Собеседование стажёра — Python / LLM Engineering

Время: ~60 мин

- 5 мин — opener / команда
- 10 мин — опыт / проект
- 15 мин — coding
- 25 мин — Python fundamentals
- 3–5 мин — LLM
- 5 мин — вопросы кандидата

Главный критерий: **есть ли у человека нормальная базовая модель Python и способен ли он рассуждать, а не угадывать ответы.**

---

0. Opener — 5 мин

Привет! Меня зовут Илья, я работаю в команде центра данных и отчётности. Ищу стажёра в нашу команду.

Формально мы ближе к data engineering, но задачи отличаются от классической роли, где основной фокус — ETL и перекладывание данных из точки А в точку Б.

У нас три основных направления:

1. Внутренняя Python-библиотека для проверки качества данных. Есть Python, местами SQL и PySpark, но важнее умение писать нормальный инженерный код.

2. Внутренние инструменты и софт для автоматизации отчётности.
3. LLM-based решения для проверки качества данных и логики их сбора. Мы не обучаем свои модели, а строим инструменты и пайплайны вокруг готовых LLM.

В разработке активно используем coding agents, поэтому важно не помнить весь Python API наизусть, а понимать язык, уметь читать код и замечать, когда решение работает неправильно или неоправданно усложнено.

Сегодня сначала немного познакомимся, потом будет небольшая задача на Python и несколько вопросов по самому языку. Если чего-то не знаешь — ничего страшного, можно рассуждать вслух.

---

1. Опыт — 10 мин

Расскажи кратко о себе и самом интересном проекте

Уточнить по ситуации:

- Что конкретно делал ты?
- Где больше всего застрял?
- Как разобрался?
- Использовал LLM / coding agents?
- Было что-нибудь, что пришлось потом переписать?
- Почему?

+

- конкретно объясняет свой вклад;
- понимает собственный проект;
- спокойно говорит «не знаю»;

- может назвать ошибку или неудачное решение;
- понимает код, который писал с AI.

–

- только «мы сделали»;
- технологии вместо сути;
- не может объяснить собственный код;
- AI сделал всё, кандидат не понимает результат.

---

2. Coding — 15 мин

Без LLM.

```
checks = [
    {"name": "users", "status": "failed"},
    {"name": "orders", "status": "passed"},
    {"name": "prices", "status": "failed"},
    {"name": "users", "status": "failed"},
]
```

Задача

Вернуть **имена упавших проверок без дублей**, сохранив порядок:

```
["users", "prices"]
```

Нормальное решение:

```
def failed_checks(checks):
    result = []
    seen = set()
```

```
    for check in checks:
        if check["status"] == "failed" and check["name"]
            result.append(check["name"])
            seen.add(check["name"])

    return result
```

Не требовать именно его.

Если застрял

> Сначала просто верни все имена со `status == "failed"`.

Потом добавить дедупликацию.

Если легко

Почему `set`?
→ уникальность, быстрый membership.

Почему не `list(set(...))`?
→ не следует полагаться на сохранение исходного порядка.

Сложность?
→ `O(n)` average.

А если нужно количество падений?

```
{"users": 2, "prices": 1}
```

→ `dict` / `Counter` / `defaultdict`.

---

3. Python fundamentals — ~25

мин

Не проходить всё обязательно. Если кандидат отвечает легко — идти глубже. Если плавает — переходить дальше.

A. Collections

Чем `list` отличается от `set`?

- list — последовательность, дубли допустимы;
- set — уникальные элементы;
- membership в set обычно быстрее.

Когда `set`, а когда `list`?

→ уникальность / частые `in` vs порядок / последовательность / дубли.

`tuple` vs `list`?

→ tuple immutable, list mutable.

Что может быть ключом `dict`?

→ hashable object.

Если не знает термин:

> Строка? int? tuple? list?

Почему list нельзя использовать ключом?

→ mutable / не hashable.

Можно?

```
(1, 2, 3)
```

→ да.

```
(1, [2, 3])
```

→ нет, внутри unhashable list.

B. OOP в Python

Начинать совсем просто.

Что такое класс и объект?

Ожидание:

- класс — описание/тип;
- объект — конкретный instance класса.

Не требовать академической формулировки.

Что делает `__init__`?

→ инициализирует состояние нового instance после его создания.

Что такое `self`?

→ ссылка на конкретный instance, с которым вызывается метод.

Что выведет?

```
class User:
    def __init__(self, name):
        self.name = name


a = User("Alice")
b = User("Bob")

print(a.name)
print(b.name)
```

→ `Alice`, `Bob`.

Атрибут instance vs атрибут класса

```
class User:
    role = "user"

    def __init__(self, name):
        self.name = name
```

Что общее для всех объектов, а что своё?

→ `role` — class attribute, `name` — instance attribute.

Что такое наследование?

Минимально:

> один класс может наследовать поведение/атрибуты другого.

```
class Animal:
    def speak(self):
        return "?"


class Dog(Animal):
    def speak(self):
        return "woof"
```

Что вернёт:

```
Dog().speak()
```

→ `"woof"`.

Что делает `super()`?

→ позволяет обратиться к реализации родительского класса, часто к `__init__`.

Не знать детали MRO для стажёра нормально.

Magic methods — если база уверенная

Как сделать, чтобы работало:

```
users = Users(...)
len(users)
```

Ожидание:

```
def __len__(self):
    ...
```

Важно: именно `len(obj)`, не `obj.len()`.

Как определить строковое представление?

```
print(user)
```

→ `__str__`.

Дополнительно:

`__repr__`?
→ developer/debug representation; точное различие знать необязательно.

Как сделать объект итерируемым?

Чтобы работало:

```
for user in users:
    ...
```

→ `__iter__`.

Если знает про iterator protocol / `__next__` — плюс.

Как сделать поддержку:

```
"user123" in users
```

→ `__contains__`.

Но можно сказать, что Python иногда умеет реализовать membership через iteration.

Как определить:

```
a == b
```

для своих объектов?

→ `__eq__`.

Продвинутый follow-up:

> А если хотим использовать экземпляры как ключи dict / элементы set?

→ нужен корректный hash/equality contract, `__hash__`.

Не требовать от стажёра деталей реализации.

C. Ссылки / mutability

Что выведет?

```
a = [1, 2]
b = a
b.append(3)

print(a)
```

→

```
[1, 2, 3]
```

Почему?
→ один объект, две ссылки.

Теперь?

```
a = [1, 2]
b = a.copy()
b.append(3)

print(a)
```

→

```
[1, 2]
```

Теперь?

```
a = [[1], [2]]
b = a.copy()

b[0].append(3)

print(a)
```

→

```
[[1, 3], [2]]
```

Почему?

→ shallow copy.

Если знает → спросить `deepcopy`.

D. Mutable default arguments

Что не так?

```
def add(x, items=[]):
    items.append(x)
    return items
```

Что вернут:

```
add(1)
add(2)
```

→

```
[1]
[1, 2]
```

Почему?
→ default object создаётся один раз.

Исправление:

```
def add(x, items=None):
    if items is None:
        items = []
```

E. Equality / identity

`==` vs `is`?

- `==` — равенство;
- `is` — один и тот же объект.

Как проверять `None`?

```
x is None
```

F. Functions

Что такое `*args` / `**kwargs`?

- `*args` — positional;
- `**kwargs` — keyword arguments.

Незнание не критично.

Что произойдёт?

```
def change(items):
    items.append(3)


a = [1, 2]
change(a)
```

```
print(a)
```

→ `[1, 2, 3]`.

Почему?
→ функция работает с тем же mutable object.

G. Exceptions

Что делает?

```
try:
    do_something()
except Exception:
    pass
```

Почему плохо?

- скрывает ошибки;
- затрудняет debugging;
- может проглотить неожиданный bug.

Что лучше?

→ ловить конкретные ожидаемые exception и осознанно обрабатывать.

H. Iterators / generators

Чем отличаются?

```
def process_all(items):
    return [process(x) for x in items]
```

и

```
def process_all(items):
    for x in items:
        yield process(x)
```

Ожидание:

- первый сразу строит list;
- второй generator / lazy;
- второй полезен для больших или потоковых данных.

I. Comprehensions

Что делает?

```
[x * 2 for x in numbers if x > 0]
```

Должен спокойно прочитать.

Можно попросить переписать обычным циклом.

J. Complexity

Не экзамен по Big-O.

```
for x in items:
    if x in seen:
        ...
```

Что быстрее для `seen`: list или set?

Обычно:

- list membership — `O(n)`;
- set — `O(1)` average.

---

4. Маленькие Python probes

Если осталось время / нужен дополнительный сигнал.

1

```
if flag == True:
```

Обычно проще:

```
if flag:
```

Не считать серьёзной ошибкой.

2

```
if len(items) == 0:
```

Обычно:

```
if not items:
```

3

```
for i in range(len(items)):
    print(items[i])
```

Если индекс не нужен:

```
for item in items:
```

4

Нужны индекс + элемент?

→ `enumerate`.

5

Идти одновременно по двум коллекциям?

→ `zip`.

Если не помнит название, но понимает идею — небольшой минус.

---

5. LLM — 3–5 мин

Только конкретные вопросы.

1

Нужно проверить, есть ли `NULL` в обязательной колонке.

LLM или SQL/Python?

→ **SQL/Python.** Детерминированная задача.

2

Просим:

```
{"score": 5}
```

Получаем:

```
Sure! Here is the result:

{"score": 5}
```

Что можно сделать?

Нормальные ответы:

- structured output / JSON mode;
- schema;
- parsing;
- validation;
- точнее задать формат.

3

LLM вернула валидный JSON:

```
{"score": "очень хорошо"}
```

Всё нормально?

→ Нет.

Нужно валидировать:

- структуру;
- типы;
- значения.

---

6. Вопросы кандидата — 5 мин

> Какие у тебя есть вопросы про команду или задачи?

Не переоценивать.

---

7. Быстрая оценка

После интервью по `1–5`.

Python familiarity

Насколько язык реально знаком?

Coding

Может ли самостоятельно написать небольшую функцию?

Mental model

Понимает ли:

- collections;
- mutability;
- references;
- hashability;
- functions;
- базовый OOP.

Reasoning

Может объяснить **почему**, а не просто назвать ответ?

Learning

Понимает собственные проекты, умеет разбираться и признавать незнание?

---

8. Интерпретация

Сильный

- спокойно пишет базовый Python;
- хорошо ориентируется в collections;

- понимает references/mutability;
- знает базовый OOP;
- уверенно читает незнакомый код;
- после уточнения способен углубляться;
- не обязан знать все magic methods, но понимает идею Python data model.

Средний

- база есть;
- некоторые темы дырявые;
- после подсказки быстро схватывает.

Для стажёра вполне может быть достаточно.

Слабый

- знает в основном синтаксис;
- путается в основных структурах данных;
- не понимает mutability/references;
- не способен написать маленькую функцию;
- угадывает ответы и не может объяснить почему.

Главное правило

Начинать с простого и углубляться только пока кандидат уверенно отвечает.

Лучше выяснить глубину понимания одной темы цепочкой конкретных вопросов:

`list → set → hashability → tuple → nested mutable object`

чем внезапно спрашивать стажёра:

> «Расскажи всё, что знаешь про внутреннее устройство словаря».