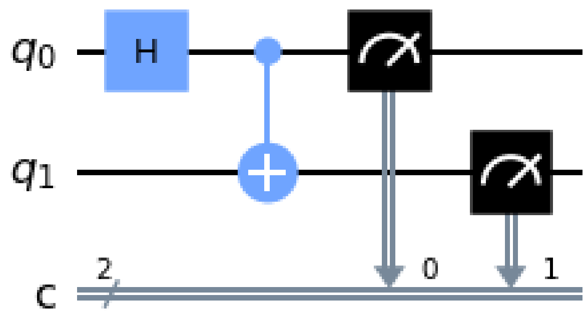
[11]:
```
# Add an CNOT gate, control qubit 0 and target qubit 1
circuit.cx(0, 1)
circuit.draw('mpl')
```

[11]:



[12]:
```
# Quantum measurement -- linking the quantum qubits to classical bits
circuit.measure([0,1], [0,1])
circuit.draw('mpl')
```

[12]:



Once we create the circuit, we can start using it. There are two options:

- Simulator
- Real Quantum Computer

3.1 Simulator

[13]:
```
from qiskit import Aer
from qiskit import execute

simulator = Aer.get_backend('qasm_simulator')

# Execute the circuit in the simulator
job = execute(circuit, simulator, shots=1024)
```

[14]:
```
# Bit messy outcome...
job.result()
```

[14]:
```
Result(backend_name='qasm_simulator', backend_version='0.8.1',
qobj_id='41fc1a03-dafd-4c9a-8e8b-4cfcec5a882f',
job_id='fc6cc27c-b1d9-4477-ab94-5cae2cc07fee', success=True,
results=[ExperimentResult(shots=1024, success=True,
meas_level=MeasLevel.CLASSIFIED, data=ExperimentResultData(counts={'0x3': 522,
'0x0': 502}), header=QobjExperimentHeader(clbit_labels=[['c', 0], ['c', 1]],
creg_sizes=[['c', 2]], global_phase=0.0, memory_slots=2, metadata=None,
n_qubits=2, name='circuit-11', qreg_sizes=[['q', 2]], qubit_labels=[['q', 0],
['q', 1]]), status=DONE, seed_simulator=2055376700,
metadata={'parallel_state_update': 8, 'parallel_shots': 1, 'measure_sampling':
True, 'method': 'stabilizer', 'fusion': {'enabled': False}},
time_taken=0.002284898)], date=2022-02-19T23:49:41.644471, status=COMPLETED,
status=QobjHeader(backend_name='qasm_simulator', backend_version='0.8.1'),
metadata={'mpi_rank': 0, 'time_taken': 0.0030557320000000002,
```

'max_gpu_memory_mb': 0, 'max_memory_mb': 8192, 'parallel_experiments': 1, 'num_mpi_processes': 1, 'omp_enabled': True}, time_taken=0.003438711166381836)

[15]:
```
# Take result
result = job.result()
```

[16]:
```
# Counts
counts = result.get_counts(circuit)
print("\nTotal count for 00 and 11 are:",counts)
```

```
Total count for 00 and 11 are: {'11': 522, '00': 502}
```

3.2 Quantum Computer

We need to have an account with IBM to access the quantum computer. In order to get the account, create an account at:

https://quantum-computing.ibm.com/

Once you have your account, generate API token. We use it to link it to our account.

from qiskit import IBMQ

IBMQ.save_account(TOKEN)

Remark: This is not executable cell as you need to run it only once. Keep the TOKEN private and do not share it.

[17]:
```
# Load account from the disk
IBMQ.load_account()
```

```
ibmqfactory.load_account:WARNING:2022-02-20 00:03:55,287: Credentials are
already in use. The existing account in the session will be replaced.
```

[17]:
```
[<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>,
 <AccountProvider for IBMQ(hub='ibm-q-startup', group='spinup-ai',
project='reservations')>]
```

[18]:
```
# The object IBMQ contains information about connections etc
IBMQ.providers()
```

[18]:
```
[<AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>,
 <AccountProvider for IBMQ(hub='ibm-q-startup', group='spinup-ai',
project='reservations')>]
```

[20]:
```
# Get a provider from the IBMQ object
provider = IBMQ.get_provider(hub = 'ibm-q')
```

[20]: <AccountProvider for IBMQ(hub='ibm-q', group='open', project='main')>

[22]:
```
# And all available backends
ibmq_backends = provider.backends()
ibmq_backends
```

[22]:
```
[<IBMQSimulator('ibmq_qasm_simulator') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_armonk') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_santiago') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_bogota') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_lima') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_quito') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_statevector') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_mps') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQSimulator('simulator_extended_stabilizer') from IBMQ(hub='ibm-q',
group='open', project='main')>,
 <IBMQSimulator('simulator_stabilizer') from IBMQ(hub='ibm-q', group='open',
project='main')>,
 <IBMQBackend('ibmq_manila') from IBMQ(hub='ibm-q', group='open',
project='main')>]
```

The physical machines can be busy and choosing the right one can be useful. We can use a simple function to choose the least busy one…

[23]:
```
from qiskit.providers.ibmq import least_busy
lb = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits␣
↪>= 2, simulator=False))
lb
```

```
VBox(children=(HTML(value="<h1 style='color:#ffffff;background-color:#000000;padding-top: 1%;pa
```

[23]: <IBMQBackend('ibmq_belem') from IBMQ(hub='ibm-q', group='open', project='main')>

[24]:
```
# Let us execute the job above on the machine
job_exp = execute(circuit, lb, shots=1024)
result_exp = job_exp.result()
```

[25]:
```python
# Counts
counts = result_exp.get_counts(circuit)
print("\nTotal count for 00 and 11 are:",counts)
```

```
Total count for 00 and 11 are: {'00': 485, '01': 34, '10': 26, '11': 479}
```

```
The real quantum computer is noisy and contains errors!
```

```
We have run the code on the real quantum computer!
```

[ ]:
```python

```

[ ]:
```python

```