Complete Django & Python Interview Questions and Answers

Core Python

Difference between deep copy and shallow copy. When would you use each?

Shallow Copy:

- Creates a new object but references to nested objects are shared
- Only copies the top-level structure
- Changes to nested objects affect both copies

Deep Copy:

- Creates completely independent copies including all nested objects
- No shared references between original and copy
- More memory intensive and slower

Use Cases:

- **Shallow Copy:** When you need a new container but can share the contained objects (e.g., copying a list of immutable strings)
- **Deep Copy:** When you need complete independence (e.g., copying complex nested data structures that will be modified independently)

```python
import copy

original = [[1, 2, 3], [4, 5, 6]]
shallow = copy.copy(original)
deep = copy.deepcopy(original)

original[0][0] = 'X'
# shallow[0][0] is also 'X', but deep[0][0] remains 1
```

Explain Python's memory management — garbage collection, reference counting, cyclic references

Reference Counting:

- Python tracks how many references point to each object
- When count reaches zero, object is immediately deallocated
- Fast for most objects but can't handle circular references

Garbage Collection:

- Handles cyclic references that reference counting can't
- Uses generational garbage collection (3 generations)
- Younger objects are collected more frequently
- Can be manually triggered with `gc.collect()`

Cyclic References:

- Objects that reference each other in a cycle
- Handled by Python's cycle detector
- Uses mark-and-sweep algorithm to find unreachable cycles

```python
import gc

# Circular reference example
class Node:
    def __init__(self):
        self.ref = None

a = Node()
b = Node()
a.ref = b
b.ref = a  # Circular reference
```

How do you handle multithreading vs multiprocessing in Python?

Threading:

- Good for I/O-bound tasks
- Limited by GIL for CPU-bound tasks
- Shared memory space
- Lower overhead

Multiprocessing:

- Better for CPU-bound tasks
- Bypasses GIL limitations
- Separate memory spaces
- Higher overhead

AsyncIO:

- Best for I/O-bound and high-level structured network code
- Single-threaded concurrency
- Event loop based

```python
import threading
import multiprocessing
import asyncio

# Threading example
def worker():
    # I/O bound work
    pass

threads = [threading.Thread(target=worker) for _ in range(4)]

# Multiprocessing example
def cpu_bound_task(n):
    return sum(i*i for i in range(n))

with multiprocessing.Pool() as pool:
    results = pool.map(cpu_bound_task, [1000, 2000, 3000])

# AsyncIO example
async def async_task():
    await asyncio.sleep(1)
    return "Done"
```

What are Python data classes and how are they different from namedtuple and **slots**?

Data Classes (Python 3.7+):

- Automatically generates special methods
- Mutable by default
- Support type hints
- Can have methods and inheritance

Named Tuples:

- Immutable
- Memory efficient
- No type checking

- Limited functionality

slots:

- Memory optimization technique
- Restricts attribute creation
- Faster attribute access
- No **dict** created

```python
from dataclasses import dataclass
from collections import namedtuple

@dataclass
class Person:
    name: str
    age: int

    def greet(self):
        return f"Hello, I'm {self.name}"

# Named tuple
PersonTuple = namedtuple('Person', ['name', 'age'])

# Slots example
class PersonSlots:
    __slots__ = ['name', 'age']

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

Explain decorators and give real-world use cases

Decorators are functions that modify or extend the functionality of other functions without permanently modifying them.

Real-world use cases:

```python
```

```python
import time
import functools
from flask import request, jsonify

# Caching decorator
def memoize(func):
    cache = {}
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        key = str(args) + str(kwargs)
        if key not in cache:
            cache[key] = func(*args, **kwargs)
        return cache[key]
    return wrapper

# Timing decorator
def timer(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"{func.__name__} took {time.time() - start:.2f} seconds")
        return result
    return wrapper

# Authentication decorator
def require_auth(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        if not request.headers.get('Authorization'):
            return jsonify({'error': 'Authentication required'}), 401
        return func(*args, **kwargs)
    return wrapper
```

How do you implement custom iterators and generators?

Custom Iterator:

python

```python
class Fibonacci:
    def __init__(self, max_count):
        self.max_count = max_count
        self.count = 0
        self.a, self.b = 0, 1

    def __iter__(self):
        return self

    def __next__(self):
        if self.count >= self.max_count:
            raise StopIteration
        self.count += 1
        self.a, self.b = self.b, self.a + self.b
        return self.a
```

Generator:

```python
def fibonacci_gen(max_count):
    a, b = 0, 1
    count = 0
    while count < max_count:
        yield b
        a, b = b, a + b
        count += 1

# Generator expression
squares = (x**2 for x in range(10))
```

What are Python context managers? How does with statement work internally?

Context managers ensure proper resource management through `__enter__` and `__exit__` methods.

```python
```

```python
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()
        return False

# Using contextlib
from contextlib import contextmanager

@contextmanager
def database_connection():
    conn = create_connection()
    try:
        yield conn
    finally:
        conn.close()
```

Difference between @staticmethod, @classmethod, and instance methods

```python
```

```python
class MyClass:
    class_var = "I'm a class variable"

    def instance_method(self):
        """Has access to instance (self) and class (cls)"""
        return f"Instance method called on {self}"

    @classmethod
    def class_method(cls):
        """Has access to class (cls) but not instance"""
        return f"Class method called on {cls.__name__}"

    @staticmethod
    def static_method():
        """No access to self or cls - just a regular function"""
        return "Static method called"
```

Explain **new** vs **init**. When would you override **new**?

- `__new__` creates the instance (constructor)
- `__init__` initializes the instance

Override new for:

- Singletons
- Immutable objects
- Metaclass programming

```python
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self):
        if not hasattr(self, 'initialized'):
            self.initialized = True
```

How does Python implement method resolution order (MRO) in multiple inheritance?

Python uses C3 linearization algorithm for MRO:

```python
class A:
    def method(self):
        print("A")

class B(A):
    def method(self):
        print("B")
        super().method()

class C(A):
    def method(self):
        print("C")
        super().method()

class D(B, C):
    def method(self):
        print("D")
        super().method()

# MRO: D -> B -> C -> A -> object
print(D.__mro__)
```

Performance & Scalability

1) How to optimize Python code for speed and memory usage?

Speed Optimization:

- Use built-in functions and libraries (NumPy, Pandas)
- Avoid unnecessary loops
- Use list comprehensions over loops
- Profile your code with cProfile
- Consider Cython for critical sections

Memory Optimization:

- Use generators instead of lists for large datasets
- Use **slots** for classes with many instances
- Delete unused variables with `del`
- Use memory-efficient data structures

```python
```

```python
# Instead of this
def slow_function(items):
    result = []
    for item in items:
        if item > 0:
            result.append(item * 2)
    return result

# Use this
def fast_function(items):
    return [item * 2 for item in items if item > 0]
```

2) Difference between list, tuple, set, and dict in terms of performance

Operation	List	Tuple	Set	Dict
Access by index	O(1)	O(1)	N/A	O(1)
Search	O(n)	O(n)	O(1) avg	O(1) avg
Insert	O(1) amortized	Immutable	O(1) avg	O(1) avg
Delete	O(n)	Immutable	O(1) avg	O(1) avg
Memory	Medium	Low	Medium	High

3) When would you use NumPy / Pandas over pure Python data structures?

NumPy:

- Numerical computations
- Large arrays of homogeneous data
- Mathematical operations
- Broadcasting operations

Pandas:

- Data analysis and manipulation
- Working with structured/tabular data
- Time series analysis
- Data cleaning and transformation

```python
```

```python
import numpy as np
import pandas as pd

# NumPy is much faster for numerical operations
arr = np.array([1, 2, 3, 4, 5])
result = arr * 2 + 1  # Vectorized operation

# Pandas for data manipulation
df = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})
result = df.groupby('A').sum()
```

4) How does Python's hashing work for dictionaries and sets?

Python uses hash tables with open addressing:

```python
# Hash function considerations
hash("string")  # Strings are hashable
hash((1, 2, 3))  # Tuples are hashable if contents are
# hash([1, 2, 3])  # Lists are not hashable

class HashableClass:
    def __init__(self, value):
        self.value = value

    def __hash__(self):
        return hash(self.value)

    def __eq__(self, other):
        return isinstance(other, HashableClass) and self.value == other.value
```

5) Explain some Profiling tools in Python

```python
```

```
import cProfile
import timeit
from memory_profiler import profile

# cProfile for detailed profiling
cProfile.run('your_function()')

# timeit for small code snippets
time_taken = timeit.timeit('sum([1, 2, 3, 4, 5])', number=1000000)

# memory_profiler for memory usage
@profile
def memory_intensive_function():
    a = [1] * (10**6)
    b = [2] * (2 * 10**7)
    del b
    return a

# line_profiler for line-by-line profiling
# pip install line_profiler
# kernprof -l -v script.py
```

Advanced Topics

Explain metaclasses in Python. Have you used them in real projects?

Metaclasses are classes whose instances are classes themselves.

python

```python
class SingletonMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class DatabaseConnection(metaclass=SingletonMeta):
    def __init__(self):
        self.connection = "DB Connection"

# Real-world use: ORM field definitions
class FieldMeta(type):
    def __new__(mcs, name, bases, attrs):
        fields = {}
        for key, value in attrs.items():
            if isinstance(value, Field):
                fields[key] = value
                value.name = key
        attrs['_fields'] = fields
        return super().__new__(mcs, name, bases, attrs)

class Model(metaclass=FieldMeta):
    pass
```

How would you implement a custom caching mechanism in Python?

python

```python
import time
import threading
from functools import wraps
from collections import OrderedDict

class LRUCache:
    def __init__(self, maxsize=128):
        self.maxsize = maxsize
        self.cache = OrderedDict()
        self.lock = threading.Lock()

    def get(self, key):
        with self.lock:
            if key in self.cache:
                # Move to end (most recently used)
                value = self.cache.pop(key)
                self.cache[key] = value
                return value
            return None

    def put(self, key, value):
        with self.lock:
            if key in self.cache:
                self.cache.pop(key)
            elif len(self.cache) >= self.maxsize:
                # Remove least recently used
                self.cache.popitem(last=False)
            self.cache[key] = value

# TTL Cache
class TTLCache:
    def __init__(self, maxsize=128, ttl=300):
        self.maxsize = maxsize
        self.ttl = ttl
        self.cache = {}
        self.timestamps = {}

    def get(self, key):
        if key in self.cache:
            if time.time() - self.timestamps[key] < self.ttl:
                return self.cache[key]
            else:
                del self.cache[key]
                del self.timestamps[key]
        return None
```

```python
    def put(self, key, value):
        self.cache[key] = value
        self.timestamps[key] = time.time()
```

Difference between sync and async programming in Python (async/await)

```python
import asyncio
import aiohttp
import requests

# Synchronous - blocking
def fetch_url_sync(url):
    response = requests.get(url)
    return response.text

def main_sync():
    urls = ['http://example.com', 'http://google.com']
    for url in urls:
        content = fetch_url_sync(url)  # Blocks until complete
        print(f"Got {len(content)} chars from {url}")

# Asynchronous - non-blocking
async def fetch_url_async(session, url):
    async with session.get(url) as response:
        content = await response.text()
        return content

async def main_async():
    urls = ['http://example.com', 'http://google.com']
    async with aiohttp.ClientSession() as session:
        tasks = [fetch_url_async(session, url) for url in urls]
        results = await asyncio.gather(*tasks)
        for url, content in zip(urls, results):
            print(f"Got {len(content)} chars from {url}")

# asyncio.run(main_async())
```

How do you handle large files (GBs of data) efficiently in Python?

```python
```

```python
# Streaming file processing
def process_large_file(filename):
    with open(filename, 'r') as file:
        for line in file:  # Reads one line at a time
            process_line(line)

# Chunked processing
def process_file_in_chunks(filename, chunk_size=8192):
    with open(filename, 'rb') as file:
        while chunk := file.read(chunk_size):
            process_chunk(chunk)

# Memory mapping for random access
import mmap

def process_with_mmap(filename):
    with open(filename, 'r+b') as file:
        with mmap.mmap(file.fileno(), 0) as mmapped_file:
            # Access file like an array
            data = mmapped_file[0:1000]

# Using generators for CSV files
import csv

def read_large_csv(filename):
    with open(filename, 'r') as file:
        reader = csv.reader(file)
        for row in reader:
            yield row

# Pandas for large datasets
import pandas as pd

def process_large_csv_pandas(filename):
    chunk_size = 10000
    for chunk in pd.read_csv(filename, chunksize=chunk_size):
        process_chunk(chunk)
```

How to implement a singleton pattern in Python?

python

```python
# Method 1: Using __new__
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

# Method 2: Using decorator
def singleton(cls):
    instances = {}
    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]
    return get_instance

@singleton
class DatabaseConnection:
    def __init__(self):
        self.connection = "DB Connection"

# Method 3: Using metaclass
class SingletonMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]

class Logger(metaclass=SingletonMeta):
    def log(self, message):
        print(f"Log: {message}")
```

Django

Explain Django's MTV architecture. How is it different from MVC?

MTV (Model-Template-View):

- **Model**: Data layer (same as MVC Model)
- **Template**: Presentation layer (similar to MVC View)
- **View**: Business logic layer (similar to MVC Controller)

Key Differences:

- Django's View contains business logic (like Controller in MVC)
- Django's Template handles presentation (like View in MVC)
- Django handles URL routing separately

```python
# models.py (Model)
class Article(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()

# views.py (View - Business Logic)
def article_list(request):
    articles = Article.objects.all()
    return render(request, 'articles/list.html', {'articles': articles})

# templates/articles/list.html (Template - Presentation)
# {% for article in articles %}
#     <h2>{{ article.title }}</h2>
# {% endfor %}
```

How does Django handle an HTTP request internally?

1. **URL Resolution**: Django matches the URL against patterns in URLconf
2. **Middleware Processing**: Request passes through middleware layers
3. **View Execution**: Matched view function/class is called
4. **Template Rendering**: Template is rendered with context data
5. **Response Middleware**: Response passes through middleware
6. **HTTP Response**: Final response sent to client

```python
```

```python
# Django request-response flow
def my_view(request):
    # View processing
    context = {'data': 'Hello World'}
    return render(request, 'template.html', context)

# Middleware example
class CustomMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # Process request
        response = self.get_response(request)
        # Process response
        return response
```

Explain Django's request-response cycle in detail

1. **WSGI Handler**: Receives HTTP request
2. **URL Dispatcher**: Matches URL to view
3. **Middleware (Request)**: Process request through middleware stack
4. **View Processing**: Execute view function/class
5. **Model Interaction**: Database queries if needed
6. **Template Rendering**: Render template with context
7. **Middleware (Response)**: Process response through middleware
8. **HTTP Response**: Send response to client

What new features in recent Django versions have you used?

Django 4.2 (LTS - April 2023):

- Psycopg 3 support
- Comments on columns and tables
- Improved async support

Django 4.1:

- async-compatible interface for ORM
- `aform` and `afield` template tags
- Constraint validation improvements

Django 4.0:

- `zoneinfo` as default timezone implementation
- Functional unique constraints
- Redis cache backend improvements

```python
# Django 4.1 async ORM example
async def async_view(request):
    articles = [article async for article in Article.objects.all()]
    return JsonResponse({'count': len(articles)})

# Django 4.0 functional constraints
from django.db import models
from django.db.models import UniqueConstraint

class Article(models.Model):
    title = models.CharField(max_length=200)
    status = models.CharField(max_length=20)

    class Meta:
        constraints = [
            UniqueConstraint(
                fields=['title'],
                condition=models.Q(status='published'),
                name='unique_published_title'
            )
        ]
```

What are Django apps, and how do you structure a large project with multiple apps?

Django apps are reusable components that encapsulate related functionality.

Best Practices for Large Projects:

```
myproject/
├── config/
│   ├── settings/
│   │   ├── base.py
│   │   ├── development.py
│   │   └── production.py
│   ├── urls.py
│   └── wsgi.py
├── apps/
│   ├── users/
│   ├── articles/
│   ├── comments/
│   └── common/
├── static/
├── media/
├── templates/
└── requirements/
```

python

```python
# apps/users/models.py
class User(AbstractUser):
    email = models.EmailField(unique=True)

# apps/articles/models.py
class Article(models.Model):
    author = models.ForeignKey('users.User', on_delete=models.CASCADE)
    title = models.CharField(max_length=200)

# config/settings/base.py
DJANGO_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    # ...
]

LOCAL_APPS = [
    'apps.users',
    'apps.articles',
    'apps.comments',
]

INSTALLED_APPS = DJANGO_APPS + LOCAL_APPS
```

What is the role of manage.py and settings.py?

manage.py:

- Command-line utility for Django projects
- Wrapper around django-admin
- Sets DJANGO_SETTINGS_MODULE environment variable

settings.py:

- Central configuration file
- Contains all project settings
- Database, middleware, installed apps configuration

```python
# Custom management command
# management/commands/custom_command.py
from django.core.management.base import BaseCommand

class Command(BaseCommand):
    help = 'Custom command description'

    def handle(self, *args, **options):
        self.stdout.write('Command executed successfully!')

# Multiple settings files
# settings/base.py
DEBUG = False
DATABASES = {...}

# settings/development.py
from .base import *
DEBUG = True
```

What are Django signals and when would you use them?

Signals allow decoupled applications to get notified when actions occur elsewhere in the framework.

```python
```

```python
from django.db.models.signals import post_save, pre_delete
from django.dispatch import receiver
from django.contrib.auth.models import User

# Create user profile when user is created
@receiver(post_save, sender=User)
def create_user_profile(sender, instance, created, **kwargs):
    if created:
        UserProfile.objects.create(user=instance)

# Log when objects are deleted
@receiver(pre_delete, sender=Article)
def log_article_deletion(sender, instance, **kwargs):
    logger.info(f"Article '{instance.title}' is being deleted")

# Custom signals
from django.dispatch import Signal

# Define custom signal
payment_completed = Signal()

# Send signal
payment_completed.send(sender=self.__class__, user=user, amount=amount)

# Connect to custom signal
@receiver(payment_completed)
def handle_payment_completion(sender, user, amount, **kwargs):
    # Send confirmation email
    send_payment_confirmation(user, amount)
```

How do middlewares work in Django? Can you write a custom middleware?

Middleware is a framework of hooks into Django's request/response processing.

python

```python
# Custom middleware
class CustomMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        # Code executed for each request before view
        start_time = time.time()

        response = self.get_response(request)

        # Code executed for each request/response after view
        duration = time.time() - start_time
        response['X-Request-Duration'] = str(duration)

        return response

    def process_view(self, request, view_func, view_args, view_kwargs):
        # Called just before Django calls the view
        pass

    def process_exception(self, request, exception):
        # Called when a view raises an exception
        pass

# Authentication middleware
class TokenAuthMiddleware:
    def __init__(self, get_response):
        self.get_response = get_response

    def __call__(self, request):
        auth_header = request.META.get('HTTP_AUTHORIZATION')
        if auth_header and auth_header.startswith('Bearer '):
            token = auth_header.split(' ')[1]
            try:
                user = User.objects.get(auth_token=token)
                request.user = user
            except User.DoesNotExist:
                pass

        return self.get_response(request)
```

What's the difference between select_related() and prefetch_related()?

Both optimize database queries but work differently:

select_related():

- Uses SQL JOIN
- For ForeignKey and OneToOneField
- Single database query

prefetch_related():

- Separate queries + Python joins
- For ManyToManyField and reverse ForeignKey
- Multiple database queries but more flexible

```python
# select_related() - SQL JOIN
# SELECT * FROM articles JOIN users ON articles.author_id = users.id
articles = Article.objects.select_related('author').all()
for article in articles:
    print(article.author.name)  # No additional query

# prefetch_related() - Separate queries
# Query 1: SELECT * FROM articles
# Query 2: SELECT * FROM users WHERE id IN (1, 2, 3, ...)
articles = Article.objects.prefetch_related('tags').all()
for article in articles:
    for tag in article.tags.all():  # No additional queries
        print(tag.name)

# Combining both
articles = Article.objects.select_related('author').prefetch_related('tags')

# Custom prefetch
from django.db.models import Prefetch

articles = Article.objects.prefetch_related(
    Prefetch('comments', queryset=Comment.objects.filter(approved=True))
)
```

Difference between function-based views and class-based views

Function-Based Views (FBVs):

- Simple and explicit
- Good for simple views
- Less reusable

Class-Based Views (CBVs):

- More reusable and extensible
- Built-in functionality
- Can be more complex

```python
# Function-based view
def article_list(request):
    articles = Article.objects.all()
    return render(request, 'articles/list.html', {'articles': articles})

def article_create(request):
    if request.method == 'POST':
        form = ArticleForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('article_list')
    else:
        form = ArticleForm()
    return render(request, 'articles/create.html', {'form': form})

# Class-based view
from django.views.generic import ListView, CreateView

class ArticleListView(ListView):
    model = Article
    template_name = 'articles/list.html'
    context_object_name = 'articles'
    paginate_by = 10

class ArticleCreateView(CreateView):
    model = Article
    form_class = ArticleForm
    template_name = 'articles/create.html'
    success_url = '/articles/'
```

Explain mixins in class-based views

Mixins provide reusable functionality that can be combined with CBVs.

```python
```

```python
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import ListView, CreateView

# Custom mixins
class AjaxResponseMixin:
    def dispatch(self, request, *args, **kwargs):
        if not request.headers.get('x-requested-with') == 'XMLHttpRequest':
            return HttpResponseBadRequest('AJAX required')
        return super().dispatch(request, *args, **kwargs)

class AuthorRequiredMixin:
    def get_queryset(self):
        return super().get_queryset().filter(author=self.request.user)

# Using mixins
class ArticleListView(LoginRequiredMixin, AuthorRequiredMixin, ListView):
    model = Article
    template_name = 'articles/list.html'

class AjaxArticleCreateView(LoginRequiredMixin, AjaxResponseMixin, CreateView):
    model = Article
    fields = ['title', 'content']

    def form_valid(self, form):
        form.instance.author = self.request.user
        return super().form_valid(form)
```

How do you implement pagination in Django?

```python
```

```python
# Views
from django.core.paginator import Paginator
from django.shortcuts import render

def article_list(request):
    articles = Article.objects.all()
    paginator = Paginator(articles, 10)  # 10 articles per page

    page_number = request.GET.get('page')
    page_obj = paginator.get_page(page_number)

    return render(request, 'articles/list.html', {'page_obj': page_obj})

# Class-based view with pagination
class ArticleListView(ListView):
    model = Article
    template_name = 'articles/list.html'
    paginate_by = 10
    ordering = ['-created_at']

# Template (list.html)
# {% for article in page_obj %}
#     <h2>{{ article.title }}</h2>
# {% endfor %}
#
# <div class="pagination">
#     {% if page_obj.has_previous %}
#         <a href="?page=1">&laquo; first</a>
#         <a href="?page={{ page_obj.previous_page_number }}">previous</a>
#     {% endif %}
#
#     Page {{ page_obj
```