Introduction to Python and Jupyter notebooks

We will learn quantum computation using the QISKIT-Toolbox, which is a Phython Interface to both Quanutm Simulators but also Quantum Hardware, provided by IBM. In the very first subchapter we shall have a quick look at the way we can use python to run simple calculations.

One way to code in Python is to use a Jupyter notebook. This is probably the best way to combine programming, text and images. In a notebook, everything is laid out in cells. Text cells and code cells are the most common. If you are viewing this section as a Jupyter notebook, the text you are now reading is in a text cell. A code cell can be found just below.

To run the contents of a code cell, you can click on it and press Shift + Enter. Or if there is a little arrow thing on the left, you can click on that.

Simple Arthmetics

In [146]:

```
1 + 1
```

Out[146]:

```
2
```

There is of course much more you can do, than just adding 1+1. The notebook gives you full accesibility to the entirety of Python's galore. Here are just a few examples.

Creating and manipulating variables

In [147]:

```
an_integer = 42 # Just an integer
an_integer #intermediate results are not displayed implicitly
print(an_integer) #use print to display instead


a_float = 0.1 # A non-integer number, up to a fixed precision
a_float = a_float * 2.1 #Multiply by 2.1
print(a_float)

a_boolean = True # A value that can be True or False
a_string = '''just enclose text between two 's, or two "s, or do what we did for this s
tring''' # Text
print(a_string)

none_of_the_above = None # The absence of any actual value or variable type
print(none_of_the_above)

a_float #last result is displayed implicity
```

```
42
0.21000000000000002
just enclose text between two 's, or two "s, or do what we did for this st
ring
None
```

Out[147]:

```
0.21000000000000002
```

Creating and manipulating lists and dictionaries

In [148]:

```
a_list = [0,1,2,3] #A very simple list of identical items
a_list = [ 42, 0.5, True, [0,1], None, 'Banana' ] #Lists can be of different types, too
print(a_list) #show list

print(a_list[0]) #access an individual list item

a_list[5] = 'apple' #change list item
print(a_list)

a_list.append( 3.14 ) #append items to list
print(a_list)
```

```
[42, 0.5, True, [0, 1], None, 'Banana']
42
[42, 0.5, True, [0, 1], None, 'apple']
[42, 0.5, True, [0, 1], None, 'apple', 3.14]
```

Dictioraries are similar to lists, the difference being that were list have 0 to n-1 indicicies, dictionaries have well-defined key:value pairs. Key must be unique, obvisouly.

In [6]:

```
a_dict = { 1:'This is the value, for the key 1', 'This is the key for a value 1':1, Fal
se:':)', (0,1):256 }
print(a_dict)
print(a_dict['This is the key for a value 1']) #you can acces any list items by it's un
ique key, whateever that may be
a_dict['new key'] = 'new_value' #add a new key:value pair
a_dict[1] = 'New Key' #Change the value for an existing key
print(a_dict)
```

```
{1: 'This is the value, for the key 1', 'This is the key for a value 1':
1, False: ':)', (0, 1): 256}
1
{1: 'New Key', 'This is the key for a value 1': 1, False: ':)', (0, 1): 25
6, 'new key': 'new_value'}
```

Looping lists and dictionaries

We start by simply looping over a range of numbers using the range(n) command to create a list running from 0 to n

In [7]:

```
for j in range(5):
    print(j)
```

```
0
1
2
3
4
```

You can also loop over any 'iterable' object, such as lists

In [8]:

```
for j in a_list:
    print(j)
```

```
42
0.5
True
[0, 1]
None
apple
3.14
```

or dictionaries, where you automatically get access to key/value pairs

In [9]:

```
for key in a_dict:
    value = a_dict[key]
    print('key =',key)
    print('value =',value)
    print()
```

```
key = 1
value = New Key

key = This is the key for a value 1
value = 1

key = False
value = :)

key = (0, 1)
value = 256

key = new key
value = new_value
```

You can also use a slightly different notation

In [10]:

```
for key, value in a_dict.items():
    print('key =',key)
    print('value =',value)
    print()
```

```
key = 1
value = New Key

key = This is the key for a value 1
value = 1

key = False
value = :)

key = (0, 1)
value = 256

key = new key
value = new_value
```

If-Then Statements

In [11]:

```
if 'strawberry' in a_list:
    print('We have a strawberry!')
elif a_list[5]=='apple':
    print('We have an apple!')
else:
    print('Not much fruit here!')
```

```
We have an apple!
```

Import Libraries, Math and Plotting

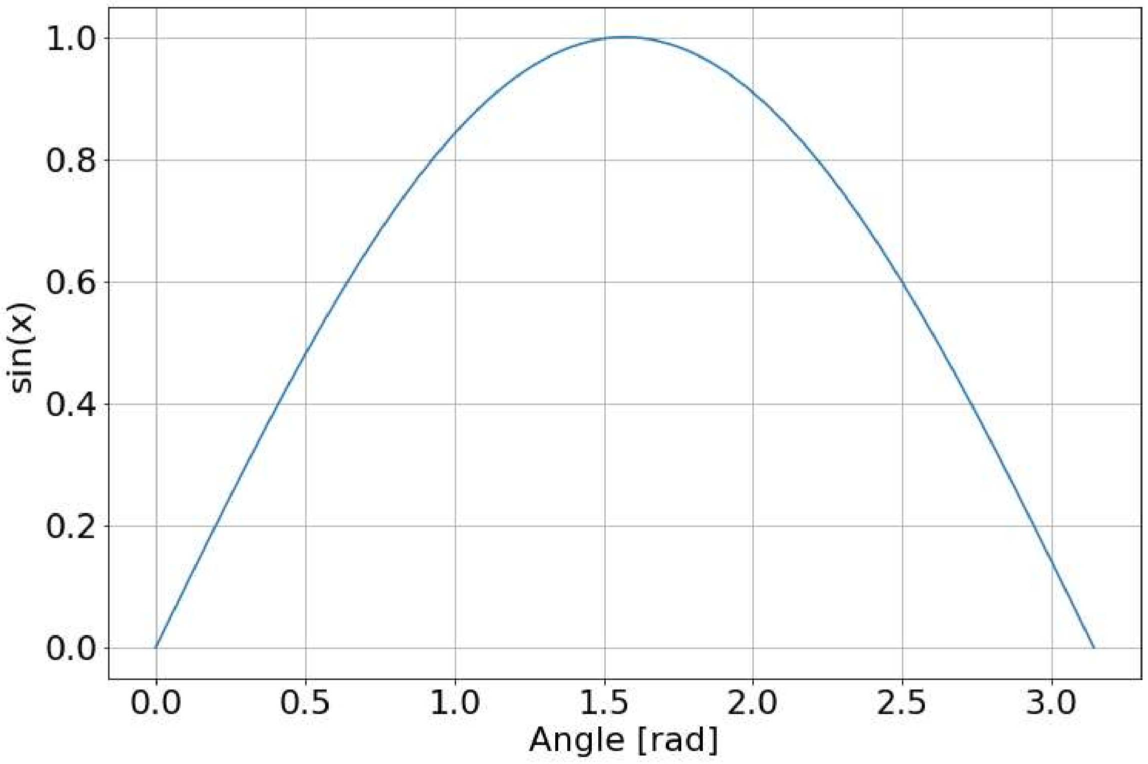
In [13]:

```
import numpy as np
print(np.sin( np.pi/2 ))
x = np.linspace(0,np.pi,100)
print(np.sin(x))
```

```
1.0
[0.00000000e+00 3.17279335e-02 6.34239197e-02 9.50560433e-02
 1.26592454e-01 1.58001396e-01 1.89251244e-01 2.20310533e-01
 2.51147987e-01 2.81732557e-01 3.12033446e-01 3.42020143e-01
 3.71662456e-01 4.00930535e-01 4.29794912e-01 4.58226522e-01
 4.86196736e-01 5.13677392e-01 5.40640817e-01 5.67059864e-01
 5.92907929e-01 6.18158986e-01 6.42787610e-01 6.66769001e-01
 6.90079011e-01 7.12694171e-01 7.34591709e-01 7.55749574e-01
 7.76146464e-01 7.95761841e-01 8.14575952e-01 8.32569855e-01
 8.49725430e-01 8.66025404e-01 8.81453363e-01 8.95993774e-01
 9.09631995e-01 9.22354294e-01 9.34147860e-01 9.45000819e-01
 9.54902241e-01 9.63842159e-01 9.71811568e-01 9.78802446e-01
 9.84807753e-01 9.89821442e-01 9.93838464e-01 9.96854776e-01
 9.98867339e-01 9.99874128e-01 9.99874128e-01 9.98867339e-01
 9.96854776e-01 9.93838464e-01 9.89821442e-01 9.84807753e-01
 9.78802446e-01 9.71811568e-01 9.63842159e-01 9.54902241e-01
 9.45000819e-01 9.34147860e-01 9.22354294e-01 9.09631995e-01
 8.95993774e-01 8.81453363e-01 8.66025404e-01 8.49725430e-01
 8.32569855e-01 8.14575952e-01 7.95761841e-01 7.76146464e-01
 7.55749574e-01 7.34591709e-01 7.12694171e-01 6.90079011e-01
 6.66769001e-01 6.42787610e-01 6.18158986e-01 5.92907929e-01
 5.67059864e-01 5.40640817e-01 5.13677392e-01 4.86196736e-01
 4.58226522e-01 4.29794912e-01 4.00930535e-01 3.71662456e-01
 3.42020143e-01 3.12033446e-01 2.81732557e-01 2.51147987e-01
 2.20310533e-01 1.89251244e-01 1.58001396e-01 1.26592454e-01
 9.50560433e-02 6.34239197e-02 3.17279335e-02 1.22464680e-16]
```

In [19]:

```
import matplotlib.pylab as plt
plt.figure(figsize=(12,8), facecolor='white')
plt.rcParams.update({'font.size': 22})
plt.plot(x, np.sin(x))
plt.xlabel('Angle [rad]')
plt.ylabel('sin(x)')
plt.grid(which='major',axis='both')
```



Defining Functions

In [21]:

```
def add_sausages ( input_list ):
    if 'sausages' not in input_list:
        input_list.append('sausages')
```

In [22]:

```
print('List before the function')
print(a_list)

add_sausages(a_list) # function called without an output

print('\nList after the function')
print(a_list)
```

```
List before the function
[42, 0.5, True, [0, 1], None, 'apple', 3.14]

List after the function
[42, 0.5, True, [0, 1], None, 'apple', 3.14, 'sausages']
```

Your first Quantum Circuits

Libraries

So far we have only looked into Pyhton/Juypiter in general. Next we start with the proper quantum. Do to so, we must import a few important libraries/classes/functions.

In [5]:

```
from qiskit import QuantumCircuit, assemble, Aer
from qiskit.visualization import plot_histogram
```

In a circuit, we typically need to do three jobs: First, encode the input, then do some actual computation, and finally extract an output. For your first quantum circuit, we'll focus on the last of these jobs. We start by creating a circuit with eight qubits and eight outputs.

In [23]:

```
n=4
qc_output = QuantumCircuit(n)
```

This circuit, which we have called `qc_output`, is created by Qiskit using `QuantumCircuit`. The `QuantumCircuit` takes the number of qubits in the quantum circuit as an argument.

The extraction of outputs in a quantum circuit is done using an operation called `measure_all()`. Each measurement tells a specific qubit to give an output to a specific output bit. The command `qc_output.measure_all()` adds a measurement to each qubit in the circuit `qc_output`, and also adds some classical bits to write the output to.

In [24]:

```
qc_output.measure_all()
```

Qubits are always initialized to give the output `0`. Since we don't do anything to our qubits in the circuit above, this is exactly the result we'll get when we measure them. We can see this by running the circuit many times and plotting the results in a histogram. We will find that the result is always `00000000`: a `0` from each qubit.

sim = Aer.get_backend('aer_simulator') result = sim.run(qc_output).result() counts = result.get_counts() plot_histogram(counts)

Initialization of QuBits

We first import a few helpful classes into visualize the state of QuBits

In [ ]:

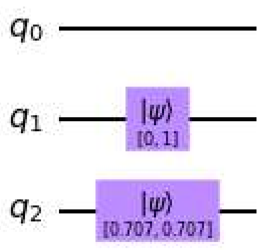
```
from qiskit.visualization import plot_histogram, plot_bloch_vector
from math import sqrt, pi
```

We then create a single quibt circuit

In [6]:

```
qc = QuantumCircuit(3) # Create a quantum circuit with two qubits
initial_stateV = [0,1]   # Definean initial_state as |1>
initial_stateU = [1/sqrt(2),1/sqrt(2)]   # Definean initial_state as |1>
qc.initialize(initial_stateV, 1) # Apply initialisation operation to the 1sr qubit
qc.initialize(initial_stateU, 2) # Apply initialisation operation to the 1sr qubit
qc.draw()  # Let's view our circuit
```

Out[6]:



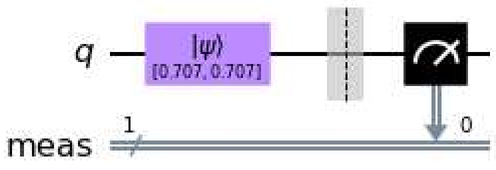
Task1: Create a balanced single-bit quantum random number generator, draw it, run it, and display the results

We first create the Quantum Circuit by inititalizing the QuBit into a 45° linear polarization state

In [7]:

```
#Deleted code. Students need to complete this.
```

Out[7]:



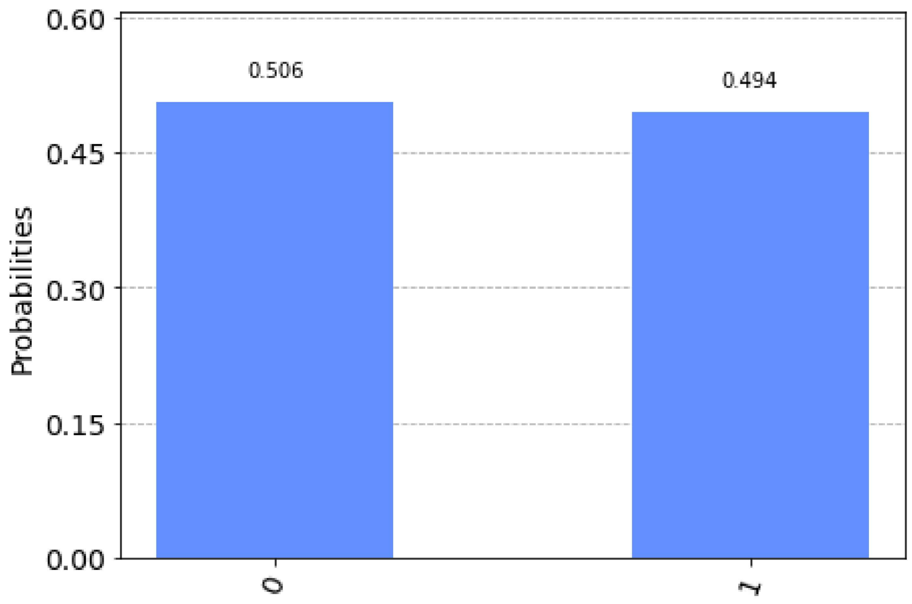
Then we run the circuit on a simulator

In [8]:

```
#Deleted code. Students need to complete this.
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning: scipy._lib.messagestrea
m.MessageStream size changed, may indicate binary incompatibility. Expecte
d 56 from C header, got 64 from PyObject
```

Out[8]:



This is close to the expected 50/50-result. However, this never exactly hits 50/50 because we are merely doing a sample over N=1000 runs, which means we get a probability distribution for the average of 0.5(1 +/- 1/sqrt(N)) = 0.5*(1 +/- 0.03). The simplified version of think about this is: we estmiate the probabilities.

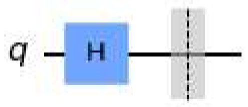
Single Qubit Gates

This is however not the preferred way of creating quanutm circuits, instead we typicall start with a |0> state and use a Hadamard-Gate (a 22.5° HWP) to create the superposition state. This time we would also like to visualize the state vector.

In [71]:

```
qc = QuantumCircuit(1)
qc.h(0) #Add a Hadamard Gate to the 0th Qubit
qc.save_statevector()
qc.draw()  # Let's view our circuit
```

Out[71]:



In [73]:

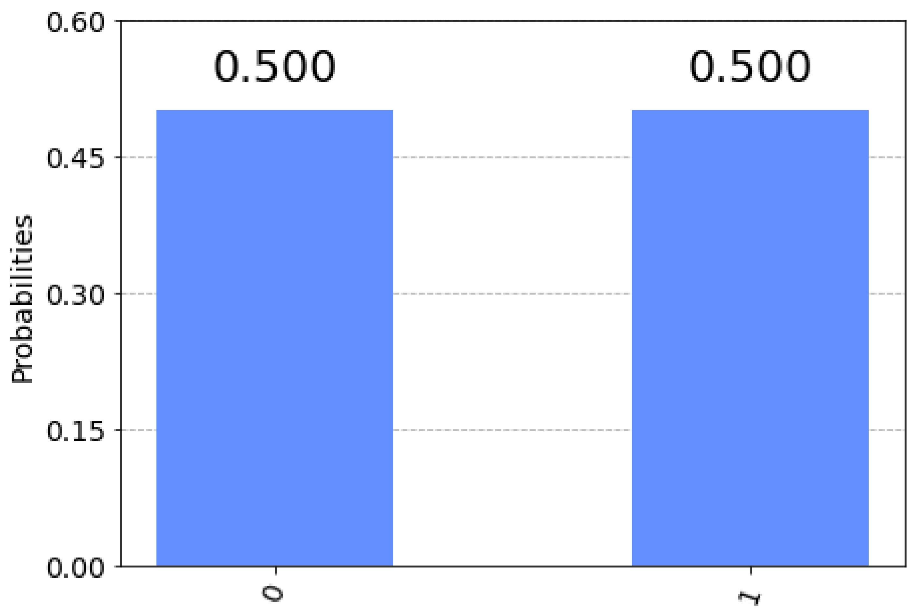
```
sim = Aer.get_backend('aer_simulator')


result = sim.run(qc).result();
state = result.get_statevector()
print("State of Measured Qubit = " + str(state))

counts = result.get_counts()
plot_histogram(counts)
```

```
State of Measured Qubit = Statevector([0.70710678+0.j, 0.70710678+0.j],
            dims=(2,))
```

Out[73]:

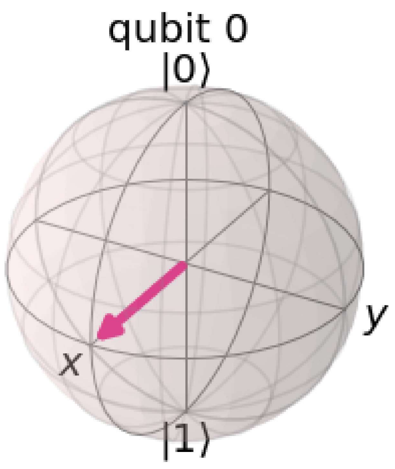


You see two things. First you see that the state-vector is kind of hard to visulaize and second you see that the omission of a specific measurement leads to the replacement of samples with properties. We can easily visulize teh state-vectors unsing built-in fucntions.

In [75]:

```
from qiskit.visualization import plot_bloch_multivector
plot_bloch_multivector(state)
```

Out[75]:



In [ ]:

```
We can nicely us this function to visualize the action of different gates
```

In [ ]:

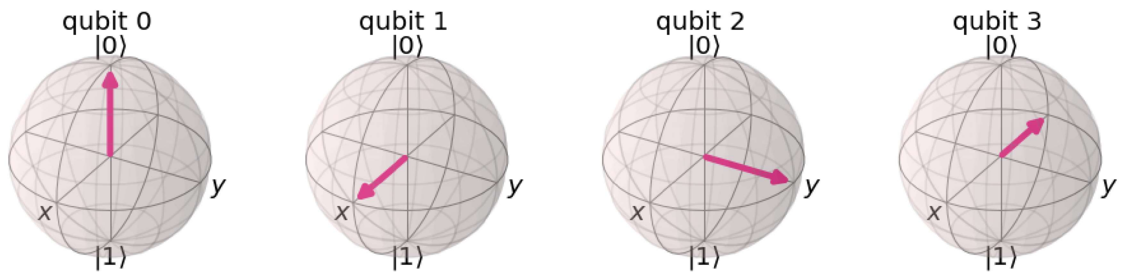
```
qc = QuantumCircuit(4)
qc.h(1) #Add a Hadamard Gate to the 1st Qubit --> diagonal polarization
qc.h(2) #Add a Hadamard Gate to the 2nd  Qubit --> diagonal polarization
qc.p(np.pi/2,2) #Add a 90 Deg Phase to the |1> part of the 2nd Quabit --> Circular Pol
qc.h(3) #Add a Hadamard Gate to the 3rd Qubit --> diagonal polarization
qc.z(3) #Add a Z- Gate to the 3rd Qubit --> inverse diagonal pol
qc.save_statevector()
qc.draw()  # Let's view our circuit
```

In [87]:

```
sim = Aer.get_backend('aer_simulator')


result = sim.run(qc).result();
state = result.get_statevector()
plot_bloch_multivector(state)
```

Out[87]:



A Real Quantum Computer Experimnent

So far we have only looked at very simple simulated exampled. Now we want to execute the quanum random number generator example for real. We first must load into the networks of QCs properly.

In [90]:

```
from qiskit import IBMQ, Aer, assemble, transpile
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit.providers.ibmq import least_busy

IBMQ.load_account()
provider = IBMQ.load_account()
provider = IBMQ.get_provider("ibm-q")
device = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits >= 3
and
                                   not x.configuration().simulator and x.status().opera
tional==True))
print("Running on current least busy device: ", device)
```

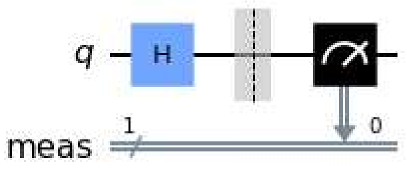
```
ibmqfactory.load_account:WARNING:2022-04-19 11:06:15,106: Credentials are
already in use. The existing account in the session will be replaced.

Running on current least busy device:  ibmq_belem
```

In [88]:

```
qc = QuantumCircuit(1)
qc.h(0) #Add a Hadamard Gate to the 0th Qubit
qc.measure_all()
qc.draw()  # Let's view our circuit
```

Out[88]:



In [91]:

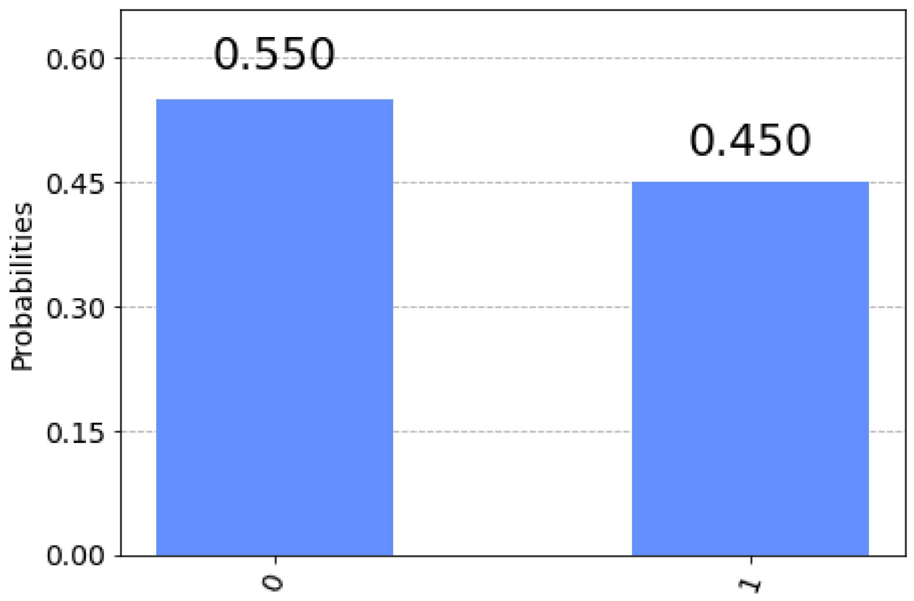
```
from qiskit.tools.monitor import job_monitor
qc_transpiled = transpile(qc, device, optimization_level=3)
job = device.run(qc_transpiled)
job_monitor(job, interval=2)
```

```
Job Status: job has successfully run
```

In [95]:

```
results = job.result()
answer = results.get_counts()
plot_histogram(answer)
```

Out[95]:



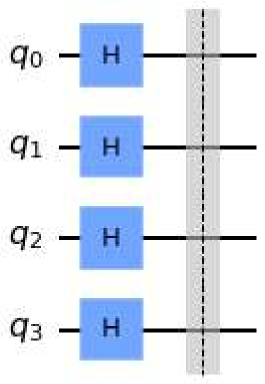
Multi-Qubit-Circuits and Entanglement

So far we have only looked into single-qubit states and hence have totally ignored the key-resource of quantum computation, that is Entanglement. We'll start out with an extremely simple example to visualize the computational power sclaing of a quanutm computer. A 4 q-bit Random number generator.

In [134]:

```
qc = QuantumCircuit(4)
qc.h(0) #Add a Hadamard Gate to the 0th Qubit
qc.h(1) #Add a Hadamard Gate to the 0th Qubit
qc.h(2) #Add a Hadamard Gate to the 0th Qubit
qc.h(3) #Add a Hadamard Gate to the 0th Qubit
qc.save_statevector()
qc.draw()  # Let's view our circuit
```

Out[134]:



Let's first display the state vector, which is 2^4=16 Dimensional:

In [135]:

```
sim = Aer.get_backend('aer_simulator')


result = sim.run(qc).result();
state = result.get_statevector()
state
```

```
Statevector([0.25+0.j, 0.25+0.j, 0.25+0.j, 0.25+0.j, 0.25+0.j, 0.25+0.j,
             0.25+0.j, 0.25+0.j, 0.25+0.j, 0.25+0.j, 0.25+0.j, 0.25+0.j,
             0.25+0.j, 0.25+0.j, 0.25+0.j, 0.25+0.j],
            dims=(2, 2, 2, 2))
```

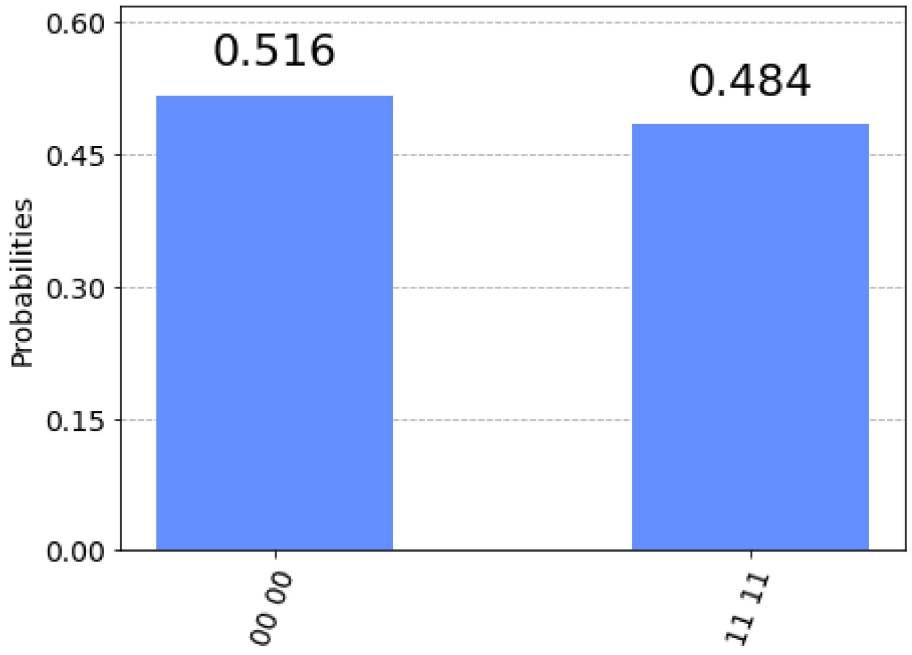
This 16-Dimensional State-Vector then collapses into a 4-bit number (ranging from 0 to 15) with qual probability. So each run of the circuit generates exactly one 4-bit number; their probability is:

In [149]:

```
qc.measure_all()

sim = Aer.get_backend('aer_simulator')
result = sim.run(qc).result()
plot_histogram(result.get_counts())
```

Out[149]:

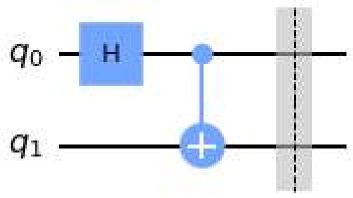


Let's now move to entanglement and true-multi-qubit gates. The most important one is the controlled-NOT gate. It is the quantum version of the q1=q1 XOR q0 gate and it can be used to create an entangled pair, in this case a ϕ state

In [141]:

```
qc = QuantumCircuit(2)
qc.h(0) #Add a Hadamard Gate to the 0th Qubit
qc.cx(0,1) #Add a Hadamard Gate to the 0th Qubit
qc.save_statevector()
qc.draw()  # Let's view our circuit
```

Out[141]:



In [142]:

```
sim = Aer.get_backend('aer_simulator')


result = sim.run(qc).result();
state = result.get_statevector()
state
```

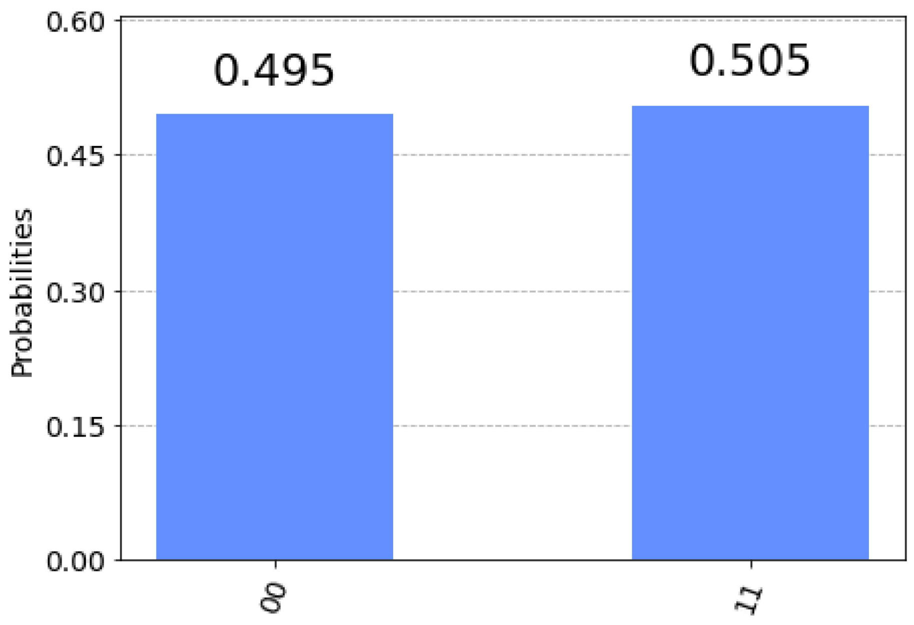
```
Statevector([0.70710678+0.j, 0.        +0.j, 0.        +0.j,
             0.70710678+0.j],
            dims=(2, 2))
```

In reality, we rather want to measure the probabilities of different results occuring:

In [143]:

```
qc.measure_all()
sim = Aer.get_backend('aer_simulator')
result = sim.run(qc).result()
plot_histogram(result.get_counts())
```

Out[143]:



As expecte we find that the results are prefectly correlated. Either both systems meansure 0 (00) or 1 (11) but never 01 (Qubit 0 in 0 and Qubit 1 in 1) or 10.

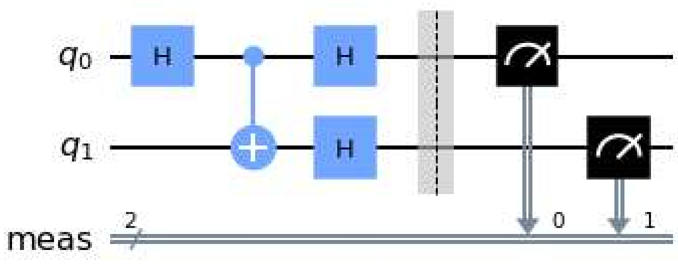
Task 2: Repeat the experiment from above but measure in the +/--Basis

Hint: Use the H-operator for 0/1 to +/- conversion

In [144]:

```
#Deleted code. Students need to complete this.
```

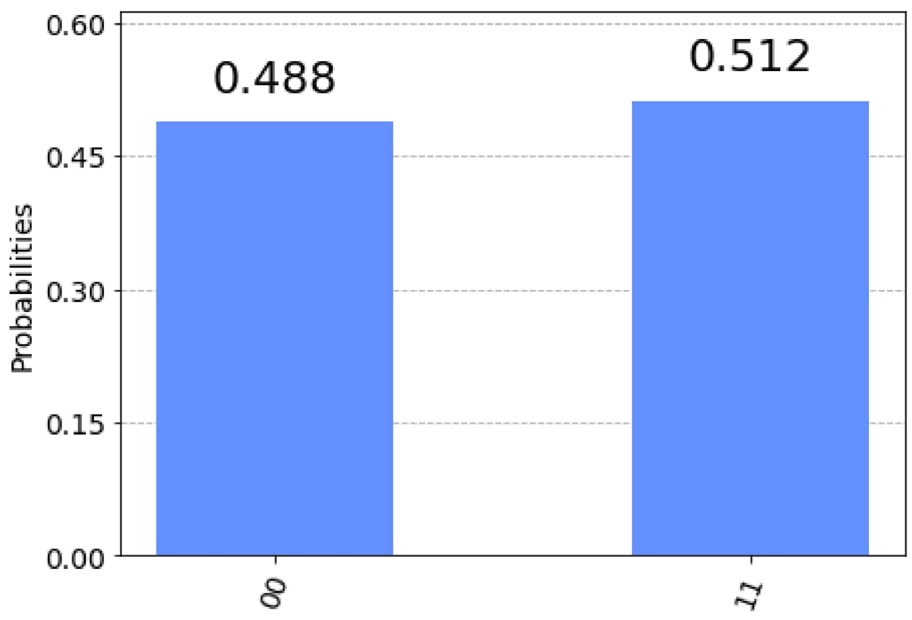
Out[144]:



In [111]:

```
#Deleted code. Students need to complete this.
```

Out[111]:



As expected, again we find that the results are indeed perfectly correlated.

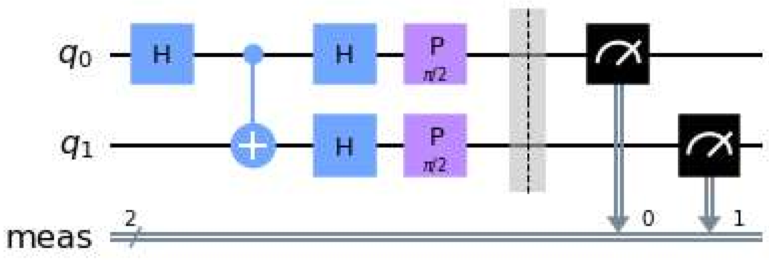
Task 3: Repeat the experiment from above but measure in the L/R-Basis

Hint: Use the H- and P-operator for 0/1 to L/R conversion

In [112]:

```
#Deleted code. Students need to complete this.
```

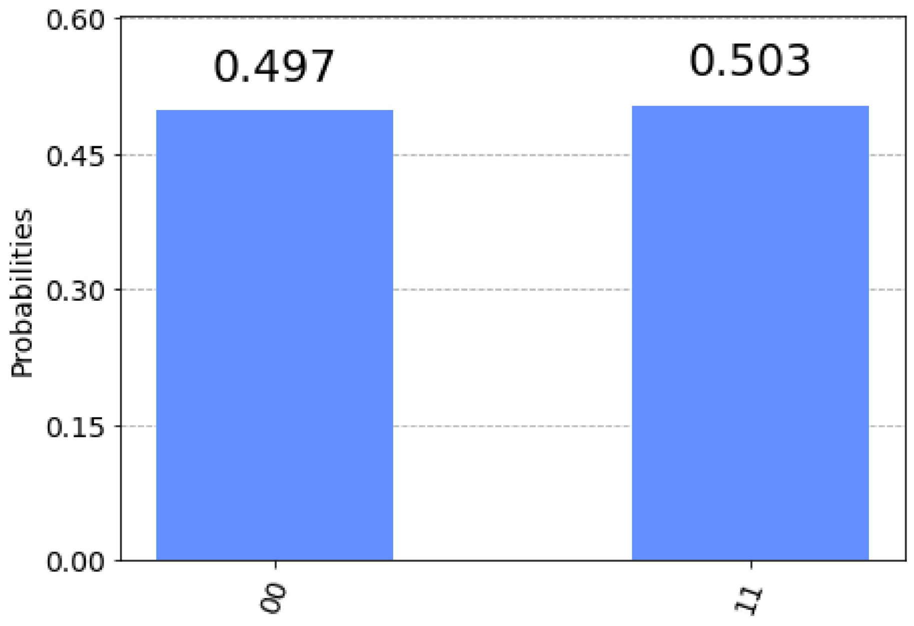
Out[112]:



In [113]:

```
#Deleted code. Students need to complete this.
```

Out[113]:


Probabilities