

Basic Python for Quantum



Muhammad Imansyah Basudewa

The University Center of Excellence for Advanced Intelligent Communications (AICOMS),
School of Electrical Engineering, Telkom University, Bandung, INDONESIA
E-mail: {Your E-mail Address}

Presented at
The 2021 AICOMS Workshop on Quantum Technology: Theory, Development, and Business
(AICOMS-Q)
Online, September 19, 2021

Outline

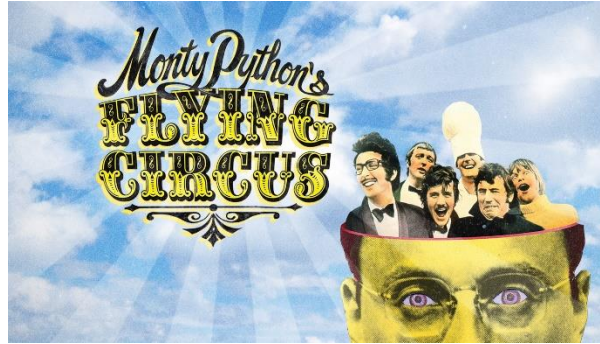
- 1 Introduction
- 2 The Basics
- 3 Operator
- 4 Loops and Functions
- 5 Library

Introduction [1/5]



- Python is a high level language, multipurpose, object oriented programming, and interactive environment.
- Python is a scripting language that can be used for website development, ML, AI, data science, data visualization, business applications and many more.
- We use Python 3 (3.6+) as our programming language and we will be working on the library Qiskit!

Introduction [2/5]



Python is an experiment in how much freedom programmers need. Too much freedom and nobody can read another's code; too little and expressiveness is endangered

- Guido Van Rossum

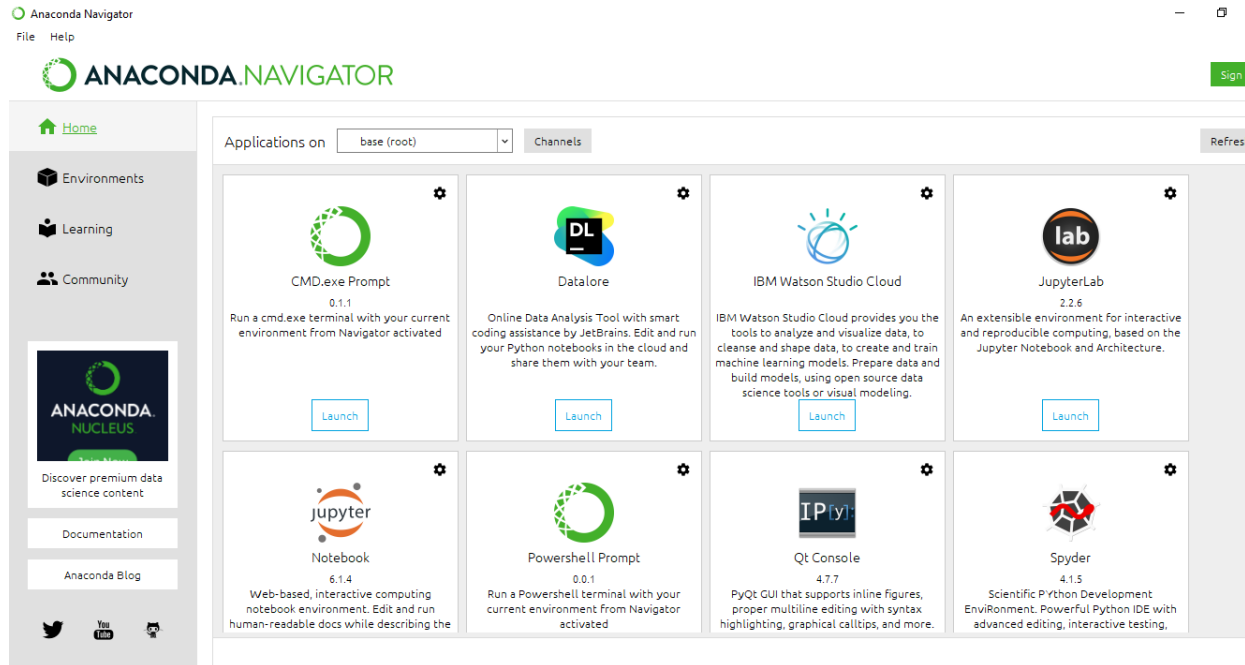
Introduction [3/5]



<https://docs.anaconda.com/anaconda/navigator/getting-started/>

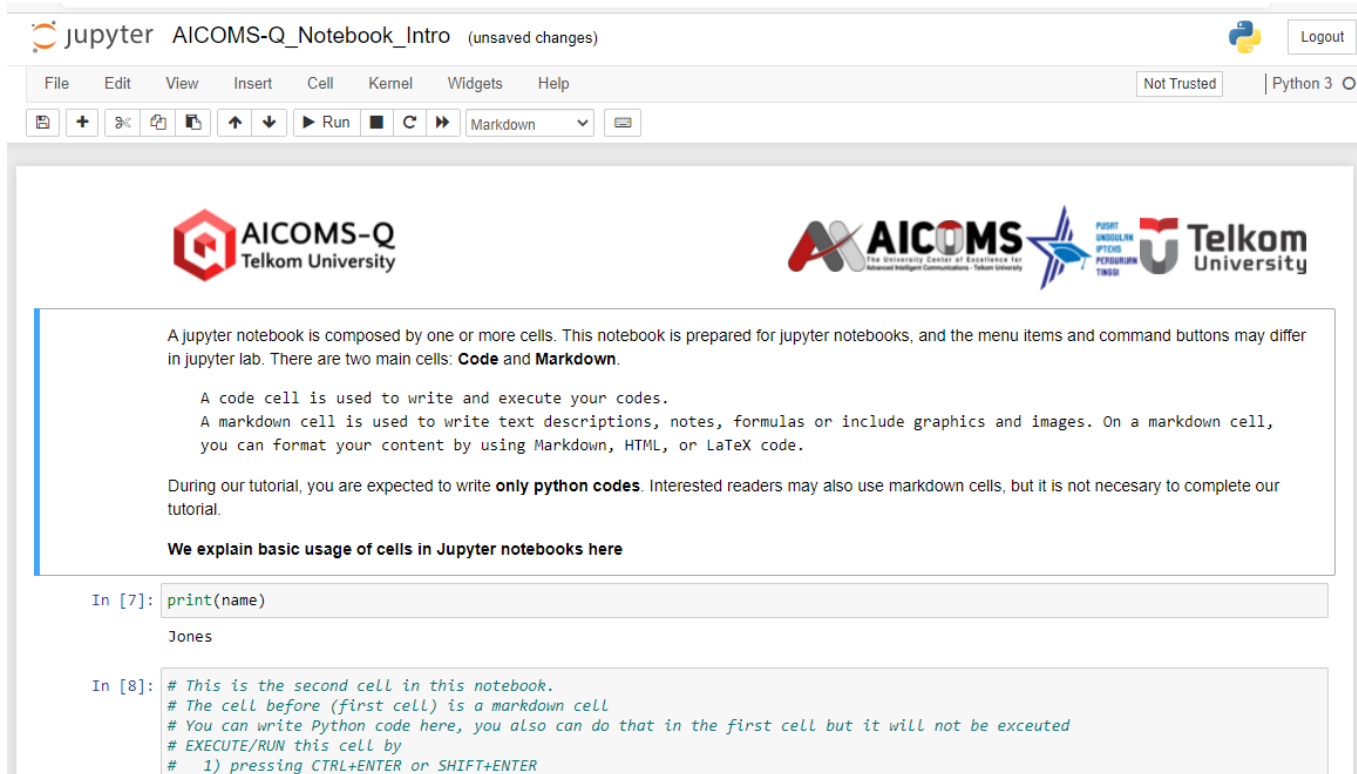
- Jupyter notebook is one of the most popular tools to create and share documents that contain interactive code, visualizations, text, etc.
- Anaconda is a free and open source distribution of the Python languages that aims to simplify package management and deployment.

Introduction [4/5]



- **Anaconda Navigator interface**
- Navigator is a desktop graphical user interface that allows you to launch applications and easily manage conda packages, environments, and channels without using command-line commands.
- The following applications are available by default in navigator
- JupyterLab, Jupyter Notebook, Spyder, PyCharm, VSCode, Glueviz, Orange 3 App, R Studio, Anaconda Prompt (Windows only), Anaconda PowerShell (Windows Only)

Introduction [5/5]



- The notebook user interface
- **Notebook kernel**
Computational engine that executes the code contained in a notebook.
- **Notebook cell**
Container for text to be displayed in a notebook or for code to be executed by the notebook's kernel.
- **Code** to be executed in the kernel. In front of the code cells are brackets that indicate the order in which the code was executed.
- In []: indicates that the code has not yet been executed.
- In [*]: indicates that the execution has not yet been completed

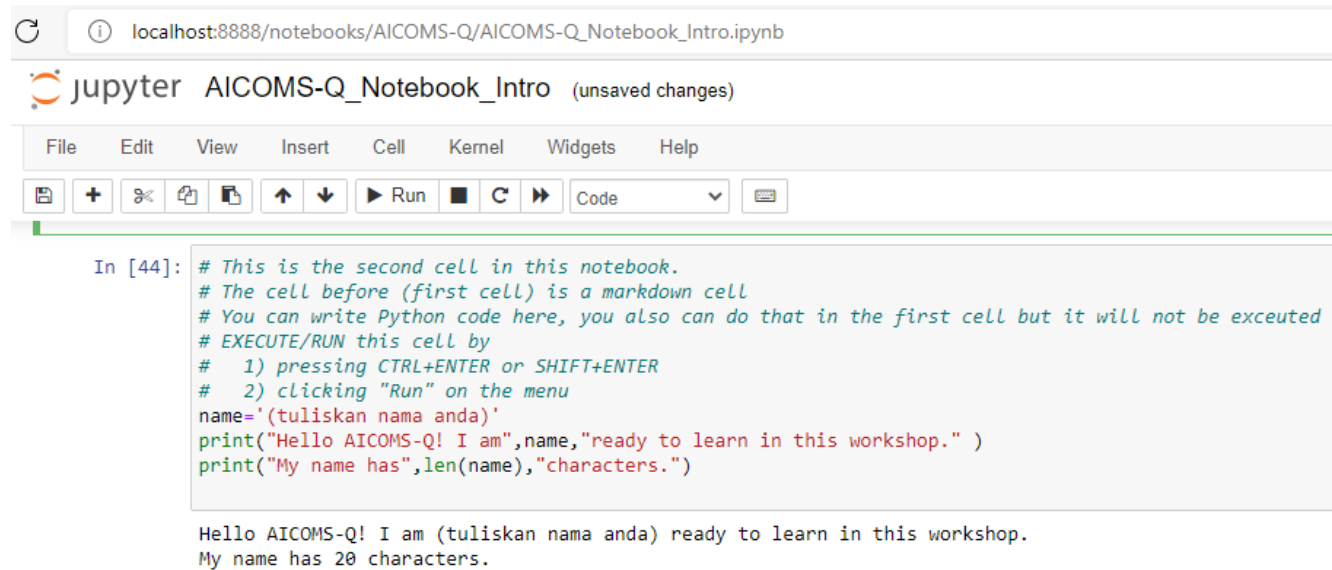
The Basics [1/7]

```
In [31]: x = 34 - 23      #A comment
         y = "Hello"
         z = 3.45
         if z == 3.45 or y == "Hello":
             x = x+1
             y = y + "World"    #String concat.
         print(x)
         print(y)

12
HelloWorld
```

- Indentation matters to code meaning
- First assignment to a variable creates it
- Variable types don't need to be declared
- Python figures out the variable types on its own
- Assignment is = and comparison is ==
- For number + - / * % are expected
- Logical operators are words (and, or, not) not symbols
- The basic printing command is "print"

The Basics [2/7]



The screenshot shows a Jupyter Notebook running in a web browser at localhost:8888. The notebook is titled "AICOMS-Q_Notebook_Intro" and has unsaved changes. The interface includes a menu bar (File, Edit, View, Insert, Cell, Kernel, Widgets, Help) and a toolbar with icons for saving, adding cells, undo, redo, and running code. The active cell is a code cell containing the following text:

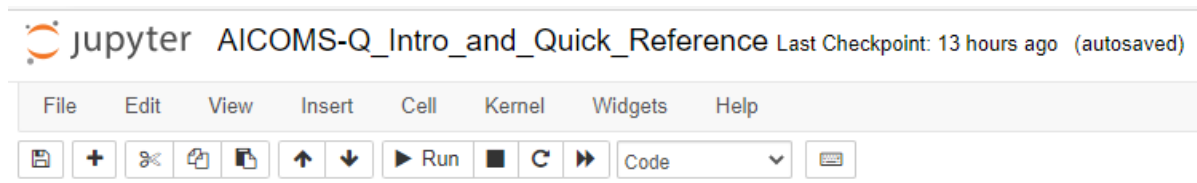
```
In [44]: # This is the second cell in this notebook.
# The cell before (first cell) is a markdown cell
# You can write Python code here, you also can do that in the first cell but it will not be executed
# EXECUTE/RUN this cell by
# 1) pressing CTRL+ENTER or SHIFT+ENTER
# 2) clicking "Run" on the menu
name='(tuliskan nama anda)'
print("Hello AICOMS-Q! I am",name,"ready to learn in this workshop.")
print("My name has",len(name),"characters.")
```

Below the code cell, the output of the code is displayed:

```
Hello AICOMS-Q! I am (tuliskan nama anda) ready to learn in this workshop.
My name has 20 characters.
```

- **Comments**
- Start comments with hash character (#) - the rest line is ignored
- Can include a “documentation string” as the first line of any new function or class that you define
- *#This is also a comment in python*
- *“”” This is an example of multiline comment that spans multilines ...”””*

The Basics [3/7]



Variables and Data Types

```
In [42]: #numbers:
number = 5 # integer
real = -3.4 # float

#complex numbers:
cpx=complex(3.0,12.3) # 3 + j 12.3
cpx_real=cpx.real
cpx_imaginary=cpx.imag

print ("The real part of complex number is : ",end="")
print (cpx.real)
print ("The imaginary part of complex number is : ",end="")
print (cpx.imag)

name = 'Imansyah' # string -> array of character, name=[R,i,c .....]
surname = "Basudewa" # also a string ('' and "" are the same in this scope)

complete_name=name+surname
print(complete_name)

boolean1 = True # 1
boolean2 = False # 0

The real part of complex number is : 3.0
The imaginary part of complex number is : 12.3
ImansyahBasudewa
```

Variable and Basic datatypes

- Integer
 $z = 5$
- Float
 $x = -3.4$
- String
- Can use “” or ‘’ to specify.
“abc” ‘abc’ (same thing).
- Unmatched can occur within the string.
“matt’s”
- Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them
“””a ‘b”c “””

The Basics [4/7]

Naming Rules

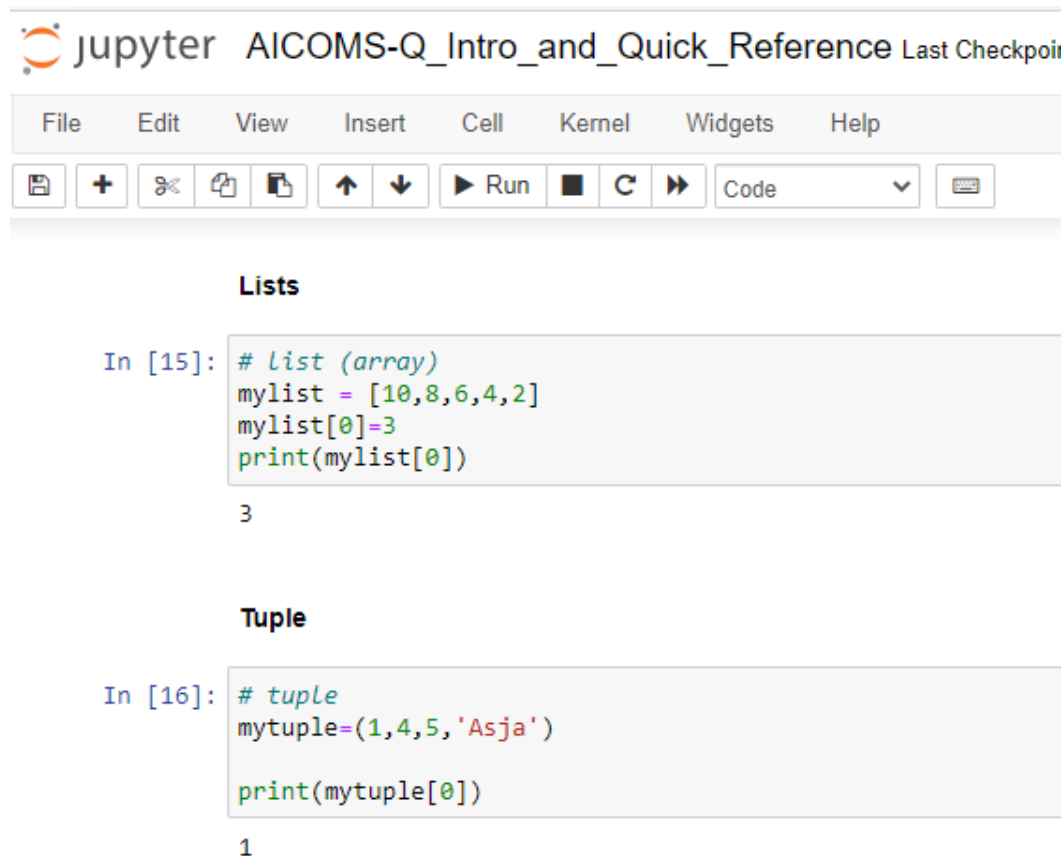
```
In [34]: data1 = 10    #example
         _data_1 = 10  #it can contain underscores
         data$ = 5
         break = 1

File "<ipython-input-34-670bb89f3a2b>", line 3
      data$ = 5
        ^
SyntaxError: invalid syntax
```

- Names are case sensitive and cannot start with a number.
- They can contain letters (a-z), numbers (0-9) and underscores (_)
name name1 _name name_1
- Symbols cannot be used
- There are some reserved words :

and assert break class continue
def del elif else except exec
finally for from global if import
in is lambda not or pass print
raise return try while

The Basics [5/7]



The image shows a Jupyter Notebook interface with the title 'AICOMS-Q_Intro_and_Quick_Reference'. The notebook contains two code cells. The first cell, titled 'Lists', shows the creation of a list 'mylist' with values [10, 8, 6, 4, 2], changing the first element to 3, and printing it. The second cell, titled 'Tuple', shows the creation of a tuple 'mytuple' with values (1, 4, 5, 'Asja') and printing the first element.

```
In [15]: # list (array)
mylist = [10,8,6,4,2]
mylist[0]=3
print(mylist[0])
3
```

Lists

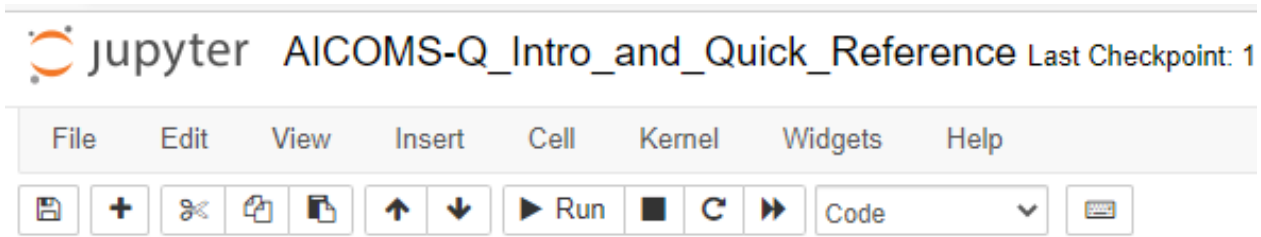
```
In [16]: # tuple
mytuple=(1,4,5,'Asja')
print(mytuple[0])
1
```

Tuple

- **Positive and negative indices**
- Positive index : count from left, starting with 0
- Negative index : count from right, starting with -1

- **Sequence Types**
- List is a mutable ordered sequences of items of mixed types
- Lists are defined using square brackets (and commas)
- Tuple is a simple immutable ordered sequence of items
- Tuples are defined using paranthesis
- **Slicing and Copying**
- Slicing : return a copy of subset
- Return a copy of a container with a subset of the original members.
t [1:4]
- Start copying at the first index and stop copying before second

The Basics [6/7]



Dictionary

```
In [17]: # dictionary
mydictionary = {
    'name' : "Imansyah",
    'surname' : 'Basudewa',
    'age' : 21
}

print(mydictionary)

print(mydictionary['name'])

{'name': 'Imansyah', 'surname': 'Basudewa', 'age': 21}
Imansyah
```

- **Sequence Types**
- Dictionaries are data structures in which each particular value is associated with a particular label
- Data collected in the dictionary has no internal order but only the definition of a key-value pair

The Basics [7/7]

List of the other objects or variables

```
In [18]: # List of the other objects or variables
list_of_other_objects =[
    mylist,
    mytuple,
    3,
    "Ada",
    mydictionary
]

print(list_of_other_objects)
print()
for el in list_of_other_objects:
    print(el)
```

[[3, 8, 6, 4, 2], (1, 4, 5, 'Asja'), 3, 'Ada', {'name': 'Imansyah', 'surname': 'Basudewa', 'age': 21}]

[3, 8, 6, 4, 2]
(1, 4, 5, 'Asja')
3
Ada
{'name': 'Imansyah', 'surname': 'Basudewa', 'age': 21}

- **List of the other objects or variables**
- List can contains list, tuple, dictionary, numbers, strings

Operator [1/3]

jupyter AICOMS-Q_Python_Quick_Reference Last Checkpoint: Yesterday at 5:32 AM (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Run

Arithmetic operators

Basic operators

```
In [13]: a = 13
b = 5
print("a =",a)
print("b =",b)
print()

# basics operators
print("a + b =",a+b)
print("a - b =",a-b)
print("a * b =",a*b)
print("a / b =",a/b)
print("a mod b =",a%b)
print("a div b =",a//b)

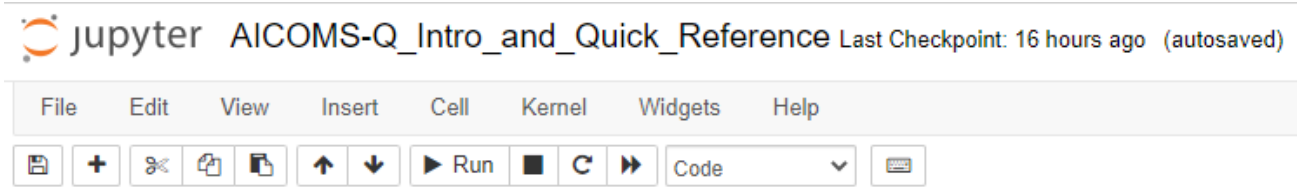
a = 13
b = 5

a + b = 18
a - b = 8
a * b = 65
a / b = 2.6
a mod b = 3
a div b = 2
```

Basic Arithmetics Operator

- Addition (+)
- Substraction (-)
- Multiplication (*)
- Division (/ or div)
- Power (** or pow())
- Modulo (%)

Operator [2/3]



Exponent operator

```
In [14]: b = 5
print("b =",b)
print()

print("b*b =",b**2)
print("b*b*b =",b**3)
print("sqrt(b)=",b**0.5)

#or
print("Or using pow() function:")
n=2
print("b^n =",pow(b,n))

b = 5

b*b = 25
b*b*b = 125
sqrt(b)= 2.23606797749979
Or using pow() function:
b^n = 25
```

Exponent operator

- We can use **
- Or using pow function pow()

Operator [3/3]

jupyter AICOMS-Q_Intro_and_Quick_Reference Last Checkpoint: 15 hours ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

Run Code

Comparison operator

```
In [38]: x = 7
y = 10

print('x =',x)
print('y =',y)
print('\n')

print('x == y the result is',x==y)
print('x != y the result is',x!=y)
print('x > y the result is',x>y)
print('x < y the result is',x<y)
print('x >= y the result is',x>=y)
print('x <= y the result is',x<=y)
```

```
x = 7
y = 10
```

```
x == y the result is False
x != y the result is True
x > y the result is False
x < y the result is True
x >= y the result is False
x <= y the result is True
```

Comparison operator

- == equal to
- != not equal to
- > more than
- < less than
- >= more than equal to
- <= less than equal to

Loops and Functions [1/3]

While-loop

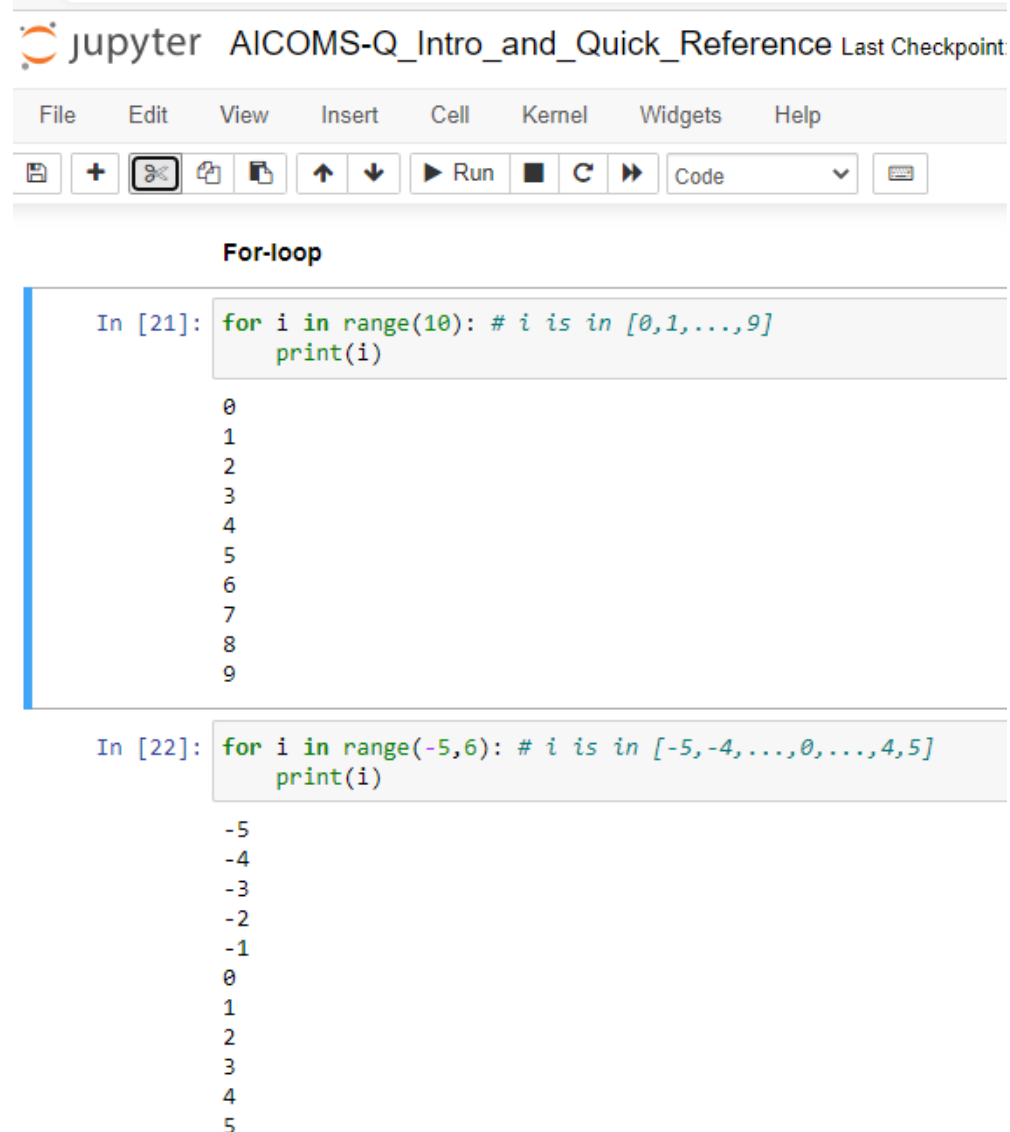
```
In [20]: i = 10
while i>0: # while condition(s):
    print(i)
    i = i - 1
print("check")
```

```
10
9
8
7
6
5
4
3
2
1
check
```

While-loop

- With the **while** loop we can execute a set of statements as long as a condition is true.

Loops and Functions [2/3]



The image shows a Jupyter Notebook interface with the title "AICOMS-Q_Intro_and_Quick_Reference Last Checkpoint". The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. The toolbar contains icons for saving, adding cells, running, and other standard Jupyter actions. The notebook content is titled "For-loop".

For-loop

```
In [21]: for i in range(10): # i is in [0,1,...,9]
          print(i)
0
1
2
3
4
5
6
7
8
9
```

```
In [22]: for i in range(-5,6): # i is in [-5,-4,...,0,...,4,5]
          print(i)
-5
-4
-3
-2
-1
0
1
2
3
4
5
```

A **for** loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).

This is less like the **for** keyword in other programming languages, and works more like an iterator method as found in other object-orientated programming languages.

With the **for** loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Loops and Functions [3/3]

jupyter AICOMS-Q_Intro_and_Quick_Reference Last Checkpoint: 16 hours ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help

Run

Functions

```
In [76]: def my_function(fname):  
         print(fname)  
  
         my_function("WORKSHOP")  
  
WORKSHOP
```

```
In [73]: def my_function(fname):  
         print(fname + " 2021")  
  
         my_function("AICOMS-Q")  
         my_function("WORKSHOP")  
  
AICOMS-Q 2021  
WORKSHOP 2021
```

Functions

- The advantage of function is that it can be used repeatedly with a variety of different values.
- The function more useful when it contains complex calculation

Library [1/5]

jupyter AICOMS-Q_Qiskit_installation Last Checkpoint: 20 hours ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Run

Check your system

Check your system, if Qiskit has already been installed:

```
In [11]: import qiskit
versions = qiskit.__qiskit_version__
print("The version of Qiskit is", versions['qiskit'])
print()
print("The version of each component:")
for key in versions:
    print(key, "->", versions[key])
```

The version of Qiskit is 0.29.1

The version of each component:
qiskit-terra -> 0.18.2
qiskit-aer -> 0.8.2
qiskit-ignis -> 0.6.0
qiskit-ibmq-provider -> 0.16.0
qiskit-aqua -> 0.9.5
qiskit -> 0.29.1
qiskit-nature -> None
qiskit-finance -> None
qiskit-optimization -> None
qiskit-machine-learning -> None

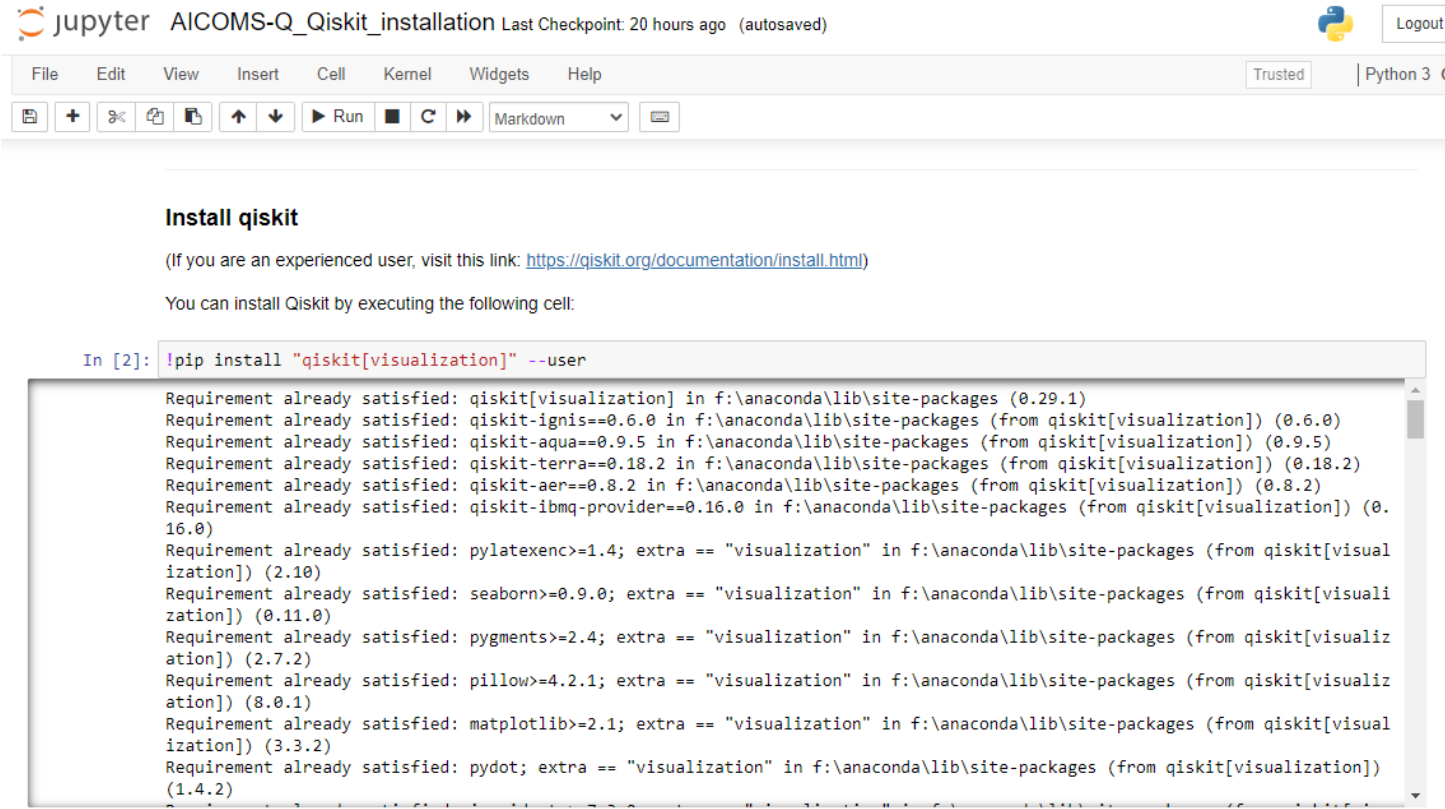
You should be able to see the version number of any library that is already installed in your system.

- Check the system
- Check the qiskit version with
`versions = qiskit.__qiskit_version__`

[Instalasi qiskit](#)

[Qiskit Textbook](#)

Library [2/5]



Jupyter AICOMS-Q_Qiskit_installation Last Checkpoint: 20 hours ago (autosaved) Logout

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

Install qiskit

(If you are an experienced user, visit this link: <https://qiskit.org/documentation/install.html>)

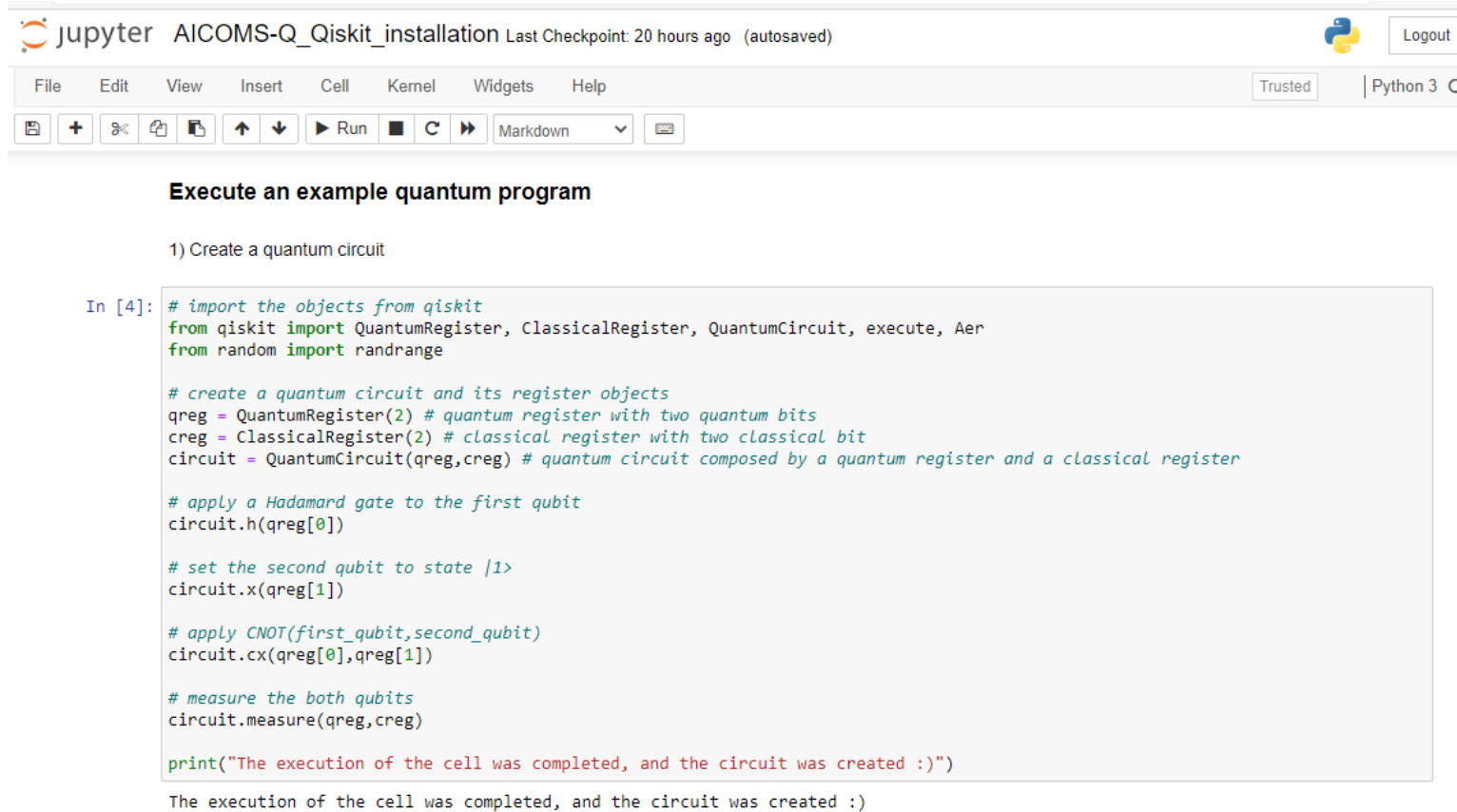
You can install Qiskit by executing the following cell:

```
In [2]: !pip install "qiskit[visualization]" --user
```

```
Requirement already satisfied: qiskit[visualization] in f:\anaconda\lib\site-packages (0.29.1)
Requirement already satisfied: qiskit-ignis==0.6.0 in f:\anaconda\lib\site-packages (from qiskit[visualization]) (0.6.0)
Requirement already satisfied: qiskit-aqua==0.9.5 in f:\anaconda\lib\site-packages (from qiskit[visualization]) (0.9.5)
Requirement already satisfied: qiskit-terra==0.18.2 in f:\anaconda\lib\site-packages (from qiskit[visualization]) (0.18.2)
Requirement already satisfied: qiskit-aer==0.8.2 in f:\anaconda\lib\site-packages (from qiskit[visualization]) (0.8.2)
Requirement already satisfied: qiskit-ibmq-provider==0.16.0 in f:\anaconda\lib\site-packages (from qiskit[visualization]) (0.16.0)
Requirement already satisfied: pylatexenc>=1.4; extra == "visualization" in f:\anaconda\lib\site-packages (from qiskit[visualization]) (2.10)
Requirement already satisfied: seaborn>=0.9.0; extra == "visualization" in f:\anaconda\lib\site-packages (from qiskit[visualization]) (0.11.0)
Requirement already satisfied: pygments>=2.4; extra == "visualization" in f:\anaconda\lib\site-packages (from qiskit[visualization]) (2.7.2)
Requirement already satisfied: pillow>=4.2.1; extra == "visualization" in f:\anaconda\lib\site-packages (from qiskit[visualization]) (8.0.1)
Requirement already satisfied: matplotlib>=2.1; extra == "visualization" in f:\anaconda\lib\site-packages (from qiskit[visualization]) (3.3.2)
Requirement already satisfied: pydot; extra == "visualization" in f:\anaconda\lib\site-packages (from qiskit[visualization]) (1.4.2)
```

- Install qiskit by using following syntax
- `!pip install "qiskit[visualization]" --user`
- Run the code in the cell

Library [3/5]



The image shows a JupyterLab interface with a title bar 'jupyter AICOMS-Q_Qiskit_installation' and a status bar 'Last Checkpoint: 20 hours ago (autosaved)'. The top menu bar includes 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. The top right has a 'Logout' button and a 'Python 3' selector. The left sidebar contains icons for file operations. The main area has a title 'Execute an example quantum program' and a sub-header '1) Create a quantum circuit'. Below this is a code cell with the following Python code:

```
In [4]: # import the objects from qiskit
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit, execute, Aer
from random import randrange

# create a quantum circuit and its register objects
qreg = QuantumRegister(2) # quantum register with two quantum bits
creg = ClassicalRegister(2) # classical register with two classical bit
circuit = QuantumCircuit(qreg,creg) # quantum circuit composed by a quantum register and a classical register

# apply a Hadamard gate to the first qubit
circuit.h(qreg[0])

# set the second qubit to state |1>
circuit.x(qreg[1])

# apply CNOT(first_qubit,second_qubit)
circuit.cx(qreg[0],qreg[1])

# measure the both qubits
circuit.measure(qreg,creg)

print("The execution of the cell was completed, and the circuit was created :)")
```

The execution output is: 'The execution of the cell was completed, and the circuit was created :)'

- Qiskit contains Quantum register, Classical register, Quantum Circuit, etc

[Circuit library](#)

Library [4/5]

jupyter AICOMS-Q_Qiskit_installation Last Checkpoint: 20 hours ago (autosaved)

File Edit View Insert Cell Kernel Widgets Help

Save + Undo Copy Paste Up Down Run Stop Restart Markdown

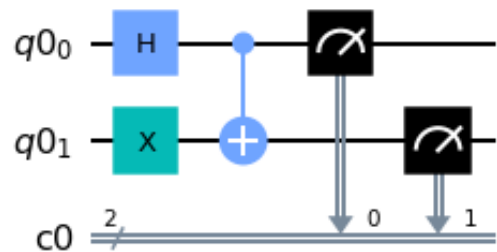
2) Draw the circuit

Run the cell once more if the figure is not shown

```
In [5]: # draw circuit
circuit.draw(output='mpl')

# the output will be a "matplotlib.Figure" object
```

Out[5]:



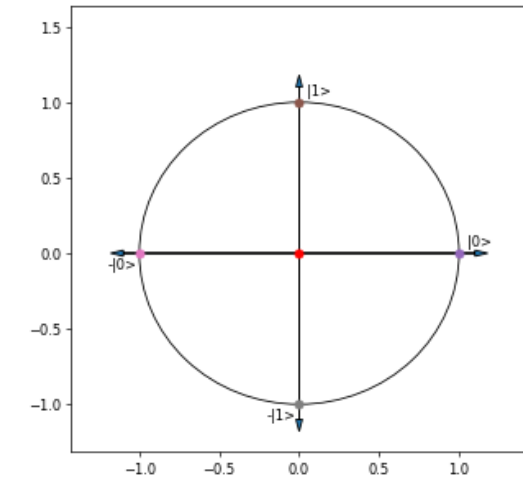
- Draw the circuit by running the code in the cell

Library [5/5]

```
In [9]: import matplotlib
def draw_qubit():
    # draw a figure
    matplotlib.pyplot.figure(figsize=(6,6), dpi=60)
    # draw the origin
    matplotlib.pyplot.plot(0,0,'ro') # a point in red color
    # drawing the axes by using one of our predefined functions
    draw_axes()
    # drawing the unit circle by using one of our predefined functions
    draw_unit_circle()
    # drawing  $|0\rangle$ 
    matplotlib.pyplot.plot(1,0,"o")
    matplotlib.pyplot.text(1.05,0.05," $|0\rangle$ ")
    # drawing  $|1\rangle$ 
    matplotlib.pyplot.plot(0,1,"o")
    matplotlib.pyplot.text(0.05,1.05," $|1\rangle$ ")
    # drawing  $-|0\rangle$ 
    matplotlib.pyplot.plot(-1,0,"o")
    matplotlib.pyplot.text(-1.2,-0.1," $-|0\rangle$ ")
    # drawing  $-|1\rangle$ 
    matplotlib.pyplot.plot(0,-1,"o")
    matplotlib.pyplot.text(-0.2,-1.1," $-|1\rangle$ ")
```

```
In [10]: draw_qubit()
```

- `draw_axes()` and `draw_unit_circle()` are functions
- Show functions by using syntax `draw_qubit()`



https://qiskit.org/documentation/intro_tutorial1.html