QUANTUM KICKSTART · PRE-COURSE SELF-CHECK

Ready for the Kickstart?

Tooling · Python · NumPy · complex numbers

A readiness diagnostic, not an entrance exam. Part 1: a working notebook and seven questions decide whether the course will work for you. Part 2: six questions decide what you prepare.
Allow 45–60 minutes; installation issues may take longer.

qubit-lab.ch

qubit-lab.ch
contact@qubit-lab.ch

September 2026 · © 2026 qubit-lab.ch · Proprietary content

How to use this document

The Quantum Kickstart is hands-on from the first morning: every concept on the slides is followed by a short piece of code that you run and change yourself. That only works if your Python environment and a few basic Python and NumPy skills are in place before you arrive. This document lets you check both.

Read this first. This is a readiness diagnostic, not an entrance exam. Only Part 1 decides anything. Part 2 looks ahead at the bits that cause friction during the course; a gap there means "spend an hour on this before the course", not "you are not ready for quantum computing". A result like **B 3/3 · C 4/4 · D 1/1 · E 2/3 · F 1/2** is a perfectly typical, well-prepared participant with one hour of preparation.

	Part	What it checks	
Part 1	A	A working Jupyter notebook with Qiskit on a laptop you bring	**Required**
	B	Python basics: loops, functions, lists, dictionaries	**Required**
	C	NumPy: vectors, matrices, matrix products and their order	**Required**
Part 2	D	Reading a Python decorator	Prepare
	E	Complex numbers: squares, conjugates, magnitudes	Prepare
	F	Polar form and radians	Prepare

Part 1: Can you run Python and NumPy? A working notebook and seven questions. This is the only part that decides whether the course will work for you. If one of the three does not work out, contact us before the course and we will agree on a short piece of preparation.

Part 2: What to prepare before day 1. Six short questions on decorators, complex numbers and angles. Nothing here decides whether you can join; it decides what you prepare. We strongly recommend understanding these questions before the course, with the videos and material listed at the end or your own LLM-assisted brush-up, so that the mornings can be spent on quantum computing rather than on catching up.

All questions are multiple choice about short pieces of code. Work out the answer on paper first, then run the code if you want to check. Every question has exactly one correct option. The answer key is at the end; its explanations are worth reading even for questions you got right, because each wrong option is a mistake that comes up during the course.

Time. Allow 45 to 60 minutes: 15 to 25 for the installation in Part A, about 20 for the questions. Installation problems on a company laptop can take longer, which is exactly why Part A comes first and comes early.

What you need. A laptop you can install software on, an internet connection for the installation (the course itself runs offline), and a pen. No quantum knowledge is assumed.

Two cheat sheets, whatever your result. Everything the course uses from linear algebra and complex numbers, each operation with its structure, one worked example and the NumPy call: qubit-lab.ch/Quantum_Kickstart_Cheat_Sheet_Vectors_Matrices.pdf (4 pages) and qubit-lab.ch/Quantum_Kickstart_Cheat_Sheet_Complex_Numbers.pdf (2 pages). Print them and keep them next to the laptop on day 1.

Part 1 — Can you run Python and NumPy?

Part A — A working notebook

REQUIRED

The goal of this part is one notebook cell that runs and prints a `READY` line. Follow the steps in order. On a normal laptop with a normal connection this takes 15 to 25 minutes, most of it waiting for the download.

> **Company laptop locked down? Use your private laptop.** Everything installed here is public, open-source software from python.org and PyPI: nothing is proprietary and nothing needs an account. The course uses **no corporate data** of any kind, so nothing from your employer ever needs to touch the machine. If your corporate laptop blocks installers or `pip`, do not spend hours with IT; run the course on a private laptop instead. That is the normal case for several participants, and it is the laptop you should bring on the first morning.

Step 1 · Check or install Python

Open a terminal (macOS: Terminal; Windows: Windows Terminal or PowerShell) and type:

```
python3 --version        # macOS / Linux
py --version             # Windows
```

You need Python **3.10 to 3.13**. If you get a version in that range, go to Step 2. Otherwise install Python from python.org/downloads. On Windows, tick **"Add python.exe to PATH"** on the first installer screen, then close and reopen the terminal. On macOS the installer works as is; Homebrew users can run `brew install python@3.13` instead.

Step 2 · Create a project folder and a virtual environment

A virtual environment is a private set of Python packages for this course. It keeps the course installation separate from anything else on your machine, and it is what we use during the sessions.

macOS / Linux:

```
mkdir ~/quantum-kickstart
cd ~/quantum-kickstart
python3 -m venv .venv
source .venv/bin/activate
```

Windows (PowerShell):

```
mkdir $HOME\quantum-kickstart
cd $HOME\quantum-kickstart
py -m venv .venv
.venv\Scripts\activate
```

After the last command the prompt starts with `(.venv)`. That prefix means the environment is active; every `pip` and `jupyter` command from now on refers to it.

> **Windows: use the native installer, not WSL2.** For this course the plain Windows Python from python.org is the simplest path; a Linux subsystem adds nothing the notebooks need and makes JupyterLab harder to reach. If your machine only allows WSL2 with Ubuntu, follow the macOS / Linux commands inside Ubuntu; `jupyter lab` then prints a `http://localhost:8888/lab?token=...` URL that you open in your normal Windows browser.

> **Windows says "running scripts is disabled on this system".** PowerShell blocks the activate script by default. Run `Set-ExecutionPolicy -Scope CurrentUser RemoteSigned` once, answer `Y`, then repeat the activate command.

Step 3 · Install the packages

With the environment active:

```
pip install --upgrade pip
pip install "qiskit>=2,<3" numpy matplotlib jupyterlab ipykernel pylatexenc
```

This downloads roughly 300 MB and takes two to five minutes. The last lines should read `Successfully installed ...` followed by a long list of packages. A warning about a newer pip version is harmless.

Optional, and worth trying now that the environment is active: the noise-model exercises use qiskit-aer, the framework comparison in Session 1 shows the same circuit in PennyLane and Cirq, and the Session 5 algorithms use scipy and scikit-learn.

```
pip install qiskit-aer pennylane cirq scipy scikit-learn
```

If one of these fails to install on your machine, everything else still works; Cirq is the one most likely to be blocked on a company laptop, and the course does not depend on it.

Step 4 · Start JupyterLab and create a notebook

Still in the same terminal:

```
jupyter lab
```

Your browser opens at `http://localhost:8888/lab`. If it does not, copy the URL printed in the terminal (it contains a token) into the browser. Leave the terminal window open; closing it stops the notebook server.

In the launcher page click **Python 3 (ipykernel)** under Notebook. An empty notebook opens with one cell. Rename it via File → Rename Notebook to `setup-check.ipynb`.

Step 5 · Run the test cell

Paste the following into the first cell and press **Shift+Enter**. The same code is in `kickstart-s0-setup-check.ipynb`, which came with this document; opening that file in JupyterLab and running its first cell is equivalent.

```
import sys, platform
import numpy as np
import matplotlib
import qiskit
from qiskit import QuantumCircuit
from qiskit.quantum_info import Statevector
from qiskit.primitives import StatevectorSampler

print("python    ", sys.version.split()[0], "on", platform.system())
print("numpy     ", np.__version__)
print("matplotlib", matplotlib.__version__)
print("qiskit    ", qiskit.__version__)
assert int(qiskit.__version__.split(".")[0]) >= 2, "This course needs Qiskit 2.x"

# 1. One gate, computed twice: by hand with NumPy and by Qiskit
H = np.array([[1, 1], [1, -1]]) / np.sqrt(2)
psi = H @ np.array([1, 0])
qc = QuantumCircuit(1)
qc.h(0)
sv = Statevector(qc).data
assert np.allclose(psi, sv), "NumPy and Qiskit disagree"
print("amplitudes   ", np.round(sv, 4))
print("probabilities", np.round(np.abs(sv) ** 2, 4))

# 2. Measure the circuit 1000 times
qc.measure_all()
print(qc.draw())
job = StatevectorSampler(seed=42).run([qc], shots=1000)
counts = job.result()[0].data.meas.get_counts()
print("counts       ", dict(sorted(counts.items())))

# 3. The line to send us
print(f"\nREADY | python {sys.version.split()[0]} | qiskit {qiskit.__version__}"
      f" | counts {dict(sorted(counts.items()))}")
```

Expected output. Your Python and package versions will differ; everything else should match, including the counts:

```
python     3.13.9 on Darwin
numpy      2.3.5
matplotlib 3.10.7
qiskit     2.5.2
amplitudes    [0.7071+0.j 0.7071+0.j]
probabilities [0.5 0.5]
     ┌───┐ ░ ┌─┐
  q: ┤ H ├─░─┤M├
     └───┘ ░ └╥┘
meas: 1/═══════╩═
               0

counts        {'0': 503, '1': 497}

READY | python 3.13.9 | qiskit 2.5.2 | counts {'0': 503, '1': 497}
```

What just happened is Session 1 in miniature: a gate as a 2×2 matrix, applied to a vector with `@`, cross-checked against Qiskit's own simulator, then measured 1000 times. If you can read most of that code already, the course will feel familiar quickly. If not, that is what the course is for.

> **Copy the** `READY` **line** into your reply to the confirmation e-mail. The counts are identical for everyone who uses the same seed, so they double as a check that the sampler works.

Step 6 · Second cell: a plot

Click the empty cell below the output (or press **B** while the first cell is selected to insert one), paste this and press **Shift+Enter**:

```
from qiskit.visualization import plot_histogram
plot_histogram(counts)
```

A bar chart with two bars near 0.5 appears below the cell. If it does, plotting inside the notebook works. If you only see text like `<Figure size ...>` and no picture, add `%matplotlib inline` as the first line of the cell and run it again. Save the notebook with **Ctrl+S** (macOS: **Cmd+S**). Done.

If something did not work

`pip` **cannot connect or times out.** You are behind a corporate proxy or a firewall that blocks `pypi.org` and `files.pythonhosted.org`. IT can configure the proxy for pip or whitelist those two hosts, but in practice the fastest fix is the private laptop (see the box at the start of Part A).

No laptop you can install on at all. Tell us in your reply and we will sort out an alternative with you before the first morning. One option is Google Colab (a notebook in the browser, Google account needed): open **colab.research.google.com**, create a notebook, run `%pip install "qiskit>=2,<3" pylatexenc -q` in the first cell, then paste the test cell from Step 5 into the second.

You prefer VS Code. Install the Python and Jupyter extensions, open the `quantum-kickstart` folder, create a file `setup-check.ipynb`, and pick the `.venv` interpreter when the notebook asks for a kernel. Steps 5 and 6 then work the same way.

Apple Silicon Macs, Intel Macs, Linux, Windows are all fine. Qiskit 2.x publishes pre-built packages for all of them.

Next time

Each time you come back to the course folder, activate the environment and start JupyterLab:

```
cd ~/quantum-kickstart
source .venv/bin/activate      # Windows: .venv\Scripts\activate
jupyter lab
```

Bring the laptop with this working on the first morning; the session notebooks go into the same folder.

Part B — Python basics

REQUIRED

Three questions. For each one, decide what the code prints.

B1

```
values = [3, 1, 4, 1, 5]
total = 0
for i, v in enumerate(values):
    if i % 2 == 0:
        total += v
print(total, len(values) / 2)
```

- ☐ (a) `12 2`
- ☐ (b) `9 2.5`
- ☐ (c) `12 2.5`
- ☐ (d) `2 2.5`

B2 A quantum computer returns its results as a dictionary of bitstrings and how often each one was observed. What does this print?

```
counts = {"000": 1012, "011": 988, "101": 4001, "110": 1999}
best = max(counts, key=counts.get)
print(best, int(best, 2), sum(counts.values()))
```

- ☐ (a) `101 5 8000`
- ☐ (b) `4001 5 8000`
- ☐ (c) `101 101 8000`
- ☐ (d) `110 6 8000`

B3

```
def scale(xs, factor=2):
    return [x * factor for x in xs if x > 1]

out = scale([1, 2, 3], 3)
print(f"{out} -> {sum(out):.1f}")
```

- ☐ (a) `[3, 6, 9] -> 18.0`
- ☐ (b) `[2, 4, 6] -> 12.0`
- ☐ (c) `[6, 9] -> 15`
- ☐ (d) `[6, 9] -> 15.0`

Part C — NumPy: vectors and matrices

REQUIRED

Four questions; `np` is `numpy` throughout. Vectors and matrices are the language of the whole course: a qubit state is a vector, a gate is a matrix, and applying a gate is a matrix product.

C1 Two products that look alike and are not.

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[0, 1], [1, 0]])
print(A * B)
print(A @ B)
```

- ☐ (a) both print `[[0 2] [3 0]]`
- ☐ (b) `[[0 2] [3 0]]` then `[[2 1] [4 3]]`
- ☐ (c) `[[2 1] [4 3]]` then `[[0 2] [3 0]]`
- ☐ (d) both print `[[2 1] [4 3]]`

C2 A matrix applied to a vector, then applied twice.

```
M = np.array([[1, 1], [1, -1]])
v = np.array([1, 0])
print(M @ v, M @ M @ v)
```

- ☐ (a) `[1 1] [2 0]`
- ☐ (b) `[1 1] [1 1]`
- ☐ (c) `[1 0] [2 0]`
- ☐ (d) `[2 0] [1 1]`

C3 Order matters. `Z` flips the sign of the second component; `S` swaps the two components.

```
Z = np.array([[1, 0], [0, -1]])
S = np.array([[0, 1], [1, 0]])
v = np.array([1, 0])
print(S @ Z @ v)
print(Z @ S @ v)
```

- ☐ (a) The order does not matter for 2×2 matrices; both lines print `[0 1]`.
- ☐ (b) Both lines print `[ 0 -1]`.
- ☐ (c) In `S @ Z @ v` the matrix `Z` acts on `v` first; the output is `[0 1]` then `[ 0 -1]`.
- ☐ (d) In `S @ Z @ v` the matrix `S` acts on `v` first; the output is `[ 0 -1]` then `[0 1]`.

C4 Transpose, identity and length.

```
A = np.array([[1, 2], [3, 4]])
I = np.eye(2)
v = np.array([3, 4])
print(A.T[0, 1], np.allclose(A @ I, A), np.linalg.norm(v))
```

- ☐ (a) `2 True 5.0`
- ☐ (b) `3 False 7.0`
- ☐ (c) `3 True 25.0`
- ☐ (d) `3 True 5.0`

Part 2 — What to prepare before day 1

Six short questions on decorators, complex numbers and angles. Nothing here decides whether you can join; it decides what you prepare. We strongly recommend understanding these questions before the course, with the videos and material listed under "Scoring" or your own LLM-assisted brush-up, so that the mornings can be spent on quantum computing rather than on catching up.

Part D — Reading a decorator

PREPARE

One question. Quantum frameworks such as PennyLane mark a function as a quantum circuit by writing `@qml.qnode(dev)` above it. You never have to write a decorator in this course, but you will read them, so it helps to know what the `@` line does.

D1

```
def normalise(fn):
    def wrapper(*args):
        v = fn(*args)
        return v / np.linalg.norm(v)
    return wrapper

@normalise
def state(a, b):
    return np.array([a, b], dtype=float)

print(state(3, 4))
```

- ☐ (a) `<function normalise.<locals>.wrapper ...>`
- ☐ (b) `[0.6 0.8]`
- ☐ (c) `[3. 4.]`
- ☐ (d) an error, because `wrapper` takes no named arguments

Part E — Complex numbers

PREPARE

Three questions. In Python the imaginary unit is written `1j`. Qubit amplitudes are complex numbers, and probabilities come from their squared magnitude, so the difference between z^2 and $|z|^2$ matters from the first morning.

E1 (1 + i)² equals

- ☐ (a) 2 + 2i
- ☐ (b) 0
- ☐ (c) 2i
- ☐ (d) 2

E2 1 / i equals

- ☐ (a) −i
- ☐ (b) i
- ☐ (c) 1
- ☐ (d) −1

E3 What does this print?

```
z = 3 + 4j
print(z * z.conjugate(), z**2, abs(z))
```

- ☐ (a) `(25+0j) (25+0j) 5.0`
- ☐ (b) `(9+16j) (-7+24j) 5.0`
- ☐ (c) `(25+0j) (-7+24j) 25.0`
- ☐ (d) `(25+0j) (-7+24j) 5.0`

Part F — Polar form and radians

PREPARE

Two questions. Every rotation gate in the course takes an angle in radians, and every phase is a complex number of the form $e^{i\varphi}$.

F1 $z_1 = 2e^{i\pi/4}$ and $z_2 = 3e^{i\pi/4}$. Then $z_1 \cdot z_2$ equals

- ☐ (a) $6i$
- ☐ (b) $5e^{i\pi/2}$
- ☐ (c) $6e^{i\pi^2/16}$
- ☐ (d) 6

F2 What does this print?

```
theta = np.pi / 2
c, s = round(np.cos(theta / 2), 3), round(np.sin(theta / 2), 3)
print(c, s, np.angle(1j) / np.pi)
```

- ☐ (a) `0.707 0.707 90.0`
- ☐ (b) `0.5 0.5 0.5`
- ☐ (c) `0.707 0.707 0.5`
- ☐ (d) `1.0 0.014 0.5`

Scoring and what to do next

Count your correct answers per part with the key on the last pages and fill in the last column. Then send us one line together with the `READY` line from Part A, for example `A ok, B 3/3, C 4/4, D 1/1, E 2/3, F 1/2`.

Part 1 — does the course work for you?

Part	Ready	Brush up first	Talk to us first	My result
A Tooling	READY printed	—	anything else	
B Python	3/3	2/3	1 or fewer	
C NumPy	4/4	3/4	2 or fewer	

Brush up means one to two hours with the material below is enough. **Talk to us first** does not mean you cannot attend; it means we should agree on preparation together, because the course does not have time to teach these basics from scratch.

Part 2 — what to prepare

Part	All correct	Otherwise, before the course	My result
D Decorators	1/1	Python decorators, about 30 minutes	
E Complex numbers	3/3	Complex numbers, polar form and radians, about 1 hour	
F Polar form, radians	2/2	(same hour as E)	

> **Reading your result.** Part 1 right, Part 2 patchy? That is the intended participant. Your computational language is ready; an hour on complex numbers before the course and you will not be slowed down on day two. Please do that preparation before the first morning. Only a gap in Part 1 needs a conversation before the course.

Material for brushing up

Python (B). The official tutorial, chapters 3 to 5, covers everything the course uses: docs.python.org/3/tutorial. Two hours.

Cheat sheets. Two short sheets in the course style, four and two pages, free to download: qubit-lab.ch/Quantum_Kickstart_Cheat_Sheet_Vectors_Matrices.pdf (scalar, matrix and vector products with their structure and a worked example, length, conjugate and dagger, tensor product up to a 4×4 gate, the NumPy calls) and qubit-lab.ch/Quantum_Kickstart_Cheat_Sheet_Complex_Numbers.pdf (cartesian and polar form, Euler, conjugate, the worked examples of Part 2, radians, the NumPy calls).

NumPy (C). The vectors-and-matrices cheat sheet above, then NumPy: the absolute basics for beginners on numpy.org (numpy.org/doc/stable/user/absolute_beginners.html), one hour. For the idea behind matrix products and their order: 3Blue1Brown, Essence of linear algebra on YouTube, chapters 3 (Linear transformations and matrices, youtu.be/kYB8IZa5AuE) and 4 (Matrix multiplication as composition, youtu.be/XkY2DOUCWMU), about 25 minutes together. Chapter 4 is exactly the right-to-left logic of C3.

For Part 2 topics, an LLM-assisted brush-up works just as well: paste the question you missed and ask for the idea behind it, then try the question again.

Decorators (D). Real Python, Primer on Python Decorators (realpython.com/primer-on-python-decorators), the first half is enough. Or Corey Schafer's YouTube video Python Tutorial: Decorators (youtu.be/FsAPt_9Bf3U), 30 minutes.

Complex numbers (E, F). Welch Labs, Imaginary Numbers Are Real on YouTube, parts 1 to 5 (youtu.be/T647CGsuOVU), about 40 minutes, for the geometric picture. 3Blue1Brown, e^(iπ) in 3.14 minutes (youtu.be/v0YEaeIClKY) for Euler's formula. The complex-numbers cheat sheet above covers exactly Parts E and F, including the NumPy functions `abs`, `np.angle`, `np.conj` and `np.exp`.

> **Not required, in case you were wondering.** Physics. Quantum mechanics. Bra-ket notation (the course introduces it and the handout has a dictionary). Probability theory beyond "probabilities add up to 1". Writing your own classes. Machine learning (Session 5 uses scikit-learn as a black box). Any cloud account or hardware access.

Answer key

The wrong options are not random; each one is a mistake we see during the course.

Question	Why
B1 (c)	`enumerate` starts at 0, so indices 0, 2, 4 are summed: 3+4+5. `/` always gives a float. (a) is integer-division thinking, (b) is 1-based counting.
B2 (a)	`max` with `key=counts.get` returns the key with the largest value, not the value. `int("101", 2)` reads the string as binary: 5.
B3 (d)	The comprehension keeps only `x > 1`, then multiplies by the explicit factor 3, not the default 2. `:.1f` prints one decimal.
C1 (b)	`*` multiplies element by element; `@` is the matrix product. Confusing them is the most common NumPy bug in quantum code.
C2 (a)	`M @ v` is the first column of `M`: `[1 1]`. Applying `M` again: `[1+1, 1-1]` = `[2 0]`. (`M` is the Hadamard gate without its 1/√2.)
C3 (c)	Products are evaluated left to right, but the matrix next to `v` acts first. `S @ Z @ v` means "Z, then S". In a circuit diagram the gates are drawn in the opposite order from how they appear in the product.
C4 (d)	`A.T[0, 1]` is `A[1, 0]` = 3. Multiplying by the identity changes nothing. The norm of `[3, 4]` is √(9+16) = 5, not 25: the norm is the length, the squared norm is 25.
D1 (b)	`state(3, 4)` actually calls `wrapper(3, 4)`, which calls the original function and divides by its norm 5.
E1 (c)	$1 + 2i + i^2 = 1 + 2i - 1 = 2i$.
E2 (a)	Multiply numerator and denominator by i: $i / i^2 = i / (-1) = -i$.
E3 (d)	$z \cdot \text{conj}(z) = 9 + 16 = 25$ is $\lvert z\rvert^2$; $z^2 = -7 + 24i$ is something else entirely; `abs(z)` is $\lvert z\rvert = 5$. Amplitude to probability is always $\lvert z\rvert^2$, never z^2.
F1 (a)	Moduli multiply (2 · 3 = 6), angles add (π/4 + π/4 = π/2): $6e^{i\pi/2} = 6i$.
F2 (c)	NumPy trigonometry works in radians: cos(π/4) = sin(π/4) ≈ 0.707; the angle of i is π/2. Option (d) is what you get if you feed degrees.

If you got everything right in Parts B to F, you are over-prepared for the maths and can spend the course on the physics and the code. See you on the first morning.