Django Backend Developer — Interview Q&A Prepared: October 2, 2025

1. CORE PYTHON ---------------- Q: What are lists, tuples, dicts, and their differences? A: Lists are ordered, mutable sequences defined by []. Tuples are ordered, immutable sequences using (). Dicts are unordered (insertion-ordered since Python 3.7), mutable mappings with key:value pairs using {}. Use tuples for fixed collections, lists for mutable ordered collections, dicts for lookups by key.

Q: Explain decorators and context managers. A: Decorators are callables that take a function/class and return a new function/class (often used with @). They add behavior (e.g., logging, auth). Context managers implement __enter__ and __exit__ or use @contextmanager from contextlib; they manage setup/teardown for resources (e.g., files, DB transactions) with the "with" statement.

Q: Difference between @staticmethod, @classmethod, and instance methods? A: Instance methods rece self (instance). @classmethod receives cls (class) and can access class-level data and constructors. @staticmethod receives no automatic first arg — it's namespaced in the class but behaves like a plain function.

Q: What are Python's data types and their mutability? A: Common types: int, float, str, bool (immutable); tuple (immutable); list, dict, set (mutable). Bytes are immutable; bytearray mutable. Immutability affects whether objects can be changed in-place.

Q: Difference between shallow and deep copy? A: Shallow copy (copy.copy) copies the container but keeps references to nested objects. Deep copy (copy.deepcopy) recursively copies nested objects, producing independent structures.

Q: How do you handle exceptions? A: Use try/except blocks, optionally finally or with context managers. Catch specific exceptions, not a bare except. Raise exceptions with informative messages and define custom exceptions for domain errors. Clean up resources in finally or via context managers.

2. DJANGO FRAMEWORK -------------------- Q: MVT Architecture? A: Model — data layer (ORM models). View — receives request, returns response (business logic). Template — presentation layer for HTML. URLs route requests to views. Templates are separate from views.

Q: ORM — How Django interacts with databases? A: Models define schema; Django translates ORM call to SQL. Manage with migrations. QuerySets are lazy; you chain filters and execute when evaluated.

Example: Query all users with age > 18: User.objects.filter(age__gt=18)

Q: Models & Migrations? A: Define models in models.py, run makemigrations to create migration files, migrate to apply them. To modify: change model, makemigrations, migrate; consider data migrations and add nullable fields or defaults when altering existing tables.

Q: Views — function-based vs class-based? A: Function-based views (FBV) are simple functions handling request -> response. Class-based views (CBV) provide reusable behavior via methods (get/post) and mixins, enabling faster composition for common patterns (ListView, DetailView, etc.). CBVs can be more abstract but DRYer.

Q: URLs & Routing? A: Define urlpatterns in urls.py and include app routes. Use path() and re_path(); name routes for reverse() usage.

Q: Templates? A: Use template inheritance (base.html -> child blocks). Use filters (|date, |default) and pass context dicts from views. Avoid heavy logic in templates.

Q: Forms — Form vs ModelForm? A: Form is for arbitrary fields and validation. ModelForm auto-generates fields from a model and handles saving to model instances.

Q: Authentication? A: Django provides authentication system (User model, login, logout, sessions). For custom user: extend AbstractUser or AbstractBaseUser. Use django.contrib.auth views or DRF auth for APIs. Protect routes with @login_required or permission classes.

Q: Middlewares? A: Middleware are hooks between request and response processing. They can modify request, response, handle exceptions, or manage headers. Order matters.

Q: Signals? A: Signals notify when certain actions occur. Example: post_save to create a user profile after a User is created; pre_delete to clean up external resources before record deletion. Use sparingly — explicit calls are often clearer.

3. REST API / DJANGO REST FRAMEWORK (DRF) -- Q: What are serializers? A: They convert model instances to primitive datatypes (for JSON) and validate/deserialize input into model instances or Python objects.

Q: ModelSerializer vs Serializer? A: ModelSerializer auto-generates fields from a model and provides create/update helpers. Serializer is manual and gives full control over fields and validation.

Q: ViewSets and Routers? A: ViewSets combine related view logic (list, retrieve, create, update, destroy) into a class. Routers auto-generate URL patterns for ViewSets.

Q: Implement authentication (JWT, token, session)? A: Session auth uses Django sessions (good for browser). Token auth uses tokens per user (DRF TokenAuthentication). JWT (JSON Web Tokens) provid stateless tokens (e.g., using djangorestframework-simplejwt). Choose based on clients and security needs.

Q: Pagination and permissions? A: DRF provides pagination classes (PageNumber, LimitOffset, Cursor). Permissions: IsAuthenticated, IsAdminUser, custom permission classes. Use throttling and scopes when needed.

4. DATABASE / SQL ------------------- Q: Writing queries in Django ORM? A: Use model managers and QuerySet methods: MyModel.objects.filter(field=value).exclude(...).annotate(...).order_by(...)

Q: Optimize queries? A: Use select_related for single-valued foreign keys (JOINs) and prefetch_related for reverse or many-many relations. Use values()/values_list for lightweight fetches. Use QuerySet.defer()/only() to limit fields. Monitor with Django debug toolbar and database EXPLAIN.

Q: Field relationships? A: OneToOneField: one-to-one relation (unique FK). ForeignKey: many-to-one. ManyToManyField: many-to-many relation.

5. DEPLOYMENT & ENVIRONMENT ---------------------------- Q: .env files? A: Store environment-specific secrets/config (SECRET_KEY, DB credentials, DEBUG flag). Load with python-dotenv or django-environ; never commit to VCS.

Q: Deploy Django to production? A: Typical stack: Gunicorn (WSGI server) + Nginx (reverse proxy, static files) + PostgreSQL. Platforms: Heroku, AWS (Elastic Beanstalk, ECS), DigitalOcean, Railway. Use containers (Docker) for reproducibility.

Q: DEBUG=True vs False? A: DEBUG=True enables helpful error pages and auto-reload but leaks sensitive info — never use in production. DEBUG=False disables debug pages; set ALLOWED_HOSTS properly.

Q: Static and media files? A: Static files served via collectstatic to a CDN or static server (nginx, S3). Media files (user uploads) should be on durable storage (S3, GCS) with proper access controls.

6. VERSION CONTROL / GIT -------------------------- Q: Merge conflicts? A: Resolve by reading changes, choosing correct edits, running tests, and making a clear commit message. Prefer feature branches and small PRs to reduce conflicts.

Q: Typical workflow? A: Git flow or trunk-based flow: create feature branch, commit frequently with meaningful messages, open PR, request review, run CI, squash/rebase if needed, merge to main.

7. SYSTEM DESIGN / ARCHITECTURE --------------------------------- Q: Design a simple blog API (high-level)? A: Entities: User, Post, Comment, Tag. Endpoints: /posts/, /posts/{id}/, /users/{id}/posts/, /comments/. Use pagination, filtering, authentication, rate limiting. Store media in object storage; use caching (Redis) for hot reads; add search with Elasticsearch or Postgres full-text.

Q: How to scale a Django app? A: Horizontal scaling (multiple gunicorn workers on multiple instances behind a load balancer), use caching layers (Redis), database read-replicas, async task queue

(Celery + Redis/RabbitMQ), optimize DB queries, use CDN for static assets, and monitor with metrics.

Q: What’s caching and how to implement it? A: Caching stores precomputed responses or query results. Implement per-view caching, template fragment caching, low-level cache APIs, and Redis/memcached as backend. Invalidate caches prudently.

8. PRACTICAL / LIVE CODING --------------------------- Examples: - Reverse a string: s[::-1] or loop. - Find duplicates in list: use set or collections.Counter. - Mini task: implement registration endpoint: validate input, hash password (make_password), create user, return token or session.

Testing: write unit tests for views, serializers, models; use pytest-django or Django’s TestCase.

Final tips: - Explain trade-offs and alternatives. - Talk through your thought process. - Mention logging, monitoring, and testing practices. - Be ready with 2–3 real project examples and metrics (e.g., improved response time, reduced DB queries).