Qiskit
Reference Deck

Must-Know Concepts, Explained (Qiskit 2.x)

Coverage	SDK Architecture, Building Circuits, Transpilation & ISA Circuits, Primitives (Sampler/Estimator V2), Running on Hardware, Aer & Simulation, Quantum Info & Operators, Advanced Features
Format	Question → Concise answer → Python code where useful
Level	Quantum Software Engineer to Applied Researcher rounds

Reflects the Qiskit 2.x generation (post Qiskit 1.0/2.0 stabilization – primitives-first workflow, ISA circuits, Pulse removed, `qiskit-aer` as a separate package). Qiskit releases frequently; always check the release notes for the exact version you install.

Generated September 13, 2026

1 SDK Architecture & Core Concepts 2

2 Building Circuits 4

3 Transpilation & ISA Circuits 6

4 Primitives: Sampler & Estimator 8

5 Running on Hardware & Qiskit Runtime 10

6 Qiskit Aer & Simulation 12

7 Quantum Info & Operators 14

8 Advanced Features 16

SDK Architecture & Core Concepts

Q1.1

What is Qiskit, and how is the SDK split into separate packages since the Qiskit 1.0 stabilization?

Qiskit is IBM's open-source Python SDK for building, transpiling, and running quantum circuits, both on local simulators and real superconducting quantum hardware accessed via IBM Quantum. Since the Qiskit 1.0 release, the project deliberately split what used to be one large, monolithic package into focused, independently-versioned components: `qiskit` (the core SDK – circuit construction, transpiler, quantum_info, primitives interfaces), `qiskit-aer` (high-performance local simulators, now installed separately via `pip install qiskit-aer`), and `qiskit-ibm-runtime` (the client for submitting jobs to real IBM Quantum hardware and cloud-based primitives). This split lets each component evolve, version, and release on its own schedule, rather than forcing every user to take on simulator or cloud-runtime dependencies just to build and manipulate circuits locally.

Q1.2

What is a `QuantumCircuit`, and what are its two fundamental register types?

`QuantumCircuit` is Qiskit's central data structure representing a quantum program as an ordered sequence of instructions (gates, measurements, control-flow blocks) acting on **qubits** and, for measurement outcomes, **classical bits**. A circuit is built from a `QuantumRegister` (a named group of qubits) and a `ClassicalRegister` (a named group of classical bits used to store measurement results) – though for quick scripting, `QuantumCircuit(n, m)` directly creates n qubits and m classical bits without needing to explicitly instantiate register objects.

```
from qiskit import QuantumCircuit

qc = QuantumCircuit(2, 2)    # 2 qubits, 2 classical bits
qc.h(0)                      # Hadamard on qubit 0
qc.cx(0, 1)                  # CNOT: qubit 0 controls qubit 1
qc.measure([0, 1], [0, 1])   # measure both qubits into both clbits
```

Q1.3

Why does Qiskit's current, recommended execution model center on `Primitives` (Sampler and Estimator) rather than the older `backend.run()` pattern?

The older pattern (`backend.run(circuit)`) returns only raw measurement counts, leaving the user responsible for manually converting those counts into whatever higher-level quantity (an expectation value, a probability distribution) their algorithm actually needs – and provides no standard, built-in hooks for error mitigation. **Primitives** instead expose two task-oriented interfaces matching what most quantum algorithms actually want: `Sampler` returns quasi-probability distributions over measurement outcomes, and `Estimator` directly returns expectation values of specified observables – with built-in, configurable error suppression/mitigation particularly important for running on real, noisy hardware. This primitives-first model is now the standard, actively promoted way to write portable Qiskit algorithm code, whether targeting a local simulator or real IBM Quantum hardware, since both expose the same `Sampler`/`Estimator` interface.

Q1.4

What is a PUB (Primitive Unified Bloc), and why did the V2 primitives introduce this concept over the V1 primitives' simpler, separate-list-based interface?

A PUB bundles everything one "unit of work" submitted to a primitive needs into a single tuple: a circuit, plus (for

`Estimator`) the observable(s) to measure, plus any parameter values to bind, plus an optional per-PUB shot count. V1 primitives instead required passing separate, index-aligned lists of circuits, observables, and parameter values, forcing awkward duplication (e.g. repeating the same circuit multiple times just to pair it with different parameter sets or observables) and making it error-prone to keep those parallel lists correctly aligned. The PUB structure lets a single circuit be efficiently paired with *many* observables and/or parameter-value sweeps in one call, and is the reason V2 primitives (`SamplerV2`, `EstimatorV2`) are the current standard – V1 primitives were fully removed from `qiskit-ibm-runtime` in August 2024.

```
from qiskit.primitives import StatevectorEstimator as Estimator
from qiskit.quantum_info import SparsePauliOp

estimator = Estimator()
observable = SparsePauliOp("ZZ")
job = estimator.run([(qc, observable)])    # a single PUB: (circuit, observable)
result = job.result()[0].data.evs          # expectation value(s)
```

Q1.5

What is the difference between `BackendV1` and `BackendV2`, and why does essentially all current Qiskit tooling now assume the `Target`-based `BackendV2` interface?

`BackendV1` described a backend's capabilities through several separate, loosely-connected pieces (a `configuration` object, a `properties` object for calibration data, and a `defaults` object for pulse defaults) that could drift out of sync and were awkward to query programmatically. `BackendV2` consolidates all of this into a single, unified **Target** object – a structured description of exactly which instructions the backend supports, on which qubits/qubit-pairs, together with their duration, error rate, and other calibration data, all in one queryable place. Modern Qiskit's transpiler, primitives, and provider integrations are all built directly around the `Target`/`BackendV2` model, and `BackendV1` support has been fully removed from current Qiskit – any lingering `BackendV1`-based provider code needs to be migrated to `BackendV2` to work with current Qiskit versions.

Building Circuits

Q2.1

What is a `Parameter`, and why is building a circuit with symbolic, unbound parameters preferable to constructing a brand-new circuit object for every different set of rotation angles?

A `Parameter` is a named, symbolic placeholder that can be used in place of a concrete numerical value anywhere a circuit expects one (most commonly, a gate's rotation angle) – letting you build the circuit's *structure* exactly once, then bind different concrete numerical values to it later, as many times as needed. This matters enormously for variational algorithms (VQE, QAOA) and parameter sweeps, where the same circuit structure must be evaluated repeatedly with many different parameter values during optimization – building and transpiling a fresh circuit object from scratch for every iteration would be wasteful, since transpilation (Section 3) is one of the more expensive steps in the pipeline; with a parameterized circuit, the expensive transpilation step is performed *once*, and only cheap numerical parameter binding is repeated on each iteration.

```
from qiskit.circuit import Parameter

theta = Parameter("theta")
qc = QuantumCircuit(1)
qc.ry(theta, 0)

bound_qc = qc.assign_parameters({theta: 1.57})    # bind a concrete value
```

Q2.2

What is the difference between `compose()` and simply appending gates directly onto an existing `QuantumCircuit`, and when is `compose()` the right tool?

Directly calling gate methods (`qc.h(0)`, `qc.cx(0,1)`) appends individual instructions one at a time to a single circuit object. `compose()` instead takes a *separate*, already-built `QuantumCircuit` (often one representing a reusable subroutine – a state-preparation block, an entangling layer, or a previously-built ansatz) and inserts its entire instruction sequence into another circuit at a specified qubit/clbit mapping, optionally at a specific insertion point. This is the standard way to build larger circuits compositionally out of smaller, independently-constructed, and independently-testable building blocks, rather than writing one long, monolithic sequence of gate calls – particularly valuable when the same reusable sub-circuit (e.g. a Bell-pair preparation routine) needs to be inserted into several different larger circuits.

Q2.3

What is the `qiskit.circuit.library`, and why does using its pre-built circuit constructs (like `EfficientSU2` or `QFT`) tend to be preferable to hand-writing the equivalent gate sequence yourself?

The circuit library is a collection of pre-built, parameterized, well-tested circuit constructs for common patterns – variational ansätze (`EfficientSU2`, `TwoLocal`, `RealAmplitudes`), the Quantum Fourier Transform (`QFT`), and various standard state-preparation and arithmetic circuits. Using these is generally preferable to hand-rolling the equivalent gate sequence because they're already validated for correctness, expose sensible, well-documented configuration options (entanglement pattern, repetition count, rotation gate choice) covering the vast majority of practical variations users actually need, and are maintained/optimized by the Qiskit team itself – reducing both the risk of a subtle hand-written bug and the amount of boilerplate circuit-construction code a project needs to maintain.

```
from qiskit.circuit.library import EfficientSU2
```

```
ansatz = EfficientSU2(num_qubits=4, reps=2, entanglement="linear")
print(ansatz.num_parameters)   # ready-to-optimize parameterized ansatz
```

Q2.4

What is the modern, recommended way to build classically-conditioned logic in a circuit (an `if_test` block), and why was the older `c_if` method removed in Qiskit 2.0?

The legacy `c_if` method attached a classical condition directly onto a single gate call (e.g. `qc.x(0).c_if(creg, 1)`), which only supported simple register-equality conditions and didn't compose well with more expressive classical logic or nested control flow. Qiskit 2.0 removed `c_if` entirely in favor of the `if_test` context manager, which wraps a genuine `IfElseOp` block around one or more instructions, supporting richer conditions (including expressions built from Qiskit's classical-expression system, not just simple register equality) and nesting cleanly with other control-flow constructs like `while_loop` and `for_loop` – part of Qiskit's broader, ongoing investment in **dynamic circuits** (classical control flow interleaved with quantum operations, mid-circuit measurement feeding real-time decisions) as a first-class capability.

```
qc = QuantumCircuit(2, 2)
qc.h(0)
qc.measure(0, 0)
with qc.if_test((qc.clbits[0], 1)):   # if clbit 0 measured as 1
    qc.x(1)
```

Transpilation & ISA Circuits

Q3.1

What does `transpile()` actually do, and why is a circuit written in "any" gates almost never directly runnable on real hardware without it?

Real quantum hardware only natively supports a small, fixed set of **basis gates** (e.g. a specific single-qubit rotation plus one two-qubit entangling gate) and has restricted **qubit connectivity** (only certain physical qubit pairs can be directly entangled) – a circuit written using arbitrary, convenient gates (a Toffoli, an arbitrary controlled-U, a SWAP between two non-adjacent qubits) generally cannot run as-is. `transpile()` (or, more flexibly, a `PassManager` built via `generate_preset_pass_manager()`) rewrites the circuit into an equivalent one expressed entirely in the target backend's native gates, with two-qubit operations remapped only onto physically connected qubit pairs (inserting SWAP gates where necessary), while also applying various optimization passes to reduce the resulting gate count and circuit depth – since deeper circuits accumulate more noise/decoherence on real, imperfect hardware.

```
from qiskit.transpiler.preset_passmanagers import generate_preset_pass_manager

pm = generate_preset_pass_manager(optimization_level=3, backend=backend)
isa_circuit = pm.run(qc)
```

Q3.2

What is an "ISA circuit," and why does Qiskit Runtime require every circuit submitted to a primitive to already be transpiled into this form before it's accepted?

An ISA (Instruction Set Architecture) circuit is one that has already been transpiled to use *only* instructions and qubit connections genuinely present in the target backend's `Target` – exactly matching what that specific piece of hardware can actually execute natively. Since March 2024, Qiskit Runtime **requires** every circuit (and, for `Estimator`, every associated observable) submitted to a primitive to already be in ISA form – submitting a non-ISA circuit raises an explicit error rather than being silently, automatically transpiled server-side. This shift makes transpilation an explicit, visible step the user controls and reasons about directly (rather than an invisible, potentially inconsistent step performed automatically behind the scenes), and it means an observable like a `SparsePauliOp` must also have its qubit **layout** explicitly applied (`observable.apply_layout(isa_circuit.layout)`) to correctly match the transpiled circuit's possibly-permuted qubit ordering.

Q3.3

What do Qiskit's four transpiler `optimization_level` settings (0 through 3) control, and what is the practical trade-off between choosing a higher versus a lower level?

The optimization level controls how much effort the transpiler's preset pass managers spend trying to reduce the final circuit's gate count/depth and improve qubit-layout selection: **level 0** does the bare minimum needed to make the circuit runnable (basic translation and routing, essentially no optimization); **level 1** adds light optimization (removing some redundant gates); **level 2** performs more thorough gate cancellation and improved routing/layout heuristics; **level 3** runs the most aggressive available optimization passes, including more expensive synthesis and noise-adaptive layout selection. The practical trade-off is straightforward: higher levels generally produce shorter, less noisy circuits (better results on real hardware) at the cost of significantly longer *classical* transpilation time – for quick local testing, level 1 is often adequate, while level 3 is typically worth its extra compile time before submitting a circuit to actual, error-prone quantum hardware.

Q3.4

What replaced the deprecated `StochasticSwap` routing pass as Qiskit's default routing algorithm, and what specific advantage does it offer?

`StochasticSwap` (deprecated in Qiskit 1.3, removed in 2.0) inserted SWAP gates using a randomized search heuristic to route two-qubit operations onto connected physical qubits. **SabreSwap** (Sabre = SWAP-based Bidirectional heuristic search) is now the default routing pass, using a more sophisticated bidirectional lookahead search that considers both the circuit's forward and reverse instruction order when deciding where to insert SWAPs and which initial qubit layout to choose – consistently producing measurably fewer inserted SWAP gates (and therefore shallower, less noisy final circuits) than the older stochastic approach, while also typically running faster during transpilation itself.

Primitives: Sampler & Estimator

Q4.1

What is the difference between `SamplerV2` and `EstimatorV2`, and how do you decide which one a given algorithm needs?

`SamplerV2` runs a circuit (which must include measurements) and returns the **raw measurement outcomes** – bitstring counts/quasi-probabilities – appropriate whenever an algorithm's actual output *is* a probability distribution over bitstrings (e.g. Grover's algorithm, QAOA's final solution readout, or general circuit sampling). `EstimatorV2` instead takes a circuit *without* measurements plus one or more observables, and directly returns **expectation values** $\langle\psi|\hat{O}|\psi\rangle$ – appropriate whenever the actual quantity of interest is an expectation value rather than a raw distribution (e.g. VQE's energy estimate, or any expectation-value-based cost function). Choosing the right primitive up front avoids manually computing an expectation value yourself from raw `Sampler` counts, a step `Estimator` already handles internally, including applying any configured error mitigation.

Q4.2

What does the `resilience_level` option on `EstimatorV2` control when running on real IBM Quantum hardware, and what specific error-mitigation technique does the higher levels typically enable?

`resilience_level` trades off result accuracy against classical/quantum runtime overhead by selecting which built-in error suppression and mitigation techniques Qiskit Runtime automatically applies. **Level 0** applies no mitigation at all (fastest, rawest result). **Level 1** (the default) applies twirled readout error extinction (TREX) to mitigate measurement/readout errors. **Level 2** additionally applies **Zero Noise Extrapolation (ZNE)** – deliberately running the circuit at several artificially amplified noise levels and extrapolating the resulting expectation values back to the (unreachable) zero-noise limit – at the cost of several times more total shots/runtime, since each noise-amplification level requires its own separate execution. Choosing the right resilience level is a direct, explicit trade-off between result fidelity and both cost and turnaround time on real hardware.

Q4.3

Why must you explicitly bind concrete parameter values to a parameterized circuit before submitting it as part of a PUB to a primitive, rather than passing the parameterized circuit directly?

While `QuantumCircuit` objects can hold symbolic, unbound `Parameter` placeholders (Section 2) for convenient, reusable circuit construction, a primitive job ultimately must execute one (or more, per PUB) fully **concrete** circuit(s) with definite numerical gate parameters – there is no such thing as "running" an unbound symbolic circuit directly. A PUB's parameter-values component is exactly what supplies these concrete numbers at submission time: `(circuit, observable, parameter_values)` for `Estimator`, or `(circuit, parameter_values)` for `Sampler` – and, importantly, a single PUB can supply an entire *array* of different parameter-value sets in one call, letting one submitted job efficiently sweep over many parameter combinations (e.g. every iteration's trial point during a VQE optimization loop) without needing a separate job submission per parameter set.

```
import numpy as np
from qiskit.primitives import StatevectorEstimator as Estimator

param_values = np.linspace(0, np.pi, 20)          # sweep 20 angle values
job = Estimator().run([(qc, observable, param_values)])
```

Q4.4

What is the difference between the plain `qiskit.primitives` reference implementations (e.g. `StatevectorSampler`) and the `qiskit_ibm_runtime` primitives, given that both expose the identical V2 interface?

The `qiskit.primitives` module's reference implementations (`StatevectorSampler`, `StatevectorEstimator`) compute results via **exact local state-vector simulation** – fast for small circuits, noise-free, and requiring no network access or IBM Quantum account, ideal for local algorithm development, unit testing, and debugging. `qiskit_ibm_runtime`'s `SamplerV2`/`EstimatorV2` instead submit the job to **actual IBM Quantum hardware or cloud-hosted simulators**, subject to real device noise (or, for cloud simulators, configurable noise models), queue wait times, and the resilience-level error-mitigation options discussed above. Because both expose the exact same primitive interface, well-written algorithm code can typically swap between the two with minimal changes – developing and validating against the fast local reference implementation before submitting the identical logic to real hardware via the runtime primitives.

Running on Hardware & Qiskit Runtime

Q5.1

What is `QiskitRuntimeService`, and what is the typical first step for connecting to and selecting an actual IBM Quantum backend?

`QiskitRuntimeService` is the main client class for authenticating with IBM Quantum's cloud platform, discovering available backends (real devices and cloud-hosted simulators), and submitting primitive jobs to them. A typical workflow instantiates the service (using saved or explicitly provided account credentials), then either selects a specific named backend or uses `least_busy()` to automatically pick whichever currently-operational device (matching any specified constraints, like a minimum qubit count) has the shortest queue – a common, practical default for routine development work where any sufficiently large, healthy device will do.

```
from qiskit_ibm_runtime import QiskitRuntimeService

service = QiskitRuntimeService()   # uses saved account credentials
backend = service.least_busy(operational=True, min_num_qubits=27)
```

Q5.2

What is the difference between running primitive jobs in plain `job mode`, inside a `Session`, and inside a `Batch`?

Job mode (the default when no session/batch context is active) submits each primitive call as an entirely independent job, queued and scheduled on its own – simplest, but offers no guarantee of low latency *between* related calls, since other users' jobs can be interleaved in the queue between yours. A **Session** reserves a period of dedicated, low-latency access to a backend across multiple sequential primitive calls – valuable for *iterative* algorithms (like VQE) where each iteration's next circuit depends on the previous iteration's measured result, and queueing delays between iterations would otherwise dominate total runtime. A **Batch** instead groups multiple, mutually *independent* jobs together for efficient parallel/bulk submission and scheduling (e.g. running many independent circuits from a parameter sweep) without needing the strict sequential low-latency guarantee a Session provides – choosing the right mode meaningfully affects both cost and turnaround time for a given workload's actual access pattern.

Q5.3

What is a `Qiskit Function`, and how does it extend the runtime execution model beyond simply running individual circuits through a primitive?

A Qiskit Function is a higher-level, packaged service (offered by IBM or third-party providers through the Qiskit Functions catalog) that accepts a higher-level problem description as input – e.g. "find the ground-state energy of this molecule" or "solve this optimization problem" – and internally handles the entire pipeline (circuit construction, transpilation, primitive calls, classical pre/post-processing, and often proprietary error-mitigation or algorithm-improvement techniques) needed to produce a result, without the calling user needing to manually implement or manage every one of those steps themselves. This reflects a broader shift in the Qiskit ecosystem toward **quantum-centric supercomputing** workflows – treating quantum execution as one specialized, callable component orchestrated within a larger, more automated classical-quantum pipeline, rather than requiring every user to hand-build the full circuit-to-result pipeline from scratch for every new problem.

Q5.4

What is a "fake backend" (e.g. `FakeManilaV2`) in Qiskit's testing ecosystem, and why is it valuable for local development before ever submitting a job to real, queued hardware?

A fake backend is a locally-runnable simulator configured with the *exact* `Target`, coupling map, and (optionally) a realistic noise model snapshot captured from an actual, real IBM Quantum device at a specific point in time – letting you transpile and run circuits against a faithful, hardware-realistic simulation entirely locally, with no network access, no queue wait, and no cost. This is valuable for developing and validating an entire algorithm pipeline end-to-end (including transpilation targeting the real device's specific basis gates and connectivity, and observing plausible, realistic noise-degraded results) before ever submitting to the real, queued device – catching transpilation errors, layout issues, or basic algorithmic bugs cheaply and instantly, reserving actual hardware time/queue slots for genuinely final validation runs.

Qiskit Aer & Simulation

Q6.1

What is `qiskit-aer`, and why was it split out of the core `qiskit` package into its own separately-installed, separately-versioned project?

`qiskit-aer` is Qiskit's high-performance simulation package, providing `AerSimulator` and related backends implemented in optimized C++ for substantially better simulation performance than the pure-Python reference implementations bundled in core Qiskit. It was split into its own package (installed via `pip install qiskit-aer`) as part of the broader Qiskit 1.0 restructuring specifically so it could be developed, versioned, and released independently of the core SDK's own release cadence – letting simulator-specific improvements (new simulation methods, GPU support, performance optimizations) ship on their own timeline without being coupled to, or blocked by, core Qiskit's separate release schedule, and vice versa.

Q6.2

What is the difference between `AerSimulator`'s `statevector`, `density_matrix`, and `matrix_product_state` simulation methods, and what specific trade-off does each represent?

Statevector simulation tracks a pure quantum state's full 2^n complex amplitude vector directly – exact and conceptually simplest, but memory usage grows exponentially with qubit count, typically capping practical simulation around 30-ish qubits on a single machine. **Density-matrix** simulation tracks the full $2^n \times 2^n$ density matrix instead, needed specifically to correctly simulate mixed states arising from noise models – but its memory cost is the *square* of the statevector method's, roughly halving the practically simulable qubit count for a given amount of memory. **Matrix product state (MPS)** simulation instead represents the state in a compressed tensor-network form that can, for circuits with **limited entanglement** between distant qubits, simulate substantially *more* qubits than statevector methods allow – at the cost of an approximation whose accuracy and efficiency both degrade for circuits generating large amounts of long-range entanglement, where the compressed MPS representation stops being an efficient approximation at all.

Q6.3

How do you attach a realistic `NoiseModel` to `AerSimulator`, and why is simulating with noise essential before assuming an algorithm will actually work on real, imperfect hardware?

A `NoiseModel` specifies error probabilities/channels (depolarizing error, thermal relaxation, readout error) to apply to specific gates and/or qubits during simulation – either built manually, or, much more commonly in practice, extracted directly from a real backend's live calibration data via `NoiseModel.from_backend(real_backend)`, producing a simulation whose noise characteristics closely mirror an actual device at the moment the calibration snapshot was taken. This matters because an algorithm that works perfectly on a noise-free statevector simulation can behave very differently – or fail outright – once realistic gate errors, decoherence, and readout errors are introduced, since many algorithms' accuracy degrades with circuit depth in ways only visible under a realistic noise model; testing against a noise-modeled Aer simulation is a standard, essential intermediate validation step between pure ideal-simulator development and actual hardware submission.

```
from qiskit_aer import AerSimulator
from qiskit_aer.noise import NoiseModel

noise_model = NoiseModel.from_backend(real_backend)
noisy_sim = AerSimulator(noise_model=noise_model)
```

Q6.4

What is the difference between running a circuit through `AerSimulator` for a full `shot-based` sampling simulation versus computing an exact `Statevector` directly via `qiskit.quantum_info`?

`AerSimulator` (used via the `run()` method, or wrapped inside a local primitive) simulates the circuit by executing it repeatedly for a specified number of **shots**, producing measurement-outcome statistics that mimic what a real quantum computer's repeated runs would produce – appropriate whenever the goal is to realistically emulate the actual sampling process, including any attached noise model. `quantum_info.Statevector.from_instruction(circuit)` instead directly and exactly computes the circuit's final quantum state vector via linear algebra, with **no** sampling/shot noise at all – appropriate for algorithm development, debugging, and verification purposes where you want to inspect the mathematically exact resulting state (amplitudes, exact probabilities) rather than emulate the statistical sampling process a real measurement or shot-based simulation would introduce.

Quantum Info & Operators

Q7.1

What is `SparsePauliOp`, and why has it become the standard way to represent observables/Hamiltonians in current Qiskit, particularly as input to `EstimatorV2`?

`SparsePauliOp` represents an operator as a weighted sum of Pauli strings (e.g. $0.5\,IIZZ - 0.3\,XXII$), storing only the non-zero Pauli terms rather than a full, dense $2^n \times 2^n$ matrix – since most physically relevant Hamiltonians (molecular electronic structure Hamiltonians, Ising-model cost functions for QAOA) are naturally expressed as a modest-sized sum of Pauli terms, this sparse representation is both far more memory-efficient and the natural, direct input format `EstimatorV2` expects for its observable argument. It also supports natural, efficient arithmetic (addition, scalar multiplication, tensor products) directly on the Pauli-sum representation, without ever needing to materialize a full dense matrix at all for typical use.

```
from qiskit.quantum_info import SparsePauliOp

H = SparsePauliOp(["ZZ", "XI", "IX"], coeffs=[1.0, 0.5, 0.5])
```

Q7.2

What is the `qiskit.quantum_info` module, and how does its combination with the primitives interface fully supersede the deprecated `qiskit.opflow` module?

`quantum_info` provides foundational data structures and algebra for quantum states and operators – `Statevector`, `DensityMatrix`, `Operator`, `SparsePauliOp`, and utilities for computing fidelities, partial traces, and other standard quantum-information quantities – all built on plain, direct linear-algebra representations rather than `opflow`'s older, more abstract, and comparatively awkward expression-tree-based operator algebra. Combined with the primitives interface (which directly accepts `quantum_info` objects like `SparsePauliOp` as observables), this combination provides everything `opflow` used to offer – but expressed through simpler, more directly-inspectable data structures – which is precisely why `opflow` was deprecated and eventually removed: its functionality was fully absorbed and re-expressed through `quantum_info` plus primitives, without needing opflow's separate, additional abstraction layer at all.

Q7.3

What is the `Operator` class, and how does it differ from `SparsePauliOp` in terms of what it represents and its typical use cases?

`Operator` represents a general unitary (or arbitrary linear) operator as an explicit, **dense** matrix – directly usable for exact matrix-level manipulation, composition, and comparison of arbitrary operators (verifying two different circuit constructions implement the identical unitary, for instance, via `Operator(qc1) == Operator(qc2)`). `SparsePauliOp`, in contrast, specifically represents an operator as a sparse sum of Pauli terms, suited to representing Hamiltonians/observables to be measured rather than general arbitrary unitaries applied within a circuit. In practice, `Operator` is most useful for small-to-moderate circuits (dense matrices again suffer the same exponential memory blowup as statevector simulation) during verification, debugging, and unit-testing that a circuit implements its intended unitary transformation exactly, while `SparsePauliOp` is the standard choice specifically for the observable side of expectation-value calculations.

Q7.4

How do you compute the fidelity between two quantum states using `quantum_info`, and why is state fidelity a standard way to quantify how close an experimentally-obtained (possibly

noisy) state is to some ideal target state?

`quantum_info.state_fidelity(state1, state2)` computes the standard fidelity measure between two quantum states (pure or mixed, given as `Statevector` or `DensityMatrix` objects) – a value of 1 indicating the two states are identical, and 0 indicating they are perfectly distinguishable/orthogonal. This is the standard tool for quantifying, in a single interpretable number, how successfully a real (noisy) experimental run, or a noisy simulation, actually prepared some intended target state – e.g. verifying that a hardware-executed Bell-state preparation circuit produced a final state with high fidelity relative to the mathematically ideal Bell state, providing a concrete, quantitative benchmark of an implementation's or a device's actual state-preparation quality.

```
from qiskit.quantum_info import Statevector, state_fidelity

ideal = Statevector.from_instruction(qc)
fidelity = state_fidelity(ideal, experimentally_reconstructed_state)
```

Advanced Features

Q8.1

What are "dynamic circuits," and what specific new capability (beyond simple, pre-planned classical conditions) do mid-circuit measurement combined with real-time classical control flow unlock?

Dynamic circuits interleave quantum operations with classical computation and control flow *during* circuit execution – most importantly, allowing a **mid-circuit measurement**'s result to be read out and used, in real time within the same circuit execution, to decide which quantum operation happens next (via `if_test`, `while_loop`, or general classical-expression-based conditions), all before the circuit fully completes and without needing a separate, later circuit submission. This directly enables important techniques impossible with a purely "measure only at the very end" circuit model – most notably, **quantum error correction** itself (which fundamentally requires repeated mid-circuit syndrome measurements that condition subsequent real-time correction operations), as well as measurement-based quantum computation and certain circuit-depth-reducing techniques that substitute mid-circuit measurement and classical feedforward for what would otherwise require additional entangling gates.

Q8.2

What is a custom `TranspilerPass`, and when would a Qiskit developer need to write one rather than simply relying on the built-in preset pass managers?

A `TranspilerPass` is a single, focused transformation or analysis step applied to a circuit's internal DAG (directed acyclic graph) representation during transpilation – Qiskit's preset pass managers (Section 3) are themselves just pre-configured pipelines chaining together many such built-in passes. A developer needs to write a *custom* pass specifically when a project requires transpilation logic the built-in passes don't already provide – e.g. a specialized gate-decomposition rule specific to a particular experimental or novel hardware platform, a custom noise-aware qubit-layout heuristic tailored to a specific device's unusual error characteristics, or a project-specific circuit optimization rule exploiting domain knowledge the general-purpose preset passes have no way to know about – and can then insert that custom pass into a `PassManager` alongside (or in place of) Qiskit's standard built-in passes.

Q8.3

What visualization tools does Qiskit provide for inspecting circuits and results, and what does each (`draw()`, `plot_histogram()`, `plot_state_city()`) specifically show?

`QuantumCircuit.draw()` renders the circuit itself as a diagram (text, matplotlib, or LaTeX output) – essential for visually verifying a circuit's structure matches what was intended, especially after transpilation has potentially rearranged and relabeled qubits. `qiskit.visualization.plot_histogram()` plots measurement-outcome counts/probabilities as a bar chart – the standard way to visually inspect a `Sampler` result's output distribution. `plot_state_city()` (and related state-visualization functions like `plot_bloch_multivector()`) visualize a quantum state's amplitudes or single-qubit Bloch-sphere representations directly – useful during development and debugging for building direct visual intuition about a small circuit's actual resulting quantum state, in a way that raw printed numbers alone don't as easily convey.

```
qc.draw("mpl")                          # circuit diagram
from qiskit.visualization import plot_histogram
plot_histogram(sampler_result[0].data.meas.get_counts())
```

Q8.4

What is meant by ”quantum-centric supercomputing,” and how does it frame Qiskit’s ongoing development direction (Qiskit Patterns, Qiskit Functions, tighter classical-HPC integration) as distinct from viewing a quantum computer as a simple, standalone accelerator?

Quantum-centric supercomputing frames a quantum processor not as an isolated, standalone device you submit one job to and wait for a result from, but as one specialized **component** tightly integrated within a larger, heterogeneous classical high-performance-computing (HPC) workflow – with classical CPUs/GPUs handling pre-processing, transpilation, error mitigation post-processing, and classical optimization loops, while the quantum processor is invoked repeatedly, efficiently, and with low latency as needed within that larger pipeline. **Qiskit Patterns** (a structured four-step workflow: map a problem to circuits/operators, optimize/transpile for the target hardware, execute via primitives, and classically post-process results) and **Qiskit Functions** (Section 5) are both concrete expressions of this philosophy – reflecting Qiskit’s current development direction as building the software infrastructure specifically for this tightly-coupled, quantum-plus-classical-HPC execution model, rather than treating quantum hardware as a simple, isolated, occasionally-useful standalone accelerator called only occasionally and independently of any larger classical computation.

End of deck. Qiskit evolves quickly – always cross-check against the current release notes at quantum.cloud.ibm.com/docs.