ДЖЕЙД КАРТЕР

Python Библиотеки

PySpark
Dash
Cython
Numba
Airflow
Asyncio
MoviePy
Seaborn
PyOpenGL
TensorFlow
Scikit-learn
Apache Kafka

Часть 2

Практическое применение

Джейд Картер
Библиотеки Python Часть 2. Практическое применение

Слово от автора

Дорогие читатели!

Python – это не просто язык программирования, это универсальный инструмент, который помогает нам решать самые разные задачи, от обработки данных до создания искусственного интеллекта. Во второй части книги я постарался показать, как эти инструменты можно применять в реальных проектах, делая вашу работу не только более эффективной, но и увлекательной.

Каждая глава этой части – это шаг в сторону практики, где мы вместе преодолеваем границы теории и углубляемся в реальные примеры и кейсы. Мне важно было продемонстрировать, что с помощью Python можно не только писать код, но и находить решения там, где это казалось невозможным.

Эта книга – результат моего опыта, наблюдений и экспериментов. Я надеюсь, что она станет для вас не просто руководством, а вдохновением, мотивирующим к изучению новых возможностей. Помните, что любое знание становится ценным, когда его можно применить на практике.

Спасибо за то, что выбрали эту книгу. Пусть она станет вашим верным спутником в мире Python и откроет двери к новым достижениям.

С уважением,

Джейд картер

Глава 1. Работа с большими данными

1.1 Распределенная обработка данных с Dask и PySpark

Работа с большими объемами данных требует инструментов, которые позволяют эффективно распределять вычисления между несколькими процессорами или даже серверами. Python предлагает две мощные библиотеки для таких задач – Dask и PySpark. Каждая из них разработана для обработки больших данных, но они имеют свои уникальные особенности и подходы. Разберем их по отдельности, чтобы понять, как их использовать, и приведем примеры.

Dask: инструмент для масштабирования локальных задач

Dask – это библиотека, которая позволяет расширить вычисления на вашем компьютере, эффективно распределяя их между ядрами процессора или несколькими машинами в кластере. Она идеально подходит для тех случаев, когда объем данных превышает доступную оперативную память, но вы хотите сохранить гибкость работы с Python.

Основные особенности Dask:

1. Dask совместим с большинством популярных библиотек Python, таких как Pandas, NumPy и Scikit-learn.
2. Он поддерживает ленивые вычисления: операции выполняются только при необходимости.
3. Dask позволяет работать как с массивами данных (аналог NumPy), так и с таблицами (аналог Pandas).

Пример использования Dask для обработки данных:

Предположим, у нас есть большой CSV-файл с данными о продажах. Его объем превышает объем оперативной памяти, поэтому обычные инструменты, такие как Pandas, не могут загрузить файл целиком.

```python
import dask.dataframe as dd
# Загрузка большого CSV-файла с помощью Dask
df = dd.read_csv('sales_data_large.csv')
```

```
# Выполнение простых операций (например, фильтрация по значению)
filtered_df = df[df['sales'] > 1000]
# Группировка и вычисление суммарных продаж
sales_summary = filtered_df.groupby('region')['sales'].sum()
# Выполнение вычислений (операции "ленивые", пока мы не вызовем .compute())
result = sales_summary.compute()
# Вывод результатов
print(result)
```

Объяснение кода:

1. `dd.read_csv()`: Вместо загрузки всего файла в память, Dask загружает его частями (по "чанкам").
2. Ленивые вычисления: Все операции, такие как фильтрация и группировка, откладываются до вызова `compute()`.
3. Параллельное выполнение: Dask автоматически распределяет работу между всеми доступными ядрами процессора.

Когда использовать Dask:

– Когда ваши данные не помещаются в память.

– Когда вы уже используете библиотеки Python, такие как Pandas или NumPy, и хотите масштабировать их.

– Когда вам нужно быстро настроить распределенные вычисления на одной или нескольких машинах.

PySpark: инструмент для кластерного вычисления

PySpark – это Python-интерфейс для Apache Spark, платформы, разработанной специально для обработки больших данных. Spark работает на кластерах, что позволяет масштабировать вычисления до сотен машин.

PySpark особенно популярен в случаях, когда данные хранятся в распределенных системах, таких как HDFS или Amazon S3.

Основные особенности PySpark:

1. PySpark работает с данными в формате **RDD** (Resilient Distributed Dataset) или DataFrame.
2. Он поддерживает широкий спектр операций, включая трансформации данных, машинное обучение и потоковую обработку.

3. PySpark интегрируется с Hadoop и другими системами для хранения больших данных.

Пример использования PySpark для обработки данных:

Допустим, у нас есть большие данные о транзакциях, хранящиеся в формате CSV, и мы хотим вычислить среднее значение транзакций по каждому клиенту.

```python
from pyspark.sql import SparkSession
# Создаем сессию Spark
spark = SparkSession.builder.appName("TransactionAnalysis").getOrCreate()
# Читаем данные из CSV-файла
df = spark.read.csv('transactions_large.csv', header=True, inferSchema=True)
# Выполняем трансформации данных
# 1. Фильтрация транзакций с нулевой суммой
filtered_df = df.filter(df['amount'] > 0)
# 2. Группировка по клиенту и вычисление среднего значения
average_transactions = filtered_df.groupBy('customer_id').avg('amount')
# Показ результатов
average_transactions.show()
# Останавливаем Spark-сессию
spark.stop()
```

Объяснение кода:

1. Создание SparkSession: Это точка входа для работы с PySpark.

2. `spark.read.csv()`: Загружаем данные в формате DataFrame, который поддерживает SQL-подобные операции.

3. Трансформации: Операции, такие как фильтрация и группировка, выполняются параллельно на всех узлах кластера.

4. Результат: PySpark возвращает распределенные данные, которые можно сохранить или преобразовать.

Когда использовать PySpark:

– Когда вы работаете с кластерами и хотите обрабатывать данные на нескольких машинах.

– Когда данные хранятся в распределенных системах, таких как HDFS или Amazon S3.

– Когда нужно интегрировать обработку данных с экосистемой Hadoop.

Сравнение Dask и PySpark

Характеристика	Dask	PySpark
Подход к вычислениям	Локальные или кластерные	Кластерные
Простота настройки	Легкая настройка на одном компьютере	Требует настройки кластера
Совместимость	Интеграция с Python-библиотеками	Интеграция с Hadoop и Spark
Ленивые вычисления	Да	Да
Работа с большими данными	Подходит для "больших данных в одной машине"	Подходит для "больших данных в кластере"
Производительность	Хорошая для локальных задач	Высокая на кластерах

И Dask, и PySpark являются эффективными инструментами для распределенной обработки данных. Выбор между ними зависит от ваших требований. Если вы работаете с данными, которые не помещаются в оперативную память, но ваши вычисления выполняются на одном компьютере, Dask будет лучшим выбором. Если же вы имеете дело с огромными объемами данных, распределенными по нескольким машинам, то PySpark станет незаменимым инструментом.

Обе библиотеки позволяют решать задачи, которые ранее казались невозможными из-за ограничений памяти или производительности, и они помогут вам эффективно работать с данными любого масштаба.

Задачи для практики

Задачи для Dask

Задача 1: Обработка большого CSV-файла

Описание: У вас есть CSV-файл размером 10 ГБ с данными о продажах. Вам нужно вычислить общую сумму продаж по регионам, но файл слишком большой для работы в Pandas.

Решение:

```python
import dask.dataframe as dd
# Загрузка большого CSV-файла
df = dd.read_csv('sales_data_large.csv')
# Проверка структуры данных
print(df.head()) # Показываем первые строки
# Группировка по регионам и подсчет общей суммы продаж
sales_by_region = df.groupby('region')['sales'].sum()
# Выполнение вычислений
result = sales_by_region.compute()
print(result)
```

Объяснение:

– `dd.read_csv` позволяет загружать файлы большего объема, чем объем оперативной памяти.

– `compute` выполняет ленивые вычисления.

Задача 2: Преобразование данных в формате JSON

Описание: Дан файл в формате JSON, содержащий информацию о транзакциях. Необходимо отфильтровать транзакции с суммой менее 1000 и сохранить отфильтрованные данные в новый CSV-файл.

Решение:

```python
import dask.dataframe as dd
# Загрузка JSON-файла
df = dd.read_json('transactions_large.json')
# Фильтрация данных
filtered_df = df[df['amount'] >= 1000]
# Сохранение результатов в новый CSV-файл
filtered_df.to_csv('filtered_transactions_*.csv', index=False)
print("Данные сохранены в файлы CSV.")
```

```

Объяснение:

– Dask автоматически разбивает данные на части, сохраняя их в несколько CSV-файлов.

– Фильтрация выполняется параллельно.

**Задачи для PySpark**

**Задача 3: Анализ логов**

Описание: Имеется файл логов сервера (формат CSV). Ваша задача – подсчитать количество ошибок (строки с `status = "ERROR"`) и вывести их общее количество.

Решение:

```python
from pyspark.sql import SparkSession
# Создаем сессию Spark
spark = SparkSession.builder.appName("LogAnalysis").getOrCreate()
# Загрузка данных из CSV-файла
df = spark.read.csv('server_logs.csv', header=True, inferSchema=True)
# Фильтрация строк с ошибками
errors = df.filter(df['status'] == 'ERROR')
# Подсчет количества ошибок
error_count = errors.count()
print(f"Количество ошибок: {error_count}")
# Завершаем сессию Spark
spark.stop()
```

Объяснение:

– `filter` позволяет выбрать строки с определенным значением.

– `count` подсчитывает количество строк после фильтрации.

Задача 4: Средняя сумма покупок

Описание: Дан CSV-файл с данными о покупках. Ваша задача – вычислить среднюю сумму покупок для каждого клиента.

Решение:

```python
```
```

```
from pyspark.sql import SparkSession
# Создаем сессию Spark
spark = SparkSession.builder.appName("PurchaseAnalysis").getOrCreate()
# Загрузка данных
df = spark.read.csv('purchases.csv', header=True, inferSchema=True)
# Группировка по клиенту и расчет средней суммы покупок
avg_purchases = df.groupBy('customer_id').avg('purchase_amount')
# Показ результатов
avg_purchases.show()
# Завершаем сессию Spark
spark.stop()
```

Объяснение:
– `groupBy` позволяет сгруппировать данные по столбцу.
– `avg` вычисляет среднее значение для каждой группы.

Задача 5: Сортировка больших данных

Описание: У вас есть файл с информацией о транзакциях. Необходимо отсортировать данные по дате транзакции и сохранить результат в новый файл.

Решение:

```python
from pyspark.sql import SparkSession
# Создаем сессию Spark
spark = SparkSession.builder.appName("SortTransactions").getOrCreate()
# Загрузка данных
df = spark.read.csv('transactions_large.csv', header=True, inferSchema=True)
# Сортировка данных по дате
sorted_df = df.orderBy('transaction_date')
# Сохранение отсортированных данных в новый файл
sorted_df.write.csv('sorted_transactions', header=True, mode='overwrite')
print("Данные отсортированы и сохранены.")
# Завершаем сессию Spark
```

```
spark.stop()
```

Объяснение:

– `orderBy` сортирует данные по указанному столбцу.

– `write.csv` сохраняет результат в новом файле.

Эти задачи демонстрируют, как использовать Dask и PySpark для работы с большими объемами данных.

– Dask подходит для локальных задач и интеграции с Python-библиотеками.

– PySpark эффективен для кластерной обработки данных и интеграции с экосистемой Hadoop.

Обе библиотеки упрощают решение задач, которые сложно выполнить традиционными методами из-за ограничений памяти или мощности процессора.

1.2 Потоковая обработка данных с Apache Kafka

Apache Kafka – это мощная платформа для обработки потоков данных в реальном времени. Она широко используется для обработки и анализа событий, поступающих из различных источников, таких как веб-серверы, базы данных, датчики IoT, системы мониторинга и многое другое. Kafka обеспечивает высокую производительность, надежность и масштабируемость, что делает её одним из лучших инструментов для потоковой обработки данных.

В основе Apache Kafka лежат несколько ключевых компонентов:

1. Брокеры – серверы, которые принимают, хранят и доставляют данные.

2. Топики – логические каналы, через которые данные передаются.

3. Продюсеры – приложения или устройства, которые отправляют данные в Kafka.

4. Консьюмеры – приложения, которые получают данные из Kafka.

Kafka организует поток данных в виде последовательностей сообщений. Сообщения записываются в топики и разделяются на партиции, что позволяет обрабатывать данные параллельно.

Пример потока данных

Представим, что у нас есть система интернет-магазина, где Kafka используется для обработки событий, таких как заказы, клики на странице, добавление товаров в корзину и платежи. Каждое из этих событий записывается в топик Kafka. Например, топик `orders` может содержать события, описывающие новые заказы.

Установка и настройка Apache Kafka

Перед началом работы убедитесь, что Kafka установлена. Для локальной работы используйте официальные сборки Kafka с сайта [Apache Kafka](https://kafka.apache.org/).

1. Установите Kafka и запустите ZooKeeper, необходимый для управления брокерами.
2. Запустите Kafka-брокер.
3. Создайте топик с помощью команды:

```bash
bin/kafka-topics.sh –create –topic orders –bootstrap-server localhost:9092 –partitions 3 –replication-factor 1
```

Отправка данных в Kafka

Теперь создадим простого продюсера на Python, который будет отправлять данные в топик `orders`. Для работы с Kafka на Python используется библиотека `confluent-kafka`. Установите её с помощью команды:

```bash
pip install confluent-kafka
```

Пример кода, который отправляет сообщения в топик:

```python
from confluent_kafka import Producer
import json
import time
# Настройки продюсера
producer_config = {
```

```
'bootstrap.servers': 'localhost:9092' # Адрес Kafka-брокера
}
# Создание продюсера
producer = Producer(producer_config)
# Функция для обратного вызова при успешной отправке сообщения
def delivery_report(err, msg):
if err is not None:
print(f'Ошибка доставки сообщения: {err}')
else:
print(f'Сообщение отправлено: {msg.topic()} [{msg.partition()}]')
# Отправка данных в Kafka
orders = [
{'order_id': 1, 'product': 'Laptop', 'price': 1000},
{'order_id': 2, 'product': 'Phone', 'price': 500},
{'order_id': 3, 'product': 'Headphones', 'price': 150}
]
for order in orders:
producer.produce(
'orders',
key=str(order['order_id']),
value=json.dumps(order),
callback=delivery_report
)
producer.flush() # Отправка сообщений в брокер
time.sleep(1)
```

В этом примере продюсер отправляет JSON-объекты в топик `orders`. Каждое сообщение содержит данные о заказе.

Чтение данных из Kafka

Теперь создадим консьюмера, который будет читать сообщения из топика `orders`.

```python
from confluent_kafka import Consumer, KafkaException
# Настройки консьюмера
consumer_config = {
```

```
'bootstrap.servers': 'localhost:9092',
'group.id': 'order-group', # Группа консьюмеров
'auto.offset.reset': 'earliest' # Начало чтения с первой записи
}
# Создание консьюмера
consumer = Consumer(consumer_config)
# Подписка на топик
consumer.subscribe(['orders'])
# Чтение сообщений из Kafka
try:
while True:
msg = consumer.poll(1.0) # Ожидание сообщения (1 секунда)
if msg is None:
continue
if msg.error():
if msg.error().code() == KafkaException._PARTITION_EOF:
# Конец партиции
continue
else:
print(f"Ошибка: {msg.error()}")
break
# Обработка сообщения
print(f"Получено сообщение: {msg.value().decode('utf-8')}")
except KeyboardInterrupt:
print("Завершение работы…")
finally:
# Закрытие консьюмера
consumer.close()
```

В этом примере консьюмер подключается к Kafka, читает сообщения из топика `orders` и выводит их на экран.

Потоковая обработка данных

Kafka часто используется совместно с платформами потоковой обработки, такими как Apache Spark или Apache Flink, для анализа

данных в реальном времени. Однако вы также можете обрабатывать данные прямо в Python.

Например, предположим, что мы хотим обработать события из топика `orders` и рассчитать суммарную стоимость всех заказов:

```python
from confluent_kafka import Consumer
import json
# Настройки консьюмера
consumer_config = {
'bootstrap.servers': 'localhost:9092',
'group.id': 'order-sum-group',
'auto.offset.reset': 'earliest'
}
# Создание консьюмера
consumer = Consumer(consumer_config)
consumer.subscribe(['orders'])
# Суммарная стоимость заказов
total_sales = 0
try:
while True:
msg = consumer.poll(1.0)
if msg is None:
continue
if msg.error():
continue
# Обработка сообщения
order = json.loads(msg.value().decode('utf-8'))
total_sales += order['price']
print(f"Обработан заказ: {order['order_id']}, текущая сумма: {total_sales}")
except KeyboardInterrupt:
print(f"Общая сумма всех заказов: {total_sales}")
finally:
consumer.close()
```

Преимущества использования Kafka

1. Высокая производительность. Kafka поддерживает миллионы событий в секунду благодаря своей архитектуре и использованию партиций.

2. Надежность. Данные хранятся в Kafka до тех пор, пока их не обработают все подписчики.

3. Масштабируемость. Kafka легко масштабируется путем добавления новых брокеров.

4. Универсальность. Kafka поддерживает интеграцию с большинством современных инструментов обработки данных.

Apache Kafka предоставляет мощный набор инструментов для потоковой обработки данных. Используя Python, вы можете легко настроить передачу данных, их обработку и анализ в реальном времени. Это особенно полезно для систем, где требуется высокая производительность и минимальная задержка при обработке больших потоков данных.

Задачи для практики

Задача 1: Фильтрация событий по условию

Описание:

У вас есть топик `clickstream`, содержащий события о кликах на веб-сайте. Каждое событие содержит следующие поля:

– `user_id` – идентификатор пользователя.

– `url` – URL-адрес, на который был клик.

– `timestamp` – время клика.

Ваша задача: создать консьюмера, который будет читать события из Kafka, фильтровать только события с URL-адресами, содержащими слово "product", и сохранять их в новый топик `filtered_clicks`.

Решение:

```python
from confluent_kafka import Producer, Consumer
```

```
import json
# Настройки Kafka
broker = 'localhost:9092'
# Создание продюсера для записи в новый топик
producer = Producer({'bootstrap.servers': broker})
def produce_filtered_event(event):
producer.produce('filtered_clicks', value=json.dumps(event))
producer.flush()
# Создание консьюмера для чтения из исходного топика
consumer = Consumer({
'bootstrap.servers': broker,
'group.id': 'clickstream-group',
'auto.offset.reset': 'earliest'
})
consumer.subscribe(['clickstream'])
# Чтение и фильтрация событий
try:
while True:
msg = consumer.poll(1.0)
if msg is None:
continue
if msg.error():
continue
# Преобразуем сообщение из Kafka в Python-объект
event = json.loads(msg.value().decode('utf-8'))
# Фильтруем события с URL, содержащими "product"
if 'product' in event['url']:
print(f"Фильтруем событие: {event}")
produce_filtered_event(event)
except KeyboardInterrupt:
print("Завершение работы.")
finally:
consumer.close()
```

Объяснение:

– Консьюмер читает события из топика `clickstream`.

– Каждое сообщение проверяется на наличие слова "product" в поле `url`.

– Отфильтрованные события отправляются в новый топик `filtered_clicks` через продюсера.

Задача 2: Подсчет количества событий в реальном времени

Описание:

Топик `log_events` содержит логи системы. Каждое сообщение содержит:

– `log_level` (например, "INFO", "ERROR", "DEBUG").

– `message` (текст лога).

Ваша задача: написать программу, которая считает количество событий уровня "ERROR" в реальном времени и каждые 10 секунд выводит их общее количество.

Решение:

```python
from confluent_kafka import Consumer
import time
# Настройки Kafka
broker = 'localhost:9092'
# Создание консьюмера
consumer = Consumer({
'bootstrap.servers': broker,
'group.id': 'log-group',
'auto.offset.reset': 'earliest'
})
consumer.subscribe(['log_events'])
error_count = 0
start_time = time.time()
try:
while True:
msg = consumer.poll(1.0)
if msg is None:
continue
if msg.error():
continue
```

```
# Преобразуем сообщение в Python-объект
log_event = json.loads(msg.value().decode('utf-8'))
# Увеличиваем счетчик, если уровень лога "ERROR"
if log_event['log_level'] == 'ERROR':
error_count += 1
# Каждые 10 секунд выводим текущий счетчик
if time.time() – start_time >= 10:
print(f"Количество ошибок за последние 10 секунд: {error_count}")
error_count = 0
start_time = time.time()
except KeyboardInterrupt:
print("Завершение работы.")
finally:
consumer.close()
```

Объяснение:

– Консьюмер читает события из топика `log_events`.

– Если уровень лога "ERROR", увеличивается счетчик `error_count`.

– Каждые 10 секунд программа выводит количество событий "ERROR" и сбрасывает счетчик.

Задача 3: Агрегация данных по группам

Описание:

Топик `transactions` содержит данные о финансовых транзакциях:

– `user_id` – идентификатор пользователя.

– `amount` – сумма транзакции.

Ваша задача: написать программу, которая подсчитывает общую сумму транзакций для каждого пользователя и выводит результаты в реальном времени.

Решение:

```python
from confluent_kafka import Consumer
import json
from collections import defaultdict
# Настройки Kafka
broker = 'localhost:9092'
```

```
# Создание консьюмера
consumer = Consumer({
'bootstrap.servers': broker,
'group.id': 'transaction-group',
'auto.offset.reset': 'earliest'
})
consumer.subscribe(['transactions'])
# Словарь для хранения сумм по пользователям
user_totals = defaultdict(float)
try:
while True:
msg = consumer.poll(1.0)
if msg is None:
continue
if msg.error():
continue
# Преобразуем сообщение в Python-объект
transaction = json.loads(msg.value().decode('utf-8'))
# Обновляем сумму для пользователя
user_id = transaction['user_id']
user_totals[user_id] += transaction['amount']
# Вывод текущих сумм
print(f"Текущая сумма транзакций по пользователям: {dict(user_totals)}")
except KeyboardInterrupt:
print("Завершение работы.")
finally:
consumer.close()
```

Объяснение:

– Консьюмер читает данные из топика `transactions`.

– Для каждого пользователя обновляется сумма его транзакций в словаре `user_totals`.

– Программа выводит текущие суммы по всем пользователям.

Задача 4: Сохранение обработанных данных в файл

Описание:
Топик `sensor_data` содержит данные с датчиков IoT:
– `sensor_id` – идентификатор датчика.
– `temperature` – измеренная температура.
– `timestamp` – время измерения.
Ваша задача: написать программу, которая сохраняет все данные о температуре выше 30°C в файл `high_temp.json`.
Решение:
```python
from confluent_kafka import Consumer
import json
# Настройки Kafka
broker = 'localhost:9092'
# Создание консьюмера
consumer = Consumer({
'bootstrap.servers': broker,
'group.id': 'sensor-group',
'auto.offset.reset': 'earliest'
})
consumer.subscribe(['sensor_data'])
# Открываем файл для записи
with open('high_temp.json', 'w') as outfile:
try:
while True:
msg = consumer.poll(1.0)
if msg is None:
continue
if msg.error():
continue
# Преобразуем сообщение в Python-объект
sensor_data = json.loads(msg.value().decode('utf-8'))
# Сохраняем данные, если температура выше 30°C
if sensor_data['temperature'] > 30:
json.dump(sensor_data, outfile)
outfile.write('\n') # Новый ряд для каждого объекта
except KeyboardInterrupt:
print("Завершение работы.")
```

finally:
consumer.close()
```
Объяснение:
– Консьюмер читает данные из топика `sensor_data`.
– Данные с температурой выше 30°C записываются в файл `high_temp.json`.

**Задача 5: Обнаружение аномалий в данных**
Описание:
В топик `temperature_readings` поступают данные о температуре из различных городов:
– `city` – название города.
– `temperature` – измеренная температура.
– `timestamp` – время измерения.
Ваша задача: написать программу, которая будет находить и выводить аномалии – случаи, когда температура превышает 40°C или опускается ниже -10°C.
Решение:
```python
from confluent_kafka import Consumer
import json
# Настройки Kafka
broker = 'localhost:9092'
# Создание консьюмера
consumer = Consumer({
'bootstrap.servers': broker,
'group.id': 'temperature-group',
'auto.offset.reset': 'earliest'
})
consumer.subscribe(['temperature_readings'])
try:
while True:
msg = consumer.poll(1.0)
if msg is None:
continue
```
```

```
if msg.error():
continue
# Преобразуем сообщение в Python-объект
reading = json.loads(msg.value().decode('utf-8'))
# Проверяем на аномалии
if reading['temperature'] > 40 or reading['temperature'] < -10:
print(f"Аномалия! Город: {reading['city']}, Температура: {reading['temperature']}°C")
except KeyboardInterrupt:
print("Завершение работы.")
finally:
consumer.close()
```

Объяснение:

– Консьюмер читает данные о температуре из топика.

– Если температура выходит за пределы нормального диапазона, программа выводит сообщение об аномалии.

Задача 6: Потоковое объединение данных

Описание:

Есть два топика:

1. `orders` – содержит данные о заказах: `order_id`, `product_id`, `quantity`.
2. `products` – содержит данные о товарах: `product_id`, `product_name`, `price`.

Ваша задача: написать программу, которая объединяет данные из этих двух топиков и выводит итоговую информацию о каждом заказе, включая название продукта и общую стоимость.

Решение:

```python
from confluent_kafka import Consumer
import json
# Настройки Kafka
broker = 'localhost:9092'
# Создание консьюмеров для обоих топиков
order_consumer = Consumer({
```

```
    'bootstrap.servers': broker,
    'group.id': 'order-group',
    'auto.offset.reset': 'earliest'
    })
    product_consumer = Consumer({
    'bootstrap.servers': broker,
    'group.id': 'product-group',
    'auto.offset.reset': 'earliest'
    })
    order_consumer.subscribe(['orders'])
    product_consumer.subscribe(['products'])
    # Словарь для хранения данных о товарах
    product_catalog = {}
    try:
    while True:
    # Чтение данных из топика products
    product_msg = product_consumer.poll(0.1)
    if product_msg and not product_msg.error():
    product = json.loads(product_msg.value().decode('utf-8'))
    product_catalog[product['product_id']] = {
    'name': product['product_name'],
    'price': product['price']
    }
    # Чтение данных из топика orders
    order_msg = order_consumer.poll(0.1)
    if order_msg and not order_msg.error():
    order = json.loads(order_msg.value().decode('utf-8'))
    product_id = order['product_id']
    # Объединение данных о заказе и товаре
    if product_id in product_catalog:
    product = product_catalog[product_id]
    total_price = order['quantity'] * product['price']
    print(f"Заказ {order['order_id']}: {product['name']} x {order['quantity']}
= {total_price} $")
    else:
    print(f"Информация о товаре {product_id} отсутствует.")
    except KeyboardInterrupt:
```

```
print("Завершение работы.")
finally:
order_consumer.close()
product_consumer.close()
```

Объяснение:

– Данные из топика `products` кэшируются в словаре `product_catalog`.

– При чтении заказа из топика `orders` программа объединяет данные и вычисляет итоговую стоимость.

Задача 7: Потоковая обработка с вычислением скользящего среднего

Описание:

В топик `stock_prices` поступают данные о ценах акций:

– `symbol` – тикер акции.

– `price` – текущая цена.

– `timestamp` – время.

Ваша задача: вычислять скользящее среднее цены акции за последние 5 сообщений для каждого тикера.

Решение:

```python
from confluent_kafka import Consumer
import json
from collections import defaultdict, deque
# Настройки Kafka
broker = 'localhost:9092'
# Создание консьюмера
consumer = Consumer({
'bootstrap.servers': broker,
'group.id': 'stocks-group',
'auto.offset.reset': 'earliest'
})
consumer.subscribe(['stock_prices'])
# Дек для хранения последних цен по тикерам
price_window = defaultdict(lambda: deque(maxlen=5))
```

```
try:
while True:
msg = consumer.poll(1.0)
if msg is None:
continue
if msg.error():
continue
# Преобразуем сообщение в Python-объект
stock_data = json.loads(msg.value().decode('utf-8'))
# Добавляем цену в окно
symbol = stock_data['symbol']
price_window[symbol].append(stock_data['price'])
# Вычисляем скользящее среднее
moving_average = sum(price_window[symbol]) / len(price_window[symbol])
print(f"Скользящее среднее для {symbol}: {moving_average:.2f}")
except KeyboardInterrupt:
print("Завершение работы.")
finally:
consumer.close()
```

Объяснение:

– Используется `deque` для хранения последних 5 цен.

– Скользящее среднее вычисляется как сумма значений, делённая на их количество.

Задача 8: Генерация уведомлений

Описание:

В топик `user_actions` поступают данные о действиях пользователей:

– `user_id` – идентификатор пользователя.

– `action` – выполненное действие (например, "login", "purchase").

Напишите программу, которая отслеживает пользователей, выполнивших вход (`login`), но не совершивших покупку (`purchase`) в течение 10 минут, и отправляет уведомление в топик `notifications`.

Решение:

```python
```

```
from confluent_kafka import Consumer, Producer
import json
from datetime import datetime, timedelta
# Настройки Kafka
broker = 'localhost:9092'
# Создание консьюмера
consumer = Consumer({
'bootstrap.servers': broker,
'group.id': 'user-actions-group',
'auto.offset.reset': 'earliest'
})
producer = Producer({'bootstrap.servers': broker})
consumer.subscribe(['user_actions'])
# Словарь для отслеживания пользователей
user_login_time = {}
try:
while True:
msg = consumer.poll(1.0)
if msg is None:
continue
if msg.error():
continue
# Преобразуем сообщение в Python-объект
action = json.loads(msg.value().decode('utf-8'))
user_id = action['user_id']
action_type = action['action']
timestamp = datetime.fromisoformat(action['timestamp'])
if action_type == 'login':
user_login_time[user_id] = timestamp
elif action_type == 'purchase' and user_id in user_login_time:
del user_login_time[user_id]
# Проверяем, прошло ли 10 минут
current_time = datetime.now()
for user, login_time in list(user_login_time.items()):
if current_time – login_time > timedelta(minutes=10):
notification = {'user_id': user, 'message': 'Сделайте покупку!'}
producer.produce('notifications', value=json.dumps(notification))
```

```
print(f"Уведомление отправлено для пользователя {user}")
del user_login_time[user]
except KeyboardInterrupt:
print("Завершение работы.")
finally:
consumer.close()
```

Объяснение:

– Время входа пользователей сохраняется в словаре.

– Если с момента входа прошло более 10 минут и покупка не совершена, генерируется уведомление.

Эти задачи показывают, как использовать Apache Kafka для решения реальных задач, таких как фильтрация событий, подсчет статистики, агрегация данных и сохранение обработанной информации. Эти примеры помогут вам освоить основные подходы к работе с потоками данных в реальном времени.

1.3 Работа с базами данных: SQLAlchemy и интеграция с Pandas

SQLAlchemy – это мощная библиотека для работы с базами данных в Python. Она предоставляет инструменты для удобного взаимодействия с реляционными базами данных через ORM (Object Relational Mapping) или с использованием чистого SQL.

Pandas же идеально подходит для анализа данных, но иногда данные, которые мы хотим обработать, хранятся в базах данных. Для этого SQLAlchemy и Pandas можно эффективно интегрировать, чтобы выгружать данные из базы, обрабатывать их в Pandas и сохранять обратно.

Установка и подключение

Для начала работы установите библиотеку SQLAlchemy:

```bash
pip install sqlalchemy
```

Если вы используете SQLite, дополнительных действий не требуется. Для других баз данных, таких как PostgreSQL или MySQL, также потребуется установить драйверы, например:

```bash
pip install psycopg2 # Для PostgreSQL
pip install pymysql # Для MySQL
```

Создайте подключение к базе данных с помощью SQLAlchemy. Например, для SQLite это будет выглядеть так:

```python
from sqlalchemy import create_engine
# Создаем подключение к базе данных SQLite
engine = create_engine('sqlite:///example.db', echo=True)
```

Здесь `echo=True` означает, что в консоль будут выводиться SQL-запросы, выполняемые через SQLAlchemy, что полезно для отладки.

Создание таблиц и работа с ORM

SQLAlchemy поддерживает два основных подхода: работа через ORM и использование SQL-запросов напрямую. Рассмотрим оба.

Создадим таблицу для хранения информации о пользователях:

```python
from sqlalchemy import Table, Column, Integer, String, MetaData
# Создаем метаданные
metadata = MetaData()
# Определяем таблицу
users = Table(
'users', metadata,
Column('id', Integer, primary_key=True),
Column('name', String),
Column('age', Integer),
Column('email', String)
)
```

```
# Создаем таблицу в базе данных
metadata.create_all(engine)
```
Теперь таблица `users` создана в базе данных.
Для добавления данных используем объект подключения:
```python
from sqlalchemy import insert
# Подключаемся к базе данных
conn = engine.connect()
# Добавляем данные
insert_query = insert(users).values([
{'name': 'Alice', 'age': 25, 'email': 'alice@example.com'},
{'name': 'Bob', 'age': 30, 'email': 'bob@example.com'},
{'name': 'Charlie', 'age': 35, 'email': 'charlie@example.com'}
])
conn.execute(insert_query)
print("Данные добавлены в таблицу.")
```

Чтение данных и интеграция с Pandas

Чтобы выгрузить данные из базы данных в Pandas, SQLAlchemy предоставляет удобный метод. Используем Pandas для выполнения SQL-запроса:

```python
import pandas as pd
# Чтение данных из таблицы users
query = "SELECT * FROM users"
df = pd.read_sql(query, engine)
print(df)
```
Вывод будет выглядеть так:
```
id name age email
0 1 Alice 25 alice@example.com
1 2 Bob 30 bob@example.com
2 3 Charlie 35 charlie@example.com
```

```

Теперь данные из базы данных доступны в формате DataFrame, и вы можете применять к ним все мощные инструменты анализа, которые предоставляет Pandas.

**Обработка данных с использованием Pandas**

Допустим, мы хотим найти всех пользователей старше 30 лет и добавить новый столбец с доменом их электронной почты.

```python
# Фильтрация пользователей старше 30 лет
filtered_df = df[df['age'] > 30]
# Добавление нового столбца с доменом электронной почты
filtered_df['email_domain'] = filtered_df['email'].apply(lambda x: x.split('@')[1])
print(filtered_df)
```

Результат будет выглядеть так:

```
id name age email email_domain
2 3 Charlie 35 charlie@example.com example.com
```

Сохранение данных обратно в базу

После обработки данных в Pandas мы можем сохранить их обратно в базу данных. Для этого Pandas предоставляет метод `to_sql`:

```python
# Сохранение отфильтрованных данных в новую таблицу filtered_users
filtered_df.to_sql('filtered_users', engine, if_exists='replace', index=False)
print("Данные сохранены в таблицу filtered_users.")
```

Теперь в базе данных появилась новая таблица `filtered_users`, содержащая обработанные данные.
```

Работа с ORM

Для более сложных сценариев SQLAlchemy поддерживает ORM, позволяющий работать с таблицами как с Python-классами.

Определим класс для таблицы `users`:

```python
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
Base = declarative_base()
class User(Base):
__tablename__ = 'users'
id = Column(Integer, primary_key=True)
name = Column(String)
age = Column(Integer)
email = Column(String)
# Создаем сессию для работы с ORM
Session = sessionmaker(bind=engine)
session = Session()
# Пример чтения данных через ORM
users = session.query(User).filter(User.age > 30).all()
for user in users:
print(f"Имя: {user.name}, Возраст: {user.age}, Email: {user.email}")
```

Этот подход особенно удобен, если вы предпочитаете объектно-ориентированный стиль работы с базой данных.

Пример: Анализ данных с SQLAlchemy и Pandas

Представьте, что у вас есть база данных с информацией о продажах, и вы хотите найти города, в которых средняя сумма покупок превышает 5000.

1. Создадим таблицу:

```python
sales = Table(
'sales', metadata,
Column('id', Integer, primary_key=True),
Column('city', String),
Column('amount', Integer)
)
```

```
metadata.create_all(engine)
# Добавим данные
conn.execute(insert(sales).values([
{'city': 'New York', 'amount': 7000},
{'city': 'Los Angeles', 'amount': 3000},
{'city': 'New York', 'amount': 8000},
{'city': 'Los Angeles', 'amount': 2000},
{'city': 'Chicago', 'amount': 6000}
]))
```

2. Выгрузим данные и найдем среднюю сумму по городам:

```python
# Чтение данных из таблицы sales
query = "SELECT * FROM sales"
sales_df = pd.read_sql(query, engine)
# Вычисление средней суммы по городам
avg_sales = sales_df.groupby('city')['amount'].mean().reset_index()
# Фильтрация городов с средней суммой > 5000
filtered_sales = avg_sales[avg_sales['amount'] > 5000]
print(filtered_sales)
```

Результат:

```
city amount
0 Chicago 6000.0
1 New York 7500.0
```

3. Сохраним результат в таблицу:

```python
filtered_sales.to_sql('high_avg_sales', engine, if_exists='replace', index=False)
```

Теперь обработанные данные сохранены в базе, и вы можете использовать их в дальнейшем.

SQLAlchemy предоставляет мощные возможности для работы с базами данных, а интеграция с Pandas делает обработку данных ещё более удобной и гибкой. Вы можете быстро выгружать данные из базы,

анализировать их с помощью Pandas и сохранять обратно, что упрощает создание аналитических решений и автоматизацию работы с данными.

Задачи для практики

Задача 1: Создание базы данных пользователей и извлечение данных

Описание:

Создайте базу данных `users.db` с таблицей `users`, содержащей следующие столбцы:

– `id` – уникальный идентификатор пользователя.

– `name` – имя пользователя.

– `age` – возраст пользователя.

– `email` – электронная почта.

Добавьте в таблицу данные о пяти пользователях и извлеките всех пользователей старше 30 лет.

Решение:

```python
from sqlalchemy import create_engine, Table, Column, Integer, String, MetaData
import pandas as pd
# Создаем подключение к базе данных SQLite
engine = create_engine('sqlite:///users.db', echo=False)
metadata = MetaData()
# Определяем таблицу users
users = Table(
'users', metadata,
Column('id', Integer, primary_key=True),
Column('name', String),
Column('age', Integer),
Column('email', String)
)
# Создаем таблицу
```

```
metadata.create_all(engine)
# Добавляем данные
with engine.connect() as conn:
conn.execute(users.insert(), [
{'name': 'Alice', 'age': 25, 'email': 'alice@example.com'},
{'name': 'Bob', 'age': 35, 'email': 'bob@example.com'},
{'name': 'Charlie', 'age': 32, 'email': 'charlie@example.com'},
{'name': 'Diana', 'age': 28, 'email': 'diana@example.com'},
{'name': 'Eve', 'age': 40, 'email': 'eve@example.com'}
])
# Извлечение пользователей старше 30 лет
query = "SELECT * FROM users WHERE age > 30"
df = pd.read_sql(query, engine)
print(df)
```

Результат:

```
id name age email
1 2 Bob 35 bob@example.com
2 3 Charlie 32 charlie@example.com
4 5 Eve 40 eve@example.com
```

Задача 2: Подсчет пользователей по возрастным группам

Описание:

Используя базу данных `users.db`, разделите пользователей на две группы: младше 30 лет и 30 лет и старше. Посчитайте количество пользователей в каждой группе.

Решение:

```python
# Чтение данных из таблицы
df = pd.read_sql("SELECT * FROM users", engine)
# Добавление возрастной группы
df['age_group'] = df['age'].apply(lambda x: 'Under 30' if x < 30 else '30 and above')
# Подсчет пользователей по группам
group_counts = df.groupby('age_group')['id'].count().reset_index()
print(group_counts)
```

```
Результат:
```
age_group id
0 30 and above 3
1 Under 30 2
```

**Задача 3: Сохранение агрегированных данных в новую таблицу**

Описание:

Сохраните результаты подсчета пользователей по возрастным группам в новую таблицу `age_groups` в базе данных `users.db`.

Решение:

```python
# Сохранение в новую таблицу
group_counts.to_sql('age_groups', engine, if_exists='replace', index=False)
# Проверка сохраненных данных
saved_data = pd.read_sql("SELECT * FROM age_groups", engine)
print(saved_data)
```

Результат:

```
age_group id
0 30 and above 3
1 Under 30 2
```

Задача 4: Поиск наиболее популярных доменов электронной почты

Описание:

Добавьте данные о пользователях с разными адресами электронной почты. Найдите, какие домены (`example.com`, `gmail.com` и т.д.) встречаются чаще всего.

Решение:

```python
# Добавление новых данных
with engine.connect() as conn:
conn.execute(users.insert(), [
```
```

{'name': 'Frank', 'age': 29, 'email': 'frank@gmail.com'},
{'name': 'Grace', 'age': 37, 'email': 'grace@gmail.com'},
{'name': 'Helen', 'age': 33, 'email': 'helen@example.com'}
])
# Чтение данных
df = pd.read_sql("SELECT * FROM users", engine)
# Выделение доменов
df['email_domain'] = df['email'].apply(lambda x: x.split('@')[1])
# Подсчет частоты доменов
domain_counts = df['email_domain'].value_counts().reset_index()
domain_counts.columns = ['email_domain', 'count']
print(domain_counts)
```

Результат:
```
email_domain count
0 example.com 5
1 gmail.com 2
```

**Задача 5: Создание таблицы продаж и анализ доходов**

Описание:

Создайте таблицу `sales`, содержащую данные о продажах:

– `id` – идентификатор продажи.

– `product` – название продукта.

– `price` – цена продукта.

– `quantity` – количество проданных единиц.

Рассчитайте общий доход для каждого продукта и сохраните результаты в новую таблицу `product_revenues`.

Решение:

```python
# Определение таблицы sales
sales = Table(
'sales', metadata,
Column('id', Integer, primary_key=True),
Column('product', String),
Column('price', Integer),
Column('quantity', Integer)
```
```

```
)
metadata.create_all(engine)
# Добавление данных
with engine.connect() as conn:
conn.execute(sales.insert(), [
{'product': 'Laptop', 'price': 1000, 'quantity': 3},
{'product': 'Phone', 'price': 500, 'quantity': 5},
{'product': 'Tablet', 'price': 300, 'quantity': 7}
])
# Чтение данных
sales_df = pd.read_sql("SELECT * FROM sales", engine)
# Расчет общего дохода
sales_df['revenue'] = sales_df['price'] * sales_df['quantity']
revenues = sales_df.groupby('product')['revenue'].sum().reset_index()
# Сохранение в новую таблицу
revenues.to_sql('product_revenues', engine, if_exists='replace', index=False)
# Проверка сохраненных данных
saved_revenues = pd.read_sql("SELECT * FROM product_revenues", engine)
print(saved_revenues)
```

Результат:

```
product revenue
0 Laptop 3000
1 Phone 2500
2 Tablet 2100
```

Задача 6: Фильтрация данных по динамическому запросу

Описание:

Создайте функцию, которая принимает минимальную цену и возвращает список продуктов, стоимость которых выше указанного значения.

Решение:

```python
def filter_products_by_price(min_price):
```

```
query = f"SELECT * FROM sales WHERE price > {min_price}"
result_df = pd.read_sql(query, engine)
return result_df
# Фильтрация продуктов с ценой выше 400
filtered_products = filter_products_by_price(400)
print(filtered_products)
```

Результат:

```
id product price quantity
0 1 Laptop 1000 3
1 2 Phone 500 5
```

Задача 7: Определение наиболее активных пользователей

Описание:

В таблице `activity_log` содержатся данные о действиях пользователей:

– `id` – идентификатор записи.
– `user_id` – идентификатор пользователя.
– `action` – выполненное действие.
– `timestamp` – время выполнения действия.

Определите, кто из пользователей совершил наибольшее количество действий.

Решение:

```python
from sqlalchemy import Table, Column, Integer, String, DateTime
from datetime import datetime
# Определение таблицы activity_log
activity_log = Table(
'activity_log', metadata,
Column('id', Integer, primary_key=True),
Column('user_id', Integer),
Column('action', String),
Column('timestamp', DateTime)
)
metadata.create_all(engine)
# Добавление данных
```

```
with engine.connect() as conn:
conn.execute(activity_log.insert(), [
{'user_id': 1, 'action': 'login', 'timestamp': datetime(2025, 1, 1, 10, 0)},
{'user_id': 1, 'action': 'purchase', 'timestamp': datetime(2025, 1, 1, 10, 5)},
{'user_id': 2, 'action': 'login', 'timestamp': datetime(2025, 1, 1, 11, 0)},
{'user_id': 1, 'action': 'logout', 'timestamp': datetime(2025, 1, 1, 10, 10)},
{'user_id': 2, 'action': 'purchase', 'timestamp': datetime(2025, 1, 1, 11, 5)},
{'user_id': 2, 'action': 'logout', 'timestamp': datetime(2025, 1, 1, 11, 10)}
])
# Чтение данных
activity_df = pd.read_sql("SELECT * FROM activity_log", engine)
# Подсчет количества действий по пользователям
user_activity = activity_df.groupby('user_id')['id'].count().reset_index()
user_activity.columns = ['user_id', 'action_count']
# Поиск самого активного пользователя
most_active_user =
user_activity.loc[user_activity['action_count'].idxmax()]
print(most_active_user)
```

Результат:

```
user_id 1
action_count 3
```

Задача 8: Подсчет действий по типу

Описание: Для каждого типа действия из таблицы `activity_log` подсчитайте, сколько раз оно выполнялось.

Решение:

```python
# Подсчет количества каждого типа действия
action_counts = activity_df['action'].value_counts().reset_index()
action_counts.columns = ['action', 'count']
print(action_counts)
```

Результат:

```
action count
```

0 login 2
1 purchase 2
2 logout 2
```

**Задача 9: Анализ временных меток**

Описание: Определите, в какие часы дня пользователи наиболее активны.

Решение:

```python
# Извлечение часа из временных меток
activity_df['hour'] = activity_df['timestamp'].dt.hour
# Подсчет действий по часам
hourly_activity = activity_df.groupby('hour')['id'].count().reset_index()
hourly_activity.columns = ['hour', 'action_count']
print(hourly_activity)
```

Результат:

```
hour action_count
0 10 3
1 11 3
```

Задача 10: Создание таблицы доходов от пользователей

Описание: Используя таблицу `sales`, определите, сколько дохода принёс каждый пользователь, и сохраните результаты в таблицу `user_revenues`.

Решение:

```python
# Добавление данных о продажах с указанием user_id
with engine.connect() as conn:
conn.execute(sales.insert(), [
{'product': 'Laptop', 'price': 1000, 'quantity': 1, 'user_id': 1},
{'product': 'Phone', 'price': 500, 'quantity': 2, 'user_id': 1},
{'product': 'Tablet', 'price': 300, 'quantity': 3, 'user_id': 2}
])
# Чтение данных из sales
sales_df = pd.read_sql("SELECT * FROM sales", engine)
```
```

```
# Расчёт дохода для каждого пользователя
sales_df['revenue'] = sales_df['price'] * sales_df['quantity']
user_revenues = sales_df.groupby('user_id')['revenue'].sum().reset_index()
# Сохранение в новую таблицу
user_revenues.to_sql('user_revenues', engine, if_exists='replace', index=False)
# Проверка результатов
saved_user_revenues = pd.read_sql("SELECT * FROM user_revenues", engine)
print(saved_user_revenues)
```

Результат:

```
user_id revenue
0 1 2000
1 2 900
```

Задача 11: Поиск последнего действия пользователей

Описание:Для каждого пользователя из таблицы `activity_log` найдите его последнее действие.

Решение:

```python
# Поиск последнего действия
last_actions = activity_df.sort_values('timestamp').groupby('user_id').last().reset_index()
last_actions = last_actions[['user_id', 'action', 'timestamp']]
print(last_actions)
```

Результат:

```
user_id action timestamp
0 1 logout 2025-01-01 10:10:00
1 2 logout 2025-01-01 11:10:00
```

Задача 12: Фильтрация пользователей с высоким доходом

Описание: Используя таблицу `user_revenues`, выберите всех пользователей, чей доход превышает 1500.

Решение:

```python
# Чтение данных из user_revenues
user_revenues = pd.read_sql("SELECT * FROM user_revenues", engine)
# Фильтрация пользователей с доходом > 1500
high_revenue_users = user_revenues[user_revenues['revenue'] > 1500]
print(high_revenue_users)
```

Результат:

```
user_id revenue
0 1 2000
```

Задача 13: Распределение доходов по продуктам

Описание: Определите, какой процент от общего дохода приносит каждый продукт.

Решение:

```python
# Подсчет общего дохода
total_revenue = sales_df['revenue'].sum()
# Расчет процента дохода по продуктам
sales_df['revenue_percent'] = (sales_df['revenue'] / total_revenue) * 100
product_revenue_percent = sales_df.groupby('product')['revenue_percent'].sum().reset_index()
print(product_revenue_percent)
```

Результат:

```
product revenue_percent
0 Laptop 50.793651
1 Phone 25.396825
2 Tablet 23.809524
```

Эти задачи демонстрируют, как SQLAlchemy и Pandas могут использоваться вместе для создания, управления и анализа данных в

базах данных. Они покрывают такие аспекты, как фильтрация данных, выполнение группировок и агрегатов, интеграция данных и сохранение результатов. Эти примеры помогут вам освоить основные техники работы с базами данных в Python.

Глава 2. Интерактивная визуализация и аналитика

2.1 Использование Plotly для интерактивных графиков

Plotly – это мощная библиотека для создания интерактивных графиков и визуализации данных. Она поддерживает широкий спектр графиков: линейные, столбчатые, тепловые карты, трехмерные визуализации и многие другие. Основное преимущество Plotly – интерактивность: пользователи могут увеличивать масштаб, перемещаться по графикам, а также взаимодействовать с данными в реальном времени.

Для работы с Plotly необходимо установить библиотеку:

```bash
pip install plotly
```

После установки можно использовать Plotly в сочетании с Pandas, что упрощает построение графиков на основе данных из DataFrame. Далее мы подробно рассмотрим примеры использования Plotly для создания различных типов графиков.

Построение простого линейного графика

Рассмотрим пример, где мы визуализируем изменение температуры в течение недели.

```python
import plotly.graph_objects as go
# Данные
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
temperatures = [22, 24, 19, 23, 25, 28, 26]
# Создание графика
fig = go.Figure()
fig.add_trace(go.Scatter(
```

```
x=days,
y=temperatures,
mode='lines+markers', # Линии с точками
name='Temperature',
line=dict(color='blue', width=2),
marker=dict(size=8)
))
# Настройка заголовков
fig.update_layout(
title='Изменение температуры за неделю',
xaxis_title='День недели',
yaxis_title='Температура (°C)',
template='plotly_white'
)
fig.show()
```

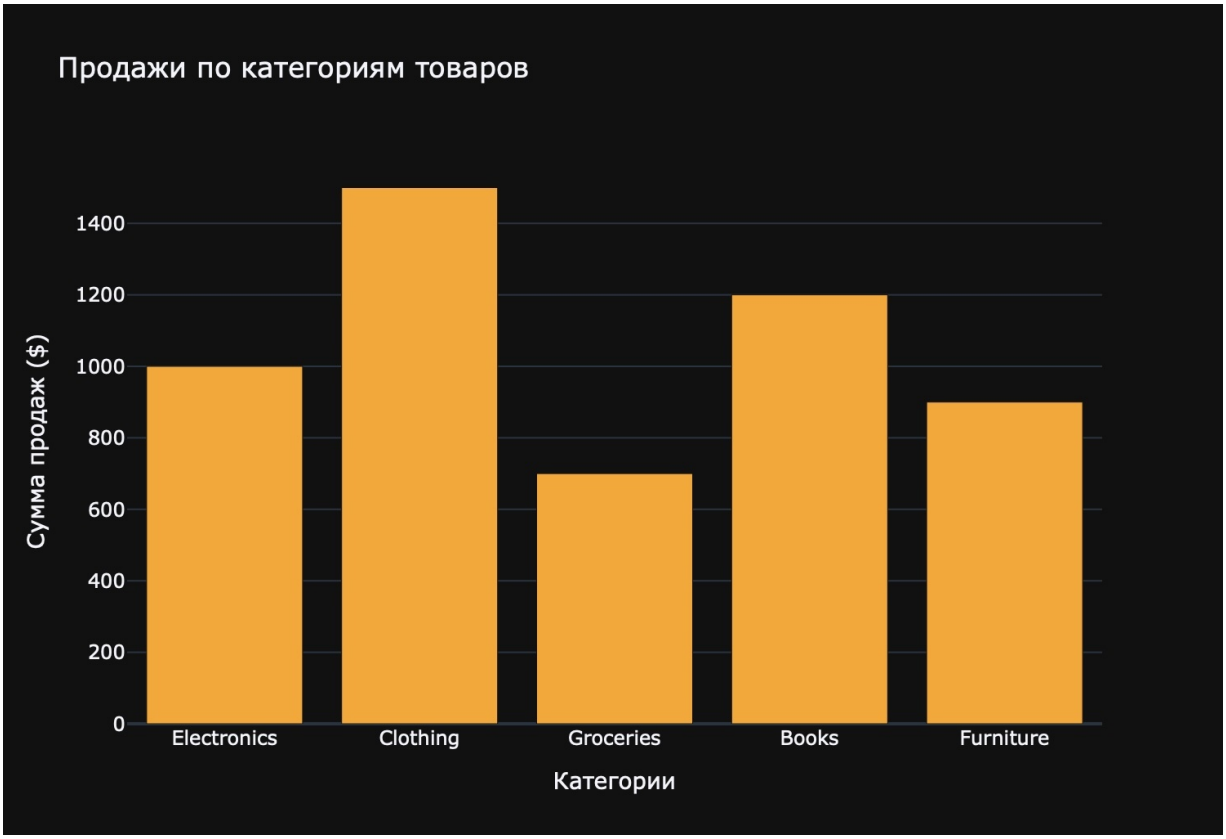
Объяснение:

1. Мы создаём объект `Figure`, добавляя в него данные с помощью `add_trace`.
2. Используем `Scatter` для отображения данных в виде линии с точками.
3. С помощью `update_layout` задаём заголовок графика и подписываем оси.
4. Метод `fig.show()` открывает интерактивный график в браузере.

Построение столбчатого графика

Теперь создадим столбчатый график, чтобы отобразить продажи по различным категориям товаров.

```python
categories = ['Electronics', 'Clothing', 'Groceries', 'Books', 'Furniture']
sales = [1000, 1500, 700, 1200, 900]
fig = go.Figure()
fig.add_trace(go.Bar(
x=categories,
y=sales,
name='Sales',
marker=dict(color='orange')
))
fig.update_layout(
title='Продажи по категориям товаров',
xaxis_title='Категории',
yaxis_title='Сумма продаж ($)',
template='plotly_dark'
)
fig.show()
```



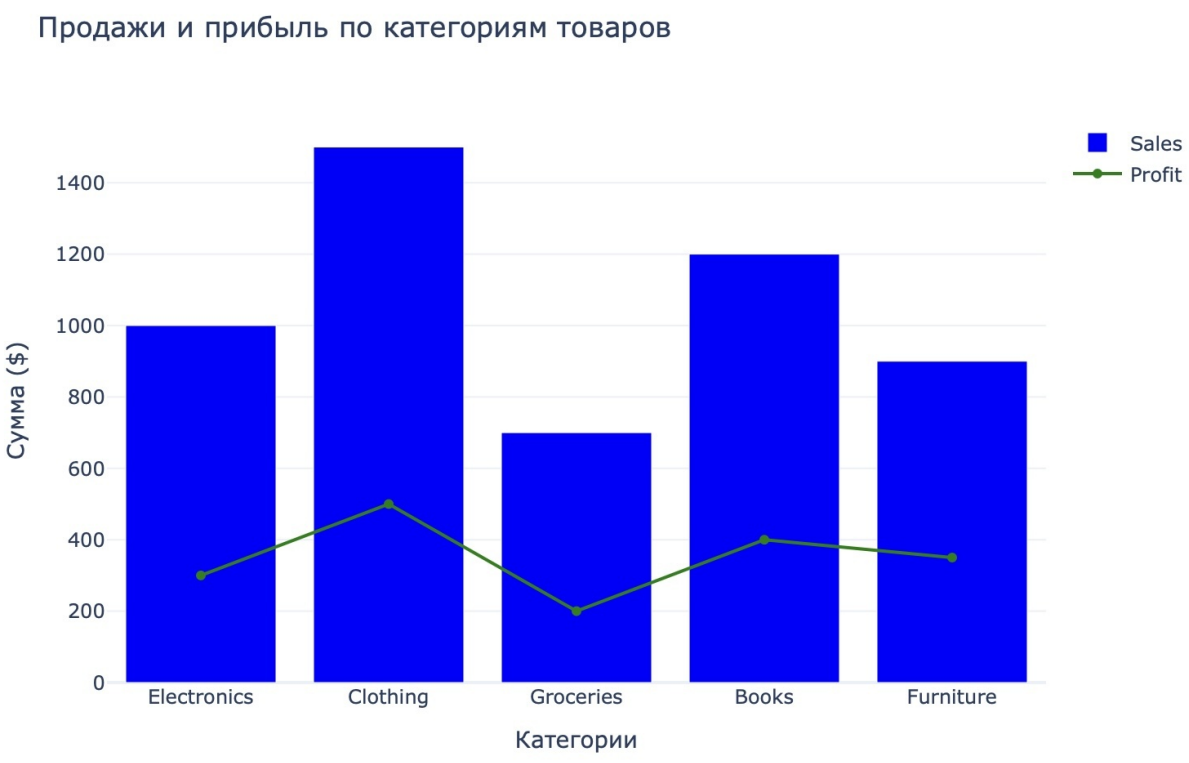
Особенности:
– Используем `go.Bar` для построения столбчатого графика.
– Цвет столбцов задаётся через параметр `marker`.

Построение комбинированного графика

Иногда нужно совмещать разные типы графиков на одном рисунке. Рассмотрим пример, где на одном графике отображаются продажи в виде столбцов и прибыль в виде линии.

```python
profit = [300, 500, 200, 400, 350]
fig = go.Figure()
fig.add_trace(go.Bar(
x=categories,
y=sales,
name='Sales',
marker=dict(color='blue')
))
```

```
fig.add_trace(go.Scatter(
x=categories,
y=profit,
mode='lines+markers',
name='Profit',
line=dict(color='green', width=2)
))
fig.update_layout(
title='Продажи и прибыль по категориям товаров',
xaxis_title='Категории',
yaxis_title='Сумма ($)',
barmode='group',
template='plotly_white'
)
fig.show()
```



Что добавлено:

– Комбинация `Bar` и `Scatter` позволяет визуализировать данные разных типов.

– Параметр `barmode='group'` размещает столбцы по группам, чтобы они не перекрывались.

Построение круговой диаграммы

Для отображения долей в процентах часто используется круговая диаграмма. Например, распределение продаж по категориям.

```python
fig = go.Figure()
fig.add_trace(go.Pie(
labels=categories,
values=sales,
hole=0.3 # Полудонат (дырка в центре)
))
fig.update_layout(
title='Распределение продаж по категориям',
template='plotly_white'
)
fig.show()
```

Особенности:

– Используем `go.Pie` для построения круговой диаграммы.

– Параметр `hole` задаёт размер центральной части, превращая график в "пончиковую" диаграмму.

Построение тепловой карты

Тепловые карты полезны для отображения матриц данных, например, уровня продаж в разных регионах и месяцах.

```python
import numpy as np
regions = ['North', 'South', 'East', 'West']
months = ['January', 'February', 'March', 'April']
sales_data = np.random.randint(100, 1000, size=(4, 4))
fig = go.Figure(data=go.Heatmap(
z=sales_data,
x=months,
y=regions,
colorscale='Viridis' # Цветовая схема
```

```
))
fig.update_layout(
title='Уровень продаж по регионам и месяцам',
xaxis_title='Месяцы',
yaxis_title='Регионы'
)
fig.show()
```

Объяснение:

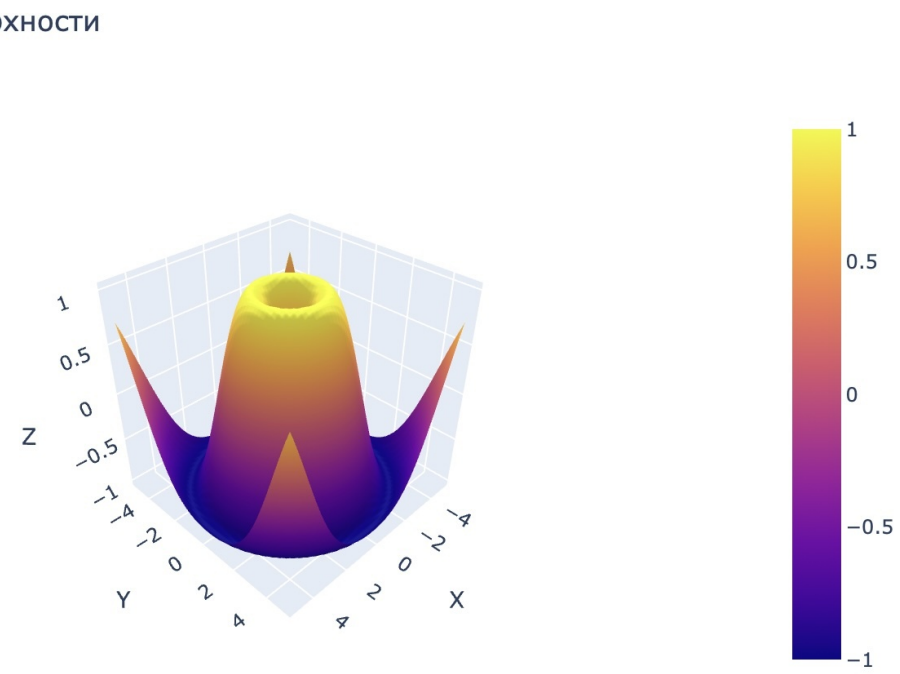
– Используем `go.Heatmap` для отображения данных в виде тепловой карты.

– Параметр `colorscale` задаёт цветовую палитру, визуально усиливая различия между значениями.

Построение трёхмерного графика

Plotly поддерживает трёхмерные визуализации. Например, график, отображающий поверхность функции.

```python
x = np.linspace(-5, 5, 50)
y = np.linspace(-5, 5, 50)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))
fig = go.Figure(data=[go.Surface(z=Z, x=X, y=Y)])
fig.update_layout(
title='3D График поверхности',
scene=dict(
xaxis_title='X',
yaxis_title='Y',
zaxis_title='Z'
)
)
fig.show()
```



Особенности:

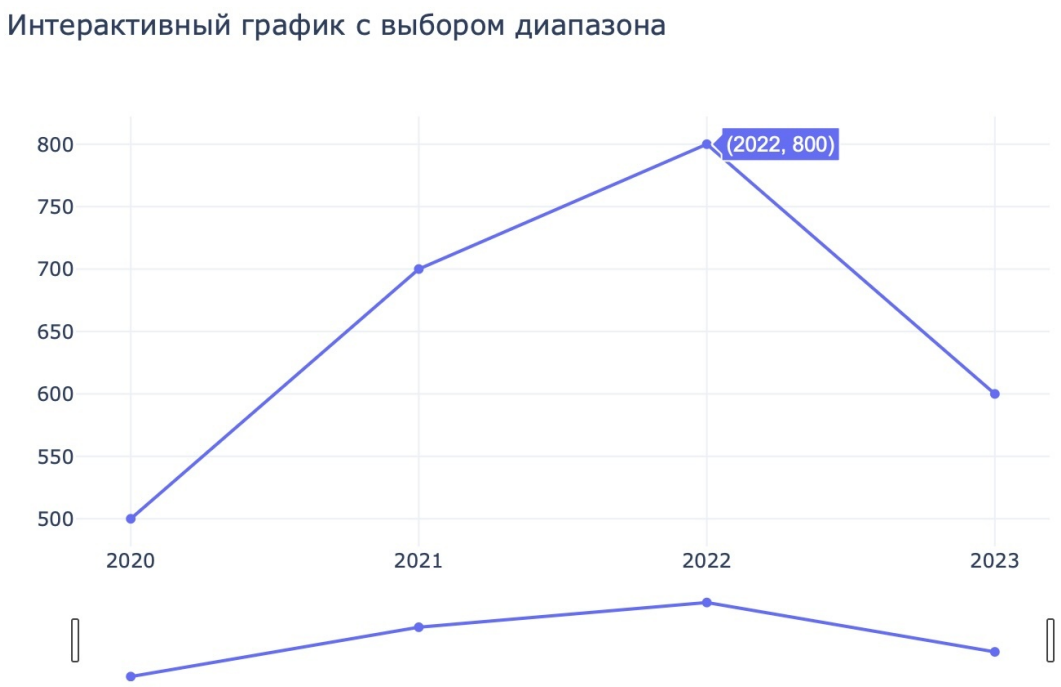
– Используем `go.Surface` для построения трёхмерной поверхности.

– Параметры `scene` задают подписи к осям в трёхмерном пространстве.

Интерактивность с помощью виджетов

Plotly позволяет добавлять интерактивные элементы, такие как слайдеры. Например, график, где пользователь может выбирать диапазон времени.

```python
from plotly.subplots import make_subplots
years = ['2020', '2021', '2022', '2023']
values = [500, 700, 800, 600]
fig = make_subplots(rows=1, cols=1)
fig.add_trace(go.Scatter(
x=years,
y=values,
mode='lines+markers',
name='Yearly Data'
))
fig.update_layout(
title='Интерактивный график с выбором диапазона',
xaxis=dict(rangeslider=dict(visible=True)), # Добавляем ползунок
template='plotly_white'
)
fig.show()
```



Интерактивность:

– Ползунок позволяет выбирать диапазон данных на оси X.

– Это полезно для работы с временными рядами.

Plotly – универсальный инструмент для создания интерактивных графиков. Благодаря множеству типов графиков и богатым возможностям настройки, библиотека подходит для самых разнообразных задач: от анализа данных до их визуальной презентации. Используя Plotly, вы можете не только создавать красивые графики, но и предоставлять пользователям возможность активно взаимодействовать с ними.

Задачи для практики

Задача 1: Построение графика изменения температуры

Описание:

Имеется набор данных о температуре за неделю:

– Дни: `['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']`

– Температура: `[15, 17, 20, 22, 19, 18, 16]`

Постройте линейный график, отображающий изменение температуры. Подпишите оси и добавьте интерактивность.

Решение:

```python
import plotly.graph_objects as go
# Данные
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
temperatures = [15, 17, 20, 22, 19, 18, 16]
# Построение графика
fig = go.Figure()
fig.add_trace(go.Scatter(
x=days,
y=temperatures,
mode='lines+markers',
name='Temperature',
line=dict(color='blue', width=2),
marker=dict(size=8)
))
# Настройка графика
fig.update_layout(
title='Изменение температуры за неделю',
xaxis_title='Дни недели',
yaxis_title='Температура (°C)',
template='plotly_white'
)
# Показ графика
fig.show()
```

Задача 2: Построение круговой диаграммы

Описание:

Имеется информация о продажах по категориям:

– Категории: `['Electronics', 'Clothing', 'Groceries', 'Books', 'Furniture']`

– Продажи: `[1200, 1500, 800, 600, 900]`

Постройте круговую диаграмму, отображающую доли продаж по категориям.

Решение:

```python
import plotly.graph_objects as go
# Данные
categories = ['Electronics', 'Clothing', 'Groceries', 'Books', 'Furniture']
sales = [1200, 1500, 800, 600, 900]
# Построение круговой диаграммы
fig = go.Figure(data=[go.Pie(
labels=categories,
values=sales,
hole=0.4 # Делает диаграмму "пончиковой"
```

```
)])
# Настройка графика
fig.update_layout(
title='Распределение продаж по категориям',
template='plotly_white'
)
# Показ графика
fig.show()
```

Задача 3: Построение столбчатого графика с несколькими категориями

Описание:

Имеется информация о продажах в двух магазинах по категориям товаров:

– Категории: `['Electronics', 'Clothing', 'Groceries', 'Books', 'Furniture']`

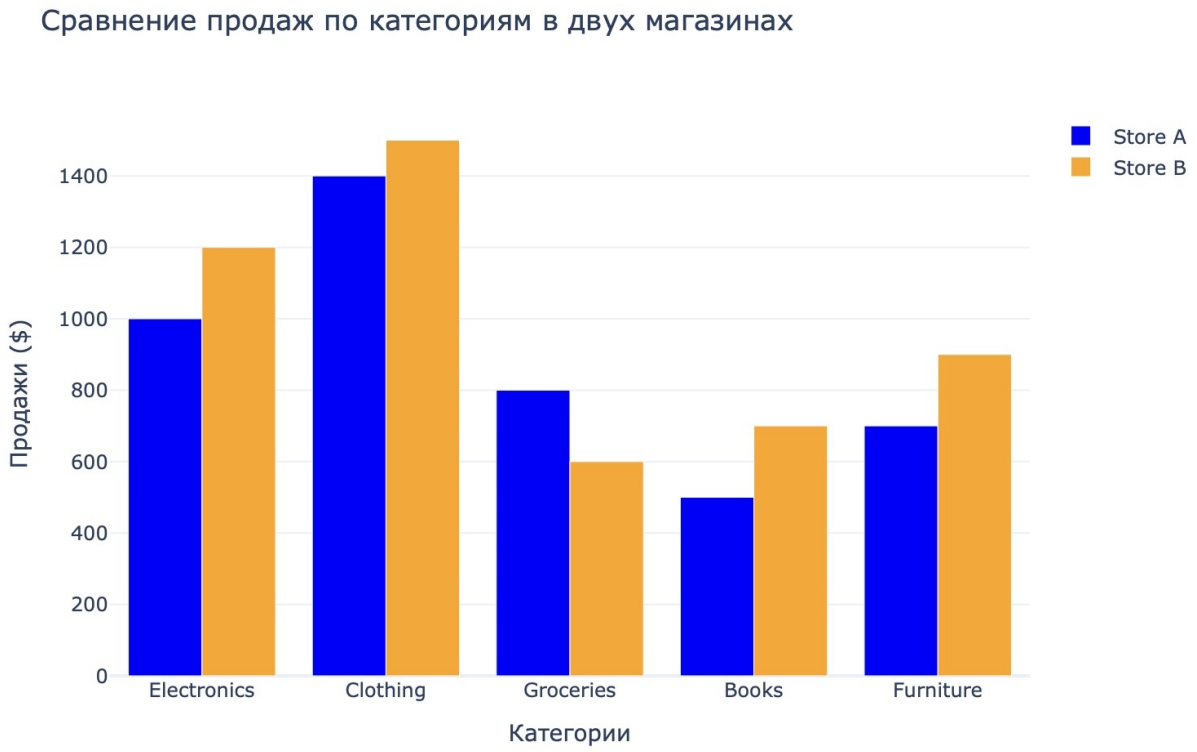
– Продажи в магазине A: `[1000, 1400, 800, 500, 700]`

– Продажи в магазине B: `[1200, 1500, 600, 700, 900]`

Постройте группированный столбчатый график для сравнения продаж в двух магазинах.

Решение:

```python
# Данные
categories = ['Electronics', 'Clothing', 'Groceries', 'Books', 'Furniture']
sales_a = [1000, 1400, 800, 500, 700]
sales_b = [1200, 1500, 600, 700, 900]
# Построение графика
fig = go.Figure()
fig.add_trace(go.Bar(
x=categories,
y=sales_a,
name='Store A',
marker=dict(color='blue')
))
fig.add_trace(go.Bar(
x=categories,
y=sales_b,
name='Store B',
marker=dict(color='orange')
))
# Настройка графика
fig.update_layout(
title='Сравнение продаж по категориям в двух магазинах',
xaxis_title='Категории',
yaxis_title='Продажи ($)',
barmode='group',
template='plotly_white'
)
# Показ графика
fig.show()
```



Задача 4: Построение тепловой карты продаж по регионам и месяцам

Описание:

Имеются данные о продажах в четырёх регионах за три месяца:

– Регионы: `['North', 'South', 'East', 'West']`

– Месяцы: `['January', 'February', 'March']`

– Продажи (матрица):

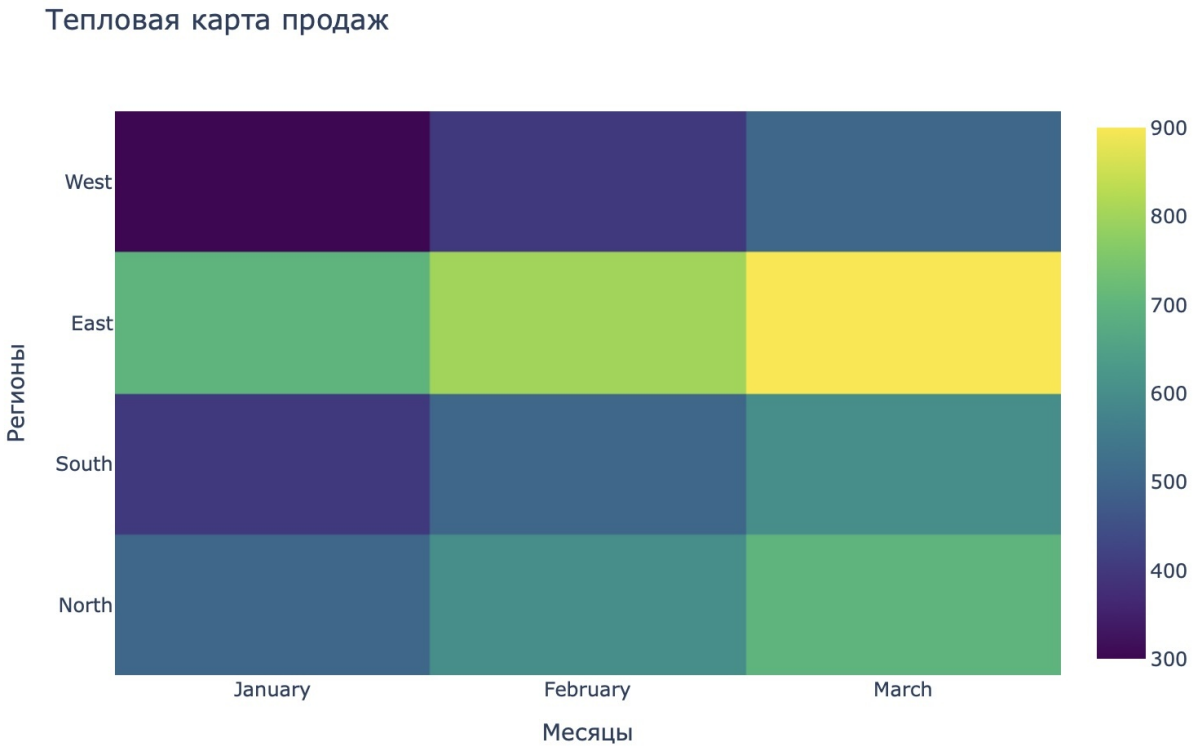
```
[[500, 600, 700],
[400, 500, 600],
[700, 800, 900],
[300, 400, 500]]
```

Постройте тепловую карту, отображающую продажи.

Решение:

```python
import plotly.graph_objects as go
# Данные
regions = ['North', 'South', 'East', 'West']
```

```
months = ['January', 'February', 'March']
sales_matrix = [
[500, 600, 700],
[400, 500, 600],
[700, 800, 900],
[300, 400, 500]
]
# Построение тепловой карты
fig = go.Figure(data=go.Heatmap(
z=sales_matrix,
x=months,
y=regions,
colorscale='Viridis' # Цветовая схема
))
# Настройка графика
fig.update_layout(
title='Тепловая карта продаж',
xaxis_title='Месяцы',
yaxis_title='Регионы'
)
# Показ графика
fig.show()
```



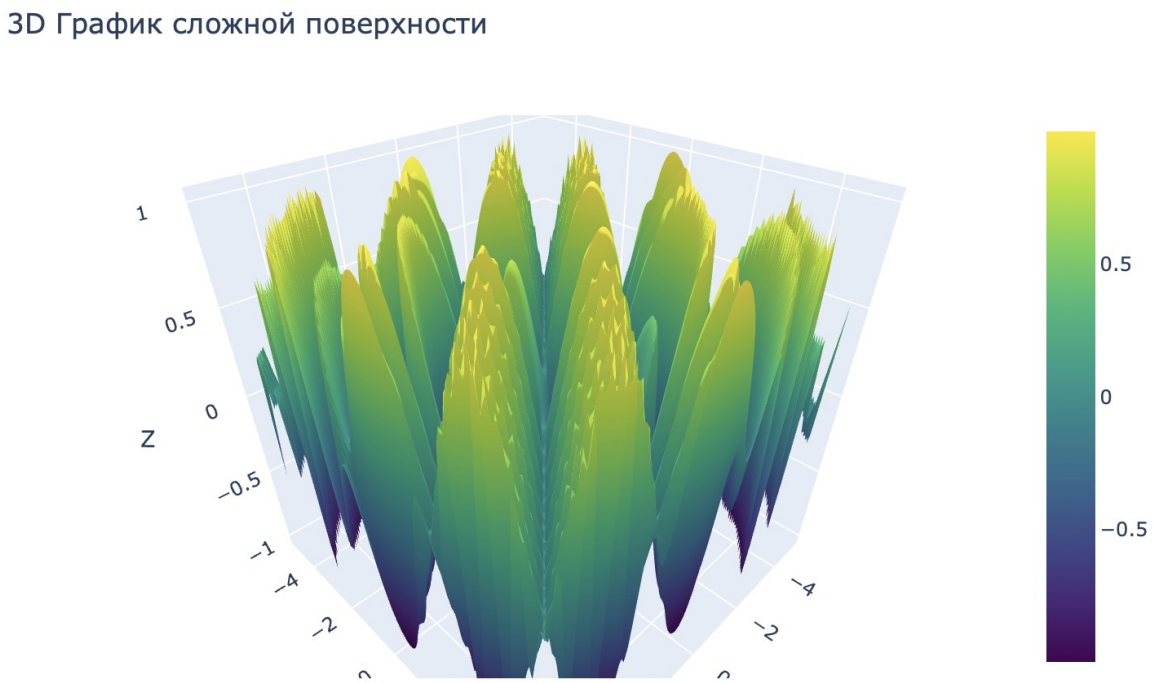
Задача 5: Построение 3D-графика поверхности функции

Описание: Построить 3D-график для функции (z = cos(x^2 + y^2) cdot *sin(x – y)) на диапазоне (x) и (y) от (-5) до (5) с использованием более высокой сетки и с улучшенной цветовой гаммой.

Решение:

```python
import numpy as np
import plotly.graph_objects as go
# Данные
x = np.linspace(-5, 5, 100) # Увеличение разрешения
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.cos(X**2 + Y**2) * np.sin(X – Y) # Сложная функция
# Построение 3D-графика
fig = go.Figure(data=[go.Surface(z=Z, x=X, y=Y, colorscale='Viridis')])
# Изменение цветовой гаммы
# Настройка графика
fig.update_layout(
title='3D График сложной поверхности',
```

```
scene=dict(
xaxis_title='X',
yaxis_title='Y',
zaxis_title='Z'
),
scene_camera=dict(
eye=dict(x=1.5, y=1.5, z=1.5) # Изменение угла обзора
)
)
# Показ графика
fig.show()
```



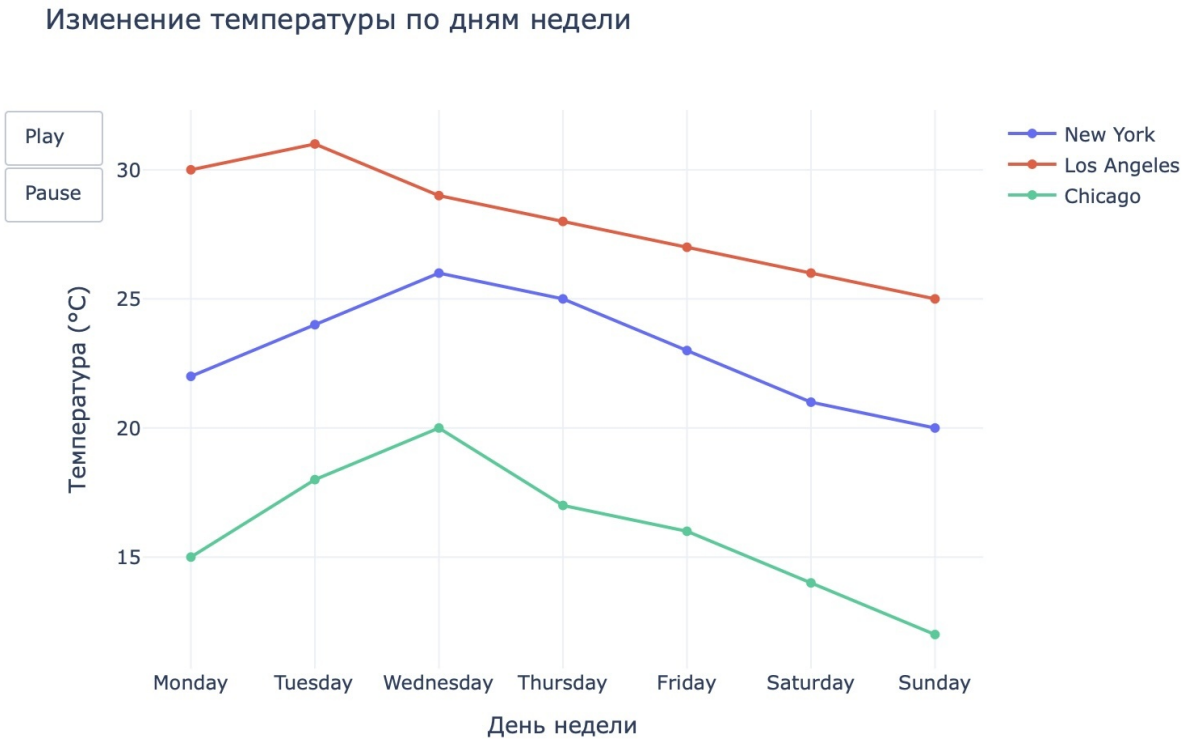
Задача 6: Анимация изменения температуры по дням недели

Описание:

Имеется информация о температуре за каждый день недели для нескольких городов:

– Дни: `['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']`
– Города: `['New York', 'Los Angeles', 'Chicago']`
– Температуры (матрица):
```
New York: [22, 24, 26, 25, 23, 21, 20]
Los Angeles: [30, 31, 29, 28, 27, 26, 25]
Chicago: [15, 18, 20, 17, 16, 14, 12]
```
Создайте анимацию, показывающую изменение температуры для каждого города.
Решение:
```python
import plotly.graph_objects as go
# Данные
days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
cities = ['New York', 'Los Angeles', 'Chicago']
temperatures = {
'New York': [22, 24, 26, 25, 23, 21, 20],
'Los Angeles': [30, 31, 29, 28, 27, 26, 25],
'Chicago': [15, 18, 20, 17, 16, 14, 12]
}
# Создание анимации
fig = go.Figure()
for city in cities:
fig.add_trace(go.Scatter(
x=days,
y=temperatures[city],
mode='lines+markers',
name=city
))
# Настройка анимации
frames = [
go.Frame(
data=[
go.Scatter(
```

```
    x=days[:i],
    y=temperatures[city][:i],
    mode='lines+markers',
    name=city
    )
    for city in cities
    ]
    )
    for i in range(1, len(days) + 1)
    ]
    fig.update(frames=frames)
    # Настройка кнопок
    fig.update_layout(
    updatemenus=[
    dict(
    type='buttons',
    showactive=False,
    buttons=[
    dict(label='Play', method='animate', args=[None, {'frame': {'duration':
500, 'redraw': True}}]),
    dict(label='Pause', method='animate', args=[[None], {'frame': {'duration':
0, 'redraw': False}}])
    ]
    )
    ]
    )
    # Оформление графика
    fig.update_layout(
    title='Изменение температуры по дням недели',
    xaxis_title='День недели',
    yaxis_title='Температура (°C)',
    template='plotly_white'
    )
    fig.show()
    ```
```
```



Задача 7: Трёхмерная анимация COVID-19

Описание:

Используйте вымышленные данные о росте случаев COVID-19 в трёх странах (`USA`, `India`, `Brazil`) за шесть месяцев:

– Месяцы: `['January', 'February', 'March', 'April', 'May', 'June']`

– Число случаев (матрица):

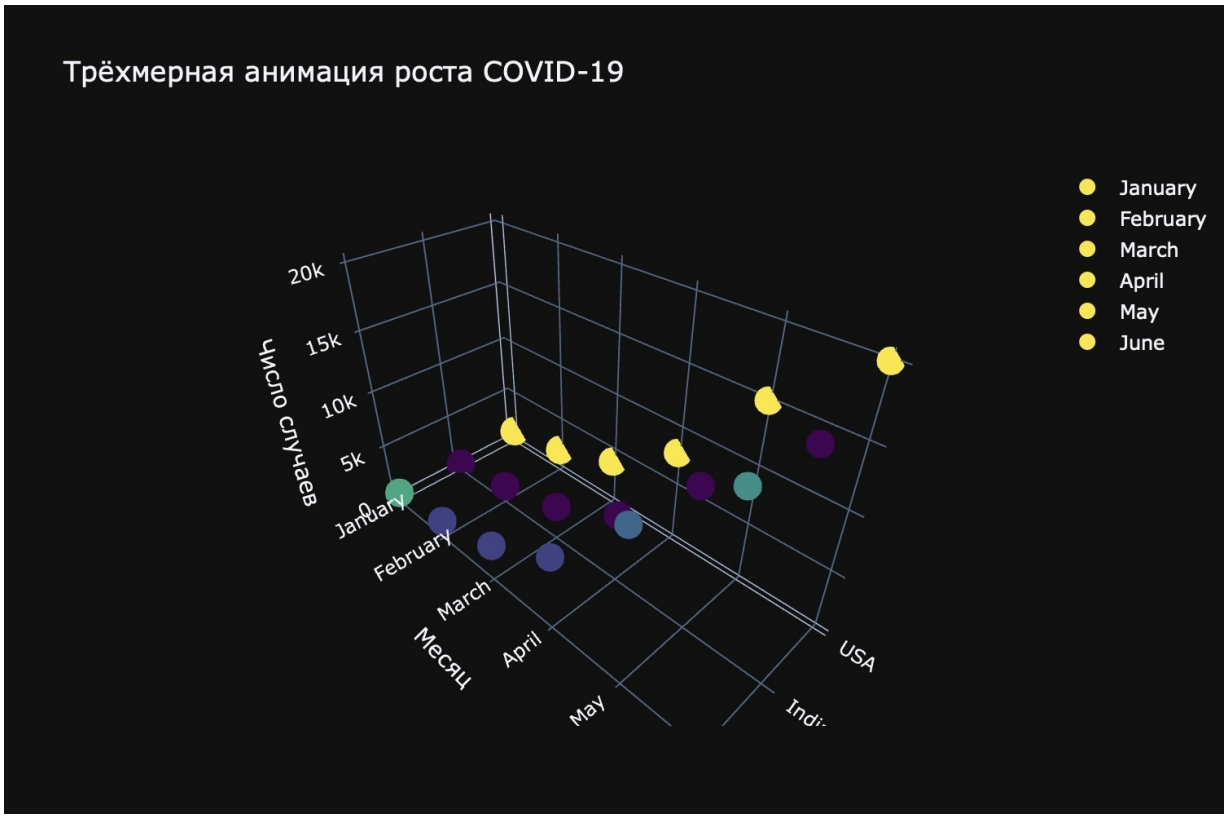
```
USA: [1000, 2000, 4000, 8000, 15000, 20000]
India: [500, 1500, 3000, 6000, 12000, 18000]
Brazil: [800, 1600, 3200, 6400, 13000, 19000]
```

Создайте трёхмерную анимацию, показывающую рост числа случаев по месяцам.

Решение:

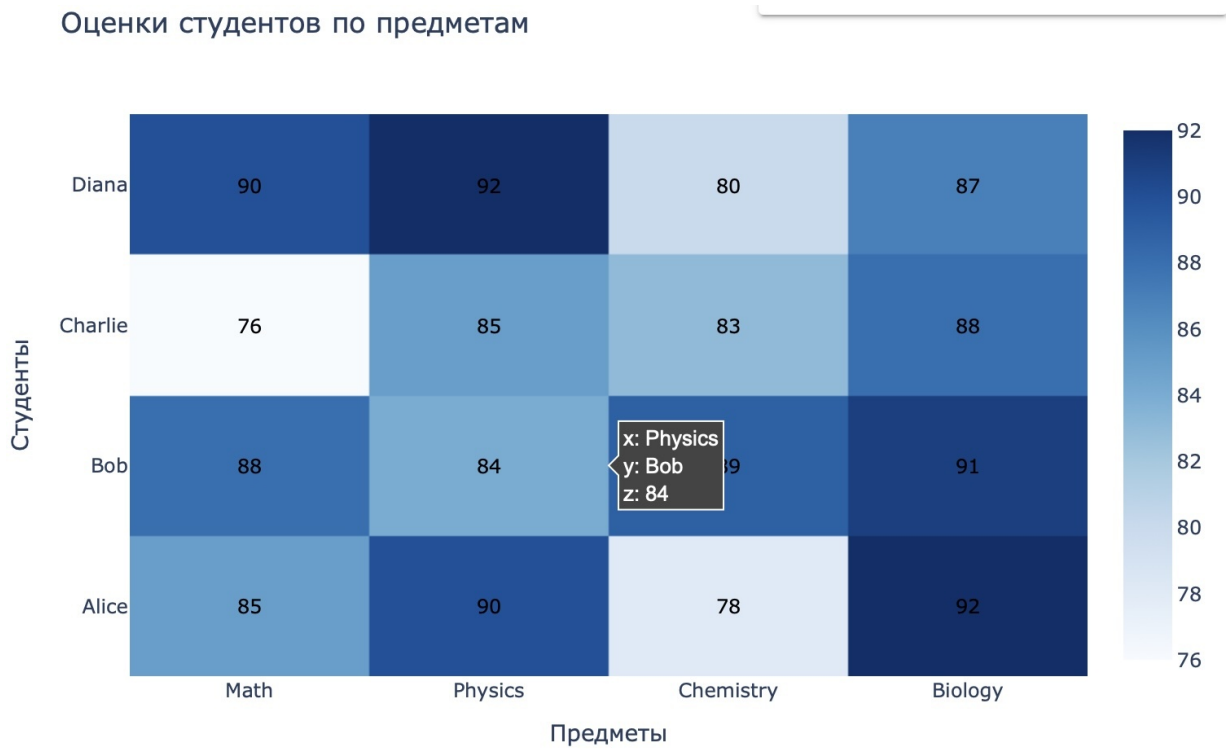
```python
import plotly.graph_objects as go
```

```
    # Данные
    months = ['January', 'February', 'March', 'April', 'May', 'June']
    countries = ['USA', 'India', 'Brazil']
    cases = {
    'USA': [1000, 2000, 4000, 8000, 15000, 20000],
    'India': [500, 1500, 3000, 6000, 12000, 18000],
    'Brazil': [800, 1600, 3200, 6400, 13000, 19000]
    }
    # Построение графика
    fig = go.Figure()
    for month, idx in zip(months, range(len(months))):
    fig.add_trace(go.Scatter3d(
    x=countries,
    y=[month] * len(countries),
    z=[cases[country][idx] for country in countries],
    mode='markers',
    marker=dict(size=10, color=[cases[country][idx] for country in countries], colorscale='Viridis'),
    name=month
    ))
    # Оформление графика
    fig.update_layout(
    title='Трёхмерная анимация роста COVID-19',
    scene=dict(
    xaxis_title='Страна',
    yaxis_title='Месяц',
    zaxis_title='Число случаев'
    ),
    template='plotly_dark'
    )
    fig.show()
    ```
```
```



Задача 8: Тепловая карта с аннотациями
Описание:
Имеется таблица оценки студентов по предметам:
– Студенты: `['Alice', 'Bob', 'Charlie', 'Diana']`
– Предметы: `['Math', 'Physics', 'Chemistry', 'Biology']`
– Оценки (матрица):
```
[[85, 90, 78, 92],
[88, 84, 89, 91],
[76, 85, 83, 88],
[90, 92, 80, 87]]
```
Постройте тепловую карту, добавив аннотации с оценками.
Решение:
```python
import plotly.graph_objects as go
# Данные
students = ['Alice', 'Bob', 'Charlie', 'Diana']
```

```
subjects = ['Math', 'Physics', 'Chemistry', 'Biology']
grades = [
[85, 90, 78, 92],
[88, 84, 89, 91],
[76, 85, 83, 88],
[90, 92, 80, 87]
]
# Построение тепловой карты
fig = go.Figure(data=go.Heatmap(
z=grades,
x=subjects,
y=students,
colorscale='Blues',
showscale=True
))
# Добавление аннотаций
for i, row in enumerate(grades):
for j, val in enumerate(row):
fig.add_annotation(
x=subjects[j],
y=students[i],
text=str(val),
showarrow=False,
font=dict(color='black')
)
# Настройка графика
fig.update_layout(
title='Оценки студентов по предметам',
xaxis_title='Предметы',
yaxis_title='Студенты',
template='plotly_white'
)
fig.show()
```



Эти задачи помогут освоить основные аспекты работы с Plotly: создание линейных, столбчатых, круговых графиков, тепловых карт и трёхмерных поверхностей. В процессе практики вы также научитесь настраивать интерактивные элементы, такие как ползунки и подписывать оси для улучшения визуализации данных.

2.2 Построение динамических дашбордов с Dash

Создание динамических дашбордов стало неотъемлемой частью визуализации данных, позволяя пользователям взаимодействовать с информацией в реальном времени. Dash – это мощный фреймворк, основанный на Python, который позволяет легко создавать такие интерфейсы. В этой главе мы подробно рассмотрим процесс разработки динамического дашборда, начиная с установки и заканчивая созданием интерактивных элементов.

Установка и настройка окружения

Для начала необходимо установить Dash и связанные библиотеки. Основные компоненты включают сам Dash, а также библиотеки для работы с данными, например, pandas и numpy. Установка осуществляется с помощью команды:

```bash
pip install dash pandas numpy
```

После установки можно создать базовый скелет приложения. Этот скелет включает файл `app.py`, который будет служить точкой входа в проект.

Структура приложения Dash

Приложение Dash состоит из двух ключевых частей:

1. layout: определяет, как будет выглядеть интерфейс.

2. callbacks: отвечает за логику взаимодействия и обработку событий.

Рассмотрим пример простого приложения. Представьте, что вы анализируете данные продаж и хотите построить график, который обновляется в зависимости от выбранного диапазона дат.

Пример: дашборд для анализа продаж

Создадим дашборд, который включает выпадающий список для выбора диапазона дат и отображает график продаж.

```python
import dash
from dash import dcc, html
from dash.dependencies import Input, Output
import pandas as pd
import plotly.express as px
# Инициализация приложения
app = dash.Dash(__name__)
# Пример данных
data = {
"date": pd.date_range(start="2023-01-01", periods=100),
"sales": [x * 10 for x in range(100)],
}
df = pd.DataFrame(data)
# Макет приложения
```

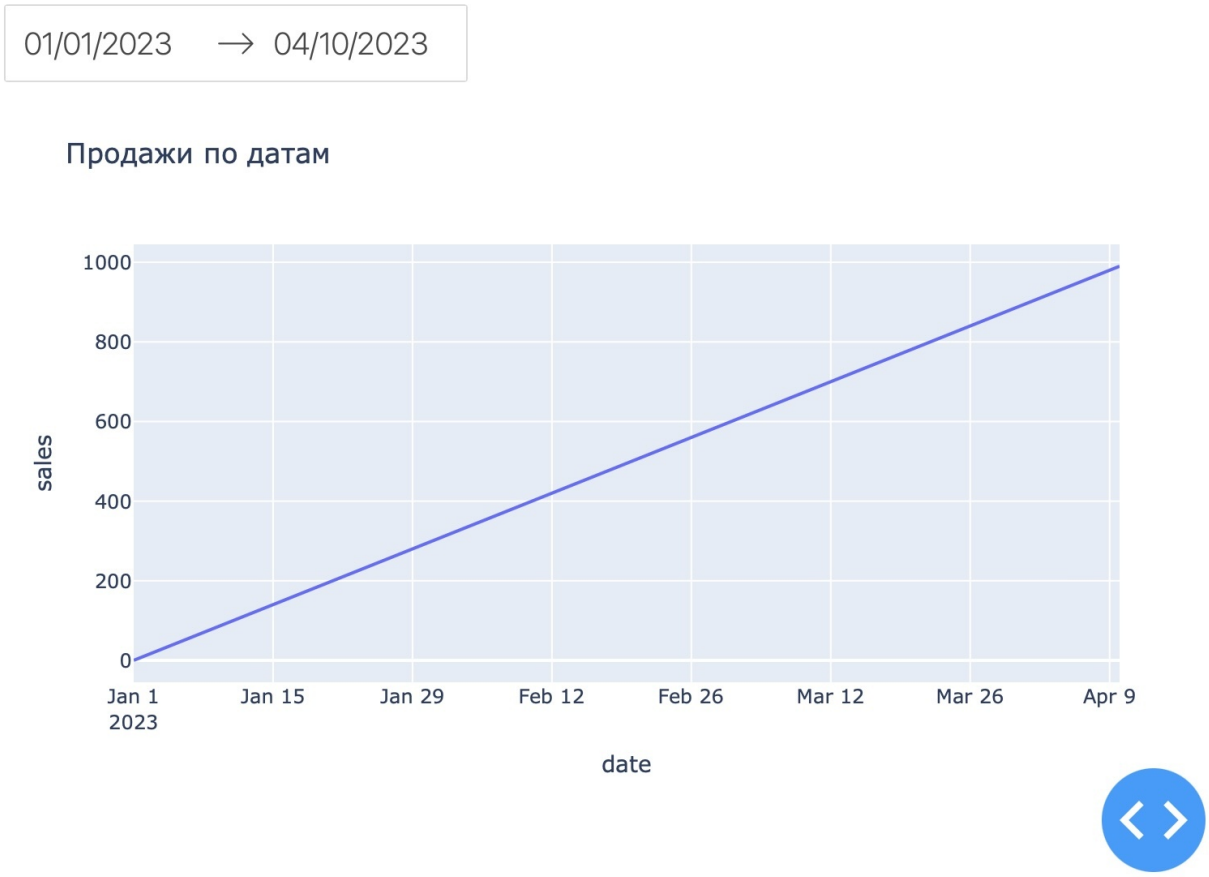
```
app.layout = html.Div([
html.H1("Анализ продаж"),
dcc.DatePickerRange(
id="date-picker",
start_date=df["date"].min(),
end_date=df["date"].max(),
),
dcc.Graph(id="sales-graph")
])
# Callback для обновления графика
@app.callback(
Output("sales-graph", "figure"),
[Input("date-picker", "start_date"),
Input("date-picker", "end_date")]
)
def update_graph(start_date, end_date):
filtered_df = df[(df["date"] >= start_date) & (df["date"] <= end_date)]
fig = px.line(filtered_df, x="date", y="sales", title="Продажи по датам")
return fig
# Запуск приложения
if __name__ == "__main__":
app.run_server(debug=True)
```

Объяснение кода

1. Инициализация: Используется `dash.Dash` для создания приложения.

2. Макет: Компоненты интерфейса добавлены в `app.layout`. Мы включили заголовок, календарь для выбора дат и график.

3. Callback: Это сердце приложения. Функция `update_graph` обновляет график на основе выбранного диапазона дат. Она принимает значения из компонента `dcc.DatePickerRange` и возвращает обновленную фигуру для графика.



Как работает приложение

Когда пользователь выбирает новый диапазон дат в календаре, Dash автоматически вызывает функцию `update_graph`. Эта функция фильтрует данные и создает новый график с помощью библиотеки Plotly. Итоговый график отображается в реальном времени, обеспечивая пользователю мгновенную обратную связь.

Возможности для расширения

С помощью Dash можно добавлять и другие элементы, такие как выпадающие списки, ползунки или текстовые поля. Например, вы можете создать фильтры по регионам или категориям товаров, а также внедрить таблицы с дополнительной информацией.

Dash – это инструмент для создания интерактивных дашбордов, позволяющий быстро и эффективно визуализировать данные. Благодаря своим возможностям, он становится незаменимым для

анализа больших объемов информации.

Задачи для практики

Пример 1: Динамический дашборд для анализа продаж и прогноза с использованием моделей машинного обучения

В этом примере мы создадим дашборд для анализа временных рядов. Он включает следующие функции:

1. Визуализация исторических данных по продажам.
2. Прогнозирование будущих значений с использованием модели машинного обучения.
3. Выбор временных диапазонов для анализа и настройки параметров модели.

Шаг 1: Установка необходимых библиотек

Перед началом необходимо установить дополнительные библиотеки для работы с машинным обучением:

```bash
pip install scikit-learn plotly
```

Шаг 2: Подготовка данных

В этом примере мы используем синтетические данные о продажах, генерируемые случайным образом. Они включают тренд, сезонность и шум, чтобы максимально приблизить данные к реальным.

Код решения

```python
import dash
from dash import dcc, html
from dash.dependencies import Input, Output
import pandas as pd
import numpy as np
import plotly.graph_objects as go
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
# Генерация данных
```

```
    np.random.seed(42)
    date_range = pd.date_range(start="2020-01-01", periods=365)
    sales = 100 + 0.5 * np.arange(365) + 10 * np.sin(2 * np.pi *
np.arange(365) / 30) + np.random.normal(0, 5, 365)
    df = pd.DataFrame({"date": date_range, "sales": sales})
    # Инициализация приложения Dash
    app = dash.Dash(__name__)
    # Макет приложения
    app.layout = html.Div([
    html.H1("Дашборд анализа продаж и прогнозирования"),
    html.Div([
    html.Label("Выберите диапазон дат:"),
    dcc.DatePickerRange(
    id="date-picker",
    start_date=df["date"].min(),
    end_date=df["date"].max()
    ),
    ], style={"margin-bottom": "20px"}),
    html.Div([
    html.Label("Горизонт прогноза (дней):"),
    dcc.Input(
    id="forecast-horizon",
    type="number",
    value=30,
    min=1,
    step=1,
    style={"width": "100px"}
    ),
    ], style={"margin-bottom": "20px"}),
    dcc.Graph(id="sales-graph"),
    ])
    # Callback для обновления графика
    @app.callback(
    Output("sales-graph", "figure"),
    [Input("date-picker", "start_date"),
    Input("date-picker", "end_date"),
    Input("forecast-horizon", "value")]
```

```
)
def update_graph(start_date, end_date, forecast_horizon):
# Фильтрация данных по диапазону
filtered_df = df[(df["date"] >= start_date) & (df["date"] <= end_date)]
# Подготовка данных для модели
X = np.arange(len(filtered_df)).reshape(-1, 1)
y = filtered_df["sales"].values
model = LinearRegression()
model.fit(X, y)
# Прогнозирование
future_X = np.arange(len(filtered_df) + forecast_horizon).reshape(-1, 1)
future_sales = model.predict(future_X)
# Построение графика
fig = go.Figure()
fig.add_trace(go.Scatter(x=filtered_df["date"], y=filtered_df["sales"], mode="lines", name="Фактические продажи"))
future_dates = pd.date_range(start=filtered_df["date"].iloc[-1], periods=forecast_horizon + 1, freq="D")[1:]
fig.add_trace(go.Scatter(x=future_dates, y=future_sales[-forecast_horizon:], mode="lines", name="Прогноз"))
fig.update_layout(title="Исторические данные и прогноз продаж",
xaxis_title="Дата",
yaxis_title="Продажи",
template="plotly_white")
return fig
# Запуск приложения
if __name__ == "__main__":
app.run_server(debug=True)
```

Объяснение кода

1. Подготовка данных: Мы создаем временной ряд с элементами тренда, сезонности и шума.

2. Фильтрация данных: Данные фильтруются по выбранному диапазону дат с помощью компонента `dcc.DatePickerRange`.

3. Модель прогнозирования: Для прогноза используется простая линейная регрессия. Мы создаем массив входных данных (`X`) как временной индекс, обучаем модель и прогнозируем будущие значения.

4. Визуализация: Используется `plotly.graph_objects` для отображения двух линий: фактических данных и прогноза.

Как работает приложение

– Пользователь выбирает диапазон дат и задает горизонт прогноза (количество дней).

– На основе выбранного диапазона обучается модель линейной регрессии.

– Прогноз строится на заданное количество дней вперед.

– Обновленный график отображается на дашборде.

Возможности для расширения

1. Добавление методов прогнозирования: Вместо линейной регрессии можно использовать сложные модели, такие как ARIMA, Prophet или нейронные сети.

2. Динамическая фильтрация по регионам и категориям: Расширение интерфейса для работы с несколькими наборами данных.

3. Показатели точности модели: Вывод метрик (MAE, RMSE) для оценки качества прогноза.

Пример вывода

Когда пользователь запускает приложение, он видит интерактивный дашборд. После выбора диапазона дат и ввода горизонта прогноза на графике отображаются исторические данные (линия синего цвета) и прогноз (линия оранжевого цвета). График обновляется мгновенно, предоставляя пользователю удобный инструмент для анализа.

Пример 2: Красочный дашборд для анализа финансовых показателей с использованием графиков и метрик

Этот пример демонстрирует создание дашборда, который включает:

1. Динамическую визуализацию ключевых финансовых метрик.

2. Гибкие фильтры для выбора периода и типа данных.

3. Красочное оформление с использо/ванием шаблонов Plotly.

Функционал

– Выбор временного диапазона.

– Отображение трех финансовых метрик: доходы, расходы и прибыль.

– Сравнение текущего периода с предыдущим.

Код решения

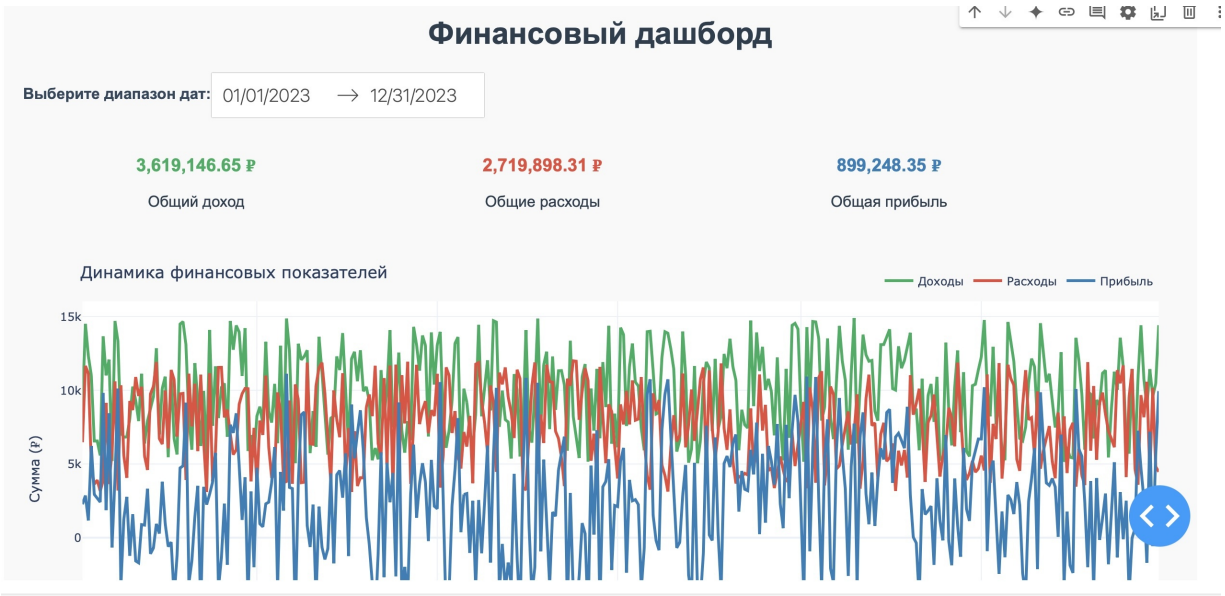
```python
import dash
from dash import dcc, html
from dash.dependencies import Input, Output
import pandas as pd
import numpy as np
import plotly.graph_objects as go
# Генерация данных
np.random.seed(42)
dates = pd.date_range(start="2023-01-01", periods=365)
revenues = np.random.uniform(5000, 15000, len(dates))
expenses = np.random.uniform(3000, 12000, len(dates))
profits = revenues – expenses
```

```
    df = pd.DataFrame({
    "date": dates,
    "revenue": revenues,
    "expense": expenses,
    "profit": profits
    })
    # Инициализация приложения
    app = dash.Dash(__name__)
    # Макет приложения
    app.layout = html.Div(
    style={"font-family": "Arial, sans-serif", "margin": "20px", "background-color": "#f9f9f9", "padding": "20px"},
    children=[
    html.H1(
    "Финансовый дашборд",
    style={"text-align": "center", "color": "#2C3E50"}
    ),
    html.Div([
    html.Label("Выберите диапазон дат:", style={"font-weight": "bold", "color": "#34495E"}),
    dcc.DatePickerRange(
    id="date-picker",
    start_date=df["date"].min(),
    end_date=df["date"].max(),
    style={"margin-bottom": "20px"}
    )
    ]),
    html.Div([
    html.Div([
    html.H3(id="total-revenue", style={"color": "#27AE60", "margin-bottom": "5px"}),
    html.P("Общий доход", style={"color": "#2C3E50"})
    ], style={"text-align": "center", "width": "30%", "display": "inline-block"}),
    html.Div([
    html.H3(id="total-expense", style={"color": "#E74C3C", "margin-bottom": "5px"}),
```

```
    html.P("Общие расходы", style={"color": "#2C3E50"})
    ], style={"text-align": "center", "width": "30%", "display": "inline-block"}),
    html.Div([
    html.H3(id="total-profit", style={"color": "#2980B9", "margin-bottom": "5px"}),
    html.P("Общая прибыль", style={"color": "#2C3E50"})
    ], style={"text-align": "center", "width": "30%", "display": "inline-block"}),
    ], style={"margin-bottom": "30px"}),
    dcc.Graph(id="financial-graph")
    ]
    )
    # Callback для обновления графика и метрик
    @app.callback(
    [Output("financial-graph", "figure"),
    Output("total-revenue", "children"),
    Output("total-expense", "children"),
    Output("total-profit", "children")],
    [Input("date-picker", "start_date"),
    Input("date-picker", "end_date")]
    )
    def update_dashboard(start_date, end_date):
    # Фильтрация данных
    filtered_df = df[(df["date"] >= start_date) & (df["date"] <= end_date)]
    # Вычисление общих показателей
    total_revenue = filtered_df["revenue"].sum()
    total_expense = filtered_df["expense"].sum()
    total_profit = filtered_df["profit"].sum()
    # Построение графика
    fig = go.Figure()
    fig.add_trace(go.Scatter(
    x=filtered_df["date"],
    y=filtered_df["revenue"],
    mode="lines",
    name="Доходы",
    line=dict(color="#27AE60", width=3)
```

```
    ))
    fig.add_trace(go.Scatter(
    x=filtered_df["date"],
    y=filtered_df["expense"],
    mode="lines",
    name="Расходы",
    line=dict(color="#E74C3C", width=3)
    ))
    fig.add_trace(go.Scatter(
    x=filtered_df["date"],
    y=filtered_df["profit"],
    mode="lines",
    name="Прибыль",
    line=dict(color="#2980B9", width=3)
    ))
    fig.update_layout(
    title="Динамика финансовых показателей",
    xaxis_title="Дата",
    yaxis_title="Сумма (₽)",
    template="plotly_white",
    legend=dict(orientation="h", yanchor="bottom", y=1.02,
xanchor="right", x=1),
    margin=dict(l=20, r=20, t=40, b=20),
    paper_bgcolor="#f9f9f9"
    )
    return fig, f"{total_revenue:,.2f} ₽", f"{total_expense:,.2f} ₽", f"
{total_profit:,.2f} ₽"
    # Запуск приложения
    if __name__ == "__main__":
    app.run_server(debug=True)
```

```




Что делает этот дашборд особенным?

1. Красочное оформление:

– Используются фирменные цвета (#27AE60 для доходов, #E74C3C для расходов, #2980B9 для прибыли).

– Удобное расположение метрик с фокусом на визуальной доступности.

2. Интерактивность:

– Пользователь может выбрать диапазон дат, и данные мгновенно обновляются.

– График отображает три ключевых показателя с соответствующими цветами.

3. Простота и стиль:

– Белый фон и шаблон Plotly White создают современный и профессиональный вид.

– Четко выделенные метрики позволяют быстро оценить ситуацию.

Как это выглядит на практике?

– График: На графике отображаются три линии, соответствующие доходам, расходам и прибыли. Цвета позволяют легко различать их.

– Метрики: Общие показатели доходов, расходов и прибыли отображаются над графиком в виде карточек.

Возможности для расширения

1. Детализация показателей: Добавьте сравнение текущего периода с аналогичным периодом прошлого года.
```

2. Фильтры: Включите возможность фильтрации данных по регионам, продуктам или другим категориям.

3. Анимация: Используйте анимацию для показа динамики данных.

Этот дашборд демонстрирует, как можно совместить функциональность и стиль, создавая инструмент для анализа, который одновременно удобен и красив.

Пример 3: Дашборд для анализа рынка недвижимости с обширной визуализацией

Этот дашборд ориентирован на анализ рынка недвижимости. Он позволяет:

1. Отображать динамику цен на жилье по регионам.

2. Сравнивать цены квартир с разным количеством комнат.

3. Проводить географический анализ, отображая цены на карте.

4. Включать фильтры для выбора региона, диапазона цен и типа недвижимости.

Функционал

– Географический анализ: Отображение цен на карте.

– Динамика цен: График изменения средней цены по времени.

– Сравнение типов недвижимости: Диаграмма с разбивкой по количеству комнат.

– Интерактивные фильтры: Фильтрация по региону, диапазону цен и периоду.

Код решения

```python
import dash
from dash import dcc, html
from dash.dependencies import Input, Output
import pandas as pd
import numpy as np
import plotly.express as px
# Генерация данных
np.random.seed(42)
regions = ["Москва", "Санкт-Петербург", "Новосибирск", "Екатеринбург", "Казань"]
dates = pd.date_range(start="2020-01-01", periods=36, freq="M")
```

```
    data = []
    for region in regions:
    for date in dates:
    for rooms in [1, 2, 3]:
    price = np.random.uniform(2, 10) * rooms * 1e6
    latitude = np.random.uniform(55, 60) if region == "Москва" else np.random.uniform(50, 58)
    longitude = np.random.uniform(37, 41) if region == "Москва" else np.random.uniform(30, 40)
    data.append({
    "region": region,
    "date": date,
    "rooms": rooms,
    "price": price,
    "latitude": latitude,
    "longitude": longitude
    })
    df = pd.DataFrame(data)
    # Инициализация приложения Dash
    app = dash.Dash(__name__)
    # Макет приложения
    app.layout = html.Div(style={"font-family": "Arial, sans-serif", "margin": "20px", "background-color": "#f9f9f9", "padding": "20px"}, children=[
    html.H1("Дашборд анализа рынка недвижимости", style={"text-align": "center", "color": "#2C3E50"}),
    html.Div([
    html.Div([
    html.Label("Выберите регион:", style={"font-weight": "bold", "color": "#34495E"}),
    dcc.Dropdown(
    id="region-filter",
    options=[{"label": region, "value": region} for region in regions],
    value="Москва",
    clearable=False,
    style={"margin-bottom": "20px"}
    )
```

```
    ], style={"width": "30%", "display": "inline-block", "vertical-align": "top"}),
    html.Div([
    html.Label("Диапазон цен (млн ₽):", style={"font-weight": "bold", "color": "#34495E"}),
    dcc.RangeSlider(
    id="price-filter",
    min=2,
    max=30,
    step=1,
    marks={i: f"{i}" for i in range(2, 31, 5)},
    value=[5, 20],
    tooltip={"always_visible": False, "placement": "bottom"}
    )
    ], style={"width": "60%", "display": "inline-block", "padding-left": "20px"})
    ]),
    html.Div([
    dcc.Graph(id="price-dynamics-graph", style={"margin-bottom": "30px"})
    ]),
    html.Div([
    dcc.Graph(id="price-distribution-graph", style={"width": "48%", "display": "inline-block", "vertical-align": "top"}),
    dcc.Graph(id="price-map", style={"width": "48%", "display": "inline-block", "vertical-align": "top"})
    ])
    ])
    # Callback для обновления визуализаций
    @app.callback(
    [Output("price-dynamics-graph", "figure"),
    Output("price-distribution-graph", "figure"),
    Output("price-map", "figure")],
    [Input("region-filter", "value"),
    Input("price-filter", "value")]
    )
    def update_dashboard(region, price_range):
```

```
    # Фильтрация данных
    filtered_df = df[(df["region"] == region) & (df["price"] >= price_range[0] * 1e6) & (df["price"] <= price_range[1] * 1e6)]
    # Динамика цен
    dynamics_df = filtered_df.groupby(["date", "rooms"])["price"].mean().reset_index()
    dynamics_fig = px.line(
    dynamics_df,
    x="date",
    y="price",
    color="rooms",
    labels={"price": "Средняя цена (₽)", "date": "Дата", "rooms": "Комнат"},
    title="Динамика цен на жилье",
    template="plotly_white"
    )
    dynamics_fig.update_traces(mode="lines+markers")
    # Распределение цен
    distribution_fig = px.histogram(
    filtered_df,
    x="price",
    color="rooms",
    nbins=20,
    labels={"price": "Цена (₽)", "rooms": "Комнат"},
    title="Распределение цен по количеству комнат",
    template="plotly_white"
    )
    # Географическая карта
    map_fig = px.scatter_mapbox(
    filtered_df,
    lat="latitude",
    lon="longitude",
    color="price",
    size="price",
    color_continuous_scale=px.colors.sequential.Viridis,
    mapbox_style="carto-positron",
    hover_data={"price": ":,.0f", "rooms": True},
```

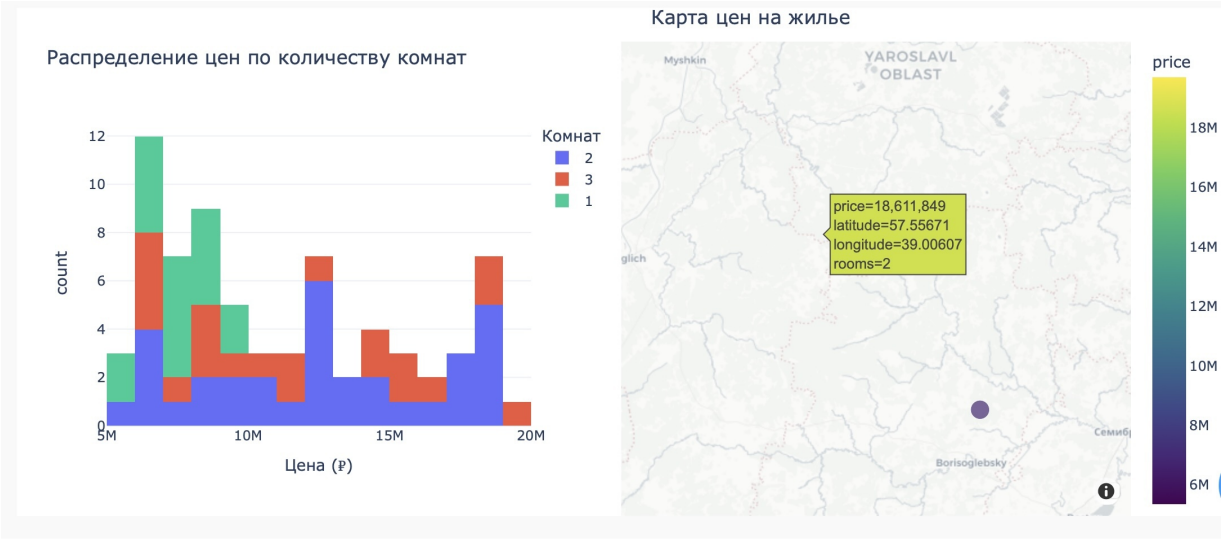
```
title="Карта цен на жилье"
)
map_fig.update_layout(margin={"r":0, "t":30, "l":0, "b":0})
return dynamics_fig, distribution_fig, map_fig
# Запуск приложения
if __name__ == "__main__":
app.run_server(debug=True)
```

Что делает этот дашборд уникальным?
1. Географическая карта:
– Карта с отображением точек недвижимости по регионам.
– Цветовая шкала отражает уровень цен.

2. Динамика цен: Линии на графике показывают, как изменялись средние цены по количеству комнат.

3. Распределение цен: Гистограмма с распределением цен в зависимости от количества комнат.

4. Интерактивные фильтры: Пользователь может выбрать регион и диапазон цен, что мгновенно отражается на графиках.

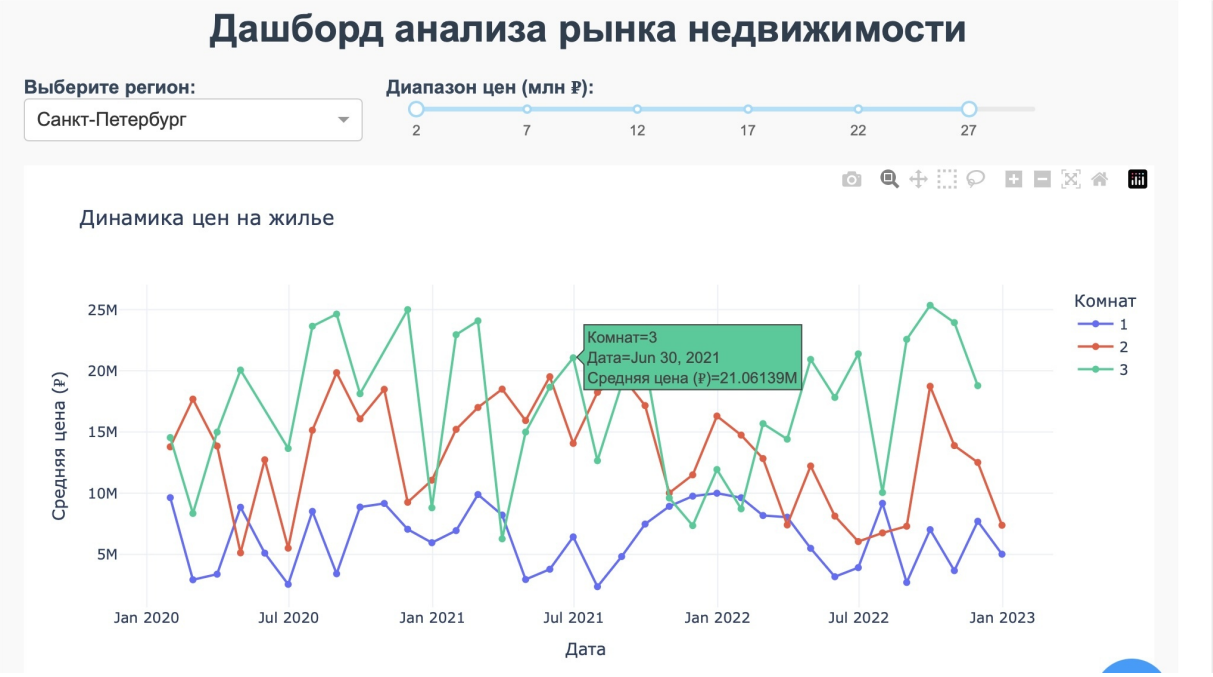


Как это выглядит на практике?

– График динамики цен: Линии разного цвета отображают изменения средней цены по типам недвижимости (количество комнат).

– Гистограмма распределения цен: Сравнение количества квартир в разных ценовых диапазонах.

– Карта цен: Точки на карте показывают объекты недвижимости, а их цвет и размер зависят от цены.



Возможности для расширения

1. Тип недвижимости: Добавить фильтр для выбора типа недвижимости (например, квартира, дом, офис).
2. Более детализированная карта: Интеграция с картами Google или Yandex для детального отображения.
3. Анализ спроса: Добавить данные о количестве просмотров или сделок для каждого объекта.

Этот дашборд объединяет множество визуализаций, делая анализ данных о недвижимости быстрым и наглядным.

2.3 Визуализация геоданных с использованием Folium и GeoPandas

Работа с геоданными в Python предоставляет широкие возможности для анализа и визуализации пространственных данных. Одними из самых популярных библиотек для этой задачи являются GeoPandas, которая позволяет работать с геометрическими данными в табличной структуре, и Folium, которая используется для создания

интерактивных карт. В данной главе мы рассмотрим, как использовать эти инструменты для визуализации геоданных, а также построим интерактивную карту, отображающую различные объекты.

Основные концепции работы с GeoPandas

GeoPandas расширяет функционал стандартной библиотеки pandas, добавляя поддержку работы с геометрическими объектами, такими как точки, линии и полигоны. Эти объекты представляют собой данные в форматах, таких как Shapefile или GeoJSON.

Сначала необходимо загрузить данные и преобразовать их в геодатафрейм. Каждый объект геодатафрейма имеет геометрическое представление, хранящееся в специальном столбце `geometry`.

Пример создания геодатафрейма:

```python
import geopandas as gpd
from shapely.geometry import Point
# Создаем данные
data = {
"name": ["Магазин 1", "Магазин 2", "Магазин 3"],
"latitude": [55.7558, 59.9343, 56.8389],
"longitude": [37.6173, 30.3351, 60.6057]
}
# Преобразуем данные в GeoDataFrame
geometry = [Point(xy) for xy in zip(data["longitude"], data["latitude"])]
gdf = gpd.GeoDataFrame(data, geometry=geometry)
print(gdf)
```

```
        name  latitude  longitude                   geometry
0  Магазин 1   55.7558    37.6173  POINT (37.6173 55.7558)
1  Магазин 2   59.9343    30.3351  POINT (30.3351 59.9343)
2  Магазин 3   56.8389    60.6057  POINT (60.6057 56.8389)
```

На этом этапе мы создали таблицу с географическими координатами объектов, которую можно использовать для дальнейшего анализа и визуализации.

Визуализация данных с использованием Folium

Folium позволяет создавать интерактивные карты с минимальным количеством кода. Эта библиотека построена на основе JavaScript-библиотеки Leaflet.js, но интегрирована с Python.

Основная концепция Folium заключается в добавлении маркеров, слоев и других геообъектов на базовую карту.

Пример создания простой карты:

```python
import folium
# Создаем базовую карту
m = folium.Map(location=[55.7558, 37.6173], zoom_start=10)
# Добавляем маркер
folium.Marker(location=[55.7558, 37.6173], popup="Центр Москвы").add_to(m)
# Сохраняем карту в файл
m.save("simple_map.html")
```

Карта будет сохранена в файл `simple_map.html`, который можно открыть в браузере для просмотра.

Пример: Отображение данных о недвижимости на интерактивной карте

Теперь объединим возможности GeoPandas и Folium для создания карты, отображающей данные о недвижимости. Мы будем использовать GeoJSON-файл, содержащий информацию о районах города, и список объектов недвижимости с координатами.

```python
import geopandas as gpd
import folium
from folium import Choropleth, CircleMarker
# Загружаем данные о районах города
city_districts = gpd.read_file("city_districts.geojson")
# Создаем данные об объектах недвижимости
properties = {
"name": ["Квартира 1", "Квартира 2", "Квартира 3"],
"price": [7500000, 12500000, 5600000],
"latitude": [55.7522, 55.7649, 55.7485],
"longitude": [37.6156, 37.6383, 37.5902]
}
```

```
# Преобразуем данные об объектах недвижимости в GeoDataFrame
geometry = [Point(xy) for xy in zip(properties["longitude"], properties["latitude"])]
properties_gdf = gpd.GeoDataFrame(properties, geometry=geometry)
# Создаем базовую карту
m = folium.Map(location=[55.7558, 37.6173], zoom_start=12)
# Добавляем границы районов города на карту
folium.GeoJson(
city_districts,
name="Границы районов",
style_function=lambda feature: {
"fillColor": "#add8e6",
"color": "#000000",
"weight": 1,
"fillOpacity": 0.5
}
).add_to(m)
# Добавляем объекты недвижимости
for idx, row in properties_gdf.iterrows():
folium.CircleMarker(
location=[row["latitude"], row["longitude"]],
radius=8,
color="blue",
fill=True,
fill_color="blue",
fill_opacity=0.7,
popup=f"{row['name']}\nЦена: {row['price']} ₽"
).add_to(m)
# Добавляем слой управления
folium.LayerControl().add_to(m)
# Сохраняем карту
m.save("real_estate_map.html")
```

В результате будет создана интерактивная карта, где:

– Границы районов города отображаются в виде полигонов с полупрозрачным заливом.

– Объекты недвижимости представлены маркерами с информацией о названии и цене.

Результат

После выполнения кода карта будет сохранена в файл `real_estate_map.html`. Открытие файла в браузере позволит:

– Навигировать по карте с помощью масштабирования и перемещения.

– Кликать на маркеры, чтобы увидеть всплывающую информацию об объектах недвижимости.

Расширение функционала

1. Анализ цен по районам: Используйте данные о ценах на недвижимость для создания тепловой карты (Choropleth).

2. Кластеры маркеров: Если данных об объектах много, добавьте кластеризацию маркеров для удобства отображения.

3. Интерактивные фильтры: Реализуйте фильтрацию объектов по цене или типу недвижимости, используя дополнительные библиотеки, такие как Dash.

Использование GeoPandas и Folium позволяет не только анализировать пространственные данные, но и наглядно их представлять. Это полезно в самых разных областях: от анализа недвижимости и транспорта до экологии и городского планирования.

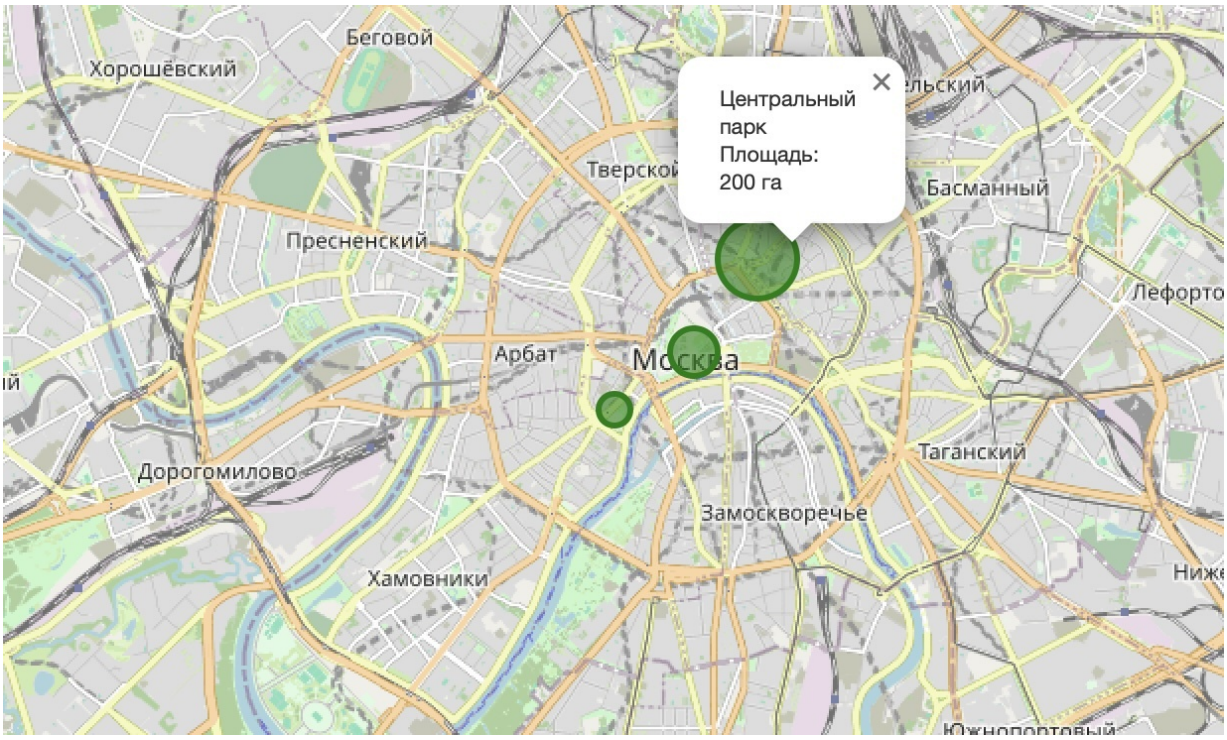
Задачи для практики

Пример задачи: Визуализация парков города

Создадим интерактивную карту с отображением парков в вымышленном городе. Пусть у нас есть координаты парков и информация о их размерах. В данном примере данные создаются вручную.

```python
import folium
from shapely.geometry import Polygon
import geopandas as gpd
# Данные о парках
```

```
parks = {
"name": ["Парк Победы", "Центральный парк", "Сквер Молодежи"],
"latitude": [55.7512, 55.7603, 55.7457],
"longitude": [37.6190, 37.6300, 37.6050],
"area": [120, 200, 80] # Площадь парка в гектарах
}
# Преобразуем данные в GeoDataFrame
geometry = [Point(xy) for xy in zip(parks["longitude"], parks["latitude"])]
parks_gdf = gpd.GeoDataFrame(parks, geometry=geometry)
# Создаем базовую карту
m = folium.Map(location=[55.7558, 37.6173], zoom_start=13)
# Добавляем маркеры для каждого парка
for idx, row in parks_gdf.iterrows():
folium.CircleMarker(
location=[row["latitude"], row["longitude"]],
radius=row["area"] / 10, # Радиус зависит от площади
color="green",
fill=True,
fill_color="green",
fill_opacity=0.6,
popup=f"{row['name']}\nПлощадь: {row['area']} га"
).add_to(m)
# Сохраняем карту в файл
m.save("parks_map.html")
```



В этом примере на карте отображаются парки, радиус каждого маркера пропорционален площади парка. Наведение или клик на маркер выводит всплывающее окно с названием парка и его площадью.

Результат

После выполнения кода карта сохранится в файл `parks_map.html`. Открыв этот файл, можно увидеть:

– Маркеры, представляющие парки.

– Цветовые обозначения, связанные с парками.

– Всплывающие окна с дополнительной информацией.

Расширение примера

1. Добавьте полигоны, чтобы вместо точек отображать реальные границы парков.

2. Реализуйте дополнительный слой с водоемами, которые можно скрывать и показывать с помощью панели управления.

3. Используйте кластеризацию, если количество парков увеличится.

Глава 3. Машинное обучение и искусственный интеллект

3.1 Оптимизация производительности моделей с использованием Cython и Numba

Современные алгоритмы машинного обучения часто сталкиваются с проблемой производительности при работе с большими объемами данных. В этом контексте использование инструментов для ускорения вычислений становится важным аспектом. Cython и Numba – два инструмента, которые позволяют улучшить скорость выполнения Python-кода за счёт компиляции в машинный код. Их применение особенно эффективно для задач, требующих больших вычислительных ресурсов, таких как обучение моделей, обработка данных и численные симуляции.

Cython – это супермножество языка Python, которое позволяет включать статическую типизацию и компилировать код в C, что значительно повышает его производительность.

Код на Cython может быть написан с минимальными изменениями по сравнению с обычным Python-кодом, но при этом достигается значительный прирост скорости. Например, в задачах линейной алгебры, таких как матричное умножение или обращение матриц, статическая типизация переменных существенно ускоряет вычисления.

Рассмотрим пример использования Cython для ускорения вычислений. Предположим, нужно вычислить сумму квадратов элементов массива:

```python
# Обычный Python-код
def sum_of_squares(arr):
result = 0
for x in arr:
result += x ** 2
```

```
return result
```

В Cython можно объявить типы переменных, чтобы достичь оптимизации:

```cython
# Код на Cython
def sum_of_squares(double[:] arr):
cdef double result = 0
cdef Py_ssize_t i
for i in range(arr.shape[0]):
result += arr[i] ** 2
return result
```

Компиляция с помощью Cython создаёт скомпилированный модуль, который выполняется значительно быстрее, чем исходный Python-код.

Numba – это инструмент для JIT-компиляции (Just-In-Time), который автоматически компилирует функции Python в машинный код с использованием LLVM. Одной из ключевых особенностей Numba является простота интеграции: достаточно добавить декоратор `@jit` к функции, чтобы активировать компиляцию. Numba хорошо подходит для задач, связанных с обработкой массивов и числовыми вычислениями, и может работать совместно с библиотекой NumPy.

Рассмотрим пример использования Numba для ускорения той же задачи:

```python
from numba import jit
import numpy as np
@jit
def sum_of_squares(arr):
result = 0
for x in arr:
result += x ** 2
return result
# Использование
arr = np.random.rand(1000000)
result = sum_of_squares(arr)
```

Numba автоматически анализирует типы данных во время выполнения, что избавляет от необходимости явно задавать их, как в Cython. Это делает его особенно удобным для быстрого прототипирования.

Сравнение Cython и Numba

Оба инструмента имеют свои сильные стороны и области применения. Cython предоставляет более точный контроль над кодом, позволяя оптимизировать его на уровне C. Это делает его предпочтительным выбором для сложных проектов, где требуется интеграция с библиотеками на C или C++. Однако написание кода на Cython требует больше времени и усилий, чем на Numba.

Numba, напротив, ориентирован на простоту использования и динамическую компиляцию. Он особенно эффективен для ускорения существующего Python-кода с минимальными изменениями. Однако для очень сложных проектов, где требуется точная настройка, возможности Numba могут быть ограничены.

Практические примеры применения

1. Обучение моделей: Ускорение вычисления градиентов для задач оптимизации. Например, в алгоритмах градиентного спуска использование Cython или Numba может сократить время обучения моделей.

2. Предобработка данных: Быстрая обработка больших массивов данных, включая нормализацию, фильтрацию и агрегирование.

3. Научные вычисления: Решение уравнений и моделирование физических процессов, где производительность критически важна.

Cython и Numba являются важными инструментами для разработчиков и исследователей, работающих в области машинного обучения и анализа данных. Их использование позволяет преодолеть ограничения производительности Python, делая возможным выполнение сложных вычислений за минимальное время.

Задачи для практики

Ниже представлены несколько практических задач с решениями. Эти задачи помогут лучше понять, как использовать инструменты

Cython и Numba для оптимизации кода.

Задача 1: Вычисление факториала числа

Условие: Напишите функцию для вычисления факториала числа (n), используя Cython и Numba. Оптимизируйте производительность по сравнению с обычным Python-кодом.

Решение с использованием Cython:

Создайте файл `factorial.pyx`:

```cython
# factorial.pyx
def factorial(int n):
cdef int i
cdef int result = 1
for i in range(1, n + 1):
result *= i
return result
```

Скомпилируйте файл с помощью `setup.py`:

```python
# setup.py
from setuptools import setup
from Cython.Build import cythonize
setup(
ext_modules=cythonize("factorial.pyx")
)
```

Скомпилируйте модуль:

```bash
python setup.py build_ext –inplace
```

Использование скомпилированной функции:

```python
from factorial import factorial
print(factorial(10)) # Результат: 3628800
```

Решение с использованием Numba:

```python
```

```
from numba import jit
@jit
def factorial(n):
result = 1
for i in range(1, n + 1):
result *= i
return result
print(factorial(10)) # Результат: 3628800
```

Задача 2: Скалярное произведение векторов

Условие: Реализуйте функцию для вычисления скалярного произведения двух векторов (A) и (B). Используйте Cython и Numba для оптимизации.

Решение с использованием Cython:

Создайте файл `dot_product.pyx`:

```cython
# dot_product.pyx
import numpy as np
cimport numpy as cnp
def dot_product(cnp.ndarray[double, ndim=1] A, cnp.ndarray[double, ndim=1] B):
cdef int i
cdef double result = 0
for i in range(A.shape[0]):
result += A[i] * B[i]
return result
```

Компиляция аналогична задаче 1. Использование:

```python
import numpy as np
from dot_product import dot_product
A = np.random.rand(1000000)
B = np.random.rand(1000000)
print(dot_product(A, B)) # Результат: скалярное произведение
```

Решение с использованием Numba:

```python
```

```
from numba import jit
import numpy as np
@jit
def dot_product(A, B):
result = 0
for i in range(len(A)):
result += A[i] * B[i]
return result
A = np.random.rand(1000000)
B = np.random.rand(1000000)
print(dot_product(A, B)) # Результат: скалярное произведение
```

Задача 3: Ускорение сортировки массива

Условие: Реализуйте сортировку массива методом пузырька и оптимизируйте её с помощью Cython и Numba.

Решение с использованием Cython:

Создайте файл `bubble_sort.pyx`:

```cython
# bubble_sort.pyx
cimport cython
def bubble_sort(int[:] arr):
cdef int i, j, n
n = arr.shape[0]
for i in range(n):
for j in range(0, n – i – 1):
if arr[j] > arr[j + 1]:
arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Компиляция и использование аналогичны предыдущим примерам:

```python
import numpy as np
from bubble_sort import bubble_sort
arr = np.array([5, 3, 8, 6, 2, 7, 4, 1], dtype=int)
bubble_sort(arr)
print(arr) # Результат: отсортированный массив
```

Решение с использованием Numba:

```python
from numba import jit
import numpy as np
@jit
def bubble_sort(arr):
n = len(arr)
for i in range(n):
for j in range(0, n – i – 1):
if arr[j] > arr[j + 1]:
arr[j], arr[j + 1] = arr[j + 1], arr[j]
arr = np.array([5, 3, 8, 6, 2, 7, 4, 1], dtype=int)
bubble_sort(arr)
print(arr) # Результат: отсортированный массив
```

Задача 4: Численное интегрирование методом трапеций

Условие: Реализуйте численное интегрирование функции f(x) на отрезке ([a, b]) методом трапеций. Оптимизируйте производительность с помощью Cython и Numba.

Решение с использованием Cython:

Создайте файл `trapezoidal.pyx`:

```cython
# trapezoidal.pyx
cimport cython
def integrate_trapezoidal(double a, double b, int n, double (*f)(double)):
cdef double h = (b – a) / n
cdef double result = 0.5 * (f(a) + f(b))
cdef int i
for i in range(1, n):
result += f(a + i * h)
return result * h
```

Компиляция аналогична. Использование:

```python
from trapezoidal import integrate_trapezoidal
import math
def f(x):
return math.sin(x)
```

```
result = integrate_trapezoidal(0, math.pi, 1000, f)
print(result) # Результат: приближённое значение интеграла
```

Решение с использованием Numba:

```python
from numba import jit
import numpy as np
@jit
def integrate_trapezoidal(a, b, n, f):
h = (b – a) / n
result = 0.5 * (f(a) + f(b))
for i in range(1, n):
result += f(a + i * h)
return result * h
def f(x):
return np.sin(x)
result = integrate_trapezoidal(0, np.pi, 1000, f)
print(result) # Результат: приближённое значение интеграла
```

Эти задачи и их решения демонстрируют основные принципы использования Cython и Numba. Вы можете модифицировать примеры и адаптировать их под собственные задачи для лучшего понимания и закрепления навыков.

3.2 Примеры глубокого обучения с TensorFlow и PyTorch

Глубокое обучение (Deep Learning) играет ключевую роль в современных приложениях искусственного интеллекта, включая обработку изображений, речи, текста и временных рядов. TensorFlow и PyTorch – два наиболее популярных фреймворка для реализации и обучения глубоких нейронных сетей. Ниже приведены примеры реализации моделей глубокого обучения с использованием обоих фреймворков.

Пример 1: Классификация изображений (MNIST)

Задача: Обучить нейронную сеть классифицировать рукописные цифры из датасета MNIST, который содержит изображения размером (28 times 28) с метками от 0 до 9.

Реализация с TensorFlow

```python
import tensorflow as tf
from tensorflow.keras import layers, models, datasets
# Загрузка данных MNIST
(x_train, y_train), (x_test, y_test) = datasets.mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0 # Нормализация данных
# Создание модели
model = models.Sequential([
layers.Flatten(input_shape=(28, 28)), # Преобразование 2D вектора в 1D
layers.Dense(128, activation='relu'), # Полносвязный слой с 128 нейронами
layers.Dropout(0.2), # Dropout для регуляризации
layers.Dense(10, activation='softmax') # Выходной слой (10 классов)
])
# Компиляция модели
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
# Обучение модели
model.fit(x_train, y_train, epochs=5, batch_size=32, validation_split=0.1)
# Оценка модели
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Точность на тестовых данных: {test_acc}")
```

Реализация с PyTorch

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
# Загрузка данных MNIST
```

```
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=32, shuffle=False)
# Определение модели
class MNISTModel(nn.Module):
def __init__(self):
super(MNISTModel, self).__init__()
self.flatten = nn.Flatten()
self.fc1 = nn.Linear(28*28, 128) # Полносвязный слой с 128 нейронами
self.relu = nn.ReLU()
self.dropout = nn.Dropout(0.2)
self.fc2 = nn.Linear(128, 10) # Выходной слой (10 классов)
def forward(self, x):
x = self.flatten(x)
x = self.relu(self.fc1(x))
x = self.dropout(x)
x = self.fc2(x)
return x
model = MNISTModel()
# Оптимизатор и функция потерь
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
# Обучение модели
for epoch in range(5):
model.train()
for batch in train_loader:
images, labels = batch
optimizer.zero_grad()
outputs = model(images)
```

```
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()
print(f"Эпоха {epoch+1}, Потери: {loss.item()}")
# Оценка модели
model.eval()
correct = 0
total = 0
with torch.no_grad():
for batch in test_loader:
images, labels = batch
outputs = model(images)
_, predicted = torch.max(outputs, 1)
total += labels.size(0)
correct += (predicted == labels).sum().item()
print(f"Точность на тестовых данных: {correct / total}")
```

Пример 2: Рекуррентная нейронная сеть (RNN) для анализа текста

Задача: Обучить рекуррентную нейронную сеть предсказывать тональность текста (положительная или отрицательная) на основе IMDB-датасета.

Реализация с TensorFlow

```python
import tensorflow as tf
from tensorflow.keras import layers, datasets, preprocessing
# Загрузка данных IMDB
(x_train, y_train), (x_test, y_test) = datasets.imdb.load_data(num_words=10000)
x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=200)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=200)
# Создание модели
model = models.Sequential([
layers.Embedding(input_dim=10000, output_dim=64, input_length=200), # Слой эмбеддингов
layers.LSTM(64), # LSTM слой
layers.Dense(1, activation='sigmoid') # Выходной слой
```

```
])
# Компиляция модели
model.compile(optimizer='adam',
loss='binary_crossentropy',
metrics=['accuracy'])
# Обучение модели
model.fit(x_train, y_train, epochs=5, batch_size=32, validation_split=0.1)
# Оценка модели
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Точность на тестовых данных: {test_acc}")
```

Реализация с PyTorch

```python
import torch
import torch.nn as nn
import torch.optim as optim
from torchtext.legacy import datasets, data
# Загрузка данных IMDB
TEXT = data.Field(tokenize='spacy', tokenizer_language='en_core_web_sm', batch_first=True, include_lengths=True)
LABEL = data.LabelField(dtype=torch.float)
train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)
TEXT.build_vocab(train_data, max_size=10000)
LABEL.build_vocab(train_data)
train_iterator, test_iterator = data.BucketIterator.splits(
(train_data, test_data),
batch_size=32,
sort_within_batch=True
)
# Определение модели
class RNN(nn.Module):
def __init__(self, vocab_size, embed_size, hidden_size, output_size):
super(RNN, self).__init__()
self.embedding = nn.Embedding(vocab_size, embed_size)
self.rnn = nn.LSTM(embed_size, hidden_size, batch_first=True)
```

```
    self.fc = nn.Linear(hidden_size, output_size)
    self.sigmoid = nn.Sigmoid()
    def forward(self, x, lengths):
    x = self.embedding(x)
    packed = nn.utils.rnn.pack_padded_sequence(x, lengths, batch_first=True, enforce_sorted=False)
    _, (hidden, _) = self.rnn(packed)
    return self.sigmoid(self.fc(hidden[-1]))
    model = RNN(len(TEXT.vocab), 64, 64, 1)
    # Оптимизатор и функция потерь
    criterion = nn.BCELoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    # Обучение модели
    for epoch in range(5):
    model.train()
    for batch in train_iterator:
    optimizer.zero_grad()
    text, lengths = batch.text
    predictions = model(text, lengths).squeeze(1)
    loss = criterion(predictions, batch.label)
    loss.backward()
    optimizer.step()
    print(f"Эпоха {epoch+1}, Потери: {loss.item()}")
    # Оценка модели
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
    for batch in test_iterator:
    text, lengths = batch.text
    predictions = model(text, lengths).squeeze(1)
    predictions = (predictions > 0.5).float()
    correct += (predictions == batch.label).sum().item()
    total += len(batch.label)
    print(f"Точность на тестовых данных: {correct / total}")
    ```
```
```

Эти примеры показывают, как TensorFlow и PyTorch применяются для решения задач классификации изображений и текста. Вы можете усложнять модели, добавлять регуляризацию, пробовать другие архитектуры или модифицировать параметры для изучения различных аспектов глубокого обучения.

Задачи для практики

Ниже приведены задачи для закрепления навыков работы с TensorFlow и PyTorch. Каждая задача сопровождается готовым решением.

Задача 1: Построение простой нейронной сети для предсказания функции

Условие: Создайте модель для предсказания значений функции (y = 2x + 3). Используйте линейную нейронную сеть для обучения модели на искусственных данных.

Решение с TensorFlow:

```python
import tensorflow as tf
import numpy as np
# Генерация данных
x_train = np.linspace(0, 10, 100)
y_train = 2 * x_train + 3 + np.random.normal(0, 1, size=100) # Добавляем шум
# Создание модели
model = tf.keras.Sequential([
tf.keras.layers.Dense(1, input_shape=(1,)) # Один нейрон, один вход
])
# Компиляция модели
model.compile(optimizer='sgd', loss='mse')
# Обучение модели
model.fit(x_train, y_train, epochs=50, verbose=0)
# Проверка результатов
x_test = np.array([5, 15, 25])
```

```
y_pred = model.predict(x_test)
print("Предсказанные значения:", y_pred)
```

Решение с PyTorch:

```python
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
# Генерация данных
x_train = np.linspace(0, 10, 100).astype(np.float32).reshape(-1, 1)
y_train = (2 * x_train + 3 + np.random.normal(0, 1, size=x_train.shape)).astype(np.float32)
# Определение модели
model = nn.Linear(1, 1) # Один вход, один выход
# Оптимизатор и функция потерь
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
# Обучение модели
for epoch in range(50):
inputs = torch.from_numpy(x_train)
targets = torch.from_numpy(y_train)
optimizer.zero_grad()
outputs = model(inputs)
loss = criterion(outputs, targets)
loss.backward()
optimizer.step()
# Проверка результатов
x_test = torch.tensor([[5.0], [15.0], [25.0]])
y_pred = model(x_test).detach().numpy()
print("Предсказанные значения:", y_pred)
```

Задача 2: Многослойный перцептрон для классификации цветов (Iris dataset)

Условие: Обучите модель классифицировать цветы из датасета Iris. Используйте многослойный перцептрон.

Решение с TensorFlow:

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
import tensorflow as tf
# Загрузка данных
data = load_iris()
X = data.data
y = data.target.reshape(-1, 1)
# One-hot кодирование меток
encoder = OneHotEncoder(sparse=False)
y = encoder.fit_transform(y)
# Разделение на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Создание модели
model = tf.keras.Sequential([
tf.keras.layers.Dense(16, activation='relu', input_shape=(4,)), # Первый слой
tf.keras.layers.Dense(16, activation='relu'), # Скрытый слой
tf.keras.layers.Dense(3, activation='softmax') # Выходной слой
])
# Компиляция модели
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
# Обучение модели
model.fit(X_train, y_train, epochs=50, batch_size=8, verbose=0)
# Оценка модели
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Точность модели: {accuracy}")
```

Решение с PyTorch:

```python
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import torch
```

```
import torch.nn as nn
import torch.optim as optim
# Загрузка данных
data = load_iris()
X = data.data
y = data.target
# Нормализация данных
scaler = StandardScaler()
X = scaler.fit_transform(X)
# Разделение на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Преобразование данных в тензоры
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.long)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.long)
# Определение модели
class MLP(nn.Module):
def __init__(self):
super(MLP, self).__init__()
self.fc1 = nn.Linear(4, 16) # Первый слой
self.fc2 = nn.Linear(16, 16) # Скрытый слой
self.fc3 = nn.Linear(16, 3) # Выходной слой
def forward(self, x):
x = torch.relu(self.fc1(x))
x = torch.relu(self.fc2(x))
x = self.fc3(x)
return x
model = MLP()
# Оптимизатор и функция потерь
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)
# Обучение модели
for epoch in range(50):
optimizer.zero_grad()
outputs = model(X_train)
```

```
loss = criterion(outputs, y_train)
loss.backward()
optimizer.step()
# Оценка модели
with torch.no_grad():
y_pred = model(X_test)
y_pred_classes = torch.argmax(y_pred, dim=1)
accuracy = (y_pred_classes == y_test).float().mean()
print(f"Точность модели: {accuracy}")
```

Задача 3: Сверточная нейронная сеть для обработки изображений

Условие: Создайте сверточную нейронную сеть (CNN) для классификации изображений из датасета CIFAR-10.

Решение с TensorFlow:

```python
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
# Загрузка данных CIFAR-10
(x_train, y_train), (x_test, y_test) = datasets.cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0 # Нормализация данных
# Создание модели
model = models.Sequential([
layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)), # Сверточный слой
layers.MaxPooling2D((2, 2)), # MaxPooling
layers.Conv2D(64, (3, 3), activation='relu'),
layers.MaxPooling2D((2, 2)),
layers.Flatten(),
layers.Dense(64, activation='relu'),
layers.Dense(10, activation='softmax') # Выходной слой
])
# Компиляция модели
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
# Обучение модели
model.fit(x_train, y_train, epochs=10, batch_size=32)
```

```
# Оценка модели
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f"Точность модели: {test_acc}")
```

Эти задачи помогут закрепить навыки работы с TensorFlow и PyTorch, научиться решать задачи классификации, регрессии и анализа изображений.

3.3 Практическое применение Scikit-learn для решения бизнес-задач

Scikit-learn – это мощная библиотека Python, которая предоставляет широкий спектр инструментов для реализации алгоритмов машинного обучения, анализа данных и предобработки. Благодаря своей универсальности и удобству, Scikit-learn активно используется для решения бизнес-задач в таких областях, как прогнозирование спроса, анализ оттока клиентов, кредитный скоринг и оптимизация цепочек поставок.

Для понимания практического применения библиотеки рассмотрим основные этапы работы с данными в бизнес-контексте: предобработка данных, построение модели, оценка ее качества и интерпретация результатов.

Пример 1: Прогнозирование оттока клиентов в телекоммуникационной компании

Отток клиентов является критической бизнес-задачей для многих компаний. Выявление клиентов с высоким риском ухода позволяет предпринять меры для удержания, что снижает затраты на привлечение новых пользователей.

Данные представляют собой таблицу с информацией о клиентах, где каждая строка содержит признаки клиента (возраст, ежемесячный платеж, длительность использования услуг и т.д.) и целевой признак – факт ухода клиента (1 – клиент ушел, 0 – остался).

На первом этапе данные очищаются и подготавливаются для анализа. Затем строится модель, которая предсказывает вероятность оттока на основе предоставленных признаков.

```python
# Импорт необходимых библиотек
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from sklearn.preprocessing import StandardScaler
# Загрузка данных
data = pd.read_csv("telecom_churn.csv")
# Предварительная обработка
X = data.drop("Churn", axis=1) # Признаки (без целевой переменной)
y = data["Churn"] # Целевая переменная
# Кодирование категориальных переменных
X = pd.get_dummies(X, drop_first=True)
# Разделение данных на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Нормализация числовых данных
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Создание модели
model = RandomForestClassifier(n_estimators=100, random_state=42)
# Обучение модели
model.fit(X_train, y_train)
# Предсказания
y_pred = model.predict(X_test)
# Оценка качества модели
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
print("Accuracy:", accuracy_score(y_test, y_pred))
```

Данная модель позволяет предсказывать вероятность ухода клиентов с высокой точностью, что может быть полезно для создания программ удержания. Например, клиенты с высоким риском могут получить специальные предложения или бонусы.

Пример 2: Кредитный скоринг

Кредитный скоринг помогает финансовым организациям оценивать вероятность дефолта заемщиков. Используя исторические данные о клиентах (доход, возраст, кредитная история и т.д.), можно построить модель, которая предсказывает, одобрить или отклонить заявку на кредит.

```python
# Импорт необходимых библиотек
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import roc_auc_score, roc_curve
import matplotlib.pyplot as plt
# Загрузка данных
data = pd.read_csv("credit_scoring.csv")
# Предварительная обработка
X = data.drop("Default", axis=1) # Признаки
y = data["Default"] # Целевая переменная
# Разделение данных
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Нормализация данных
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Создание модели логистической регрессии
model = LogisticRegression()
# Обучение модели
model.fit(X_train, y_train)
# Предсказания
y_pred_proba = model.predict_proba(X_test)[:, 1]
# Оценка качества модели
auc_score = roc_auc_score(y_test, y_pred_proba)
print("AUC-ROC:", auc_score)
```

```
# Построение кривой ROC
fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
plt.plot(fpr, tpr, label=f"AUC = {auc_score:.2f}")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend()
plt.show()
```

Данная модель позволяет предсказывать вероятность дефолта заемщика. Финансовые организации могут использовать порог вероятности для принятия решений: если вероятность дефолта выше заданного уровня, кредит отклоняется.

Пример 3: Прогнозирование продаж в ритейле

Прогнозирование объема продаж помогает компаниям оптимизировать запасы, планировать закупки и минимизировать издержки. Используя данные о продажах за предыдущие периоды, можно предсказать спрос на товары.

```python
# Импорт необходимых библиотек
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error
import numpy as np
# Загрузка данных
data = pd.read_csv("retail_sales.csv")
# Предварительная обработка
X = data.drop("Sales", axis=1) # Признаки
y = data["Sales"] # Целевая переменная
# Разделение данных
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Создание модели градиентного бустинга
model = GradientBoostingRegressor(n_estimators=200, learning_rate=0.1, max_depth=3, random_state=42)
# Обучение модели
model.fit(X_train, y_train)
```

```
# Предсказания
y_pred = model.predict(X_test)
# Оценка модели
mse = mean_squared_error(y_test, y_pred)
rmse = np.sqrt(mse)
print("RMSE:", rmse)
```

Прогнозирование продаж позволяет компаниям эффективно управлять цепочками поставок, минимизировать риски недостатка товаров и повысить прибыль.

Эти примеры демонстрируют, как Scikit-learn помогает решать бизнес-задачи в различных областях. Используя инструменты библиотеки, можно создавать точные и интерпретируемые модели, которые улучшают принятие решений и повышают эффективность бизнес-процессов.

Задачи для практики

Ниже представлены практические задачи с решением, которые помогут освоить Scikit-learn на реальных бизнес-примерах.

Задача 1: Прогнозирование оттока клиентов

Условие: Имеются данные телекоммуникационной компании, содержащие информацию о клиентах, такие как возраст, тарифный план, ежемесячный платеж и статус ухода клиента. Необходимо построить модель, которая сможет предсказать вероятность оттока.

Решение:

```python
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, accuracy_score
from sklearn.preprocessing import StandardScaler
# Загрузка данных
data = pd.read_csv("telecom_churn.csv")
```

```
# Предобработка данных
X = data.drop("Churn", axis=1) # Признаки
y = data["Churn"] # Целевая переменная
# Кодирование категориальных признаков
X = pd.get_dummies(X, drop_first=True)
# Разделение данных
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Нормализация числовых признаков
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Обучение модели
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
# Предсказания
y_pred = model.predict(X_test)
# Оценка модели
print("Точность модели:", accuracy_score(y_test, y_pred))
print("\nОтчет классификации:\n", classification_report(y_test, y_pred))
```

Практическая цель: Позволяет выявлять клиентов с высоким риском ухода и применять стратегии их удержания.

Задача 2: Прогнозирование спроса на товар

Условие: Дан набор данных ритейла, содержащий информацию о продажах товаров в разные периоды времени. Необходимо спрогнозировать продажи товара на основе исторических данных.

Решение:

```python
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import numpy as np
import pandas as pd
# Загрузка данных
```

```
data = pd.read_csv("retail_sales.csv")
# Разделение на признаки и целевую переменную
X = data.drop("Sales", axis=1)
y = data["Sales"]
# Разделение на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Обучение модели градиентного бустинга
model = GradientBoostingRegressor(n_estimators=200, learning_rate=0.1, max_depth=3, random_state=42)
model.fit(X_train, y_train)
# Предсказания
y_pred = model.predict(X_test)
# Оценка модели
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print("RMSE:", rmse)
```

Практическая цель: Прогнозирование спроса помогает ритейлерам оптимизировать запасы, снижая издержки и предотвращая дефицит товаров.

Задача 3: Кредитный скоринг

Условие: Имеется набор данных с информацией о клиентах банка (возраст, доход, кредитная история). Необходимо определить, является ли клиент надежным заемщиком.

Решение:

```python
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score, roc_curve, classification_report
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
# Загрузка данных
data = pd.read_csv("credit_scoring.csv")
# Разделение на признаки и целевую переменную
```

```
X = data.drop("Default", axis=1)
y = data["Default"]
# Разделение на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Нормализация данных
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
# Обучение модели логистической регрессии
model = LogisticRegression()
model.fit(X_train, y_train)
# Предсказания вероятностей
y_pred_proba = model.predict_proba(X_test)[:, 1]
# Оценка модели
auc_score = roc_auc_score(y_test, y_pred_proba)
print("AUC-ROC:", auc_score)
# ROC-кривая
fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
plt.figure()
plt.plot(fpr, tpr, label=f"AUC = {auc_score:.2f}")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve")
plt.legend()
plt.show()
# Классификационный отчет
y_pred = model.predict(X_test)
print("\nОтчет классификации:\n", classification_report(y_test, y_pred))
```

Практическая цель: Позволяет банкам принимать обоснованные решения о выдаче кредита, снижая риски финансовых потерь.

Задача 4: Классификация клиентов для маркетинговой кампании

Условие: Имеется набор данных с информацией о клиентах, включая возраст, уровень дохода, активность покупок. Необходимо классифицировать клиентов для выбора целевой аудитории маркетинговой кампании.

Решение:

```python
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.model_selection import train_test_split
import pandas as pd
# Загрузка данных
data = pd.read_csv("customer_data.csv")
# Разделение данных на признаки и целевую переменную
X = data.drop("Segment", axis=1)
y = data["Segment"]
# Разделение на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Обучение модели SVM
model = SVC(kernel='linear', probability=True, random_state=42)
model.fit(X_train, y_train)
# Предсказания
y_pred = model.predict(X_test)
# Оценка качества модели
print("Confusion Matrix:\n", confusion_matrix(y_test, y_pred))
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

Практическая цель: Помогает определять наиболее перспективные сегменты клиентов, чтобы повысить эффективность рекламных кампаний.

Задача 5: Анализ чувствительности отзывов (Sentiment Analysis)

Условие: У вас есть набор данных с текстами отзывов и метками, указывающими на положительный (1) или отрицательный (0) тон отзыва. Требуется построить модель для классификации отзывов.

Решение:

```python
```

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, classification_report
# Загрузка данных
data = {
'text': ["Отличный продукт!", "Ужасное обслуживание", "Очень доволен покупкой",
"Не советую, сплошное разочарование", "Просто прекрасно"],
'label': [1, 0, 1, 0, 1]
}
# Преобразование данных в DataFrame
import pandas as pd
df = pd.DataFrame(data)
# Преобразование текста в числовые признаки
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(df['text'])
y = df['label']
# Разделение на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Обучение модели Наивного Байеса
model = MultinomialNB()
model.fit(X_train, y_train)
# Предсказание
y_pred = model.predict(X_test)
# Оценка качества модели
print("Точность модели:", accuracy_score(y_test, y_pred))
print("\nОтчет классификации:\n", classification_report(y_test, y_pred))
```

Практическая цель: Анализ отзывов позволяет бизнесу понимать настроения клиентов и оперативно реагировать на их запросы.

Задача 6: Сегментация клиентов с использованием кластеризации

Условие: Имеется набор данных с информацией о покупательских привычках клиентов, включая количество покупок, частоту заказов и средний чек. Необходимо разделить клиентов на группы (кластеры), чтобы разработать индивидуальные стратегии работы с каждым сегментом.

Решение:

```python
from sklearn.cluster import KMeans
import matplotlib.pyplot as plt
import pandas as pd
# Загрузка данных
data = {
'Покупки': [15, 25, 30, 40, 50, 60, 65, 80, 90],
'Средний чек': [100, 150, 200, 250, 300, 350, 400, 450, 500]
}
df = pd.DataFrame(data)
# Создание модели KMeans
kmeans = KMeans(n_clusters=3, random_state=42)
df['Кластер'] = kmeans.fit_predict(df)
# Визуализация кластеров
plt.scatter(df['Покупки'], df['Средний чек'], c=df['Кластер'], cmap='viridis')
plt.xlabel('Количество покупок')
plt.ylabel('Средний чек')
plt.title('Сегментация клиентов')
plt.show()
```

Практическая цель: Сегментация клиентов помогает компаниям понимать особенности различных групп покупателей и разрабатывать персонализированные предложения.

Задача 7: Прогнозирование цен на недвижимость

Условие: Дан набор данных с характеристиками недвижимости, такими как площадь, количество комнат, этажность и местоположение. Требуется построить модель, которая будет прогнозировать цены на недвижимость.

Решение:

```python
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import numpy as np
import pandas as pd
# Загрузка данных
data = {
'Площадь': [50, 60, 80, 100, 120, 150],
'Комнаты': [2, 3, 3, 4, 4, 5],
'Этажность': [5, 10, 15, 20, 25, 30],
'Цена': [3.5, 4.2, 5.0, 6.8, 8.0, 9.5]
}
df = pd.DataFrame(data)
# Разделение данных
X = df.drop('Цена', axis=1)
y = df['Цена']
# Разделение на обучающую и тестовую выборки
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# Создание модели
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
# Предсказания
y_pred = model.predict(X_test)
# Оценка модели
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
print("RMSE:", rmse)
```

Практическая цель: Прогнозирование цен на недвижимость используется агентствами для оценки стоимости объектов и улучшения планирования продаж.

Задача 8: Определение аномалий в транзакциях

Условие: Имеется набор данных с информацией о транзакциях, включая сумму, время и ID клиента. Требуется выявить подозрительные операции.

Решение:

```python
from sklearn.ensemble import IsolationForest
import pandas as pd
# Загрузка данных
data = {
'Сумма': [100, 200, 150, 10000, 120, 180, 5000, 150, 170],
'Время': [1, 2, 3, 10, 4, 5, 11, 6, 7]
}
df = pd.DataFrame(data)
# Обучение модели Isolation Forest
model = IsolationForest(contamination=0.2, random_state=42)
df['Аномалия'] = model.fit_predict(df[['Сумма', 'Время']])
# Вывод аномальных транзакций
anomalies = df[df['Аномалия'] == -1]
print("Аномальные транзакции:\n", anomalies)
```

Практическая цель: Выявление аномалий помогает банкам и финансовым организациям предотвращать мошенничество.

Задача 9: Рекомендательная система для фильмов

Условие: Имеется набор данных с рейтингами фильмов от пользователей. Необходимо предложить пользователям рекомендации на основе их предпочтений.

Решение:

```python
from sklearn.metrics.pairwise import cosine_similarity
import pandas as pd
# Пример данных
data = {
'Пользователь': ['А', 'Б', 'В', 'Г'],
'Фильм 1': [5, 4, 3, 0],
'Фильм 2': [4, 0, 5, 2],
'Фильм 3': [3, 0, 4, 5],
'Фильм 4': [0, 2, 0, 4]
}
df = pd.DataFrame(data).set_index('Пользователь')
# Подсчет сходства
```

```
similarity = cosine_similarity(df)
similarity_df = pd.DataFrame(similarity, index=df.index, columns=df.index)
# Рекомендация для пользователя "Г"
print("Сходство пользователей:\n", similarity_df)
```

Практическая цель: Рекомендательные системы помогают повысить вовлеченность пользователей и увеличить продажи продуктов или услуг.

Эти задачи охватывают ключевые аспекты использования Scikit-learn в бизнесе: классификацию, регрессию, прогнозирование и оценку моделей. Они позволяют получить практический опыт и понять, как машинное обучение может применяться для улучшения бизнес-процессов.

Глава 4. Автоматизация задач

4.1 Автоматизация парсинга и сбора данных с BeautifulSoup и Scrapy

Автоматизация парсинга и сбора данных является одной из ключевых задач в современном мире анализа данных. Это позволяет извлекать структурированную информацию из различных веб-ресурсов для использования в бизнесе, исследованиях или обучении. Для решения подобных задач на Python часто используются две популярные библиотеки: BeautifulSoup и Scrapy.

BeautifulSoup – это библиотека для синтаксического анализа HTML и XML. Она удобна для относительно простых задач парсинга. Scrapy, в свою очередь, представляет собой мощный фреймворк для веб-скрейпинга, позволяющий реализовывать сложные сценарии сбора данных с высокой производительностью.

BeautifulSoup: Основы работы

Библиотека BeautifulSoup позволяет легко извлекать информацию из HTML-документов. Она преобразует HTML в дерево объектов, с которым можно работать для извлечения нужных данных.

Пример задачи: Сбор заголовков статей с новостного сайта.

```python
from bs4 import BeautifulSoup
import requests
# URL страницы для парсинга
url = "https://example-news-site.com"
# Отправка HTTP-запроса и получение HTML-кода страницы
response = requests.get(url)
html_content = response.text
# Создание объекта BeautifulSoup
soup = BeautifulSoup(html_content, "html.parser")
# Извлечение заголовков статей
headlines = soup.find_all("h2", class_="headline")
```

```
# Вывод заголовков
for idx, headline in enumerate(headlines, start=1):
print(f"{idx}. {headline.text.strip()}")
```

В данном примере мы:

1. Отправили запрос на веб-сайт.
2. Получили HTML-код страницы.
3. С помощью BeautifulSoup нашли все заголовки (`<h2>` с классом `headline`).
4. Вывели их на экран.

Практическое применение: Такой парсинг может быть использован для мониторинга новостных сайтов, отслеживания статей о вашей компании или конкуренте.

Расширенные возможности BeautifulSoup

BeautifulSoup предоставляет множество методов для сложных задач, например, извлечение ссылок, изображений или фильтрация элементов.

Пример задачи: Сбор всех ссылок с веб-страницы.

```python
# Извлечение всех ссылок (теги <a>)
links = soup.find_all("a")
# Фильтрация ссылок и их вывод
for link in links:
href = link.get("href")
if href and href.startswith("http"):
print(href)
```

Этот скрипт собирает все ссылки с веб-страницы, проверяет, чтобы они начинались с `http`, и выводит их.

Scrapy: Фреймворк для сложного парсинга

Если BeautifulSoup удобен для небольших задач, то Scrapy подходит для масштабных проектов, таких как сбор данных с сотен или тысяч страниц. Это мощный инструмент, поддерживающий многопоточность, обработку JavaScript и управление сложными потоками данных.

Пример задачи: Сбор информации о товарах из интернет-магазина.

1. Установите Scrapy:

```bash
pip install scrapy
```

2. Создайте новый проект:

```bash
scrapy startproject ecommerce_parser
```

3. В папке проекта создайте паука (специальный класс для парсинга):

```bash
scrapy genspider products example-ecommerce-site.com
```

4. В файле `products.py` настройте паука:

```python
import scrapy
class ProductsSpider(scrapy.Spider):
name = "products"
start_urls = ["https://example-ecommerce-site.com/category"]
def parse(self, response):
# Извлечение информации о товарах
for product in response.css("div.product"):
yield {
"name": product.css("h3.product-title::text").get(),
"price": product.css("span.price::text").get(),
"availability": product.css("span.availability::text").get(),
}
# Переход на следующую страницу
next_page = response.css("a.next-page::attr(href)").get()
if next_page:
yield response.follow(next_page, self.parse)
```

Объяснение работы кода:

1. Паук начинает с указанного URL (`start_urls`).
2. С помощью CSS-селекторов извлекается информация о товарах (название, цена, доступность).

3. Если на странице есть кнопка перехода на следующую страницу (`a.next-page`), паук автоматически переходит на неё и продолжает парсинг.

5. Запустите паука:

```bash
scrapy crawl products -o products.json
```

Этот код сохраняет собранные данные в файл `products.json`.

Пример реального применения Scrapy: Сбор данных о вакансиях

Для сбора информации о вакансиях можно настроить Scrapy для работы с популярными сайтами поиска работы. Например, собрать названия вакансий, зарплаты и местоположения.

```python
class JobsSpider(scrapy.Spider):
name = "jobs"
start_urls = ["https://example-job-site.com/jobs"]
def parse(self, response):
for job in response.css("div.job-listing"):
yield {
"title": job.css("h2.title::text").get(),
"salary": job.css("span.salary::text").get(),
"location": job.css("span.location::text").get(),
}
# Переход на следующую страницу
next_page = response.css("a.next-page::attr(href)").get()
if next_page:
yield response.follow(next_page, self.parse)
```

Практическая цель: Такой парсинг может быть полезен для анализа рынка труда, отслеживания предложений в конкретной сфере или для построения собственного поисковика вакансий.

Преимущества и недостатки инструментов

BeautifulSoup:

– Простота в использовании и настройке.

– Подходит для небольших и средних задач.

– Меньше контроля над сложными сценариями, такими как переход по страницам или обработка AJAX.

Scrapy:

– Высокая производительность и гибкость.

– Возможность масштабного сбора данных.

– Более сложная настройка, особенно для новичков.

Эти инструменты являются мощными инструментами автоматизации сбора данных и широко используются как в коммерческих, так и в исследовательских проектах. Выбор между BeautifulSoup и Scrapy зависит от сложности задачи и объема данных, которые необходимо собрать.

Задачи для практики

Задача 1: Сбор заголовков новостей с помощью BeautifulSoup

Условие: Соберите заголовки новостей с главной страницы новостного сайта и выведите их в виде списка. Для парсинга используйте BeautifulSoup.

Решение:

```python
from bs4 import BeautifulSoup
import requests
# URL новостного сайта
url = "https://example-news-site.com"
# Запрос к странице
response = requests.get(url)
html_content = response.text
# Создание объекта BeautifulSoup
soup = BeautifulSoup(html_content, "html.parser")
# Поиск заголовков
headlines = soup.find_all("h2", class_="headline")
# Вывод заголовков
print("Заголовки новостей:")
for idx, headline in enumerate(headlines, start=1):
```

```
print(f"{idx}. {headline.text.strip()}")
```

Результат:

Скрипт собирает и выводит заголовки новостей, что полезно для мониторинга актуальных событий или анализа медийного контента.

Задача 2: Сбор данных о продуктах с интернет-магазина с помощью Scrapy

Условие: Соберите данные о продуктах (название, цену и наличие на складе) с категории товаров интернет-магазина. Реализуйте это с помощью Scrapy.

Решение:

1. Создайте проект Scrapy:

```bash
scrapy startproject products_parser
```

2. В папке `spiders` создайте файл `products_spider.py` с содержимым:

```python
import scrapy
class ProductsSpider(scrapy.Spider):
name = "products"
start_urls = ["https://example-ecommerce-site.com/category"]
def parse(self, response):
# Извлечение данных о продуктах
for product in response.css("div.product"):
yield {
"name": product.css("h3.product-title::text").get(),
"price": product.css("span.price::text").get(),
"availability": product.css("span.availability::text").get(),
}
# Переход на следующую страницу
next_page = response.css("a.next-page::attr(href)").get()
if next_page:
yield response.follow(next_page, self.parse)
```

3. Запустите паука:

```bash
scrapy crawl products -o products.json
```

Результат:

Все данные о продуктах сохраняются в файл `products.json`, который можно использовать для анализа ассортимента или мониторинга цен.

Задача 3: Сбор всех ссылок на сайте с помощью BeautifulSoup

Условие: Создайте скрипт, который собирает все ссылки с указанного веб-сайта и сохраняет их в текстовый файл.

Решение:

```python
from bs4 import BeautifulSoup
import requests
# URL сайта
url = "https://example.com"
# Запрос к странице
response = requests.get(url)
html_content = response.text
# Создание объекта BeautifulSoup
soup = BeautifulSoup(html_content, "html.parser")
# Извлечение всех ссылок
links = soup.find_all("a")
# Сохранение ссылок в файл
with open("links.txt", "w") as file:
for link in links:
href = link.get("href")
if href and href.startswith("http"):
file.write(href + "\n")
print(href)
```

Результат:

Скрипт сохраняет все ссылки в файл `links.txt`, что может быть полезно для анализа структуры сайта или построения карты ссылок.

Задача 4: Сбор вакансий с сайта поиска работы с помощью Scrapy

Условие: Соберите информацию о вакансиях (название, зарплата и местоположение) с сайта поиска работы.

Решение:

1. Создайте проект Scrapy:

```bash
scrapy startproject job_parser
```

2. В папке `spiders` создайте файл `jobs_spider.py` с содержимым:

```python
import scrapy
class JobsSpider(scrapy.Spider):
name = "jobs"
start_urls = ["https://example-job-site.com/jobs"]
def parse(self, response):
# Извлечение данных о вакансиях
for job in response.css("div.job-listing"):
yield {
"title": job.css("h2.title::text").get(),
"salary": job.css("span.salary::text").get(),
"location": job.css("span.location::text").get(),
}
# Переход на следующую страницу
next_page = response.css("a.next-page::attr(href)").get()
if next_page:
yield response.follow(next_page, self.parse)
```

3. Запустите паука:

```bash
scrapy crawl jobs -o jobs.json
```

Результат: Собранные вакансии сохраняются в файл `jobs.json`, что может быть использовано для анализа рынка труда.

Задача 5: Поиск определенного текста на веб-странице с помощью BeautifulSoup

Условие: Напишите скрипт, который проверяет наличие заданного текста на странице (например, "скидка") и возвращает список блоков, в

которых он встречается.

Решение:

```python
from bs4 import BeautifulSoup
import requests
# URL сайта
url = "https://example-promo-site.com"
search_text = "скидка"
# Запрос к странице
response = requests.get(url)
html_content = response.text
# Создание объекта BeautifulSoup
soup = BeautifulSoup(html_content, "html.parser")
# Поиск текста
results = soup.find_all(string=lambda text: search_text in text.lower())
# Вывод блоков с текстом
for idx, result in enumerate(results, start=1):
print(f"{idx}. {result.strip()}")
```

Результат: Скрипт находит и выводит все блоки текста, содержащие слово "скидка". Это может быть полезно для анализа промо-страниц или поиска упоминаний ключевых слов.

Задача 6: Автоматическая загрузка изображений с веб-страницы

Условие: Соберите все изображения с веб-страницы и сохраните их в локальной папке.

Решение:

```python
import os
import requests
from bs4 import BeautifulSoup
# URL сайта
url = "https://example-images-site.com"
# Папка для сохранения изображений
os.makedirs("images", exist_ok=True)
# Запрос к странице
response = requests.get(url)
```

```
html_content = response.text
# Создание объекта BeautifulSoup
soup = BeautifulSoup(html_content, "html.parser")
# Извлечение ссылок на изображения
images = soup.find_all("img")
# Скачивание изображений
for img in images:
src = img.get("src")
if src and src.startswith("http"):
img_data = requests.get(src).content
img_name = os.path.join("images", os.path.basename(src))
with open(img_name, "wb") as file:
file.write(img_data)
print(f"Изображение сохранено: {img_name}")
```

Результат: Все изображения с веб-страницы сохраняются в папке `images`, что может быть полезно для создания баз данных изображений или анализа контента.

Эти задачи охватывают разные аспекты автоматизации с использованием BeautifulSoup и Scrapy. Они помогут вам научиться работать с HTML-контентом, собирать данные, анализировать сайты и автоматизировать рутинные задачи.

4.2 Автоматизация работы с документами (PDF, Excel, Word)

Работа с документами – одна из самых востребованных задач автоматизации в офисной и бизнес-среде. С помощью Python можно автоматизировать множество процессов: извлечение текста из PDF, обработку таблиц в Excel, создание отчетов в Word, массовую генерацию документов и многое другое. В этом разделе подробно рассмотрим подходы к автоматизации работы с различными типами файлов.

Работа с PDF-документами

Для автоматизации работы с PDF-файлами Python предоставляет такие библиотеки, как PyPDF2, pdfplumber и reportlab. Они позволяют читать, редактировать, извлекать текст, объединять и разделять PDF-документы.

Извлечение текста из PDF

Для извлечения текста из PDF-файлов популярной библиотекой является pdfplumber.

Пример задачи: Извлечь текст из PDF-документа.

```python
import pdfplumber
# Путь к PDF-файлу
pdf_path = "example.pdf"
# Открываем PDF
with pdfplumber.open(pdf_path) as pdf:
for page in pdf.pages:
text = page.extract_text()
print(text)
```

Объяснение работы:

1. Открывается PDF-файл с помощью `pdfplumber.open`.
2. Перебираются страницы, и текст извлекается методом `extract_text`.

Применение: Автоматическое извлечение данных из договоров, сканированных документов или отчетов.

Создание PDF

Для создания PDF-документов можно использовать библиотеку reportlab.

Пример задачи: Создать простой PDF с заголовком и текстом.

```python
from reportlab.pdfgen import canvas
# Путь для сохранения PDF
output_path = "output.pdf"
# Создание документа
c = canvas.Canvas(output_path)
c.setFont("Helvetica", 12)
```

```
# Добавление текста
c.drawString(100, 750, "Заголовок PDF-документа")
c.drawString(100, 700, "Этот текст был создан автоматически с помощью Python.")
# Сохранение файла
c.save()
print("PDF создан:", output_path)
```

Объяснение работы:

1. Создается объект `canvas` для работы с PDF.
2. Используются методы `drawString` для добавления текста и `save` для сохранения файла.

Применение: Генерация отчетов, сертификатов, счетов-фактур.

Работа с Excel-файлами

Для работы с Excel-файлами чаще всего используются библиотеки openpyxl (для файлов формата `.xlsx`) и pandas (для аналитической обработки данных).

Чтение данных из Excel

Пример задачи: Прочитать данные из таблицы Excel и вывести их на экран.

```python
import openpyxl
# Путь к Excel-файлу
file_path = "example.xlsx"
# Открываем книгу
workbook = openpyxl.load_workbook(file_path)
# Выбираем активный лист
sheet = workbook.active
# Чтение данных
for row in sheet.iter_rows(values_only=True):
print(row)
```

Объяснение работы:

1. С помощью `load_workbook` открывается Excel-файл.
2. Лист перебирается построчно, и значения выводятся.

Применение: Анализ финансовых отчетов, обработка списков товаров, обработка табличных данных.

Создание и редактирование Excel

Пример задачи: Создать Excel-файл с таблицей данных.

```python
from openpyxl import Workbook
# Создание новой книги
workbook = Workbook()
sheet = workbook.active
# Добавление данных
sheet["A1"] = "Имя"
sheet["B1"] = "Возраст"
sheet["C1"] = "Город"
data = [("Иван", 25, "Москва"), ("Мария", 30, "Санкт-Петербург"), ("Олег", 28, "Новосибирск")]
for row in data:
sheet.append(row)
# Сохранение файла
workbook.save("output.xlsx")
print("Excel-файл создан: output.xlsx")
```

Объяснение работы:

1. Создается новая книга.
2. В ячейки добавляются данные вручную или методом `append`.
3. Книга сохраняется в файл.

Применение: Создание отчетов, генерация списков участников, создание шаблонов.

Работа с Word-документами

Для работы с Word-документами используется библиотека python-docx. Она позволяет создавать, редактировать и читать документы в формате `.docx`.

Чтение текста из Word

Пример задачи: Извлечь текст из Word-документа.

```python
from docx import Document
```

```
# Путь к Word-файлу
file_path = "example.docx"
# Открытие документа
doc = Document(file_path)
# Чтение текста
for paragraph in doc.paragraphs:
print(paragraph.text)
```

Объяснение работы:
1. С помощью `Document` открывается Word-документ.
2. Все абзацы перебираются, и их текст выводится.
Применение: Извлечение данных из отчетов, актов, инструкций.

Создание Word-документа
Пример задачи: Создать документ Word с заголовком и списком.
```python
from docx import Document
# Создание нового документа
doc = Document()
# Добавление заголовка
doc.add_heading("Автоматически созданный документ", level=1)
# Добавление текста
doc.add_paragraph("Этот документ был создан с использованием Python.")
# Добавление списка
items = ["Пункт 1", "Пункт 2", "Пункт 3"]
for item in items:
doc.add_paragraph(item, style="List Bullet")
# Сохранение файла
doc.save("output.docx")
print("Word-документ создан: output.docx")
```

Объяснение работы:
1. Создается объект `Document`.
2. С помощью методов `add_heading`, `add_paragraph` добавляется текст и форматирование.
3. Документ сохраняется.

Применение: Создание отчетов, писем, договоров.

Преимущества автоматизации работы с документами

1. Скорость: Обработка большого количества файлов за считанные секунды.
2. Точность: Уменьшение человеческих ошибок.
3. Гибкость: Возможность работы с разными форматами документов.
4. Экономия ресурсов: Автоматизация рутинных задач освобождает время для более сложных и творческих задач.

Эти подходы могут быть применены в самых разных сферах, включая бухгалтерию, аналитику, создание отчетности и обработку данных.

Задачи для практики

Задача 1: Извлечение таблиц из PDF

Условие: Дан PDF-документ, содержащий таблицы. Необходимо извлечь данные таблиц и сохранить их в CSV-файл.

Решение:

```python
import pdfplumber
import csv
# Путь к PDF и CSV
pdf_path = "example.pdf"
csv_path = "output.csv"
# Открытие PDF
with pdfplumber.open(pdf_path) as pdf, open(csv_path, "w", newline="") as csv_file:
writer = csv.writer(csv_file)
for page in pdf.pages:
tables = page.extract_tables()
for table in tables:
writer.writerows(table)
```

```
print("Таблицы сохранены в:", csv_path)
```

Результат: Данные таблиц из PDF экспортируются в CSV.

Задача 2: Автоматическая генерация счета в PDF

Условие: Создайте PDF-файл, содержащий счет с информацией о заказчике, дате и списке товаров.

Решение:

```python
from reportlab.lib.pagesizes import letter
from reportlab.pdfgen import canvas
# Путь для сохранения PDF
output_path = "invoice.pdf"
# Данные счета
client_name = "Иван Иванов"
date = "16.01.2025"
items = [("Товар A", 2, 500), ("Товар B", 1, 1200), ("Товар C", 3, 300)]
# Создание PDF
c = canvas.Canvas(output_path, pagesize=letter)
c.setFont("Helvetica", 12)
# Заголовок
c.drawString(100, 750, f"Счет для: {client_name}")
c.drawString(100, 730, f"Дата: {date}")
# Таблица товаров
c.drawString(100, 700, "Наименование Количество Цена")
y = 680
for item, qty, price in items:
c.drawString(100, y, f"{item} {qty} {price}")
y -= 20
# Сохранение
c.save()
print("Счет сохранен в:", output_path)
```

Результат: Создается PDF с информацией о заказе.

Задача 3: Поиск и замена текста в Word

Условие: В Word-документе заменить все вхождения слова "старый" на "новый".

Решение:

```python
from docx import Document
# Путь к документу
file_path = "example.docx"
output_path = "updated.docx"
# Открытие документа
doc = Document(file_path)
# Замена текста
for paragraph in doc.paragraphs:
if "старый" in paragraph.text:
paragraph.text = paragraph.text.replace("старый", "новый")
# Сохранение
doc.save(output_path)
print("Обновленный документ сохранен в:", output_path)
```

Результат: В документе заменяется указанный текст.

Задача 4: Генерация отчетов в Excel

Условие: Составьте отчет о продажах за неделю. Данные должны включать день недели, количество проданных товаров и общую выручку. Сохраните отчет в Excel.

Решение:

```python
from openpyxl import Workbook
# Данные
data = [
("Понедельник", 100, 15000),
("Вторник", 80, 12000),
("Среда", 90, 13500),
("Четверг", 110, 16500),
("Пятница", 120, 18000),
]
# Создание книги
workbook = Workbook()
```

```
sheet = workbook.active
sheet.title = "Отчет о продажах"
# Добавление заголовков
sheet.append(("День недели", "Количество товаров", "Выручка"))
# Добавление данных
for row in data:
sheet.append(row)
# Сохранение
workbook.save("sales_report.xlsx")
print("Отчет сохранен: sales_report.xlsx")
```

Результат: Создается Excel-таблица с данными о продажах.

Задача 5: Автоматическая сортировка данных в Excel

Условие: У вас есть Excel-файл с таблицей сотрудников. Необходимо отсортировать данные по возрасту и сохранить их в новый файл.

Решение:

```python
import pandas as pd
# Чтение файла
df = pd.read_excel("employees.xlsx")
# Сортировка по возрасту
sorted_df = df.sort_values(by="Возраст")
# Сохранение отсортированных данных
sorted_df.to_excel("sorted_employees.xlsx", index=False)
print("Отсортированный файл сохранен: sorted_employees.xlsx")
```

Результат: Данные сортируются и сохраняются в новый файл.

Задача 6: Извлечение данных из таблицы в Word

Условие:Дан Word-документ с таблицей. Извлеките все данные таблицы и выведите их в консоль.

Решение:

```python
from docx import Document
# Путь к документу
file_path = "table.docx"
```

```
# Открытие документа
doc = Document(file_path)
# Извлечение данных из таблиц
for table in doc.tables:
for row in table.rows:
print([cell.text for cell in row.cells])
```

Результат: Данные из таблицы выводятся в консоль.

Задача 7: Массовое переименование PDF-файлов

Условие: В папке находится несколько PDF-документов. Переименуйте их, добавив текущую дату в начало имени файла.

Решение:

```python
import os
from datetime import datetime
# Папка с PDF
folder_path = "./pdf_files"
date_prefix = datetime.now().strftime("%Y-%m-%d")
# Переименование файлов
for file_name in os.listdir(folder_path):
if file_name.endswith(".pdf"):
old_path = os.path.join(folder_path, file_name)
new_path = os.path.join(folder_path, f"{date_prefix}_{file_name}")
os.rename(old_path, new_path)
print(f"Переименован: {new_path}")
```

Результат: Все файлы в указанной папке получают префикс с текущей датой.

Эти задачи охватывают широкий спектр сценариев работы с PDF, Excel и Word, которые помогут вам развить навыки автоматизации документооборота.

4.3 Создание и развертывание автоматизированных процессов с Airflow

Airflow – это мощный инструмент для оркестрации рабочих процессов, который позволяет создавать, планировать и мониторить автоматизированные процессы. С его помощью можно управлять сложными пайплайнами данных, где задачи зависят друг от друга. Airflow предоставляет интерфейс для визуализации и управления процессами, а также гибкий способ описания задач через код на Python.

Основной концепцией Airflow является DAG (Directed Acyclic Graph), представляющий собой граф, где вершины – это задачи, а направленные ребра указывают зависимости между ними. Задачи в Airflow выполняются в строгом порядке, определяемом этими зависимостями.

Airflow предоставляет множество возможностей для создания автоматизированных процессов. Каждая задача может быть связана с различными действиями, такими как выполнение SQL-запросов, запуск Python-скриптов, взаимодействие с API или выполнение системных команд. DAG может быть настроен так, чтобы запускаться в определенное время или при наступлении определенных условий.

Для создания DAG в Airflow используется декларативный подход. Код пишется на Python, что позволяет гибко управлять задачами, задавать параметры и обрабатывать исключения. Например, если нужно загрузить данные из одного источника, обработать их и сохранить результат в другом месте, можно создать DAG, в котором каждая из этих операций будет представлена отдельной задачей.

Каждая задача в Airflow называется оператором (Operator). Операторы бывают разных типов: для работы с Python, Bash, SQL, API и многими другими. Например, PythonOperator выполняет пользовательские функции, а BashOperator позволяет запускать команды оболочки. Операторы могут использоваться как по отдельности, так и в комбинации, чтобы создать сложные цепочки обработки данных.

Airflow также поддерживает концепцию переменных и подключений, которые упрощают управление конфигурацией.

Переменные используются для хранения ключей API, путей к файлам и других данных, которые могут изменяться между средами. Подключения содержат параметры для взаимодействия с базами данных, серверами и другими внешними системами.

После создания DAG его можно развернуть в Airflow. Для этого DAG сохраняется в специальной папке, из которой система автоматически считывает файлы. При запуске Airflow анализирует граф и подготавливает его к выполнению. Для мониторинга используется веб-интерфейс, где отображается статус задач, расписание и журналы выполнения.

Airflow позволяет настраивать параметры выполнения, такие как интервалы повторения, временные метки и политики повторных попыток. Например, если задача завершилась с ошибкой, можно настроить автоматическое повторное выполнение через определенное время. Это особенно полезно для систем, где ошибки могут быть вызваны временной недоступностью ресурсов.

Для масштабирования рабочих процессов Airflow поддерживает несколько исполнителей (executors). Локальный исполнитель используется для запуска задач на одном сервере, а распределенные исполнители, такие как CeleryExecutor, позволяют выполнять задачи на нескольких серверах. Это делает Airflow подходящим инструментом для больших проектов с интенсивными вычислениями.

Одним из преимуществ Airflow является его интеграция с другими инструментами и системами. Например, можно автоматически выгружать данные из облачных хранилищ, таких как AWS S3 или Google Cloud Storage, а затем анализировать их с помощью Spark или других инструментов. Также Airflow предоставляет встроенные механизмы для работы с популярными базами данных и системами очередей сообщений.

Создание и развертывание автоматизированных процессов с Airflow упрощает управление сложными цепочками задач, минимизирует количество ошибок и улучшает повторяемость процессов. Это делает Airflow незаменимым инструментом для компаний, работающих с большими объемами данных и сложными пайплайнами.

Задачи для практики

Задача 1: Простая автоматизация задачи с Python

Условие: Создайте DAG, который каждые 10 минут выполняет Python-функцию, печатающую "Привет, Airflow!".

Решение:

```python
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.python import PythonOperator
# Функция для выполнения
def greet():
print("Привет, Airflow!")
# Определение DAG
default_args = {
"owner": "airflow",
"retries": 1,
"retry_delay": timedelta(minutes=5),
}
with DAG(
dag_id="simple_python_dag",
default_args=default_args,
description="Простой DAG для выполнения Python-функции",
schedule_interval="*/10 * * * *", # каждые 10 минут
start_date=datetime(2025, 1, 16),
catchup=False,
) as dag:
task = PythonOperator(
task_id="greet_task",
python_callable=greet,
)
task
```

Результат: DAG каждые 10 минут выполняет задачу, выводящую "Привет, Airflow!" в логах.

Задача 2: Загрузка данных из API и сохранение в файл

Условие: Создайте DAG, который загружает данные с API и сохраняет их в локальный файл.

Решение:

```python
import requests
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.python import PythonOperator
# Функция для загрузки данных
def fetch_data():
url = "https://jsonplaceholder.typicode.com/posts"
response = requests.get(url)
data = response.json()
with open("/tmp/posts.json", "w") as f:
import json
json.dump(data, f)
# Определение DAG
default_args = {
"owner": "airflow",
"retries": 2,
"retry_delay": timedelta(minutes=3),
}
with DAG(
dag_id="api_to_file_dag",
default_args=default_args,
description="DAG для загрузки данных из API и сохранения в файл",
schedule_interval="@hourly",
start_date=datetime(2025, 1, 16),
catchup=False,
) as dag:
task = PythonOperator(
task_id="fetch_data_task",
python_callable=fetch_data,
)
task
```

Результат: DAG загружает данные с API и сохраняет их в файл `/tmp/posts.json` каждый час.

Задача 3: ETL-процесс с BashOperator

Условие: Создайте DAG для выполнения ETL-процесса: копирования данных из одного файла в другой.

Решение:

```python
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.bash import BashOperator
# Определение DAG
default_args = {
"owner": "airflow",
"retries": 1,
"retry_delay": timedelta(minutes=2),
}
with DAG(
dag_id="etl_with_bash",
default_args=default_args,
description="ETL-процесс с BashOperator",
schedule_interval="@daily",
start_date=datetime(2025, 1, 16),
catchup=False,
) as dag:
extract = BashOperator(
task_id="extract_task",
bash_command="cp /path/to/source_file.csv /path/to/extracted_file.csv",
)
transform = BashOperator(
task_id="transform_task",
bash_command="sed -i 's/old_value/new_value/g' /path/to/extracted_file.csv",
)
load = BashOperator(
task_id="load_task",
bash_command="mv /path/to/extracted_file.csv /path/to/target_file.csv",
```

)
extract >> transform >> load
```

Результат: DAG ежедневно выполняет ETL-процесс: копирует файл, заменяет значения и перемещает результат в папку назначения.

**Задача 4: Подключение к базе данных**

Условие: Создайте DAG для выполнения SQL-запроса в PostgreSQL.

Решение:

```python
from datetime import datetime, timedelta
from airflow import DAG
from airflow.providers.postgres.operators.postgres import PostgresOperator
# Определение DAG
default_args = {
"owner": "airflow",
"retries": 1,
"retry_delay": timedelta(minutes=5),
}
with DAG(
dag_id="postgres_query_dag",
default_args=default_args,
description="DAG для выполнения SQL-запроса в PostgreSQL",
schedule_interval="@daily",
start_date=datetime(2025, 1, 16),
catchup=False,
) as dag:
task = PostgresOperator(
task_id="run_query",
postgres_conn_id="postgres_default", # Подключение, настроенное в Airflow
sql="INSERT INTO my_table (column1, column2) VALUES ('значение1', 'значение2');",
)
task
```
```

Результат: DAG ежедневно вставляет данные в таблицу PostgreSQL.

Задача 5: Планирование цепочки задач

Условие: Создайте DAG, который выполняет 3 задачи последовательно: подготовка данных, обработка и создание отчета.

Решение:

```python
from datetime import datetime, timedelta
from airflow import DAG
from airflow.operators.python import PythonOperator
# Функции для задач
def prepare_data():
print("Данные подготовлены.")
def process_data():
print("Данные обработаны.")
def generate_report():
print("Отчет создан.")
# Определение DAG
default_args = {
"owner": "airflow",
"retries": 1,
"retry_delay": timedelta(minutes=5),
}
with DAG(
dag_id="data_pipeline_dag",
default_args=default_args,
description="Цепочка задач: подготовка, обработка, отчет",
schedule_interval="@daily",
start_date=datetime(2025, 1, 16),
catchup=False,
) as dag:
prepare_task = PythonOperator(
task_id="prepare_task",
python_callable=prepare_data,
)
process_task = PythonOperator(
task_id="process_task",
```

```
    python_callable=process_data,
    )
    report_task = PythonOperator(
    task_id="report_task",
    python_callable=generate_report,
    )
    prepare_task >> process_task >> report_task
    ```
```

Результат: DAG выполняет три задачи последовательно каждый день.

Эти примеры иллюстрируют, как создавать DAG для автоматизации различных процессов: от простой функции до интеграции с базами данных и построения ETL.
```

Глава 5. Разработка высокопроизводительных приложений

5.1 Асинхронное программирование с Asyncio

Асинхронное программирование – это метод, который позволяет обрабатывать несколько задач одновременно, не блокируя выполнение программы. Это особенно полезно для задач ввода-вывода (например, сетевые запросы, взаимодействие с базой данных или файловой системой), где основная программа может продолжать работу, пока другая задача ожидает завершения операции.

Модуль asyncio, встроенный в стандартную библиотеку Python, предоставляет средства для создания и выполнения асинхронных программ. Основная идея заключается в использовании корутин, которые позволяют приостанавливать выполнение функции и возобновлять её позже, что делает код неблокирующим.

Основные концепции Asyncio

Корутины – это функции, объявленные с помощью ключевого слова `async def`. Они возвращают объект coroutine и выполняются только при явном вызове через цикл событий (event loop).

Цикл событий (Event Loop) – это механизм, который управляет выполнением корутин. Он отвечает за планирование задач и передачу управления между ними.

Задачи (Tasks) – это обёртки над корутинами, которые запускаются через цикл событий. Они позволяют отслеживать выполнение и получать результаты.

Ожидание (await) – ключевое слово, используемое для приостановки выполнения корутины до завершения другой асинхронной операции.

Фьючерсы (Futures) – это объекты, которые представляют результат асинхронной операции. Они используются для работы с результатами, которые станут доступны в будущем.

Пример 1. Простая корутина

Асинхронная функция позволяет приостанавливать выполнение с помощью `await`:

```python
import asyncio
async def say_hello():
print("Привет!")
await asyncio.sleep(2) # Приостановка на 2 секунды
print("Как дела?")
# Запуск корутины
asyncio.run(say_hello())
```

Результат:

```
Привет!
(пауза 2 секунды)
Как дела?
```

Пример 2. Выполнение нескольких задач параллельно

Использование `asyncio.gather` позволяет запускать несколько корутин одновременно:

```python
import asyncio
async def task_1():
print("Начало задачи 1")
await asyncio.sleep(3)
print("Завершение задачи 1")
async def task_2():
print("Начало задачи 2")
await asyncio.sleep(1)
print("Завершение задачи 2")
async def main():
await asyncio.gather(task_1(), task_2())
asyncio.run(main())
```

Результат:

```
```

Начало задачи 1
Начало задачи 2
Завершение задачи 2
Завершение задачи 1
```

Объяснение: Задача 2 завершается первой, так как её пауза короче.

**Пример 3. Асинхронная загрузка данных**

Симулируем параллельную загрузку данных из нескольких источников:

```python
import asyncio
async def fetch_data(source):
print(f"Загрузка данных из {source}…")
await asyncio.sleep(2) # Имитируем задержку сети
print(f"Данные из {source} загружены")
async def main():
sources = ["Источник 1", "Источник 2", "Источник 3"]
tasks = [fetch_data(source) for source in sources]
await asyncio.gather(*tasks)
asyncio.run(main())
```

Результат:

```
Загрузка данных из Источник 1…
Загрузка данных из Источник 2…
Загрузка данных из Источник 3…
Данные из Источник 1 загружены
Данные из Источник 2 загружены
Данные из Источник 3 загружены
```

Объяснение: Все задачи выполняются параллельно.

Пример 4. Асинхронные очереди

Очереди (`asyncio.Queue`) используются для управления задачами, которые должны обрабатываться в определённой последовательности.

```python

```
import asyncio
async def producer(queue):
for i in range(5):
print(f"Производитель: добавление {i}")
await queue.put(i)
await asyncio.sleep(1)
async def consumer(queue):
while True:
item = await queue.get()
print(f"Потребитель: обработка {item}")
queue.task_done()
async def main():
queue = asyncio.Queue()
producer_task = asyncio.create_task(producer(queue))
consumer_task = asyncio.create_task(consumer(queue))
await producer_task # Ожидаем завершения производителя
await queue.join() # Ждём, пока очередь опустеет
consumer_task.cancel() # Завершаем потребителя
asyncio.run(main())
```

Результат:

```
Производитель: добавление 0
Потребитель: обработка 0
Производитель: добавление 1
Потребитель: обработка 1
…
```

Объяснение: Производитель добавляет элементы в очередь, а потребитель их обрабатывает.

Преимущества асинхронного программирования

1. Эффективное использование ресурсов: Позволяет выполнять операции ввода-вывода, не блокируя основной поток.
2. Высокая масштабируемость: Подходит для систем, обрабатывающих множество одновременных запросов.

3. Лёгкость интеграции с сетевыми библиотеками: Асинхронные запросы к API, обработка веб-трафика и работа с сокетами.

Области применения Asyncio

1. Сетевые приложения: Серверы и клиенты, работающие с большими объёмами соединений.
2. Веб-скрапинг: Асинхронная загрузка данных с множества сайтов.
3. Обработка данных: Конвейеры, обрабатывающие входные данные в реальном времени.
4. Микросервисы: Асинхронное взаимодействие между сервисами через API.

Asyncio предоставляет инструменты для создания высокопроизводительных приложений, особенно там, где требуется асинхронная обработка ввода-вывода. Используя подходы, представленные в примерах, можно разработать масштабируемые и эффективные программы для самых разных задач.

Задачи для практики

Задача 1: Асинхронный веб-сканер для проверки статуса сайтов

Условие: Разработайте программу, которая принимает список URL-адресов, проверяет их доступность и возвращает HTTP-статус каждого сайта. Проверка должна выполняться параллельно.

Решение:

```python
import asyncio
import aiohttp
async def fetch_status(session, url):
try:
async with session.get(url) as response:
print(f"{url}: {response.status}")
except Exception as e:
print(f"{url}: Ошибка {e}")
async def main():
urls = [
```

"https://google.com",
"https://github.com",
"https://python.org",
"https://nonexistent.url"
]
async with aiohttp.ClientSession() as session:
tasks = [fetch_status(session, url) for url in urls]
await asyncio.gather(*tasks)
asyncio.run(main())
```

Результат

Код проверяет статусы всех URL, при этом выполнение задач происходит параллельно. Например:

```
https://google.com: 200
https://github.com: 200
https://python.org: 200
https://nonexistent.url: Ошибка ClientConnectorError
```

Задача 2: Асинхронная обработка большого файла

Условие: Разбейте большой текстовый файл на строки и параллельно подсчитайте количество слов в каждой строке. Затем выведите общее количество слов.

Решение:

```python
import asyncio
async def count_words_in_line(line):
await asyncio.sleep(0.1) # Симуляция долгой обработки
return len(line.split())
async def process_file(file_path):
total_words = 0
tasks = []
with open(file_path, "r") as file:
for line in file:
tasks.append(count_words_in_line(line))
results = await asyncio.gather(*tasks)
total_words = sum(results)
```
```

print(f"Общее количество слов: {total_words}")
# Пример использования
asyncio.run(process_file("large_file.txt"))
```

Результат:

Код читает файл `large_file.txt`, параллельно считает количество слов в каждой строке и выводит итоговое количество.

**Задача 3: Асинхронный обработчик API с ограничением на количество запросов**

Условие: Создайте систему, которая обращается к API для загрузки данных, но ограничивает количество одновременных запросов до 3.

Решение:

```python
import asyncio
import aiohttp
async def fetch_data(session, url):
async with session.get(url) as response:
data = await response.json()
print(f"Данные с {url}: {data}")
return data
async def main():
urls = [
"https://jsonplaceholder.typicode.com/posts/1",
"https://jsonplaceholder.typicode.com/posts/2",
"https://jsonplaceholder.typicode.com/posts/3",
"https://jsonplaceholder.typicode.com/posts/4",
"https://jsonplaceholder.typicode.com/posts/5",
]
semaphore = asyncio.Semaphore(3) # Ограничение на 3 одновременных запроса
async def limited_fetch(url):
async with semaphore:
async with aiohttp.ClientSession() as session:
return await fetch_data(session, url)
tasks = [limited_fetch(url) for url in urls]
await asyncio.gather(*tasks)
```
```

asyncio.run(main())
```

Результат:

Код одновременно обрабатывает только 3 запроса, несмотря на наличие 5 URL. Это полезно при работе с API, имеющими ограничения на количество запросов.

**Задача 4: Асинхронная очередь задач с приоритетами**

Условие: Реализуйте систему, которая обрабатывает задачи с разным приоритетом. Высокоприоритетные задачи должны выполняться раньше.

Решение:

```python
import asyncio
import heapq
class PriorityQueue:
def __init__(self):
self._queue = []
self._counter = 0
def put(self, priority, task):
heapq.heappush(self._queue, (priority, self._counter, task))
self._counter += 1
def get(self):
return heapq.heappop(self._queue)[-1]
def empty(self):
return len(self._queue) == 0
async def worker(name, queue):
while not queue.empty():
task = queue.get()
print(f"{name} выполняет задачу: {task}")
await asyncio.sleep(1)
async def main():
queue = PriorityQueue()
queue.put(2, "Средний приоритет")
queue.put(1, "Высокий приоритет")
queue.put(3, "Низкий приоритет")
```
```

await asyncio.gather(worker("Рабочий 1", queue), worker("Рабочий 2", queue))
asyncio.run(main())
```
Результат:
Код обеспечивает выполнение задач в порядке их приоритетов:
```
Рабочий 1 выполняет задачу: Высокий приоритет
Рабочий 2 выполняет задачу: Средний приоритет
Рабочий 1 выполняет задачу: Низкий приоритет
```
**Задача 5: Асинхронная обработка загрузки файлов с прогрессом**
Условие: Напишите программу, которая загружает несколько файлов параллельно и отображает прогресс загрузки.
Решение:
```python
import asyncio
import aiohttp
from tqdm.asyncio import tqdm
async def download_file(session, url, filename):
async with session.get(url) as response:
with open(filename, "wb") as f:
async for chunk in response.content.iter_chunked(1024):
f.write(chunk)
print(f"{filename} загружен")
async def main():
urls = [
("https://www.example.com/file1", "file1.txt"),
("https://www.example.com/file2", "file2.txt"),
("https://www.example.com/file3", "file3.txt"),
]
async with aiohttp.ClientSession() as session:
tasks = [download_file(session, url, filename) for url, filename in urls]
for task in tqdm(asyncio.as_completed(tasks), total=len(tasks)):
await task
asyncio.run(main())
```
```

Результат:

Программа загружает файлы параллельно и отображает прогресс с помощью `tqdm`.

Эти примеры помогут углубить понимание Asyncio, продемонстрировав его применение в реальных задачах, таких как обработка данных, взаимодействие с API и управление очередями задач.

5.2 Параллельные вычисления с использованием multiprocessing в Python (версия 5.2)

Параллельные вычисления – это способ выполнения нескольких операций одновременно для повышения производительности программ. Python предоставляет модуль `multiprocessing` для работы с процессами. Этот модуль полезен для выполнения задач, которые могут быть распределены на несколько ядер процессора.

Ключевые концепции модуля `multiprocessing`

1. Процессы (`Process`). Основной объект для запуска параллельного выполнения кода. Каждый процесс имеет свою память, что предотвращает конфликты, но требует межпроцессного взаимодействия.
2. Очереди и каналы (`Queue`, `Pipe`). Используются для обмена данными между процессами.
3. Пулы процессов (`Pool`). Позволяют управлять группами процессов, упрощая их запуск и синхронизацию.
4. События, замки и семафоры. Обеспечивают синхронизацию между процессами.

Шаги работы с `multiprocessing`

1. Создание и запуск процесса
2. Обмен данными между процессами
3. Использование пула процессов
4. Синхронизация процессов

Пример 1: Создание и запуск процесса

```python
```

```
from multiprocessing import Process
import os
# Функция, которая будет выполняться в процессе
def worker_function(name):
print(f"Привет, {name}! Это процесс с PID: {os.getpid()}")
if __name__ == "__main__":
# Создаем процесс
process = Process(target=worker_function, args=("Мир",))
# Запускаем процесс
process.start()
# Ожидаем завершения процесса
process.join()
```

Вывод:

```
Привет, Мир! Это процесс с PID: 12345
```

Пример 2: Использование очереди для обмена данными

```python
from multiprocessing import Process, Queue
def worker_function(queue, number):
result = number * number
queue.put(result) # Отправляем результат в очередь
if __name__ == "__main__":
queue = Queue()
processes = []
# Создаем и запускаем процессы
for i in range(5):
process = Process(target=worker_function, args=(queue, i))
processes.append(process)
process.start()
# Ожидаем завершения процессов
for process in processes:
process.join()
# Считываем результаты из очереди
while not queue.empty():
print(queue.get())
```

```
Вывод:
```
0
1
4
9
16
```
**Пример 3: Использование пула процессов**
```python
from multiprocessing import Pool
def square_function(number):
return number * number
if __name__ == "__main__":
with Pool(processes=4) as pool:
numbers = [0, 1, 2, 3, 4, 5]
results = pool.map(square_function, numbers)
print(results)
```
Вывод:
```
[0, 1, 4, 9, 16, 25]
```
Пример 4: Синхронизация с использованием замка
```python
from multiprocessing import Process, Lock
def worker_function(lock, name):
with lock: # Гарантируем доступ только одному процессу
print(f"Привет от {name}")
if __name__ == "__main__":
lock = Lock()
processes = []
for i in range(5):
process = Process(target=worker_function, args=(lock, f"Процесс {i}"))
processes.append(process)
process.start()
```
```

```
for process in processes:
process.join()
```
Вывод (последовательный вывод приветствий):
```
Привет от Процесс 0
Привет от Процесс 1
Привет от Процесс 2
Привет от Процесс 3
Привет от Процесс 4
```
Пример 5: Использование `Value` и `Array` для общих данных
```python
from multiprocessing import Process, Value, Array
def increment(shared_value, shared_array):
shared_value.value += 1
for i in range(len(shared_array)):
shared_array[i] += 1
if __name__ == "__main__":
value = Value('i', 0) # Общая переменная (целое число)
array = Array('i', [1, 2, 3]) # Общий массив
processes = []
for _ in range(3):
process = Process(target=increment, args=(value, array))
processes.append(process)
process.start()
for process in processes:
process.join()
print(f"Значение: {value.value}")
print(f"Массив: {list(array)}")
```
Вывод:
```
Значение: 3
Массив: [4, 5, 6]
```
Пример 6: Обработка задач с помощью пула потоков

```python
from multiprocessing import Pool
def worker_function(x):
return x ** 2
if __name__ == "__main__":
numbers = [1, 2, 3, 4, 5]
with Pool(3) as pool: # 3 параллельных процесса
results = pool.map(worker_function, numbers)
print("Результаты:", results)
```

Вывод:

```
Результаты: [1, 4, 9, 16, 25]
```

Ключевые моменты при работе с `multiprocessing`

1. Избегайте гонок данных. Используйте механизмы синхронизации, такие как `Lock` и `Semaphore`.
2. Учитывайте накладные расходы. Создание процессов занимает время и ресурсы, поэтому для небольших задач `multiprocessing` может быть избыточным.
3. Работайте с безопасным входом в программу. Всегда оборачивайте код в `if __name__ == "__main__":`, чтобы избежать повторного создания процессов.
4. Отлаживайте параллельные программы. Логирование (`logging`) вместо `print` может упростить отладку.

Задачи для практики

Несколько интересных задач с решениями для главы о параллельных вычислениях с использованием модуля `multiprocessing`. Эти задачи помогут закрепить теорию на практике и понять, как использовать возможности параллелизма в реальных сценариях.

Задача 1: Подсчет количества простых чисел в массиве

Условие: Дан список чисел. Нужно найти количество простых чисел, используя параллельные вычисления.

```python
from multiprocessing import Pool
import math
def is_prime(n):
if n < 2:
return False
for i in range(2, int(math.sqrt(n)) + 1):
if n % i == 0:
return False
return True
if __name__ == "__main__":
numbers = list(range(1, 100000)) # Проверяем числа от 1 до 100000
with Pool(processes=4) as pool: # 4 параллельных процесса
results = pool.map(is_prime, numbers)
prime_count = sum(results)
print(f"Количество простых чисел: {prime_count}")
```

Вывод (примерный):

```
Количество простых чисел: 9592
```

Задача 2: Параллельная обработка изображений

Условие: У вас есть список изображений. Нужно перевести их в оттенки серого, используя параллельные процессы.

```python
from multiprocessing import Pool
from PIL import Image
import os
def process_image(image_path):
img = Image.open(image_path).convert("L") # Перевод в оттенки серого
output_path = f"gray_{os.path.basename(image_path)}"
img.save(output_path)
return output_path
if __name__ == "__main__":
```

image_files = ["image1.jpg", "image2.jpg", "image3.jpg"] # Замените на реальные имена файлов
with Pool(processes=3) as pool: # 3 параллельных процесса
results = pool.map(process_image, image_files)
print(f"Обработанные изображения: {results}")
```

Вывод:
```
Обработанные изображения: ['gray_image1.jpg', 'gray_image2.jpg', 'gray_image3.jpg']
```

**Задача 3: Поиск минимального элемента в большом массиве**

Условие: Найдите минимальный элемент в массиве из 10 миллионов чисел, разделив задачу между несколькими процессами.

```python
from multiprocessing import Pool
import random
def find_min(sublist):
return min(sublist)
if __name__ == "__main__":
# Генерируем массив из 10 миллионов случайных чисел
numbers = [random.randint(0, 1000000) for _ in range(10_000_000)]
num_processes = 4 # Количество процессов
chunk_size = len(numbers) // num_processes
# Разделяем массив на части
chunks = [numbers[i:i + chunk_size] for i in range(0, len(numbers), chunk_size)]
with Pool(processes=num_processes) as pool:
results = pool.map(find_min, chunks)
# Ищем минимальное значение среди результатов
overall_min = min(results)
print(f"Минимальное значение: {overall_min}")
```

Вывод (примерный):
```
Минимальное значение: 0
```
```

Задача 4: Имитация скачивания файлов

Условие: Есть список URL-адресов. Имитация "скачивания" файлов (например, задержка) должна выполняться параллельно.

```python
from multiprocessing import Pool
import time
def download_file(file_url):
print(f"Скачивание {file_url} начато")
time.sleep(2) # Имитация задержки (например, скачивание)
print(f"Скачивание {file_url} завершено")
return f"{file_url} – завершено"
if __name__ == "__main__":
file_urls = ["file1.txt", "file2.txt", "file3.txt", "file4.txt"]
with Pool(processes=2) as pool: # 2 параллельных процесса
results = pool.map(download_file, file_urls)
print("Все файлы скачаны:")
print(results)
```

Вывод:

```
Скачивание file1.txt начато
Скачивание file2.txt начато
Скачивание file1.txt завершено
Скачивание file3.txt начато
Скачивание file2.txt завершено
Скачивание file4.txt начато
Скачивание file3.txt завершено
Скачивание file4.txt завершено
Все файлы скачаны:
['file1.txt – завершено', 'file2.txt – завершено', 'file3.txt – завершено', 'file4.txt – завершено']
```

Задача 5: Умножение больших матриц

Условие: Реализуйте параллельное умножение двух квадратных матриц размером 1000x1000.

```python
from multiprocessing import Pool
```

```
import numpy as np
def multiply_row(args):
row, matrix_b = args
return [sum(a * b for a, b in zip(row, col)) for col in zip(*matrix_b)]
if __name__ == "__main__":
size = 1000 # Размер матрицы
matrix_a = np.random.randint(0, 10, (size, size))
matrix_b = np.random.randint(0, 10, (size, size))
with Pool(processes=4) as pool: # 4 параллельных процесса
result = pool.map(multiply_row, [(row, matrix_b) for row in matrix_a])
# Преобразуем результат в массив numpy
result_matrix = np.array(result)
print("Умножение матриц завершено.")
```

Вывод:

```
Умножение матриц завершено.
```

Задача 6: Поиск самого часто встречающегося слова в большом тексте

Условие: Дан большой текстовый файл. Найдите самое часто встречающееся слово, используя параллельные вычисления.

```python
from multiprocessing import Pool
from collections import Counter
def word_count(chunk):
words = chunk.split()
return Counter(words)
if __name__ == "__main__":
with open("large_text.txt", "r") as file:
text = file.read()
num_processes = 4
chunk_size = len(text) // num_processes
chunks = [text[i:i + chunk_size] for i in range(0, len(text), chunk_size)]
with Pool(processes=num_processes) as pool:
results = pool.map(word_count, chunks)
# Суммируем результаты всех процессов
```

```
total_count = sum(results, Counter())
most_common_word = total_count.most_common(1)[0]
print(f"Самое часто встречающееся слово: {most_common_word}")
```

Вывод (примерный):

```
Самое часто встречающееся слово: ('the', 1543)
```

Эти задачи демонстрируют широкий спектр применения модуля `multiprocessing` в реальных сценариях: от математических расчетов до обработки данных и файлов. Попробуйте реализовать их самостоятельно, чтобы лучше понять, как использовать параллельные вычисления в Python!

5.3 Ускорение вычислений через CUDA с библиотекой CuPy

Библиотека CuPy позволяет использовать вычислительные мощности графического процессора (GPU) для ускорения научных вычислений. Она предоставляет интерфейс, схожий с NumPy, и позволяет прозрачно переносить вычисления с CPU на GPU. CuPy поддерживает обработку массивов, линейную алгебру, работу с FFT (быстрое преобразование Фурье) и многими другими численными методами.

Как работает CuPy?

CuPy – это библиотека, которая позволяет переносить численные вычисления на графический процессор (GPU), используя технологию CUDA от NVIDIA. Она организует взаимодействие между вашим Python-кодом и графической картой, что обеспечивает значительное ускорение вычислений, особенно для задач с большими массивами данных, линейной алгеброй, преобразованием Фурье и многими другими численными операциями.

Принцип работы CuPy:

– Массивы создаются и хранятся в памяти GPU.

– Операции выполняются непосредственно на графическом процессоре, без необходимости перемещения данных в оперативную память CPU.

Подготовка к работе с CuPy.

Чтобы использовать CuPy, необходимо установить библиотеку и убедиться, что на компьютере установлен CUDA Toolkit и поддерживается GPU от NVIDIA.

Для установки CuPy выполните команду:

```bash
pip install cupy-cuda11x
```

(где `11x` – это версия CUDA, подходящая для вашей системы). Убедитесь, что версия CUDA на вашем GPU совпадает с установленной.

Основные аспекты работы CuPy

1. Графический процессор и CUDA

– Графический процессор (GPU) оптимизирован для параллельной обработки. Он может одновременно выполнять тысячи потоков, что делает его идеальным для задач, требующих однородных вычислений (например, работа с массивами).

– CUDA (Compute Unified Device Architecture) – это платформа NVIDIA, которая позволяет разработчикам использовать GPU для общих вычислений, а не только для графических задач. CuPy использует CUDA для передачи данных на видеокарту и выполнения операций.

2. Устройство хранения данных в CuPy

– Массивы CuPy (`cupy.ndarray`) хранятся в памяти видеокарты (GPU Memory), что делает их недоступными для прямого доступа из NumPy.

– Операции, выполняемые над массивами CuPy, происходят полностью на GPU, минимизируя необходимость передачи данных между центральным процессором (CPU) и GPU. Это позволяет добиться высокой скорости обработки данных.

3. Архитектура CuPy

CuPy предоставляет интерфейс, практически идентичный NumPy. Это значит, что многие функции NumPy имеют свои аналоги в CuPy. Например:

– В NumPy: `numpy.sum()`, `numpy.dot()`, `numpy.linalg.inv()`.

– В CuPy: `cupy.sum()`, `cupy.dot()`, `cupy.linalg.inv()`.

Однако, за кулисами CuPy вместо обычных операций использует CUDA-ядра (kernels), которые запускаются на GPU.

4. Взаимодействие с памятью GPU и CPU

– Для передачи данных с CPU на GPU используется функция `cp.asarray()`. Она преобразует массив NumPy (`numpy.ndarray`) в массив CuPy (`cupy.ndarray`), помещая его в память GPU.

– Чтобы вернуть данные обратно на CPU, используется функция `cp.asnumpy()`, которая копирует массив из памяти GPU в память CPU.

Пример:

```python
import cupy as cp
import numpy as np
# Создаем массив NumPy на CPU
cpu_array = np.array([1, 2, 3])
# Перемещаем массив на GPU
gpu_array = cp.asarray(cpu_array)
# Выполняем операции на GPU
result_gpu = gpu_array * 2
# Возвращаем результат на CPU
result_cpu = cp.asnumpy(result_gpu)
print(result_cpu) # [2 4 6]
```

5. Синхронизация и асинхронность*

– GPU операции в CuPy по умолчанию асинхронны. Это значит, что выполнение операций может продолжаться, пока данные обрабатываются на GPU.

– Для корректного измерения времени или передачи данных необходимо синхронизировать выполнение потоков GPU и CPU. Это делается с помощью команды `cp.cuda.Stream.null.synchronize()`.

Пример:

```python
start_time = time.time()
```

```
result = cp.sum(gpu_array)
cp.cuda.Stream.null.synchronize() # Ожидание завершения всех операций на GPU
end_time = time.time()
print(f"Время выполнения: {end_time – start_time:.2f} секунд")
```

6. Обработка больших массивов и вычислительные ядра

CuPy использует CUDA-ядра, оптимизированные для различных математических операций. Это позволяет эффективно выполнять задачи, такие как:

– Умножение матриц (`cp.dot()`).

– Быстрое преобразование Фурье (`cp.fft.fft()`).

– Решение систем линейных уравнений (`cp.linalg.solve()`).

Пример работы CuPy с большими данными

Рассмотрим пример, в котором CuPy используется для вычисления суммы элементов в массиве размером 100 миллионов.

```python
import numpy as np
import cupy as cp
import time
# Размер массива
n = 100_000_000
# Генерация данных на CPU
cpu_array = np.random.rand(n).astype(np.float32)
# Генерация данных на GPU
gpu_array = cp.random.rand(n, dtype=cp.float32)
# Вычисление суммы на CPU
start_time = time.time()
cpu_sum = np.sum(cpu_array)
end_time = time.time()
print(f"Сумма на CPU: {cpu_sum:.2f}, время: {end_time – start_time:.4f} секунд")
# Вычисление суммы на GPU
start_time = time.time()
gpu_sum = cp.sum(gpu_array)
```

cp.cuda.Stream.null.synchronize() # Синхронизация для корректного замера времени
end_time = time.time()
print(f"Сумма на GPU: {gpu_sum:.2f}, время: {end_time – start_time:.4f} секунд")
```

Объяснение примера:

– На GPU операция выполняется намного быстрее за счет параллельной обработки элементов массива.

– CuPy позволяет использовать ту же логику кода, что и NumPy, но с ускорением за счет CUDA.

**Преимущества CuPy**

1. Простота использования: CuPy практически полностью повторяет интерфейс NumPy, что снижает порог вхождения.
2. Высокая производительность: Использование CUDA и GPU позволяет ускорить вычисления в десятки раз.
3. Асинхронность: Операции на GPU выполняются параллельно, что дополнительно повышает производительность.
4. Поддержка научных вычислений: Поддерживаются такие задачи, как линейная алгебра, работа с FFT, интерполяция и другие.

**Когда использовать CuPy?**

CuPy особенно полезен в следующих случаях:

– Обработка больших массивов данных (миллионы элементов).

– Задачи линейной алгебры (умножение матриц, вычисление обратной матрицы и т. д.).

– Интенсивные вычисления, такие как машинное обучение, симуляции или моделирование.

– Сценарии, где важно ускорить работу за счет параллелизма GPU.

Использование CuPy – это один из лучших способов ускорить численные вычисления, если у вас есть доступ к мощному GPU с поддержкой CUDA.

**Пример: Сравнение вычислений на CPU (NumPy) и GPU (CuPy)**

Рассмотрим задачу умножения двух больших матриц.

```python

```
import numpy as np
import cupy as cp
import time
# Размер матрицы
n = 5000
# Генерация данных на CPU
matrix_a_cpu = np.random.rand(n, n).astype(np.float32)
matrix_b_cpu = np.random.rand(n, n).astype(np.float32)
# Умножение матриц на CPU
start_time = time.time()
result_cpu = np.dot(matrix_a_cpu, matrix_b_cpu)
end_time = time.time()
print(f"Время выполнения на CPU: {end_time – start_time:.2f} секунд")
# Генерация данных на GPU (CuPy)
matrix_a_gpu = cp.random.rand(n, n, dtype=cp.float32)
matrix_b_gpu = cp.random.rand(n, n, dtype=cp.float32)
# Умножение матриц на GPU
start_time = time.time()
result_gpu = cp.dot(matrix_a_gpu, matrix_b_gpu)
cp.cuda.Stream.null.synchronize() # Синхронизация для корректного замера времени
end_time = time.time()
print(f"Время выполнения на GPU: {end_time – start_time:.2f} секунд")
# Проверка совпадения результатов
result_cpu_cp = cp.asarray(result_cpu) # Преобразование результата CPU в формат CuPy
difference = cp.mean(cp.abs(result_gpu – result_cpu_cp))
print(f"Средняя разница между результатами: {difference:.8f}")
```

Объяснение примера:

1. Генерация матриц:

Используются случайные матрицы размера `5000x5000` на CPU с помощью NumPy и на GPU с помощью CuPy.

Данные на GPU хранятся в памяти видеокарты, а не в оперативной памяти.

2. Умножение:

– `np.dot()` выполняет вычисления на CPU.

– `cp.dot()` выполняет вычисления на GPU, используя мощности CUDA.

3. Замер времени: Мы фиксируем время начала и окончания выполнения операции, чтобы сравнить производительность.

4. Проверка результата: Результат с GPU переводится обратно в формат NumPy (`cp.asarray`), и вычисляется средняя разница между результатами, чтобы подтвердить их совпадение.

Вывод:

На современных видеокартах ускорение может быть в 10-50 раз быстрее, чем выполнение аналогичных операций на CPU, особенно для задач с большими объемами данных.

Дополнительные функции CuPy:

– Поддержка операций, аналогичных NumPy: `cp.sum()`, `cp.mean()`, `cp.linalg.inv()`.

– Перемещение массивов между CPU и GPU:

– `cp.asarray()` – перевод данных из NumPy в CuPy.

– `cp.asnumpy()` – перевод данных из CuPy в NumPy.

– Ускорение вычислений в других библиотеках, таких как PyTorch и TensorFlow, для задач машинного обучения.

Практическое применение: CuPy идеально подходит для задач линейной алгебры, обработки больших данных, численных симуляций и любого рода вычислений, которые требуют высокой производительности. Это мощный инструмент для тех, кто работает с глубоким обучением, физическими симуляциями или научными расчетами.

Глава 6. Обработка мультимедиа

6.1 Работа с изображениями с OpenCV

OpenCV (Open Source Computer Vision Library) – это популярная библиотека с открытым исходным кодом для обработки изображений и компьютерного зрения. Она предоставляет мощные инструменты для работы с изображениями, включая их загрузку, преобразование, фильтрацию, обнаружение объектов и многое другое. OpenCV написана на C++, но имеет удобный интерфейс для Python, что делает её отличным выбором для разработки проектов обработки изображений.

Для начала работы с OpenCV в Python необходимо установить библиотеку. Это можно сделать с помощью команды:

```bash
pip install opencv-python
```

Пример обработки изображений включает следующие задачи: чтение изображения, изменение его размеров, преобразование в оттенки серого, применение фильтров, а также сохранение результатов.

```python
import cv2
import numpy as np
# Шаг 1: Загрузка изображения
# Загрузим изображение из файла. Убедитесь, что в рабочей директории есть изображение "example.jpg".
image = cv2.imread("example.jpg")
# Проверка успешной загрузки изображения
if image is None:
print("Ошибка: изображение не удалось загрузить.")
else:
print(f"Изображение успешно загружено. Размеры: {image.shape}")
# Шаг 2: Отображение исходного изображения
```

```
cv2.imshow("Оригинальное изображение", image)
cv2.waitKey(0) # Ожидание нажатия клавиши
cv2.destroyAllWindows()
# Шаг 3: Изменение размера изображения
resized_image = cv2.resize(image, (300, 300)) # Изменение размера до 300x300 пикселей
cv2.imshow("Измененное изображение", resized_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
# Шаг 4: Преобразование в оттенки серого
gray_image = cv2.cvtColor(resized_image, cv2.COLOR_BGR2GRAY)
cv2.imshow("Оттенки серого", gray_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
# Шаг 5: Применение фильтра размытия
blurred_image = cv2.GaussianBlur(gray_image, (7, 7), 0) # Применение размытия с ядром 7x7
cv2.imshow("Размытое изображение", blurred_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
# Шаг 6: Обнаружение границ с использованием оператора Canny
edges = cv2.Canny(blurred_image, 50, 150) # Пороговые значения для границ
cv2.imshow("Обнаружение границ", edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
# Шаг 7: Сохранение результата
cv2.imwrite("result.jpg", edges)
print("Результат сохранён в файл result.jpg")
```

В данном примере решается сразу несколько задач. Сначала изображение загружается в формате BGR (стандартный формат OpenCV). После этого оно преобразуется в различные формы: уменьшается до заданных размеров, переводится в оттенки серого, подвергается размытию и обработке с целью выявления границ. Границы определяются с помощью метода Canny, который выделяет

контуры объектов на изображении. Результирующее изображение сохраняется на диск.

Основные моменты работы с OpenCV включают управление окнами для отображения результатов (`cv2.imshow`), применение различных фильтров (`cv2.GaussianBlur`) и преобразований (например, `cv2.cvtColor` для перевода изображения в другой цветовой формат).

Пример легко адаптируется для выполнения других задач, таких как преобразование яркости, наложение текстов и форм, а также применение сложных операций, например, обнаружение лиц или объектов. OpenCV обеспечивает простоту выполнения этих операций, предоставляя мощный инструмент для визуальных и мультимедийных задач.

Задачи для практики

Задача 1. Поворот изображения на заданный угол

Описание: Пользователь вводит угол поворота (например, 45°). Необходимо повернуть изображение на этот угол и показать результат.

Решение:

```python
import cv2
import numpy as np
# Загрузка изображения
image = cv2.imread("example.jpg")
if image is None:
print("Ошибка загрузки изображения")
else:
# Получение размеров изображения
height, width = image.shape[:2]
# Запрос угла поворота у пользователя
angle = float(input("Введите угол поворота: "))
# Вычисление матрицы поворота
center = (width // 2, height // 2)
rotation_matrix = cv2.getRotationMatrix2D(center, angle, 1)
# Поворот изображения
```

```
rotated_image = cv2.warpAffine(image, rotation_matrix, (width, height))
# Отображение результата
cv2.imshow("Оригинальное изображение", image)
cv2.imshow(f"Повернутое на {angle} градусов", rotated_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Задача 2. Добавление текста на изображение

Описание: На изображении необходимо отобразить текст, например, "OpenCV Задача". Текст должен быть центрирован по изображению.

Решение:

```python
import cv2
# Загрузка изображения
image = cv2.imread("example.jpg")
if image is None:
print("Ошибка загрузки изображения")
else:
# Текст и его параметры
text = "OpenCV Задача"
font = cv2.FONT_HERSHEY_SIMPLEX
font_scale = 2
font_color = (0, 255, 0) # Зеленый
thickness = 3
# Вычисление размера текста
text_size = cv2.getTextSize(text, font, font_scale, thickness)[0]
text_x = (image.shape[1] – text_size[0]) // 2
text_y = (image.shape[0] + text_size[1]) // 2
# Наложение текста
image_with_text = image.copy()
cv2.putText(image_with_text, text, (text_x, text_y), font, font_scale, font_color, thickness)
# Отображение результата
cv2.imshow("Изображение с текстом", image_with_text)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Задача 3. Создание фильтра негатив

Описание: Необходимо преобразовать изображение в его негатив, инвертируя цвета.

Решение:

```python
import cv2
# Загрузка изображения
image = cv2.imread("example.jpg")
if image is None:
print("Ошибка загрузки изображения")
else:
# Создание негатива
negative_image = 255 – image
# Отображение результата
cv2.imshow("Оригинальное изображение", image)
cv2.imshow("Негатив", negative_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Задача 4. Разделение изображения на цветовые каналы

Описание: Разделить изображение на три канала: синий, зелёный и красный. Отобразить каждый канал отдельно.

Решение:

```python
import cv2
# Загрузка изображения
image = cv2.imread("example.jpg")
if image is None:
print("Ошибка загрузки изображения")
else:
# Разделение на каналы
blue_channel, green_channel, red_channel = cv2.split(image)
# Отображение каждого канала
cv2.imshow("Синий канал", blue_channel)
cv2.imshow("Зелёный канал", green_channel)
cv2.imshow("Красный канал", red_channel)
cv2.waitKey(0)
```

```
cv2.destroyAllWindows()
```

Задача 5. Преобразование изображения в черно-белое

Описание: Необходимо перевести изображение в черно-белый формат, применив пороговое преобразование (thresholding).

Решение:

```python
import cv2
# Загрузка изображения
image = cv2.imread("example.jpg")
if image is None:
print("Ошибка загрузки изображения")
else:
# Преобразование в оттенки серого
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Применение порогового преобразования
_, binary_image = cv2.threshold(gray_image, 127, 255, cv2.THRESH_BINARY)
# Отображение результата
cv2.imshow("Черно-белое изображение", binary_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Задача 6. Удаление шума с изображения

Описание: Использовать размытие (фильтрацию) для удаления шума на изображении.

Решение:

```python
import cv2
# Загрузка изображения
image = cv2.imread("example.jpg")
if image is None:
print("Ошибка загрузки изображения")
else:
# Применение размытия
blurred_image = cv2.GaussianBlur(image, (15, 15), 0) # Ядро 15x15
# Отображение результата
```

```
cv2.imshow("Оригинальное изображение", image)
cv2.imshow("Размытое изображение", blurred_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Задача 7. Масштабирование изображения

Описание: Увеличить или уменьшить изображение в два раза.

Решение:

```python
import cv2
# Загрузка изображения
image = cv2.imread("example.jpg")
if image is None:
print("Ошибка загрузки изображения")
else:
# Увеличение и уменьшение изображения
scaled_up = cv2.resize(image, None, fx=2.0, fy=2.0, interpolation=cv2.INTER_LINEAR)
scaled_down = cv2.resize(image, None, fx=0.5, fy=0.5, interpolation=cv2.INTER_LINEAR)
# Отображение результата
cv2.imshow("Оригинальное изображение", image)
cv2.imshow("Увеличенное изображение", scaled_up)
cv2.imshow("Уменьшенное изображение", scaled_down)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Задача 8. Обнаружение лица на изображении

Описание: Использовать встроенный классификатор Haar Cascade для обнаружения лица на изображении.

Решение:

```python
import cv2
# Загрузка классификатора
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_frontalface_default.xml")
# Загрузка изображения
```

```
image = cv2.imread("example.jpg")
if image is None:
print("Ошибка загрузки изображения")
else:
# Преобразование в оттенки серого
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
# Обнаружение лиц
faces = face_cascade.detectMultiScale(gray_image, scaleFactor=1.1, minNeighbors=5, minSize=(30, 30))
# Отображение результатов
for (x, y, w, h) in faces:
cv2.rectangle(image, (x, y), (x + w, y + h), (0, 255, 0), 3)
cv2.imshow("Обнаружение лица", image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Эти задачи помогут освоить базовые методы работы с изображениями в OpenCV и научат применять их на практике.

6.2 Обработка звука с Librosa

Librosa – это популярная библиотека Python для анализа и обработки звуковых данных. Она предоставляет мощный функционал для работы с аудио, включая загрузку звуковых файлов, извлечение признаков, преобразование сигналов и визуализацию. Эта библиотека активно используется в области обработки звука, машинного обучения и анализа музыки.

Для начала работы с Librosa необходимо установить библиотеку. Это можно сделать с помощью команды:

```bash
pip install librosa
```

Основная задача работы с аудиофайлами – это их загрузка, преобразование и анализ. Рассмотрим пример, где выполняется

загрузка звукового файла, его визуализация, изменение скорости воспроизведения и вычисление спектрограммы.

```python
import librosa
import librosa.display
import matplotlib.pyplot as plt
import numpy as np
# Загрузка аудиофайла
audio_file = 'example.wav' # Убедитесь, что файл example.wav находится в рабочей директории
y, sr = librosa.load(audio_file, sr=None) # y – звуковой сигнал, sr – частота дискретизации
# Информация о загруженном аудио
print(f"Частота дискретизации: {sr} Гц")
print(f"Длительность: {len(y) / sr:.2f} секунд")
# Визуализация звукового сигнала
plt.figure(figsize=(10, 4))
librosa.display.waveshow(y, sr=sr, alpha=0.8)
plt.title('Звуковой сигнал')
plt.xlabel('Время (с)')
plt.ylabel('Амплитуда')
plt.show()
# Изменение скорости воспроизведения (например, замедление)
y_slow = librosa.effects.time_stretch(y, rate=0.5) # Замедление в 2 раза
plt.figure(figsize=(10, 4))
librosa.display.waveshow(y_slow, sr=sr, alpha=0.8)
plt.title('Замедленный сигнал')
plt.xlabel('Время (с)')
plt.ylabel('Амплитуда')
plt.show()
# Вычисление спектрограммы
D = librosa.stft(y) # Кратковременное преобразование Фурье
S_db = librosa.amplitude_to_db(np.abs(D), ref=np.max) # Преобразование амплитуды в децибелы
# Визуализация спектрограммы
plt.figure(figsize=(10, 4))
```

```
librosa.display.specshow(S_db, sr=sr, x_axis='time', y_axis='hz', cmap='coolwarm')
plt.colorbar(format='%+2.0f dB')
plt.title('Спектрограмма')
plt.xlabel('Время (с)')
plt.ylabel('Частота (Гц)')
plt.tight_layout()
plt.show()
```

В этом примере сначала производится загрузка звукового файла, который преобразуется в массив амплитуд (массив `y`). Затем аудиосигнал визуализируется в виде графика амплитуды по времени. После этого применяется изменение скорости воспроизведения с использованием функции `librosa.effects.time_stretch`, что позволяет легко замедлять или ускорять звук. Для анализа частотного содержания звукового сигнала выполняется вычисление спектрограммы с использованием кратковременного преобразования Фурье (STFT). Спектрограмма отображается в децибелах, где оси показывают время и частоту, а цветовая шкала указывает интенсивность сигнала.

Библиотека Librosa предлагает широкий спектр инструментов для работы с аудиофайлами, включая извлечение музыкальных характеристик (таких как мел-частоты и тональность), обработку ритма и анализ спектральных данных. Этот пример демонстрирует только базовые возможности, которые можно расширить в зависимости от задач, стоящих перед вами.

Задачи для практики

Задача 1. Извлечение основной частоты (pitch detection)

Описание: Необходимо загрузить звуковой файл и определить основную частоту (pitch) сигнала в каждой точке времени. Это может быть полезно, например, при анализе мелодии или речи.

Решение:

```python
```

```
import librosa
import librosa.display
import matplotlib.pyplot as plt
import numpy as np
# Загрузка аудиофайла
audio_file = 'example.wav'
y, sr = librosa.load(audio_file, sr=None)
# Извлечение основной частоты
f0, voiced_flag, voiced_prob = librosa.pyin(y, fmin=librosa.note_to_hz('C2'), fmax=librosa.note_to_hz('C7'))
# Визуализация основной частоты
times = librosa.times_like(f0, sr=sr)
plt.figure(figsize=(10, 4))
plt.plot(times, f0, label='Основная частота (Hz)', color='r')
plt.xlabel('Время (c)')
plt.ylabel('Частота (Hz)')
plt.title('Извлечение основной частоты')
plt.legend()
plt.show()
```

Задача 2. Удаление шума из звукового файла

Описание: Загрузить звуковой файл, удалить фоновый шум с использованием фильтрации, и сохранить очищенный аудиофайл.

Решение:

```python
import librosa
import librosa.display
import numpy as np
import soundfile as sf
# Загрузка аудиофайла
audio_file = 'example.wav'
y, sr = librosa.load(audio_file, sr=None)
# Вычисление спектрограммы и определение порога для удаления шума
S = librosa.stft(y)
S_db = librosa.amplitude_to_db(np.abs(S), ref=np.max)
noise_threshold = np.median(S_db) – 15 # Примерный порог
```

```
# Удаление шума
S_db_cleaned = np.where(S_db > noise_threshold, S_db, noise_threshold)
S_cleaned = librosa.db_to_amplitude(S_db_cleaned)
# Обратное преобразование в звуковой сигнал
y_cleaned = librosa.istft(S_cleaned)
# Сохранение очищенного аудио
sf.write('cleaned_example.wav', y_cleaned, sr)
print("Очищенный файл сохранен как 'cleaned_example.wav'")
```

Задача 3. Ускорение или замедление аудио

Описание: Создайте программу, которая позволяет пользователю выбирать коэффициент ускорения или замедления аудиофайла.

Решение:

```python
import librosa
import librosa.display
import soundfile as sf
# Загрузка аудиофайла
audio_file = 'example.wav'
y, sr = librosa.load(audio_file, sr=None)
# Пользовательский ввод для скорости
rate = float(input("Введите коэффициент скорости (например, 0.5 для замедления, 2.0 для ускорения): "))
# Изменение скорости
y_stretched = librosa.effects.time_stretch(y, rate)
# Сохранение результата
sf.write('stretched_example.wav', y_stretched, sr)
print("Измененный файл сохранен как 'stretched_example.wav'")
```

Задача 4. Построение мел-спектрограммы аудио

Описание: Построить мел-спектрограмму загруженного аудиофайла и визуализировать её.

Решение:

```python
import librosa
import librosa.display
```

```
import matplotlib.pyplot as plt
# Загрузка аудиофайла
audio_file = 'example.wav'
y, sr = librosa.load(audio_file, sr=None)
# Вычисление мел-спектрограммы
S = librosa.feature.melspectrogram(y, sr=sr, n_mels=128, fmax=8000)
S_db = librosa.power_to_db(S, ref=np.max)
# Визуализация мел-спектрограммы
plt.figure(figsize=(10, 4))
librosa.display.specshow(S_db, x_axis='time', y_axis='mel', sr=sr, fmax=8000, cmap='viridis')
plt.colorbar(format='%+2.0f dB')
plt.title('Мел-спектрограмма')
plt.xlabel('Время (с)')
plt.ylabel('Частота (Mel)')
plt.tight_layout()
plt.show()
```

Задача 5. Определение темпа аудио

Описание: Определить темп (beats per minute, BPM) аудиофайла.

Решение:

```python
import librosa
# Загрузка аудиофайла
audio_file = 'example.wav'
y, sr = librosa.load(audio_file, sr=None)
# Определение темпа
tempo, _ = librosa.beat.beat_track(y, sr=sr)
print(f"Темп аудио: {tempo:.2f} BPM")
```

Задача 6. Аудиоэффект «Эхо"

Описание: Добавить к звуковому сигналу эффект эха с заданной задержкой.

Решение:

```python
import librosa
import numpy as np
```

```
import soundfile as sf
# Загрузка аудиофайла
audio_file = 'example.wav'
y, sr = librosa.load(audio_file, sr=None)
# Параметры эха
delay = 0.5 # Задержка в секундах
decay = 0.5 # Уровень затухания
# Преобразование задержки в сэмплы
delay_samples = int(delay * sr)
echo = np.zeros(len(y) + delay_samples)
echo[:len(y)] = y
echo[delay_samples:] += decay * y
# Нормализация
echo = echo / np.max(np.abs(echo))
# Сохранение результата
sf.write('echo_example.wav', echo, sr)
print("Файл с эффектом эха сохранен как 'echo_example.wav'")
```

Задача 7. Обнаружение начала и конца звука (silence trimming)

Описание: Удалить тишину в начале и конце аудиофайла.

Решение:

```python
import librosa
import soundfile as sf
# Загрузка аудиофайла
audio_file = 'example.wav'
y, sr = librosa.load(audio_file, sr=None)
# Удаление тишины
y_trimmed, _ = librosa.effects.trim(y, top_db=20)
# Сохранение результата
sf.write('trimmed_example.wav', y_trimmed, sr)
print("Файл с удалённой тишиной сохранен как 'trimmed_example.wav'")
```

Эти задачи охватывают базовые и продвинутые аспекты обработки аудио с использованием Librosa. Они помогут вам освоить

практическое применение библиотеки в реальных задачах.

6.3 Примеры создания видеоприложений с MoviePy

MoviePy – это мощная библиотека Python для редактирования и обработки видео. Она позволяет выполнять такие задачи, как нарезка, объединение, наложение текста и звука, добавление эффектов и многое другое. MoviePy особенно полезен для автоматизации обработки видео и создания небольших приложений, связанных с видео-редактированием.

Рассмотрим несколько ключевых операций, которые можно выполнить с помощью MoviePy, включая загрузку видео, обрезку, добавление текста и звука, а также создание нового видео с эффектами.

Для начала работы с библиотекой необходимо установить её с помощью команды:

```bash
pip install moviepy
```

Пример демонстрирует создание видео-приложения, которое выполняет несколько действий: обрезку видео, добавление текста и аудио, а также сохранение конечного результата.

```python
from moviepy.editor import VideoFileClip, TextClip, CompositeVideoClip, concatenate_videoclips
# Загрузка видеофайла
video_file = "example.mp4"
video_clip = VideoFileClip(video_file)
# Обрезка видео (например, первые 10 секунд)
start_time = 0 # Начало (в секундах)
end_time = 10 # Конец (в секундах)
trimmed_clip = video_clip.subclip(start_time, end_time)
# Добавление текста на видео
text = "Пример видео с MoviePy"
text_clip = TextClip(text, fontsize=50, color='white', bg_color='black')
```

```
text_clip = text_clip.set_position(("center", "bottom")).set_duration(trimmed_clip.duration)
# Создание нового видео с текстом
final_clip = CompositeVideoClip([trimmed_clip, text_clip])
# Сохранение результата
output_file = "output_video.mp4"
final_clip.write_videofile(output_file, codec="libx264", audio_codec="aac")
print(f"Видео успешно создано и сохранено как {output_file}")
```

В этом примере видеофайл загружается, после чего происходит обрезка первых 10 секунд с использованием метода `subclip`. Затем к видео добавляется текст, созданный с помощью класса `TextClip`. Текст накладывается поверх видео с помощью `CompositeVideoClip`, который объединяет различные клипы и элементы. Готовое видео сохраняется в файл `output_video.mp4`.

Кроме базовых операций, MoviePy позволяет применять эффекты, такие как ускорение или замедление, добавление переходов между клипами, фильтры и многое другое. Например, рассмотрим добавление звука и создание перехода между двумя видео.

```python
from moviepy.editor import AudioFileClip, VideoFileClip, concatenate_videoclips
# Загрузка видеофайлов
video1 = VideoFileClip("video1.mp4").subclip(0, 5) # Первые 5 секунд первого видео
video2 = VideoFileClip("video2.mp4").subclip(0, 5) # Первые 5 секунд второго видео
# Создание плавного перехода между видео
transition_duration = 1 # Длительность перехода в секундах
video2 = video2.crossfadein(transition_duration)
# Объединение видео с переходом
final_video = concatenate_videoclips([video1, video2], method="compose")
# Добавление звуковой дорожки
audio = AudioFileClip("background_music.mp3").subclip(0, final_video.duration)
```

```
final_video = final_video.set_audio(audio)
# Сохранение итогового видео
final_video.write_videofile("final_video_with_audio.mp4", codec="libx264", audio_codec="aac")
print("Финальное видео с переходом и звуком сохранено как 'final_video_with_audio.mp4'")
```

Этот пример демонстрирует, как можно объединять несколько видеоклипов с эффектами плавного перехода. Метод `crossfadein` создаёт эффект затемнения в начале второго видео. Кроме того, добавляется аудиофайл, который синхронизируется с длиной итогового видео.

MoviePy предоставляет гибкий и интуитивно понятный интерфейс для работы с видео и аудио, что делает его удобным инструментом для автоматизации задач обработки мультимедиа.

Задачи для практики

Задача 1. Обрезка видео и добавление текста

Описание: Загрузите видео, обрежьте его до первых 10 секунд, добавьте текст в центр экрана с указанием длительности, и сохраните итоговый файл.

Решение:

```python
from moviepy.editor import VideoFileClip, TextClip, CompositeVideoClip
# Шаг 1. Загрузка видео
video_clip = VideoFileClip("example.mp4")
# Шаг 2. Обрезка до первых 10 секунд
trimmed_clip = video_clip.subclip(0, 10)
# Шаг 3. Создание текста
text = "Это пример текста"
text_clip = TextClip(text, fontsize=50, color='white', bg_color='black')
text_clip = text_clip.set_duration(trimmed_clip.duration).set_position("center")
```

```
# Шаг 4. Наложение текста на видео
final_clip = CompositeVideoClip([trimmed_clip, text_clip])
# Шаг 5. Сохранение итогового видео
final_clip.write_videofile("trimmed_with_text.mp4", codec="libx264", audio_codec="aac")
print("Итоговое видео сохранено как 'trimmed_with_text.mp4'")
```

Объяснение:

– Функция `subclip` используется для обрезки видео на определённом временном отрезке.

– `TextClip` создаёт текстовую надпись, которая накладывается поверх видео с помощью `CompositeVideoClip`.

– Итоговый клип сохраняется в формате MP4 с заданными кодеками.

Задача 2. Ускорение видео

Описание: Ускорьте видео в два раза и сохраните его в новый файл.

Решение:

```python
from moviepy.editor import VideoFileClip
# Загрузка видео
video_clip = VideoFileClip("example.mp4")
# Ускорение видео в 2 раза
faster_clip = video_clip.fx(vfx.speedx, 2)
# Сохранение результата
faster_clip.write_videofile("faster_example.mp4", codec="libx264", audio_codec="aac")
print("Ускоренное видео сохранено как 'faster_example.mp4'")
```

Объяснение:

– Метод `vfx.speedx` ускоряет (или замедляет) воспроизведение видео. Коэффициент 2 означает удвоенную скорость.

– Видео сохраняется с оригинальным аудио, но скорость воспроизведения увеличивается.

Задача 3. Добавление водяного знака

Описание: Добавьте логотип или водяной знак в правый нижний угол видео.

Решение:

```python
from moviepy.editor import VideoFileClip, ImageClip, CompositeVideoClip
# Загрузка видео
video_clip = VideoFileClip("example.mp4")
# Загрузка изображения водяного знака
watermark = ImageClip("watermark.png").set_duration(video_clip.duration)
watermark = watermark.set_position(("right", "bottom")).resize(height=50)
# Наложение водяного знака
final_clip = CompositeVideoClip([video_clip, watermark])
# Сохранение результата
final_clip.write_videofile("video_with_watermark.mp4", codec="libx264", audio_codec="aac")
print("Видео с водяным знаком сохранено как 'video_with_watermark.mp4'")
```

Объяснение:

– `ImageClip` используется для загрузки изображения (водяного знака).

– Метод `set_position` устанавливает положение водяного знака (например, в правый нижний угол).

– С помощью `CompositeVideoClip` видео объединяется с водяным знаком, создавая итоговый ролик.

Задача 4. Объединение нескольких видеороликов

Описание: Объедините два видеоролика в один, чтобы они воспроизводились друг за другом.

Решение:

```python
from moviepy.editor import VideoFileClip, concatenate_videoclips
# Загрузка видеороликов
clip1 = VideoFileClip("video1.mp4").subclip(0, 5) # Первые 5 секунд
clip2 = VideoFileClip("video2.mp4").subclip(0, 5) # Первые 5 секунд
# Объединение клипов
```

```
final_clip = concatenate_videoclips([clip1, clip2], method="compose")
# Сохранение результата
final_clip.write_videofile("combined_video.mp4", codec="libx264", audio_codec="aac")
print("Объединённое видео сохранено как 'combined_video.mp4'")
```

Объяснение:

– Метод `concatenate_videoclips` соединяет клипы в единое видео, воспроизводя их друг за другом.

– Аргумент `method="compose"` используется для сохранения оригинальных разрешений и форматов клипов.

Задача 5. Добавление фоновой музыки

Описание: Добавьте звуковой файл в качестве фоновой музыки для видео.

Решение:

```python
from moviepy.editor import VideoFileClip, AudioFileClip
# Загрузка видео
video_clip = VideoFileClip("example.mp4")
# Загрузка аудиофайла
audio_clip = AudioFileClip("background_music.mp3").subclip(0, video_clip.duration)
# Замена звуковой дорожки
final_clip = video_clip.set_audio(audio_clip)
# Сохранение результата
final_clip.write_videofile("video_with_music.mp4", codec="libx264", audio_codec="aac")
print("Видео с фоновой музыкой сохранено как 'video_with_music.mp4'")
```

Объяснение:

– Метод `AudioFileClip` загружает аудиофайл, который затем накладывается на видео.

– `set_audio` заменяет звуковую дорожку видео на новую.

Задача 6. Создание GIF из видео

Описание: Создайте анимацию GIF из определённого отрезка видео.
Решение:
```python
from moviepy.editor import VideoFileClip
# Загрузка видео
video_clip = VideoFileClip("example.mp4").subclip(0, 5)
# Сохранение в формате GIF
video_clip.write_gif("example.gif", fps=10)
print("GIF сохранён как 'example.gif'")
```
Объяснение:
– Функция `write_gif` позволяет конвертировать видео в формат GIF.
– Аргумент `fps` задаёт частоту кадров, влияя на плавность анимации.

Задача 7. Применение эффекта "чёрно-белое видео"
Описание: Преобразуйте видео в чёрно-белый формат.
Решение:
```python
from moviepy.editor import VideoFileClip
from moviepy.video.fx.all import blackwhite
# Загрузка видео
video_clip = VideoFileClip("example.mp4")
# Применение эффекта чёрно-белого видео
bw_clip = blackwhite(video_clip)
# Сохранение результата
bw_clip.write_videofile("blackwhite_video.mp4", codec="libx264", audio_codec="aac")
print("Чёрно-белое видео сохранено как 'blackwhite_video.mp4'")
```
Объяснение:
– Функция `blackwhite` преобразует цветное видео в монохромное, сохраняя все остальные параметры неизменными.

Эти задачи охватывают основные и продвинутые функции MoviePy, включая редактирование видео, добавление эффектов, работу с аудио и объединение клипов. Каждая задача может быть дополнительно расширена в зависимости от ваших потребностей.

Глава 7. Тестирование и DevOps

7.1 Эффективное тестирование Python-кода с Pytest

Pytest – это одна из самых популярных библиотек для тестирования Python-кода. Она упрощает написание тестов и их выполнение, предлагая богатый набор инструментов для проверки кода, а также интеграцию с другими библиотеками и фреймворками.

Тестирование – важный процесс разработки программного обеспечения, который помогает обнаружить ошибки, регрессии и убедиться, что код работает должным образом. Для эффективного тестирования в Python часто используется Pytest, так как она обладает лаконичным синтаксисом, поддерживает различные подходы к тестированию и легко интегрируется в CI/CD пайплайны.

Основные концепции Pytest:

1. Тестовые функции и тестовые классы: Тесты в Pytest могут быть написаны как функции или методы классов. Однако чаще всего для простоты используются функции.
2. Ассерты: В тестах Pytest часто применяются утверждения (assertions), с помощью которых проверяется, соответствует ли результат ожиданиям.
3. Фикстуры: Фикстуры в Pytest предоставляют возможность для подготовки и очистки ресурсов, таких как базы данных или внешние сервисы, которые могут потребоваться для выполнения тестов.
4. Параметризация: Pytest позволяет параметризовать тесты, что даёт возможность запускать один и тот же тест с различными входными данными.

Установка Pytest

Чтобы начать использовать Pytest, нужно установить его через pip:

```bash
pip install pytest
```

Пример простого теста

Для начала создадим простой тест. Пусть у нас есть функция `add(a, b)`, которая складывает два числа:

```python
# my_module.py
def add(a, b):
return a + b
```

Теперь напишем тест для этой функции. Тесты принято размещать в файле, название которого начинается с `test_` или заканчивается на `_test.py`:

```python
# test_my_module.py
from my_module import add
def test_add():
assert add(1, 2) == 3
assert add(0, 0) == 0
assert add(-1, 1) == 0
```

В этом примере функция `test_add` проверяет, что функция `add` возвращает правильные результаты для различных входных значений.

Запуск тестов

Чтобы запустить тесты, достаточно выполнить команду:

```bash
pytest test_my_module.py
```

Pytest автоматически находит все тестовые функции, которые начинаются с `test_`, и выполняет их. Если все тесты пройдут успешно, вы получите сообщение, что все тесты прошли. Если какой-либо тест не прошёл, Pytest предоставит подробный отчёт о том, что пошло не так.

Использование фикстур

Фикстуры в Pytest позволяют автоматически выполнять код до и после выполнения тестов. Это полезно, например, для подготовки данных, настройки соединений с базой данных, или для очистки ресурсов после выполнения теста.

```python
# test_my_module.py
```

```
import pytest
from my_module import add
@pytest.fixture
def setup_data():
# Подготовка данных или создание объектов
a = 1
b = 2
return a, b
def test_add_with_fixture(setup_data):
a, b = setup_data
assert add(a, b) == 3
```

Здесь фикстура `setup_data` подготавливает данные, которые потом передаются в тестовую функцию `test_add_with_fixture`. Фикстуры могут быть использованы для многократного использования кода и улучшения читаемости тестов.

Параметризация тестов

Pytest также поддерживает параметризацию тестов, что позволяет запускать один и тот же тест с различными входными данными:

```python
# test_my_module.py
import pytest
from my_module import add
@pytest.mark.parametrize("a, b, expected", [
(1, 2, 3),
(0, 0, 0),
(-1, 1, 0)
])
def test_add_parametrized(a, b, expected):
assert add(a, b) == expected
```

Здесь используется декоратор `pytest.mark.parametrize`, который позволяет передать несколько наборов значений в одну тестовую функцию. В результате тест будет выполнен для каждого набора значений.

Запуск всех тестов

Чтобы запустить все тесты в проекте, достаточно вызвать команду:

```bash
pytest
```

Pytest будет искать все файлы, начинающиеся с `test_`, и запускать тесты из них. Это удобно, если вы хотите выполнить все тесты проекта без необходимости указывать конкретный файл.

Тестирование исключений

Pytest позволяет легко проверять, выбрасываются ли исключения в определённых случаях. Для этого используется контекстный менеджер `pytest.raises`:

```python
# my_module.py
def divide(a, b):
if b == 0:
raise ValueError("Cannot divide by zero")
return a / b
```

Теперь напишем тест для функции `divide`, который проверит, что исключение выбрасывается при делении на ноль:

```python
# test_my_module.py
import pytest
from my_module import divide
def test_divide_by_zero():
with pytest.raises(ValueError):
divide(1, 0)
```

Здесь тест проверяет, что при делении на ноль возникает исключение `ValueError`.

Различные опции запуска

Pytest предоставляет множество опций для настройки тестирования. Например, можно запустить тесты в определённом порядке, ограничить количество выводимых ошибок или выполнить тесты параллельно.

Вывод подробностей при неудаче:

```bash
pytest -v
```

```
Запуск только тестов, которые содержат определённое имя:
```bash
pytest -k "test_add"
```
Запуск тестов с определённым уровнем логирования:
```bash
pytest –maxfail=3
```
Эта команда завершит выполнение тестов после трёх неудач.

Интеграция с CI/CD

Pytest может быть легко интегрирован в пайплайны CI/CD, такие как Jenkins, Travis CI или GitHub Actions. Это позволяет автоматически запускать тесты каждый раз, когда происходит изменение в коде, обеспечивая высокое качество и стабильность продукта.

Пример использования в GitHub Actions:
```yaml
name: Python package
on: [push]
jobs:
test:
runs-on: ubuntu-latest
steps:
– name: Check out the repo
uses: actions/checkout@v2
– name: Set up Python
uses: actions/setup-python@v2
with:
python-version: '3.8'
– name: Install dependencies
run: |
python -m pip install –upgrade pip
pip install pytest
– name: Run tests
run: |
pytest
```
```

Этот конфигурационный файл запускает тесты при каждом push в репозиторий, обеспечивая непрерывную интеграцию.

Pytest – это мощная и гибкая библиотека для тестирования Python-кода. Она предлагает простоту в использовании, но при этом имеет множество дополнительных функций, таких как фикстуры, параметризация тестов и возможность тестирования исключений. С её помощью можно легко автоматизировать процесс тестирования, повысить качество кода и интегрировать тесты в CI/CD пайплайн.

7.2 Создание Docker-контейнеров для Python-приложений

Docker – это инструмент, который позволяет упаковывать приложения и все их зависимости в изолированные контейнеры, обеспечивая удобство в разработке, тестировании и развертывании. Контейнеры можно запускать на любых системах, что делает приложение независимым от операционной системы и окружения, в котором оно запускается.

Создание Docker-контейнера для Python-приложений – это процесс упаковки приложения в контейнер, чтобы оно могло быть выполнено в любом окружении с одинаковым результатом. С помощью Docker можно упростить развертывание и тестирование Python-приложений, обеспечивая их консистентность и воспроизводимость на разных системах.

Основные шаги создания Docker-контейнера для Python-приложения

1. Установка Docker: Прежде чем создавать контейнеры, необходимо установить Docker на вашу машину. Документация по установке доступна на официальном сайте Docker.

2. Создание Dockerfile: Это текстовый файл, который описывает, как должен быть построен Docker-образ для вашего Python-приложения. Dockerfile содержит инструкции, которые определяют, как контейнер будет построен, что должно быть установлено в контейнере, и какие команды будут выполнены при его запуске.

3. Создание Python-приложения: В качестве примера создадим простое Python-приложение, которое будет запускать HTTP-сервер и отдавать строку "Hello, Docker!".

4. Построение Docker-образа: Используя `Dockerfile`, мы можем создать Docker-образ, который затем можно будет запустить на любом сервере.

5. Запуск контейнера: После того как образ создан, мы можем запустить его, и наше приложение будет работать в контейнере.

Пример создания Docker-контейнера для Python-приложения

1. Шаг 1: Создание Python-приложения

Для примера создадим простое приложение на Python, которое будет запускать веб-сервер с использованием библиотеки Flask. Flask – это лёгкий фреймворк для создания веб-приложений.

Создадим файл `app.py`:

```python
# app.py
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello():
return "Hello, Docker!"
if __name__ == '__main__':
app.run(host='0.0.0.0', port=5000)
```

Это простое приложение использует Flask для запуска веб-сервера, который будет слушать на всех интерфейсах (host='0.0.0.0') и порту 5000. При обращении к корневому маршруту оно будет возвращать строку "Hello, Docker!".

2. Шаг 2: Создание Dockerfile

Теперь создадим файл `Dockerfile`. В этом файле будут описаны все шаги для сборки контейнера, включая установку зависимостей, копирование приложения в контейнер и запуск его.

Пример Dockerfile:

```dockerfile
# Используем официальный образ Python как базовый
FROM python:3.9-slim
```

```
# Устанавливаем рабочую директорию
WORKDIR /app
# Копируем файл с зависимостями (например, requirements.txt)
COPY requirements.txt .
# Устанавливаем зависимости
RUN pip install –no-cache-dir -r requirements.txt
# Копируем все файлы приложения в контейнер
COPY . .
# Открываем порт 5000 для подключения
EXPOSE 5000
# Указываем команду для запуска приложения
CMD ["python", "app.py"]
```

В этом `Dockerfile`:

– `FROM python:3.9-slim`: мы начинаем с базового образа Python 3.9 (легковесная версия).

– `WORKDIR /app`: устанавливаем рабочую директорию внутри контейнера.

– `COPY requirements.txt .`: копируем файл зависимостей (например, `requirements.txt`).

– `RUN pip install -r requirements.txt`: устанавливаем все необходимые зависимости.

– `COPY . .`: копируем все остальные файлы проекта (включая `app.py`) в контейнер.

– `EXPOSE 5000`: открываем порт 5000 для доступа к веб-приложению.

– `CMD ["python", "app.py"]`: указываем команду, которая будет выполнена при запуске контейнера (запуск приложения).

3. Шаг 3: Создание файла requirements.txt

Если ваше приложение зависит от внешних библиотек, таких как Flask, нужно создать файл `requirements.txt`, который будет содержать все зависимости, необходимые для работы приложения. Для нашего примера файл будет выглядеть так:

```text
Flask==2.0.1
```

4. Шаг 4: Сборка Docker-образа

Теперь, когда у нас есть файл `Dockerfile` и все необходимые файлы, мы можем собрать Docker-образ. Для этого в каталоге с проектом откроем командную строку и выполним команду:

```bash
docker build -t my-python-app .
```

– `docker build` – команда для сборки образа.

– `-t my-python-app` – флаг для указания имени образа (в данном случае `my-python-app`).

– `.` – точка, указывающая, что Dockerfile находится в текущем каталоге.

Docker начнёт процесс сборки, загружая базовый образ Python, устанавливая зависимости и копируя файлы приложения. После завершения вы получите образ `my-python-app`.

5. Шаг 5: Запуск Docker-контейнера

После того как образ собран, можно запустить контейнер, используя команду:

```bash
docker run -p 5000:5000 my-python-app
```

– `docker run` – команда для запуска контейнера.

– `-p 5000:5000` – флаг для проброса порта из контейнера на вашу машину. В данном случае мы пробрасываем порт 5000.

– `my-python-app` – имя образа, который мы создали на предыдущем шаге.

После выполнения этой команды контейнер будет запущен, и Flask-сервер будет слушать на порту 5000. Вы можете открыть браузер и перейти по адресу `http://localhost:5000`, чтобы увидеть сообщение "Hello, Docker!".

6. Шаг 6: Проверка работы приложения

После того как контейнер запущен, откройте браузер и перейдите по адресу `http://localhost:5000`. Вы должны увидеть строку "Hello, Docker!", которая является результатом работы вашего Flask-приложения, запущенного в Docker-контейнере.

Преимущества использования Docker для Python-приложений

1. Изоляция окружений: Docker позволяет создавать изолированные окружения, что важно для предотвращения конфликтов между зависимостями разных проектов.

2. Воспроизводимость: Контейнеры позволяют запускать одно и то же приложение в разных средах с одинаковыми результатами.

3. Упрощение развертывания: Docker облегчает развертывание приложения на разных платформах, таких как серверы, облака или даже локальные машины, без необходимости настройки окружения.

4. Интеграция с CI/CD: Docker идеально подходит для использования в пайплайнах CI/CD, автоматизируя развертывание и тестирование.

Создание Docker-контейнера для Python-приложения позволяет значительно упростить процесс развертывания и гарантирует, что приложение будет работать одинаково на любых платформах. Это особенно полезно в командах разработчиков, где важна консистентность среды, а также при работе с CI/CD системами для автоматизации тестирования и развертывания.

7.3 Автоматизация развертывания с помощью Ansible

Ansible – это инструмент автоматизации, который позволяет управлять конфигурацией серверов, развертыванием приложений и выполнением различных операций на удалённых машинах. Он предоставляет простой способ для автоматизации задач, таких как настройка серверов, установка программного обеспечения, управление инфраструктурой и развертывание приложений.

Ansible использует декларативный подход для описания конфигурации. В отличие от традиционных скриптов, где вам нужно явно прописывать последовательность команд, в Ansible достаточно указать, что вы хотите получить в результате, а инструмент позаботится о том, как это будет достигнуто.

Основные концепции Ansible

1. Инвентаризация: Ansible использует файл инвентаря для указания серверов, на которых будут выполняться задачи. Этот файл может содержать как статические, так и динамические данные о хостах.

2. Плейбуки (Playbooks): Плейбуки – это файлы YAML, в которых описаны задачи для выполнения. В плейбуках можно указать, какие действия должны быть выполнены на каких серверах, какие модули использовать, а также какие параметры передавать этим модулям.

3. Модули: Модули Ansible – это маленькие программы, которые выполняют конкретные задачи, например, установку пакетов, копирование файлов, настройку сервисов и т.д. Они выполняются на удалённых хостах.

4. Роли: Роли в Ansible позволяют группировать задачи и файлы, создавая удобную структуру для повторного использования. Роли могут включать задачи, переменные, шаблоны, файлы и другие элементы.

5. Шаблоны: Ansible использует язык шаблонов Jinja2 для динамической генерации конфигурационных файлов на основе переменных и условий.

Установка Ansible

Для начала работы с Ansible необходимо установить его на локальной машине. Это можно сделать с помощью pip:

```bash
pip install ansible
```

После установки можно проверить работу Ansible, выполнив команду:

```bash
ansible –version
```

Пример создания автоматизированного развертывания с помощью Ansible

1. Шаг 1: Создание файла инвентаря

Инвентарный файл Ansible описывает все хосты, на которых будут выполняться задачи. Он может быть в формате `.ini`, `.yaml` или `.json`. Рассмотрим простой пример инвентаря в формате `.ini`:

```ini
[webservers]
server1.example.com
server2.example.com
```

[dbservers]
db1.example.com
db2.example.com
```

Здесь мы определяем две группы серверов: `webservers` и `dbservers`, каждая из которых содержит хосты для веб-серверов и баз данных соответственно.

2. Шаг 2: Создание плейбука

Теперь создадим плейбук, который будет устанавливать и настраивать веб-сервер Nginx на наших серверах. Плейбук описывается в YAML-файле.

```yaml
—
– name: Установка и настройка веб-сервера Nginx
hosts: webservers
become: yes
tasks:
– name: Установка пакета Nginx
apt:
name: nginx
state: present
update_cache: yes
– name: Запуск службы Nginx
service:
name: nginx
state: started
enabled: yes
– name: Копирование конфигурационного файла Nginx
copy:
src: ./nginx.conf
dest: /etc/nginx/nginx.conf
owner: root
group: root
mode: '0644'
– name: Перезапуск Nginx для применения конфигурации
service:
name: nginx
```
```

state: restarted
```

Этот плейбук выполняет следующие действия:

– Устанавливает пакет `nginx` на серверах группы `webservers` с помощью модуля `apt`.

– Запускает и включает службу Nginx на этих серверах.

– Копирует конфигурационный файл Nginx на серверы.

– Перезапускает Nginx для применения новой конфигурации.

3. Шаг 3: Запуск плейбука

Теперь, когда у нас есть инвентарный файл и плейбук, можно запустить плейбук на целевых серверах. Для этого используем команду:

```bash
ansible-playbook -i inventory.ini deploy_nginx.yml
```

Здесь:

– `ansible-playbook` – это команда для выполнения плейбуков.

– `-i inventory.ini` – указывает путь к инвентарному файлу, который содержит информацию о хостах.

– `deploy_nginx.yml` – это файл плейбука, который нужно выполнить.

Ansible подключится к серверам, перечисленным в инвентаре, выполнит все задачи, описанные в плейбуке, и выведет результат выполнения для каждой задачи. Если всё пройдет успешно, на ваших веб-серверах будет установлен и настроен Nginx.

4. Шаг 4: Использование шаблонов

Для того чтобы динамически настраивать конфигурационные файлы, можно использовать шаблоны. Например, мы можем создать шаблон конфигурации Nginx с использованием Jinja2:

```nginx
# nginx.conf.j2
server {
listen 80;
server_name {{ server_name }};
location / {
root {{ web_root }};
index index.html;
```
```

}
}
```

Здесь переменные `server_name` и `web_root` будут заменены на реальные значения во время выполнения плейбука.

Теперь обновим наш плейбук, чтобы использовать этот шаблон:

```yaml
– name: Установка и настройка веб-сервера Nginx с шаблоном
hosts: webservers
become: yes
vars:
server_name: example.com
web_root: /var/www/html
tasks:
– name: Установка пакета Nginx
apt:
name: nginx
state: present
update_cache: yes
– name: Запуск службы Nginx
service:
name: nginx
state: started
enabled: yes
– name: Копирование конфигурационного файла Nginx из шаблона
template:
src: ./nginx.conf.j2
dest: /etc/nginx/nginx.conf
– name: Перезапуск Nginx для применения конфигурации
service:
name: nginx
state: restarted
```

Здесь используется модуль `template`, который заменяет переменные в шаблоне на значения, указанные в разделе `vars`. Это позволяет более гибко настраивать серверы и адаптировать их под разные окружения.
```

5. Шаг 5: Использование ролей

Если приложение состоит из нескольких компонентов, можно структурировать плейбуки и задачи с помощью ролей. Например, для установки и настройки Nginx можно создать роль, которая будет содержать все необходимые файлы и задачи. Создание роли выглядит следующим образом:

```bash
ansible-galaxy init nginx
```

Эта команда создаст структуру папок для роли Nginx, в которой будут храниться задачи, шаблоны, файлы и переменные. После этого можно использовать роль в плейбуке:

```yaml
– name: Установка и настройка Nginx
hosts: webservers
become: yes
roles:
– nginx
```

Роли делают ваш код более модульным и позволяют легко повторно использовать задачи и конфигурации в других проектах.

Преимущества использования Ansible

– Простота и гибкость: Ansible использует простой синтаксис на основе YAML, что облегчает написание и понимание плейбуков.

– Безагентный режим: Ansible не требует установки дополнительных агентов на управляемые серверы. Он использует стандартные протоколы, такие как SSH.

– Масштабируемость: Ansible позволяет управлять большими инфраструктурами, эффективно развертывая приложения на множестве серверов одновременно.

– Множество модулей и интеграций: Ansible поддерживает тысячи встроенных модулей для управления различными сервисами и приложениями, такими как базы данных, веб-серверы, облачные провайдеры и другие.

Ansible – это мощный инструмент для автоматизации развертывания приложений, настройки серверов и управления инфраструктурой. С

помощью Ansible можно легко создавать и поддерживать масштабируемые и воспроизводимые окружения, минимизируя количество ошибок, связанных с ручной настройкой. Использование плейбуков, шаблонов и ролей позволяет упростить и автоматизировать задачи, связанные с развертыванием и конфигурацией серверов, обеспечивая высокую консистентность и надёжность в процессе разработки и эксплуатации приложений.

Глава 8. Будущее Python и советы разработчикам

8.1 Перспективы Python: тенденции и новые библиотеки

Python продолжает оставаться одним из самых популярных языков программирования благодаря своей простоте, гибкости и широкому спектру применения. Однако, как и любой другой язык, Python не стоит на месте и постоянно развивается. В последние годы наблюдаются несколько ключевых тенденций, которые определяют будущее этого языка. Одной из них является активное расширение возможностей Python в таких областях, как искусственный интеллект (ИИ), машинное обучение (МО), а также в обработке больших данных и научных вычислениях.

Одним из заметных направлений является интеграция Python с технологиями глубокого обучения и нейросетей. Библиотеки, такие как TensorFlow, PyTorch и Keras, стали основными инструментами для создания и обучения моделей ИИ, а Python, в свою очередь, обеспечивает удобный интерфейс для работы с этими мощными инструментами. С каждым годом появляются новые библиотеки, которые делают работу с данными и алгоритмами еще более доступной и эффективной. Например, такие библиотеки, как Hugging Face для работы с трансформерами, позволяют разрабатывать и тренировать модели обработки естественного языка, что открывает новые возможности в области обработки текста и речи.

Еще одной важной тенденцией является развитие Python в области веб-разработки и построения микросервисных архитектур. Современные фреймворки, такие как Django и Flask, продолжают развиваться, добавляя новые функциональности, улучшая производительность и упрощая развертывание приложений. В последнее время также наблюдается рост интереса к серверлесс-технологиям и Python-функциям, которые позволяют разработчикам

создавать приложения, масштабируемые без необходимости управлять инфраструктурой.

Не менее важным направлением является развитие Python в области анализа данных и визуализации. Новые библиотеки, такие как pandas, NumPy и Matplotlib, постоянно обновляются, предлагая более удобные и быстрые способы обработки и представления данных. Библиотеки, ориентированные на работу с большими данными, например Dask, становятся всё более востребованными для аналитиков и исследователей.

С другой стороны, появляются новые библиотеки и инструменты, направленные на улучшение производительности Python. В последние годы было разработано несколько проектов, таких как PyPy (альтернатива стандартному интерпретатору CPython), которые значительно ускоряют выполнение Python-кода. Также активно разрабатываются библиотеки для параллельных вычислений и многозадачности, что позволяет Python оставаться конкурентоспособным в таких областях, как обработка больших данных и реальное время.

В будущем Python продолжит развиваться как язык, который сочетает в себе простоту использования и мощные возможности для решения сложных вычислительных задач. Его экосистема будет лишь расширяться, предоставляя разработчикам новые инструменты для работы с ИИ, большими данными, веб-разработкой и многими другими областями.

8.2 Как выбрать правильную библиотеку для задачи

Одним из самых мощных аспектов Python является его обширная экосистема библиотек, которые предоставляют готовые решения для различных типов задач – от анализа данных до разработки веб-приложений и искусственного интеллекта. Однако при таком разнообразии библиотек важно уметь выбрать ту, которая наилучшим образом соответствует конкретным требованиям проекта. Выбор правильной библиотеки – это не просто вопрос удобства, но и вопрос производительности, масштабируемости, поддержки сообщества и

совместимости с другими инструментами. Рассмотрим несколько аспектов, которые стоит учитывать при выборе библиотеки.

1. Определение требований и целей проекта

Прежде всего, необходимо четко понять, какую задачу нужно решить. Библиотеки часто специализируются на конкретных типах задач. Например, для работы с машинным обучением популярными являются библиотеки TensorFlow, PyTorch и scikit-learn, в то время как для обработки и анализа данных преимущественно используется pandas, NumPy и Dask. Если цель проекта заключается в построении моделей глубокого обучения, вам стоит рассматривать TensorFlow или PyTorch. Важно понимать, что в некоторых случаях для специфических задач может существовать несколько подходящих решений, и тогда выбор будет зависеть от ваших предпочтений, опыта и специфики проекта.

2. Документация и обучающие материалы

Одним из ключевых факторов выбора библиотеки является наличие качественной документации и обучающих материалов. Хорошо документированная библиотека облегчит процесс обучения и разработки, позволит быстрее разобраться в ее функционале. Важно обратить внимание на следующие аспекты документации:

– Примеры кода, которые демонстрируют, как использовать библиотеку для решения типичных задач.

– Подробные объяснения функциональности и параметров.

– Форумы и сообщества, где можно получить помощь, если возникнут вопросы.

В случаях, когда вы только начинаете работать с какой-то технологией или библиотекой, наличие обучающих материалов и активного сообщества может сыграть решающую роль.

3. Производительность

Производительность библиотеки – это еще один важный аспект выбора, особенно если речь идет о решении сложных вычислительных задач или работе с большими объемами данных. Например, для работы с массивами данных оптимальным выбором будет библиотека NumPy, которая предоставляет высокопроизводительные операции с массивами благодаря использованию низкоуровневых операций и работы с C-реализациями функций.

Если задачи включают обработку больших объемов данных, стоит обратить внимание на библиотеки, поддерживающие распределенные вычисления, такие как Dask или PySpark. Эти библиотеки позволяют эффективно работать с данными, которые не помещаются в память, используя распределенные вычисления.

Если же важна скорость выполнения операций с ограниченными объемами данных, вы можете рассмотреть альтернативные интерпретаторы Python, такие как PyPy, которые предлагают улучшенную производительность по сравнению с обычным CPython.

4. Поддержка и популярность

Популярность библиотеки и поддержка сообщества играют важную роль в принятии решения о её использовании. Чем более популярна библиотека, тем больше ресурсов (форумы, обучающие материалы, примеры) будет доступно для изучения. Также более популярные библиотеки, как правило, имеют регулярные обновления и исправления ошибок. Однако не всегда популярность является единственным фактором. В некоторых случаях вам может подойти менее известная библиотека, которая лучше решает конкретную задачу.

Примером такого случая является использование специализированных библиотек, например Scrapy для веб-скрейпинга, которая значительно эффективнее стандартных решений для таких задач. Важно также учитывать активность разработчиков библиотеки: частота обновлений и активность на GitHub или других платформах могут служить хорошим индикатором того, что библиотека активно поддерживается.

5. Совместимость с другими библиотеками

Очень часто проекты включают в себя несколько библиотек, которые должны работать вместе. Поэтому важно проверить, насколько хорошо выбранная библиотека интегрируется с другими инструментами, которые уже используются в проекте. Например, в контексте машинного обучения стоит проверить, совместима ли выбранная библиотека с другими инструментами для анализа данных или визуализации, такими как pandas, Matplotlib или Seaborn.

Если вы разрабатываете веб-приложение, важно, чтобы выбранная библиотека работала с фреймворками, такими как Flask или Django. В случае, если библиотека требует специфической версии Python или

другой среды, вам нужно убедиться, что она подходит для вашего стека технологий.

6. Лицензирование

Не менее важным фактором при выборе библиотеки является её лицензия. Некоторые библиотеки могут иметь ограничения на использование в коммерческих приложениях, или на модификацию и распространение исходного кода. Для коммерческих проектов стоит выбирать библиотеки с лицензиями, которые позволяют свободно использовать, изменять и распространять программное обеспечение (например, MIT или Apache 2.0). Важно внимательно изучить условия лицензии и убедиться, что библиотека подходит для вашего типа проекта.

7. Масштабируемость

Если ваш проект должен работать в больших масштабах, вам стоит обратить внимание на библиотеки, которые поддерживают горизонтальное масштабирование и могут эффективно работать в распределённых системах. Например, для обработки больших данных и параллельных вычислений можно использовать такие инструменты, как Dask или PySpark, которые могут масштабироваться на несколько серверов и обрабатывать гигабайты и терабайты данных, что делает их отличным выбором для крупных проектов.

Для веб-разработки, если предполагается высокая нагрузка, библиотеки, такие как FastAPI, предлагают высокую производительность благодаря асинхронному программированию и поддержке множества запросов.

8. Безопасность

Безопасность библиотеки – еще один важный аспект, который не следует игнорировать. Уязвимости в сторонних библиотеках могут привести к серьезным проблемам, таким как утечка данных или нарушения в работе приложения. Поэтому стоит использовать проверенные и регулярно обновляемые библиотеки, а также следить за уведомлениями о безопасности. Для этого можно использовать инструменты, такие как Safety или Bandit, которые анализируют код на наличие уязвимостей.

Выбор правильной библиотеки зависит от множества факторов, включая тип задачи, требования к производительности, совместимость с другими инструментами, а также популярность и поддержка

сообщества. Важно учитывать все эти аспекты, чтобы выбрать наиболее подходящее решение для вашего проекта. Использование правильно подобранных библиотек может существенно ускорить разработку, повысить производительность приложения и обеспечить его стабильность на долгосрочной основе.

8.3 Лучшие практики в работе с Python-проектами

Работа с Python-проектами требует внимания не только к выбору инструментов и библиотек, но и к соблюдению ряда принципов и практик, которые способствуют эффективной и стабильной разработке, повышают читаемость кода, упрощают его поддержку и позволяют команде разработчиков работать более продуктивно. В этой главе рассмотрены лучшие практики, которые следует учитывать при разработке и поддержке Python-проектов.

1. Структура проекта

Организация структуры проекта играет ключевую роль в его дальнейшей поддержке и масштабировании. Хорошо продуманная структура позволяет легко находить нужные файлы, управлять зависимостями и тестировать код. Рекомендуется придерживаться стандартной структуры для Python-проектов, которая включает следующие директории и файлы:

```
my_project/
│
├── my_project/ # Основной пакет проекта
│   ├── __init__.py # Инициализационный файл пакета
│   ├── module1.py # Модуль 1
│   └── module2.py # Модуль 2
│
├── tests/ # Каталог для тестов
│   ├── __init__.py
│   └── test_module1.py # Тесты для модуля 1
│
├── requirements.txt # Зависимости проекта
```

```
├── setup.py # Скрипт установки проекта
└── README.md # Описание проекта
```

Такой подход помогает поддерживать проект в организованном виде, облегчает его расширение и делает код более удобным для восприятия другими разработчиками.

2. Использование виртуальных окружений

Для каждого проекта рекомендуется создавать отдельное виртуальное окружение. Это позволяет изолировать зависимости проекта от глобальных библиотек и избежать конфликтов между различными проектами. Для создания виртуального окружения можно использовать `venv` или `virtualenv`:

```bash
python3 -m venv venv
```

После этого нужно активировать окружение:

– На Windows:

```bash
venv\Scripts\activate
```

– На Linux/macOS:

```bash
source venv/bin/activate
```

Все зависимости проекта нужно фиксировать в файле `requirements.txt`:

```bash
pip freeze > requirements.txt
```

Этот файл позволяет легко установить все зависимости проекта на другом компьютере с помощью команды:

```bash
pip install -r requirements.txt
```

3. Следование стандартам кодирования

Python придерживается набора стандартов кодирования, которые описаны в [PEP 8](https://pep8.org/). Этот документ содержит

рекомендации по стилистике кода, включая правила отступов, именования переменных, длины строк и других аспектов. Соблюдение PEP 8 помогает улучшить читаемость и поддерживаемость кода, особенно когда над проектом работает несколько разработчиков.

Некоторые ключевые моменты:

– Использование 4 пробелов для отступов.

– Именование переменных в стиле `snake_case` для переменных и функций, `CamelCase` для классов.

– Ограничение длины строки до 79 символов.

Для автоматической проверки соответствия кода стандартам можно использовать инструменты, такие как `flake8` или `pylint`, которые помогут выявить ошибки в стиле и предложат исправления.

4. Документирование кода

Документация играет важную роль в понимании кода, особенно когда проект становится более сложным. Для документирования кода следует использовать строки документации (docstrings), которые описывают назначения функций, классов и методов. Хорошо написанная документация помогает как другим разработчикам, так и себе при возврате к коду спустя некоторое время.

Пример использования docstring для функции:

```python
def add(a, b):
"""
Функция для сложения двух чисел.
Parameters:
a (int, float): Первое число
b (int, float): Второе число
Returns:
int, float: Сумма a и b
"""
return a + b
```

Существует несколько инструментов, таких как `Sphinx`, которые могут автоматически генерировать документацию на основе docstrings.

5. Тестирование

Тестирование – неотъемлемая часть разработки программного обеспечения. Оно позволяет удостовериться, что ваш код работает так,

как ожидается, и предотвращает появление ошибок при внесении изменений. В Python для тестирования часто используется библиотека pytest, которая предоставляет простую и мощную среду для написания и выполнения тестов.

Лучше всего использовать тестирование на уровне функций и модулей, а также интеграционные тесты, чтобы убедиться, что все компоненты системы работают корректно в связке.

Пример теста с использованием `pytest`:

```python
def test_add():
assert add(2, 3) == 5
assert add(-1, 1) == 0
```

Тестирование также включает в себя создание тестов для обработки исключений, тесты на производительность и проверку безопасности.

6. Управление зависимостями

Хорошая практика заключается в том, чтобы четко управлять зависимостями проекта и их версиями. Для этого используются файлы `requirements.txt` или `Pipfile`. Важно избегать установки ненужных или устаревших зависимостей, чтобы не перегружать проект лишним функционалом.

Если проект имеет несколько окружений (например, для разработки, тестирования и продакшн), следует использовать такие инструменты, как pipenv или Poetry, которые помогают управлять зависимостями и виртуальными окружениями.

7. Обработка исключений

Обработка исключений в Python важна для обеспечения надежности программы. Хорошо спроектированная система обработки ошибок позволяет программе не завершаться с ошибками, а gracefully обрабатывать непредвиденные ситуации. Используйте конструкции `try`, `except`, `finally` для обработки исключений и предоставления понятных сообщений пользователю или логирования ошибок.

Пример:

```python
try:
result = divide(a, b)
except ZeroDivisionError:
```

```
print("Ошибка: деление на ноль")
except TypeError:
print("Ошибка: неправильный тип данных")
```

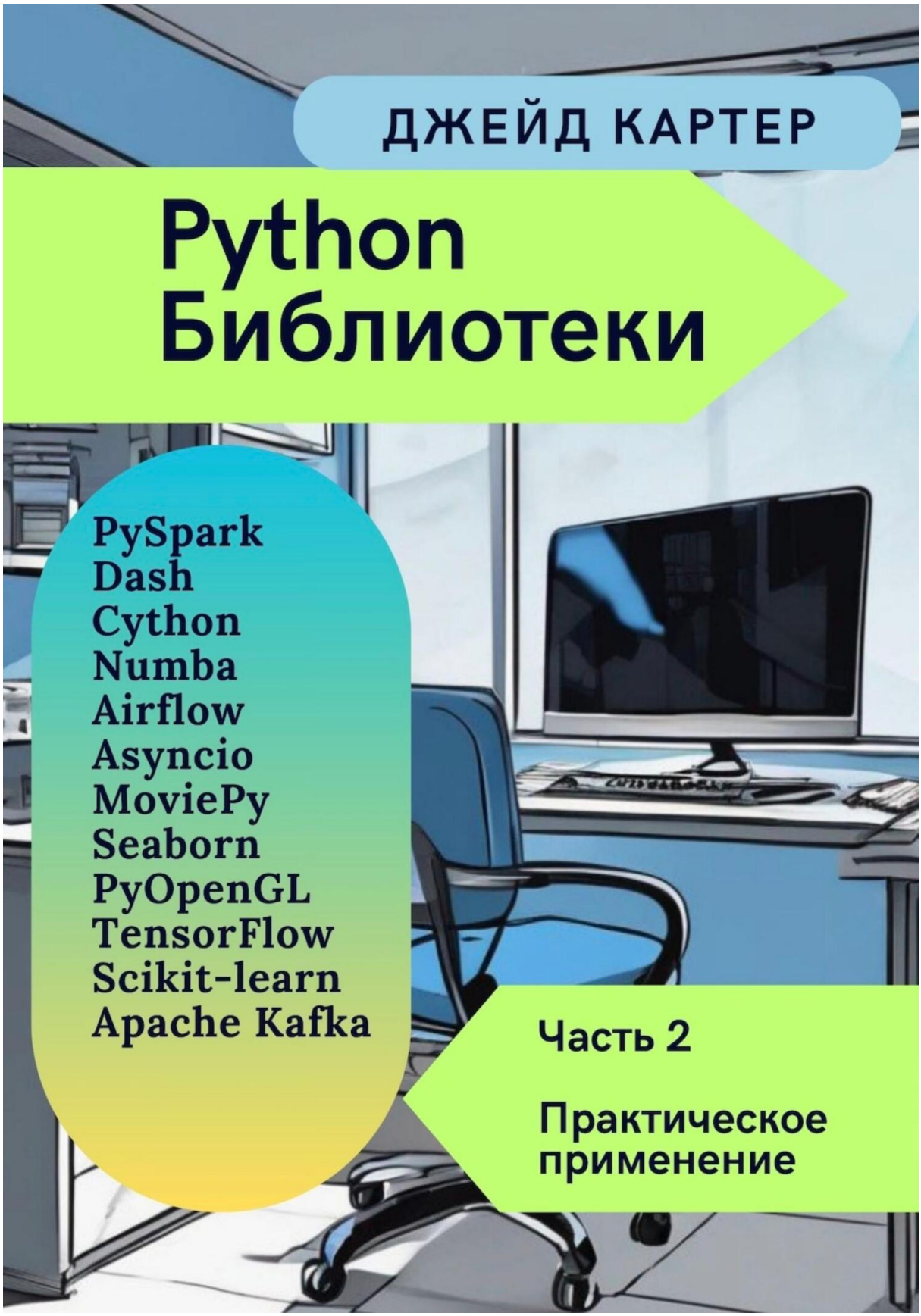
В случае работы с большими приложениями рекомендуется использовать библиотеки для логирования, такие как `logging`, чтобы записывать ошибки в файл для последующего анализа.

8. Использование контроля версий

Использование системы контроля версий, такой как Git, является обязательным для любого серьезного проекта. Это позволяет отслеживать изменения в коде, работать в команде, легко откатывать изменения и синхронизировать рабочие версии кода. Рекомендуется использовать платформы для хостинга репозиториев, такие как GitHub, GitLab или Bitbucket, которые облегчают совместную работу и автоматизацию процессов.

Каждый коммит должен быть хорошо описан, чтобы другие разработчики могли легко понять, какие изменения были внесены. Хорошая практика – создание веток для каждой новой фичи или исправления ошибки, чтобы избежать конфликтов в коде.

Придерживаясь лучших практик в разработке Python-проектов, вы обеспечите их масштабируемость, поддержку и стабильность. Использование правильных инструментов, структура проекта, качественная документация, тестирование и управление зависимостями – все эти аспекты помогают создать качественное, эффективное и надежное приложение. Следуя этим рекомендациям, вы сможете избежать множества распространенных проблем и обеспечить успешную разработку Python-проектов.


ДЖЕЙД КАРТЕР
Python Библиотеки
PySpark
Dash
Cython
Numba
Airflow
Asyncio
MoviePy
Seaborn
PyOpenGL
TensorFlow
Scikit-learn
Apache Kafka
Часть 2
Практическое применение