Бакалов Д.А.

Программируем финансы на Python

криптовалюта, биржа, торговые и Телеграм боты

- Основы Python и библиотек
- Реализация Биткойн и Альткойн на Python
- Создание Биржи бинарных опционов
- Бэктрейдинг и автоторговля, Пенсионный Фонд своими руками

НиТ
издательство

Бакалов Д. А.

Программируем финансы на Python

Криптовалюта, биржа, торговые и Телеграм боты

"Издательство Наука и Техника"

Санкт-Петербург

УДК 004.42
ББК 32.973

Бакалов Д. А.

Программируем финансы на Python: криптовалюта, биржа, торговые и Телеграм боты — СПб.: Издательство Наука и Техника, 2025. — 320 с., ил.

ISBN 978-5-907592-79-7

Эта книга поможет Вам стать реально богаче используя автоматизацию финансов с помощью Python: Вы не только создадите прототип биржи и собственные скринеры в Телеграм, но и научитесь грабить популярные биржи с помощью ими же предоставленных средств.

Вам НЕ потребуются знания в области программирования, т.к. всё необходимое для базового навыка программиста Вы получите в первых двух главах, в которых мы разработаем биржу бинарных опционов, реализуем Биткойн и Альткойн, а также освоим объектно-ориентированное программирование, без которого невозможно эффективно пользоваться библиотеками для бэкстестинга и торговли.

Далее мы разработаем вспомогательные инструменты для работы трейдера (к примеру, перенесём биржу опционов из первой главы в Телеграм), научим Вас создавать одновременно простых и полезных чат-ассистентов, которые будут мониторить рынки и сигнализировать о его состоянии. Отдельное внимание уделим тестированию торговых стратегий и, получив в первых главах навыки программирования, разработаем автоматизированную торговую систему со средним потенциалом возврата инвестиций до 70% годовых на падающем рынке.

Автор приложил много усилий для простоты и чистоты кода, чтобы код примеров было легко читать даже новичкам (например, Телеграм скринеры удалось реализовать без использования специализированных библиотек, используя лишь простые HTTP-запросы и JSON-файлы), ну а опытные разработчики смогут оценить, насколько просто такой код поддерживать и получат рабочие прототипы за считанные часы (а не недели (!)).

Приведённые пошаговые примеры имеют прикладной характер и используются в ботах, которые работают на автора ежедневно, при этом примеры могут быть легко адаптированы под Ваши собственные цели и стратегии.

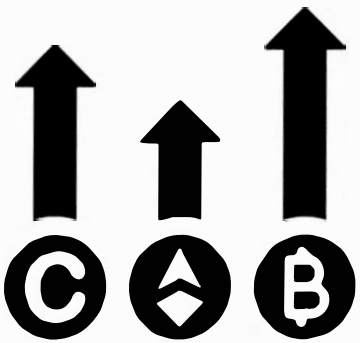
Книга рассчитана на всех, кто хочет создать дополнительный источник дохода используя автоматизацию финансов с помощью Python.

Дерзайте и берегите своё здоровье! (и депозит ☺).

ISBN 978-5-907592-79-7

Контактные телефоны издательства:
(812) 412 70 26
Официальный сайт: www.nit.com.ru

Содержание


C
B

Об авторе

Приветствую, дорогой читатель!

Меня зовут Даниил Бакалов. Я предприниматель и DJ.

Немного нескромной информации о себе:

- в 2008 году первым в СНГ основал гибридный Techno/D&B лейбл «Future13 Recordings».
- в 2018 году основал школу Технических Искусств WWW.ASTARTA-EDU.COM.
- в 2022 году создал бренд уличной одежды WWW.XOZAAA.COM, а в 2025 году маркетплейс WWW.BARMALDA.RU.

Также принимал участие в различных проектах банковского сектора, в том числе в компании VISA. Повысил техническую квалификацию десятков сотрудников: СБЕР, МТС, Центр Финансовых Технологий и правительства города Москвы.

Автоматизация финансов моя вторая страсть после музыки. Впервые увидев системный блок компьютера, я подумал, что из разъёма для флоппи-диска должна печататься наличность! В целом, большей пользы от владения персональным компьютером лично мне так и НЕ удалось обнаружить. Музыку ведь можно писать и на других устройствах…

Книга, которую Вы держите в руках, является концентрацией моего 15-летнего практического и научного изыскания сразу в двух областях:

1. Эффективное обучение (переобучение) базовому программированию.
2. Поведение манипуляторов и толпы на финансовых рынках.

С помощью языка программирования Python вы научитесь создавать вспомогательные инструменты для торговли на финасовых рынках, создадите прототип собственной биржи в Телеграм и научитесь действовать аналогично институциональным трейдерам. Разберётесь в обратной стороне

финансовых учреждений, поймёте, как мыслят те, кто находится по ту сторону рекламных компаний, предлагающих Бонус за очередной депозит.

Чтобы это произошло, незамедлительно станьте владельцем этой книги, если вы ещё этого не сделали ☺! Будет открыто много секретов. Если Вы уже купили книгу, ни кому о ней НЕ рассказывайте, пусть она станет Вашей Тайной.

Если вы давно пытаетесь освоить базовый навык программиста, то описанный в первых двух главах подход, который мы назвали «Понимай, или Умри», точно Вам поможет! Метод оправдан результатами сотен моих студентов (совершенно разных возрастов), и при переобучении действующих инженеров.

Мои контакты:

Мой блог: www.XOZAAA.blogspot.com

Чат книги: www.t.me/AUTOMATION_FINANCE

Если Вам необходимо сопроводить сложный проект: founder@XOZAAA.com

Если Вы ищете наставника: support@ASTARTA-EDU.com

Чтобы арендовать торгового Робота: www.AUTO-MAFIA.tilda.ws

Заказать торгового или сигнального Робота: AUTO-MAFIA@inbox.ru

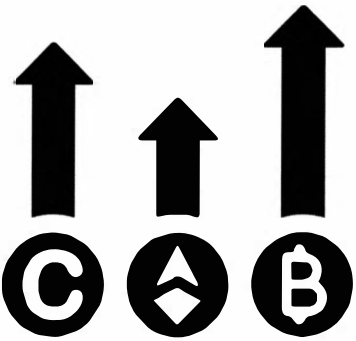
Удачи в обучении!

С Уважением, Bakalov D.

Книга посвящается Заслуженному Учителю Российской Федерации,

Бакаловой Лидии Викторовне

Классический Технический Анализ Не Работает На Большинстве Крипто Монет, Ибо Любой Школьник, Добравшийся До Кредитки Отца, Может Устроить Манипуляцию. (с)



Предисловие

Компьютерные системы для автоматизации финансов существуют со дня создания первых ЭВМ. Банковский сектор, покрывая технические потребности авангардными разработками, надолго стал пионером в отрасли автоматизации крупного капитала.

Первые квантовые ЭВМ тоже были использованы финансистами. Самая сильная разработка по обучению квантового псевдо-ИИ называется Aladdin. Она была создана кампанией BlackRock Inc. в 1980 году. Машина 10 лет анализировала данные обо всех трейдах, ордерах и сделках на всех доступных на тот момент рынках и теперь даёт инвестиционные советы на триллионы долларов десяткам тысяч предприятий.

В огромных компаниях до сих пор работают программы, написанные 40 лет назад на языках программирования COBOL, ADA и FORTRAN. Эти программы генерируют сотни миллионов долларов прибыли каждый год.

Существуют боты, в том числе и браузерные, которые берут на себя автоматизацию повседневных рутинных действий. Например, размещают ставки у букмекеров, делают бесплатные рассылки в Ватсап, анализируют новости в Телеграм-каналах.

Большая часть подобного рода ботов находится в приватном пространстве и недоступна широкой публике. Мне посчастливилось разработать НЕ один десяток ботов, которые прямо или косвенно приносят деньги в период своего жизненного цикла. Многие боты работают годами и до сих пор обслуживают интересы бизнеса.

По большей части в публичном пространстве мы привыкли пользоваться ботами, которые создаются для продвижения услуг. Такие боты прямой

пользы не приносят, лишь помогают бизнесам удерживать клиента или развлекать пользователя с целью фиксации внимания для доставки рекламы.

Вам посчастливится применить Пайтон в самой приятной области (на мой субъективный взгляд) — автоматизации финансов. При этом вы не утонете в бездне сухой технической терминологии. Я сделаю всё возможное, чтобы путешествие прошло весело!

Благодаря этому руководству вы не только создадите прототип биржи и собственные скринеры в Телеграм, но и научитесь грабить популярные биржи с помощью ими же предоставленных средств. Биржи страшно боятся ботоводов (!). К счастью, рынок диктует конкурентные правила и выбора, как предоставлять открытый API-доступ к торговле, у них нет. Мы злостно этим воспользуемся!

Вам НЕ потребуются знания в области программирования. Всё необходимое для базового навыка программиста вы получите в первых двух главах. Тем не менее, желателен опыт торговли на финансовых рынках, в частности криптовалютных.

С первых строк книги для получения базового навыка программиста вы будете разрабатывать биржу бинарных опционов. Данный образовательный пример показал эффективность при обучении (и переобучении) десятков моих студентов и позволит вам уверенно овладеть функциональным программированием.

Во второй главе вы на достаточном уровне освоите объектно-ориентированное программирование, без которого невозможно эффективно пользоваться библиотеками для бэктестинга и торговли.

В третьей части книги вы разработаете вспомогательные инструменты для работы трейдера, научитесь создавать одновременно простые и безумно полезные чат-ассистенты, которые будут мониторить рынок и сигнализировать о его состоянии. А также попробуете перенести биржу опционов из первой главы в Телеграм.

В четвертой главе вы примените навык программиста для тестирования торговых стратегий и получите полный набор инструментов институционального трейдера.

В пятой главе вы выстроите автоматизированную торговую систему со средним потенциалом возврата инвестиций до 70% годовых на падающем рынке.

Благодаря навыку программирования и автоматизации торговых стратегий можно не только отставиться от большинства трейдеров, теряющих деньги, но и начать действовать как институционал.

Большая часть приведённых примеров имеет прикладной характер и используется в ботах, которые работают на меня ежедневно. Примеры могут быть легко адаптированы под ваши собственные цели и стратегии.

При этом НЕ забывайте, что алгоритмическая торговля тоже риск. Роботы не являются гарантией прибыли. Роботы — это НЕ панацея. Нет такого робота, который по ошибке НЕ сольёт депозит. Робот лишь снижает психическую нагрузку на человека и позволяет практически молниеносно реагировать на формацию в рынке.

В целом и общем, эта книга является скромной попыткой вернуться к истинным истокам технического прогресса. Ведь НЕ для того мы были созданы, чтобы обслуживать документы EXCEL...

Глава 1.

Функциональное программирование.

Создаём биржу бинарных опционов на Биткойн

Содержание главы:

1.1. Коротко о Пайтон

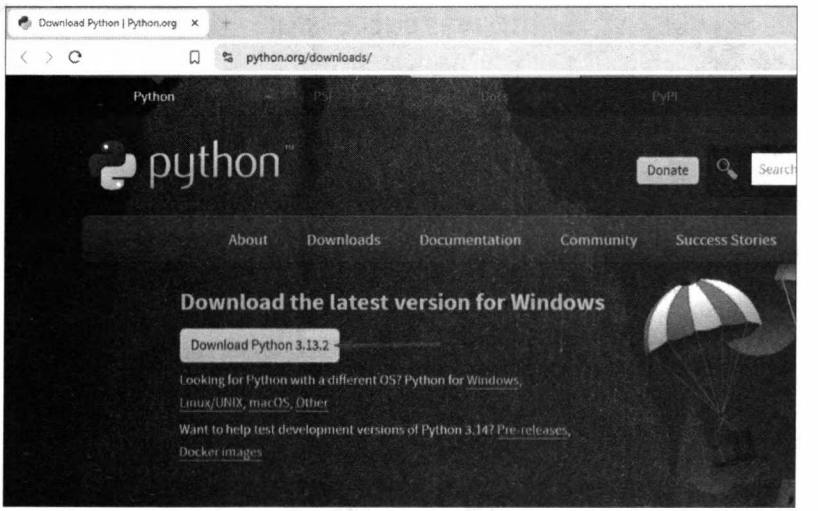
Пайтон был разработан Гвидо Ван Россумом в 1990 году и получил своё название благодаря комедийной группе 30-х годов Монти Пайтон. Наибольшую популярность Пайтон получил в разработке финансовых инструментов, анализе данных и ИИ.

Пайтон — интерпретируемый язык программирования, выполняющийся в режиме реального времени. Поддерживает объектно-ориентированное, функциональное и императивное программирование.

Пайтон идеально подходит в качестве первого языка программирования. Его недостатки — это медлительность (исправляется с помощью Cython), динамическая типизация и зависимость синтаксиса от вложенности конструкций.

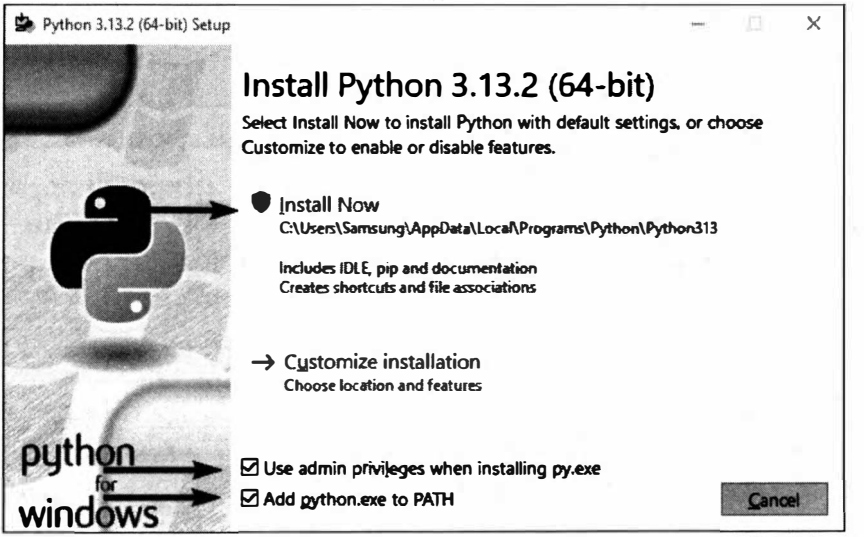
1.2. Установка Python

1. Перейдите по ссылке www.python.org/downloads и нажмите на **Download**.



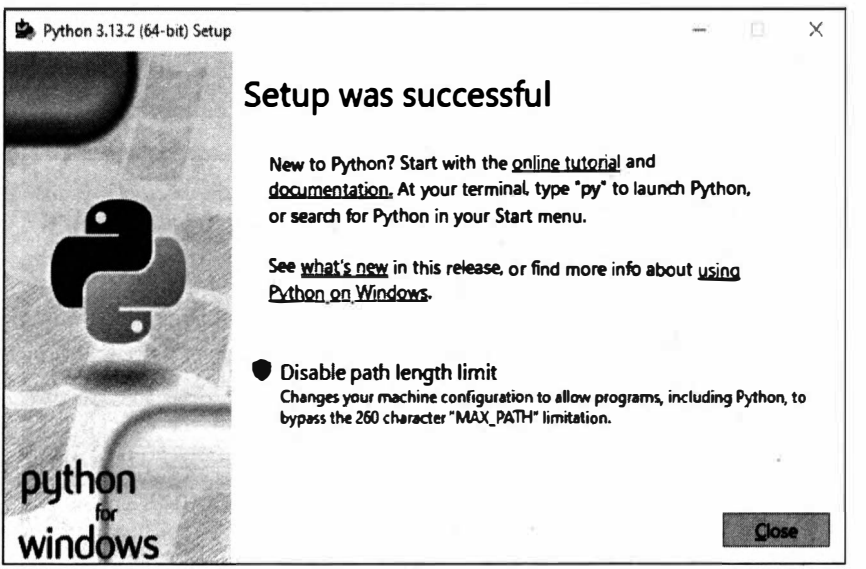
Изображение 1.1.

2. После скачивания инсталлятора запустите его, поставьте галочки напротив **Use admin privileges when installing py.exe** и **Add python.exe to PATH**, затем нажмите кнопку **Install Now** и дождитесь установки интерпретатора.



Изображение 1.2.

3. Если всё прошло успешно, вы увидите примерно следующее:



Изображение 1.3.

Для быстрого старта вы можете пропустить установку Пайтона и развертывание среды разработки на локальном компьютере. Вместо этого воспользуйтесь браузерным редактором кода Google Colab по адресу:

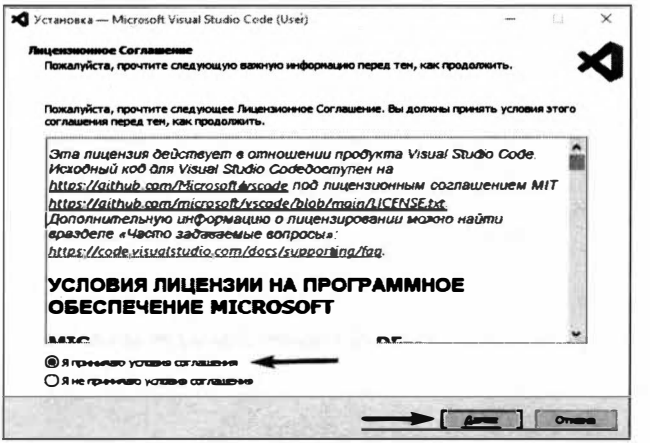
www.colab.research.google.com

1.3. Установка Visual Studio Code

1. Перейдите по ссылке www.code.visualstudio.com и нажмите на **Download**.

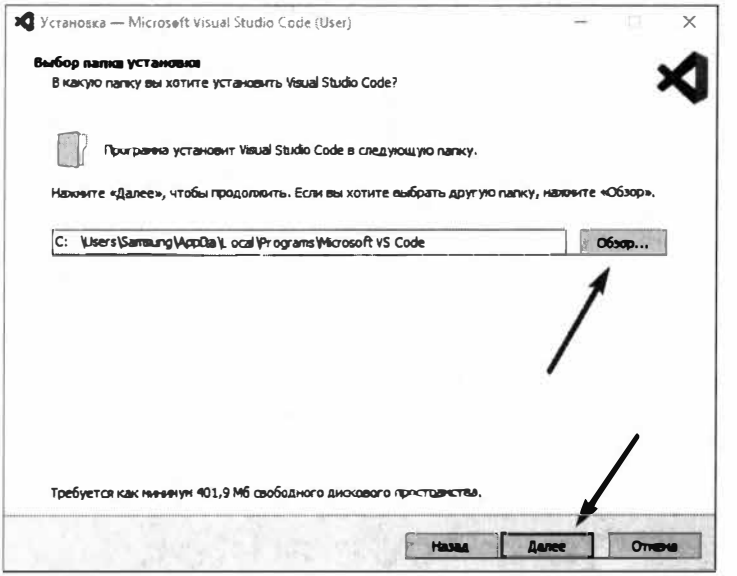
Изображение 1.4.

2. После скачивания инсталлятора запустите его, примите условия соглашения и нажмите кнопку **Далее**.



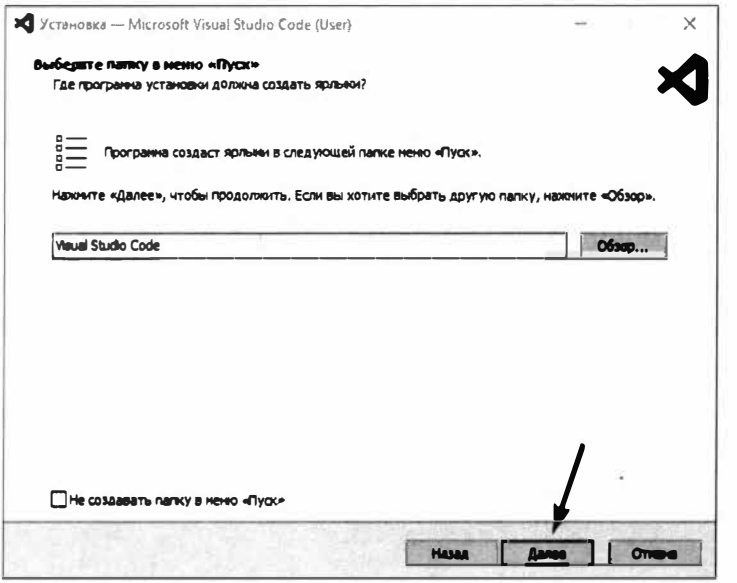
Изображение 1.5.

3. Выберите место для установки и снова нажмите кнопку **Далее**.



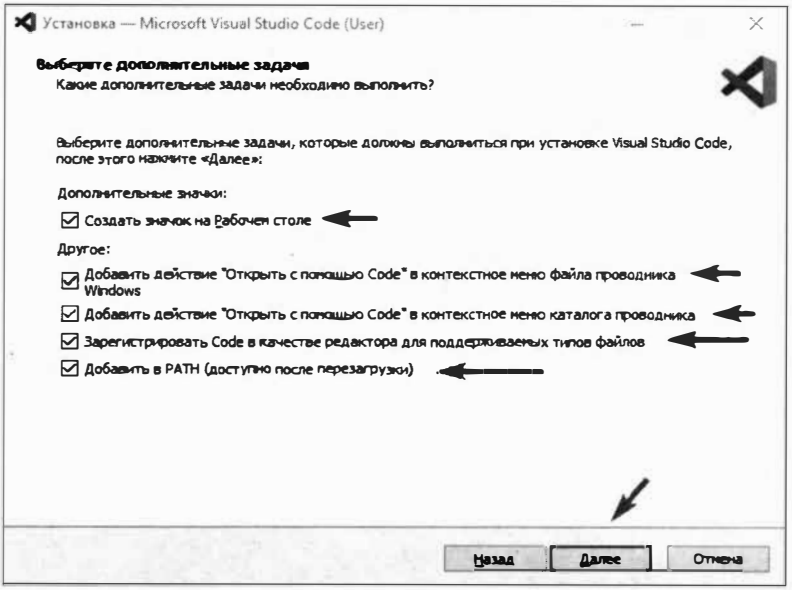
Изображение 1.6.

4. Шаг для выбора папки в меню **Пуск** оставьте как есть и нажмите **Далее**.



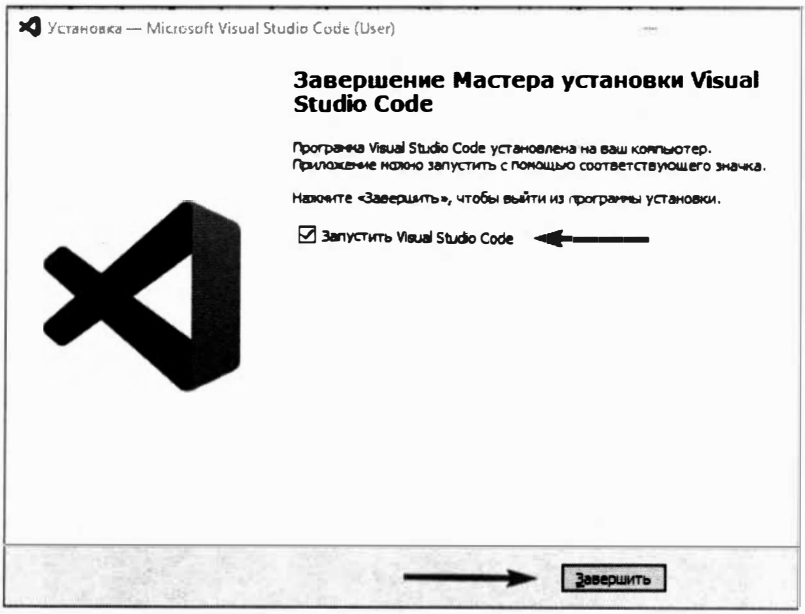
Изображение 1.7.

5. В дополнительных задачах поставьте галочки напротив всех пунктов, нажмите кнопку **Далее**, а затем кнопку **Установить**.



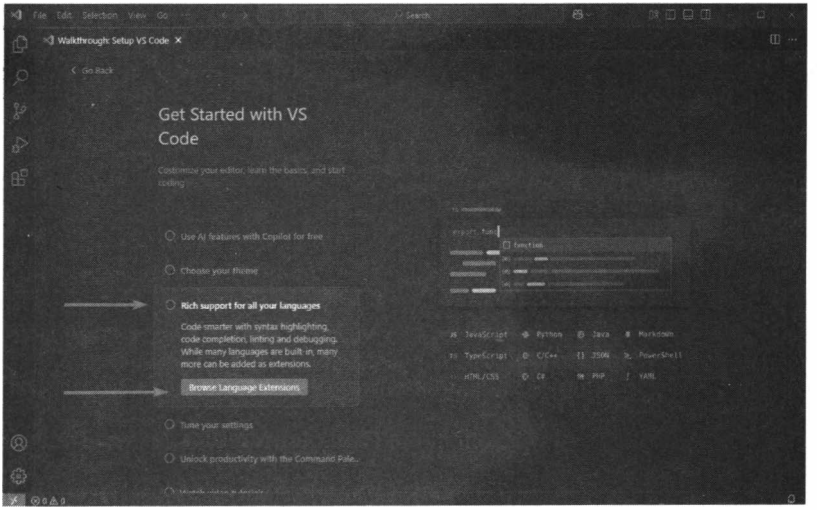
Изображение 1.8.

6. На финальном шаге нажмите кнопку **Завершить** и дождитесь запуска.



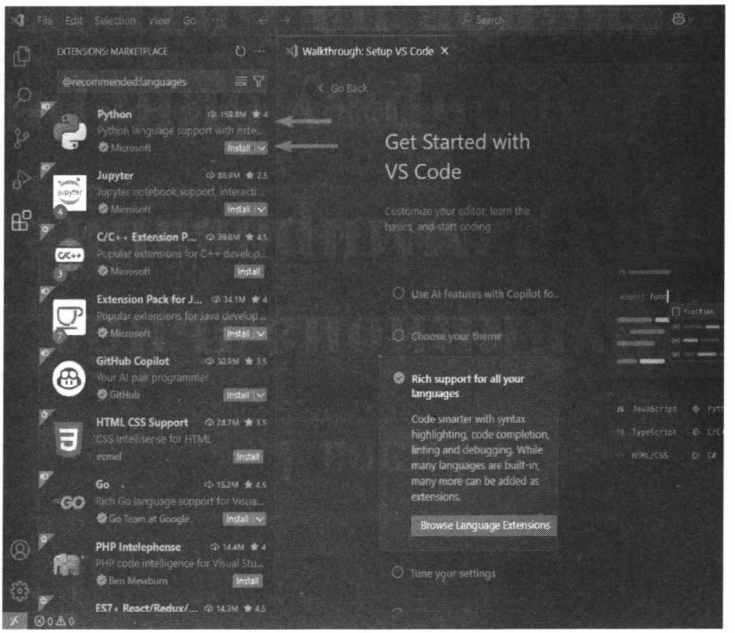
Изображение 1.9.

7. После запуска среды разработки выберите пункт **Rich support for all languages** и нажмите на кнопку **Browse Language Extensions**.



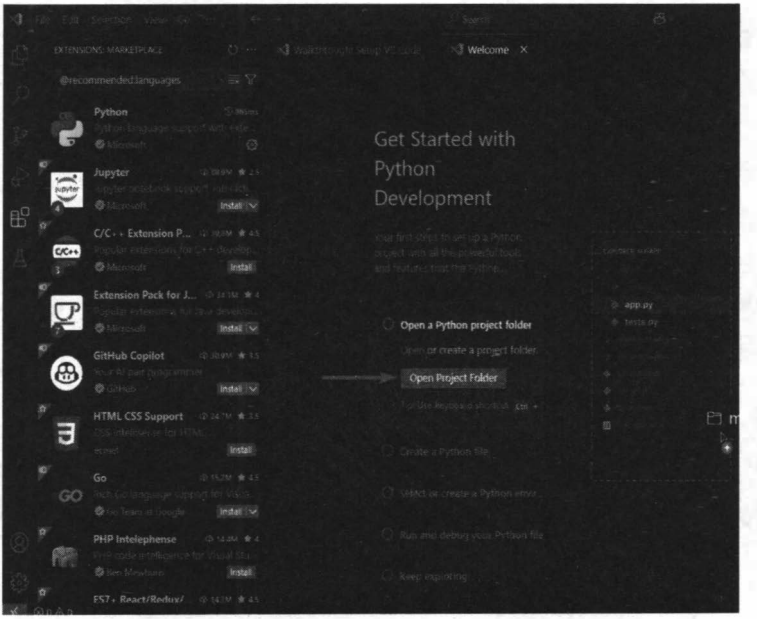
Изображение 1.10.

8. Дождитесь, когда в правой части редактора появится список из популярных языковых расширений, и установите расширение для Python. Установка займет около 1 минуты.



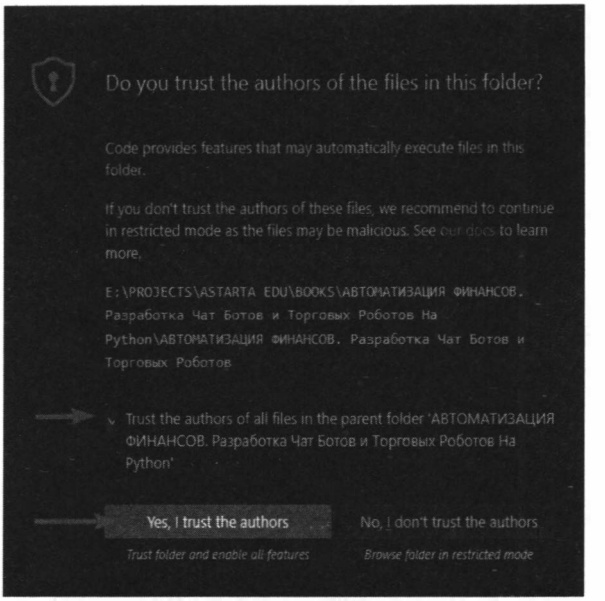
Изображение 1.11.

9. В любом удобном месте на вашем жёстком диске создайте папку **Автоматизация финансов**. Вернитесь в редактор, нажмите на кнопку **Open Project Folder** и выберите только что созданную папку.



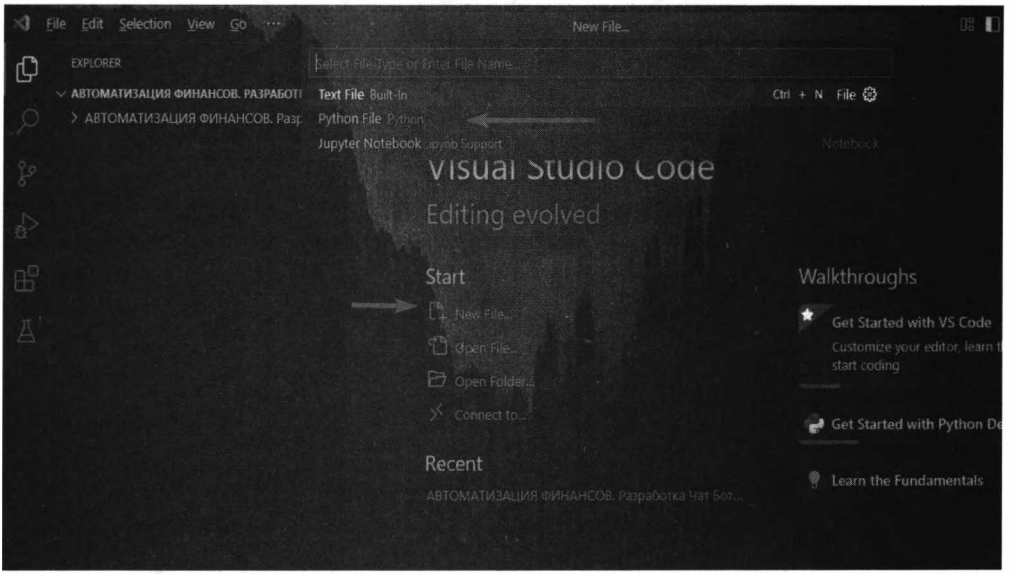
Изображение 1.12.

10. В появившемся окне установите галочку напротив выбранной папки, нажмите кнопку **Yes, I trust the authors** и на всякий случай перезапустите редактор.



Изображение 1.13.

11. После перезагрузки редактора в разделе **Welcome** нажмите на пункт **New File** и в выпадающем меню выберите пункт **Python File**.



Изображение 1.14.

12. После создания файла нажмите сочетание клавиш **CTRL + S**, присвойте файлу название **BinaryOptions** и нажмите кнопку **Сохранить**.

Изображение 1.15.

Поздравляю! Вы подготовили среду разработки и готовы написать первый код.

1.4. Техническое задание Биржи бинарных опционов

Бизнес бинарных опционов достаточно прост и основан на знании теории вероятности и анализе статистики. Технически, совершая сделку на бирже бинарных опционов, вы просто делаете ставку на рост или падение актива. Обещание выплаты в 80% — это ставка на коэффициент 1.80. Большая часть опционных бирж скрывают, что являются букмекерскими конторами.

В целом криптобиржа тоже мало чем отличается от букмекерской конторы, с одной лишь разницей — при работе с букмекерами вы заранее знаете сумму, которую проиграете или выиграете.

Никто не знает, куда пойдет цена и где она окажется через 30 секунд (конечно, если вы не манипулируете рынком, но даже у манипуляторов присутствуют риски). Вероятность роста или падения цены от текущего момента равна 50/50. Статистика показывает, что из 100% людей, которые делают ставку, 50% угадывает, 50% нет.

Задача биржи — взять 100% проигрышей, выплатить 80% от суммы выигравшим и 20% забрать себе. На деле доходы биржи гораздо выше, поскольку лудоманы НЕ выводят прибыль, увеличивая оборотные средства биржи, соответственно, и её доходы.

В случае проигрыша вы теряете 100% суммы. По этой причине математическое ожидание при ставке на бинарные опционы отрицательное, и, если вы хотите заработать на этом рынке, проходимость сделок должна быть в районе 60–65%.

Коротко опишем, что должна уметь будущая биржа:

1. Хранить и отображать баланс, имя, название программы.
2. Устанавливать имя пользователя.

3. Добавлять сумму на баланс.
4. Проводить аудит баланса и присваивать статус.
5. Рассчитывать потенциальный выигрыш.
6. Получать текущую котировку Биткойна.
7. Принимать пари.
8. Сохранять историю сделок.
9. Учитывать оборотку средств.
10. Учитывать прибыльные сделки.
11. Учитывать убыточные сделки.
12. Показывать меню для выбора действия.

Не забывайте, что решение задач, которые описаны в конце всех практических подразделов, является неотъемлемой частью вашего успеха. Чтобы научиться танцевать, нужно танцевать. Чтобы научиться программировать, нужно решать задачи и писать код.

Перед каждой новой темой я буду стараться прикладывать исходник с решённой предыдущей задачей. Постарайтесь решить задачи самостоятельно, прежде чем переходить к новой теме.

1.5. Комментарии

> Всё находящееся в исходнике после знака **#** является **комментарием**. Комментарии игнорируются интерпретатором и служат пометками для программистов.

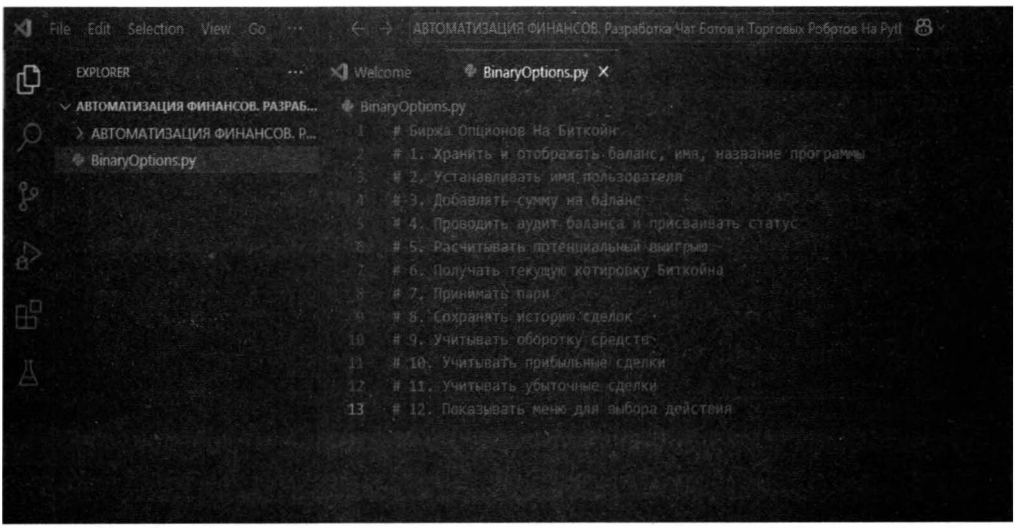
В частности, с помощью комментариев можно временно отключать куски кода. Мы будем использовать комментарии повсеместно.

Практически каждая строчка вашего кода должна быть прокомментирована. Это гарантия того, что после длительного перерыва вы легко восстановите свою память.

Задача 1.1

Перенести ТЗ в самое начало файла BinaryOptions.py в виде комментариев. Просто перепишите текст ТЗ и в начале каждой строки поставьте знак #. Это поможет контролировать как ещё нерешенные задачи, так и решённые.

Решение задачи 1.1:



Изображение 1.16.

1.6. Переменные

Если говорить техническим языком, то переменная — это именованная ячейка в оперативной памяти компьютера. Переменные хранят в себе состояние программы и очищаются после закрытия софта.

Представьте, что переменная — это подписанный коробок спичек, в котором лежат булавки, монетки, камушки и т.д.

Хорошее название переменной отражает данные, которые в ней находятся. К именам переменных нужно относиться как к именам своих детей. Хорошего программиста от плохого отличает навык именования переменных и функций.

Имена переменных НЕ могут начинаться с цифры, зависимы от регистра и позволяют хранить в себе различные типы данных.

Переменные бывают 4 основных типов:

1. **Строка** (String) — "Я строка, хочу в тетрадь".
2. **Целое число** (Integer) — 20000.
3. **Число с плавающей точкой** (Float) — 1.337.
4. **Логический** (Boolean) — True/False.

Существуют и другие типы данных, с которыми вы научитесь работать позднее.

Для того чтобы создать переменную, нужно сначала дать ей имя, затем с помощью оператора = присвоить ей значение.

Вернитесь к своему проекту **BinaryOptions.py** и допишите после ТЗ следующий код:

```
program_name = "Опционы на Биткойн" # Тип данных – строка
balance = 20000 # Тип данных – число
payout = 1.80 # Тип данных – число с плавающей точкой
verified = False # Логический тип. Истина True. Ложь False
full_name = None # Пустая переменная без значения
```

Я создал 5 переменных и в комментариях напротив каждой написал тип данных, к которому переменная относится.

Процесс создания переменных также называется объявлением.

Обратите внимание, что содержимое переменной `program_name` окружено двойными кавычками. Благодаря кавычкам предложение "Опционы на Биткойн" определяется Пайтоном как строка.

Содержимое переменной может быть перезаписано или изменено. Также переменные можно складывать, умножать, делить и т.д.

Допишите в исходник следующие строки:

```
balance = 30000 # Перезапись переменной balance.
balance = balance + 300 # Добавляем к предыдущему значению ещё 300
# В итоге получаем 30300
```

Когда вы только объявили переменную `balance`, то присвоили ей содержимое 20000. Строка `balance = 30000` заменит её предыдущее значение и сохранит новое.

Строка `balance = balance + 300` возьмет предыдущее значение переменной, добавит к ней целое число 300 и сохранит результат, заменив предыдущее значение.

Задача 1.2

Объявите пустую переменную для хранения всех сумм сделок.
Объявите пустую переменную для хранения выигрышных сделок.
Объявите пустую переменную для хранения проигрышных сделок.
Объявите пустую переменную для хранения архива сделок.

Решение задачи 1.2:

```
# ТЗ пропущено
program_name = "Опционы на Биткйон" # Тип данных — строка
balance = 20000 # Тип данных — число
payout = 1.80 # Тип данных — число с плавающей точкой
verified = False # Логический тип. Истина = True. Ложь = False
full_name = None # Пустая переменная без значения

balance = 30000 # Перезапись переменной balance
balance = balance + 300 # Предыдущее значение + ещё 300. В итоге
получаем 30300

# Решение задачи 1.2
all_deals = None # Заготовка для суммы всех сделок
win_deals = None # Заготовка для выигрышных сделок
lose_deals = None # Заготовка для проигрышей
deals_archive = None # Для архива сделок
```

1.7. Основные математические операторы

Операторы в программировании совершают действия над данными. Например, оператор + просто складывает 2 числа.

Основные математические операторы

\+ — сложение;

\- — вычитание;

* — умножение;

/ — деление;

// — деление с остатком;

** — возведение в степень;

% — остаток от деления.

Вы также можете перекрывать приоритет выполнения математических операций с помощью круглых скобок, как в школьной программе. После изучения следующего подраздела найдите время на эксперименты с математическими операторами. Для экспериментов создавайте отдельные проекты и давайте им соответствующие названия.

1.8. Функции. Используем микропрограммы

Функции — это мини-программы (или подпрограммы), которые выполняют 1 конкретное действие. Например, функция *print()* просто печатает на экране переданные ей данные.

Функции создаются для повторного использования кусков кода, без копирования исходника. Функции помогают сделать код семантически изящным и переносимым из проекта в проект. Из функций можно построить проект любого масштаба.

Основные свойства функций

1. Функции принимают входящие данные в переменные аргумента.
2. Функции обрабатывают данные из переменных аргумента.
3. Функции отдают результат в переменные или другие функции.
4. Имя функции отражает действие, которое совершает.
5. Рекурсивность. Функция может вызывать саму себя.

Мы подробно изучим каждое свойство на практике, научимся работать со встроенными функциями и создавать собственные.

Для начала изучим две встроенные в Пайтон функции — print() и input().

1.9. Функция *print()*. Выводим сообщения на экран

Функция print() просто печатает на экране переданные ей данные. Откройте исходник, в самый конец добавьте следующий код и запустите.

```
# Пример работы функции print()
print(program_name) # Выводим содержимое переменной program_name
print("Баланс до начисления:", balance) # Передаём 2 аргумента
print("Коэффициент выигрыша:", payout) # Передаём 2 аргумента
```

Мы добавили 3 строчки с выводом данных на экран.

Входящие данные, которые передаются функции, называются аргументами.

В первой строке мы просто вывели содержимое переменной `program_name`, передав её в качестве аргумента функции print().

Во второй строке через запятую передали функции 2 аргумента:

» 1 аргумент — строка без переменной `"Баланс до начисления:"`.

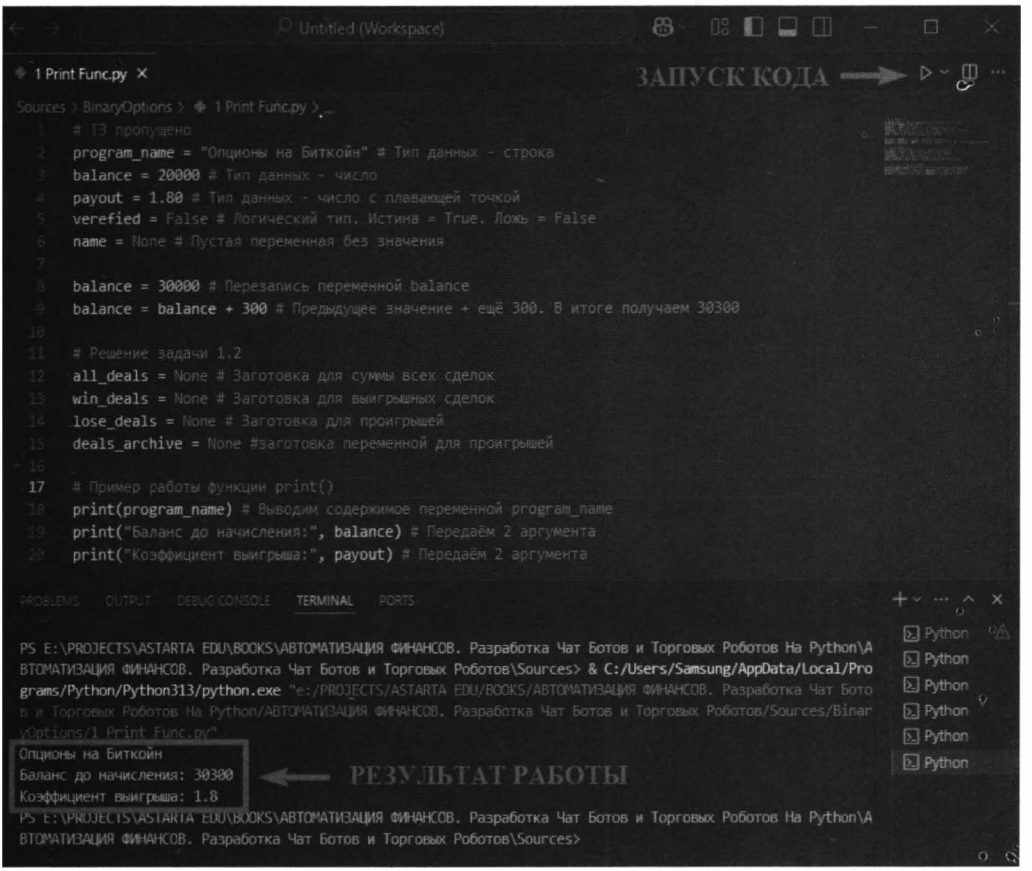
» 2 аргумент — переменная `balance`.

В третьей строке передали 2 аргумента:

» 1 аргумент — строка без переменной `"Коэффициент выигрыша:"`.

» 2 аргумент — переменная `payout`.

Запустите исходник. Если всё сделано правильно, вы увидите содержимое, представленное на скриншоте:



Изображение 1.17.

Задача 1.3

Самостоятельно выведите статус верификации пользователя.

Самостоятельно выведите коэффициент выигрыша.

Самостоятельно выведите имя пользователя.

Решение задачи 1.3:

```
print("Статус верификации:", verified) # Передаём два аргумента
print("Коэффициент выигрыша:", payout) # Передаём два аргумента
print("Имя пользователя:", full_name) # Передаём два аргумента
```

1.10. Функция *input()*. Принимаем ввод от пользователя

Скучно, если нет возможности общаться с пользователем. Функция input() ожидает ввод от пользователя и сохраняет введённые данные в переменной или отдаёт другой функции. Функция input() принимает только один аргумент — строку. Переданная в качестве аргумента строка будет отображена для пояснения, какие данные мы хотим получить от пользователя.

Функция input() дождётся, пока пользователь введёт данные, затем сохранит введённые данные в переменной или отдаст другой функции. Функция input() всегда отдаёт результат работы в виде строки, даже если ввести число. При вводе числа 3 оно сохранится как "3". В конце основного исходника допишите следующие строчки и сразу запустите на тестирование.

```
# Сохранение данных из функции input() в переменную
# Берём у пользователя имя и перезаписываем переменную full_name
full_name = input("Введите свое имя>>>")
print("Приветствую", full_name)
```

Благодаря этому коду мы перезаписали переменную *full_name*, которую объявили пустой в начале исходника. Если всё сделано правильно, вы увидите содержимое, представленное на скриншоте.

Изображение 1.18.

Поздравляю! Вы освоили базовое общение с машиной посредством языка программирования Python.

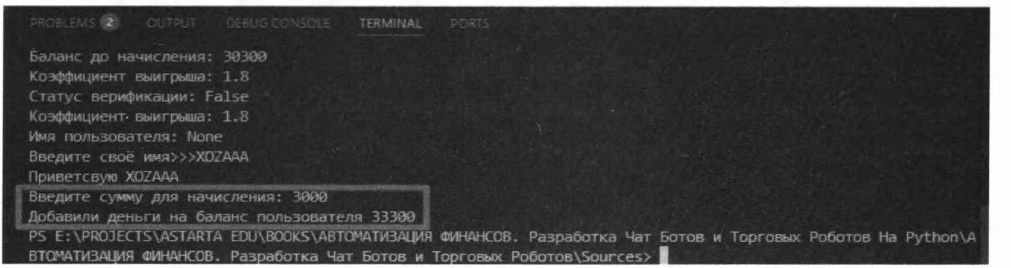
1.11. Преобразование типов данных

Напомню, что в Пайтоне имеется четыре основных типа данных: строка (String), целое число (Integer), число с плавающей точкой (Float) и логический тип (Boolean). Из-за динамической типизации вы должны самостоятельно следить за типами данных, находящихся в ваших переменных, поскольку нельзя сложить число со строкой, так же как умножить логический тип на число с плавающей точкой. Это алогично.

Для преобразования в основные типы данных существуют встроенные в Пайтон функции: str(), int(), float(), boolean(). По названиям функций легко можно предположить, в какой тип они преобразуют данные, полученные в аргументе. Напомню, функция input() всегда отдаёт результат своей работы в виде строки. Даже если мы ввели число 30, то на выходе получим "30", соответственно, при попытке сложить строку с числом вы получите ошибку интерпретатора. Основные ошибки, которые потенциально могут возникнуть в процессе выполнения кода, вы легко найдёте и исправите с помощью поисковика. Давайте попытаемся начислить к текущему балансу ещё средств. В конце исходника вашего основного проекта допишите следующее строчки:

```
# Ввели в функцию input() 30, получили "30"
# Если ввести число, всё равно получится строка.
user_sum = input("Введите сумму для начисления: ")
user_sum = int(user_sum) # Преобразуем в тип Integer
balance = balance + user_sum # Обновляем переменную balance
print("Добавили деньги на баланс пользователя", balance)
```

Результат работы данного кода:



Изображение 1.19.

Вы познакомились с некоторыми функциями, встроенными в Пайтон, и их основными свойствами. Научились выводить данные на экран, получать данные от пользователя и сохранять их в переменных. Также научились преобразовывать типы данных в случае необходимости. Этих знаний достаточно, чтобы начать писать собственные функции.

1.12. Пишем собственные функции

Напомню, функции — это мини-программы (или подпрограммы), которые выполняют одно конкретное действие. Функции создаются для повторного использования кусков кода, помогают сделать код семантически изящным и переносимым из проекта в проект.

Ещё раз перечислю основные свойства функций:

1. Функции принимают входящие данные в переменные аргумента.
2. Функции обрабатывают данные из переменных аргумента.
3. Функции отдают результат в переменные или другие функции.
4. Имя функции отражает действие, которое совершает.
5. Рекурсивность. Функция может вызывать саму себя.

Чтобы объявить функцию, необходимо воспользоваться ключевым словом def, затем дать ей осознанное название, добавить *круглые скобки* и *знак двоеточия*. Это укажет интерпретатору, что объявление закончено. После объявления при нажатии клавиши Enter вы попадёте внутрь функции. Внутренняя часть функции называется телом.

Функция не может существовать без тела. Для указания того, что функция объявлена и пустая, используется ключевое слово **pass**. После объявления функции её можно вызвать сколько угодно раз в любой части исходника (в том числе в других функциях), просто написав имя функции с круглыми скобками в конце.

Не забывайте, что вызов функции должен происходить после объявления, поскольку код на Пайтон исполняется сверху вниз.

При объявлении функции хорошим тоном является описание её работы в комментариях. Это позволит вашим коллегам легче вникать в принцип работы функции. Чем проще и приятнее поддерживать ваш код, тем больше уважения вы получите от коллег.

Ниже приведён демонстрационный пример объявления функции и последующего её вызова (несколько раз). Пример нет необходимости переписывать.

```
def get_open_interest():
    # Функция получает индикатор открытого интереса
    # Не принимает аргументов
    # Не отдаёт результат
    pass # Означает, что функция объявлена, но пустая

get_open_interest() # Вызываем функцию первый раз
get_open_interest() # Вызываем функцию второй раз
get_open_interest() # Вызываем функцию третий раз
```

Содержимое тела функции должно быть отбито табуляцией или 4 пробелами. Как уже говорилось в предисловии, одним из недостатков лаконичного синтаксиса Пайтон является необходимость следить за вложенностью в тело конструкций. Если вы смешаете табуляции и пробелы, то получите ошибку.

Далее, мы воспользуемся приёмом, который часто используют художники и называют "от общего к частному". Удобно в начале проекта описать пустыми все функции, которые могут понадобится в соответствии с ТЗ, а затем их реализовывать.

В листинге 1.1 представлен код с решёнными заданиями из прошлых разделов и объявлением пустых функций, необходимых для работы биржи. Заготовленные функции соответствуют техническому заданию, НЕ представленному в листинге.

Приведите ваш основной исходник к тому же виду. Не переживайте, если участки кода с переменными аргумента пока неясны.

Листинг 1.1

```
# ТЗ пропущено
program_name = "Опционы на Биткойн" # Тип данных - строка
balance = 20000 # Тип данных - число
payout = 1.80 # Тип данных - число с плавающей точкой
verified = False # Логический тип. Истина = True. Ложь = False
full_name = None # Пустая переменная без значения

balance = 30000 # Перезапись переменной balance
balance = balance + 300 # Предыдущее значение + ещё 300
# В итоге в balance получаем 30300

# Решение задачи 1.2
all_deals = None # Заготовка для суммы всех сделок
win_deals = None # Заготовка для выигрышных сделок
lose_deals = None # Заготовка для проигрышей
deals_archive = None # Для архива сделок

# Пример работы функции print()
print(program_name) # Выводим содержимое переменной program_name
print("Баланс до начисления:", balance) # Передаём 2 аргумента
print("Коэффициент выигрыша:", payout) # Передаём 2 аргумента

# Решение задачи 1.3
print("Статус верификации:", verified) # Передаём два аргумента
print("Коэффициент выигрыша:", payout) # Передаём два аргумента
print("Имя пользователя:", full_name) # Передаём два аргумента

# Сохранение данных из функции input() в переменную
# Берём у пользователя имя и перезаписываем переменную full_name
full_name = input("Введите свое имя>>>")
print("Приветствую", full_name)

# Ввели в функцию input() 30, получили "30"
# Если ввести число, всё равно получится строка.
user_sum = input("Введите сумму для начисления: ")
user_sum = int(user_sum) # Преобразуем в тип Integer
balance = balance + user_sum # Обновляем переменную balance
print("Добавили деньги на баланс пользователя", balance)

# Заготовки функций биржи
def show_info():
    # Функция отображает данные на программу
    # Не принимает аргументов
    # Не отдаёт результат
    pass # PASS означает, что функция объявлена, но пустая
```

```
def set_user_name(first_name, last_name):
    # Функция устанавливает полное имя пользователя
    # Принимаем два аргумента: имя и фамилию
    # Отдаёт введенное имя
    pass

def add_sum_to_balance(sum):
    # Функция добавляет деньги на баланс
    # Принимает 1 аргумент — сумму добавления
    # Отдаёт текущий баланс
    pass

def audit_balance():
    # Функция анализирует баланс пользователя и присваивает статус
    # Не принимает аргументов
    # Не отдаёт результат
    pass

def get_btc_price():
    # Функция получает котировку биткойна по API
    # Не принимает аргументы
    # Отдаёт полученную цену
    pass

def win_calculate(sum):
    # Функция рассчитывает потенциальную сумму выигрыша
    # Принимает 1 аргумент
    # Отдаёт результат выигрыша
    pass

def create_deal(sum, time, duration):
    # Функция создаёт сделку
    # Принимает 3 аргумента: сумму сделки,
    # направление цены и время экспирации
    # Ничего не отдаёт
   pass

def show_statistic():
    # Функция показывает статистику сделок
    # Не принимает значения
    # Ничего не отдаёт
    pass

def show_win_deals():
    # Функция показывает статистику выигрышных сделок
    # Не принимает значения
    # Ничего не отдаёт
    pass
```

```
def show_loses_deals():
    # Функция показывает статистику проигрышных сделок
    # Не принимает значения
    # Ничего не отдаёт
    pass

def menu():
    # Функция отображает меню для управления биржей
    # Не принимает аргументов и не отдаёт результат
    pass # PASS означает, что функция объявлена, но пустая
```

Мы объявили все необходимые функции для работы будущей биржи. Дали им осознанные, симпатичные названия, которые легко читаются и чётко описывают действия, которые они совершают. Также определили переменные аргумента, в которые будут попадать входящие данные для внутренней обработки. Подробнее о переменных аргумента вы узнаете через одну подглаву. Самое главное, мы прокомментировали работу каждой функции, что сделает нашу работу более простой и приятной. В целом провели потрясающую подготовительную работу!

Постепенно вы реализуете каждую функцию, параллельно изучая новые конструкции и возможности языка Пайтон.

1.13. Область видимости

Прежде чем приступить к описанию полноценных функций, необходимо познакомиться с понятием области видимости.

Областей видимости существует две:

1. **Глобальная область видимости** — это область, которая находится за пределами функций. В нашем случае это все переменные, которые мы описали до объявления функций.
2. **Локальная область видимости** — это область, ограниченная только функцией.

Из этого следует, что переменная *balance*, объявленная за пределами функции, и переменная *balance*, объявленная внутри функции, являются разными переменными! Ниже представлен демонстрационный пример.

```
# ГЛОБАЛЬНАЯ ОБЛАСТЬ ВИДИМОСТИ
balance = 20000 # Это глобальная переменная

def show_info():
    # ЛОКАЛЬНАЯ ОБЛАСТЬ ВИДИМОСТИ
    # Функция отображает данные на программу
    # Не принимает аргументов
    # Не отдаёт результат
    balance = 1000 # Эта локальная переменная
```

Для того чтобы "подтянуть" переменную из глобальной области видимости внутрь области видимости функции, сразу после слова `global` необходимо перечислить имена нужных переменных. Можно использовать как построчный импорт, так и импорт через запятую.

Существует нюанс. Если вы хотите просто получить доступ к содержимому переменной, находящейся в глобальной области видимости (без изменения), то нет необходимости в дополнительных действиях. Просто напишете её имя и если интерпретатор НЕ найдет переменную в локальной области видимости, то попытается отыскать её в глобальной. Если стоит задача изменить содержимое из глобальной области видимости, в этом случае использование ключевого слова `global` и перечисление после него всех необходимых переменных обязательно!

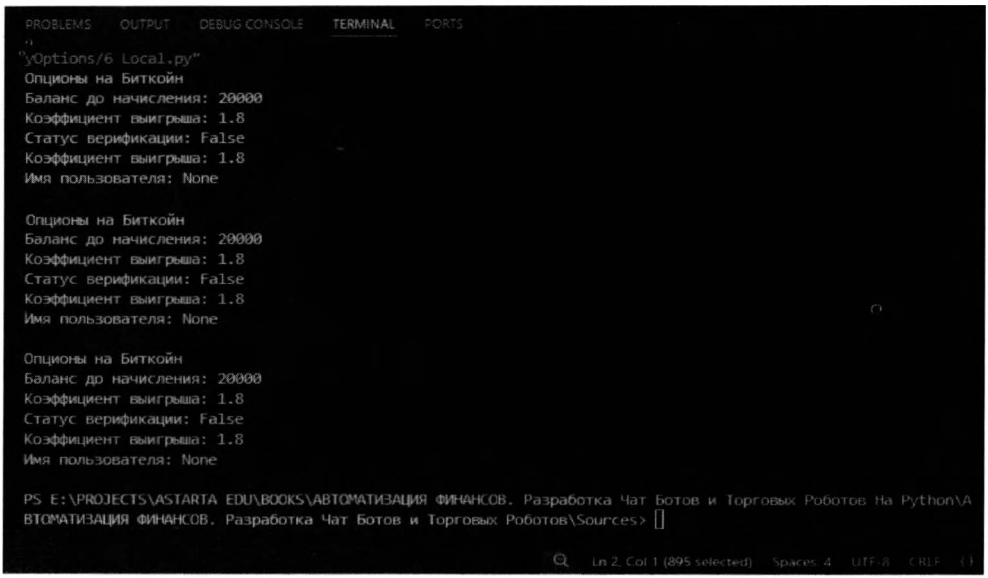
Для наглядности в наших примерах мы будем использовать объявление глобальных переменных практически всегда, когда потребуется к ним обратиться. Теперь давайте подумаем, что делать, если нам понадобится многократно показывать информацию о текущем состоянии программы (название программы, баланс, имя пользователя и т.д.)? Допустим, мы хотим видеть эту информацию при запуске биржи, после пополнения баланса и после исполненной сделки. Если вы подумали, что необходимо объявить функцию и использовать её, где вздумается, то совершенно правы, если же нет, срочно исправим это!

Давайте просто перенесём код, ответственный за вывод информации, в функцию show_info(). Вырежьте код, ответственный за информацию о программе, и вставьте в функцию show_info(). Вы получите примерно следующее:

```
# ГЛОБАЛЬНАЯ ОБЛАСТЬ ВИДИМОСТИ
program_name = "Опционы на Биткойн" # Тип данных – строка
```

```
balance = 20000 # Тип данных — число
payout = 1.80 # Тип данных — число с плавающей точкой
verified = False # Логический тип. Истина = True. Ложь = False
full_name = None # Пустая переменная без значения

def show_info():
    # ЛОКАЛЬНАЯ ОБЛАСТЬ ВИДИМОСТИ
    # После слова global импортируем внешние переменные
    global program_name, balance, payout, verified, full_name
    print(program_name) # Выводим содержимое переменной program_name
    print("Баланс до начисления:", balance) # Передаём 2 аргумента
    print("Коэффициент выигрыша:", payout) # Передаём 2 аргумента
    print("Статус верификации:", verified) # Передаём два аргумента
    print("Коэффициент выигрыша:", payout) # Передаём два аргумента
    print("Имя пользователя:", full_name) # Передаём два аргумента
    print() # Пустая строка для разделителя

show_info() # Вызов функции первый раз
show_info() # Вызов функции второй раз
show_info() # Вызов функции третий раз
```

Сразу после объявления функции вызовите её несколько раз, дабы наглядно увидеть, что нет необходимости копировать код, чтобы использовать его повторно. В будущем постарайтесь организовать свой исходный код таким образом, чтобы объявление функций всегда было в начале файла, а тестирование в конце. Благодаря этому при большом количестве кода вы НЕ будете тратить время на поиск места, в котором запущена та или иная функция.



Изображение 1.20.

Поздравляю, вы создали и использовали свою первую функцию!

1.14. Переменные аргумента

Аргумент функции — это переменная (или несколько переменных), которая принимает входящие значения для дальнейшей обработки внутри функции. Чтобы научить функцию принимать аргументы, при её объявлении в круглых скобках необходимо указать название переменной. Это и будет переменная аргумента. Переданные данные в переменные аргумента называются параметрами, но мы не будем заострять на этом термине внимание.

Давайте реализуем функцию set_user_name(). Вырежьте код, который отвечал за установление имени пользователя, из глобальной области видимости и адаптируйте его внутри функции.

```
def set_user_name(first_name, last_name):
    # Функция устанавливает имя пользователя
    # Принимаем имя и фамилию в качестве аргументов
    # Возвращает полное имя
    global full_name  # Обращаемся к глобальной переменной
    print("Текущее имя:", full_name)
    full_name = first_name + " " + last_name  # Склеиваем строки
    print("Новое имя пользователя:", full_name)

set_user_name(first_name="Даниил", last_name="Бакалов")
set_user_name(first_name="Аарон", last_name="Шварц")
set_user_name(first_name="Питер", last_name="Тилль")
```

```
Текущее имя: None
Новое имя пользователя: Даниил Бакалов

Текущее имя: Даниил Бакалов
Новое имя пользователя: Аарон Шварц

Текущее имя: Аарон Шварц
Новое имя пользователя: Питер Тилль

PS E:\PROJECTS\ASTARTA EDU\BOOKS\АВТОМАТИЗАЦИЯ ФИНАНСОВ. Разработка Чат Ботов и Торговых Роботов На Python\А
ВТОМАТИЗАЦИЯ ФИНАНСОВ. Разработка Чат Ботов и Торговых Роботов\Sources>
```


Изображение 1.21.

В комментариях хорошо описано, как устроена функция и где находятся переменные аргумента. Теперь, поскольку мы объявили две переменные аргумента, при вызове функции мы обязаны передать им значения. Переменных аргумента может быть сколько угодно, соответственно, каждая из них должна быть заполнена при вызове функции.

Переданные на вход значения будут сохранены в переменных аргумента (в данном случае это *first_name* и *last_name*) и использованы в теле функции.

Строка `full_name = first_name + " " + last_name` добавит пробел между именем и фамилией и перезапишет глобальную переменную `full_name`, поскольку мы предварительно её импортировали.

Процесс склеивания строк между собой называется конкатенацией.

Обратите внимание, что переменная аргумента принимает данные в том виде, в котором они поступают. То есть если мы передадим функции переменную, в которой лежит строка, то в переменной аргумента тоже будет находиться строка.

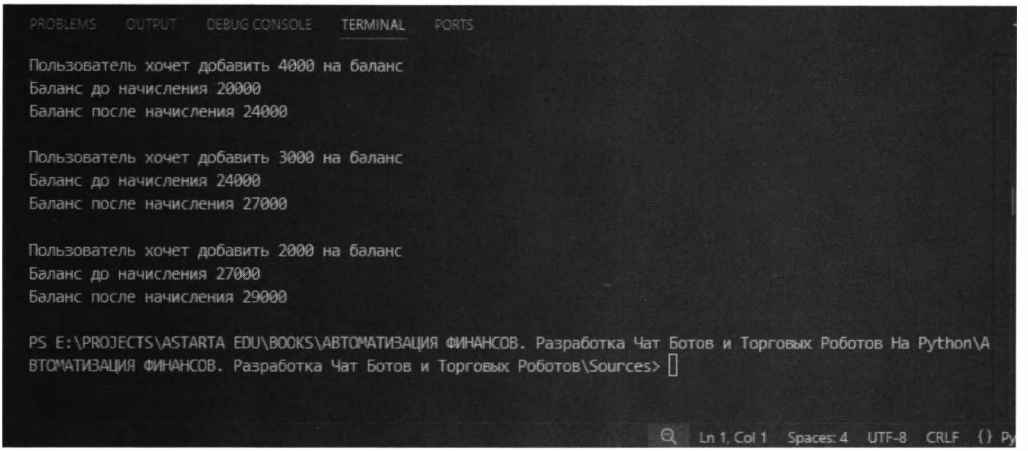
Задача 1.4

Попробуйте реализовать функцию add_sum_to_balance(). Функция должна принимать один аргумент (сумма пополнения), затем подтягивать переменную `balance` из глобальной области видимости и добавлять к текущему балансу.

Решение задачи 1.4:

```
def add_sum_to_balance(sum):
    # Добавляет деньги на баланс
    # Принимает 1 аргумент - сумму пополнения
    # Возвращает текущий баланс
    global balance # Подтягиваем переменную из глобальной области
    print("Пользователь хочет добавить", sum, "на баланс")
    print("Баланс до начисления", balance)
    balance = balance + sum # Суммируем с предыдущим значением
    print("Баланс после начисления", balance)
```

```
add_sum_to_balance(sum=4000) # Первый вызов функции
add_sum_to_balance(sum=3000) # Второй вызов функции
add_sum_to_balance(sum=2000) # Третий вызов функции
```



Изображение 1.22.

1.15. Отдаём результат работы функции

Мы подобрались к предпоследнему основному свойству функций, понимание которого понадобится для эффективного программирования. Представим ситуацию, в которой нам необходимо перехватить результат работы функции.

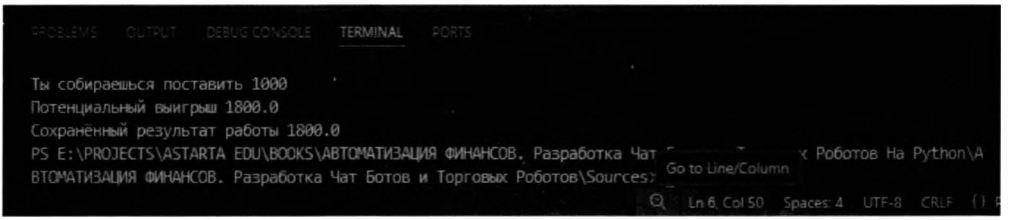
Допустим, при создании ордера мы хотим рассчитать потенциальный выигрыш и использовать полученный результат в дальнейшем. Давайте опишем функцию win_calculate() в основной части проекта.

```
def win_calculate(sum):
    # Функция рассчитывает потенциальную сумму выигрыша
    # Принимает 1 аргумент
    # Возвращает результат выигрыша
    global payout # Импортируем переменную из глобальной области
    result = int(sum) * payout # Умножаем сумму сделки на кэф
    print("Ты собираешься поставить", sum)
    print("Потенциальный выигрыш", result)

win_calculate(sum=3000) # Тестируем функцию сразу после объявления
```

Функция работает отлично, но мы не можем сохранить результат её работы в промежуточной переменной или передать результат другой функции.

Для того чтобы отдать результат работы функции, существует оператор **return**, после которого указывается значение, которое функция должна отдать. В самом конце функции win_calculate() напишите `return result`.

```
def win_calculate(sum):
    # Функция рассчитывает потенциальную сумму выигрыша
    # Принимает 1 аргумент — сумму ставки
    # Возвращает результат выигрыша
    global payout # Импортируем переменную из глобальной области
    result = int(sum) * payout # Умножаем сумму сделки на коэффициент
    print("Ты собираешься поставить", sum)
    print("Потенциальный выигрыш", result)
    return result # Отдаём результат работы функции

potential_profit = win_calculate(sum=1000) # Вызов функции и сохранение
print("Сохраненный результат работы", potential_profit) # Вывод результата
```



Изображение 1.23.

Теперь можно сохранить работу функции в переменной, как это сделано в случае со строкой `potential_profit = win_calculate(sum=1000)`. Функция win_calculate() отработает и отдаст результат своей работы в переменную `potential_profit`. Аналогичным образом можно передавать результат в другие функции.

Например, так: float(int(potential_win())). Читается справа налево. Сначала отработает функция potential_win() и отдаст результат функции int(), затем функция int() отдаст результат своей работы в функцию float().

Обратите внимание, что, помимо того, что оператор **return** отдаёт результат работы функции, он ещё и останавливает её работу. Ниже приведён пример кода, в котором всё, что находится после вызова оператора **return**, не исполнится.

```
def stop():
    print("Старт работы функции")
    return # Останавливаем функцию, ничего не отдав
    print("Эту строку вы уже не увидите!")

stop() # Вызов функции
```

Мы договоримся о том, что будем вызывать оператор **return** в конце каждой функции, даже если она не отдаёт результата. Это необходимо для того, чтобы гарантировать завершение функции при рекурсивной работе.

1.16. Рекурсия. Самоперезапуск функции

Последней деталью в освоении функций станет изучение понятия рекурсии. Надеюсь, вы НЕ забыли, что в вашем исходнике присутствует глобальная область видимости и локальная. Глобальная область видимости — это всё, что находится за пределами тела функций, локальная относится только к содержимому самой функции.

Из этого можно сделать вывод, что в контексте выполнения функции она сама себя "осознаёт", и мы можем вызвать функцию внутри её самой.

Создайте новый проект и реализуйте рекурсивную функцию **crazy_strings()**.

```
def crazy_strings():
    print("Я строка, хочу в тетрадь!") # Выводим результат на экран
    crazy_strings() # Рекурсивно вызываем функцию
    return # Завершаем текущее выполнение функции, ничего не отдав

crazy_strings() # Запускаем функцию
```

Функция хорошо прокомментирована и не требует дополнительных объяснений работы. Максимальное количество раз, которое функция может вызвать саму себя, — 1000. Это значение можно изменить, но проще учитывать стандартное поведение.

Немного позже мы используем рекурсию для обработки ошибок ввода и "бесконечного" перезапуска меню.

Хотите верьте, хотите нет, если вы добрались до текущего раздела и самостоятельно смогли решить хотя бы часть задач, вас уже можно считать действующим программистом. Вы способны решить немалое количество рутинных задач. Осталась буквально пара конструкций, которые увенчают ваш навык программиста.

1.17. Логика и Ветвления. Делаем функции умнее. Конструкции IF: ELIF: ELSE:

Многие считают, что для эффективного программирования необходимо обладать недюжинными знаниями математики и несколько лет проучиться на инженера. Это не так. Для большинства задач достаточно обладать знаниями математики, который каждый получает в первых трех классах школы, и иметь малейшее представление о логике.

В данном разделе мы научимся сравнивать значения и переменные между собой и на базе сравнений производить гибкие действия.

Для работы с ветвлениями используется конструкция IF: ELIF: ELSE: и операторы сравнения, перечисленные ниже:

== равно
!= не равно
> больше
< меньше
>= больше или равно
<= меньше или равно
AND и
OR или
NOT нет

1.18. Конструкция IF: (Что, если?)

Давайте договоримся, что отныне каждую новую конструкцию вы будете тестировать в отдельном проекте. Если пытаться тянуть изучение новых

конструкций или библиотек кода (глава 1.21) в основной проект, можно надолго увязнуть.

Профессионалы всегда изучают новое для себя направление на коротеньких примерах, в отдельных микропроектах. Создайте новый файл, назовите его LOGICTest.py и введите следующий код:

```
btc_price = 30000 # Объявляю переменную с текущей стоимостью BTC

if btc_price == 30000: # Если проверка вернет истину (True)
   print("Продаю Ламбу, покупаю Биткойн") # Выполнится этот блок кода

>>> Продаю Ламбу, покупаю Биткойн
```

Конструкция IF: всегда возвращает истину или ложь.

Благодаря лаконичности языка Пайтон можно легко предположить, что происходит в коде. Дословно мы сравниваем, является ли содержимое переменной `btc_price` равным 30000. Исключительно в том случае, если сравнение является истиной и суммы равны, выполнится строка `print("Продаю Ламбу, покупаю Биткойн".`

1.19. Конструкция ELIF:

Давайте попробуем немного развить логику нашей программы. Наверняка мы бы хотели по-разному отреагировать на различные цены. Для дополнительной проверки используется конструкция ELIF:.

Обратите внимание, что конструкция ELIF: является необязательной и используется тогда, когда входящие данные нужно проверить на множество соответствий.

```
btc_price = 60000 # Объявляю переменную с текущей стоимостью Биткойна

if btc_price == 30000: # Если проверка вернет истину (True)
   print("Продаю Ламбу, покупаю Биткойн") # Выполнится эта строка
elif btc_price == 60000: # В данном случае выполнится эта часть
   print("Фиксирую прибыль ")
elif btc_price == 100000:
   print("Жду коррекции")
```

Стало значительно гибче, теперь в зависимости от стоимости Биткойна вы можете принимать решения и печатать соответствующие сообщения. Важно запомнить, что выполнится та часть проверки, которая первой вернула истину, остальные блоки будут проигнорированы.

Данный пример имеет очевидный недостаток, даже если мы будем обращаться за ценой Биткойна каждую секунду, есть шанс пропустить необходимую нам цену. Давайте воспользуемся вспомогательным оператором AND для указания диапазона, в котором мы хотим получать сигналы.

```
btc_price = 30000 # Объявляю переменную с текущей стоимостью Биткойна

if btc_price >= 30000 and btc_price < 59000:
    # Если цена больше или равна 30000 и при этом меньше 59000
    print("Продаю Ламбу, покупаю Биткойн") # Выполнится эта строка
elif btc_price >= 69000 and btc_price < 76000:
    # Если цена больше или равна 69000 и при этом меньше 76000
    print("Время фиксировать прибыль")
elif btc_price >= 100000: # Если цена больше или равна 100000
    print("Стоит дождаться коррекции")
```

Теперь можно считать, что мы написали практически полноценного помощника, который будет советовать нам, когда заходить в рынок, фиксировать прибыль или ждать коррекции.

1.20. Конструкция ELSE:

Последней деталью в изучении логики станет конструкция ELSE:. Данная конструкция срабатывает тогда, когда ни одна из предыдущих проверок не вернула True. В нашем простом примере это случай, когда цена Биткойна ниже 30000. Если мы пропустили все сигналы из предыдущих примеров, самое время найти дополнительные средства на покупку Биткойна!

```
btc_price = 30000 # Объявляю переменную с текущей стоимостью
биткойна

if btc_price >= 30000 and btc_price < 59000:
    # Если цена больше или равна 30000 и при этом меньше 59000
    print("Продаю Ламбу, покупаю биткойн") # Выполнится эта строка
elif btc_price >= 69000 and btc_price < 76000:
    # Если цена больше или равна 69000 и при этом меньше 76000
```

```
    print("Время фиксировать прибыль")
elif btc_price >= 100000:
    # Если цена больше 100000
    print("Стоит дождаться коррекции")
else:# Обработка всех остальных случаев
    print("Биткойн упал ниже плинтуса!")
```

Обратите внимание, что любая конструкция IF: запускает собственную цепочку проверок и конструкции ELIF: и ELSE: работают только в связке с конструкцией IF:.

Конструкцию IF: ELIF: ELSE: можно использовать как самостоятельно, так и внутри функций. При описании более сложной логики старайтесь избегать большого количества вложенностей, это не сделает ваш код изящнее.

Обязательно создайте ещё один проект для собственных экспериментов и попробуйте поработать с каждым оператором сравнения самостоятельно!

Как мной уже было упомянуто, успех в обучении программированию зиждется на 3 китах: 1 — переписывать примеры, 2 — решать задачи, 3 — экспериментировать.

Задача 1.5

В основном проекте реализуйте функцию audit_balance() таким образом, чтобы на экране отображался статус пользователя в зависимости от его депозита.

- Если баланс равен 0 — показываем "Ты банкрот".
- Если баланс меньше или равен 1000 — присваиваем статус "Бронза".
- Если баланс больше 1000 и меньше 5000 — присваиваем статус "Серебро".
- Если баланс больше 5000, но меньше 15000 — присваиваем статус "Золото".
- Во всех остальных случаях присваиваем статус "Платина".

Решение задачи 1.5:

```
def audit_balance():
    # Функция анализирует баланс пользователя и присваивает статус
    # Не принимает аргументов
    # Не отдаёт результат
    global balance # Импортируем переменную из глобальной области

    if balance == 0: # Проверка на True или False
        print("Ты банкрот") # Выполнится этот кусок кода
    elif balance <= 1000: # Если balance меньше или равнен 1000
        print("Ты бронза")
    elif balance > 1000 and balance < 5000:
        # Если balance больше 1000 и меньше 5000
        print("Ты серебро")
    elif balance > 5000 and balance < 15000:
        # Если balance больше 5000 и меньше 15000
        print("Ты золото")
    else: # Если предыдущие проверки вернули False
        print("Ты платина, твой баланс", balance)
    return # Завершаем функцию
```

Задача 1.6

Перепишите функцию add_sum_to_balance() таким образом, чтобы при добавлении на баланс происходила проверка значения, переданного в аргументах.

Если сумма меньше или равна 0, мы должны показать пользователю уведомление "Нельзя добавить 0 долларов и меньше" и остановить программу с помощью оператора **return**.

В случае если всё прошло хорошо, добавляем пользователю сумму на баланс и вызываем функцию audit_balance().

Решение задачи 1.6:

```
def add_sum_to_balance(sum):
    # Добавляет денег на баланс
    # Принимает 1 аргумент — сумму добавления
    # Возвращает текущий баланс
```

```
global balance # Импортируем глобальную переменную
print("Пользователь хочет добавить", sum, "на баланс")
print("Баланс до начисления", balance)

if int(sum) <= 0: # Решение задачи 1.6
   # Если сумма депозита меньше или равно 0
   print("Нельзя добавить 0 долларов и меньше")
   return # Останавливаем функцию аварийно

# Если функция не была остановлена аварийно
balance = balance + sum # Начисляем на баланс
print("Баланс после начисления", balance)
audit_balance() # Вызываем функцию аудита баланса
return balance # Завершаем функцию и отдаём баланс
```

Задача 1.7

Реализуйте функцию menu() таким образом, чтобы она печатала на экран пункты для выбора функционала биржи, затем спрашивала у пользователя номер пункта и в зависимости от выбора запускала соответствующую функцию.

- Если пользователь выбрал цифру 1 — вызываем функцию add_sum_to_balance().
- Если пользователь выбрал цифру 2 — запускаем работу с ордером функцией create_deal().
- Если пользователь выбрал цифру 3 — запускаем статистику функцией show_statistic().
- Если пользователь выбрал цифру 4 — запускаем функцию установки имени set_user_name().
- Если пользователь выбрал цифру 5 — прощаемся с игроком и вызываем функцию exit().
- Если пользователь выбрал цифру и её нет в указанном диапазоне, уведомляем об этом.

Решение задачи 1.7:

```
def menu():
    # Функция отображает меню для управления биржей
    # Не принимает аргументов
    # Не отдаёт результат
    print("Привет. Выбери один из пунктов")
    print("1 если хочешь добавить денег на баланс")
    print("2 если хочешь сделать сделку")
    print("3 если хочешь посмотреть статистику")
    print("4 если хочешь установить имя")
    print("5 если хочешь уйти")

    user_choise = input("Введи пункт: ") # Ждём ответ пользователя
    user_choise = int(user_choise) # Преобразовываем в число

    # Проверяем введённые пользователем данные на соотвествие
    if user_choise == 1:
        add_sum = input("Введи сумму для начисления")
        add_sum = int(add_sum) # Преобразовываем в число
        add_sum_to_balance(sum=add_sum) # Передаём в качестве аргумента
    elif user_choise == 2:
        # Тут лучше сразу взять данные у пользователя через input()
        create_deal(sum=1000, time=5, duration=0 # Данные для заглушки
    elif user_choise == 3:
        show_statistic() # Показываем статистику
    elif user_choise == 4:
        first_name = input("Введи имя") # Берём имя
        last_name = input("Введи фамилию") # Берём фамилию
        set_user_name(first_name=first_name, last_name=last_name)
    elif user_choise == 5:
        print("До свидания")
        exit() # Встроенная функция для выхода
    else:
        print("Ты выбрал, не верный пункт. Перезапускаю меню")
    return # Завершаем функцию
```

1.21. Библиотеки функций

В текущем подразделе мы изучим работу с библиотеками функций, которые уже встроены в Пайтон. Чуть позже вы научитесь устанавливать и использовать сторонние библиотеки.

Библиотеки объединяют в себе узконаправленные функции, которые инженеры создавали при решении тех или иных задач, а затем поделились с

сообществом. Большая часть библиотек распространяется на безвозмездной основе, и все высокотехнологичные корпорации, генерирующие прибыль, основаны на труде энтузиастов.

Для более наглядного примера перечислю некоторые библиотеки, которые мы будем активно использовать:

- Библиотека **random** — предлагает готовые функции для рандомизации чисел.
- Библиотека **datetime** — предлагает функции и объекты (глава 2) для работы со временем.
- Библиотека **os** — помогает работать с операционной системой.
- Библиотека **sqlite3** — поставляет порт для работы с базой данной SQLite.

Это лишь малая часть доступных библиотек из коробки. Обратитесь к поисковым системам, чтобы больше узнать о стандартных библиотеках языка Пайтон. Также рекомендую заранее ознакомиться с наиболее популярными внешними библиотеками, которые требуют установки.

Создайте новый проект под названием LIBRARIESTest.py и перепишите следующий код:

```
# Библиотеки — это наборы узконаправленных готовых функций
# Из библиотеки RANDOM импортируем функцию RANDINT
from random import randint

# Из библиотеки TIME импортируем функцию SLEEP
from time import sleep

# Из библиотеки DATETIME импортируем объект DATETIME
from datetime import datetime

# Пример работы с библиотекой RANDOM
result_int = randint(0, 1000)  # Возвращает целое число в диапазоне
print(result_int) # Выводим сгенерированное число на экран

# Пример работы с библиотекой TIME
print("До сна", datetime.now()) # Пример работы с DATETIME
sleep(5) # Заставляет скрипт уснуть на время в секундах
print("После сна")
print("Текущее время", datetime.now())
```

В первых строчках мы импортируем нужные нам функции из библиотек. Сначала указывается, откуда импортируем, затем — какую функцию. Если необходимо импортировать несколько функций, их можно перечислить через запятую.

Затем приведён пример работы генератора случайных чисел с помощью функции randint() из библиотеки random. Эта функция принимает два аргумента (начало диапазона и конец) и в рамках переданного ей диапазона возвращает случайное целое число. Мы будем использовать эту функцию для прототипирования функции get_btc_price().

В последней строчке примера datetime.now() можно увидеть новую конструкцию, которая называется объект. Мы обращаемся к *datetime* и через точку вызываем у него функцию now(), которая показывает текущее время. Для глубокого понимания работы объектов предназначен второй раздел книги.

Как уже говорилось, большая часть многомиллиардных компаний построена на труде энтузиастов, которые делятся своими исходниками с сообществом на безвозмездной основе. Библиотек написано бесчисленное множество, и с помощью них можно решить самые разные задачи — от работы с файловой системой до создания веб-приложений, нейросетей или программирования холодильников.

Задача 1.8

Реализуйте функцию get_btc_price() таким образом, чтобы она возвращала псевдокотировку Биткойна в диапазоне от 16000 до 100000.

Решение задачи 1.8:

```
def get_btc_price():
    from random import randint # Импортируем функцию RANDINT
    # Получает котировку биткойна по API
    # Не принимает аргументы
    # Отдаёт полученную цену
    result = randint(16000, 100000) # Генерируем псевдокотировку
    print("Сгенерирована псевдокотировка", result)
    return result # Отдаём котировку
```

1.22. Реализация функции create_deal()

Мы подобрались к самому сердцу нашей биржи — функции create_deal(). Функция принимает сумму сделки, направление, в котором предположительно пойдет цена актива, и время, через которое ордер должен исполниться.

На этапе прототипа функция должна сгенерировать и показать текущую псевдокотировку BTC, вычесть из общего баланса сумму сделки, уснуть на указанное пользователем время, затем сгенерировать новую котировку и рассчитать результат. Если игрок отгадал, мы обязаны умножить сумму ставки на значение, хранящееся в переменной `payout`, которая находится в глобальной области видимости, и начислить выигрыш на баланс. Если игрок проиграл, сумма проигрыша уже будет вычтена с баланса, и мы просто уведомляем его о неудаче. Если цена Биткойна не изменится, вернем ставку.

Условимся при приёме ставки, что баланс пользователя не находится на нуле и сумма сделки не превышает баланс. Направление сделки должно быть отражено цифрами: 0 — цена пойдёт вниз, 1 — цена пойдёт вверх.

Для решения задачи у вас уже должна быть готова функция get_btc_price(), которая генерирует псевдокотировку. В конце 3 раздела вы научитесь получать реальные данные о цене Биткойна через API биржи Bybit.

Реализуйте функцию create_deal() в своей копии исходника, как описано в листинге 1.2.

Листинг 1.2

```
def create_deal(sum, time, direction): #  direction = 0 вниз, 1 вверх
    # Создаёт сделку
    # Принимает 3 аргумента: сумму сделки, время экспирации и
    # направление сделки
    # Ничего не отдаёт
    global balance # Импортируем баланс из глобальной области
    print("<<<<<<---СДЕЛКА НАЧАТА--->>>>>>")
    print("Сумма", sum)
    print("Время", time)
    print("Направление", direction)
    print("Баланс до сделки", balance)

    # Если сумма сделки больше баланса или меньше 0
    if sum > balance or sum <= 0:
```

```
        print("Ты не можешь поставить больше чем есть или меньше 0")
        return # Останавливаем работу функции аварийно

    # Если сумма сделки верна продолжаем работу
    print("Твоя сделка принята")
    balance = balance - sum # Вычитаем сумму ставки из баланса
    win_sum = win_calculate(sum) # Расчитываем потенциальный выигрыш

    first_rand = get_btc_price() # Котировка в начале сделки
    print("Стартовое значение котировки", first_rand)

    sleep(time) # Засыпаем на секунды переданные в аргументе time

    second_rand = get_btc_price() # Котировка после ожидания
    print("Финальное значение котировки", second_rand)

    if direction == 0 and second_rand < first_rand:
        # Ставил на падение и котировка упала
        print("Ты выиграл", win_sum)
        balance = balance + win_sum # Начисляем выигрыш на баланс
    elif direction == 0 and second_rand > first_rand:
        # Ставил на падение и котировка выросла
        print("Ты проиграл")
    elif direction == 1 and second_rand > first_rand:
        # Ставил повышение и котировка выросла
        print("Ты выиграл", win_sum)
        balance = balance + win_sum # Начисляем выигрыш на баланс
    elif direction == 1 and second_rand < first_rand:
        # Ставил повышение и котировка упала
        print("Ты проиграл")
    else: # Если котировки одинаковые
        print("Возврат ставки")
        balance = balance + sum # Начисляем возврат на баланс
    print(">>>>>>---СДЕЛКА ЗАВЕРШЕНА---<<<<<<")
    return # Завершаем функцию

create_deal(sum=100, time=10, direction=0) # Вызываем функцию на тест
create_deal(sum=100000, time=10, direction=1) # Провальный тест
create_deal(sum=0, time=5, direction=1) # Провальный тест
```

Задача 1.9

Перепишите функцию menu() таким образом, чтобы после отработки соответствующих пунктов функция menu() сама себя перезапускала.

Решение задачи 1.9:

Решение этой задачи представлено в финальной версии проекта.

1.23. Списки (массивы)

Давайте представим ситуацию, в которой мы хотим хранить суммы всех сделок, которые совершил пользователь во время работы с программой.

Плохим решением было бы заранее определить несколько десятков переменных и записывать каждую новую сделку в одну из них. Во-первых, неизвестно, сколько сделок совершит игрок за одну сессию, во-вторых, это создаст огромное количество избыточного, нагроможденного кода, который невозможно будет поддерживать. Подобного рода проекты называют "Индусским кодом" или "Лапшой".

На помощь приходит новая конструкция, именуемая **списком**. В большинстве языков программирования эта конструкция называется **массивом**. На мой взгляд, не было ни одной причины давать этой конструкции новое название, но решать создателю языка.

Список — это изменяемая, упорядоченная коллекция данных, которая является самостоятельным типом данных (list). Списки могут содержать данные любых типов, но в основном хранятся данные одного типа.

Чтобы объявить список, нужно создать переменную и внутри квадратных скобок перечислить элементы через запятую.

Каждый элемент списка имеет порядковый номер, именуемый **индексом**. Индексация начинается с 0.

Создайте новый проект в Google Colab, назовите его LISTSTest.py и перепишите следующие строки:

```
coins = ["BTC", "LTC", "XMR", "GHOST"] # Объявляем список с монетами
print(coins) # Печатаем весь список на экран
# Выведет ["BTC", "LTC", "XMR", "GHOST"]
```

Здесь всё предельно просто, мы объявили список из четырех элементов и сохранили его в переменной *coins*.

1.24. Доступ к элементам списка

Для получения доступа к элементу списка необходимо в конце переменной, в которой список хранится, внутри квадратных скобок указать индекс элемента. Напомню, что индексация начинается с 0. Допишите в конец исходника LISTSTest.py следующие строки:

```
print("Доступ к первому элементу в списке", coins[0]) # Выведет BTC
print("Доступ ко второму элементу в списке", coins[1]) # Выведет LTC
print("Доступ к третьему элементу в списке", coins[2]) # Выведет XMR
print("Доступ к четвертому элементу в списке", coins[3]) # Выведет GHOST
```

В нашем примере перечисление элементов по индексам будет звучать следующим образом: "По индексу 0 хранится значение BTC", "По индексу 1 хранится значение LTC", "По индексу 2 хранится значение XMR", "По индексу 3 хранится значение GHOST".

1.25. Функции списков

Для работы с коллекциями, такими как список, есть вспомогательные функции. Например, функция len() измерит количество элементов в списке. Допишите в исходник LISTSTest.py следующую строку:

```
print("Кол-во элементов", len(coins)) # Измеряем количество элементов.
Выведет 4
```

Данная строка выведет, сколько элементов хранится в списке. В функцию len() в качестве аргумента была передана переменная, в которой хранится список.

Если ваш список состоит исключительно из чисел, то вы можете использовать следующие функции:

- max() — для поиска максимального значения в списке.
- min() — для поиска минимального значения в списке.
- sum() — для получения суммы всех элементов.

> Функции max(), min() и sum() работают аналогично функции len() и принимают на вход список.

Существует ряд функций, которые предназначены исключительно для работы со списками. С помощью функций списков мы можем добавить новый элемент в список, удалить один из элементов, заместить элемент и так далее. Эти функции вызываются через точку напротив переменной.

Функции списка, вызывающиеся через точку, ещё называют **методами**. С этим термином мы поближе познакомимся во второй главе.

Допишите в файл LISTSTest.py следующие строки:

```
coins.append("AAVE")  # Добавляет элемент в конец списка
print("Новое содержимое списка", coins)

# coins.insert() устанавливает новый элемент по индексу со смещением
coins.insert(3, "PARTCL")
print("Добавили элемент и сместили предыдущий", coins)
coins.remove("BTC")  # Удаляет первый найденный по значению элемент
print("Содержимое списка после удаления элемента", coins)

# coins.pop(0) удаляет элемент по индексу и возвращает его
deleted_element = coins.pop(0)
print("Удалённый элемент", deleted_element)  # Удалённый элемент списка
print("Список после удаления", coins)

print(coins.count("GHOST"))  # Считает кол-во элементов по значению
coins.reverse()  # Развернуть список - изменяет исходный список
print("Перевёрнутый список", coins)
new_coins = coins.copy()  # Поверхностная копия списка
new_coins.clear()  # Очистка списка
print("Размер скопированного списка после очистки", len(new_coins))
```

По большей части комментарии хорошо описывают работу каждой функции, и дополнительных объяснений не требуется. Остановимся подробнее лишь на последних строках.

Поверхностное копирование, описанное в строке `new_coins = coins.copy()`, предполагает, что новый список не будет ссылаться на старый. Т.е. если исходный список будет изменён, то скопированный список нет. Для так называемого глубокого копирования существуют библиотеки, которые вы без проблем найдете самостоятельно.

Задача 1.10

Перепишите глобальные переменные all_deals, win_deals, lose_deals, orders_archive таким образом, чтобы вместо значения по умолчанию *None* они хранили пустые списки.

Решение задачи 1.10:

```
orders = [] # Список для хранения сумм ордеров
win_orders = [] # Выигрышные ордера
lose_orders = [] # Проигрышные ордера
orders_archive = [] # Список для подробностей обо всех сделках
```

Задача 1.11

В функции create_deal() при подтверждении ордера (до исполнения) добавляйте сумму сделки в переменную *orders*. Если пользователь выиграл, добавляйте сумму выигрыша в список с выигрышами. Если пользователь проиграл, добавьте сумму проигрыша в соответствующий список.

Решение задачи 1.11:

Представлено в финальном листинге.

Задача 1.12

Реализуйте функцию show_win_deals() таким образом, чтобы она показывала общее количество прибыльных сделок, максимальный выигрыш, минимальный выигрыш и сумму всех выигрышных сделок.

Решение задачи 1.12:

Представлено в финальном листинге.

Задача 1.13

Реализуйте функцию show_lose_deals() таким образом, чтобы она показывала общее количество убыточных сделок, максимальный убыток, минимальный убыток и сумму всех убыточных сделок.

Решение задачи 1.13:

Представлено в финальном листинге.

Задача 1.14

Реализуйте функцию show_statistic () таким образом, чтобы она показывала оборот всех сделок, высчитывала максимальную ставку, минимальную ставку.

Решение задачи 1.14:

Представлено в финальном листинге.

1.26. Срезы

Срезы позволяют организовать гибкую выборку данных списка.

Ниже представлен демонстрационный пример.

```
coins = ["BTC", "LTC", "XMR", "GHOST"] # Объявляем список с монетами
print(coins[2:4]) # Выведет со 2-го по 4-й элемент НЕ включительно
>>> ["XMR", "GHOST"]

2 – Начало диапазона
4 – Окончание диапазона
```

Обратитесь к сети, чтобы более глубоко погрузиться в концепцию срезов.

1.27. Циклы

Когда речь касается повторного запуска некоторого куска кода, помимо рекурсии используют циклы. Существует два основных цикла: FOR и WHILE.

- Цикл FOR — используется, когда заранее известно, сколько раз необходимо повторить выполнение кода, перечисленного в его теле. Преимущественно применяется для прохода по всем элементам списка.
- Цикл WHILE — повторяет выполнение кода, перечисленного в его теле, до тех пор, пока условия для выполнения возвращают истину (True). Этот цикл небезопасен, порождает утечки в оперативной памяти и не рекомендуется к применению. Лучше НЕ использовать его в своей практике.

То, что обычно непрофессионалы возлагают на цикл WHILE, мы научимся делать с помощью рекурсии и утилиты CRON. По этой причине его изучение выходит за рамки данного руководства. Вы легко разберетёсь с ним сами, если хоть немного умеете использовать поисковики.

Создайте новый файл, назовите его CYCLETest.py и перенесите список из предыдущего раздела:

```
# Список из строковых элементов
coins = ["BTC", "LTC", "XMR", "GHOST"]

for coin in coins: # Запускаем цикл по словарю coins
    print("Элемент списка:", coin) # 1 проход — это 1 итерация
```

Запустив код, вы увидите следующее:

```
>>> "BTC"
>>> "LTC"
>>> "XMR"
>>> "GHOST"
```

При запуске перебора списка создаётся промежуточная переменная *coin*, в которую последовательно попадёт каждый элемент из списка *coins*. Промежуточная переменная будет изменяться до тех пор, пока цикл FOR: не пройдёт по всем элементам списка.

Имя переменной списка лучше указывать во множественном числе, а промежуточная переменная в основном описывается в единственном.

Добавьте следующие строки для примера работы со списком, состоящим из чисел:

```
orders = [45, 67, 89] # Список из числовых элементов

for order in orders: # Запускаем цикл по списку orders
   result = order ** 2
   print("Элемент списка", order, "во второй степени:", result)
```

Запустив код, вы увидите следующее:

```
>>> Элемент списка 45 во второй степени: 2025
>>> Элемент списка 67 во второй степени: 4489
>>> Элемент списка 89 во второй степени: 7921
```

В этом примере мы создали список `orders`, состоящий из целых чисел, запустили по нему цикл и поочередно возвели в степень каждый элемент списка. Наконец, вывели результат на экран. Как уже говорилось, в большинстве случаев заранее известно количество раз, которое циклу предстоит отработать. Перебор элементов будет продолжаться до тех пор, пока список не закончится. Один проход по всему содержимому тела цикла называется **итерацией**.

Задача 1.15

В функцию show_loses_deals() с помощью цикла FOR: добавьте перебор списка сумм всех убыточных сделок. Поочередно выведите на экран его содержимое.

Решение задачи 1.15:

Представлено в финальном листинге.

Задача 1.16

В функцию show_win_deals() с помощью цикла FOR: добавьте перебор списка сумм всех прибыльных сделок. Поочередно выведите на экран его содержимое.

Решение задачи 1.16:

Представлено в финальном листинге.

1.28. Словари (ассоциативные массивы, или хеши)

Поздравляю. Вы добрались до крайней конструкции, которая необходима в освоении базового программирования.

Словари — это неупорядоченная коллекция данных, где доступ к значению осуществляется по ключам (минуя индексы). Словари также называют "хешами" или "ассоциативными массивами". Словарь — это самостоятельный тип данных наряду со строками, числами и списками.

Словари используются для объединения разрозненных данных и помогают более изящно и лаконично обращаться к ним. Например, в словарь можно собрать различные показатели по штатам, регионам или странам и т.д.

Каждый элемент в словаре соответствует парной схеме "ключ: значение". Для получения доступа к значению используется соответствующий ключ. Для создания словаря пары "ключ: значение" необходимо перечислить между двух фигурных скобок. Каждая новая пара должна быть отделена друг от друга запятой.

Пример на псевдокоде:

```
имя_переменной_для_словаря = {"ключ": значение, "ключ": значение,
"ключ": значение}
```

Создайте новый файл, назовите DICTIONARESTest.py и перепишите следующий код:

```
ghost_info = { # Создаём словарь
    "tiker_name": "GHOST",
    "blockchain_type": "ANON",
    "capitalization": 1000000,
    "cold_staking": True,
    "year_profit": 20,
    "domain": "ghostprivacy.net",
    "team": ["Tizymandias", "Tuxprint", "Secret Recipe", "Netrunner"],
}

print("Название монеты:", ghost_info["tiker_name"])
print("Тип блокчейна:", ghost_info["blockchain_type"])
print("Капитализация:", ghost_info["capitalization"])
print("Холодный стейкинг:", ghost_info["cold_staking"])
print("Годовая доходность в процентах:", ghost_info["year_profit"])
print("Домен:", ghost_info["domain"])
```

Запустив исходник, вы увидите следующий вывод:

```
>>> Название монеты: GHOST
>>> Тип блокчейна: ANON
>>> Капитализация: 1000000
>>> Холодный стейкинг: True
>>> Годовая доходность в процентах: 20
>>> Домен: ghostprivacy.net
```

Мы объявили словарь **ghost_info**, который хранит информацию о анонимной монете GHOST. Затем в конце переменной в квадратных скобках мы перечислили ключи, по которым хранятся соответствующие им значения.

Обратите внимание, что после объявления словаря значения в нем могут храниться в произвольном порядке.

Ниже представлен пример того, как можно перебрать список, хранящийся в словаре.

Добавьте в конец тестового проекта следующий код:

```
for worker in ghost_info["team"]:
   print("Участник команды:", worker)
```

Запустив исходник, вы увидите следующий вывод на экране:

```
>>> Участник команды: Tizymandias
>>> Участник команды: Tuxprint
>>> Участник команды: Secret Recipe
>>> Участник команды: Netrunner
```

По ключу ghost_info["team"] мы получили доступ к списку, в котором хранятся имена участников команды. В промежуточную переменную *worker* поочередно попадёт каждое имя и выведется на экран.

Для того чтобы изменить одно из значений словаря, достаточно обратиться к нему по ключу и с помощью оператора "равно" (=) присвоить новое значение.

```
ghost_info["year_profit"] = 30
print("Годовая доходность в процентах после изменения:",
ghost_info["year_profit"])
ghost_info["team"].append("Morning Ghost")
# Добавляем ещё одного участника в команду
```

Выведет на экран:

```
>>> Годовая доходность в процентах после изменения: 30
```

1.29. Функции словарей

У словарей, как и у списков, есть собственные функции, которые вызываются через точку.

Ниже приведён список некоторых из них. Обязательно добавьте их в свой исходник, предназначенный для тестирования словарей.

- ghost_info.clear() — очищает словарь.
- ghost_info.copy() — возвращает копию словаря.
- ghost_info.items() — возвращает пары (ключ, значение).
- ghost_info.keys() — возвращает ключи в словаре.
- ghost_info.values() — возвращает значения в словаре.

Задача 1.17

Создайте список для хранения архива сделок с подробной информацией. Во время сделки сформируйте словарь, в котором соберётся подробная информация на текущий ордер.

Информация должна собираться следующая: время начала, время окончания, направление сделки (вверх или вниз), стартовое значение котировки, конечное значение котировки, статус (выиграла или нет). После того как сделка исполнилась, отправьте сформированный словарь в список с архивом сделок.

Решение задачи 1.17:

Представлено в финальном листинге.

Подсказка: в функции create_order() заранее сформируйте переменную со словарем, где перечислите ключи и пустые значения. Если предполагается хранение текстового значения после ключа, укажите две двойные кавычки без содержимого "". Если планируется числовое значение, просто укажите 0. Если планируется логическое значение, укажите False. Затем по мере исполнения ордера замещайте значения по умолчанию теми, что актуальны на момент исполнения кода.

Задача 1.18

В функции show_statistic() переберите архив сделок, в котором хранится список словарей, и красиво выведите каждое значение по ключу. Подробное решение данной задачи представлено в финальном листинге этой главы.

Решение задачи 1.18:

Представлено в финальном листинге.

1.30. Знакомство с JSON

Для того чтобы закончить биржу, вам необходимо познакомиться с текстовым форматом передачи данных, именуемым JSON. По сути, это просто текст, который по своей структуре аналогичен словарям. Этот формат используется преимущественно для передачи данных между серверами и браузером.

В вашей практике, скорее всего, понадобится как возможность получения данных от серверов в этом формате, так и возможность сохранения JSON-файлов для последующей работы с ними.

В Пайтон уже встроена специальная библиотека для работы с JSON. В этой библиотеке хранятся функции как для преобразования строк и словарей в JSON, так и для обратного преобразования из JSON в словарь.

Создайте новый файл, назовите его JSONTest.py и перепишите следующий код.

```
# Импортируем две функции из библиотеки json
from json import dumps, loads

btc_price_info = {"price": 110000, "volume": 234566}
btc_price_info_to_json = dumps(btc_price_info) # Словарь в JSON
```

Теперь можно беспрепятственно передать содержимое переменной btc_price_info_to_json на сервер или записать в JSON-файл. Без преобразования интерпретатор выдаст ошибку.

Для обратного преобразования из JSON-формата в словарь используется функция loads():

```
btc_price_from_json_to_dict = loads(btc_price_info_to_json)
```

Теперь снова можно обращаться к ключам словаря без ошибок интерпретатора. Рекомендую прошерстить Интернет для более глубокого понимания этого формата.

1.31. Библиотека REQUESTS. Получаем котировку BTC через API

Для получения реальных данных о котировке BTC нам понадобится публичный API, к которому можно сделать HTTP-запрос. Для быстрого старта в качестве публичного API можно обратиться к сайту *https://coinlore.net*.

Прежде чем получить доступ к котировке программно с помощью Пайтон, наберите в браузере следующий URL:

https://api.coinlore.net/api/ticker/?id=90

В ответе в браузере вы увидите примерно следующее:

```
[{"id":"90","symbol":"BTC",
  "name":"Bitcoin",
  "nameid":"bitcoin",
  "rank":1,"price_usd":"99172.24",
  "percent_change_24h":"-1.51",
  "percent_change_1h":"0.01",
  "percent_change_7d":"3.01",
  "price_btc":"1.00",
  "market_cap_usd":"1959726172145.90",
  "volume24":86006814745.11156,
  "volume24a":99684040793.47655,
  "csupply":"19760834.00",
  "tsupply":"19760834",
"msupply":"21000000"}]
```

Поскольку ответ находится в квадратных скобках, сервер вернул список. В списке находится всего один элемент, который, в свою очередь, очень похож на словарь. Всё это и есть ответ в формате JSON.

Вероятно, отдавать список, в котором хранится всего один словарь, НЕ лучшее решение с точки зрения дизайна API. Можно было сразу вернуть словарь, минуя список. Зато к этому API можно обратиться совершенно бесплатно без предварительного получения ключей доступа.

Для обеспечения работы с HTTP-запросами существует специальная библиотека, и называется она REQUESTS.

Объяснение того, что такое HTTP-запрос и как работает протокол TCP/IP, выходит за рамки данного руководства. Пожалуйста, уделите немного времени самостоятельному ознакомлению с этими терминами.

Если вкратце, существует два основных типа запросов:

1. **GET**-запрос — когда вы просто запрашиваете информацию у сервера.
2. **POST**-запрос — когда вы хотите отправить на сервер какие-то данные.

Каждый раз, когда вы заходите на какой-то сайт из вашего браузера, то на сервер, на котором сайт расположен, отправляется HTTP-запрос, специально подготовленный для этих целей. В ответ сервер либо отдаёт данные, к которым вы хотели получить доступ, либо отвечает ошибкой.

Например, если вы зайдете на сайт WWW.ASTARTA-EDU.COM, то выполнится GET-запрос, а если вы отправите регистрационные данные в форме, то выполнится POST-запрос.

Публичные API, в свою очередь, позволяют обращаться к серверам без использования браузеров. Например, из консольных программ, написанных вашими собственными руками.

Большая часть современных сервисов предоставляют публичные API, в том числе для тех целей, которые мы преследуем в этой книге.

Зайдите в ПУСК, наберите КОМАНДНАЯ СТРОКА и введите:

```
pip install requests
```

Эта команда установит библиотеку REQUESTS с помощью менеджера пакетов **pip**. Создайте новый файл, назовите его REQUESTSTest.py и перепишите следующий код:

```
# Импортируем функцию get() из библиотеки requests
from requests import get

# Делаем HTTP-запрос к API
get_btc_data = get("https://api.coinlore.net/api/ticker/?id=90")

# Извлекаем из ответа данные в формате JSON
btc_data_in_json = get_btc_data.json()
```

```
btc_price = btc_data[0]["price_usd"] # Получаем доступ к котировке BTC
print("Текущая цена BTC", btc_price) # Выводим на экран только котировку
```

Код хорошо прокомментирован, и дополнительного разъяснения требует, пожалуй, только строка `btc_price = btc_data[0]["price_usd"]`.

Поскольку данное API возвращает список, по индексу 0 мы получаем доступ к первому его элементу. Этот элемент является словарем. По ключу price_usd мы получаем текущее значение цены BTC.

Задача 1.19

Перенесите код из предыдущего примера в функцию get_btc_price() таким образом, чтобы функция отдавала в качестве завершения своей работы реальные данные о котировке.

Решение задачи 1.19:

Представлено в финальном листинге.

1.32. Финальный листинг биржи бинарных опционов

Поздравляю, дамы и господа! Если вы решили все задачи самостоятельно, то практически в совершенстве овладели функциональным программированием. Ваш навык уже на достаточном уровне, чтобы создавать полноценные и очень мощные программы.

Если что-то не получилось, не переживайте! Ниже представлен полный листинг биржи опционов на Биткойн. Перепишите его или адаптируйте под текущее состояние своей версии проекта.

В третьей главе книги мы частично адаптируем его для Телеграма.

Листинг 1.3

```
from random import randint # Импортируем функцию RANDINT
from time import sleep  # Из библиотеки TIME импортируем функцию SLEEP
from requests import get # Для GET запросов
from datetime import datetime # Для работы со временем

# ГЛОБАЛЬНАЯ ОБЛАСТЬ ВИДИМОСТИ
program_name = "Опционы на Биткойн" # Тип данных строка
balance = 20000 # Тип данных число
payout = 1.80 # Тип данных число с плавающей точкой
verefied = False # Логический тип. Истина True. Ложь False
full_name = None # Пустая переменная без значения

orders = []  # Список для хранения сум ордеров. Решение задачи 1.10
win_orders = [] # Выигрышные ордера. Решение задачи 1.10
lose_orders = [] # Проигрышные ордера. Решение задачи 1.10
orders_archive = [] # Список для архива всех сделок. Решение задачи 1.10

def audit_balance(): # Решение задачи 1.5
    # Функция анализирует баланс пользователя и присваивает статус
    # Не принимает аргументов
    # Не отдаёт результат
    global balance # Импортируем переменную из глобальной области видимости

    if balance == 0:
        # Проверка на истину или ложь. Если проверка верна (True)
        print("Ты банкрот") # Выполнится этот кусок кода
    elif balance <= 1000: # Если balance меньше или равнен 1000
        print("Ты бронза")
    elif balance > 1000 and balance < 5000:
        # Если balance больше 1000 и меньше 5000
        print("Ты серебро")
    elif balance > 5000 and balance < 15000:
        # Если balance больше 5000 и меньше 15000
        print("Ты золото")
    else: # Если предыдущие проверки вернули False
        print("Ты платина, твой баланс", balance)
    return # Завершаем функцию

def show_info():
    # ЛОКАЛЬНАЯ ОБЛАСТЬ ВИДИМОСТИ
    # Функция отображает данные на программу
    # Не принимает аргументов. Не отдаёт результат
    global program_name, balance, payout, verefied, full_name
    print("Программа называется:", program_name)
    print("Баланс:", balance)
```

```
    print("Сумма выплаты:", payout)
    print("Подтверждение аккаунта:", verefied)
    print("Имя пользователя:", full_name)

    if full_name:  # Проверяет существут значение или нет
        print("Твоё имя заполнено", full_name)
    else:
        print("Заполните имя")

    if verefied == True:  # Аналогично предыдущей проверке
        # == True можно опустить
        print("Ваш аккаунт верифицирован")
    else:
        print("Ваш аккаунт неверифицирован")
    audit_balance() # Вызываем аудит баланса
    return # Завершаем функцию

def set_user_name(first_name, last_name):
    # Функция устанавливает имя пользователя
    # Принимаем имя пользователя в качестве аргумента
    # Возвращает имя
    global full_name # Обращаемся к глобальной переменной
    print("Текущее имя:", full_name)
    full_name = first_name + " " + last_name  # Конкатенация
    print("Полное имя пользователя:", full_name)
    return full_name # Отдаём имя

def add_sum_to_balance(sum):
    # Добавляет денег на баланс
    # Принимает 1 аргумент сумму добавления
    # Возвращает текущий баланс
    global balance # Импортируем глобальную переменную
    print("Пользователь хочет добавить", sum, "на баланс")
    print("Баланс до начисления", balance)

    if int(sum) <= 0: # Решение задачи 1.6.
        # Если сумма ставки меньше или равно 0
        print("Нельзя добавить 0 долларов и меньше")
        return # Останавливаем функцию аварийно

    # Если функция не была остановлена аварийно
    balance = balance + sum # Начисляем на баланс
    print("Баланс после начисления", balance)
    audit_balance() # Вызываем функцию аудита баланса
    return balance # Завершаем функцию и отдаём баланс
```

```
def get_btc_price():
    # Получает котировку биткойна по API
    # Не принимает аргументы
    # Отдаёт полученную цену
    # Решение задачи 1.19
    btc_data = get("https://api.coinlore.net/api/ticker/?id=90")
    btc_data = btc_data.json() # Получили ответ в формате JSON
    print("Текущая цена биткойна", btc_data[0]["price_usd"])
    return float(btc_data[0]["price_usd"]) # Возвращаем цену

def win_calculate(sum):
    # Функция рассчитывает потенциальную сумму выигрыша
    # Принимает 1 аргумент сумму ставки
    # Возвращает результат выигрыша
    global payout # Импортируем переменную из глобальной области
    result = int(sum) * payout # Умножаем сумму сделки на коэффициент
    print("Ты собираешься поставить", sum)
    print("Потенциальный выигрыш", result)
    return result # Отдаём результат работы функции

def create_deal(sum, time, direction):
    # Создаёт сделку
    # Принимает 3 аргумента: сумму сделки, время экспирации и
    # направление сделки
    # Ничего не отдаёт
    # Импортируем глобальные переменные
    global balance, orders, win_orders, lose_orders, orders_archive
    print("<<<<<<---СДЕЛКА НАЧАТА--->>>>>>")

    # Объявляем словарь для подробного хранения сделки
    order_full_info = { # Решение задачи 1.17
        "sum": "",
        "start_btc": "",
        "end_btc": "",
        "win_status": "",
        "win_sum": 0,
        "start_time": "",
        "end_time": ""
    }

    # Если сумма сделки больше баланса или меньше 0
    if sum > balance or sum <= 0:
        print("Ты не можешь поставить больше чем есть или меньше 0")
        return # Аварийно останавливаем работу функции

    # Если сумма сделки верна продолжаем работу
    print("Твоя сделка принята")
```

```
balance = balance - sum # Вычитаем сумму ставки из баланса
print("Текущий баланс", balance)
win_sum = win_calculate(sum) # Расчитываем потенциальный выигрыш
first_rand = get_btc_price() # Котировка в начале сделки
print("Стартовое значение котировки", first_rand)
order_full_info["start_time"] = str(datetime.now()) # Задача 1.17

sleep(time) # Засыпаем на секунды переданные в аргументе time

second_rand = get_btc_price() # Котировка после ожидания
print("Финальное значение котировки", second_rand)
order_full_info["end_time"] = str(datetime.now()) # Решение задачи 1.17
order_full_info["sum"] = sum # Решение задачи 1.17
order_full_info["start_btc"] = first_rand # Решение задачи 1.17
order_full_info["end_btc"] = second_rand # Решение задачи 1.17
orders.append(sum) # Добавляем сумму в общий список. Решение задачи 1.11

if direction == 0 and second_rand < first_rand:
    # Ставил на падение и цена упала
    print("Ты выиграл", win_sum)
    balance = balance + win_sum # Начисляем выигрыш на баланс
    win_orders.append(win_sum) # Добавляем сумму в выигрыша. Задача 1.11
    order_full_info["win_sum"] = win_sum # Решение задачи 1.17
    order_full_info["win_status"] = True # Решение задачи 1.17
elif direction == 0 and second_rand > first_rand:
    # Ставил на падение и цена выросла
    print("Ты проиграл")
    lose_orders.append(sum) # Добавляем сумму в проигрыши. Задача 1.11
    order_full_info["win_sum"] = 0 # Решение задачи 1.17
    order_full_info["win_status"] = False # Решение задачи 1.17
elif direction == 1 and second_rand > first_rand:
    # Ставил повышение и котировка выросла
    print("Ты выиграл", win_sum)
    balance = balance + win_sum # Начисляем выигрыш на баланс
    win_orders.append(win_sum) # Добавляем сумму в выигрышы. Задача 1.11
    order_full_info["win_sum"] = win_sum # Решение задачи 1.17
    order_full_info["win_status"] = True # Решение задачи 1.17
elif direction == 1 and second_rand < first_rand:
    # Ставил повышение и котировка упала
    print("Ты проиграл")
    lose_orders.append(sum) # Добавляем сумму в проигрыши. Задача 1.11
    order_full_info["win_sum"] = 0 # Решение задачи 1.17
    order_full_info["win_status"] = False # Решение задачи 1.17
else:
    print("Возврат ставки")
    balance = balance + sum # Возвращаем ставку на баланс
```

```
    orders_archive.append(order_full_info) # Решение задачи 1.17
    print(">>>>>>---СДЕЛКА ЗАВЕРШЕНА---<<<<<<")
    menu() # Запускаем меню после завершения сделки
    return # Завершаем функцию create_deal()

def menu():
    # Функция отображает меню для управления биржей
    # Функция отображает данные на программу
    # Не принимает аргументов
    # Не отдаёт результат
    show_info() # Показываем информацию на программу

    print("Привет. Выбери один из пунктов")
    print("1 если хочешь добавить денег на баланс")
    print("2 если хочешь сделать сделку")
    print("3 если хочешь посмотреть статистику")
    print("4 если хочешь установить имя")
    print("5 если хочешь уйти")

    user_choise = input("Введи пункт: ")
    user_choise = int(user_choise)
    print()

    if user_choise == 1: # Если игрок выбрал пополнение баланса
        user_sum = int(input("Введи сумму пополнения: "))
        add_sum_to_balance(user_sum)
        menu() # Рекурсивный вызов menu(). Решение задачи 1.9
        return # Завершаем текущее выполнение menu()
    elif user_choise == 2: # Если пользователь выбрал сделку
        print("Текущая котировка", get_btc_price())
        get_user_sum = input("Вводи сумму которую хочешь поставить: ")
        get_user_sum = int(get_user_sum) # Приводим к числу
        get_user_time = int(input("Введи время сделки: ")) # К числу
        get_user_direction = int(input("Введи направление 0 или 1: "))
        create_deal(sum=get_user_sum,
            time=get_user_time, direction=get_user_direction)
        menu() # Рекурсивный вызов menu(). Решение задачи 1.9
        return # Завершаем текущее выполнение menu()
    elif user_choise == 3: # если пользователь выбрал посмотреть статистику
        show_statistic() # Показываем статистику
        menu() # Рекурсивный вызов menu(). Решение задачи 1.9
        return # Завершаем текущее выполнение menu()
    elif user_choise == 4: # если пользователь выбрал установить имя
        f_name = input("Введи имя: ")
        l_name = input("Введи фамилию: ")
        set_user_name(first_name=f_name, last_name=l_name)
        menu() # Рекурсивный вызов menu(). Решение задачи 1.9
```

```
        return # Завершаем текущее выполнение menu()
    elif user_choise == 5: # если пользователь решил уйти
        print("До свидания")
        exit() # Выходим из программы
    else:
        print("Ты выбрал не верный пункт. Перезапускаю меню")
        menu() # Рекурсивный вызов menu(). Решение задачи 1.9
        return # Завершаем текущее выполнение menu()

def show_statistic():
    # Показывает статистику выигрышных сделок
    # Не принимает значения
    # Ничего не отдаёт
    global orders, orders_archive # Импорт глобальных переменных

    if len(orders) != 0: # Если ордера есть
        print("Сумма твоих сделок", sum(orders)) # Решение задачи 1.14
        print("Максимальная ставка", max(orders)) # Решение задачи 1.14
        print("Минимальная ставка", min(orders)) # Решение задачи 1.14

    show_win_deals() # Показываем выигрышные сделки
    show_loses_deals() # Показываем проигрышные сделки

    print("!!!!!АРХИВ СДЕЛОК!!!!!")
    if len(orders_archive) != 0: # Если список с архивом НЕ пуст
        for order in orders_archive: # Решение задачи 1.18
            print("Сумма сделки", order["sum"]) # Решение задачи 1.18
            print("Начало сделки", order["start_time"]) # Решение задачи 1.18
            print("Конец сделки", order["end_time"]) # Решение задачи 1.18
            print("Цена BTC старт", order["start_btc"]) # Решение задачи 1.18
            print("Цена BTC конец", order["end_btc"]) # Решение задачи 1.18
            print("Сумма выигрыша", order["win_sum"]) # Решение задачи 1.18
            print("Статус сделки", order["win_status"]) # Решение задачи 1.18
    return # Завершаем функцию show_statistic()

def show_win_deals():
    # Показывает статистику выигрышных сделок
    # Не принимает значения
    # Ничего не отдаёт
    if len(win_orders) != 0: # Если список НЕ пуст
        print("Ты выиграл", sum(win_orders)) # Решение задачи 1.12
        print("Максимальный выигрыш", max(win_orders)) # Решение задачи 1.12
        print("Минимальный выигрыш", min(win_orders)) # Решение задачи 1.12
        print("Все выигрыши ордера")
        for win in win_orders: # Решение задачи 1.16
            print("Сумма выигрышной сделки", win)
    return # Завершаем работу функции
```

```
def show_loses_deals():
    # Показывает статистику выигрышных сделок
    # Не принимает значения
    # Ничего не отдаёт
    if len(lose_orders) != 0:
        print("Ты проиграл", sum(lose_orders)) # Решение задачи 1.13
        print("Максимальный проигрыш", max(lose_orders)) # Задачи 1.13
        print("Минимальный проигрыш", min(lose_orders)) # Решение задачи 1.13
        print("Все проигрыши")
        for lose in lose_orders: # Решение задачи 1.15
          print("Сумма проигрышной сделки", lose)
    return # Завершаем работу функции

menu() # Запускаем биржу
```

Поздравляю!

Вы в кратчайшие сроки научились эффективному функциональному программированию и при этом закончили свой первый проект. Это повод для гордости!

1.33. Утилита TickerGrub.py. Получаем исторические данные о свечах

Если программист НЕ умеет работать с файлами, значит он НЕ программист! Для бэктестов нам понадобятся исторические данные о свечах в формате CSV.

К сожалению, страница для скачивания данных с сайта биржи Байбит не работает, по этой причине мне пришлось написать утилиту TickerGrub.py, которая поможет скачать дневные свечи за последнюю тысячу дней.

Полный код утилиты представлен в листинге 1.4.

Прежде чем переписывать в своем терминале, наберите:

```
pip install pybit
```

Эта библиотека поможет нам обратиться к историческим данным биржи Bybit.

Листинг 1.4

```
from pybit.unified_trading import HTTP # Объект для работы с Bybit
from pprint import pprint # Для красивого отображения словарей
from datetime import datetime # Объект для работы с временем

intro = """
<<<---TickerGrub--->>>
Привет, Человек!
Я соберу исторические данные с Bybit ;)
Затем упакую в CSV файл :P
Формат свечей: D1 (1000 свечей)
Формат тикеров:
BTCUSDT, LTCUSDT, ETHUSDT
>>>---TickerGrub---<<<
"""

pprint(intro) # Печатаем интро
ticker_name = input("Введите имя тикера: ")
interval = "D" # Тип свечей Дневки

# Создаём объект класса HTTP из библиотек Pybit
session = HTTP(testnet = False)

def start_grub():
    # Функция грабит данные с Bybit
    global ticker_name, interval # Работаем с глобальными переменными

    # Обращаемся за историческими данными в Bybit
    get_candles = session.get_kline(
        category="linear",
        symbol=ticker_name,
        interval=interval,
        limit=1000) # Приходит словарь

    pprint(get_candles)

    # Из пришедшего словаря забираем список
    candles = get_candles["result"]["list"]
    candles.reverse() # Разворачиваем список

    # Формат данных пришедших от session.get_kline()
    # Двумерный список(список списоков)
```

```
# candles = [
#         ["Date", "Open", "High", "Low", "Close", "Volume"],
#         ["123234536457", 28.345, 45.3424, 45.3424, 4225.3424],
#         ["123234536457", 28.345, 45.3424, 45.3424, 4225.3424],
#         ["123234536457", 28.345, 45.3424, 45.3424, 4225.3424]
# ]
# Файлы с одинаковыми названиями будут перезаписаны

# Конкатенируем тикер и интервал для имени файла
csv_file_name = ticker_name + "_" + interval + ".csv"

# Первая строка файла
first_row = "datetime, open, high, low, close, volume"

# Открываем файл в безопасном режиме на запись
# Прочитайте как работает with
with open(csv_file_name, mode="w") as file:
    # Записываем первую строку в файл для подписи ячеек
    file.write(first_row + "\n")

    for cundle in candles: # Перебираем все свечи
        # В cundle лежит список:
        # ["1234567", 28.345, 45.3424, 45.3424]
        # Приводим время Timestemp к Unix Time,
        # а затем к человеческому виду
        # Прочитайтк что такое TIMESTAMP и UNIXTIME

        # Приводим UNIX TIME к Timestemp
        unix_time = int(cundle[0]) / 1000

        # Получаем человеческое время
        date_time = datetime.fromtimestamp(unix_time)

        # Не создаём новый список, заменяем UNIX время на ходу
        # У первого элемента свечи меняем дату. ОПАСНО!
        cundle[0] = date_time.strftime("%Y-%m-%d")

        print("Date", cundle[0])
        print("Open", cundle[1])
        print("High", cundle[2])
        print("Low", cundle[3])
        print("Close", cundle[4])
        print("Volume", cundle[5])
        print()
        # ПОСЛЕ ПРЕОБРАЗОВАНИЙ
        #  ["2023-10-05", 28.345, 45.3424, 45.3424]
        # Не забываем что cundle это список. Приводим к строке
        # При этом сохранить порядок и запятые
```

```
            # MAP делает из каждого элемента часть строки
            cundle_maped = map(str, cundle)

            # Добавляет разделить после каждого элемента
            cundle_join = ",".join(cundle_maped)
            file.write(cundle_join + "\n") # записываем строку в файл

    print("Добавлено", len(candles), "строк")
    print("CSV файл", csv_file_name, "успешно создан")
    return # Завершаем функцию

def menu(): # Функция для выбора тикера
    global intro, ticker_name, interval # Импорт глобальных переменных
    pprint(intro) # Печатаем интро
    print("Введите 1 если хотите начать новый поиск: ")
    print("Введите 2 если хотите eqnb: ")
    answer = input(">>>")

    if answer == "1":
        ticker_name = input("Введи имя тикера: ")
        start_grub()
    elif answer == "2":
        print("Пока")
        exit() # Выходим из программы
    else:
        print("Попробуй ещё раз")
        menu() # Рекурсивный перезапуск menu()
    return # Завершаем текущее выполнение menu()

start_grub() # Стартуем грабер
```

Утилита хорошо написана и прокомментирована, дополнительных разъяснений не требует.

В блок TRY помещается критически важный кусок кода; если происходит ошибка, то срабатывает блок EXCEPT. Тем самым мы убережём программу от падения.

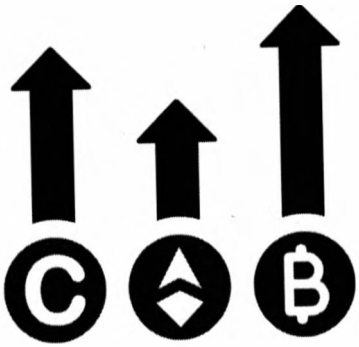
Для того чтобы хорошо понять, как работает строка

`session = HTTP(testnet = False)`, прочтите главу 2.

Задача 1.20

Изучите самостоятельно новые для вас конструкции WITH/OPEN.

Изучите самостоятельно функции: open(), join(), map().



Глава 2.

Объектно-ориентированное программирование.

Создаём Биткойн и Альткойн

Содержание главы:

2.1. Как появилось ООП?

Объектно-ориентированное программирование (далее ООП) является надставкой над функциональным программированием и распространилось около 30 лет назад благодаря языку программирования C++ и платформе JAVA (не путать с JavaScript).

Многие разработчики совершают ошибку, начиная изучение программирования с ООП. По этой причине огромный процент действующих программистов НЕ понимают, что делают, пишут код по референсам из Интернета и НЕ способны повторить уже написанные программы.

Можно долго рассуждать, является ли ООП успешной или провальной концепцией.

Мое мнение: ООП избыточно, сложно и, если есть возможность НЕ создавать собственные классы, лучше НЕ создавать. Единственное, на мой взгляд, направление, в котором ООП действительно удобно, — это разработка игр.

Некоторые современные языки программирования, например GO, разработанный компанией Google, не поддерживает ООП в принципе. Благодаря этому работают с молниеносной скоростью и быстро осваиваются.

В качестве доказательства избыточности ООП приведу простой пример. Главная книга по языку функционального программирования Си, которая пережила два десятка переизданий, содержит в себе буквально 100 страниц технического текста.

Настольная книга C++ (ООП-версия языка Си) насчитывает 1700 страниц. При этом на момент выпуска C++ можно было создать одинаково работающие программы как на Си, так и на C++.

Тем не менее, хотим мы этого или нет, ООП плотно проникло в жизнь разработчиков, и без понимания основных концепций быть эффективным НЕ получится. Конструкцию CLASS и объекты, созданные на базе класса, мы изучить обязаны.

Прежде чем приступить к написанию кода, необходимо разобрать основные принципы, на которых зиждется ООП. Если говорить техническим языком, то основных принципа три:

1. **Наследование** — позволяет наследовать уже написанные классы.
2. **Инкапсуляция** — предполагает сокрытие и изоляцию объектов.
3. **Полиморфизм** — предполагает множество видов объектов.

Я не стану грузить вас техническими подробностями этих понятий, в Интернете достаточно информации на этот счёт. Сразу перейдем к более частным деталям и пофилософствуем. Не переживайте, если какие-то концепции ООП будут не ясны сразу. ООП требует времени на осмысление, практики и некоторого упорства в освоении.

2.2. Что есть объект?

Если вникнуть, то всё окружающее человека (в том числе и сам человек) является объектом. При этом каждый объект обладает какими-то свойствами, которые хранят его состояние, и действиями, которые объект может совершать.

При этом объекты одного типа обладают одинаковыми свойствами и действиями, но изолированы друг от друга. 1000 стиральных машин одной модели совершенно одинаковы по своей сути, но являются независимыми друг от друга объектами.

Давайте опишем свойства и действия практически любого автомобиля.

Свойства:

Производитель
Марка
Цвет
Количество дверей
Вес
Максимальная скорость

Действия:

Ехать вперёд
Ехать назад
Остановиться
Включить фары
Выключить фары
Открыть двери
Закрыть двери
Включить дворники
Выключить дворники

Конечно, можно перечислить значительно больше свойств и действий. Например, в свойства можно добавить текущее состояние дворников или дверей, сколько было владельцев автомобиля, участвовал ли автомобиль в ДТП. Пофантазируйте самостоятельно на этот счёт.

Давайте опишем свойства и действия одушевленного объекта, например кошки.

Свойства:

Кличка
Порода
Год рождения
Стерилизация

Действия:

Спать
Есть
Мяукать
Кусаться
Веселиться
Скучать по хозяину

Напоследок для хорошего понимания примеров из разных сфер жизни опишем объект Биткойна, который впоследствии реализуем.

Свойства:

Адрес

Дата генерации

Сумма

Цена

Хеш-сумма последнего блока

Транзакции

Действия:

Отправить

Принять

Майнить

В целом если подумать, то сама по себе технология Биткойна примитивна и имеет не так много возможностей, кроме достаточно необходимых.

Подведем итог. Все объекты, которые мы будем создавать в будущем, обладают свойствами и действиями.

> В контексте объектно-ориентированного программирования **свойства** — это *обычные переменные*, которые хранят состояние объекта, а **действия** — это *обычные функции*, которые, в свою очередь, могут менять состояние объектов. Функции в контексте классов именуются **методами**.

2.3. Что такое класс?

Для программирования объектов нам понадобится изучить новую конструкцию, именуемую CLASS.

Класс — это проект будущих объектов. Именно в классах описываются свойства и поведение всех будущих экземпляров. На базе одного класса может быть создано сколько угодно объектов. Все объекты, созданные на базе класса, будут обладать одинаковыми свойствами и действиями, но при этом изолированы друг от друга.

Самый простой пример — это форма для выливания леденцов. Форма существует в единственном числе, и на базе этого слепка создаётся огромное количество леденцов.

Все леденцы являются производными от формы, совершено одинаковы, но изолированы друг от друга. Так и классы позволяют из описанных свойств и действий создавать огромное количество объектов с одинаковым поведением.

2.4. Создаём класс Bitcoin

Создадим демонстрационный класс Биткойн, опишем его основные свойства и по старинке опишем пустыми будущие действия. Как уже говорилось, свойства — это обычные переменные, к которым вы уже могли привыкнуть, а методы — обычные функции.

Обратите внимание, что при описании функций в контексте класса каждая функция должна обладать обязательным аргументом *self*, который заполняется Пайтоном автоматически и помогает интерпретатору изолировать будущие объекты друг от друга.

Создайте новый файл BITCOINoop.py и перепишите код из листинга 2.1.

Листинг 2.1

```
class Bitcoin():
    # Описываем свойства будущих объектов
    price = 50.000 # Стоимость биткойна
    sum = 0.5 # Количество
    wallet = "" # Адрес кошелька
    transactions = [] # Транзакции

    # Описываем действия будущих объектов
    def show_info(self): # В self хранится ссылка на текущий объект
        # Аргумент self изолирует объекты друг от друга
        # Благодаря self интерпретатор понимает, в отношении какого
        # объекта вызвано действие
        pass

    def add_wallet(self, address):
        # Изменяет свойство wallet
        # Принимает один аргумент
        # Ничего не отдаёт
        pass

    def add_sum(self, new_sum):
        # Добавляет деньги на кошелёк
        # Принимает один аргумент
        # Ничего не отдаёт
        pass

    def minus_sum(self, sum):
        # Снимает деньги с кошелька
        # Принимает один аргумент
        # Ничего не отдаёт
        pass

    def create_transaction(self, sum, opponent):
        # Создаёт транзакцию
        # Аргумент opponent принимает объект Биткойна
        # Ничего не отдаёт
        pass

my_btc = Bitcoin() # Создаём объект Биткойн
mom_btc = Bitcoin() # Создаём второй объект Биткойн
dad_btc = Bitcoin() # Создаём третий объект Биткойн

my_btc.price = 60.0000 # Изменяем свойство одного из объектов
print("Изменяем свойство одного из объектов", my_btc s.price)
print("Цена объекта по умолчанию", mom_btc s.price)
```

Мы описали класс и создали 3 экземпляра.

Каждый экземпляр является самостоятельным объектом. Все три объекта обладают общим поведением и изолированы друг от друга.

В первой строчке можно увидеть новое ключевое слово *class*, после которого идёт имя Bitcoin(). По соглашению имена классов начинаются с большой буквы. Так при создании объекта его можно отличить от функции.

Затем мы объявили несколько свойств: price, sum, wallet, transactions. В них хранятся значения по умолчанию для каждого будущего объекта.

> Напомню, **свойства** — это обычные переменные, только в контексте класса.

Далее мы объявили несколько пустых методов: show_info(), add_wallet(), add_sum(), minus_sum(), create_transaction(). Функции в контексте класса называются методами и требуют обязательного аргумента *self*.

Затем создали 3 объекта. Все объекты принадлежат одному типу данных Bitcoin, обладают одинаковыми свойствами и функциями и при этом не зависят друг от друга.

Обратите внимание, что, создавая новый класс, вы создаёте новый тип данных. Все переменные, которые вы создали ранее, имеют свой собственный тип данных, а значит принадлежат какому-то классу. Это легко проверить, если использовать функцию type().

Ниже приведёно несколько примеров:

```
print("Класс строки", type("BITCOIN PRICE"))
print("Класс числа", type(1337))
print("Класс числа с плавающей точкой", type(56.000))
print("Класс логики", type(True))
print("Класс списка", type([1,2,3,4]))
print("Класс словаря", type({"btc_price": 56.000}))

my_btc = Bitcoin() # Создаём объект Биткойн
print("Класс Биткойн", type(my_btc)
```

2.5. Что такое *self*?

> В обязательном аргументе **self** хранится ссылка на текущий объект, в отношении которого вызван метод или использовано свойство.

Технически вызов любой функции объекта должен выглядеть так:

```
my_btc.show_info(self=my_btc) # Передаём самого себя
```

Интерпретатор берет на себя автоматическое заполнение аргумента *self*. Благодаря *self* объекты изолируются друг от друга и интерпретатор не только понимает, в отношении какого объекта вызвана функция, но и какому объекту принадлежит свойство.

2.6. Описываем методы объектов

Реализуйте метод show_info(), как описано ниже. Сразу запустите его у разных объектов.

```
def show_info(self): # В self хранится ссылка на текущий объект
    # Благодаря self интерпретатор понимает, в отношении какого
    # объекта вызван метод или свойство
    # self изолирует объекты друг от друга
    print("Price", self.price)
    print("Sum", self.sum)
    print("Wallet", self.wallet)
    print("Transactions", self.transactions)
    print()
    return

my_btc = Bitcoin() # Создаём объект Биткойн
my_btc.price = 60.0000 # Изменяем свойство одного из объектов
print("Изменяем свойство одного из объектов", my_btc s.price)
print("Цена объекта по умолчанию", mom_btc s.price)
my_btc.show_info() # Тестируем метод show_info()
```

Метод show_info() просто выводит на экран информацию на каждый объект, в отношении которого был вызван.

Реализуйте метод add_wallet(), как описано ниже.

```
def add_wallet(self, address):
    # Изменяет свойство wallet
    # Принимает один аргумент
    # Ничего не отдаёт
    self.wallet = address
    print("Установлен новый кошелёк", self.wallet)
    return # Завершаем метод ничего не отдав

my_btc = Bitcoin() # Создаём объект Биткойн
my_btc.add_wallet("B32p938589uert ") # Тест метода add_wallet()
```

Метод add_wallet() принимает аргумент *address*, в который передаётся хеш-сумма будущего кошелька. Затем свойству текущего объекта self.wallet присваивается новое значение.

Реализуйте методы add_sum() и minus_sum(). Они будут использованы внутри метода create_transaction().

```
def add_sum(self, new_sum):
    # Добавляет деньги на кошелёк
    # Принимает один аргумент
    # Ничего не отдаёт
    print("Добавляю новую сумму", new_sum)
    print("Текущее значение свойства self.sum", self.sum)
    self.sum += new_sum
    print("Новое значение свойства self.sum", self.sum)
    return # Завершаем метод, ничего не отдав

def minus_sum(self, new_sum):
    # Снимает деньги с кошелька
    # Принимает один аргумент
    # Ничего не отдаёт
    print("Вычитаю новую сумму", new_sum)
    print("Текущее значение свойства self.sum", self.sum)
    self.sum -= new_sum
    print("Новое значение свойства self.sum", self.sum)
    return # Завершаем метод, ничего не отдав
```

2.7. Взаимодействие объектов

Как можно догадаться, объекты должны иметь возможность взаимодействовать друг с другом. Ничего не мешает передать объект Bitcoin другому объекту Bitcoin в качестве аргумента. Реализуйте метод create_transaction(), как описано ниже.

```
def create_transaction(self, sum, oponent):
    # Создаёт транзакцию
    # Аргумент oponent принимает объект Биткойн
    # Ничего не отдаёт
    from random import randint # Функция-генератор случайного числа
    tid = randint(109000, 600000900) # Генерируем ID транзакции
    print("Переводим деньги")
    print("Отнимаем у отправителя", sum)

    # У объекта создавшего транзакцию вычитаем сумму перевода
    self.minus_sum(new_sum=sum)

    # Добавляем транзакцию объекту создавшему транзу
    self.transactions.append(tid)

    print("Добавляем оппоненту", oponent, sum, "Денег")

    # Объекту оппонента добавляем сумму перевода
    oponent.add_sum(new_sum=sum)
    oponent.transactions.append(tid) # Добавляем транзу оппоненту
    return # Завершаем метод ничего не отдав
```

Данный метод принимает два аргумента: сумму для отправки и кому она будет отправлена. В качестве второго аргумента мы передаём другой объект Bitcoin, который попадает в переменную аргумента *opponent*. Поскольку теперь там хранится объект Bitcoin, мы можем вызвать у него функцию opponent.add_sum() и начислить ему перевод.

Как видите, через *self* можно получить доступ к текущему свойству объекта и менять его.

Протестируйте вызов методов объекта, как описано ниже.

```
# ОБЪЯВЛЯЕМ 3 ОБЪЕКТА ТИПА БИТКОЙН
my_btc = Bitcoin() # Создаём объект Биткойн
```

```
mom_btc = Bitcoin() # Создаём объект Биткойн
dad_btc = Bitcoin() # Создаём объект Биткойн

my_btc.create_transaction(sum=10, opponent=mom_btc)
# Передали BTC в качестве аргумента
```

2.8. Магический метод __init__(). Конструктор объектов

Магические методы позволяют переопределить стандартное поведение объектов. Магических методов достаточно много, и мы не станем углубляться в них всех. Особенностью магических методов является то, что их имена заранее известны и обрамляются двумя нижними подчеркиваниями с каждой стороны имени.

> Магический метод __init__() позволяет инициализировать свойства объекта в момент создания. Магический метод __init__() ещё называют "конструктором объекта".

В самом начале класса Bitcoin, перед другими функциями, введите следующий код:

```
class Bitcoin():
    def __init__(self, p, s, w):
        # КОНСТРУКТОР ОБЪЕКТА. Помогает инициализировать свойства
        self.price = p # Из аргумента p заполнится свойство self.price
        self.sum = s # Из аргумента s заполнится свойство self.sum
        self.wallet =  w # Из w заполнится свойство self.wallet
        self.transactions = [] # Свойство с пустым списком

new_btc = Bitcoin(p=30000, s=100, w="B3030f0or")
```

Теперь при создании объекта вы сразу можете передать в переменные аргумента значения, которые станут свойствами объекта. Интерпретатор сам вызовет конструктор __init__() и заполнит необходимые свойства.

Согласитесь, это удобнее и занимает меньше кода. Посвятите немного времени знакомству с другими магическими методами.

2.9. Наследование. Создаём Альткойн

Наследование позволяет создать новый класс на базе существующего, что позволяет избежать дублирования кода. Класс, на базе которого происходит наследование, называется родительским, а производный от него класс — дочерним.

При наследовании дочерний класс полностью наследует поведение родителя. При этом в дочерний класс можно добавлять новые свойства и методы. Унаследованные методы можно "перегружать" (переписывать поведение родителя).

Давайте создадим продвинутый Альткойн, который будет полностью наследовать поведение Биткойна и дополнительно иметь возможность анонимных транзакций.

Создайте новый файл, дайте ему название ALTCOINoop.py и скопируйте в него код из листинга 2.2.

Листинг 2.2

```
class Bitcoin():
    def __init__(self, p, s, w):
        # КОНСТРУКТОР ОБЪЕКТА. Помогает инициализировать свойства
        self.price = p # Из p заполнится свойство self.price
        self.sum = s # Из аргумента s заполнится свойство self.sum
        self.wallet =  w # Из w заполнится свойство self.wallet
        self.transactions = [] # Свойство с пустым списком

    # Описываем действия будущих объектов
    def show_info(self): # В self хранится ссылка на текущий объект
        # Благодаря self интерпретатор понимает, в отношении какого
        # объекта вызван метод или изменено свойство
        # self изолирует объекты друг от друга
        print("Price", self.price)
        print("Sum", self.sum)
        print("Wallet", self.wallet)
        print("Transactions", self.transactions)
        return # Завершаем метод ничего не отдав

    def add_wallet(self, address):
        # Изменяет свойство wallet
```

```
        # Принимает один аргумент
        # Ничего не отдаёт
        self.wallet = address
        print("Установлен новый кошелёк", self.wallet)
        return # Завершаем метод ничего не отдав

    def add_sum(self, new_sum):
        # Добавляет деньги на кошелёк
        # Принимает один аргумент
        # Ничего не отдаёт
        print("Добавляю новую сумму", new_sum)
        print("Текущее значение свойства self.sum", self.sum)
        self.sum += new_sum
        print("Новое значение свойства self.sum", self.sum)
        return # Завершаем метод ничего не отдав

    def minus_sum(self, new_sum):
        # Снимает деньги с кошелька
        # Принимает один аргумент
        # Ничего не отдаёт
        print("Вычитаю новую сумму", new_sum)
        print("Текущее значение свойства self.sum", self.sum)
        self.sum -= new_sum
        print("Новое значение свойства self.sum", self.sum)
        return # Завершаем метод ничего не отдав

    def create_transaction(self, sum, oponent):
        # Создаёт транзакцию
        # Аргумент oponent принимает объект Биткойн
        # Ничего не отдаёт
        from random import randint # Функция-генератор случайного числа
        tid = randint(109000, 600000900) # Генерируем ID транзакции
        print("Переводим деньги")
        print("Отнимаем у отправителя", sum)

        # У объекта создавшего транзакцию вычитаем сумму перевода
        self.minus_sum(new_sum=sum)

        # Добавляем транзакцию объекту создавшему транзу
        self.transactions.append(tid)

        print("Добавляем оппоненту", oponent, sum, "Денег")

        # Объекту оппонента добавляем сумму перевода
        oponent.add_sum(new_sum=sum)
    oponent.transactions.append(tid) # Добавляем транзу оппоненту
    return # Завершаем метод ничего не отдав
```

```
# Наследуемся от Биткойна. Биткойн-родитель - Ghost-сын
class Ghost(Bitcoin):
    def __init__(self, p, s, w):
        super().__init__(p, s, w)  # Вызываем конструктор родителя
        self.anon = True # Добавляем новое свойство
        self.anon_transactions = [] # Добавляем новое свойство

    def show_info(self): # ПЕРЕГРУЗКА РОДИТЕЛЬСКОГО МЕТОДА
        print("Price", self.price)
        print("Sum", self.sum)
        print("Wallet", self.wallet)
        print("Transactions", self.transactions)
        print("Anon", self.anon)
        print("self.anon_transactions", self.anon_transactions)
        return

ghost = Ghost(p=2, s=20000, w="Gldfgk456") # Создаём объект Ghost
ghost.show_info() # Унаследованный и перегруженный от родителя метод
ghost.add_sum(30) # Унаследованный метод родителя
```

Как видите, при объявлении класса Ghost в круглых скобках мы передали имя класса Bitcoin. Это заставит интерпретатор полностью унаследовать поведение Bitcoin в классе Ghost.

Класс Bitcoin выступает в роли родителя для класса Ghost.

Класс Ghost является дочерним по отношению к Bitcoin.

В конструкторе __init__() мы добавили новые свойства self.anon = True и self.anon_transactions = [], чтобы дать явное различие между этими классами и расширить поведение Ghost.

Функция super() существует для того, чтобы обращаться к функциям родительского класса. В данном случае мы передаём конструктору родителя аргументы из конструктора Ghost. Как по мне, выглядит откровенно уродливо, но приходится пользоваться.

Теперь, поскольку класс Ghost унаследовал полное поведение класса Bitcoin, мы можем использовать как функции Bitcoin, так и добавлять новые функции Ghost.

В строке def show_info(): мы заново описали поведение родительского метода show_info(). Просто добавили вывод информации о возможности

анонимных транзакций. Переопределение родительского метода называется "перегрузкой".

2.10. Перегрузка методов

Перегрузка методов позволяет переопределять поведение функций из родительского класса в дочернем.

В родительском классе у нас есть метод show_info(), но он не отображает информацию об анонимных транзакциях.

Поскольку мы НЕ использовали новое имя для этой функции, интерпретатор переопределит поведение родительской функции. Это и есть перегрузка метода. Соответственно, не ломая интерфейс использования, мы создали элегантный и более продвинутый Альткойн.

На этом закончим базовое изучение ООП. Этого достаточно, чтобы эффективно пользоваться большинством доступных библиотек и понимать, что происходит, во время работы от унаследованных классов.

2.11. Знакомимся с API BYBIT

Теперь, поскольку вы познакомились с ООП, можем приступать к практической работе. Давайте сразу "пощупаем" библиотеку для работы с биржей Bybit.

В командной строке наберите:

```
pip install pybit
```

Библиотека pybit позволяет получить доступ как к открытым, так и закрытым данным биржи Байбит. К открытым данным относится получение информации о монете и свечах, а к закрытым — размещение и контроль ордеров.

Напомню, что практически любая биржа обладает API-доступом, который делится на два типа:

1. Публичный доступ.

2. Приватный доступ.

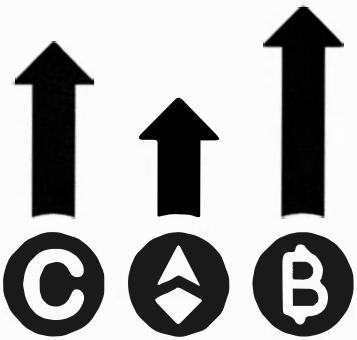
Публичный доступ позволяет получить информацию обо всех парах, размещённых на биржах. Можно получить информацию о ценах активов, об объёме торгов, открытых ордерах, исполненных сделках и т. д. Для открытия сделок необходим приватный ключ. Инструкция по получению приватного ключа есть в главе 5.

2.12. Тестируем API Bybit

Создайте новый файл PYBITTest.py и перепишите следующие строки:

```
from pybit.unified_trading import HTTP # Класс подключения к Bybit
from pprint import pprint # Для красивого вывода Словарей

session = HTTP( # Создаём объект подключения к Bybit
   testnet=False)

# Получаем информацию о последних 13 свечах
pprint(session.get_kline(
   category="linear",
   symbol="LTCUSDT",
   interval=1,
   limit=13))

Выведет словарь: >>>
{"result": {"category": "linear",
            "list": [["1749313200000",
                      "89.13",
                      "89.13",
                      "89.12",
                      "89.12",
                      "2.7",
                      "240.624"]...],
            "symbol": "LTCUSDT"},
 "retCode": 0,
 "retExtInfo": {},
 "retMsg": "OK",
 "time": 1749313206006}
```

```
# Получаем данные об открытом интересе
pprint(session.get_open_interest(
   category="inverse",
   symbol="BTCUSD",
   intervalTime="5min",
   startTime=1669571100000,
   endTime=1669571400000))

Выведет словарь: >>>
{"result": {"category": "inverse",
            "list": [{"openInterest": "344751118.0000",
                      "timestamp": "1669571400000"},
                     {"openInterest": "344762343.0000",
                      "timestamp": "1669571100000"}],
            "nextPageCursor": "",
            "symbol": "BTCUSD"},
 "retCode": 0,
 "retExtInfo": {},
 "retMsg": "OK",
 "time": 1749313206852}

# Информация об инструменте
pprint(session.get_instruments_info(
 category="linear",
 symbol="BTCUSDT"))

Выведет словарь: >>>
{"list": [{"baseCoin": "BTC",
           "contractType": "LinearPerpetual",
           "copyTrading": "both",
           "fundingInterval": 480,
           "isPreListing": False,
           "launchTime": "1584230400000",
           "leverageFilter": {"leverageStep": "0.01",
                              "maxLeverage": "100.00",
                              "minLeverage": "1"},
           "lotSizeFilter": {"maxMktOrderQty": "119.000",
                             "maxOrderQty": "1190.000",
                             "minNotionalValue": "5",
                             "minOrderQty": "0.001",
                             "postOnlyMaxOrderQty": "1190.000",
                             "qtyStep": "0.001"},
           "priceScale": "2",
           "quoteCoin": "USDT",
           "riskParameters": {"priceLimitRatioX": "0.01",
                              "priceLimitRatioY": "0.02"},
           "unifiedMarginTrade": True,
           "upperFundingRate": "0.005"}],
"nextPageCursor": "}
```


C
B

Глава 3.

Создание чат-ботов. Финансовые помощники в Телеграме

Содержание главы:

3.1. Как превратить чат-бота в помощника?

В этом разделе на базе полученного навыка из прошлых двух глав вы научитесь создавать простых и эффективных помощников, которые помогут повысить эффективность и позволят освободить часть своего внимания для более приятных занятий. В конце вы перенесёте в Телеграм код Биржи опционов из первой главы в качестве прототипа настоящей биржи.

Как уже не раз говорилось, рынок криптовалют является манипулятивным и, по сути, начинает расти тогда, когда в монету заносятся деньги. Подобные вливания отражаются на её стоимости и, благодаря своевременному отслеживанию резких изменений цены, можно находить как начало крупного движения, так и его завершение.

Для начала мы разработаем простейшего бота, который будет следить за ценой вашего основного актива и сигнализировать каждый раз, когда произойдет пересечение интересных для вас цен.

Затем вы научитесь следить за всем рынком одновременно, наблюдая за изменениями цены в последние 15 минут. Если цена актива изменится на 5% и более, вероятно, в монету заливаются деньги. Если деньги заливаются, значит она будет продолжать расти либо уже находится на своем пике.

Дополнительно мы создадим скринер открытого интереса, который учитывает ещё НЕ исполненные позиции рынка. Именно на фьючерсах происходит до 90% всей торговли на рынке криптовалют, поскольку легко завлечь людей кредитным плечом, позволяющим начать торговлю с меньшим депозитом. Соответственно, фьючерсный рынок напрямую влияет на спотовую торговлю, и мы сможем легко использовать информацию о его состоянии в собственных целях.

Увенчаем всё отслеживанием плотностей в стакане. Существует стратегия так называемого "отскока от плотности", которая предполагает, что чем крупнее заявка по той или иной цене, тем больше шансов, что цена отскочит от неё.

Также мы добавим возможность многопользовательской работы благодаря внедрению баз данных.

Большая часть подобных ботов платная, несмотря на то что все данные о изменениях цен и показателей открытого интереса находятся в свободном доступе. Достаточно лишь аккумулировать их простыми самописными программами.

3.2. Знакомимся с API Telegram

Телеграм достаточно популярный мессенджер, несмотря на то что мы часто подвергаем критике лживую политику конфиденциальности, а также поведение его владельцев. Тем не менее, мессенджер плотно проник в нашу жизнь и даёт достаточно продвинутую платформу для создания сторонних приложений.

С некоторых пор, помимо обычных чат-ботов, Телеграм даёт возможность создавать полноценные приложения на базе веб-технологий, так называемые Telegram Apps. Если вы немного освоите JavaScript, то легко сможете адаптировать некоторые боты, созданные в этой главе, для Telegram Apps.

Пайтон имеет достаточно большое количество библиотек для управления API Телеграм, но они все мне НЕ по душе. Порог входа для начала использования этих библиотек достаточно высок, при том что само по себе API Телеграм достаточно простое.

Дабы избежать излишней нагрузки при освоении новых библиотек, мы обойдемся простыми HTTP-запросами, достаточными для работы. Официальная документация находится по адресу *https://core.telegram.org/*.

Вероятно, это не самый элегантный подход, и есть библиотеки, которые возьмут на себя часть рутины по обработке входящих сообщений от пользователей. Зато данный способ позволит напрямую прочувствовать работу с API, и при переходе на одну из библиотек она НЕ будет казаться для вас "чёрным ящиком".

В идеале перед началом разработки необходимо прочитать документацию полностью. На это может уйти несколько дней и в большинстве случаев профессиональные разработчики не жалеют времени на предварительное исследование возможностей выбранной библиотеки.

Условно разделить работу с API можно на две части:

1. Получение сообщений от пользователя.

2. Обработка сообщений от пользователя.

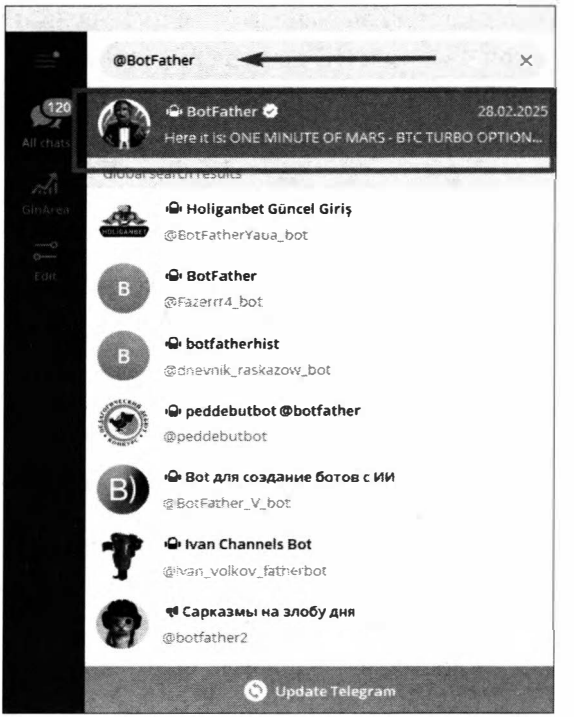
Наша задача лишь обеспечить непрерывную слежку за поступлением новых данных от пользователя и, конечно же, успевать эти данные обрабатывать.

Перейдём от слов к делу! Создадим первого бота и получим уникальный токен для доступа к нему по API.

3.3. Botfather — создаём первого бота

Для управления всеми ботами, созданными в Telegram, существует главный бот @BotFather. Создание новых ботов и управление уже существующими происходит только через него.

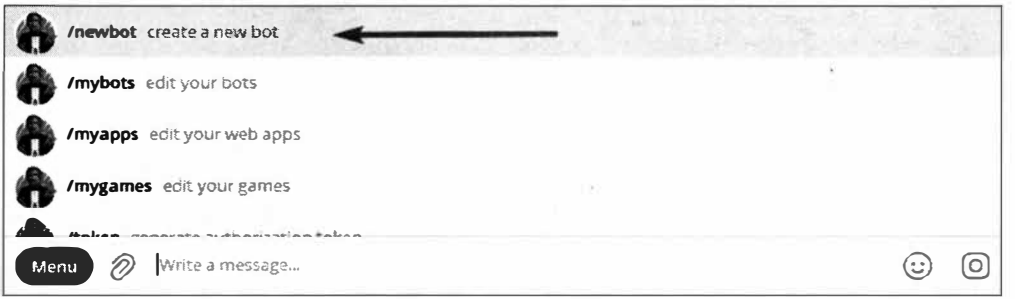
1. В поисковой строке Телеграм наберите @BotFather.



Изображение 3.1.

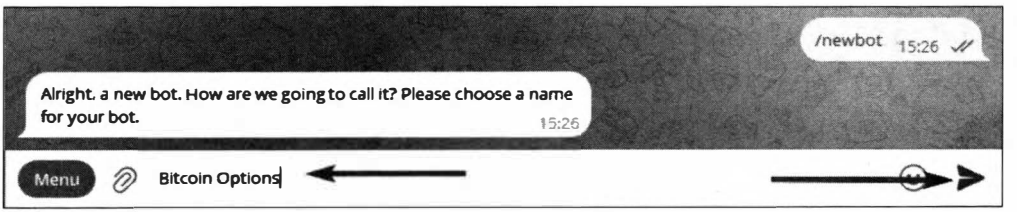
2. Перейдите в @BotFather и нажмите START. Убедитесь, что бот официальный. У него должно быть не менее 2 миллионов ежемесячных пользователей.

3. Нажмите кнопку MENU и выберите команду /newbot.



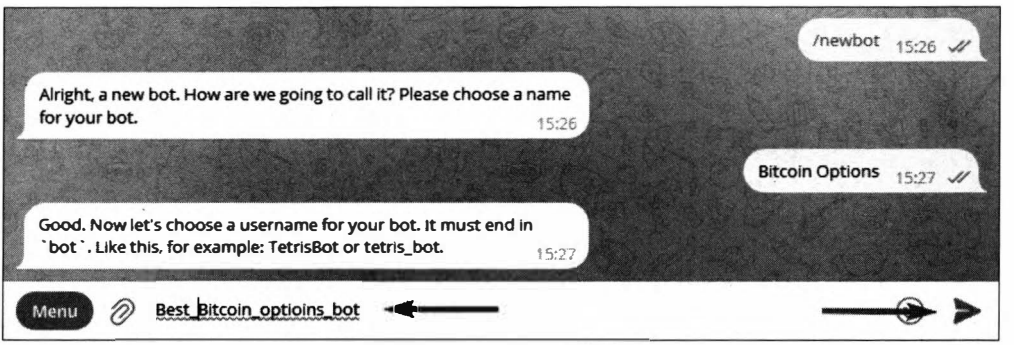
Изображение 3.2.

4. Дайте название своему боту. Например, Straight_Back_bot.



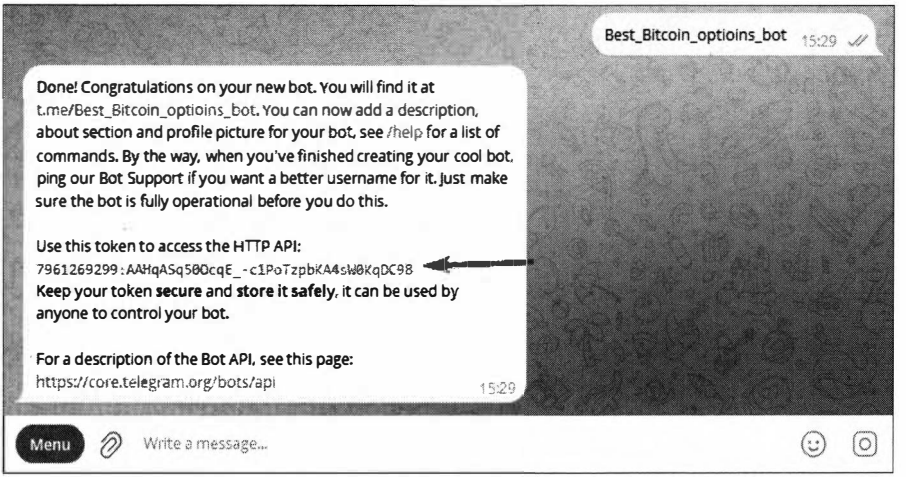
Изображение 3.3.

5. Придумайте ссылку для бота. Обратите внимание, что любая ссылка должна заканчиваться словом *bot*.



Изображение 3.4.

6. На последнем шаге вы получите токен доступа. Он необходим, чтобы иметь возможность отправлять боту команды и следить за обновлениями. Сохраните его в надёжном месте.



Изображение 3.5.

3.4. Получаем свой CHAT_ID

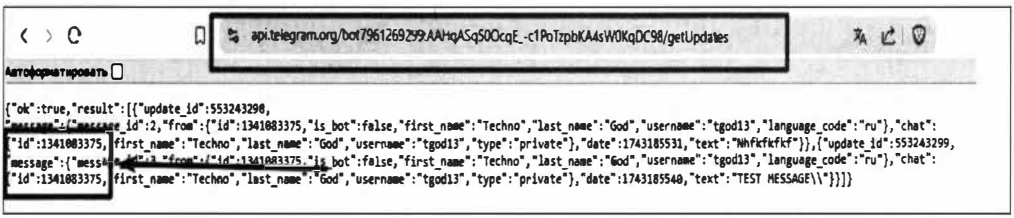
После того как бот был создан, запустите его и отправьте ему любое сообщение. Затем перейдите по ссылке для получения обновлений.

Ссылка для получения обновлений будет выглядеть так:

https://api.telegram.org/bot[BOT_ID]/getUpdates

где [BOT_ID] — это токен вашего бота.

Вы увидите примерно следующее содержимое:



Изображение 3.6.

Обратите внимание, что это ответ в формате JSON и он очень похож на обычный словарь.

Найдите текст своего сообщения. Перед ним будет ключ *chat*. Значение *id*, которое хранится по этому ключу, является уникальным идентификатором чата, он позволит отправлять индивидуальные сообщения.

Сохраните идентификатор в надёжном месте, он нам понадобится.

3.5. Отправляем сообщения себе

Ссылка для отправки сообщений пользователю будет выглядеть примерно так:

https://api.telegram.org/bot[BOT_ID]/sendMessage?chat_id=[CHAT_ID]&text="ПРИВЕТ!"

У нас есть всё необходимое, чтобы начать отправлять сообщения самим себе. Создайте новый файл SENDTGMessage.py и опишите функцию send_message(), как описано ниже.

```
from requests import post # Для отправки POST-запросов

TG_API_URL = "https://api.telegram.org/bot"
TGBOT = "7579403178:AAEU_aJ809TM1IM1OFZYaOLboWvLDrEZa7A"

def send_message(msg="Тестовое сообщение", chat_id=1349083375):
    # Функция для отправки сообщений в Телеграм
    # Принимает два аргумента: текст сообщение и id чата
    # В переменной final_url формируется ссылка для отправки
    # Возвращает ответ от ТГ
    url1 = TG_API_URL+TGBOT
    url2 = "/sendMessage?chat_id="+str(chat_id)+"&text="+str(msg)
    final_url =  url1+ url2 # Конкатенируем ссылку
    response = post(final_url) # Отправляем и сохраняем результат
    return response # Завершаем функцию

send_message() # Вызываем функцию отправки сообщений
send_message(msg="Я очень разговорчивая программа")
send_message(msg="")
```

Функция предельно проста. Она конструирует **url** для отправки сообщения по HTTP и отправляет в ваши личные сообщения всё, что ей было передано в единственном аргументе *msg*.

Запустите исходник. Если Телеграм прислал вам сообщение, поздравляю! Это успех!

Задача 3.1

Запустите функцию рекурсивно с задержкой в 1 секунду. Телеграм пришлет вам 1000 сообщений.

3.6. Получаем сообщения от пользователей бота

Напишем небольшую программу, которая будет следить за поступлением новых сообщений от пользователя. Эту заготовку мы будем использовать в дальнейшем для обработки команд от пользователей. Создайте файл GetTgUpdates.py и перенесите код из листинга 3.1.

Листинг 3.1

```
latest_update_id = None # Для хранения ID последнего сообщения

def get_updates(offset="-1"):
    # Функция для получения последнего сообщения
    # Принимает один аргумент со значением по умолчанию
    # Отдаёт последнее сообщение отправленному в бот
    # Формат ответа:
    # {'message': {'chat': {'first_name': 'Techno',
    #                       'id': 1341083375,
    #                       'last_name': 'God',
    #                       'type': 'private',
    #                       'username': 'tgod13'},
    #                'date': 1740601322,
    #                'entities': [{'length': 6,
    #                              'offset': 0,
    #                              'type': 'bot_command'}],
    #                'from': {'first_name': 'Techno',
    #                         'id': 1341083375,
    #                         'is_bot': False,
    #                         'language_code': 'en',
    #                         'last_name': 'God',
    #                         'username': 'tgod13'},
    #                'message_id': 3168,
    #                'text': '/order'},
```

```
    #           'update_id': 577412870}

    # Делаем GET запрос и сохраняем ответ
    url = TG_API_URL+TGBOT+"/getUpdates?offset="+offset
    result = get(url).json() # Сохраняем ответ в JSON
    try:
        # Пытемся отдать последнее сообщение
        return result["result"][0]
    except:
        return None # Завершаем функцию

def run():
    # Функция для запуска обработки регистраций и других команд
    # Ничего не принимает
    # Ничего не отдаёт
    global latest_update_id # Подтягиваем из глобальной области
    sleep(2) # Засыпаем на 2 секунды
    print("latest_update_id", latest_update_id)
    check_update = get_updates() # Получаем последнее сообщение

    # Если id последнего сообщения бота НЕ обновилось
    if latest_update_id == check_update["update_id"]:
        print("Обновлений нет")
    else:
        # Если id последнего сообщения бота обновилось
        print("Получил обновления")
        latest_update_id = check_update["update_id"] # ID обновления
        chat_id = int(check_update["message"]["chat"]["id"])

        print("ID пользователя:", chat_id,
            "Сообщение:", check_update["message"]["text"])
        # Здесь можно отслеживать содержимое сообщения и реагировать
    run() # Рекурсивно перезапускаем функцию
    return # Завершаем текущее выполнение функции

run() # Запускаем отслеживание сообщений
```

Мы рассмотрели две простые функции, которые хорошо описаны и должны быть вам понятны.

Аргумент *offset=-1 get_updates()* говорит о том, что мы пытаемся получить не все сообщения, отправленные боту за последнее время, а только самое последнее.

Функция run() будет перезапускаться каждые 2 секунды и отслеживать, пришло ли новое сообщение от пользователя или нет. Отправьте несколько сообщений в бот для теста.

3.7. Следим за осанкой и ценой Биткойн

Занимаясь финансами, важно позаботимся и о своем здоровье. Создадим простого бота, который каждый час будет присылать напоминание о том, что нужно выпрямить спину вдобавок к текущей котировке BTC.

Если ещё этого не сделали, то наберите в терминале:

```
pip install pybit
```

Создайте новый проект и перепишите исходник, как описано ниже:

```
from time import sleep # Для засыпания скрипта
from pybit.unified_trading import HTTP # Класс для подключения к бирже
from requests import post # Для отправки POST-запросов

TG_API_URL = "https://api.telegram.org/bot"
TGBOT = "7579403178:AAEU_aJ809TM1IM1OFZYaOLboWvLDrEZa7A"
session = HTTP(testnet=False) # Создаём объект подключения к бирже
# testnet=False означает, что работаем с реальными данными

def send_message(msg="Тестовое сообщение", chat_id=1349083375):
    # Функция для отправки сообщений в Телеграм
    # Принимает два аргумента: текст сообщение и id чата
    # В переменной final_url формируется ссылка для отправки
    # Возвращает ответ от ТГ
    url1 = TG_API_URL+TGBOT
    url2 = "/sendMessage?chat_id="+str(chat_id)+"&text="+str(msg)
    final_url =  url1+ url2 # Конкатенируем ссылку
    response = post(final_url) # Отправляем и сохраняем результат
    return response # Завершаем функцию

def get_symbol_current_price(symbol="BTCUSDT"):
    # Функция-обертка для получения последней цены актива
    # Принимает один аргумент symbol со значением по умолчанию BTCUSDT
    # Отдаёт текущую цену

    # У объекта session вызываем функцию-метод get_tickers()
    # Передаём два аргумента:
    # category - тип рынка (спот, фьючерсы) и символ
    result = session.get_tickers(
        category="linear",
        symbol=symbol)
```

```
    # Извлекаем только цену
    current_price = result["result"]["list"][0]["lastPrice"]
    print("Текущая цена", current_price)
    return current_price # Отдаём результат

for a in range(1, 24):
    # Получаем текущую цену BTCUSDT
    cur_price = get_symbol_current_price()
    msg = "Текущая цена BTC: " + str(cur_price) # Формируем сообщение
    send_message("Выпрями спину! " + msg)  # Запускаем на тест
    sleep(3600) # Засыпаем на 1 час
```

В первую очередь на всякий случай мы установили новую зависимость, библиотеку **pybit**.

Она позволяет получить доступ как к открытым, так и к закрытым данным биржи Bybit.

К открытым данным относится получение информации о монете и свечах, а к закрытым — размещение и контроль ордеров. Затем импортировали функцию post() из библиотеки **requests** для отправки сообщений в Телеграм API.

Далее была объявлена функция get_symbol_current_price(), принимающая один аргумент со значением по умолчанию. Если вызвать функцию и не передать никаких данных на вход, переменная аргумента *symbol* заполнится значением BTCUSDT.

После был создан объект session, через который происходит общение с биржей. Напомню, что HTTP(testnet=False) — это конструктор класса HTTP(), и мы заполнили один из его аргументов строкой testnet=False. Если указать testnet=True, то биржа будет отдавать тестовые данные, которые не являются актуальными.

В переменной *result* функции get_symbol_current_price() был сохранен ответ от session.get_tickers(). Если в метод get_tickers() не передать аргумент *symbol*, придёт ответ о текущем состоянии всего рынка. В конце функции из переменной *result* мы извлекли только текущую цену и отдали результат. Функция send_message() уже вам знакома и, надеюсь, хорошо понятна.

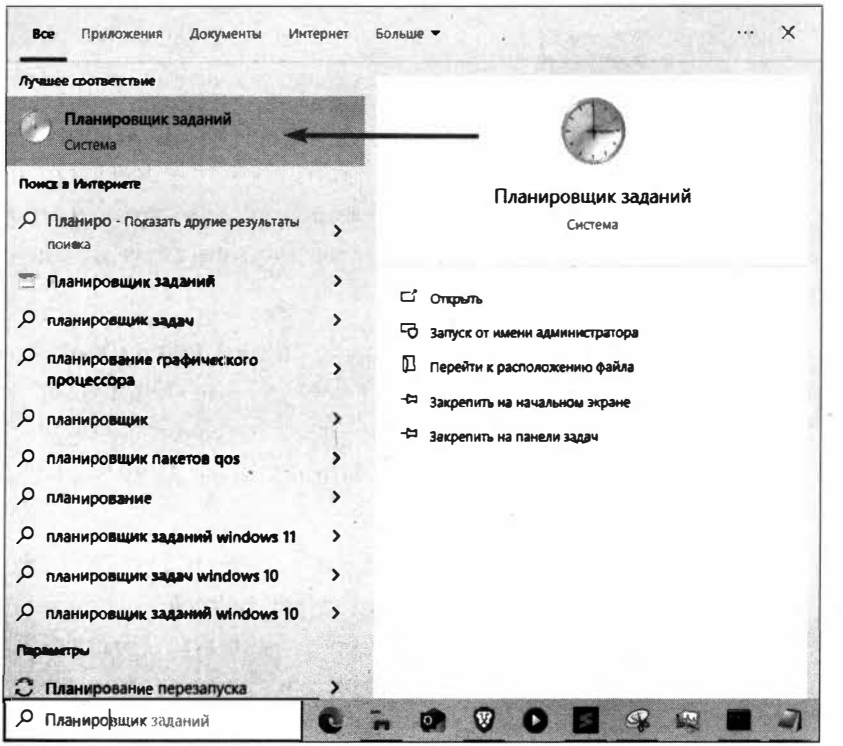
Она просто отправит данные, полученные от get_symbol_current_price(), в ваш Телеграм.

Затем мы запустили цикл FOR:. Функция range() создаст список из 24 элементов, что кратно 24 часам. Внутри цикла мы отправляем сообщение и засыпаем на 1 час.

3.8. Автоматизируем перезапуск бота локально

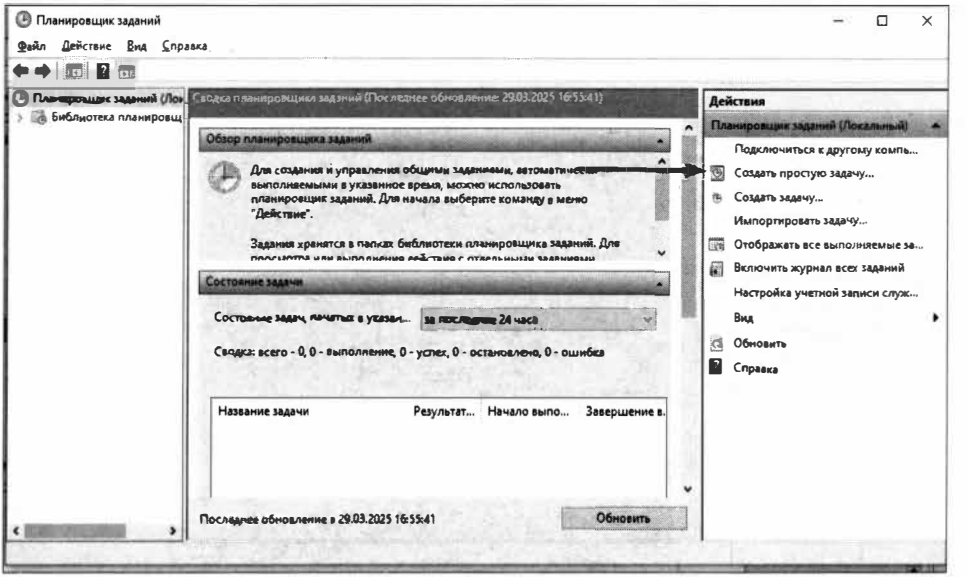
Для самых простых ботов в качестве сервера можно использовать ваш собственный компьютер. В системах WINDOWS есть приложение, которое позволяет перезапускать любое приложение по расписанию.

1. Нажмите **Пуск** и наберите **Планировщик заданий**.



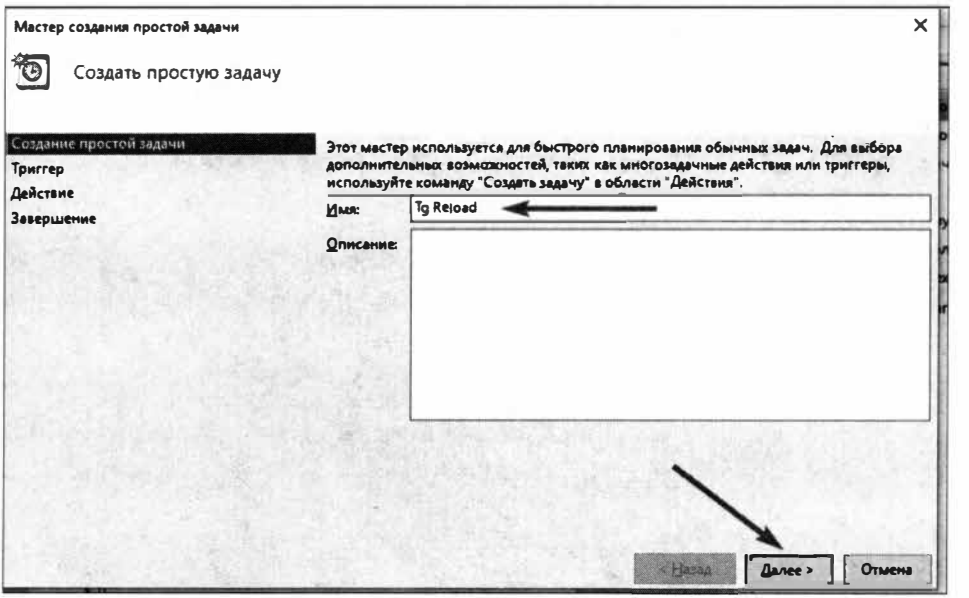
Изображение 3.7.

2. В появившемся окне нажмите **Создать простую задачу**.



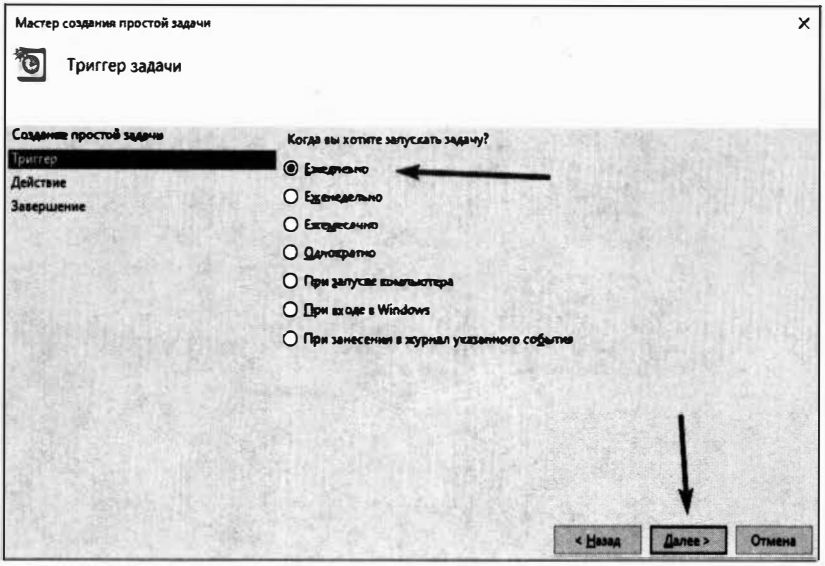
Изображение 3.8.

3. В появившемся окне введите название задачи и нажмите кнопку **Далее**.



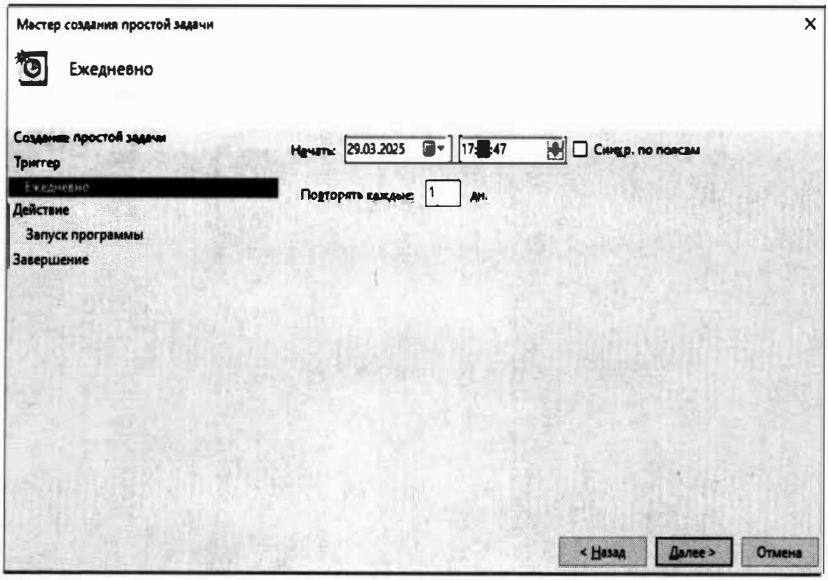
Изображение 3.9.

4. Затем выберите **Ежедневно** и нажмите **Далее**.



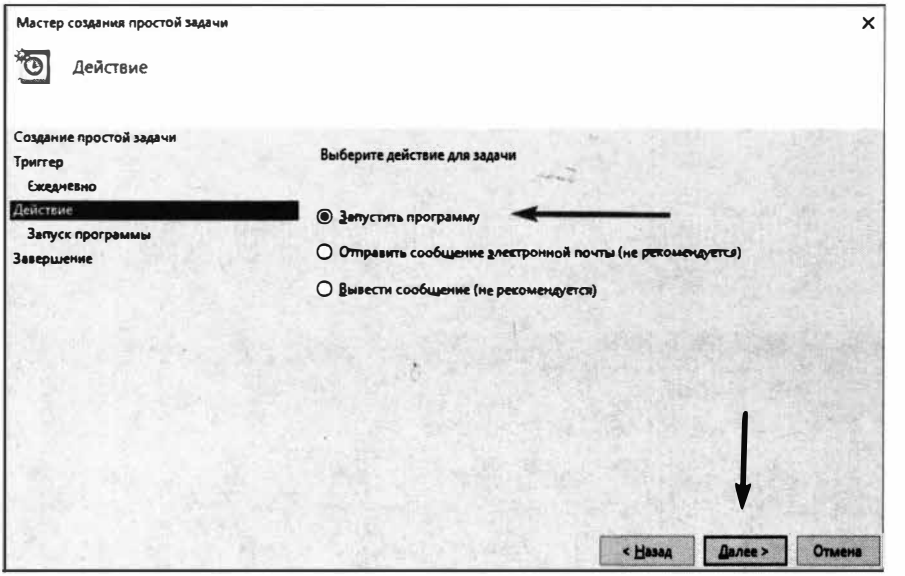
Изображение 3.10.

5. Выберите время начала выполнения задачи и интервал повторения.



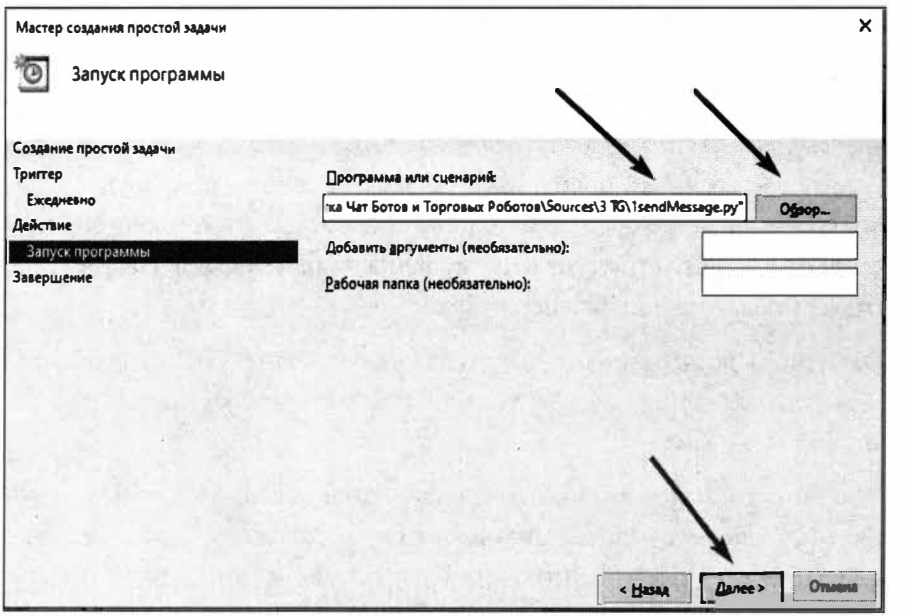
Изображение 3.11.

6. Выберите пункт **Запустить программу** и нажмите **Далее**.



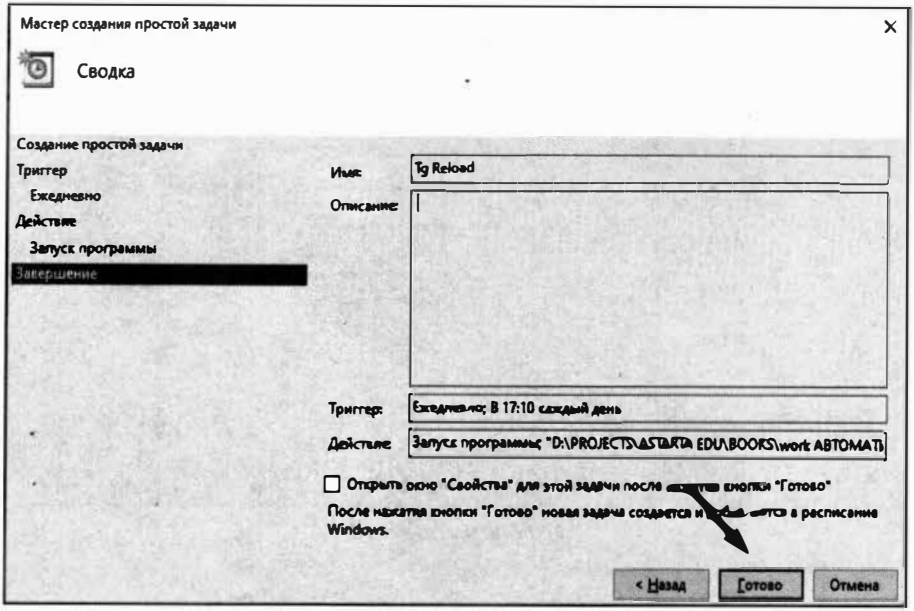
Изображение 3.12.

7. Выберите ваш файл со скриптом и нажмите **Далее**.



Изображение 3.13.

8. Просто нажмите на кнопку **Готово** и дождитесь, когда придёт первое сообщение.



Изображение 3.14.

Пожалуйста, воспользуйтесь поисковиком, и вы найдёте более гибкие настройки планировщика заданий.

Если у вас возникли проблемы с запуском скрипта, попробуйте сделать из .py-файла полноценный .exe. В поиске найдётся достаточно библиотек, которые в несколько строк создадут исполняемый .exe-файл. Например, рассмотрите библиотеки Pyinstaller, py2exe.

Отлично! Простейший бот для отслеживания цены BTC готов. Как оформить исходный код для перезапуска в LINUX-системах с помощью CRON, будет описано позже.

Для того чтобы грамотно отслеживать одновременно несколько активов, а также обеспечить многопользовательскую рассылку, вам понадобится навык работы с базами данных. Немного позже вы его получите, а сейчас попробуем отслеживать интересные цены BTC.

3.9. Следим за интересными ценами BTC

Работу текущего бота можно описать буквально несколькими этапами:

1. Запускаем бота и получаем текущую цену BTC.
2. Проверяем, находится ли цена в указанном диапазоне.
3. Если цена находится в указанных диапазонах, отправляем сообщение в Телеграм.

Для начала создайте новый IntrestingPriceBtc.py и перенесите в него функции send_message() и get get_symbol_current_price(). Затем добавьте код для отслеживания диапазона цен из листинга 3.2.

Листинг 3.2

```
from pybit.unified_trading import HTTP # Класс для подключения к бирже
from requests import post # Для отправки POST-запросов

TG_API_URL = "https://api.telegram.org/bot"
TGBOT = "7579403178:AAEU_aJ809TM1IM1OFZYaOLboWvLDrEZa7A"

session = HTTP(testnet=False) # Создаём объект подключения к бирже
# testnet=False означает, что работаем с реальными данными

def send_message(msg="Тестовое сообщение", chat_id=1349083375):
    # Функция для отправки сообщений в Телеграм
    # Принимает два аргумента: текст сообщение и id чата
    # В переменной final_url формируется ссылка для отправки
    # Возвращает ответ от ТГ
    url1 = TG_API_URL+TGBOT
    url2 = "/sendMessage?chat_id="+str(chat_id)+"&text="+str(msg)
    final_url =  url1+ url2 # Конкатенируем ссылку
    response = post(final_url) # Отправляем и сохраняем результат
    return response # Завершаем функцию

def get_symbol_current_price(symbol="BTCUSDT"):
    # Функция-обертка для получения последней цены актива
    # Принимает один аргумент symbol со значением по умолчанию BTCUSDT
    # Отдаёт текущую цену

    # У объекта session вызываем функцию (метод) get_tickers()
    # Передаём два аргумента:
    # category - тип рынка (спот, фьючерсы) и символ
```

```
    result = session.get_tickers(
        category="linear",
        symbol=symbol)

    # Извлекаем только цену
    current_price = result["result"]["list"][0]["lastPrice"]
    print("Текущая цена", current_price)
    return current_price # Отдаём результат

# Получаем текущую цену BTCUSDT
cur_price = float(get_symbol_current_price())
msg = "Цена BTC: " + str(cur_price) # Формируем сообщение


# Проверяем диапазоны цен и уведомляем пользователя
if cur_price >= 56000 and cur_price <= 59000:
    send_message(msg)
    send_message("Время открыть позицию")
elif cur_price >= 69000 and cur_price <= 76000:
    send_message(msg)
    send_message("Время фиксировать прибыль")
if cur_price >= 100000:
    send_message(msg + " Стоит дождаться коррекции")
else:
    print("Условия для входа или фиксации не подходят")
```

Как всегда, были импортированы необходимые библиотеки и перенесены функции get_symbol_current_price() и send_message() из предыдущих примеров.

Получив цену и проверив диапазон, мы проверили значение на соответствие. Если одна из проверок вернет истину, то будет отправлено соответствующее сообщение. Теперь каждый раз, когда исходник будет запущен, бот пришлёт уведомление, если условия для этого подходят. Настройте перезапуск приложения, как показано в предыдущем примере.

3.10. Что такое скринер криптовалют?

Скринеры криптовалют позволяют отслеживать резкие изменения на рынке и снимают с человека нагрузку по круглосуточному мониторингу.

Скринеры помогают высвободить время и силы для анализа точки входа на рынок. Хорошим примером бесплатного скринера является сайт WWW.МАЯК.ГОСТ.РУС.

Когда цена монеты взлетает на 5% и более за последние 15 минут, вероятно, её накачивают деньгами. По сути, скринер помогает отслеживать начало или окончание манипуляций на рынке.

Отслеживать можно резкие взлеты цены, резкие падения, резкие изменения открытого интереса или появление крупных лимитных заявок, от которых часто происходит отскок цены.

Также можно пробовать мониторить последние трейды и находить крупные исполненные сделки (выход объёмов), которые помогают росту или падению цены.

В целом скринеры отличные помощники в отслеживании движений на рынке. То, что для человека является крайне обременительным, для бота легче лёгкого.

Основной недостаток — это то, что большая часть скринеров платная, и, если речь идёт о скринере в Телеграме, мы не можем быть уверены, что сигнал пришёл вовремя, поскольку не имеем доступа к серверу, с которого пришла рассылка.

Давайте реализуем несколько собственных скринеров.

3.11. Создаём памповый скринер

Самым простым сигналом для отслеживания пампов (резких лонговых движений) можно считать изменение цены на 5% и более за последние 15 минут. Как всегда, в первую очередь распишем ТЗ и решение задачи.

1. Обеспечиваем отправку уведомлений.
2. Обеспечиваем получение имен всех монет.
3. Обеспечиваем получение и сравнение цен монеты за последние 15 минут.

Создайте новый файл PUMPScrinner.py и перепишите код из листинга 3.3.

Листинг 3.3

```
from pybit.unified_trading import HTTP # Класс для подключения к бирже
from requests import post # Для отправки POST-запросов
from datetime import datetime # Объект для работы со временем

TG_API_URL = "https://api.telegram.org/bot"
TGBOT = "7579403178:AAEU_aJ809TMlIM1OFZYaOLboWvLDrEZa7A"
session = HTTP(testnet=False) # Создаём объект подключения к бирже

def send_message(msg="Тестовое сообщение", chat_id=1349083375):
    # Функция для отправки сообщений в Телеграм
    # Принимает два аргумента: текст сообщение и id чата
    # В переменной final_url формируется ссылка для отправки
    # Возвращает ответ от ТГ
    url1 = TG_API_URL+TGBOT
    url2 = "/sendMessage?chat_id="+str(chat_id)+"&text="+str(msg)
    final_url =  url1+ url2 # Конкатенируем ссылку
    response = post(final_url) # Отправляем и сохраняем результат
    return response # Завершаем функцию

def time_formater(unix=1731196800000):
    # Функция для конвертации времени unix-формата
    # Принимает 1 аргумент
    # Отдаёт отформатированную для человека функцию
    timestamp = unix / 1000 # Приводим unix к timestamp
    dt_object = datetime.fromtimestamp(timestamp)
    return dt_object # Отдаём читабельный результат

def get_all_symbols_names():
    # Функция для получения имен всех доступных пар
    # Не принимает аргументов
    # Отдаёт список названий
    print("Пытаюсь получить все символы")
    # Получаем общую информацию обо всех монетах
    response = session.get_instruments_info(category="linear")
    only_names = [] # Подготавливаем пустой список для имен
    # Перебираем информацию в ответе
    for coin in response["result"]["list"]:
        # Извлекаем имя и отправляем в общий список
        only_names.append(coin["symbol"])
    print("Получилось")
    return only_names # Отдаём результат

def check_signal_by_symbol(symbol="BTCUSDT"):
    # Функция получает информацию о разнице цен за последние 15 минут
```

```
    # Если разница в цене больше или равна 5%, присылает сигнал
    # Принимает 1 аргумент со значением по умолчанию
    # Ничего не отдаёт

    # Получаем информацию о последних 13 свечах
    result = session.get_kline(
        category="linear",
        symbol=symbol,
        interval=1,
        limit=13)

    # Извлекаем текущую цену и цену 15-минутной давности
    cur_price = float(result["result"]["list"][0][4])
    ago_15min_price = float(result["result"]["list"][-1:][0][4])

    # Высчитываем, на сколько изменилась цена в процентах
    price_change = (cur_price)-ago_15min_price / cur_price * 100
    print(symbol, price_change)

    # Если цена изменилась на 5% и более
    if price_change >= 5:
        # Подготавливаем сообщение и отправляем сигнал
        msg = symbol + " вырос на " + str(price_change) + "%"
        send_message(msg)
    return # Завершаем функцию

def run():
    # Функция для запуска скринера
    # Получаем имена всех символов биржи
    all_symbols = get_all_symbols_names()
    # Перебираем каждый символ
    for symbol in all_symbols:
        # Проверяем, есть ли сигнал
        check_signal_by_symbol(symbol)
    return # Завершаем функцию

run() # Запуск скринера
```

После импорта библиотек и объявления констант описаны функции send_message(), time_formater() и get_all_symbols_names(). Они хорошо вам знакомы и перетянуты из прошлых проектов.

Функция check_signal_by_symbol() принимает один аргумент — имя монеты. Функция получает данные о свечах за последние 13 минут с помощью метода session.get_kline().

В переменной *result* хранится список из 13 свечей, полученных от биржи. В переменные *cur_price* и *ago_15min_price* извлекаются данные о закрытии необходимых для нас цен.

Затем в переменную *price_change* мы сохраняем изменение цены в процентах. Если цена изменилась на 5% и более, посылаем себе уведомление в Телеграм.

Функция run() запускает скринер. В первую очередь, с помощью функции get_all_symbols_names() мы сохраняем имена всех тикеров в переменную *all_symbols*.

Затем перебираем и с помощью функции check_signal_by_symbol() проверяем каждый символ на изменения.

Данный скрипт необходимо перезапускать каждые 3 минуты. В многопользовательской версии будет представлен пример перезапуска в LINUX-системах. Недостатком скринера из предыдущего примера является тот факт, что перебор всех свечей займет около 3 минут. Для улучшения показателей попробуйте решить следующую задачу:

Задача 3.2

1. Каждые 5 секунд мониторим текущее состояние всех монет (просто цены).
2. Записываем полученные данные в JSON-файл.
3. Сравниваем текущие данные с данными 15-минутной давности.
4. Если цена изменилась на 5% и более, отправляем сообщение.

Для решения задачи вам понадобится JSON-файл, в котором будут храниться списки котировок на каждую монету. В качестве ключа будет выступать название монеты, а в качестве значения — обычный список. Формат внутреннего содержимого следующий:

```
{
      "LTCUSDT": [92.22, 93.33, 94.67, 95.66],
      "LTCUSDT": [92.22, 93.33, 94.67, 95.66],
      "LTCUSDT": [92.22, 93.33, 94.67, 95.66],
}
```

Обновление данных + пауза между запросами в среднем занимает 10 секунд. За минуту выходит 6 запросов. За 15 минут получается около 90 записей. Соответственно, если в списке на любую из монет накапливается больше 90 записей, прежде чем добавить новую запись в конец списка, мы должны удалить первую запись.

Сравнение разницы цены в процентах между первой записью и последней позволит сформировать сигнал.

Вам понадобится одна основная функция run(), в которой будет происходить львиная доля работы. Кроме того, понадобятся 3 вспомогательные функции, которые помогут получить имена всех монет, котировки, а также подготовят JSON-файл для хранения истории в необходимом формате.

3.12. Создаём скринер крупных заявок

Среди профессиональных скальперов, которые торгуют по биржевому стакану, часто можно встретить стратегию отскока от крупных заявок. Для мониторинга крупных заявок преимущественно используются специальные терминалы по типу CScalp.

С помощью этих терминалов трейдеру приходится в режиме реального времени отслеживать появление крупных заявок и реагировать на них, что в свою очередь накладывает дополнительную нагрузку на психическое состояние.

Недостатком данного способа торговли является тот факт, что зачастую крупные заявки являются так называемыми "стенками", которые двигает манимейкер с помощью роботов.

Несмотря на это, есть значительный пласт трейдеров, которые благодаря умелому управлению рисками успешно отрабатывают эту формацию, поэтому иметь такой скринер в арсенале — отличная идея!

Создайте новый файл HIORDERSSkriner.py и перенесите код из листинга 3.4.

Листинг 3.4

```
from pybit.unified_trading import HTTP # Класс для подключения к бирже
from requests import post # Для отправки POST-запросов

TG_API_URL = "https://api.telegram.org/bot"
TGBOT = "7579403178:AAEU_aJ809TMlIMlOFZYaOLboWvLDrEZa7A"

session = HTTP(testnet=False) # Создаём объект подключения к бирже
# testnet=False означает, что работаем с реальными данными

def send_message(msg="Тестовое сообщение", chat_id=1349083375):
    # Функция для отправки сообщений в Телеграм
    # Принимает два аргумента: текст сообщение и id чата
    # В переменной final_url формируется ссылка для отправки
    # Возвращает ответ от ТГ
    url1 = TG_API_URL+TGBOT
    url2 = "/sendMessage?chat_id="+str(chat_id)+"&text="+str(msg)
    final_url =  url1+ url2 # Конкатенируем ссылку
    response = post(final_url) # Отправляем и сохраняем результат
    return response # Завершаем функцию

def sort_func(element):
    # Функция для помощи в сортировке вложенных списков
    # Принимает 1 аргумент - список
    # Возвращает второй элемент из переданного списка
    return float(element[1])

def get_symbol_order_book(symbol="BTCUSDT"):
    # Функция для получения и обработки текущих ордеров актива
    # Принимает 1 аргумент со значением по умолчанию
    # Отправляет 5 самых крупных заявок и их цены
    # Ничего не отдаёт

    # Функция get_orderbook() объекта session возвращает книгу ордеров
    result = session.get_orderbook(
        category="linear",
        symbol=symbol,
        limit="500")

    arr = result["result"]["b"] # Извлекаем ордера из ответа

    # Для сортировки списка списков по второму значению
    arr.sort(key = sort_func)

    # С помощью среза получаем последние 5 элементов
    for elem in arr[-5:]:
        # Извлекаем цену, на которой сконцентрированы заявки
        price = str(elem[0])
```

```
            value = str(elem[1]) # Извлекаем сумму заявок
            msg = "На цене: " + price + " крупный объём заявок: " + value
            print(msg)
            send_message(msg) # Отправляем сообщение
        return # Завершаем функцию

get_symbol_order_book() # Запускаем функцию на тест
```

После импорта всех библиотек и объявления констант была создана функция sort_func(). **Константа** — это неизменяемая переменная, для наглядности объявляется заглавными буквами. Она помогает сортировать список списков. Затем перенесли уже стандартную для каждого проекта функцию send_message().

Функция get_symbol_order_book() принимает один аргумент и затем занимается поиском и сортировкой наиболее крупных заявок. В переменную `result` сохранили результат работы метода session.get_orderbook(). Этот метод возвращает все ордера из стакана, в нашем случае в количестве 500 штук. В переменную `arr` мы извлекли список из данных на каждый ордер.

Строка `arr.sort(key = sort_func)` сортирует список списков от меньшего к большему. В качестве аргумента передаётся функция sort_func(), в которую автоматически будут попадать данные на каждый ордер. Более подробную информацию о сортировке вложенных списков вы найдете в сети.

В завершение в цикле FOR: мы используем срез для получения последних 5 элементов, в которых хранятся наиболее крупные заявки. Информация о каждой заявке будет отправлена в ваш Телеграм. Самостоятельно автоматизируйте перезапуск функции локально.

Отлично! Теперь вы можете мониторить появление крупных заявок и реагировать на них, если рыночная ситуация позволяет.

3.13. Многопользовательские скринеры. Теория баз данных

Первые базы данных появились в 60-х годах. Потребность в организации и структуризации данных финансовых учреждений, таких как банки, поро-

дило появление первых коммерческих баз. Официально первую промышленную базу данных представили в IBM. Я уверен, что они разработали её для военных, как минимум за 50 лет до массового использования.

Базы данных позволяют хранить, структурировать и обрабатывать данные различных типов и создавать связи между записями. Благодаря тому, что между объектами баз данных часто выстраиваются отношения и зависимости, их назвали реляционными (от лат. *relatio* — сообщение).

На мой взгляд, технология реляционных БД — одна из самых живучих на рынке информационных технологий. Старше из по сей день процветающих систем, пожалуй, только радио.

Если опустить излишние технические детали, любую работу с базой данных можно разбить на три основных этапа:

1. Создание базы данных.
2. Описание таблиц.
3. Обработка записей в таблицах (добавление, удаление, обновление, чтение).

Если вы не пропустили раздел, посвященный объектно-ориентированному программированию, то легко проложите аналогию между таблицами и записями, такую же как между классами и объектами.

В ООП мы сначала описываем класс, затем создаём объекты.

В БД мы сначала описываем таблицы, затем создаём записи.

После создания базы данных проектируются **таблицы**. Каждая таблица содержит в себе поля будущих записей. По аналогии со свойствами объектов в ООП.

Поля записей бывают четырех основных типов и уже хорошо вам известны:

1. Строка.

2. Целое число.
3. Число с плавающей точкой.
4. Логический тип.

После объявления структуры таблицы можно добавлять в неё записи. Основной процесс работы с записями в таблице называется CRUD (сокращение от: Create — создание, Read — чтение, Update — обновление, Delete — удаление).

Давайте спроектируем простую базу данных с одной таблицей. Называться база данных будет Candels. В ней мы будем хранить информацию о свечах. Для описания воспользуемся псевдокодом.

Название базы данных: CandelsBD

Название таблицы: Candels

Структура полей таблицы Candels:

ticker_name — STRING
open_time — INT
open_price — FLOAT
high_ price — FLOAT
low_ price — FLOAT
close_ price — FLOAT
volume — FLOAT

Ничего страшного, если в базе данных будет создана всего лишь одна таблица. Если нет потребности создавать несколько таблиц, не нужно.

3.14. Отношения между таблицами и записями

Как уже упоминалось выше, слово "реляция" значит сообщение, соответственно, суть реляционных баз данных в том, что между таблицами и записями в них можно объявлять связи.

Существует 3 основных типа связей между таблицами:

1. Один ко многим (Belongs to).
2. Один к одному (One to one).

3. Многие ко многим (Many to many).

- Связь "Один ко многим" объявляется тогда, когда одна запись может иметь связь со множеством других. Например, когда один пользователь добавляет множество сигналов.
- Связь "Один к одному" используется, когда каждая запись в таблице может иметь не более одной связи с другой таблицей. Например, когда пользователь создаёт аккаунт и одновременно создаётся профиль, в котором хранятся подробные данные на пользователя.
- Связь "Многие ко многим" объявляется, когда необходимо множество записей связать с другим множеством записей. Чаще всего эта связь используется в интернет-магазинах, когда товары можно одновременно отнести ко множеству категорий. Также встречается при создании тегов к постам в социальных сетях.

В 90% случаев достаточно будет первого типа связей "Один ко многим". В рамках данного руководства мы будем пользоваться только этой связью. Вообще в переводе с английского она звучит как "Многие к одному", но для нашего славянского уха вариант "Один ко многим", на мой взгляд, звучит понятнее.

Для наглядности спроектируем ещё одну базу данных, но уже с двумя таблицами, и объявим между ними связь "Один ко многим".

Название базы данных: UsersAndSignalsBD

Название таблицы: Users

Структура полей таблицы Candels:

ticker_name — STRING
open_time — INT
open_price — FLOAT
high_ price — FLOAT
low_ price — FLOAT
close_ price — FLOAT
volume — FLOAT

Название таблицы: Signals

Структура полей таблицы Signals:

ticker_name — STRING
signal_type — INT
signal — FLOAT
high_ price — FLOAT
low_ price — FLOAT
close_ price — FLOAT
user_id — FLOAT

В таблице Signals мы добавили поле *user_id*. В этом поле будет храниться *id* пользователя, который хочет получить тот или иной сигнал. На языке баз данных это означает, что таблица Signals принадлежит таблице Users и один пользователь может создать много сигналов.

3.15. База данных SQLite

Существуют десятки реляционных баз данных, которые в том или ином виде вы незаметно для себя используете каждый день. Большинство из них составляют промышленные базы данных, которые используются в веб-приложениях. Самыми распространенными являются MYSQL, POSTGRESQL, MSSQIL, SQLITE.

База данных SQLite, которой посвящен этот подраздел, ошибочно считается непригодной для промышленного использования, что совсем не соответствует действительности. Наибольшее распространение она получила благодаря внедрению портативности в большинство известных миру приложений (как десктопных, так и мобильных). Базой SQLite вы пользуетесь ежедневно в своих приложениях на телефоне, независимо от того, какая операционная система у вас установлена.

Особенностью SQLite является отсутствие необходимости в отдельно выделенном сервере (в отличие от тех БД, которые были перечислены выше). База SQLite — это просто один файл, доступ для управления к которому можно получить из любого языка программирования.

Из-за своей простоты использования, легкости и скорости многие разработчики полагают, что ничего, кроме автономных приложений, эта БД под-

держивать не может. На деле один файл базы SQLite может хранить в себе до 278 терабайт данных!

SQLite в состоянии обрабатывать запросы миллионов пользователей. Достигнуто это было в том числе за счёт ограничения длины символов, которые может хранить одна запись в своих полях.

База данных SQLite является приложением с открытым исходным кодом, что позволяет пользоваться ею без ограничений и страха, что уже работающий код перестанет функционировать. При этом разработчики уделяют огромное внимание отказоустойчивости, и для тестирования написано гораздо больше кода, чем для самой SQLite.

3.16. Язык программирования SQL

Для обслуживания реляционных баз данных существует единый стандарт языка структурированных запросов SQL. Каждая СУБД реализовывает этот стандарт по-своему, но в целом различия незначительны. Перенос запросов из одной базы в другую практически незаметен.

SQL (Язык Структурированных Запросов) — это полноценный язык программирования, который поддерживает различные типы данных, позволяет прописывать логику, хранить процедуры (предшественники функций) и многое, многое другое.

Для эффективной работы вам потребуется знать буквально 5% синтаксиса языка SQL, чтобы обслуживать основные потребности. Этого будет достаточно, поскольку большую часть логики по работе с БД мы опишем с помощью Пайтон.

Напомню, основная рутина по работе с записями в таблицах называется CRUD и расшифровывается как Create (Создание), Read (Чтение), Update (Обновление), Delete (Удаление).

Разберем несколько SQL-запросов для решения этих задач. Договоримся на берегу, что все SQL-запросы будут описаны нами заглавными буквами.

3.17. Create — создание записей в таблице

Пример SQL-запроса для создания записи в таблице:

```
INSERT INTO candles (date_time, open, high, low, close, be_trade)
VALUES ("21-25-31 13:31:00", 38.4, 56.4, 65.8, 56.3, 0);
```

> Команда INSERT заставляет базу данных создать запись в таблице.

После команды INSERT идёт команда INTO candles для указания, в какую таблицу добавится запись. После указания имени таблицы в скобках перечисляются поля таблицы, которые будут заполнены. В нашем случае это date_time, open, high, low, close, be_trade.

> Команда VALUES отвечает за то, какими данными поля будут заполнены.

Значения перечислены в круглых скобках:

- В поле date_time попадёт значение 21-25-31 13:31:00.
- В поле open попадёт значение 38.4.
- В поле high попадёт значение 56.4.
- В поле low попадёт значение 65.8.
- В поле close попадёт значение 56.3.
- В поле be_trade попадёт значение 0.

Знак точка с запятой (;) указывает, что запрос завершен. Соответственно, через точку с запятой можно перечислить несколько SQL-запросов.

3.18. Read — чтение данных

Пример SQL-запроса для выборки всех записей в таблице:

```
SELECT * FROM candles;
```

> Команда SELECT отвечает за выборку данных из таблицы.

Знак * указывает, что выбрать нужно все поля записи, а команда FROM указывает, из какой таблицы. Прочитать можно так так: ВЫБРАТЬ ВСЕ ЗАПИСИ И ПОЛЯ ИЗ ТАБЛИЦЫ candles.

```
SELECT date_time, open FROM candles;
```

Этот запрос также вернет все записи, находящиеся в таблице candles, но в ответе будут содержаться только поля *date_time* и *open*.

```
SELECT * FROM candles WHERE be_trade=0;
```

Данный запрос выберет те свечи, у которых поле be_trade равно 0.

```
SELECT * FROM candles WHERE be_trade=0 AND open=38.4;
```

Финальный пример выведет те свечи, где трейда не было и цена равна 38.4.

3.19. Update — обновление данных в уже существующих записях

Пример SQL-запроса для обновления записи в таблице:

```
UPDATE candles SET be_trade=1 WHERE be_trade=0;
```

Команда UPDATE после указания имени таблицы установит в поле be_trade значение 1 тем записям, у которых оно равнялось 0.

После команды SET указывается новое значение, а WHERE укажет, в каких записях данные необходимо обновить.

3.20. Delete — удаление уже созданных записей

Пример SQL-запроса для удаления записи из таблицы:

```
DELETE FROM candles WHERE be_trade=1
```

Дословно запрос звучит так: УДАЛИТЬ ЗАПИСИ ИЗ ТАБЛИЦЫ candles, ГДЕ ПОЛЕ be_trade РАВНЯЕТСЯ 1.

Процедуре удаления записей разработчики уделяют особенное внимание. Всё дело в том, что удаление записей из БД — это наиболее энергозатратная часть с точки зрения потребления ресурсов сервера, на котором база размещена.

Дисковое пространство, на котором хранится база данных, гораздо дешевле, чем оперативная память, с помощью которой данные удаляются. По этой причине большинство приложений никогда НЕ удаляют записи из своих БД. Программисты просто обновляют статус отображения, помечая метками те данные, которые якобы удалились с сервера.

Рекомендую 10 раз думать, прежде чем что-то писать в Интернете, даже в приватных на первый взгляд приложениях.

В следующем подразделе мы познакомимся с SQL-запросом для создания новой таблицы и применим уже знакомые запросы в связке с Пайтон.

3.21. Базовые операции с базой данных на Python (CRUD)

Для работы с базой данных SQLite в Пайтон встроена библиотека sqlite3. Создайте новый файл TestSqlite.py и перепишите следующий код:

```
from sqlite3 import connect # Функция для подключения к БД
from datetime import datetime # Объект для работы со временем

db = connect("candles.db") # Создаём объект БД
prepare_sql_query = db.cursor() # Объект для подготовки запросов
```

В строке `db = connect("candles.db")` мы указываем имя базы данных для подключения. Если БД ещё не создана, то создастся новая.

В строке `prepare_sql_query = db.cursor()` создаём объект для выполнения SQL-запросов.

3.22. Создаём таблицу с помощью Python

Допишите в файл TESTSqlite.py следующий код:

```
# Подготавливаем SQL-запрос
sql = """
CREATE TABLE IF NOT EXISTS candles(
    date_time TEXT,
    open REAL,
    high REAL
    low REAL,
    close REAL,
    be_trade INTEGER)
"""
prepare_sql_query.execute(sql) # Подготавливаем запрос к исполнению
db.commit() # Выполняем подготовленный запрос
```

Если вы запустили исходник и интерпретатор не выдал никаких ошибок, значит база данных и таблица в ней были созданы.

В первой строке из встроенной библиотеки sqlite3 импортируется функция connect(). Эта функция поможет создать объект для подключения к базе данных. Следом импортируется уже знакомый вам объект **datetime** из одноименной библиотеки.

В строке `db = connect("candles.db")` мы пытаемся подключиться к базе данных candles.db. Если базы не существует, то функция connect() создаст её самостоятельно. Строка `prepare_sql_query = db.cursor()` создаст объект для подготовки SQL-запросов к БД.

В переменной `sql` между тройных кавычек хранится многострочный текст. Этот запрос отвечает за создание таблицы **candles**. Ничего страшного, если база данных и таблица будут названы одинаково. Команда CREATE TABLE IF NOT EXISTS НЕ будет пытаться создавать таблицу, если она уже существует.

Также из запроса можно сделать вывод, что в таблице шесть полей.

Ниже представлены примеры уже знакомых вам запросов.

3.23. Создаём запись с помощью Python

Создадим запись с помощью Python.

```
sql = """
INSERT INTO candles (date_time, open, high, low, close, be_trade)
VALUES ("21-25-31 13:31:00", 38.4, 56.4, 65.8, 56.3, 0)"""

prepare_sql_query.execute(sql) # Подготавливаем запрос к исполнению
db.commit() # Выполняем подготовленный запрос
```

Данный запрос просто добавит новую запись в таблицу, столько раз скрипт будет запущен.

3.24. Создаём запись с параметрами с помощью Python

Для того чтобы элегантно передать данные, полученные от пользователя, существует возможность заполнения SQL-запроса списком параметров. Ниже представлен пример такого заполнения.

```
sql = """
INSERT INTO candles (date_time, open, high, low, close, be_trade)
VALUES (?, ?, ?, ?, ?, ?)"""

# Этими параметрами будут заполнены знаки ?
params = [str(datetime.now()), 38.4, 56.4, 65.8, 56.3, 0]

# Подготавливаем запрос к исполнению
prepare_sql_query.execute(sql, params)
db.commit() # Выполняем подготовленный запрос
```

Мы подготовили SQL-запрос, в котором знаками вопроса (?) отметили данные, которые заранее НЕ известны. Затем объявили переменную `params` и перечислили значения, которые в порядке объявления подставятся вместо знаков вопроса.

Строка `prepare_sql_query.execute(sql, params)` подготовит запрос и параметры запроса, а строка `db.commit()` применит запрос к БД.

3.25. Читаем записи в таблице с помощью Python

Ниже представлен самый простой запрос для выборки всех записей и полей из БД:

```
sql = "SELECT * FROM candles" # Запрос для выборки всех данных
prepare_sql_query.execute(sql) # Подготавливаем запрос к исполнению
candles = prepare_sql_query.fetchall() # Делаем выборку и сохраняем

# Перебираем каждую запись и выводим данные на экран
for candle in candles:
    print("Дата:", candle[0])
    print("Открытие:", candle[1])
```

```
    print("Максимум:", candle[2])
    print("Минимум:", candle[3])
    print("Закрытие:", candle[4])
    print("Был трейд?:", candle[5])
    print()
```

Первым делом мы, как всегда, описали SQL-запрос, затем подготовили его для выполнения. Обратите внимание, что когда речь касается чтения данных, нам нет необходимости обращаться к переменной *db*.

В строке `prepare_sql_query.execute(sql)` мы подготовили данные, а в строке `candles = prepare_sql_query.fetchall()` изъяли данные из БД. В ответе вы получите список кортежей. Напомню, что **кортежи** — это НЕ изменяемый список, его можно объявить и изменить только вручную. Значения в кортеже перечисляются в круглых скобках: (1, 2, 3, 4, 5).

Для того чтобы выбрать только первую запись, которая подходит под SQL-запрос, воспользуйтесь строкой result = prepare_sql_query.fetcone().

В предыдущем разделе представлены более сложные запросы для выборки данных. Попробуйте самостоятельно добавить записи с различными данными и применить выборку более сложными запросами.

3.26. Обновляем записи в таблице с помощью Python

Для обновления данных воспользуемся SQL-запросом, уже знакомым по предыдущим примерам:

```
# Устанавливаем записи, у которых поле be_trade было 0, в 1
sql = "UPDATE candles SET be_trade = 1 WHERE be_trade=0"
prepare_sql_query.execute(sql) # Подготавливаем запрос к исполнению
db.commit() # Выполняем подготовленный запрос
```

Вариант обновления данных с подстановкой параметров:

```
sql = "UPDATE candles SET be_trade = ? WHERE be_trade=?"
prepare_sql_query.execute(sql,  [0, 1])
db.commit() # Выполняем подготовленный запрос
```

Здесь мы передали методу execute() два аргумента: первым аргументом стала строка с SQLзапросом, а вторым — список параметров, которыми будут заполнены знаки вопроса.

3.27. Удаление записей из таблиц с помощью Python

В конце концов удалим все записи, у которых поле *be_trade* равняется 1.

```
# Удаляем записи, где be_trade равно 1
sql = "DELETE FROM candles WHERE be_trade=1"
prepare_sql_query.execute(sql) # Подготавливаем запрос
db.commit() # Выполняем запрос
```

Поздравляю! Вы овладели азами программирования баз данных на Пайтон. В следующих разделах мы применим эти знания на практических примерах.

3.28. Многопользовательский скринер открытого интереса

Открытый интерес — это показатель всех неисполненных фьючерсных контрактов (как лонговых, так и шортовых). Если показатель открытого интереса растет вместе с ценой, можно быть уверенным в продолжении тренда.

Пожалуй, это единственный показатель, используемый в индикаторах, который учитывает НЕ прошлые данные, а будущие, ведь заявки ещё НЕ исполнены. Обязательно углубитесь в работу этого показателя, чтобы улучшить свою торговлю.

Техническое задание на скринер открытого интереса будет выглядеть примерно так:

1. Создать таблицу для сохранения chat_id пользователя.
2. Отслеживать запуск бота (появление команды /start).
3. Сохранять в таблицу новых пользователей и дату регистрации.
4. Уведомлять, если пользователь уже зарегистрирован.
5. Следить за рынком и уведомлять юзера об изменении ОИ на 5% и более.

Для реализации задачи нам понадобятся два скрипта. Первый будет следить за сообщениями, поступающими в бот, и создавать записи в таблице пользователей. Второй будет следить за рынком и уведомлять юзеров об изменениях ОИ.

Создайте нового бота по инструкции из самого начала главы. Получите его уникальный ID и свой CHAT_ID для тестов. Следом создайте новую папку для проекта с именем ForUsers. Внутри папки проекта создайте два файла: UserRegistrations.py и OiMonitoring.py. Перепишите в файл UserRegistrations.py код из листинга 3.5.

Листинг 3.5

```
from pybit.unified_trading import HTTP # Класс для подключения к бирже
from requests import get, post # Для отправки POST- и GET-запросов
from datetime import datetime # Объект для работы со временем
from sqlite3 import connect # Функция для подключения к БД
from uuid import uuid4 # Функция для создания уникального id
from time import sleep # Функция для засыпания скрипта

TG_API_URL = "https://api.telegram.org/bot"
TGBOT = "7579403178:AAEU_aJ809TM1IM1OFZYaOLboWvLDrEZa7A"
latest_update_id = None # Переменная для id последнего сообщения
db = connect("open_interest_users.db") # Подключаемся к базе данных
prepare_sql_query = db.cursor() # Создаём объект для подготовки запросов

def send_message(msg="Тестовое сообщение", chat_id=1349083375):
    # Функция для отправки сообщений в Телеграм
    # Принимает два аргумента: текст сообщение и id чата
    # В переменной final_url формируется ссылка для отправки
    # Возвращает ответ от ТГ
    url1 = TG_API_URL+TGBOT
    url2 = "/sendMessage?chat_id="+str(chat_id)+"&text="+str(msg)
    final_url =  url1+ url2 # Конкатенируем ссылку
```

```
    response = post(final_url) # Отправляем и сохраняем результат
    return response # Завершаем функцию

def get_updates(offset=-1):
    # Функция для получения последнего сообщения
    # Принимает один аргумент со значением по умолчанию
    # Отдаёт последнее сообщение отправленному в бот
    result = get(TG_API_URL+TGBOT+"/"+"getUpdates?offset="+str(offset)).json()
    return result["result"][0] # Отдаём последнее сообщение полностью

def create_users_table():
    # Функция для создания таблицы users, если ещё не создана
    # Не принимает аргументов
    # Ничего не отдаёт
    # Переменные из глобальной области видимости
    global db, prepare_sql_query
    sql = """CREATE TABLE IF NOT EXISTS users(
        chat_id INT UNIQUE, # Поле chat_id должно быть уникальным
        created_at TEXT,
        active TEXT)"""

    prepare_sql_query.execute(sql) # Подготавливаем SQL-запрос
    db.commit() # Выполняем запрос
    return # Завершаем функцию

def create_user(chat_id):
    # Функция для создания записи в таблице users=
    # Принимает один аргумент chat_id
    # Ничего не отдаёт

    sql = "INSERT INTO users (chat_id, created_at, active) VALUES (?, ?, ?)"
    params = [int(chat_id), str(datetime.now()), "YES"] # Параметры запроса

    try: # Обертка для обработки ошибок
        prepare_sql_query.execute(sql, params) # Подготавливаем SQL-запрос
        db.commit() # Выполняем запрос
    except Exception: # Если запись уже существует
        send_message(msg="Ты уже зарегистрирован", chat_id=chat_id)
        return # Заканчиваем работу аварийно
    # Если не произошло никаких ошибок
    send_message(msg="Добро пожаловать!", chat_id=chat_id)
    return # Заканчиваем работу

def run():
    # Функция для запуска обработки регистраций
    # Ничего не принимает
    # Ничего не отдаёт
    global latest_update_id  # Переменная из глобальной области видимости
```

```
    sleep(2) # Засыпаем на 2 секунды
    print("latest_update_id", latest_update_id)

    check_update = get_updates() # Получаем последнее сообщение
    # Если id последнего сообщения бота НЕ обновилось
    if latest_update_id == check_update["update_id"]:
        print("Обновлений нет")
    else:
        # Если id последнего сообщения бота обновилось
        print("Получил обновления")
        # Присваиваем id обновления
        latest_update_id = check_update["update_id"]
        chat_id = int(check_update["message"]["chat"]["id"]) # id чата
        print("ID пользователя:", chat_id, "Сообщение:", check_update
        ["message"["text"])

        # Если в сообщении содержится строка "/start"
        if "/start" in check_update["message"]["text"]:
            print("Ты пытаешься зарегистрироваться")
            create_user(chat_id=chat_id) # Создаём нового пользователя
    run() # Рекурсивно перезапускаем функцию
    return # Завершаем текущее выполнение функции

create_users_table() # Создаём таблицу
run() # Запускаем скрипт
```

Первым делом импортируем функции из уже знакомых нам библиотек и объявляем глобальные переменные. Глобальная переменная latest_update_id будет хранить в себе *id* последнего сообщения, отправленного в бот. Функции send_message() и get_updates() были перенесены из прошлых проектов и, надеюсь, хорошо вам понятны.

Функция create_users_table() создаст таблицу для хранения информации о зарегистрированных пользователях. Полей в таблице будет всего три: *chat_id* для хранения чата клиента, *created_at* для хранения даты регистрации и *active* для хранения статуса активности пользователя.

В SQL-запросе обратите внимание на строчку chat_id INT UNIQUE. Эта строка заставит базу данных следить за тем, чтобы при создании записи поле *chat_id* было уникальным. В случае если произойдёт попытка повторного добавления записи с одинаковым *chat_id*, интерпретатор выдаст ошибку. Функцию create_users_table() достаточно вызвать один раз.

Функция create_user() принимает один аргумент *chat_id* и зарегистрирует нового пользователя. SQL-запрос сформирован таким образом, чтобы вместо знаков вопроса (?) подставились параметры, перечисленные в списке *params*. В блоке TRY: мы пытаемся выполнить подготовленный SQL-запрос для регистрации нового клиента. На случай, если пользователь уже существует и интерпретатор вернул ошибку, существует блок CATCH:. В блоке CATCH: мы обрабатываем ошибку и с помощью функции send_message() уведомляем пользователя, что он уже зарегистрирован.

Рекурсивная функция run() создана для отслеживания поступления новых сообщений в вашего бота. Первым делом функция подтягивает переменную `latest_update_id`, в которой хранится *id* последнего сообщения. Функция засыпает на 2 секунды и пытается получить последнее сообщение с помощью вспомогательной функции get_updates(). Далее из полученного сообщения извлекается его *id* и сравнивается со значением, хранящимся в переменной `latest_update_id`. Если id последнего сообщения изменилось, мы присваиваем переменной `latest_update_id` новое значение и сохраняем *id* чата в переменную `chat_id`. Следом анализируем содержимое нового сообщения.

Если в сообщении содержится строка "/start", пытаемся зарегистрировать пользователя с помощью функции create_user(). Как было описано ранее, функция create_user() уведомит пользователя, в случае если он уже зарегистрирован. Во всех остальных случаях мы просто игнорируем всё, что написали пользователи.

Как вы, надеюсь, не забыли, по умолчанию рекурсия работает 1000 раз, затем останавливается. С учетом засыпания функции и ожидания ответа от Телеграм, функция run() проработает примерно 3000 секунд. Соответственно, перезапуск скрипта должен происходить каждые 50 минут. Цифра условная, и вы можете самостоятельно её отрегулировать.

3.29. Запуск отслеживания сигналов открытого интереса на сервере

Для круглосуточной работы скрипта и гарантии отказоустойчивости вам понадобится сервер.

Для примера используем хостинг Jino.ru. Это надежный хостинг с удобной web-консолью для работы без использования SSH.

1. Зайдите на Jino.ru и зарегистрируйте аккаунт.
2. По ссылке *https://cp.jino.ru/vps/new/* выберите самый недорогой VPS-сервер на Ubuntu.
3. Приобретите сервер сразу на несколько месяцев.
4. Дождитесь создания сервера.
5. По ссылке *https://cp.jino.ru/* в разделе VPS найдите созданный сервер и кликните на него.
6. В разделе **Доступ к серверу** нажмите на кнопку **Показать консоль**.
7. Создайте нового пользователя *ghost* с помощью команды:
   `adduser ghost`

   При создании пользователя заполните следующие данные:

```
New password:  ЗДЕСЬ ВВОДИТЕ ПАРОЛЬ
Retype new password: ЗДЕСЬ ПОДТВЕРЖДАЕТЕ
passwd: password updated successfully
Changing the user information for ghost
Enter the new value, or press ENTER for the default
        Full Name []: ЖМЁТЕ ENTER
        Room Number []: ЖМЁТЕ ENTER
        Work Phone []: ЖМЁТЕ ENTER
        Home Phone []: ЖМЁТЕ ENTER
        Other []: ЖМЁТЕ ENTER
Is the information correct? [Y/n] НАЖИМАЕТЕ Y
```

8. Выдайте новому пользователю права доступа. Команда:
   `usermod -aG sudo ghost`
9. Переключитесь на нового пользователя с помощью команды: `su ghost`
10. Введите команду для обновления системы, обновления репозитория, скачивания редактора кода **nano** и установки менеджера пакетов **pip**:

```
sudo apt update && sudo apt upgrade -y && sudo apt install nano
&& sudo apt install pip
```

11. Установите основные зависимости:

```
pip install pybit && pip install requests
```

12. Перейдите в рабочий каталог пользователя с помощью команды:
    `cd /home/ghost`
13. Создайте папку для работы с оповестителем OI: `mkdir openinterest`
14. Перейдите внутрь только что созданной папки: `cd openinterest`
15. Создайте файл с помощью команды: `nano UserRegistrations.py`
16. Зайдите в исходник на локальном компьютере, полностью выделите его и скопируйте.
17. Вернитесь в консоль и вставьте исходник в файл UserRegistrations.py.
18. Нажмите сочетание клавиш CTRL + O, а затем ENTER, чтобы сохранить исходник.
19. Наберите команду `python3 UserRegistrations.py`

Если после запуска вы увидели уже знакомый текст, значит всё в порядке. Для обеспечения круглосуточного мониторинга есть два пути:

1. Просто запустить скрипт в бесконечном цикле и получить утечку памяти.
2. Использовать утилиту CRON для установки расписания запуска.

Первый вариант используют начинающие разработчики, которые, скорее всего, не сталкивались с утечками памяти. Мы пойдем вторым путем.

3.30. Устанавливаем запуск по расписанию с помощью CRON

CRON — это утилита, которая позволяет управлять запуском программ по расписанию. Например, мы хотим, чтобы наша утилита для отслеживания открытого интереса запускалась каждый час целый день.

Для проверки работоспособности CRON в консоли сервера наберите:

```
systemctl enable cron
```

Если вы увидели ответ, значит всё в порядке.

Для управления расписанием необходимо отредактировать всего лишь один файл.

Чтобы открыть файл для установки расписания, в консоли сервера наберите: *crontab -e*.

```
ghost@b1deaebbb:$ crontab -e
```

При первом запуске вас попросят установить редактор по умолчанию. Просто выбираете цифру 1, чтобы установить редактор nano для редактирования CRON-файла.

```
no crontab for ghost - using an empty one

Select an editor.  To change later, run 'select-editor'.
  1. /bin/nano        <---- easiest
  2. /usr/bin/vim.basic
  3. /bin/ed

Choose 1-3 [1]: ЗДЕСЬ ВЫБЕРИТЕ 1 И НАЖМИТЕ ENTER
crontab: installing new crontab
```

CRON устроен очень просто. Сначала мы указываем время или интервал запуска. Затем команду и путь к файлу программы, которую запускаем.

Шаблон и расшифровка последовательности команд:

```
1 2 3 4 5 python3 /home/ghost/openinterest/UserRegistrations.py
```

1: Минуты (0-59).

2: Часы (0-23).

3: День (0-31).

4: Месяц (0-12 [12 == Декабрь]).

5: День недели (0-7 [7 или 0 == sunday]).

Пример. Каждые 5 минут:

```
*/5 * * * * python3 /home/ghost/openinterest/UserRegistrations.py
```

Каждый день в 23:59:

```
59 23 * * *  python3 /home/ghost/openinterest/UserRegistrations.py
```

Каждый день в час ночи:

```
0 1 * * * python3 /home/ghost/openinterest/UserRegistrations.py
```

Каждый месяц первого числа в три часа ночи пятнадцать минут:

```
15 3 1 * * python3 /home/ghost/openinterest/UserRegistrations.py
```

В нашем случае для перезапуска каждый час строка должна выглядеть так:

```
0 * * * * python3 /home/ghost/openinterest/UserRegistrations.py
```

Нажмите сочетание клавиш CTRL + O, а затем Enter, чтобы сохранить cron-файл.

3.31. Скрипт для рассылки сигналов открытого интереса

После того как запущен скрипт для регистрации новых пользователей, пора организовать рассылку сигналов.

Перепишите листинг 3.6 в файл OiMonitoring.py.

Листинг 3.6

```
from pybit.unified_trading import HTTP # Класс для подключения к бирже
from requests import post # Для отправки POST- и GET-запросов
from datetime import datetime # Объект для работы с временем
from sqlite3 import connect # Функция для подключения к БД
from time import sleep # Функция для засыпания скрипта

TGBOT = "7579403178:AAEU_aJ809TMlIM1OFZYaOLboWvLDrEZa7A" # https://
api.telegram.org/bot[BOT_ID]/getUpdates?offset=-1
CHATID = "1341083375" #https://api.telegram.org/bot[BOT_API_KEY]/
sendMessage?chat_id=[CHAT_ID]&text=[MESSAGE_TEXT]
TG_API_URL = "https://api.telegram.org/bot" # Ссылка на TG API
session = HTTP(testnet=False) # Для работы с Bybit

def send_message(msg="Тестовое сообщение", chat_id=1349083375):
    # Функция для отправки сообщений в Телеграм
    # Принимает два аргумента: текст сообщение и id чата
    # В переменной final_url формируется ссылка для отправки
    # Возвращает ответ от ТГ
    url1 = TG_API_URL+TGBOT
    url2 = "/sendMessage?chat_id="+str(chat_id)+"&text="+str(msg)
    final_url =  url1+ url2 # Конкатенируем ссылку
    response = post(final_url) # Отправляем и сохраняем результат
    return response # Завершаем функцию

def get_all_symbols():
   # Функция для получения имен всех монет
   # Ничего не принимает
   # Отдаёт список имен
   print("Пытаюсь получить все символы")
   response = session.get_instruments_info(category="linear")
   only_names = [] # Список для имен
   print("Размер списка с монетами", len(response["result"]["list"]))

   for coin in response["result"]["list"]: # Перебираем ответ
      only_names.append(coin["symbol"]) # Извлекаем только имена
   return only_names # Отдаём список имен

def send_signals_to_all_users(symbol, percent_change):
   # Функция для рассылки сигналов
   # Принимает два аргумента
   # Ничего не отдаёт
   db = connect("open_interest_users.db") # Объект подключения к BD
   prepare_sql_query = db.cursor() # Создаём объект для запросов

   # Описываем SQL запрос
   sql = "SELECT * FROM users"
```

```
    prepare_sql_query.execute(sql) # Подготавливаем SQL-запрос

    all_users = prepare_sql_query.fetchall() # Получаем всех пользователей

    for user in all_users: # Перебираем всех пользователей
        print("ID чата:", user[0])
        msg = symbol + " ОИ вырос  на " + str(percent_change)
        send_message(chat_id=user[0], msg=msg) # Отправляем сообщения
    return # Завершаем функцию, ничего не отдав

def check_open_interest(symbol="BTCUSD"):
    # Функция для отслеживания изменения ОИ
    # Принимает 1 аргумент — символ
    # Ничего не отдаёт
    response = session.get_open_interest(
        category="inverse",
        symbol=symbol,
        intervalTime="5min",
        limit=4
    ) # Получаем 4 последних значения ОИ

    cur_oi = response["result"]["list"][0]["openInterest"]
    ago_15min_oi = response["result"]["list"][3]["openInterest"]
    percent_change = (float(cur_oi)-float(ago_15min_oi)) / float(cur_oi) * 100

    if percent_change >= 5: # Если ОИ вырос на 5% и более
        print("Появился сигнал! Отправляем всем пользователям!")
        send_signals_to_all_users(symbol, percent_change)
    return # Завершаем функцию, ничего не отдав

all_symbols = get_all_symbols() # Получаем имена всех символов
for symbol in all_symbols: # Перебираем символы
    # Проверяем ОИ и делаем рассылки
    check_open_interestopen_interest(symbol)
```

Импорт функций из библиотек и объявление глобальных переменных ничем не отличается от прошлых проектов. Функция send_message() также хорошо вам знакома.

Функция get_all_symbols() просто получает имена всех монет, размещённых на Bybit. Затем по этим именам запускается цикл для отслеживания показателя открытого интереса.

Функция send_signals_to_all_users() разошлёт всем пользователям уведомление, что отрытый интерес одной из монет резко вырос.

В функции check_open_interest() проверяем, насколько вырос ОИ той или иной монеты. Если ОИ вырос на 5 процентов и более, вызовется функция send_signals_to_all_users().

Ниже представлен пример CRON-файла для перезапуска скрипта каждые 5 минут.

```
*/5 * * * * python3 /home/ghost/openinterest/OiMonitoring.py
```

3.32. Многопользовательское слежение за несколькими активами

У нас уже есть простой бот, который позволяет следить за одним активом, и работает только для нас ненаглядных. Дописывать вручную слежение за новыми активами или ценовыми диапазонами — не лучшая идея. Добавим базу данных с одной таблицей, которая будет хранить в себе активы и цены, по достижении которых должен прийти сигнал. В том числе предоставим доступ к своему боту другим пользователям.

Техническое задание будет выглядеть примерно так:

1. Подготавливаем таблицу для хранения сигналов.

Поля таблицы и типы данных:

```
signal_id TEXT UNIQUE # signal_id записи должен быть уникальным
      chat_id TEXT
      created_at TEXT
      active TEXT
      first_price REAL
      second_price REAL
```

2. Обеспечиваем слежение за поступлениями команд.

Основные команды:

Добавление сигнала.
Удаление сигнала.
Просмотр всех активных сигналов.

3. Обеспечиваем отправку уведомлений клиенту.

Основные уведомления:

Добавление сигнала.
Удаление сигнала.
Просмотр всех активных сигналов.
Ошибка при вводе команды.

4. Круглосуточный мониторинг рынка.

Создайте нового бота с помощью @Botfather и получите ключи доступа, как указано в начале главы.

Создайте новую папку для проекта и перепишите в файл PriceSignalsBD.py код из листинга 3.7.

Листинг 3.7

```
from pybit.unified_trading import HTTP # Класс для подключения к бирже
from requests import get, post # Для отправки POST- и GET-запросов
from datetime import datetime # Объект для работы со временем
from sqlite3 import connect # Функция для подключения к БД
from uuid import uuid4 # Функция для создания уникального id
from time import sleep # Функция для засыпания скрипта

TG_API_URL = "https://api.telegram.org/bot"
TGBOT = "7579403178:AAEU_aJ809TMlIM1OFZYaOLboWvLDrEZa7A"
latest_update_id = None # Переменная для id последнего сообщения
session = HTTP(testnet=False) # Создаём объект подключения к бирже
db = connect("signals.db") # Создаём объект подключения БД
prepare_sql_query = db.cursor() # Создаём объект для запросов к БД

def send_message(msg="Тестовое сообщение", chat_id=1349083375):
    # Функция для отправки сообщений в Телеграм
    # Принимает два аргумента: текст сообщение и id чата
    # В переменной final_url формируется ссылка для отправки
    # Возвращает ответ от ТГ
    url1 = TG_API_URL+TGBOT
    url2 = "/sendMessage?chat_id="+str(chat_id)+"&text="+str(msg)
    final_url =  url1+ url2 # Конкатенируем ссылку
    response = post(final_url) # Отправляем и сохраняем результат
    return response # Завершаем функцию

def get_updates(offset="-1")
    # Функция для получения последнего сообщения
```

```
    # Принимает один аргумент со значением по умолчанию
    # Отдаёт последнее сообщение отправленному в бот
    # Делаем GET запрос и сохраняем ответ
    url = TG_API_URL+TGBOT+"/getUpdates?offset="+offset
    result = get(url).json() # Сохраняем ответ в JSON
    try:
        # Пытемся отдать последнее сообщение
        return.result["result"][0]
    except:
        return None # Завершаем функцию

def create_signals_table():
    # Функция создания таблицы для хранения сигналов
    # Не принимает аргументов
    # Не отдаёт результат
    # Переменные из глобальной области видимости
    global db, prepare_sql_query

    # SQL-запрос для создания таблицы, если ее нет
    sql = """CREATE TABLE IF NOT EXISTS signals(
        signal_id TEXT UNIQUE, # signal_id записи должен быть уникальным
        chat_id TEXT,
        created_at TEXT,
        active TEXT,
        first_price REAL,
        second_price REAL)"""

    prepare_sql_query.execute(sql) # Подготавливаем SQL-запрос
    db.commit() # Выполняем SQL-запрос
    return # Завершаем функцию


def create_signal(chat_id, active, first_price, second_price):
    # Функция для создания записи в таблицу signals
    # Принимает 4 аргумента
    # Ничего не отдаёт
    # SQL-запрос с параметрами
    sql = """INSERT INTO signals (signal_id,
        chat_id,
        created_at,
        active,
        first_price,
        second_price)
    VALUES (?, ?, ?, ?, ?, ?)"""

    # Параметры запроса
    params = [str(uuid4()), int(chat_id), str(datetime.now()),
        active.upper(), first_price, second_price]
```

```
    prepare_sql_query.execute(sql, params) # Подготавливаем SQL-запрос
    db.commit() # Выполняем SQL-запрос

    # Уведомляем юзера, создавшего сигнал
    send_message(msg="Сигнал добавлен", chat_id=chat_id)
    print("Сигнал создан")
    return # Завершаем функцию

def show_all_signals(chat_id):
    # Функция для получения всех активных сигналов юзера
    # Принимает 1 аргумент
    # Ничего не отдаёт

    # SQL-запрос для выборки сигналов
    sql = "SELECT * FROM signals WHERE chat_id = ?"

    # Подготавливаем SQL-запрос с параметрами
    prepare_sql_query.execute(sql, [chat_id])

    # Выбираем все записи пользователя
    all_signals = prepare_sql_query.fetchall()

    # Перебираем и по очереди отправляем каждую запись
    for signal in all_signals:
        print("ID сигнала:", signal[0])
        print("ID чата:", signal[1])
        print("Дата создания:", signal[2])
        print("Актив:", signal[3])
        print("Первая цена:", signal[4])
        print("Вторая цена:", signal[5])

        msg = signal[0]+"\n"+signal[3]+" "+str(signal[4])+" "+str(signal[5])

        message_data = {  # Ещё один способ формирования сообщений
            "chat_id": chat_id, # Куда отправляем сообщение
            "text": msg, # Текст сообщения
            "parse_mode": "HTML" # Тип форматирования
        }

        # Отправляем POST-запрос в Телеграм с сообщением
        post(TG_API_URL+TGBOT+"/sendMessage", data=message_data)
    return # Завершаем функцию show_all_signals()

def delete_signal(signal_id, chat_id):
    # Функция для удаления сигнала
    # Принимает 2 аргумента
    # Ничего не отдаёт

    # SQL запрос для для удаления сигнала по id
```

```
    sql = "DELETE FROM signals WHERE signal_id=?"

    # Подготавливаем запрос с параметрами
    prepare_sql_query.execute(sql, [signal_id])
    db.commit() # Выполняем запрос
    send_message(msg="Сигнал удален!", chat_id=chat_id)
    return # Завершаем функцию delete_signal()
def run():
    # Функция для обработки команд от клиентов
    # Ничего не принимает
    # Ничего не отдаёт
    global latest_update_id # Глобальная переменная
    sleep(2) # Засыпаем на 2 секунды
    print("latest_update_id", latest_update_id)
    check_update = get_updates() # Получаем последнее сообщение

    # Если id последнего сообщения бота НЕ обновилось
    if latest_update_id == check_update["update_id"]:
        print("Обновлений нет")
    else: # Если id последнего сообщения бота обновилось
        print("Получил обновления")

        # Присваиваем id обновления
        latest_update_id = check_update["update_id"]

        # Получаем id чата
        chat_id = int(check_update["message"]["chat"]["id"])
        print("ID пользователя:", chat_id, "Сообщение:",
              check_update["message"]["text"])

        # Серия для проверок команд пришедших от пользователя
        if "/create" in check_update["message"]["text"]:
            # Если содержится команда /create
            print("Ты хочешь создать сигнал")

            # Разделяем строку на список
            # 1 - команда, 2 - актив, 3 - первая цифра, 4 - вторая цифра

            string_to_list = check_update["message"]["text"].split()

            if len(string_to_list) != 4:
                # Если размер списка меньше 4 или больше 4
                # Уведомляем юзера, что формат заполнен не верно
                send_message(msg="Не верный формат", chat_id=chat_id)
                run()  # Перезапускаем функцию
                return # Не даём функции продолжить текущую работу

            # Создаём сигнал если с размером всё в порядке.
            create_signal(chat_id=chat_id,
```

```
                active=string_to_list[1],
                first_price=float(string_to_list[2]),
                second_price=float(string_to_list[3])
            )
        elif "/all" in check_update["message"]["text"]:
            # Если в сообщении содержится команда /all
            print("Ты хочешь увидеть все сигналы")
            # Отправляем все активные сигналы
            show_all_signals(chat_id=chat_id)
        elif "/delete" in check_update["message"]["text"]:
            # Если в сообщении содержится команда /delete
            print("Ты хочешь удалить сигнал")

            # Разбиваем строку на список с помощью функции split()
            # В сообщении должно быть 2 элемента команда и ID
            string_to_list = check_update["message"]["text"].split()

            if len(string_to_list) != 2:
                # Если в списке НЕ 2 элемента
                send_message(msg="Не верный формат", chat_id=chat_id)
                run() # Рекурсивно перезапускаем функцию
                return # Не даём функции продолжить работу
            # Удаляем сигнал из базы если формат сообщения верный
            delete_signal(id=string_to_list[1], chat_id=chat_id)
        else:
            # На случай если юзер ввёл неизвестную команду
            print("Неизвестная команда")
            send_message(msg="Не понимаю о чём ты!",
            chat_id=chat_id)
    run() # Перезапускаем функцию
    return # Завершаем текущее выполнение функции

run() # Запускаем функцию
```

Первым делом импортируем функции из уже знакомых вам библиотек и объявляем глобальные переменные. Новой для вас станет библиотека **uuid**, из которой импортируется функция **uuid4**. Эта функция обеспечит уникальный идентификатор каждой записи в таблице сигналов.

Переменная *latest_update_id* будет хранить в себе *id* последнего отправленного в бот сообщения. Функции send_message() и get_updates() были перенесены из прошлых проектов.

Функция create_signals_table() создаст таблицу для хранения информации о сигналах и *id* чатов пользователей. В SQL-запросе обратите внимание

на строчку `signal_id TEXT UNIQUE`. Эта строка заставит базу данных следить за тем, чтобы при создании записи поле *signal_id* было уникальным.

Функция create_signal() принимает четыре аргумента и регистрирует сигнал пользователя в системе. SQL-запрос сформирован таким образом, чтобы вместо знаков вопроса (?) подставились параметры, перечисленные в списке params. Обратите внимание, что первым элементом списка параметров является элемент `str(uuid4())`. Это вызов функции для генерации уникального айдишника, и она возвращает объект типа **uuid4**, который мы приводим к строке. Если сигнал без ошибок добавлен, информируем об успехе клиента.

Функция show_all_signals() принимает в качестве аргумента chat_id клиента. Затем ныряет в базу данных и выуживает все сигналы пользователя. По полученному от БД списку запускается цикл. В словарь message_data собираются данные на сигнал и отправляются клиенту в качестве аргумента функции post().

Функция delete_signal() удалит сигнал, переданный в переменную аргумента signal_id, и отправит сообщение с подтверждением об удалении пользователю, переданному в переменной аргумента *chat_id*.

Рекурсивная функция run() создана для отслеживания и обработки новых сообщений в Вашем боте. Первым делом функция подтягивает переменную `latest_update_id`, в которой будет хранить *id* последнего сообщения.

Функция run() засыпает на 2 секунды и пытается получить последнее сообщение с помощью вспомогательной функции get_updates(). Далее из полученного сообщения извлекается *id* и сравнивается со значением, хранящимся в переменной `latest_update_id`.

Если *id* последнего сообщения изменилось, мы присваиваем переменной `latest_update_id` новое значение. В следующей строке сохраняем *id* чата в переменную `chat_id`.

Затем анализируем содержимое свежего сообщения. Если в сообщении содержится строка "/create", валидируем введенные пользователем данные и пытаемся создать сигнал с функцией create_signal().

Если в сообщении содержится строка "/all", просто вызываем функцию show_all_signals() и передаём ей chat_id клиента, сделавшего запрос. Если

в сообщении содержится строка "/delete", валидируем данные, введенные пользователем, и удаляем сигнал функцией delete_signal().

Основных этапов валидации шесть:

1. Проверить, не пустая ли строка.
2. Разделить строку на 3 части.
3. Проверить, правильно ли введена валютная пара.
4. Проверить начало диапазона и его тип.
5. Проверить окончание диапазона и его тип.
6. Уведомить пользователя в случае ошибки.

Попробуйте реализовать недостающие этапы валидации самостоятельно.

В нашем случае для перезапуска каждый час строка должна выглядеть так:

```
0 * * * * python3 /home/ghost/pricesignals/PriceSignalsBD.py
```

3.33. Мониторинг цен и отправка сигналов пользователям

После того как реализован сервис для отслеживания сообщений, приходящих от пользователей, реализуем мониторинг и рассылку уведомлений. Создайте в текущем проекте новый файл MonitoringUsersSignals.py и перепишите следующий код:

Листинг 3.8

```
from pybit.unified_trading import HTTP # Класс для подключения к бирже
from requests import post # Для отправки POST- и GET-запросов
from datetime import datetime # Объект для работы со временем
from sqlite3 import connect # Функция для подключения к БД
from time import sleep # Функция для засыпания скрипта

TG_API_URL = "https://api.telegram.org/bot"
TGBOT = "7579403178:AAEU_aJ809TM1IM1OFZYaOLboWvLDrEZa7A"
```

```
session = HTTP(testnet=False) # Создаём объект подключения к бирже
db = connect("signals.db") # Создаём объект подключения БД
prepare_sql_query = db.cursor() # Создаём объект для запросов к БД

def send_message(msg="Тестовое сообщение", chat_id=1349083375):
    # Функция для отправки сообщений в Телеграм
    # Принимает два аргумента: текст сообщение и id чата
    # В переменной final_url формируется ссылка для отправки
    # Возвращает ответ от ТГ
    url1 = TG_API_URL+TGBOT
    url2 = "/sendMessage?chat_id="+str(chat_id)+"&text="+str(msg)
    final_url =  url1+ url2 # Конкатенируем ссылку
    response = post(final_url) # Отправляем и сохраняем результат
    return response # Завершаем функцию

def get_symbol_current_price(symbol="BTCUSDT"):
    # Функция для получения текущей цены монеты
    # Принимает 1 аргумент со значением по умолчанию
    # Отдаёт текущую цену
    # Получаем информацию о текущем состоянии монеты
    result = session.get_tickers(
        category="linear",
        symbol=symbol)
    # Извлекаем из ответа текущую цену
    current_price = result["result"]["list"][0]["lastPrice"]
    print("Текущая цена", current_price)
    return float(current_price) # Отдаём результат

def monitoring_all_signals():
    # Функция для мониторинга всех сигналов всех юзеров
    # Не принимает аргументов
    # Не отдаёт результат
    sql = "SELECT * FROM signals" # SQL запрос для выборки сигналов
    prepare_sql_query.execute(sql) # Подготавливаем запрос
    all_signals = prepare_sql_query.fetchall() # Извлекаем все сигналы

    for signal in all_signals: # Перебираем все сигналы
        print("ID сигнала:", signal[0])
        print("ID чата:", signal[1])
        print("Дата создания:", signal[2])
        print("Актив:", signal[3])
        print("Первая цена:", signal[4])
        print("Вторая цена:", signal[5])

        try: # Запускаем в безопасном режиме
            cur_price = get_symbol_cur_price(signal[3]) # Текущая цена
            print(cur_price, signal[4], signal[5])
            # Если цена находится в указанном диапазоне,
            # отправляем сообщение
```

```
            if cur_price >= signal[4] and cur_price <= signal[5]:
                msg = "Актив: "+signal[3]+" на цене: "+str(cur_price)
                send_message(msg=msg, chat_id=signal[1])
        except Exception:
            # Обрабатываем ошибку и не даём скрипту упасть
            print("Вероятна ошибка в символе")
    return # Завершаем текущее исполение функции

monitoring_all_signals() # Запускаем мониторинг
```

Импорт функций из библиотек и объявление глобальных переменных ничем не отличается от прошлых проектов. Функции send_message() и get_symbol_current_price() также не требуют дополнительных объяснений.

Фуннкция monitoring_all_signals() ныряет в базу данных, выбирает все записи из таблицы signals и перебирает их в цикле FOR:. В блоке TRY: мы получаем текущую цену монеты, проверяем на соответствие диапазону и уведомляем пользователя. В блоке CATCH: идёт обработка возможных ошибок.

Ниже пример содержимого CRON-файла для перезапуска скрипта каждые 30 минут:

```
*/30 * * * * python3 /home/ghost/pricesignals/MonitoringUsersSignals.py
```

Все команды, которые отслеживает ваш бот, можно добавить в виде *меню*. Как добавить команды в меню Телеграм бота, будет описано далее.

3.34. Продвинутый и быстрый скринер резких всплесков цены

В ближайших подглавах мы реализуем полноценный памповый скринер у которого НЕ будет изъянов, связанных с медленной аналитикой исторических данных.

Напомню, что одной из лучших стратегий выявления манипуляций на крипторынке, является отслеживание цены актива за последние 15 минут. Если цена за этот период выросла на 5 процентов и более, вероятно, что в монету заносятся деньги. Крупный капитал закупается именно на 15 минутных свечах (вне зависимости от рынка).

Идеальным сочетанием является момент, когда произошёл одновременный памп цены и открытого интереса. При таком раскладе с большой долей вероятности можно забрать 1% чистого движения и уйти с рынка.

Вы легко сможете реализовать такой скринер самостоятельно по исходникам из этой главы. Мало того, если приложите усилия, то сможете автоматизировать входы в рынок по этой стратегии. Хорошенько вникните в последний раздел этой книги, напишите ТЗ, а затем реализуйте.

Для реализации пампового скринера понадобится 3 файла. Можете заготовить их заранее.

1. **PriceScaner.py** – один раз в 5 секунд обращается к рынку и вытаскивает текущие цены всех монет. Собирает историю цен за последние 15 минут. Создаёт файл с сигналами.

2. **RegisterUser.py** – просто регистрирует пользователя в боте.

3. **SendSignals.py** – рассылает сигналы пользователям.

Создайте нового бота Телеграм и получите от него ключи.

3.35. Быстрая аналитика цен и сохранение истории в JSON-файл

В листинге 3.9 представлены функции для файла PriceScaner.py. После листинга есть краткое описание для каждой функции.

Листинг 3.9

```
from pybit.unified_trading import HTTP # Объект для работы с Байбит
from datetime import datetime # Объект для работы с временем
from json import dump, load # Функции для работы с JSON и обратно
from time import sleep # Спим когда захотим

session = HTTP(testnet=False) # Объект для работы с Bybit

def get_all_symbols_names():
```

```
    # Функция для получения имён всех монет размещённых на Bybit
    print("Пытаюсь получить имена всех тикеров")
    # Общая инфо по всем монетам
    response = session.get_instruments_info(category="linear")

    only_names = [] # Подготовка списка для имён монета
    print("Размер массива с монетами", len(response["result"]["list"]))

    # Перебираем результат ответа
    for coin in response["result"]["list"]:
        only_names.append(coin["symbol"]) # Извлекаем только имя монеты

    print("Получилось")
    return only_names # Отдаём список с именами

def get_all_tickers():
    # Функция получит и отдаст текущие цены монет
    # Получаем текущее состояние монет
    response = session.get_tickers(category="linear")
    return response["result"]["list"] # Извлекаем список из запроса

def prepare_symbols_for_json_file():
    # Функция для подготовки JSON файла с историей
    # В него собирается история последних 15 минут
    # Ничего не отдаёт и не принимает. Формат подготовленных данных:
    # final_json = {"BTCUSDT": [], "LTCUSDT": [], "XMRUSDT": []}
    final_json = {} # Подготавливаем словарь для JSON файла
    all_symbols_names = get_all_symbols_names() # Берём имена символов

    for symbol in all_symbols_names: # Перебираем имена
        # Создаём в словаре список. Имя монеты это ключ
        final_json[symbol] = []

    # Открываем файл в безопасном режиме
    with open("history.json", "w") as file:
        dump(final_json, file) # Записываем словарь в JSON формате
    return # Завершаем функцию ничего не отдав

def start():
    # Функция запуска сканера цен и сигналов
    # Ничего не принимает и не отдаёт
    # Работает со сбором истории за последние 15 минут
    # Работает со сбором сигналов если цена изменилась на 5% и более
    # История и сигналы хранятся в виде JSON файлов
    signals = [] # Подготавливаем массив для сигналов

    now = datetime.now()
    print("Запустил функцию", now)
```

```
try: # Критический кусок кода
    # Обращаемся к фалу истории на чтение
    file = open("history.json", "r")
    history_data = load(file) # Вытаскиваем данные в виде словаря
    file.close() # Закрываем файл
except Exception as e: # Если упало
    print(e) # Выводим ошибку на экран
    # Перезапускам подготовку JSON файла с историей
    prepare_symbols_for_json_file()
    return

symbols = get_all_tickers() # Получаем данные на монеты

for symbol in symbols: # Перебираем ответ с данными на монеты
    print(symbol["symbol"]) # Извлекаем имя монеты
    print(symbol["lastPrice"]) # Извлекаем текущую цену
    print()
    try: # Критический кусок кода
        # Измеряем сколько данных накопилось в списке монеты
        if len(history_data[symbol["symbol"]]) > 180:
            # Если больше 180 элементов
            # Удаляем первый элемент
            history_data[symbol["symbol"]].pop(0)

        # В конец списка с ценами на монету, добавляем текущую цену
        history_data[symbol["symbol"]].append((float(symbol["lastPrice"])))

        # Получаем последнюю цену
        cur_price = history_data[symbol["symbol"]][-1]

        # Цена 15 минут назад
        ago_15min_price = history_data[symbol["symbol"]][0]

        # Измеряем изменение цены
        price_change = (cur_price-float(ago_15min_price)) / cur_price * 100

        # Подготавливаем словарь с именем символа и ценой актива
        symbol_and_percent_change = {"symbol": symbol["symbol"],
             "price_change": price_change}

        if price_change >= 5: # Если цена выросла на 5% и более
            # Добавляем словарь в список с сигналами
            signals.append(symbol_and_percent_change)
    except Exception as e: # Отлавливаем ошибку
        print(e) # Просто выводим её на экран и едем дальше

sleep(1) # Засыпаем на 1 секунду перед записью файлов
```

```
    print("Записываю историю")
    try: # Критический кусок кода
        file = open("history.json", "w") # Открываем файл истории на запись
        dump(history_data, file) # Добавляем исторические данные в JSON
        file.close() # Закрываем файл
    except Exception as e: # Если упало
        print(e) # Выводим ошибку
        # Вызываем функцию для подготовки истории
        prepare_symbols_for_json_file()
        return # Завершаем функцию start()

    print("Записываю сигналы")
    try: # Критический кусок кода
        file = open("signals.json", "w") # Открываем файл сигналов на запись
        dump(signals, file) # Добавляем сигналы в JSON формате
        file.close() # Закрываем файл
    except Exception as e: # Если упало
        print(e) # Выводим ошибку
        return # Завершаем функцию start()

    sleep(5) # Засыпаем на 5 секунд перед следующим сбором
    return # Завершаем текущее выполнение функции start()

prepare_symbols_for_json_file() # Подготавливаем JSON файл
for i in range(1, 500): # 500 раз хватит примерно на 1 час
    start() # Запускаем сбор истории и сигналов
```

- Функция get_all_symbols_names() получает имена всех тикеров, доступных на бирже. Они понадобятся при создании JSON-файла и хранения историй цены за последние 15 минут.
- Функция get_all_tickers() получит информацию на текущие цены всех монет одновременно. Т.е. нам не придётся делать запрос цены на каждую монету отдельно.
- Функция prepare_symbols_for_json_file() подготовит JSON файл для хранения истории. История будет собираться и храниться на каждую монету, размещённую на бирже Bybit. Имя монеты будет выступать в качестве ключа к списку с историей цен.
- Функция start() запускает сканер и первым делом обращается к JSON-файлу с историей. Данные из JSON-файла преобразуются в обычный словарь. Затем с помощью get_all_tickers() мы получаем текущие данные на все монеты и запускаем по ним перебор с помощью цикла FOR:.

Если в списке истории монеты накопилось больше 180 элементов (при перезапуске каждые 5 секунд это примерно 15 минут) мы удаляем первый элемент.

Затем сравниваем текущее значение цены с той, что была 15 минут назад. Если цена выросла на 5 процентов и более, добавляем в список с сигналами словарь с названием монеты и процентом изменения цены.

Затем записываем актуализированные исторические данные и сигналы в уже существующие JSON-файлы. Не забываем заснуть на 5 секунд перед следующим сбором.

В конце запускаем функцию start() в цикл FOR: 500 раз. Этого хватит на 1 час. Обратите внимание, что история будет сбрасываться каждый час: это позволит раньше других отслеживать пампы.

3.36. Регистрация пользователей пампового скринера

В листинге 3.10 представлен код файла RegisterUsers.py.

Листинг 3.10

```
from pybit.unified_trading import HTTP # Класс для подключения к бирже
from requests import get, post # Для отправки POST и GET запросов
from datetime import datetime # Объект для работы с временем
from sqlite3 import connect # Функция для подключения к БД
from uuid import uuid4 # Функция для создания уникального id
from time import sleep # Функция для засыпания скрипта

TG_API_URL = "https://api.telegram.org/bot"
TGBOT = "7579403178:AAEU_aJ809TMlIM1OFZYaOLboWvLDrEZa7A"
latest_update_id = None # Переменная для id последнего сообщения
db = connect("pump_users.db") # Подключаемся к базе данных
prepare_sql_query = db.cursor() # Создаём объект для подготовки запросов

def send_message(msg="Тестовое сообщение", chat_id=1349083375):
    # Функция для отправки сообщений в Телеграм
    # Принимает два аргумента: текст сообщение и id чата
    # В переменной final_url формируется ссылка для отправки
    # Возвращает ответ от ТГ
    url1 = TG_API_URL+TGBOT
    url2 = "/sendMessage?chat_id="+str(chat_id)+"&text="+str(msg)
```

```
    final_url =  url1+ url2 # Конкатенируем ссылку
    response = post(final_url) # Отправляем и сохраняем результат
    return response # Завершаем функцию

def get_updates(offset="-1")
    # Функция для получения последнего сообщения
    # Принимает один аргумент со значением по умолчанию
    # Отдаёт последнее сообщение отправленному в бот
    # Делаем GET запрос и сохраняем ответ
    url = TG_API_URL+TGBOT+"/getUpdates?offset="+offset
    result = get(url).json() # Сохраняем ответ в JSON
    try:
        # Пытемся отдать последнее сообщение
        return result["result"][0]
    except:
        return None # Завершаем функцию

def create_users_table():
    # Функция для создания таблицы users если ещё не создана
    # Не принимает аргументов
    # Ничего не отдаёт

    # Подтягиваем переменные из глобальной области видимости
    global db, prepare_sql_query

    sql = """CREATE TABLE IF NOT EXISTS users(
    chat_id INT UNIQUE, # Поле chat_id должно быть уникальным
    created_at TEXT,
    active TEXT)"""

    prepare_sql_query.execute(sql) # Подготавливаем SQL запрос
    db.commit() # Выполняем запрос
    return

def create_user(chat_id):
    # Функция для создания записи в таблице users=
    # Принимает один аргумент chat_id
    # Ничего не отдаёт

    sql = "INSERT INTO users (chat_id, created_at, active) VALUES (?, ?, ?)"
    params = [int(chat_id), str(datetime.now()), "YES"] # Параметры

    try: # Обёртка для обработки ошибок
        # Подготавливаем SQL запрос
        prepare_sql_query.execute(sql, params)
        db.commit() # Выполняем запрос
    except Exception: # Если запись уже существует
        send_message(msg="Ты уже зарегестрирован", chat_id=chat_id)
        return # Заканчиваем работу
```

```
    # Если не произошло ни каких ошибок
    send_message(msg="Добро пожаловать! Теперь ты будешь получать пампы!",
        chat_id=chat_id)
    return # Заканчиваем работу по добавлению пользователя в БД

def run():
    # Функция для запуска обработки регистраций
    # Ничего не принимает и не отдаёт

    global latest_update_id # Глобальная переменная
    sleep(2) # Засыпаем на 2 секунды
    print("latest_update_id", latest_update_id)
    check_update = get_updates() # Получаем последнее сообщение

    # Если id последнего сообщения бота НЕ обновилось
    if latest_update_id == check_update["update_id"]:
        print("Обновлений нет")
    else:
        # Если id последнего сообщения бота обновилось
        print("Получил обновления")
        latest_update_id = check_update["update_id"] # id обновления

        # Получаем id чата
        chat_id = int(check_update["message"]["chat"]["id"])
        print("ID пользователя:", chat_id,
              "Сообщение:", check_update["message"]["text"])

        # Если в сообщении содержится строка "/start"
        if "/start" in check_update["message"]["text"]:
            print("Ты пытаешься зарегестрироваться")
            create_user(chat_id=chat_id) # Создаём нового пользователя
    run() # Рекурсивно перезапускаем функцию
    return # Завершаем текущее выполнение

create_users_table() # Создаём таблицу пользователей
run() # Запуск
```

Код практически не отличается от того, что мы описывали при создании скринера ОИ.

3.37. Рассылка памповых сигналов пользователям

В листинге 3.11 представлен код файла SendSignals.py.

Листинг 3.11

```
from pybit.unified_trading import HTTP # Класс для подключения к бирже
from requests import post # Для отправки POST и GET запросов
from datetime import datetime # Объект для работы со временем
from sqlite3 import connect # Функция для подключения к БД
from time import sleep # Функция для засыпания скрипта

TGBOT = "7579403178:AAEU_456aOLboWvLDrEZa7A" # Хэш робота
TG_API_URL = "https://api.telegram.org/bot" # Ссылка на TG API
session = HTTP(testnet=False)

def send_message(msg="Тестовое сообщение", chat_id=1349083375):
    # Функция для отправки сообщений в Телеграм
    # Принимает два аргумента: текст сообщение и id чата
    # В переменной final_url формируется ссылка для отправки
    # Возвращает ответ от ТГ
    url1 = TG_API_URL+TGBOT
    url2 = "/sendMessage?chat_id="+str(chat_id)+"&text="+str(msg)
    final_url =  url1+ url2 # Конкатенируем ссылку
    response = post(final_url) # Отправляем и сохраняем результат
    return response # Завершаем функцию

def send_signals_to_all_users(symbol, percent_change):
   # Функция для рассылки сигналов
   # Принимает два аргумента
   # Ничего не отдаёт
   db = connect("pump_users.db") #
   prepare_sql_query = db.cursor() # объект для подготовки запросов

   # Описываем SQL запрос
   sql = "SELECT * FROM users"
   prepare_sql_query.execute(sql) # Подготавливаем SQL-запрос

   all_users = prepare_sql_query.fetchall() # Получаем всех пользователей

   for user in all_users: # Перебираем всех пользователей
      print("ID чата:", user[0])
      msg = symbol + " цена резко выросла на " + str(percent_change)
      send_message(chat_id=user[0], msg=msg) # Отправляем сообщения
   return # Завершаем рассылку

def check_pump_signals():
   # Функция для рассылки сигналов
   # Ничего не отдаёт и не принимает

    try: # Критический кусок кода
```

```
        # Обращаемся к файлу сигналов на чтение
        file = open("signals.json", "r")
        signals = load(file) # Вытаскиваем из JSON данные в виде списка
        file.close() # Закрываем файл
    except Exception as e: # Если упало
        print(e) # Выводим ошибку на экран
        check_pump_signals() # Перезапускаем функцию
        return # Завершаем текущее выполнение

    if len(signals) != 0: # Если сигналы есть
        for signal in signals: # Делаем рассылку по пользователям
            send_signals_to_all_users(signal["symbol"],
            signal["percent_change"])
    return # Завершаем функцию ничего не отдав

# 3 запуска в минуту
for a in range(1, 180): # 180 запусков за час
    check_pump_signals()
    sleep(20)
```

- Функции send_message() и send_signals_to_all_users() знакомы Вам по предыдущим исходникам.
- Функция check_pump_signals() просто заходит в JSON-файл с сигналами и если список НЕ пустой, отправляет информацию о пампе всем пользователям.

Работать скрипт будет 1 час, затем перезапустится. Соответственно в цикле FOR: установлено примерное количество перезапусков, которое успеет сделать скрипт за это время.

3.38. Настройка перезапуска CRON для пампового скринера

Настало время перезапустить все скрипты скринера с помощью CRON. В домашней директории своего сервера с помощью команды *mkdir pumper* создайте папку для файлов скринера.

С помощью редактора **nano** перенесите код из каждого файла. Затем наберите команду *cron -e* и впишите в CRON файл следующий код.

```
0 * * * * python3 /home/ghost/pumper/PriceScaner.py >> /home/ghost/
pumper/PS.txt

0 * * * * python3 /home/ghost/pumper/RegisterUsers.py >> /home/ghost/
pumper/RU.txt

0 * * * * python3 /home/ghost/pumper/SendSignals.py >> /home/ghost/
pumper/SS.txt
```

Все скрипты будут перезапущены в начале часа. После знаков >> указаны файлы, в которые будут писаться логи скрипта. Поможет при отладке.

Готово! Ваш продвинутый скринер готов!

В качестве практической задачи самостоятельно реализуйте ссылку на монету в сообщении и научитесь вести учёт количества сигналов, пришедших на монету в течении дня.

3.39. Биржа опционов в Телеграме

У нас уже есть потрясающий прототип биржи опционов. Почему бы нам не сделать из него Телеграм-приложение? Сказано — сделано!

Договоримся, что этот пример будет предельно простым и все неупомянутые функции вы реализуете самостоятельно. Биржа будет принимать только лонговые сделки, а шортовые вы допишете самостоятельно. Все референсы у вас есть. Как всегда, при создании многопользовательских ботов реализуем два сервиса, первый для обработки пунктов меню, второй — для исполнения сделок.

3.39.1. Сервис для обработки пунктов меню

1. Регистрация.

   Просто приветствуем пользователя.

2. Создать сделку.

   Сделка просто создаётся как есть без времени экспирации.

   Делаем расчёт сделок 1 раз в минуту.

3. Показать историю сделок.

   Собираем сделку и исход.

Проектируем базу данных.

Название базы данных: bitcoin_options.db

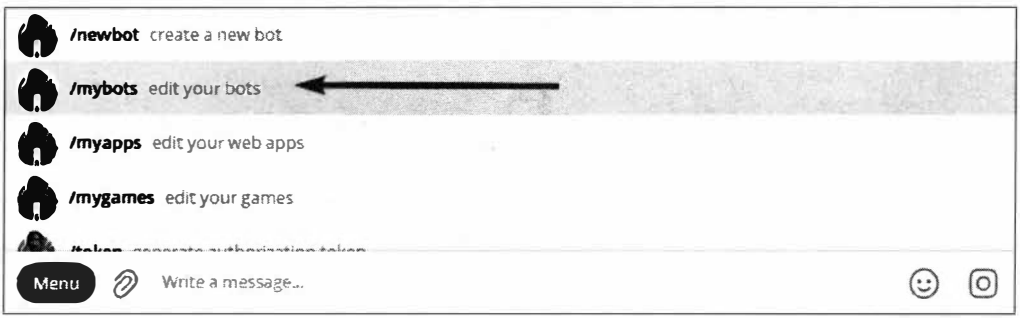
Таблица Orders:

```
id TEXT UNIQUE
chat_id INT
created_at TEXT
direction INT
sum REAL
price_at_start REAL
price_at_end REAL
executed INT DEFAULT 0
order_win_status INT DEFAULT 0
profit REAL
```

Создайте нового бота Телеграм, сохраните значения ключа API и свой CHAT_ID.

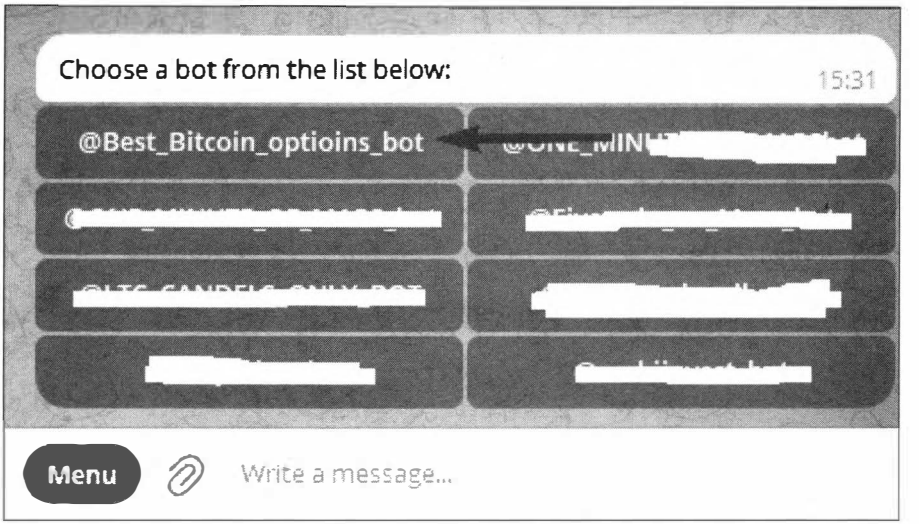
Для создания меню следуйте следующей инструкции:

1. После создания бота вернитесь назад в @BotFather и в меню найдите / mybots.



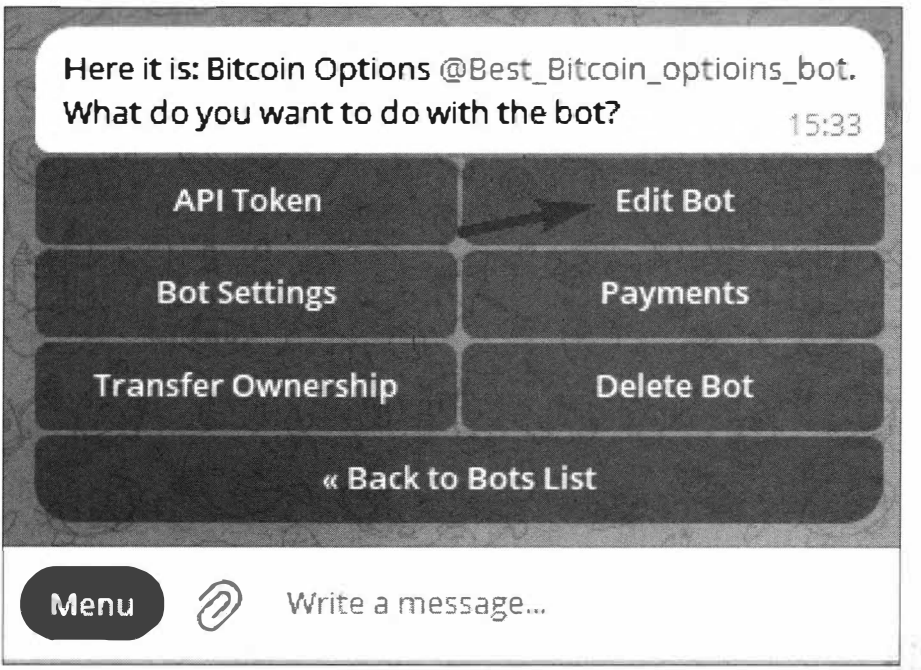
Изображение 3.15.

2. В появившемся списке выберите только что созданного бота.



Изображение 3.16.

3. Выберите пункт EDIT BOT.



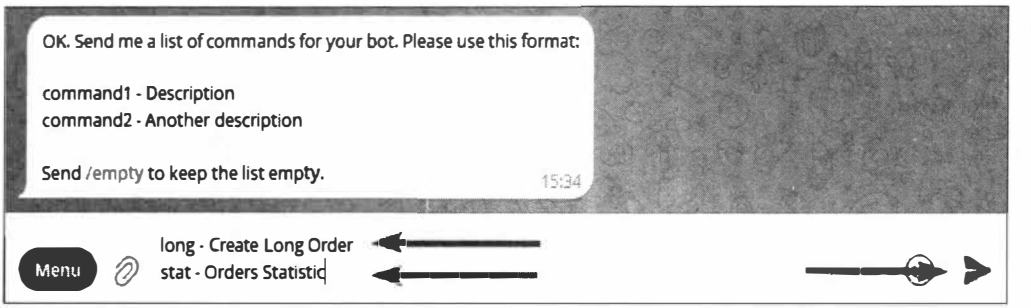
Изображение 3.17.

4. Выберите пункт EDIT COMMANDS.

Изображение 3.18.

5. Введите сразу 2 строки и отправьте в @BotFather:

```
long - Create Long Order
stat - Orders Statistic
```

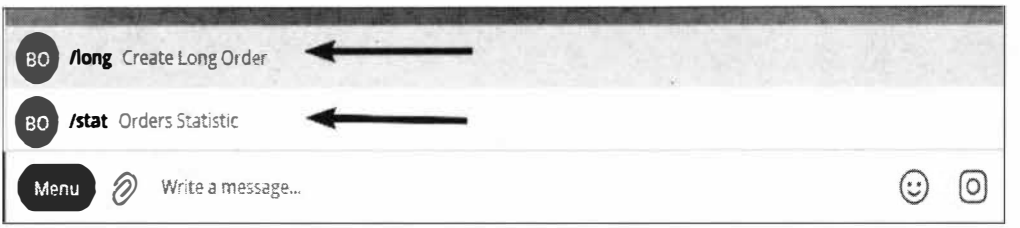


Изображение 3.19.

6. Зайдите в своего бота.

Изображение 3.20.

7. Теперь в боте доступны два пункта меню.



Изображение 3.21.

Создайте новую папку с названием BitcoinOptions, внутри папки создайте файл CreateOrders.py и перенесите в него код из листинга 3.12.

Листинг 3.12

```
from random import randint  # импортируем функцию RANDINT
from time import sleep  # Из библиотеки TIME импортируем функцию SLEEP
from requests import get # Импорт функции для GET-запросов
from datetime import datetime # Импорт объекта для работы со временем
from sqlite3 import connect # Импорт объекта для работы с БД
from time import sleep # Функция для паузы скрипта
from pybit.unified_trading import HTTP # Объект для BYBIT
```

```
from requests import post # Импорт функции для POST-запросов
from uuid import uuid4 # Функция для генерации id

program_name = "Ставки на Bitcoin"
payout = 1.89 # Процент выплаты
TG_API_URL = "https://api.telegram.org/bot" # Константа API
TGBOT = "7579403178:AAEU_aJ809TMlIM1OFZYaOLboWvLDrEZa7A" # Бот id
latest_update_id = None # id последнего сообщения
db = connect("bitcoin_options.db") # Подключаемся к базе данных
prepare_sql_query = db.cursor() # Создаём объект для подготовки запросов
session = HTTP(testnet=False) # Создаём объект подключения к бирже

def send_message(msg="Тестовое сообщение", chat_id=1349083375):
    # Функция для отправки сообщений в Телеграм
    # Принимает два аргумента: текст сообщение и id чата
    # В переменной final_url формируется ссылка для отправки
    # Возвращает ответ от ТГ
    url1 = TG_API_URL+TGBOT
    url2 = "/sendMessage?chat_id="+str(chat_id)+"&text="+str(msg)
    final_url =  url1+ url2 # Конкатенируем ссылку
    response = post(final_url) # Отправляем и сохраняем результат
    return response # Завершаем функцию

def get_updates(offset=-1):
    # Делаем GET-запрос и сохраняем
    result = get(TG_API_URL+TGBOT+"/"+"getUpdates?offset="+str(offset)).json()
    try:
        # Пытаемся отдать последнее сообщение
        return result["result"][0]
    except:
        return None # Завершаем функцию

def create_orders_table():
    # Функция для создания таблицы orders, если ещё не создана
    # Не принимает аргументов
    # Ничего не отдаёт
    # Подтягиваем переменные из глобальной области видимости
    global db, prepare_sql_query

    sql = """CREATE TABLE IF NOT EXISTS orders(
    id TEXT UNIQUE,
    chat_id INT,
    created_at TEXT,
    direction INT,
    sum REAL,
    price_at_start REAL,
    price_at_end REAL,
    executed INT DEFAULT 0,
```

```
    order_win_status INT DEFAULT 0,
    profit REAL)"""

    prepare_sql_query.execute(sql) # Подготавливаем SQL-запрос
    db.commit() # Выполняем запрос
    return # Завершаем функцию

create_orders_table() # Создаём таблицу ордеров

def get_btc_price():
    # Функция для получения текущей цены монеты
    # Принимает 1 аргумент со значением по умолчанию
    # Отдаёт текущую цену
    # Получаем информацию о текущем состоянии BTC
    result = session.get_tickers(
        category="linear",
        symbol="BTCUSDT")
    # Извлекаем из ответа текущую цену
    current_price = result["result"]["list"][0]["lastPrice"]
    print("Текущая цена", current_price)
    return float(current_price) # Отдаём результат

def win_calculate(sum):
    # Функция рассчитывает потенциальную сумму выигрыша
    # Принимает 1 аргумент
    # Возвращает результат выигрыша
    global payout
    result = int(sum) * payout
    print("Ты собираешься поставить ", sum)
    print("Потенциальный выигрыш", result)
    return result # Отдаём результат работы функции

def create_deal(chat_id=1341083375, sum=1000, time=1, direction=1):
    # Создаёт сделку
    # Принимает 4 аргумента: id чата, время экспирации, сумму сделки,
    # направление цены (1 — вверх, 0 — вниз)
    # Ничего не отдаёт

    print("<<<<<<---ПРИНЯТА СДЕЛКА-->>>>>>")
    print("Сумма", sum)
    print("Время", time)
    print("Направление", direction)

    price_at_start = get_btc_price() # Реальная котировка на старте
    potential_profit = win_calculate(sum) # Потенциальный выигрыш

    sql = """INSERT INTO orders (
        id,
```

```
        chat_id,
        created_at,
        direction,
        sum,
        price_at_start,
        profit)
    VALUES (?, ?, ?, ?, ?, ?, ?)"""

    # Параметры SQL-запроса
    order_id = str(uuid4())
    params = [order_id, int(chat_id), str(datetime.now()),
        direction, sum, price_at_start, potential_profit]

    try: # Обёртка для обработки ошибок
        prepare_sql_query.execute(sql, params) # Подготавливаем SQL-запрос
        db.commit() # Выполняем запрос
    except Exception as error: # Если запись уже существует
        print(error)
        send_message(msg=error, chat_id=chat_id)
        return # Заканчиваем работу
        # Если НЕ произошло ни каких ошибок
        msg1 = "Ордер: " + order_id
        msg2 = "\n Принята сделка на " + str(sum)
        msg3 = " по цене " + str(price_at_start)
        msg4 = msg1 + msg2 + msg3
        send_message(msg=msg4, chat_id=chat_id) # Шлём в ТГ
        return # Заканчиваем работу


def run():
    # Функция для запуска обработки регистраций
    # Ничего не принимает
    # Ничего не отдаёт
    # Подтягиваем переменную из глобальной области видимости
    global latest_update_id
    sleep(2) # Засыпаем на 2 секунды
    print("latest_update_id", latest_update_id)
    check_update = get_updates() # Получаем последнее сообщение

    # Если get_updates() вернула None
    if check_update == None:
        print("Новых сообщений давно не было")
        return # Аварийно завершаем функцию

    # Если id последнего сообщения бота НЕ обновилось
    if latest_update_id == check_update["update_id"]:
        print("Обновлений нет")
    else:
```

```
        # Если id последнего сообщения бота обновилось
        print("Получил обновления")
        latest_update_id = check_update["update_id"] # id обновления
        chat_id = int(check_update["message"]["chat"]["id"]) # id чата

        print("ID пользователя:", chat_id,
            "Сообщение:", check_update["message"]["text"])

        # Если в сообщении содержится строка "/start"
        if "/start" in check_update["message"]["text"]:
            print(chat_id, "Пытаешься зарегистрироваться")
            send_message(msg="Добро пожаловать в игру!", chat_id=chat_id)
        elif "/order" in check_update["message"]["text"]:
            print(chat_id, "создал ордер")
            create_deal(chat_id=chat_id)
        else:
            print("Нет такой команды")
            send_message(msg="Нет такой команды", chat_id=chat_id)
    run() # Рекурсивно перезапускаем функцию
    return # Завершаем текущее исполнение функции

run() # Запускаем отслеживание создания сделок
```

Функции send_message() и get_updates() перенесены из прошлых проектов и хорошо вам знакомы. Функция create_orders_table() создаст таблицу для ордеров. Статус исполненного ордера в поле executed помечаем как 1, неисполненного — как 0. Аналогично поступаем с полем order_win_status.

Функция get_btc_price() получает и отдаёт текущую цену BTC. Вызывается до исполнения ордера и после. Функция win_calculate() рассчитывает потенциальный выигрыш и ничем не отличается от той, что мы написали в первой главе. Функция create_deal() создаст ордер и запишет его в базу данных.

Пример CRON-файла для листинга 3.12:

```
0 * * * * python3 /home/ghost/BitcoinOptions/CreateOrders.py >>
users.txt
```

Обратите внимание, что в конце мы добавили >> users.txt. Это позволит вам вести логи программы и отслеживать непредвиденные ошибки.

3.39.2. Сервис для обработки неисполненных сделок

1. Перезапускаем 1 раз в минуту и фиксируем текущую цену Bitcoin.
2. Выбираем все сделки, которые НЕ были исполнены.
3. Проверяем зашла сделка или нет.
4. Обновляем статус выигрыша в базе данных.
5. Уведомляем клиента.

В листинге 3.13 представлен сервис для исполнения сделок. Создайте файл ExecuteOrders.py и перенесите в него код из листинга.

Листинг 3.13

```
from random import randint  # импортируем функцию RANDINT
from time import sleep  # Из библиотеки TIME импортируем функцию SLEEP
from requests import get # Импорт функции для GET-запросов
from datetime import datetime # Импорт объекта для работы со временем
from sqlite3 import connect # Импорт объекта для работы с БД
from time import sleep # Функция для паузы скрипта
from pybit.unified_trading import HTTP # Объект для BYBIT
from requests import post # Импорт функции для POST-запросов

TG_API_URL = "https://api.telegram.org/bot" # Константа API
TGBOT = "7579403178:AAEU_aJ809TMlIMlOFZYaOLboWvLDrEZa7A" # Бот id
db = connect("bitcoin_options.db") # Подключаемся к базе данных
prepare_sql_query = db.cursor() # Объект для подготовки запросов
session = HTTP(testnet=False) # Объект подключения к бирже

def get_btc_price():
    # Функция для получения текущей цены монеты
    # Принимает 1 аргумент со значением по умолчанию
    # Отдаёт текущую цену
    # Получаем информацию о текущем состоянии BTC
    result = session.get_tickers(
        category="linear",
        symbol="BTCUSDT")
    # Извлекаем из ответа текущую цену
    current_price = result["result"]["list"][0]["lastPrice"]
    print("Текущая цена", current_price)
    return float(current_price) # Отдаём результат

def send_message(msg="Тестовое сообщение", chat_id=1349083375):
    # Функция для отправки сообщений в Телеграм
    # Принимает два аргумента: текст сообщение и id чата
```

```
    # В переменной final_url формируется ссылка для отправки
    # Возвращает ответ от ТГ
    url1 = TG_API_URL+TGBOT
    url2 = "/sendMessage?chat_id="+str(chat_id)+"&text="+str(msg)
    final_url =  url1+ url2 # Конкатенируем ссылку
    response = post(final_url) # Отправляем и сохраняем результат
    return response # Завершаем функцию

def change_execute_status(id, win_status, price_at_end):
    # Меняем статус ордера на исполненный
    # Фиксируем успех  / проигрыш
    # Ничего не возвращает
    sql = """UPDATE orders SET executed = 1,
        order_win_status = ?,
        price_at_end = ? WHERE id = ?"""
    prepare_sql_query.execute(sql, (win_status, price_at_end, id))
    db.commit() # Выполняем подготовленный запрос
    return # Завершаем работу

def calculate_unexecuted_orders():
    price_at_end = get_btc_price() # Фиксируем цену BTC
    # Получаем неисполненные ордера
    sql = "SELECT * FROM orders WHERE executed = 0"
    prepare_sql_query.execute(sql)
    unexecuted_orders = prepare_sql_query.fetchall()

    for order in unexecuted_orders:
        id = order[0] # ID ордера
        chat_id = order[1] # ID чата
        created_at = order[2] # Дата создания
        direction = order[3] # Направление сделки
        sum = order[4] # Сумма сделки
        price_at_start = order[5] # Цена открытия
        profit = order[9] # Потенциальный профит

        print("ID:", id)
        print("ID чата:", chat_id)
        print("Дата создания:", created_at)
        print("Направление:", direction)
        print("Сумма:", sum)
        print("Цена открытия:", price_at_start)
        print("Цена закрытия:", price_at_end)
        print("Потенциальный профит:", profit)

        msg1 = "Ордер: " + id + "\n Цена на старте: " + str(price_at_start)
        msg2 = "\n Цена в конце: " + str(price_at_end)

        # Если зашла ставка на падение
```

```
        if direction == 1 and price_at_end > price_at_start:
            print(chat_id, "выиграл", profit) # Уведомляем админа
            msg3 = "\n Ты выиграл: " + str(profit)
            final_msg = msg1 + msg2 + msg3 # Конкатенируем строки
            change_execute_status(id=id, win_status=1,
                price_at_end=price_at_end) # Меняем статус сделки
            send_message(msg=final_msg, chat_id=chat_id) # Шлём в ТГ
        elif direction == 1 and price_at_end < price_at_start:
            print(chat_id, "проиграл", sum) # Уведомляем админа
            msg3 = "\n Ты проиграл: " + str(sum)
            final_msg = msg1 + msg2 + msg3 # Конкатенируем строки
            change_execute_status(id=id, win_status=1,
                price_at_end=price_at_end) # Меняем статус сделки
            send_message(msg=final_msg, chat_id=chat_id)# Шлём в ТГ
        else:
            print("Возврат ставки", sum)
            msg3 = "\n Возврат ставки: " + str(sum)
            final_msg = msg1 + msg2 + msg3 # Конкатенируем строки
            change_execute_status(id=id, win_status=0,
                price_at_end=price_at_end) # Меняем статус сделки
            send_message(msg=final_msg, chat_id=chat_id)# Шлём в ТГ
    return  # Завершаем функцию, ничего не отдав

calculate_unexecuted_orders() # Вызываем функцию для расчёта ордеров.
```

- Функции get_btc_price() и send_message() хорошо вам знакомы и не требуют объяснений.
- Функция change_execute_status() изменит статус ордера и зафиксирует прибыль/убыток.
- Функция calculate_unexecuted_orders() нырнет в базу данных, выберет все неисполненные ордера, сравнит выбранное пользователем направление сделки (в нашем примере только лонг) и в зависимости от текущей котировки рассчитает результат сделки.

Пример CRON-файла для листинга 3.13:

```
* * * * * python3 /home/ghost/BitcoinOptions/ExecuteOrders.py >>
orders.txt
```

Есть один нюанс: при перезапуске скрипта с помощью CRON он повторно обработает последнее сообщение, пришедшее от пользователя. Я оставлю решение этой проблемы в качестве задачи. Просто фиксируйте, был ли скрипт запущен в первый раз, и игнорируйте последнее сообщение.

3.40. Принимаем оплаты от пользователей

Мы создали отличные инструменты, которыми можем пользоваться сами и отдавать людям на безвозмездной основе. Подобного рода инструменты в состоянии приносить деньги, а значит, тоже должны стоить денег.

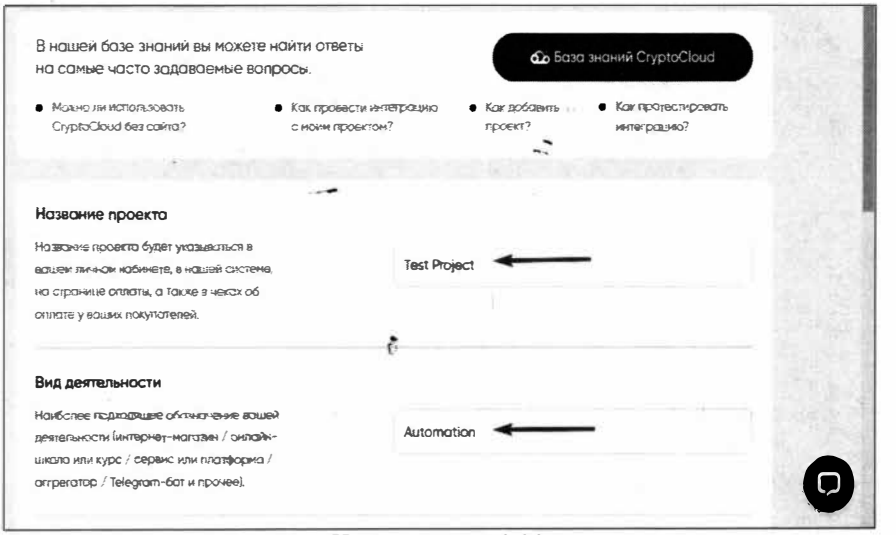
В конце концов на разработку было потрачено время и силы, а серверы нужно оплачивать каждый месяц. Я предоставляю вам простой демонстрационный пример, который вы сможете адаптировать под свои цели.

Для приема платежей мы используем сервис *https://cryptocloud.plus*. Сервис НЕ требует подтверждения личности и работает с проектами, которым присвоен статус высокорисковых.

1. Зарегистрируйтесь на *https://cryptocloud.plus*.
2. Зайдите в аккаунт и нажмите **Добавить новый проект**.

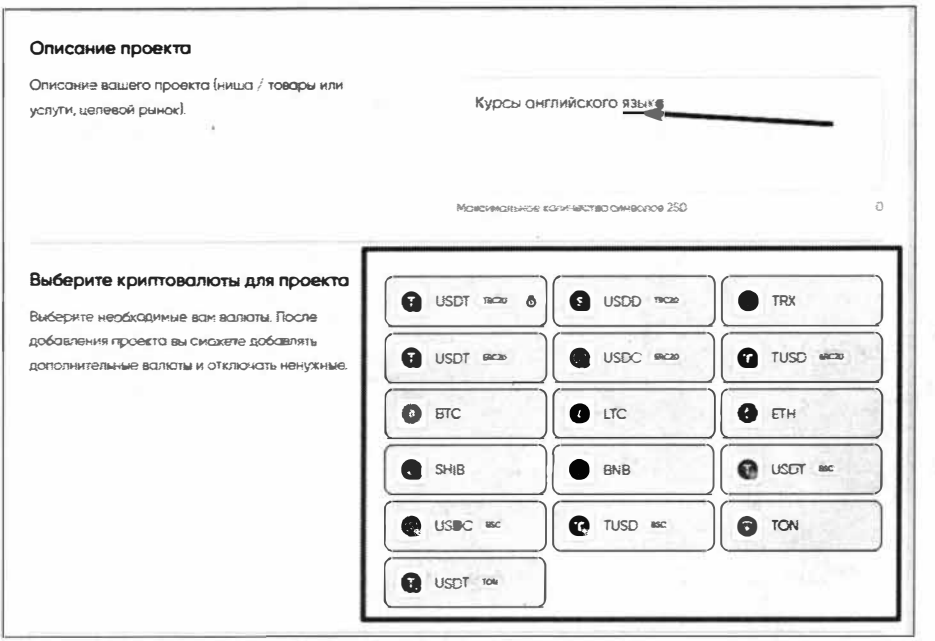
Изображение 3.22.

3. Дайте проекту название и вид деятельности. Следом прокрутите страницу вниз.



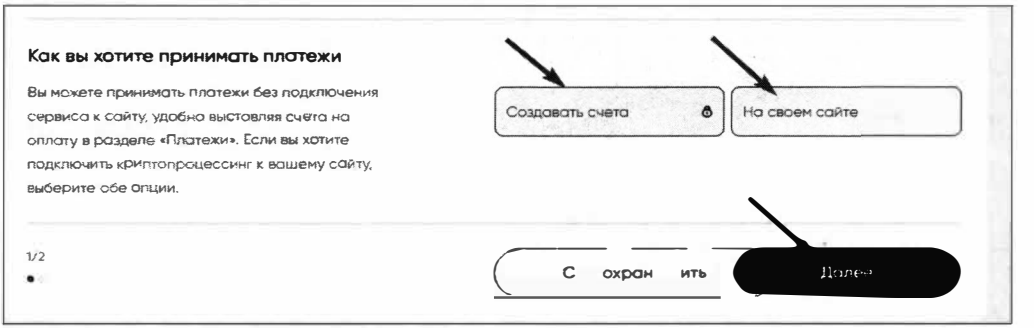
Изображение 3.23.

4. Дайте описание проекта и выберите монеты, которые хотите принимать.



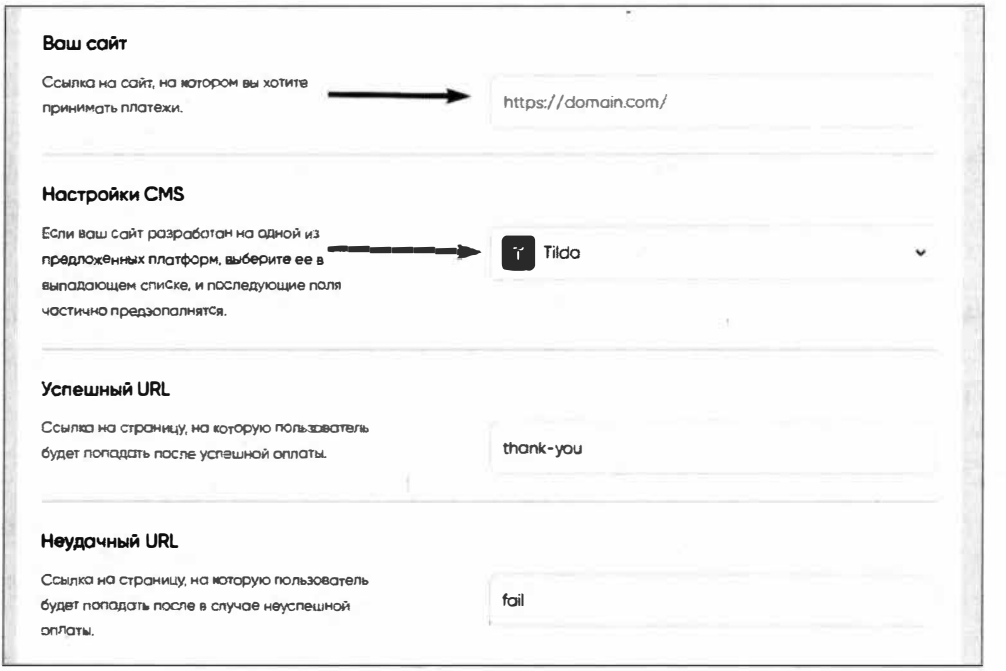
Изображение 3.24.

5. Выберите оба пункта для получения платежей.



Изображение 3.25.

6. Выберите в настройках CMS Tilda и введите любой Tilda Url.

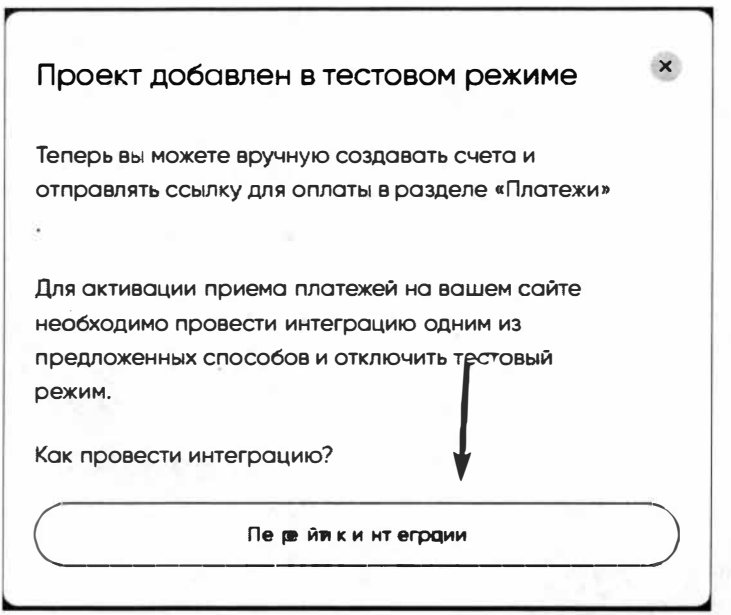


Изображение 3.26.

7. Пролистайте вниз и нажмите **Далее**.

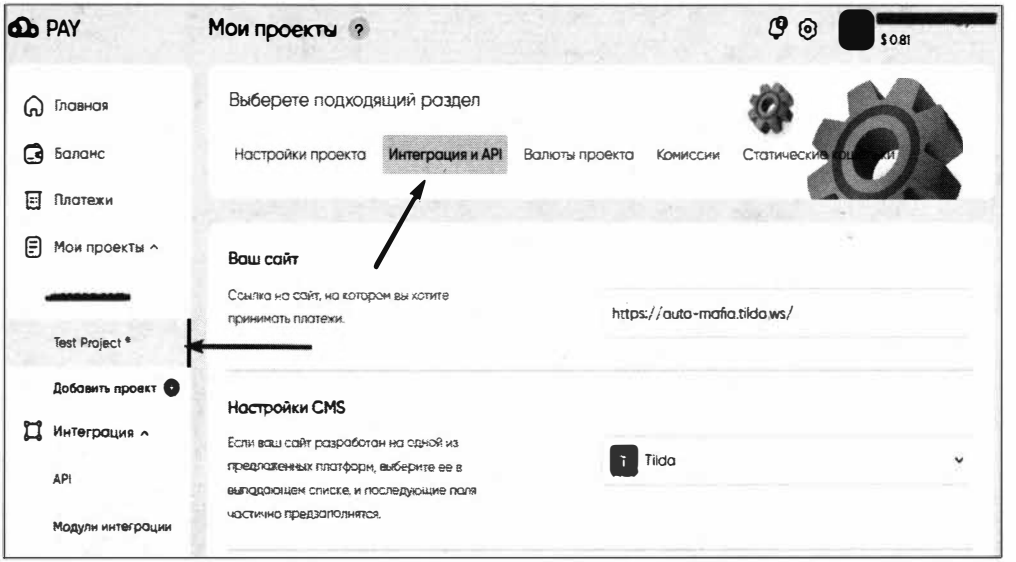
Изображение 3.27.

8. В появившемся окне нажмите **Перейти к интеграции**.



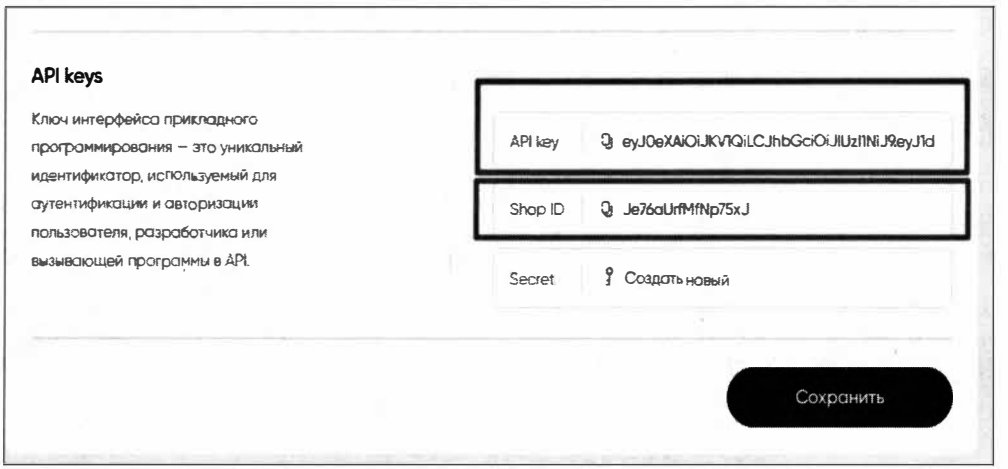
Изображение 3.28.

9. Перейдите в созданный проект и нажмите **Интеграция и API**.



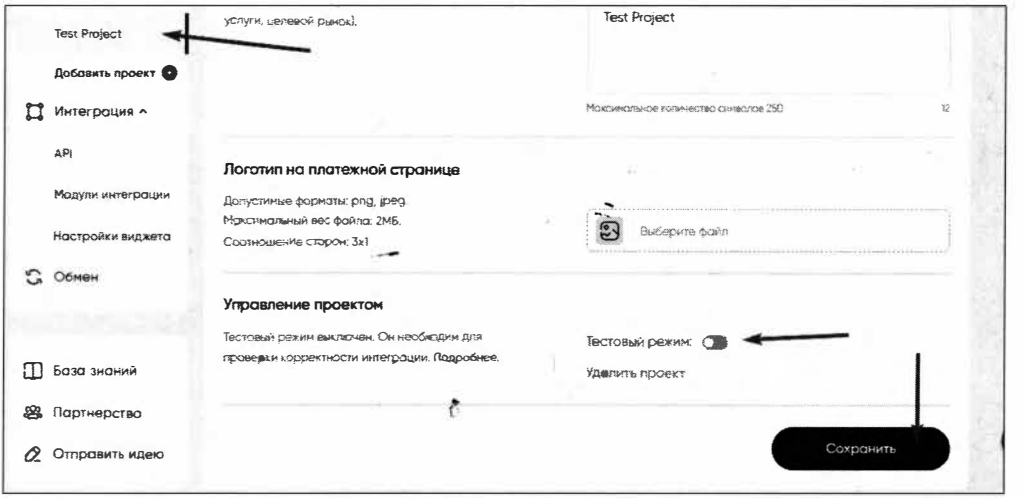
Изображение 3.29.

10. Пролистайте вниз страницы и скопируйте ключи доступа к API.



Изображение 3.30.

11. Перейдите в проект и в конце страницы уберите **Тестовый режим**.



Изображение 3.31.

Сохраните ключ для доступа к API и первым делом прочитайте документацию хотя бы по диагонали *https://docs.cryptocloud.plus/ru*.

В профессиональной разработке чтение документации для используемых библиотек является не менее важной частью процесса, чем написание кода.

Листинг 3.14

```
from requests import post

# Ссылка для создания статического кошелька
url = "https://api.cryptocloud.plus/v2/invoice/static/create"
TOKEN = "" # Ключ API
SHOP_ID = "" # ID Шопа

headers = { # Заголовок с ключом API
    "Authorization": TOKEN
}

body = { # Тело запроса с ID шопа, валютой кошелька и идентификатором
    "shop_id": SHOP_ID,
    "currency": "BTC",
    "identify": "F4jSjfpwsdfsdfgdfhfdgyjurftfgbdsfyrtyhfgcbndfjgff4t",
}
```

```
def create_static_wallet():
    # Создаст статический кошелёк
    # Вернёт созданный кошелёк

    # Делаем запрос на создание кошелька
    response = post(url, headers=headers, json=body)

    if response.status_code == 200: # Если статус ответа успешен
        print("Успех:", response.json())
        return response["result"]["address"] # Отдаём адрес кошелька
    else: # Если ответ НЕ равен 200
        # Печатаем сообщение об ошибке
        print("Ошибка:", response.status_code, response.text)
        return # Завершаем функцию

new_wallet = create_static_wallet() # Создаём новый кошелёк
```

Если вы получили в ответ адрес кошелька, значит всё в порядке и можно работать дальше.

Теперь при регистрации нового пользователя вы можете сразу создать для него статический кошелёк, который навсегда будет привязан к его депозитам.

Более продвинутые примеры ищите в документации по адресу:

https://docs.cryptocloud.plus/ru

Если у вас возникли проблемы при тестировании, напишите в службу поддержки. Вам помогут.

Глава 4.

Его величество Бэктрейдер. Бэктест торговых стратегий

Содержание главы:

4.1. Как появились биржи и как существуют сейчас?

Первые биржи появились, когда процветало ростовщичество. В те времена очень многие вожделели открыть крупный бизнес и, естественно, необходимы были займы. Толстосумы собирались, чтобы продавать и покупать долги, а ценной бумагой выступала долговая расписка.

Если брать современный рынок, который мы видим сейчас, его зачатки появились в Голландии. Там появились такие понятия, как опционы, затем они переросли в бинарные опционы, но всё это был неофициальный рынок, что-то подпольное и слабо регулируемое.

Первая более или менее официальная фондовая биржа появилась в Англии. Из Англии она перетекла в Америку, в Нью-Йорк, и в том виде, в котором мы видим сейчас, она с тех пор и существует.

После появления фондовой Нью-Йоркской биржи появился рынок Форекс. Это банковский сектор, к которому лично у меня очень слабое доверие. Они скрывают информацию об объёмах, прикрываясь огромностью рынка, при этом каждый форекс-брокер подключен к глобальному банковскому котлу, а значит, информация об объёмах есть.

Разглашать информацию о текучке своих средств банки, конечно же, не собираются, поскольку это упростит работу трейдеров. Официально известно, что ЦБ РФ потерял на этом рынке более 50 000 миллиардов долларов. Несмотря на это, рынок Форекс отлично подходит для консервативной алгоритмической торговли.

Самым молодым рынком на момент написания книги является криптовалютный. Я называю его "Рынком с человеческим лицом". В том или ином виде криптовалюты сделали людей намного свободнее. Криптовалюта позволяет исключить посредника в виде банка и скрывать транзакции от внешнего мира (GHOST, MONERO, etc).

Те деньги, которые находятся на вашем холодном или аппаратном кошельке, принадлежат вам. При этом само по себе появление этой технологии окутано тайной, и до сих пор неизвестно, кто сделал этот дар людям. В этом дух свободы криптовалют.

Первые 10 лет криптовалюта НЕ была интересна крупному капиталу, поскольку не было достаточной ликвидности. Достаточно необходимого предложения не существовало, и киты банально не могли закупаться на те суммы, которыми оперируют.

Лишь в 2020-2021 годах этот рынок стал интересен институционалам, и теперь они активно им манипулируют. BlackRock Inc. даже спасают там свои капиталы в темные времена.

Главное отличие рынка криптовалют в том, что он **манипулятивный**. Любой имеющий 30000$ может искусственно разогнать цену на споте, на небольшой монете, потянув за собой "хомяков" через сеть пабликов ВК или Телеграма, а затем встать в шорт с высоким плечом на фьючерсах, тем самым "побрить рынок".

За подобные манипуляции на фондовой бирже в США дают очень солидные уголовные сроки. Криптовалютный же рынок всё ещё является Диким Западом и собирает вокруг себя огромное количество мошенников или просто желающих поживиться на людях. Ярким примером криптоманипуляторов являются Дональд Трамп и Илон Маск, которых, созданная Маском нейронка, порекомендовала казнить за популизм.

Ещё одним важным отличием от остального рынка является то, что львиная доля технического анализа не работает на большинстве монет по всё той же причине манипулятивной природы.

Несмотря на все кажущиеся недостатки, именно благодаря криптовалютам простые люди тоже могут участвовать в международном рынке, при этом сохраняя свою анонимность.

До появления Форекса и крипторынка, а также современных средств автоматизации обычному парню из Бронкса или Саратова нечего было противопоставить институционалам. Теперь нам эти средства доступны.

4.2. Кто присутствует на рынке?

Институциональные трейдеры — это команды разработчиков, которые создают автоматизированные торговые системы и управляют средним и крупным капиталом.

Большую часть их средств образуют хедж-фонды и бытовые инвесторы. Работают со времён появления первых компьютерных систем. Основные манипуляторы на рынке криптовалют.

Бытовые трейдеры — это мы с вами. Ребята, у которых есть ЭВМ и желание не зависеть от хозяина. Бытовые трейдеры-новички создают основной источник дохода остальных игроков.

Именно бытовых трейдеров-непрофессионалов называют "Хомяками".

Институциональные инвесторы — крупные и средние фонды, большую часть средств которых составляют бытовые инвесторы. Не заинтересованы в краткосрочном извлечении прибыли.

Наиболее известным примером является компания BlackRock Inc.

Бытовые инвесторы — частные инвесторы, которые заинтересованы в долгосрочном извлечении прибыли с различной степенью риска. Используют рынок криптовалют в первую очередь для сохранения и накопления собственных средств, а также для защиты от инфляции.

Манимейкеры — это преимущественно программисты, создающие псевдоликвидность с помощью роботов.

Благодаря им создаются фейковые крупные лимитные заявки, которые автоматически сдвигаются. Делается это для концентрации трейдеров у уровней. Зарабатывают преимущественно на долевом участии в прибыли от биржевых комиссий.

Биржи — посредники между покупателями и продавцами. Практически не имеют собственных средств и зарабатывают на убытках своих клиентов. Биржи имеют большой штат роботов, которые усредняют спотовый рынок с фьючерсным, на котором происходит 90% всех торгов.

4.3. Как зарабатывают биржи?

В идеальном мире биржа должна зарабатывать исключительно на комиссии. Пришёл продавец, пришёл покупатель — биржа за "знакомство" имеет свою комиссию и все риски за обеспечение сделок берет на себя.

На деле биржи научились зарабатывать на выбивании стоп-лоссов и ликвидациях благодаря созданию фьючерсов с кредитным плечом. Это их основной источник заработка.

Как только появились первые технические средства для бытовых трейдеров, так называемые "торговые терминалы", биржа стала видеть все стоплоссы, которые выставляет трейдер. Ей ничего не стоит за счёт собственных технических средств или манимейкеров (которые автоматизированно двигают крупные лимитные заявки) нарисовать длинную тень на свече, которая поглотит вашу позицию.

Примерно так же поступают крупные институциональные трейдеры, сдвигая цену к уровням ликвидности. Именно за уровнями поддержки и

сопротивления, где размещены стопы и фьючерсные ликвидации, охотятся институциональные трейдеры. Без этой ликвидности они не могут открыть крупную позицию.

4.4. Как не стать наживой для китов?

Мы, как бытовые трейдеры, с точки зрения потенциала для заработка находимся в уязвимом положении, потому что играем на чужой территории. При этом человек сам по себе слабое звено, и все статистические данные показывают, что ручной трейдер в 93% случаев убыточен.

Мы обязаны осознавать, учитывать и помнить: биржа нам НЕ друг! И лучшее, что мы можем сделать для своего депозита, а следовательно, и здоровья, — это исключить себя из ручной торговли. Тем не менее, кое-что мы можем противопоставить рынку, поскольку в наших руках появились инструменты, ранее доступные только Крупному Капиталу.

Главным нашим преимуществом является тот факт, что нам НЕ нужна огромная ликвидность, чтобы открыть позицию и, в отличие от институциональных трейдеров, мы можем работать в значительно больших рыночных ситуациях!

Примеры бэктестов, описанных в этом разделе, применимы как для форекс-рынков, так и для рынка акций. Главное, благодаря бэктестам найти инструменты, на которых формация или индикатор хорошо отрабатывают, и любой ценой автоматизировать размещение сделок и управление рисками.

4.5. Его Величество Backtrader

Перед нами стоят 3 основные задачи:

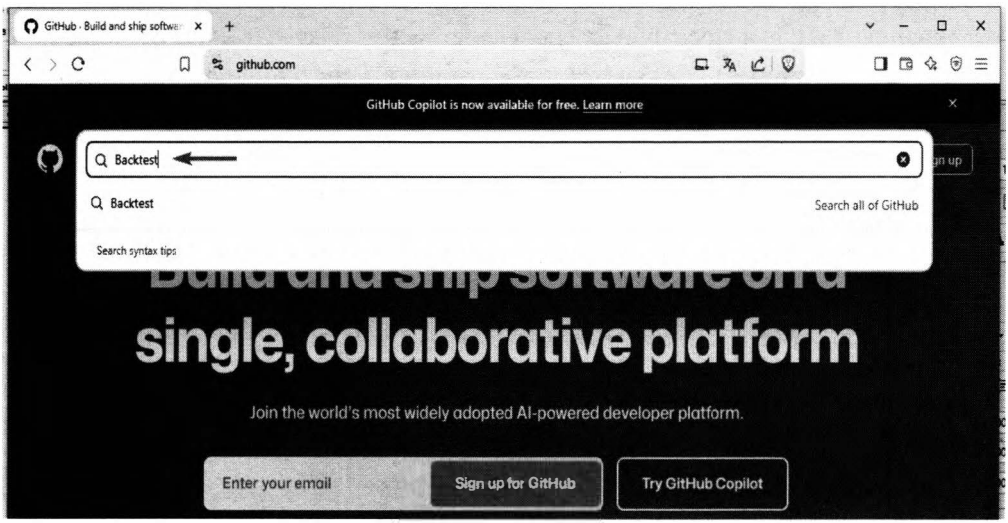
1. Выбрать актив для торговли.
2. Обрести инструмент для тестирования стратегий на истории.
3. Автоматизировать успешную стратегию.

В качестве решения первой задачи мы остановимся на Bitcoin, поскольку это первая криптовалюта в мире и она является поводырём для большинства других монет. Второй монетой станет Litecoin, который давно стал "тихой гаванью". По своему движению LTC напоминает очень хорошую акцию, что позволяет отрабатывать простые и эффективные формации.

Для тестирования стратегий и автоматизации исполнения ордеров необходимо правильно отобрать библиотеки. Основных критериев при выборе инструмента для разработки три:

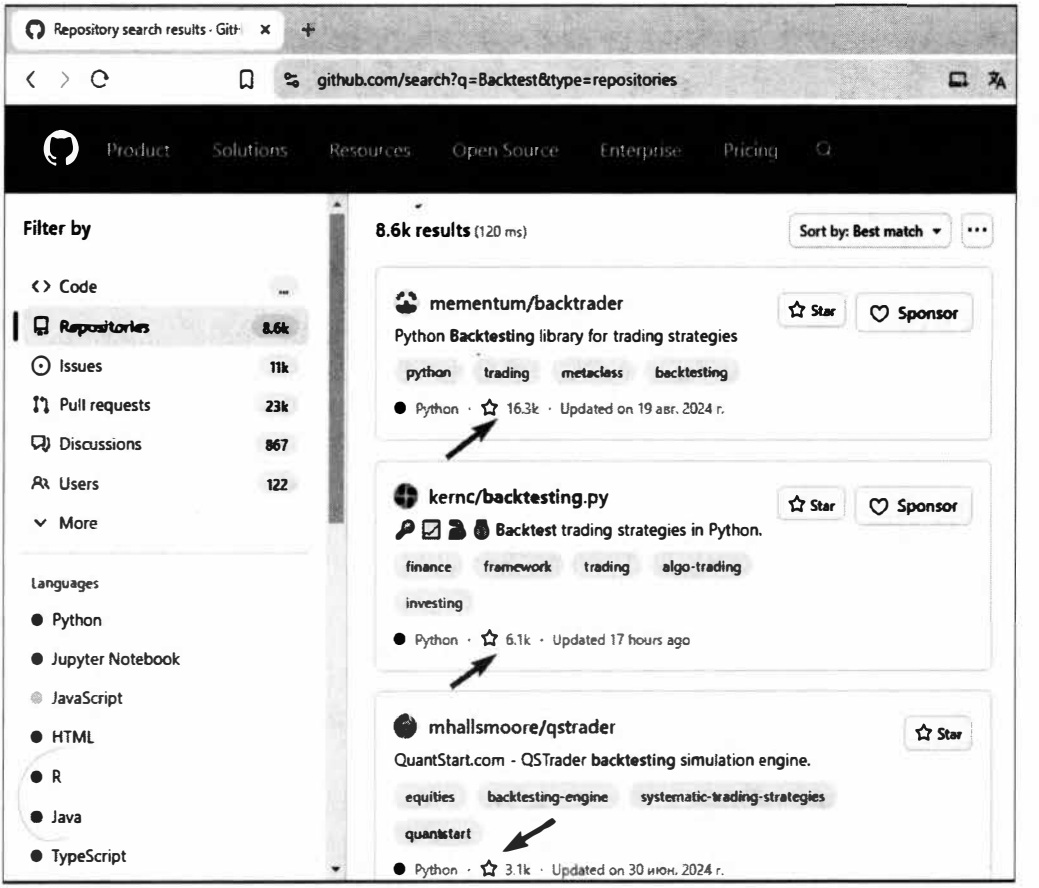
1. Зрелость.
2. Документация.
3. Сообщество.

Зайдите на сайт www.github.com, вбейте в строку для поиска слово *Backtest* и нажмите клавишу **Enter**.



Изображение 4.1.

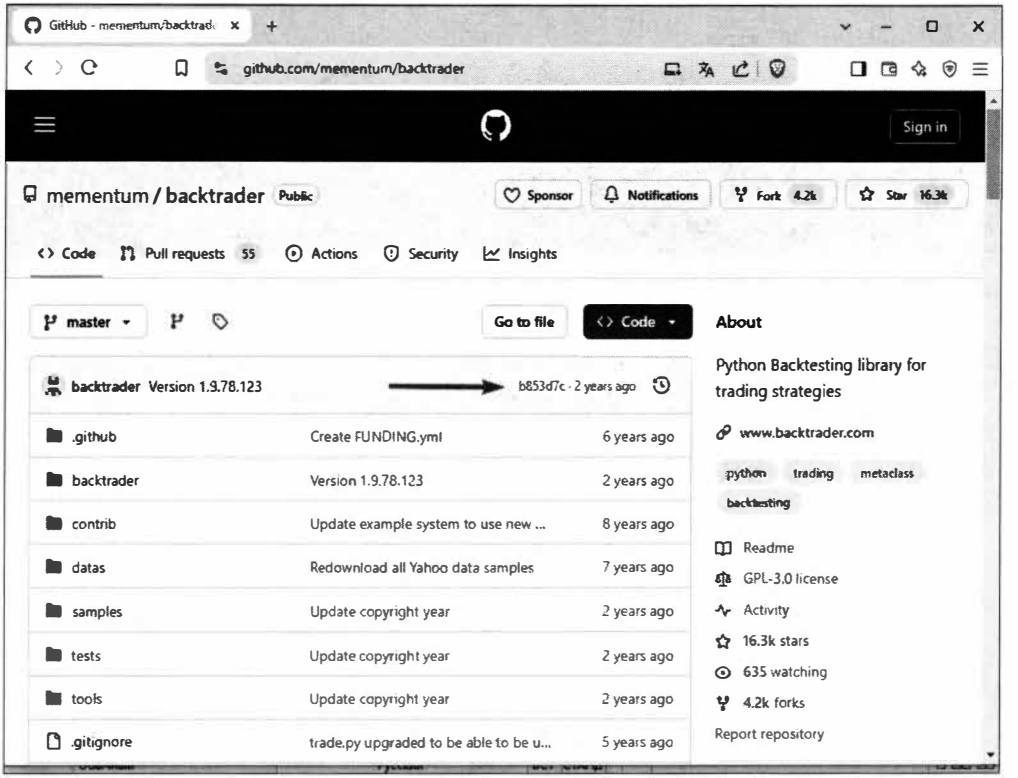
В ответ на поисковый запрос вы увидите примерно следующее:



Изображение 4.2.

В первую очередь вы должны обращать внимание на количество звезд на GitHub. Чем больше звезд, тем больше людей крутится вокруг библиотеки. Это самый примитивный и простой способ.

Затем заходим на страницу библиотеки и смотрим дату последнего коммита. Это даёт нам информацию о её актуальности. Поддерживается она создателем или нет.



Изображение 4.3.

Дальше стоит обратить внимание на дату создания и определить, насколько библиотека зрелая. Чем старше библиотека, тем лучше, потому что модное исчезает так же быстро, как и появляется.

Следующий пункт — это объём документации. Один из самых важных пунктов. Бывает так, что библиотека хороша, но плохо документирована, и вам придётся лезть в исходники, чтобы выяснить недокументированные функции.

> **Простота кода** — тоже важный критерий при выборе инструмента. Когда вы перебираете библиотеки, необходимо обращать внимание, как выглядит её код. Приятно ли его читать, понятно ли его читать, хочется ли писать на этой библиотеке...

Последний пункт — это активное сообщество. В разработке вообще в одиночку оставаться не следует. Вы должны быть уверены в том, что сможете получить помощь в случае возникновения проблем. Обращайтесь за помощью в чат, посвященный этой книге.

Обычно в топ-5 поисковой выдачи содержится то, что вам необходимо. Мы остановимся на Backtrader. 13 тысяч звезд, создана около 10 лет назад и прекрасно документирована.

Я считаю Backtrader одной из самых гениальных библиотек на своей памяти. Её исходный код прост и понятен. Тем не менее, порог входа всё ещё остается высоким, поскольку документация написана для программистов с большим опытом и комплексных примеров кода найти сложно. Этот раздел книги значительно порог входа понижает.

К сожалению, есть несколько НО... Библиотека считается завершённой, практически НЕ обновляется (это хорошо). Она потеряла сообщество (это плохо). С другой стороны, она настолько хорошо написана, настолько фундаментально выстроена и проста, что вокруг неё до сих пор строятся всевозможные дополнения. А именно...

В 2024 году товарищи написали для неё вспомогательные библиотеки, которые могут подключаться к Qwik, TradingView, Финам, Тинькофф, Allora, Bybit и Binance. Мы будем работать с Bybit.

Это единственная библиотека, которая позволяет из коробки бесплатно тестировать стратегии без ограничений, и, в отличие от конкурентов, её расширения позволяют бесплатно заниматься размещением сделок на биржах.

Backtrader живет децентрализованно за счёт энтузиазма отдельных лиц, и, по сути, по возможностям её до сих пор никто не обогнал.

Что делать относительно сообщества? Как уже говорилось, находиться в среде единомышленников и с теми, у кого больше опыта, очень-очень важно.

Я нашёл несколько решений:

- Reddit.com.
- Группы в Telegram.

В Reddit есть ветка Algotrading. Там 1.8 миллиона человек, обязательно на неё подпишитесь и читайте. Почему?

Хорошие идеи нужно красть, ведь это законно! В этой ветке проскальзывают хорошие мысли для фондового рынка, и, поскольку криптовалюта постепенно становится китовой, какие-то идеи можно использовать.

В том числе людям можно писать в личные сообщения. Кто-то более открыт, кто-то менее, но в целом сообщество общительное.

Мы отобрали библиотеку, не побоявшись потратить на это время. Что дальше?

Обычно, прежде чем начать писать на выбранной библиотеке, первое, что нужно сделать, — это полностью прочесть документацию хотя бы по диагонали. Что-то отложится на подкорке, и вы будете понимать, в какой части документации искать референсы кода.

Это первый шаг для того, чтобы чувствовать себя наиболее уверенно с тем инструментом, с которым планируете работать в ближайшие годы.

> Русская документация находится на сайте *www.backtrader.ru*.

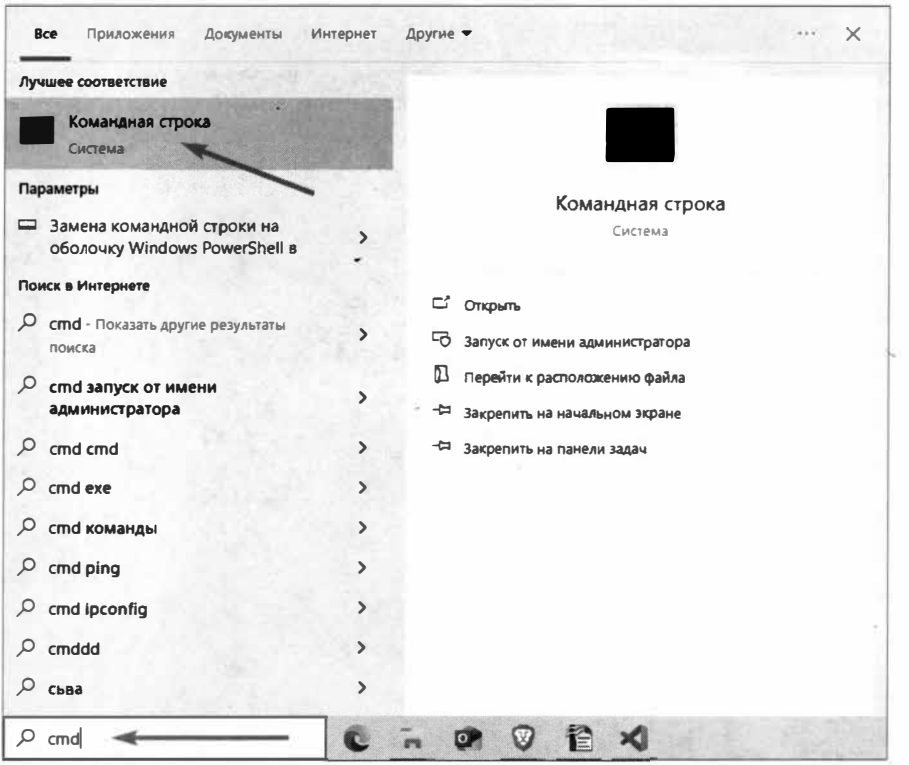
4.6. Установка необходимых библиотек

Нам понадобятся библиотеки для бэктестинга (backtrader), визуализации бэктеста (plotting), анализа наилучших исторических данных (pandas).

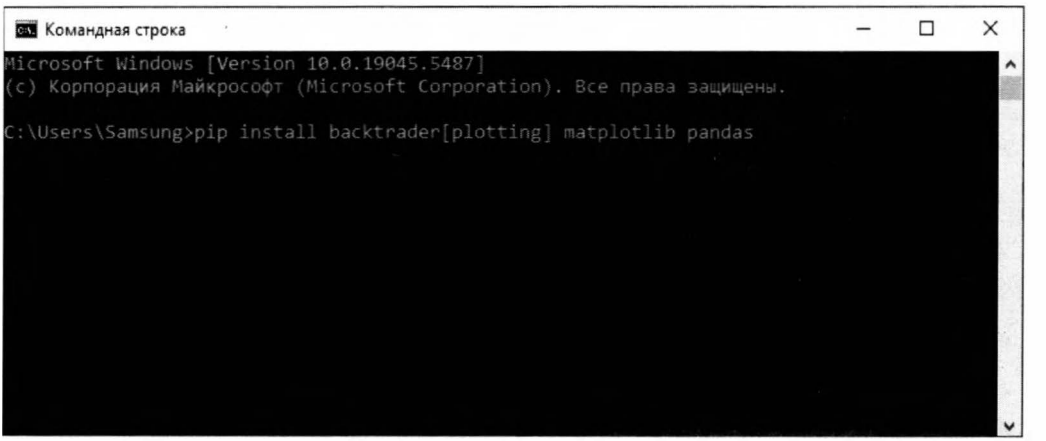
1. Нажмите на кнопку **Пуск** в левой нижней части экрана, наберите символы CMD и откройте терминал, как показано на изображении 4.4.

2. В терминале наберите следующий код и нажмите клавишу **Enter**:

```
pip install backtrader[plotting] matplotlib pandas
```

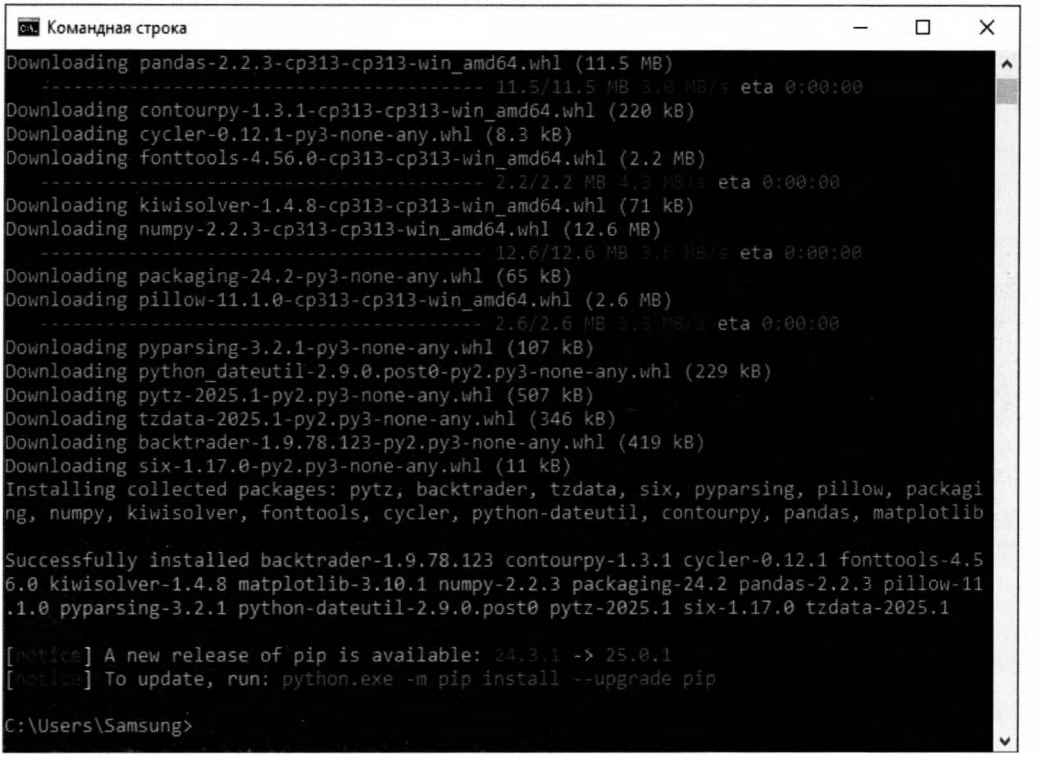


Изображение 4.4.



Изображение 4.5.

3. После запуска вы увидите примерно следующий вывод:



Изображение 4.6.

Данный код установит все необходимые зависимости и библиотеки, которые понадобятся для работы.

4.7. Устройство библиотеки Бэктрейдер и торговля на откатах

Большая часть компонентов Бэктрейдер написана буквально в 50–150 строк, что делает библиотеку очень надёжной, ведь чем меньше кода в инструменте, тем меньше ошибок.

Условно можно разбить написание кода Бэктрейдер на три основных этапа:

1. Класс стратегии, где мы отслеживаем поступление новых свечей, описываем логику открытия и закрытия ордеров, управляем рисками, индикаторами, отслеживаем трейды.
2. Подключение исторических данных.
3. Аналитика и визуализация бэктестов.

Условимся, что файл каждой стратегии будет также разделен на 3 основные части:

1. Самая верхняя часть файла для импорта библиотеки.
2. Средняя часть для объявления класса стратегии и описания её поведения.
3. Нижняя часть для подключения аналитики и визуализации данных.

Первым делом необходимо разобраться, как устроено получение и отслеживание исторических свечей Бэктрейдер, а также изучить его основные части.

Создайте проект с названием CandelsOnly. Внутри проекта создайте папку **data** и перенесите в неё утилиту TickerGrub.py. Скачайте свечные данные для BTC и LTC.

Создайте файл для знакомства с основными компонентами Backtrader и перепишите следующий код:

Листинг 4.1

```
from os.path import abspath # Функция для работы с путями
from datetime import datetime # Объект для работы со временем
from backtrader import Cerebro, Strategy, sizers, analyzers # Компоненты
from backtrader.feeds import GenericCSVData # Основной класс для CSV

class CandlesOnly(Strategy): # Наследуемся от класса  Strategy
    # Параметры торговой системы в виде словаря
    params = (
        "ExitCandles": 5,  # Через сколько дней выйдем из позиции
    )

    def __init__(self):
```

```
        # Инициализация стратегии и временных рядов
        # В self.data хранится временной ряд первого источника данных
        # Сокращаем внешний вид переменных с данными о свечах
        self.time = self.data.datetime # Время свечей
        self.open = self.data.open # Цены открытия
        self.close = self.data.close # Цены закрытия
        self.high = self.data.high # Цены максимума
        self.low = self.data.low # Цены минимума
        self.volume = self.data.volume # Объёмы
        self.order = None # Храним ордер
        self.win_trades = [] # Суммы прибыльных трейдов
        self.lose_trades = [] # Суммы убыточных трейдов

    def notify_order(self, order):
        # Срабатывает, когда изменяется статус ордера
        print("Статус ордера изменился", order.status)

    def notify_trade(self, trade):
        # Срабатывает, когда произошел трейд
        print("Произошел трейд", trade.pnl) # Финальная сумма с трейда

    def next(self):
        # Появление новой свечи и логика стратегии
        # Индекс используется для получения бара
        # Индекс 0 — это текущая свеча, индекс -1 — предыдущая
        print("Datetime", self.time[0])
        print("Open", self.open[0])
        print("Close", self.close[0])
        print("High", self.high[0])
        print("Low", self.low[0])
        print("Volume", self.volume[0])

    def stop(self):
        # Срабатывает после завершения бэктеста
        pass

# Подготовка источника свечных данных
# MyCSVData — основной класс для загрузки CSV-файла
class MyCSVData(GenericCSVData):
    # В параметрах указываем, какой столбец CSV-файла
    # за какие данные отвечает
    params = (
        "dtformat": "%Y-%m-%d", # Формат времени для парсинга
        "datetime": 0,
        "open": 1,
        "high": 2,
        "low": 3,
        "close": 4,
```

```
        "volume": 5,
        "openinterest": -1, # -1 — это последний столбец
    }

csv_file_path = abspath("./data/BTCUSDT_D.csv") # Путь к источнику данных
data = MyCSVData( # Объект с источником свечных данных
    dataname=csv_file_path,
    fromdate=datetime(2022, 1, 24),
    todate=datetime(2024, 8, 10),
    reverse=False)


# Подключение источника данных, аналитика и запуск бэктеста
cerebro = Cerebro() # Создаём объект Церебро. Мозг Бэктрейдера
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data
cerebro.addstrategy(CandlesOnly) # Добавляем стратегию CandlesOnly
cerebro.broker.setcash(10000) # Устанавливаем стартовый баланс
cerebro.addsizer(sizers.FixedSize, stake=0.01) # Размер позиции
cerebro.broker.setcommission(commission=0.0003) # Комиссия брокера
cerebro.run() # Запускаем бэктест
cerebro.plot(style="candle") # Рисуем свечной график
```

Разберем исходник поэтапно.

1. В первой части исходника мы импортируем все необходимые классы и дополнительные библиотеки.
2. Затем объявляем класс CandelsOnly(), унаследованный от базового класса Strategy.
3. Сразу после объявления класса CandelsOnly(), перед методом __init__(), мы добавили параметр стратегии params = {"ExitCandles": 5}. Это обычный словарь, в котором мы храним одну пару "ключ: значение". Этот параметр поможет нам выйти из позиции на шестой день.
4. В классе CandelsOnly() мы перегрузили 5 методов:
   - Метод __init__() — конструктор для инициализации свойств и индикаторов.
   - Метод notify_order() — срабатывает, когда изменяется статус ордера.
   - Метод notify_trade() — срабатывает, когда происходит трейд.

» Метод next() — логика стратегии. Срабатывает при появлении новой свечи.

» Метод stop() — срабатывает по завершении бэктеста.

5. В конце подключили источник свечных данных в объект Церебро и запустили бэктест.

Остановимся на каждом этапе и компоненте более подробно.

- В магическом методе __init__() мы делаем свойства подключенных в Церебро свечей более читабельными, а также добавляем свойства self.win_trades и self.lose_trades для хранения сумм прибыльных и убыточных сделок. По сути, в переменной *self* хранится тот набор данных, который был передан в CSV-файлах.

  В self.data хранится первый подключенный источник свечей. Следующий поток временных рядов будет храниться в self.data1. Можно добавить весь рынок и использовать список self.datas[0...n]. Если углубиться в детали, то каждое свойство, которое мы объявили, является списком данных, соответствующих столбцам в CSV.

- Метод notify_order() срабатывает каждый раз, когда изменяется статус ордера. В этом методе можно фиксировать цену исполнения, отслеживать статусы исполнения ордера, в том числе отмену, и т.д. Основные статусы следующие: Submitted, Accepted, Completed, Canceled, Rejected, Margin. Мы реализуем этот метод чуть позже.

- Метод notify_trade() срабатывает каждый раз, когда изменяется статус трейда. Например, если трейд (вход в рынок и выход из него) был успешно совершен, можно записать прибыль или убыток в соответствующие списки. Мы также реализуем этот метод чуть позже.

- Затем мы перегрузили метод next(), который отслеживает появление новой свечи, и вывели данные о свечах на экран. Именно в этой функции происходит большая часть логики по открытию и закрытию позиций. Обратите внимание, после того как мы подготовили более лаконичное отображение свойств в конструкторе__init__(), внутри метода next() к данным свечей мы получаем доступ через индекс.

Индекс 0 означает текущую свечу, в свою очередь индекс *-1* означает предыдущую и так далее. Например, в строке

```
print("Close", self.close[0])
```

мы получим доступ к цене закрытия сегодняшнего дня, а в строке `print("Close", self.close[-1])` — к вчерашней цене закрытия. Благодаря этому мы можем отслеживать откаты или определять свечные разворотные формации.

- Метод stop() срабатывает по завершении бэктеста и позволяет провести базовую аналитику, например, учесть процент прибыльных сделок. Мы реализуем его в финале.

В конце файла мы объявили класс MyCSVData(), в котором описали формат для подгрузки исторических данных, следом сразу создали объект data, который впоследствии передадим в Бэктрейдер.

- Затем в строке `datapath = abspath("./data/BTCUSDT_D.csv")` мы указали, откуда Бэктрейдер возьмет данные для загрузки свечей. Перед тем как подгрузить данные в Бэктрейдер, на базе класса MyCSVData() мы создали объект для подключения потока свечных данных.
- Строка `cerebro = Cerebro()` говорит о том, что мы практически готовы запустить Бэктрейдер. Объект cerebro — это сердце и мозг Бэктрейдера. Именно в этом объекте происходит вся внутренняя работа.
- Подключение источника свечных данных произошло в строке `cerebro.adddata(data)`. После подключения в классе CandlesOnly() эти данные становятся доступны в свойстве self.data или в self.datas[0]. Выбирайте любой удобный формат.

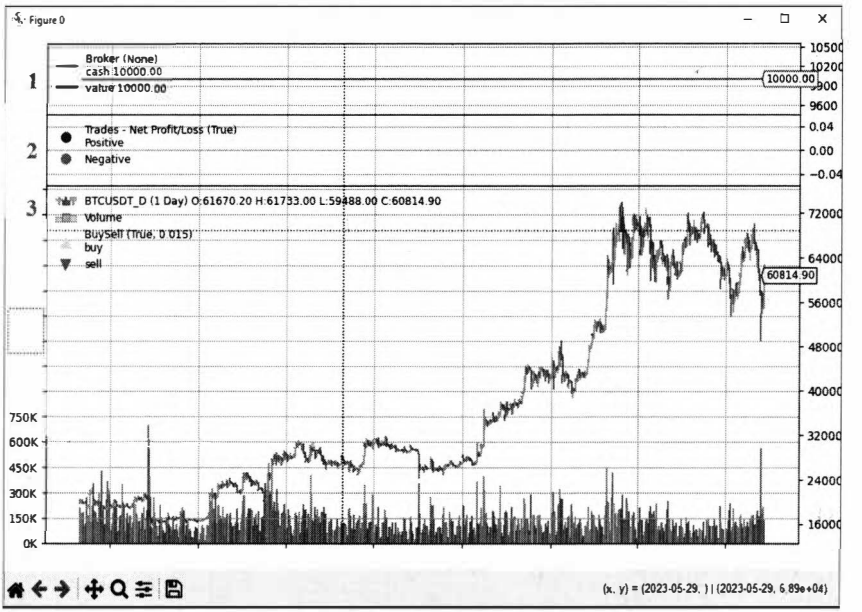
Поскольку в Бэктрейдер можно передать одновременно несколько источников данных, например свечи LTC и BTC, обращаться к ним можно через self.data и self.data1 соответственно. Если вы выбрали формат self.datas[0], то индекс необходим для того, чтобы получить доступ к первому списку данных. Технически можно передать и отслеживать хоть весь рынок. Каждый новый источник данных будет иметь соответствующий ID.

В строке `cerebro.addstrategy(CandlesOnly)` добавляем стратегию CandlesOnly. Обратите внимание, что, поскольку в нашей стратегии объявлен параметр, его можно передать при вызове и заместить стандартное значение: `cerebro.addstrategy(CandlesOnly, ExitCndles=10)`. Обратите внимание, что мы НЕ создаём объект, а просто передаём имя класса. Бэктрейдер под капотом самостоятельно разберётся с созданием объекта.

В строке `cerebro.broker.setcash(10000.0)` устанавливаем стартовый баланс. От этого зависит немало. Например, бэктесты на 100 долларов, 1000 долларов и 10 000 долларов могут быть совершенно разными.

В строке `cerebro.addsizer(sizers.FixedSize, stake=0.01)` мы устанавливаем, каким лотом будем входить в позицию. Например, если торгуем BTC и указываем stake=1, то позиция откроется на 1 Биткойн.

Комиссия, установленная в строке `cerebro.broker.setcommission (commission=0.0003)`, будет вычитаться как при входе в позицию, так и при выходе из неё. Рассчитывается до сотых. Если у брокера комиссия 0.3, то вписывать надо 0.003. Далее запускаем бэктест с помощью строки `cerebro.run()` и отрисовываем данные на графике с помощью строки `cerebro.plot()`. Если вы всё сделали правильно, то после запуска исходника в появившемся окне увидите примерно следующее:



Изображение 4.7.

Структура окна Плота:

1. Часть для визуализации объёма средств, находящихся в позиции.
2. Позитивные и негативные трейды.
3. График и точки входа.

В консоли будут просто отображаться данные при появлении новой свечи.

4.8. Первая сделка. Прописываем условия входа в рынок

Теперь добавим логику открытия ордера.

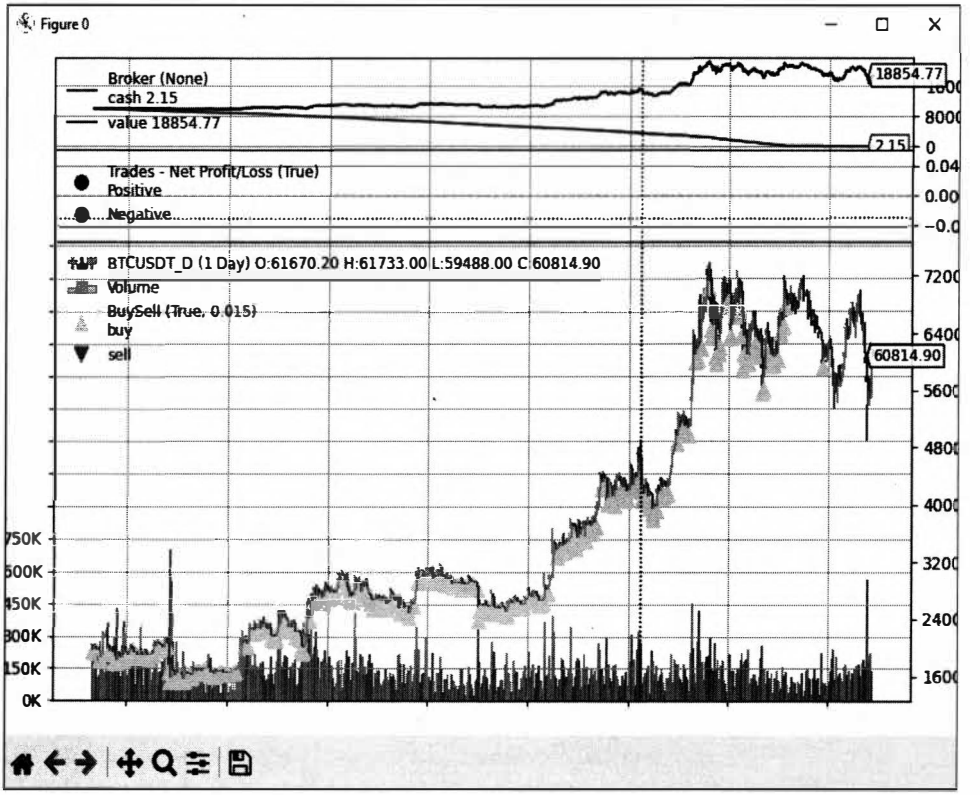
В метод next() допишите следующий код:

```
def next(self):
    # Появление новой свечи и логика стратегии
    # Индекс используется для получения бара
    # Индекс 0, это текущая свеча, если -1 то предыдущая
    # Выводим значения свечей. Лежат в CSV файле
    print("Datetime", self.time[0])
    print("Open", self.open[0])
    print("Close", self.close[0])
    print() # Разделитель между свечами

    # Если цена сегодня была меньше чем вчера и позавчера
    is_buy = self.close[0] < self.close[-1] and self.close[-2]

    if is_buy: # Если в is_buy находится True
        print("Совершаю покупку")
        self.order = self.buy() # Покупаем
```

Мы добавили простую логику по отслеживанию свечей.

Звучит она так:

Звучит она так: если цена закрытия текущего бара, который находится под индексом 0, меньше, чем вчерашняя цена и позавчерашняя, — входим в позицию и сохраняем ордер в свойстве *self.order*.



Изображение 4.8.

Обратите внимание, что раздел с трейдами не заполнен, поскольку мы просто покупаем на откатах, ничего не продавая, соответственно, трейдов не было.

А вот на части с графиком появились точки входа в рынок.

4.9. Отслеживаем ордер и его статусы

Напомню, что для выхода из сделки мы будем использовать удержание некоторого количества свечей. Для этого нам понадобится отслеживание, на какой свече была исполнена заявка на покупку.

После того как мы зафиксируем свечу, на которой была исполнена сделка, мы легко сможем отследить любое количество свечей после неё и закрыть позицию.

Для работы с отслеживанием статуса ордеров существует специальный метод notify_order(), который позволяет реагировать каждый раз, когда статус ордера изменён.

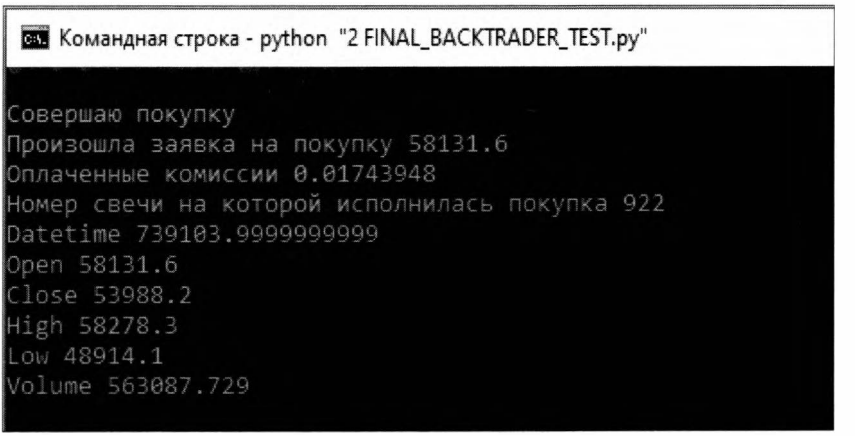
Добавьте в метод notify_order() следующий код:

```
def notify_order(self, order):
    # Срабатывает когда изменяется статус ордера
    # Если заявка принята, но не исполнена
    if order.status in [order.Submitted, order.Accepted]:
        return # Выходим и дальше не продолжаем

    if order.status in [order.Completed]: # Если заявка исполнена
        if order.isbuy(): # Если заявка была на покупку
            print("Произошла заявка на покупку", order.executed.price)
            print("Оплаченные комиссии", order.executed.comm)
            # Номер свечи на которой была исполнена заявка
            self.bar_executed = len(self)
            print("Номер свечи на которой исполнилась покупка",
                self.bar_executed)
        elif order.issell(): # Если заявка была на продажу
            print("Произошла заявка на продажу", order.executed.price)
            print("Оплаченные комиссии", order.executed.comm)
    # Если заявка отклонена, нет средств, отклонена брокером
    elif order.status in [order.Canceled, order.Margin, order.Rejected]:
        print("order.Canceled, order.Margin, order.Rejected")
    self.order = None # Сбрасываем ордер поскольку он отработал
```

Мы написали обработчик для каждого статуса ордера. Если произошел ордер на покупку, то выводится информация о цене и комиссии за вход.

В строке `self.bar_executed = len(self)` мы зафиксировали свечу, на которой произошла покупка ордера.

В случае если ордер был по каким-то причинам отклонен, то появляется уведомление об этом. В самом конце в строке `self.order = None` мы сбросили ордер, так как он отработал. Произошёл трейд, и предыдущий ордер нам не нужен. После запуска в своём терминале вы увидите примерно следующее:



Изображение 4.9.

4.10. Стратегия выхода из сделки

После того как мы зафиксировали, на какой свече произошла покупка, предположим, что через 5 дней пора фиксировать прибыль или убыток.

Для этого приведем метод next() к следующему виду:

```
def next(self):
    # Появление новой свечи и логика стратегии
    # Индекс используется для получения бара
    # Индекс 0, это текущая свеча, если -1 то предыдущая
    if self.order: # Если есть НЕ исполненный ордер
        return # Выходим и дальше не продолжаем

    if not self.position: # Если позиции нет
        # Проверяем условия входа
        # Если цена сегодня была меньше чем вчера,
```

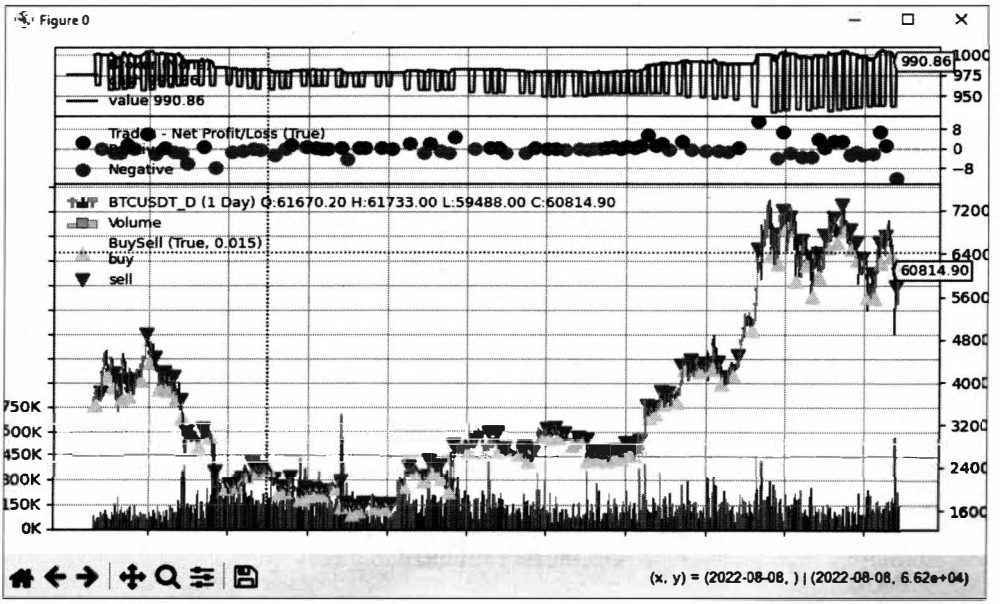
```
        # Если цена сегодня была меньше чем вчера и позавчера
        is_buy = self.close[0] < self.close[-1] and self.close[-2]

        if is_buy: # Если в is_buy находится True
            print("Совершаю покупку")
            self.order = self.buy() # Заявка на покупку 1 позиции
    else: # Если позиция есть
        # Прошло не менее 5-и свечей с момента входа
        is_sell = len(self) - self.bar_executed >= self.params.
ExitCandles
        if is_sell: # Если в is_sell находится True
            print("Совершаю продажу")
            self.order = self.sell() # Заявка на продажу 1 позиции
```

Мы добавили несколько дополнительных проверок, чтобы пропускать свечу, если есть неисполненный ордер, а также добавили логику отслеживания позиции.

Дословно в строке

```
is_signal_cell = len(self) - self.bar_executed >= self.params.ExitBars
```

мы вычитаем из текущей свечи 5 дней и проверяем, больше получившееся значение, чем зафиксировано при открытии, или нет. Вкратце, выходим из позиции на 6-й день.



Изображение 4.10.

Теперь все части Плота заполнены, мы видим прямую доходности, трейды и места, где были открыты и закрыты ордера.

4.11. Метод notify_trade(). Фиксируем результаты каждого трейда

Теперь, поскольку добавлено условие выхода из сделки, пора отследить прибыльные или убыточные трейды.

Опишите метод notify_trade(), как указано ниже.

```
def notify_trade(self, trade):
    # Срабатывает когда произошёл трейд
    if trade.isclosed: # Если трейд произошёл
        # Запоминаем прибыли по сделкам
        # информируем об итогах каждой
        if trade.pnlcomm > 0: # Если сумма прибыли больше 0
            # Добавляем в профит
            self.win_trades.append(trade.pnlcomm)
            print("Профит", trade.pnlcomm)
        else: # Если сумма прибыли меньше 0
            # Добавляем в убытки
            self.lose_trades.append(trade.pnlcomm)
            print("Потеря", trade.pnlcomm)
        print() # Разделитель между трейдами
```

Теперь при каждом трейде мы фиксируем прибыль или убыток и добавляем финальную сумму трейда в соответствующий список. Благодаря этому после совершения каждой сделки можно увидеть, с прибылью или убытком закрылась сделка, а также рассчитать показатель винрейта.

4.12. Метод stop(). Добавляем показатель винрейта

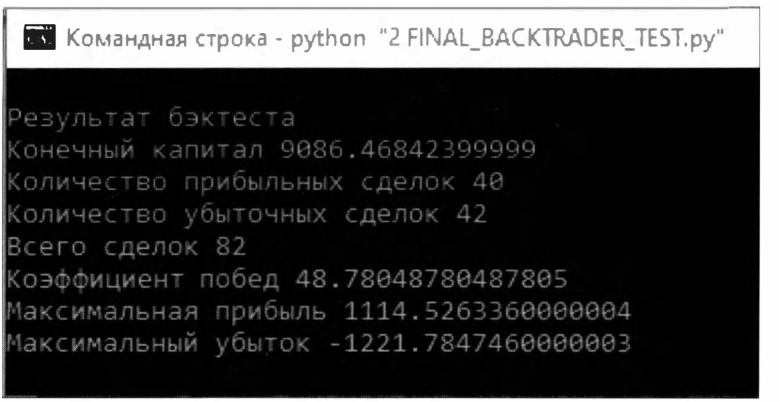
Очень сильно хочется знать процент проходимости сделок. Сколько максимально было заработано с одного трейда, сколько потеряно.

Для этого мы уже добавили два свойства self.win_trades и self.lose_trades в метод __init__(). Эти свойства содержат суммы прибыли или убытка для каждого трейда. Опишите метод stop(), как представлено ниже.

```
def stop(self):
    # Срабатывает после завершения бэктеста
    print("Результат бэктеста")
    print("Конечный капитал", self.broker.getvalue())
    print("Количество прибыльных сделок", len(self.win_trades))
    print("Количество убыточных сделок", len(self.lose_trades))
    all_trade_len = len(self.win_trades) + len(self.lose_trades)
    win_rate = (len(self.win_trades) / all_trade_len) * 100
    print("Всего сделок", all_trade_len)
    print("Коэффициент побед", win_rate)
    print("Максимальная прибыль", max(self.win_trades))
    print("Максимальный убыток", min(self.lose_trades))
```

Мы подробно расписали, сколько было прибыльных сделок, сколько убыточных сделок, рассчитали коэффициент выигрышных сделок, вывели максимальную прибыль, максимальный убыток и теперь имеем гораздо больше информации.

Запустите исходный код и посмотрите на результат:



Изображение 4.11.

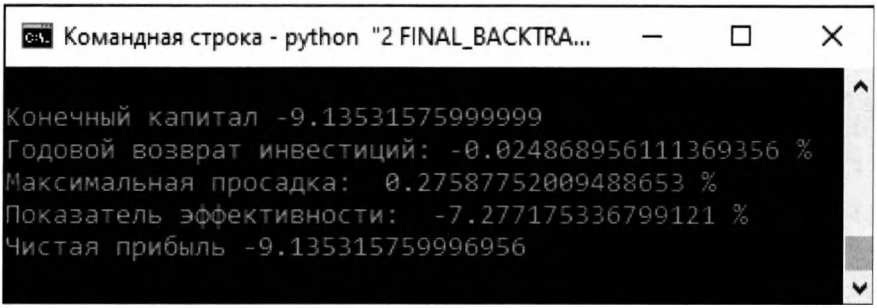
4.13. Аналитика бэктеста с помощью Cerebro

Аналитика доступна не только внутри стратегии в методе stop(), но и в Церебро. Вы вольны использовать тот способ, который вам приятнее. В качестве демонстрации возможности работать с анализаторами после создания объекта со свечными данными допишите следующий код:

```
# Подключение источника данных, аналитика и запуск бэктеста
cerebro = Cerebro() # Создаём объект Церебро. Мозг Бектрейдера
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data
cerebro.addstrategy(CandlesOnly) # Добавляем стратегию
cerebro.broker.setcash(10000.0) # Устанавливаем стартовый баланс
cerebro.addsizer(sizers.FixedSize, stake=0.1) # Размер позиции
cerebro.broker.setcommission(commission=0.0003) # Комиссия брокера

# Подключаем аналитику стратегии с помощью Церебро
start_capital = cerebro.broker.getvalue() # Баланс до старта
cerebro.addanalyzer(analyzers.Returns) # Годовой возврат инвестиций
cerebro.addanalyzer(analyzers.DrawDown) # Максимальная просадка
cerebro.addanalyzer(analyzers.SharpeRatio) # Эффективность
cerebro.addanalyzer(analyzers.TradeAnalyzer) # Анализатор трейда

result = cerebro.run() # Запускаем бэктест и сохраняем результат

# Извлекаем аналитику после завершения бэктеста
final_bank = result[0].analyzers.tradeanalyzer.get_analysis()["pnl"]
                                                    ["net"]["total"]

year_return = result[0].analyzers.returns.get_analysis()["rnorm100"]

max_drow_down = result[0].analyzers.drawdown.get_analysis()["max"]
                                                          ["drawdown"]

sharpe_ratio = result[0].analyzers.sharperatio.get_analysis()
                                                       ["sharperatio"]

print("Конечный капитал", final_bank)
print("Годовой возврат инвестиций:", year_return, "%")
print("Максимальная просадка: ", max_drow_down, "%")
print("Показатель эффективности: ", sharpe_ratio, "%")
print("Чистая прибыль", cerebro.broker.getvalue() - start_capital)
cerebro.plot(style="candle") # Рисуем свечной график
```

Запустите исходник.

В консоли вы должны увидеть примерно следующее:



Изображение 4.12.

Поздравляю! Мы реализовали базовый скелет для любой стратегии. Познакомились с основными компонентами Бэктрейдера и теперь готовы погрузиться в частные детали.

4.14. Индикаторы

Я большой любитель пользоваться исключительно свечным анализом при построении автоматизированных торговых стратегий, любой индикатор работает задним числом, и бэкстесты могут быть очень красивы, но реальные результаты могут НЕ порадовать. Тем не менее, при грамотном применении индикаторы могут послужить отличным фильтром для визуализации данных. Например, благодаря бэктесту можно выстроить прибыльную стратегию ручной торговли.

В демонстрационных примерах протестируем как самые популярные классические индикаторы, так и специфические, такие как ATR и свечи Хайкин-Аши. В том числе попытаемся выстроить стратегию, основанную на комбинировании нескольких из них.

4.15. Трендовая стратегия по одной SMA

Первым добавим индикатор, который по праву является одним из фундаментальных, надежных и проверенных временем. SMA (скользящая сред-

няя) позволяет визуализировать текущее направление рынка и находить потенциальные сломы тренда.

Немало успешных частных инвесторов используют только одну SMA с показателем 200 (на дневных графиках) для принятия долгосрочных инвестиционных решений. По их мнению, если цена находится выше линии индикатора, значит тренд продолжается и можно продолжать закупать актив.

Создайте файл SMABacktest.py и перепишите следующий код:

Листинг 4.2

```
from os.path import abspath # Функция для работы с путями
from datetime import datetime # Объект для работы со временем
from backtrader import Cerebro, Strategy, sizers, analyzers # Компоненты
from backtrader.feeds import GenericCSVData # Класс для загрузки CSV
from backtrader.indicators import SimpleMovingAverage # Импорт SMA

class SingleSma(Strategy):
    # Параметры торговой системы в виде словаря
    params = (
        "SMAperiod": 39, # Период дней для расчёта СМА
    )

    def __init__(self):
        # Инициализация стратегии и временных рядов
        # В self.data хранится временной ряд первого источника данных
        self.close = self.data.close # Цены закрытия
        self.sma = SimpleMovingAverage(self.data,
            period=self.params.SMAperiod) # Подключаем SMA
        self.order = None # Храним ордер

    def notify_order(self, order):
        # Срабатывает когда изменяется статус ордера
        # Если заявка принята, но не исполнена
        if order.status in [order.Submitted, order.Accepted]:
            return # Выходим и дальше не продолжаем

        if order.status in [order.Completed]: # Если заявка исполнена
            if order.isbuy(): # Если заявка была на покупку
                print("Произошла заявка на покупку", order.executed.price)
            elif order.issell(): # Если заявка была на продажу
                print("Произошла заявка на продажу", order.executed.price)
        # Если заявка отклонена, нет средств, отклонена брокером
        elif order.status in [order.Canceled, order.Margin, order.Rejected]:
```

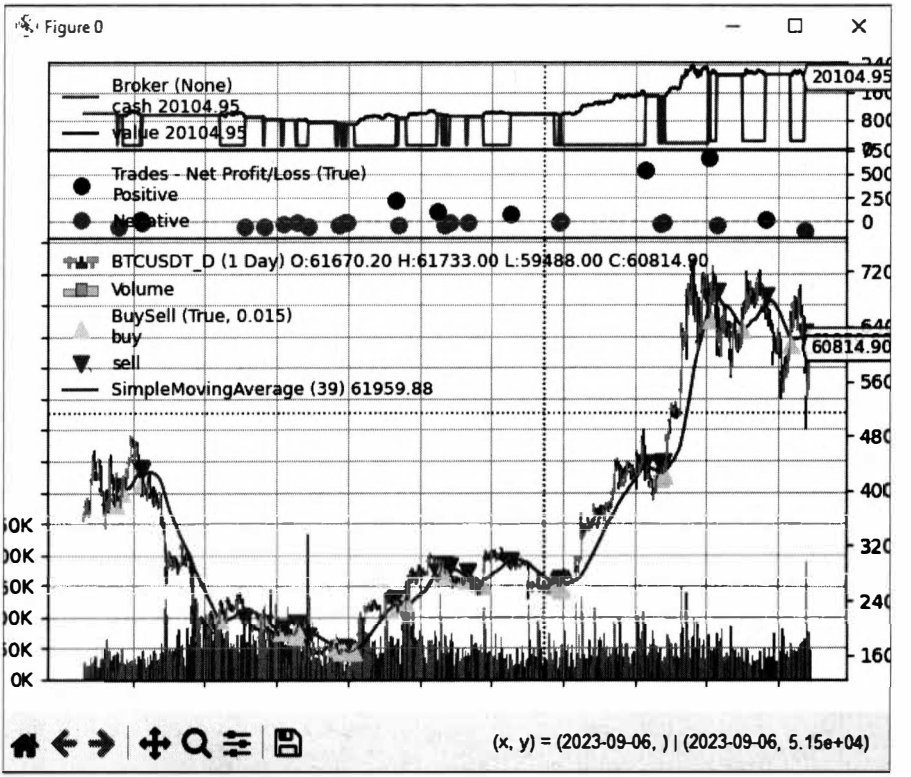
```
            print("order.Canceled, order.Margin, order.Rejected")
        self.order = None # Сбрасываем ордер поскольку он отработал

    def notify_trade(self, trade):
        # Срабатывает когда произошёл трейд
        print("Произошёл трейд", trade.pnlcomm) # Финальная сумма с трейда
        if self.order: # Если есть НЕ исполненный ордер
            return # Выходим и дальше не продолжаем

    def next(self):
        # Появление новой свечи и логика стратегии
        # Индекс используется для получения бара
        # Индекс 0, это текущая свеча, если -1 то предыдущая

        if not self.position: # Если позиции нет
            is_buy = self.close[0] > self.sma[0] # Цена закрылась выше СМА
            if is_buy: # Если в is_buy находится True
                print("Совершаю покупку")
                self.order = self.buy() # Заявка на покупку 1 позиции
        else: # Если позиция есть
            is_sell = self.close[0] < self.sma[0] # Цена закрылась ниже СМА
            if is_sell: # Если в is_sell находится True
                print("Совершаю продажу")
                self.order = self.sell() # Заявка на продажу 1 позиции

    def stop(self):
        # Срабатывает после завершения бэктеста
        print("Результат бектеста")
        print("Параметр индикатора SMA:", self.params.SMAperiod)
        print("Конечный капитал:", self.broker.getvalue())

# Подготовка источника свечных данных
# MyCSVData основной класс для загрузки CSV файла
class MyCSVData(GenericCSVData):
    # В параметрах указываем какой столбец CSV файла
    # за какие данные отвечает
    params = (
        "dtformat": "%Y-%m-%d", # Формат времени для парсинга
        "datetime": 0,
        "open": 1,
        "high": 2,
        "low": 3,
        "close": 4,
        "volume": 5,
        "openinterest": -1) # -1 это последний столбец
csv_file_path = abspath("./data/BTCUSDT_D.csv") # Путь к источнику данных
data = MyCSVData( # Объект с источником свечных данных
```

```
    dataname=csv_file_path,
    fromdate=datetime(2022, 1, 24),
    todate=datetime(2024, 8, 10),
    reverse=False)

# Подключение источника данных, аналитика и запуск бэктеста
cerebro = Cerebro() # Создаём объект Церебро. Мозг Бектрейдера
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data
cerebro.addstrategy(SingleSma, SMAperiod=39) # Добавляем стратегию
cerebro.broker.setcash(10000.0) # Устанавливаем стартовый баланс
cerebro.addsizer(sizers.PercentSizer, percents=90) # Позиция в %
cerebro.broker.setcommission(commission=0.0003) # Комиссия брокера

# Подключаем аналитику стратегии с помощью Церебро
start_capital = cerebro.broker.getvalue() # Фиксируем баланс до старта
result = cerebro.run() # Запускаем бэктест и сохраняем результат
# Извлекаем аналитику после завершения бэктеста
print("Чистая прибыль", cerebro.broker.getvalue() - start_capital)
cerebro.plot(style="candle") # Рисуем свечной график
```

- После импорта объектов и функций мы объявили класс стратегии SingleSma с параметром SMAPeriod.
- В методе __init__() извлекли цены закрытия свечей и добавили свойство для хранения ордера.
- Встроке `self.sma = SimpleMovingAverage(self.data, period=self.params.SMAperiod)` подключили индикатор SMA к источнику данных, хранящихся в self.data.
- В методе next() описали условия входа и выхода из сделки.
- Строка `is_buy = self.close[0] > self.sma[0]` отслеживает, закрылась ли цена выше SMA.
- Строка `is_sell = self.close[0] < self.sma[0]` отслеживает, закрылась ли цена ниже SMA.
- В конце файла подключили источник данных и вывели минимально необходимую аналитику.

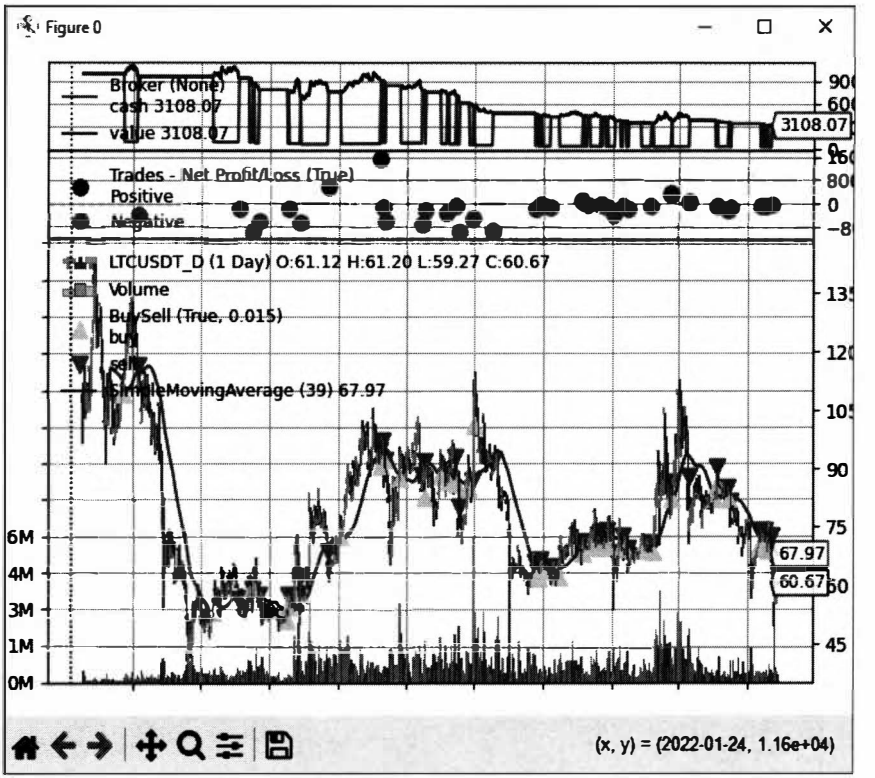
Ниже представлены выходные данные для BTC и LTC.

```
Результат бэктеста BTC:
Стартовый капитал: 10000
Конечный капитал 20104.94868125274
Годовой возврат инвестиций: 20.931748664394334 %
Максимальная просадка: 36.195148405066035 %
Чистая прибыль 10104.94868125274
```



Изображение 4.13.

```
Результат бэктеста LTC:
Стартовый капитал: 10000
Конечный капитал 3108.0739529514194
Годовой возврат инвестиций: -27.16611150655817 %
Максимальная просадка: 72.03437412414188 %
Чистая прибыль -6891.926047048581
```



Изображение 4.14.

4.16. Осцилляторы

Термин осциллятор берет свое начало от латинского *oscillare* и означает "качаться". Впервые осцилляторы появились примерно в 30-х годах. Осцилляторы используются даже в физике.

Биржевые осцилляторы помогают трейдерам находить зоны перепроданности и перекупленности, а также следить за направлением тренда.

Большая часть осцилляторов основана на двух принципах:

1. Измерение динамики движения цены.
2. Измерение ценового диапазона.

Измерение динамики напоминает процесс спуска или подъема по лестнице с закрытыми глазами. Если каждые 10 шагов вы оказываетесь на несколько ступеней выше, значит идёт восхождение. Аналогичен процесс нисхождения. Если за 10 шагов вы не поднялись ни на одну ступень, значит достигли этажа.

4.17. Стохастический осциллятор

Один из самых популярных осцилляторов Stochastic изобрел Джордж Лэйн в конце 50-х годов.

> Стохастик берет текущую цену актива и вычитает количество предыдущих дней, которые заданы в параметрах индикатора (по умолчанию это 14 дней). Затем высчитывает процентное соотношение между открытием и закрытием (от 0 до 100). Благодаря этим расчётам можно отследить динамику цены и потенциальные зоны разворота рынка.

По умолчанию зоной, когда заканчивается нисходящее движение, считается диапазон от 0% до 20%. Зоной, когда заканчивается восходящее движение от 80% до 100%. На большинстве активов цена может некоторое время задержаться в одной из зон, и многие теряют деньги, поскольку не дожидаются выхода из зоны.

Лично мне по душе Стохастик. Практически всегда, для большей уверенности в сделке, использую его зону перепроданности в качестве финального подтверждения для входа в лонговую позицию. Не забывайте, что важен тест на историю конкретного инструмента, прежде чем начать пользоваться индикатором в живой или автоматической торговле. Например, при торговле LTC для фильтрации лонговых сделок Стохастик

отрабатывает просто потрясающе (на дневных свечах). На BTC ситуация не такая радужная и, на мой взгляд, Стохастик не особо пригоден для торговли этим активом.

Очень часто можно увидеть видео, где показывают торговлю только по Стохастику. Конечно же, без бэктеста такой подход не является достоверным. Давайте убедимся, будет ли торговля только по Стохастику прибыльной или убыточной.

Создайте файл STOHBacktest.py и перепишите следующий код:

Листинг 4.3

```
from os.path import abspath # Функция для работы с путями
from datetime import datetime # Объект для работы со временем
from backtrader import Cerebro, Strategy, sizers # Компоненты БТ
from backtrader.feeds import GenericCSVData # Основной класс для CSV
from backtrader.indicators import Stochastic # Импорт индикаторов

class StochasticStrategy(Strategy): # Наследуемся от класса Стратегий
    params = ( # Параметры индикатора Стохастик
        "period": 14, # Количество дней для расчёта
        "pfast": 3, # Длинная скользящая
        "pslow": 3, # Короткая скользящая
        "upperLimit": 80, # Зона перекупленности
        "lowerLimit": 20 # Зона перепроданности
    )

    def __init__(self): # Инициализация стратегии и временных рядов
        self.order = None # Храним ордер
        self.executed_price = None # Цена исполненного ордера

        self.close = self.data.close # Цена закрытия свечи

        # Инициализируем Стохастик с помощью параметров
        self.stochastic = Stochastic(self.data, # Источник данных
            period=self.params.period, # Период дней
            period_dfast=self.params.pfast, # Быстрая скользящая
            period_dslow=self.params.pslow, # Медленная скользящая
            upperband=self.params.upperLimit, # Зона перекупленности
            lowerband=self.params.lowerLimit # Зона перепроданности
        )

    def notify_order(self, order):
        # Срабатывает, когда изменяется статус ордера
        # Если ордер принят, то ничего не делаем
```

```
        if order.status in [order.Submitted, order.Accepted]:
            return # Выходим и дальше не продолжаем

        # Если ордер исполнен и произошла покупка
        if order.status == order.Completed:
            if order.isbuy():
                # Фиксируем цену входа
                self.executed_price = order.executed.price
        # Если произошла отмена ордера или недостаток маржи
        elif order.status in [order.Rejected, order.Margin]:
            print("Ордер отменен")
        self.order = None # Сбрасываем ордер после работы

    def next(self): # Появление новой свечи и логика стратегии
        print("Текущий период стохастика", self.stochastic.lines.percD[0])

        if self.order: # Если есть неисполненный ордер
            return # Выходим и дальше не продолжаем


        if not self.position: # Если позиции нет
            # Проверяем, является ли линия Стохастика ниже или равной 25
            is_signal_buy = self.stochastic.lines.percD[0] <= 25
            if is_signal_buy: # Если пришёл сигнал на покупку
                print("Совершаю покупку")
                self.order = self.buy() # Покупаем
        else: # Если позиция уже есть, проверяем условие для выхода
            print("Цена открытия сделки:", self.executed_price)
            print("Цена закрытия текущей свечи:", self.close[0])


            # Формула расчёта разницы между суммами c=(b-a)/a*100
            down_percent = (self.close[0]-self.executed_price) / self.close[0] * 100
            print("Разница цены в процентах:", down_percent)

            if down_percent <= -15: # Максимальная просадка
                print("Сработал стоп при разнице процентов:", down_percent)
                self.order = self.sell() # Продаём позицию
                return # Дальше не идем, ждём следующей свечи

            # Если НЕ сработал стоп, проверяем, является ли Стохастик
            # выше или равным 70
            is_signal_cell = self.stochastic.lines.percD[0] >= 70
            if is_signal_cell:
                print("Совершаю продажу")
                self.order = self.sell() # Продаём
        return # Завершаем метод next()
```

```
# Подготовка источника свечных данных
# MyCSVData — основной класс для загрузки CSV-файла
class MyCSVData(GenericCSVData):
    # В параметрах указываем, какой столбец CSV-файла
    # за какие данные отвечает
    params = (
        "dtformat": "%Y-%m-%d", # Формат времени для парсинга
        "datetime": 0,
        "open": 1,
        "high": 2,
        "low": 3,
        "close": 4,
        "volume": 5,
        "openinterest": -1, # -1 — это последний столбец
    )

csv_file_path = abspath("./data/LTCUSDT_D.csv") # Путь к источнику данных
data = MyCSVData( # Объект с источником свечных данных
    dataname=csv_file_path,
    fromdate=datetime(2022, 1, 24),
    todate=datetime(2024, 8, 10),
    reverse=False)

# Подключение источника данных, аналитика и запуск бэктеста
cerebro = Cerebro() # Создаём объект Церебро. Мозг Бэктрейдера
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data
cerebro.addstrategy(StochasticStrategy) # Добавляем стратегию
cerebro.broker.setcash(10000.0) # Устанавливаем стартовый баланс
cerebro.addsizer(sizers.PercentSizer, percents=85) # Размер позиции
cerebro.broker.setcommission(commission=0.0003) # Комиссия брокера
start_capital = cerebro.broker.getvalue() # Фиксируем баланс до старта
cerebro.run() # Запускаем бэктест и сохраняем результат

print("Конечный капитал:", cerebro.broker.getvalue())
print("Чистая прибыль:", cerebro.broker.getvalue() - start_capital)
cerebro.plot(style="candle") # Рисуем свечной график
```

В начале исходника мы, как всегда, импортируем необходимые библиотеки, а также индикатор Стохастик. Сразу после объявления класса стратегии устанавливаем необходимые для инициализации Стохастика параметры.

В свойствах метода __init__() устанавливаем все объекты, необходимые для работы стратегии, в частности, инициализируем индикатор Стохастик.

В методе notify_order() отслеживаем статус ордера и цену, по которой зашли в позицию. Отслеживание цены, по которой была исполнена позиция, позволит вам создать условия для срабатывания стоп-лосса.

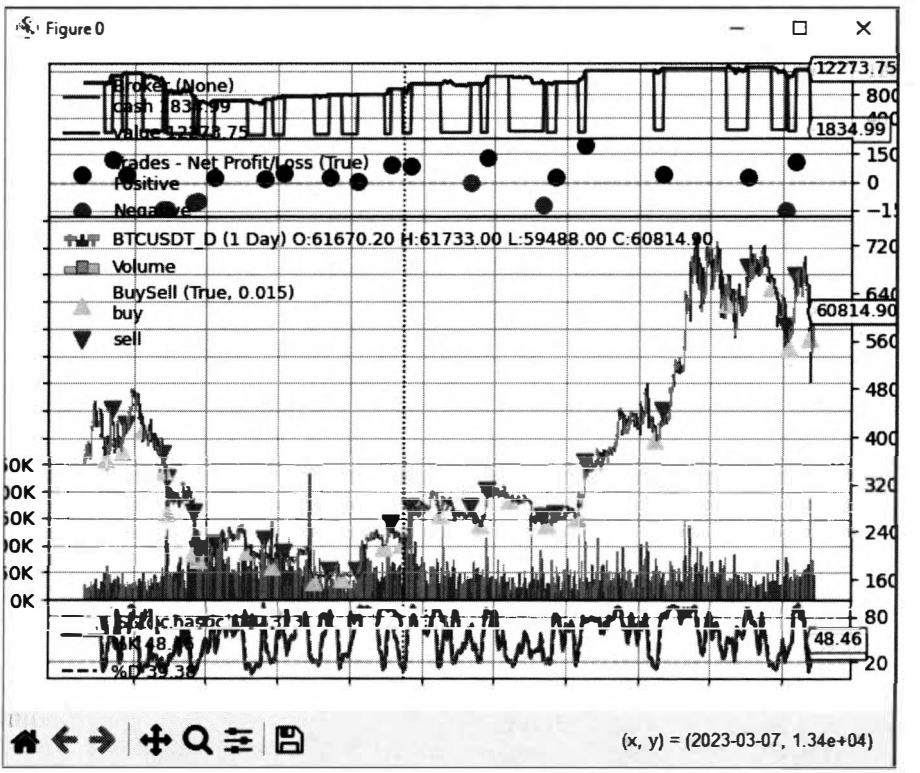
В методе next() в первую очередь печатаем на экран, в какой зоне находится индикатор Стохастик. Далее, если открытой позиции нет, проверяем, является ли Стохастик ниже или равным 25. Если да, то заходим в позицию. В случае если позиция открыта, проверяем условия для выхода. Если Стохастик находится в положении 70 или выше, закрываем позицию.

В строке `if down_percent <= -15:` мы сделали простейший риск-менеджмент и аварийно закроем позицию, если цена просядет на 15 процентов и более.

В строке `cerebro.addsizer(sizers.PercentSizer, percents=90)` мы указали НЕ фиксированный размер лота, а процент от доступного капитала.

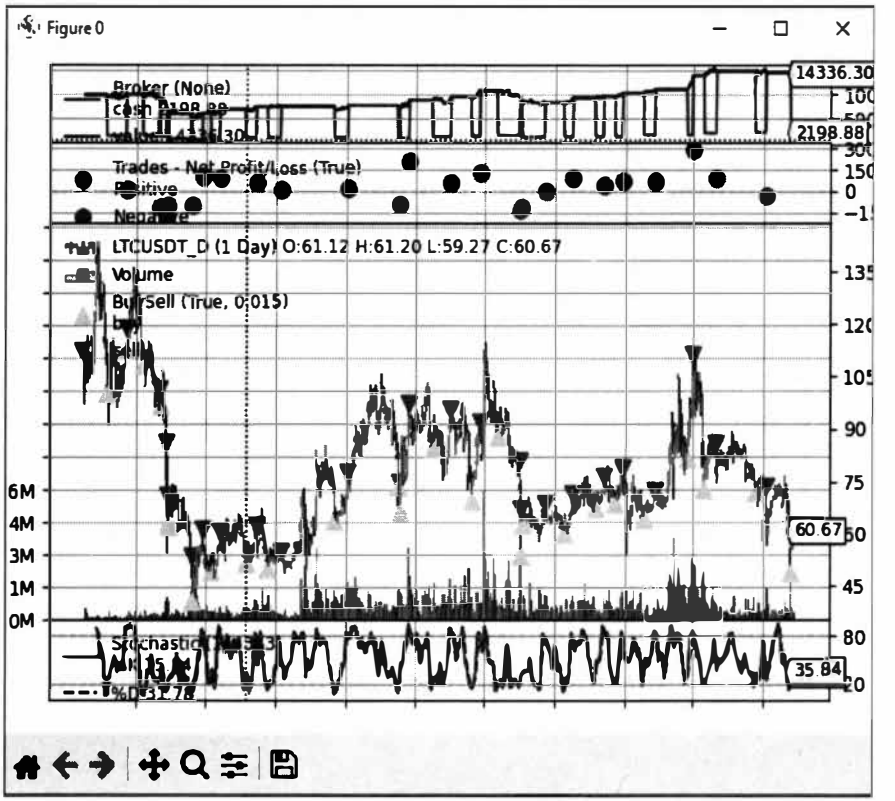
Ниже представлены выходные данные для BTC и LTC.

```
Результат бэктеста BTC:
Стартовый капитал: 10000
Конечный капитал: 12273.752522477502
Чистая прибыль: 2273.7525224775018
```



Изображение 4.15.

```
Результат бэктеста LTC:
Стартовый капитал: 10000
Конечный капитал 14336.302236566804
Чистая прибыль 4336.302236566804
```



Изображение 4.16.

4.18. Индикатор RSI. Отслеживаем пересечение сигнальных линий

Индикатор относительной силы RSI был разработан Уэлсом Уайлдером в 1978 году и благодаря своей простоте плотно вошёл в обиход участников рынка, особенно среди инвесторов.

RSI используется как для отслеживания тренда, так и для определения разворотов.

RSI использует усреднение как положительных, так и отрицательных движений цены, применяя экспоненциальное сглаживание.

Создайте новый файл RSIBacktest.py и перепишите следующий код:

Листинг 4.4

```
from os.path import abspath # Функция для работы с путями
from datetime import datetime # Объект для работы со временем
from backtrader import Cerebro, Strategy, sizers # Компоненты BT
from backtrader.feeds import GenericCSVData # Основной класс для загрузки CSV
from backtrader.indicators import RSI_EMA, CrossOver # Импорт индикаторов

class RsiStrategy(Strategy): # Наследуемся от класса Стратегий
    params = ( # Параметры индикатора RSI
        "rsi_period": 14, # Период RSI
    )

    def __init__(self): # Инициализация стратегии и временных рядов
        self.rsi = RSI_EMA(self.data, period=self.params.rsi_period)
        # Индикатор для сигналов на вход на базе RSI:
        # 1 - входим лонг, 0 — ждём
        # Пересечение линии перепроданности
        self.enter_long = CrossOver(self.rsi, 20)

        # Индикатор для сигналов на выход из лонга на базе RSI:
        # 1 - выходим, 0 — ждём
        # Пересечение линии перекупленности
        self.exit_long = CrossOver(self.rsi, 80)

    def next(self): # Появление новой свечи и логика стратегии
        if not self.position: # Если нет открытой позиции
            # Проверяем сигнал на вход
            if self.enter_long[0] > 0: # Если сигнал на вход больше 1
                print("Сигнал на вход!")
                self.buy() # Открываем ордер
        else: # Проверяем сигнал на выход из позиции
            if self.exit_long[0] > 0:
                print("Сигнал на выход!")
                self.sell() # Продаём позицию
```

```
# Подготовка источника свечных данных
# MyCSVData основной класс для загрузки CSV файла
class MyCSVData(GenericCSVData):
    # В параметрах указываем какой столбец CSV файла
    # за какие данные отвечает
    params = (
        "dtformat": "%Y-%m-%d", # Формат времени для парсинга
        "datetime": 0,
        "open": 1,
        "high": 2,
        "low": 3,
        "close": 4,
        "volume": 5,
        "openinterest": -1, # -1 игнорирование столбца
    )

csv_file_path = abspath("./data/BTCUSDT_D.csv") # Путь к источнику данных
data = MyCSVData( # Объект с источником свечных данных
    dataname=csv_file_path,
    fromdate=datetime(2022, 1, 24),
    todate=datetime(2024, 8, 10),
    reverse=False)

# Подключение источника данных, аналитика и запуск бэктеста
cerebro = Cerebro() # Создаём объект Церебро. Мозг Бектрейдера
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data
cerebro.addstrategy(RsiStrategy) # Добавляем стратегию
cerebro.broker.setcash(10000.0) # Устанавливаем стартовый баланс
cerebro.addsizer(sizers.PercentSizer, percents=90) # Позиция в %
cerebro.broker.setcommission(commission=0.0003) # Комиссия брокера

# Подключаем аналитику стратегии с помощью Церебро
start_capital = cerebro.broker.getvalue() # Фиксируем баланс до старта
cerebro.run() # Запускаем бэктест
print("Конечный капитал", cerebro.broker.getvalue())
print("Чистая прибыль", cerebro.broker.getvalue() - start_capital)
cerebro.plot(style="candle") # Рисуем свечной график
```

В начале исходника мы, как всегда, импортируем необходимые библиотеки, а также инидкатор RSI_EMA и индикатор пересечения CrossOver.

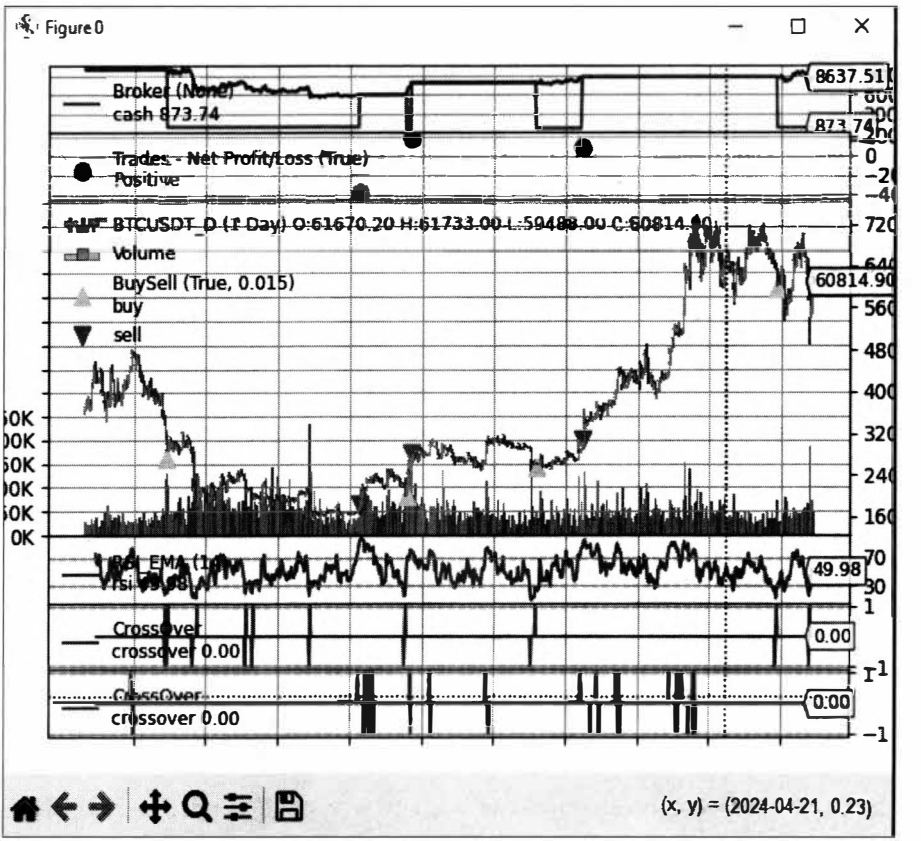
Сразу после объявления класса стратегии в переменной `params` устанавливаем необходимые для инициализации RSI параметры.

В свойствах метода __init__() инициализируем индикатор RSI и индикаторы пересечения нижней сигнальной линии (перепроданности) и верхней (перекупленности).

В методе next() отслеживаем пересечение линии перепроданности для входа в сделку и пересечение линии перекупленности для выхода. Обратите внимание, что мы НЕ использовали сохранение ордера в отдельном свойстве.

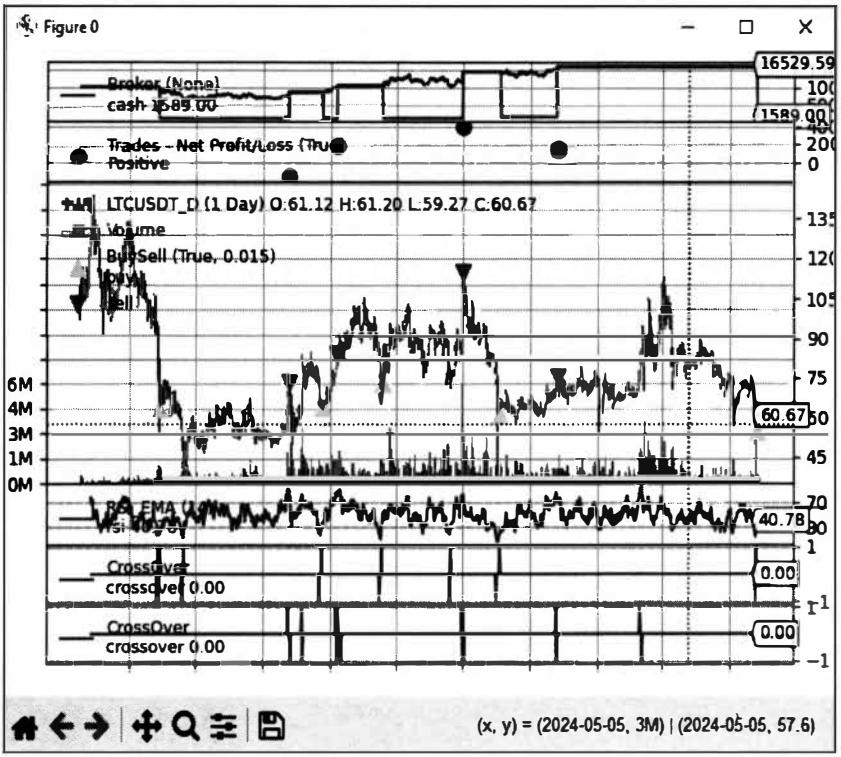
Ниже представлены выходные данные для BTC и LTC.

```
Результат бэктеста BTC:
Конечный капитал 8637.505701391392
Чистая прибыль -1362.494298608608
```



Изображение 4.17.

```
Результат бэктеста LTC:
Конечный капитал 16529.58978228419
Чистая прибыль 6529.589782284191
```



Изображение 4.18.

4.19. Индикатор MACD

Финальным осциллятором, который мы научимся использовать, станет MACD. MACD был создан в середине 70-х годов Джеральдом Аппелем. Джеральд пытался устранить недостатки стратегии по пересечению двух скользящих средних. Во-первых, важно было вычистить ложные пересечения длинной скользящей, во-вторых, устранить запаздывание сигналов.

Напомню, что при уверенном восходящем тренде короткая скользящая средняя находится выше длинной скользящей. При этом короткая скользящая находится в более динамическом движении.

Во время завершения и смены тренда две скользящие сходятся.

Вместо того чтобы полагаться на пересечение двух скользящих, Джеральд решил, что достаточно будет отслеживать их схождение и расхождение и при этом применять экспоненциальное сглаживание.

Если НЕ вдаваться в математические детали, то экспоненциальное сглаживание снижает вес предыдущих значений, и основной акцент делается на последнее значение цены.

MACD состоит из двух основных показателей:

1. Основная линия MACD.
2. Сигнальная линия.

Линия MACD — это разница между 12-дневной и 26-дневной скользящими средними.

Сигнальная линия — это 9-дневная экспоненциальная скользящая средняя линия MACD.

Когда линия MACD закрывается выше сигнальной линии, создаётся сигнал на покупку. Когда линия MACD закрывается ниже сигнальной линии, создаётся сигнал на продажу.

Создайте файл MACDBacktest.py и перепишите следующий код:

Листинг 4.5

```
from os.path import abspath # Функция для работы с путями
from datetime import datetime # Объект для работы со временем
from backtrader import Cerebro, Strategy, sizers # Компоненты BT
from backtrader.feeds import GenericCSVData # Основной класс для CSV
from backtrader.indicators import MACD, CrossOver # Импорт индикаторов
```

```
class MACDStrategy(Strategy): # Наследуемся от класса Стратегий
    params = ( # Параметры индикатора MACD
        "macd1": 12, # 12 дневная скользящая
        "macd2": 26, # 26 дневная скользящая
        "macdsig": 9, # Период сигнальной линии
    )

    def __init__(self): # Инициализация стратегии и временных рядов
        self.order = None # Переменная ордера

        # Устанавливаем индикатор MACD с параметрами
        self.macd = MACD(
            self.data,
            period_me1=self.params.macd1,
            period_me2=self.params.macd2,
            period_signal=self.params.macdsig,
        )
        # Отслеживаем пересечение сигнальной линии
        self.macd_cross = CrossOver(self.macd.macd, self.macd.signal)

    # Отслеживаем трейды
    def notify_trade(self, trade): # Срабатывает когда произошёл трейд
        # Если трейд не завершён
        if not trade.isclosed:
            return # Пропускаем
        self.order = None # Сбрасываем ордер после трейда

    def next(self): # Появление новой свечи и логика стратегии
        print("DateTime", self.data.datetime.datetime(0))
        print("Price", self.data[0], " Mcross", self.macd_cross[0])
        print("Position", self.position.upopened)

        if not self.order:  # Если НЕ находимся в рынке
            # Пересечение над сигнальной линией
            if self.macd_cross[0] > 0.0:
                print("Начинаю покупку")
                self.order = self.buy() # Покупаем
        else:  # Если находимся в рынке
            # Пересечение под сигнальной линией
            if self.macd_cross[0] < 0.0:
                print("Начинаю продажу")
                self.order = self.sell() # Продаём


# Подготовка источника свечных данных
# MyCSVData основной класс для загрузки CSV файла
class MyCSVData(GenericCSVData):
```

```
    # В параметрах указываем какой столбец CSV файла
    # за какие данные отвечает
    params = (
        "dtformat": "%Y-%m-%d", # Формат времени для парсинга
        "datetime": 0,
        "open": 1,
        "high": 2,
        "low": 3,
        "close": 4,
        "volume": 5,
        "openinterest": -1, # -1 это последний столбец
    )

csv_file_path = abspath("./data/LTCUSDT_D.csv") # Источник данных
data = MyCSVData( # Объект с источником свечных данных
    dataname=csv_file_path,
    fromdate=datetime(2022, 1, 24),
    todate=datetime(2024, 8, 10),
    reverse=False)


# Подключение источника данных, аналитика и запуск бэктеста
cerebro = Cerebro() # Создаём объект Церебро. Мозг Бектрейдера
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data
cerebro.addstrategy(MACDStrategy) # Добавляем стратегию
cerebro.broker.setcash(10000.0) # Устанавливаем стартовый баланс
cerebro.addsizer(sizers.PercentSizer, percents=85) # Позиция в %
cerebro.broker.setcommission(commission=0.0003) # Комиссия брокера
start_capital = cerebro.broker.getvalue() # Фиксируем баланс до старта
cerebro.run() # Запускаем бэктест

print("Конечный капитал:", cerebro.broker.getvalue())
print("Чистая прибыль:", cerebro.broker.getvalue() - start_capital)
cerebro.plot(style="candle") # Рисуем свечной график
```

Большая часть кода уже вам знакома по предыдущим примерам.

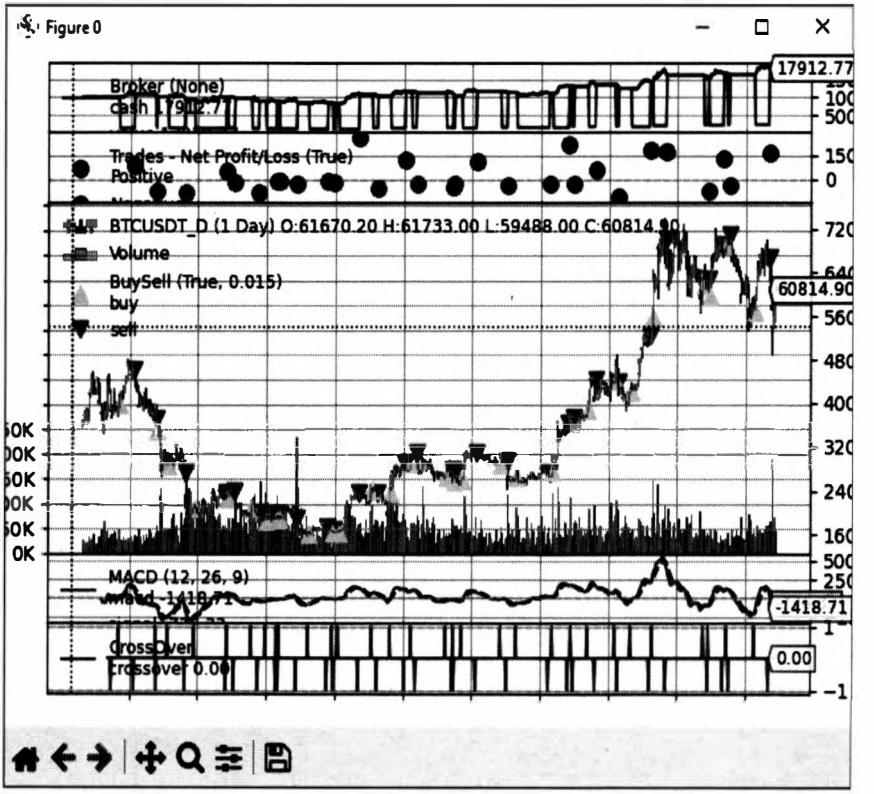
После создания словаря параметров для инициализации в методе __init__() мы установили переменную ордера и подключили индикатор MACD.

В свойство *self.mcross* добавили отслеживание пересечения сигнальной линии MACD.

В самом начале метода next() показываем информацию о текущем состоянии индикатора MACD. Затем, если линия MACD пересекается над сигнальной линией, открываем позицию. В обратном случае закрываем позицию.

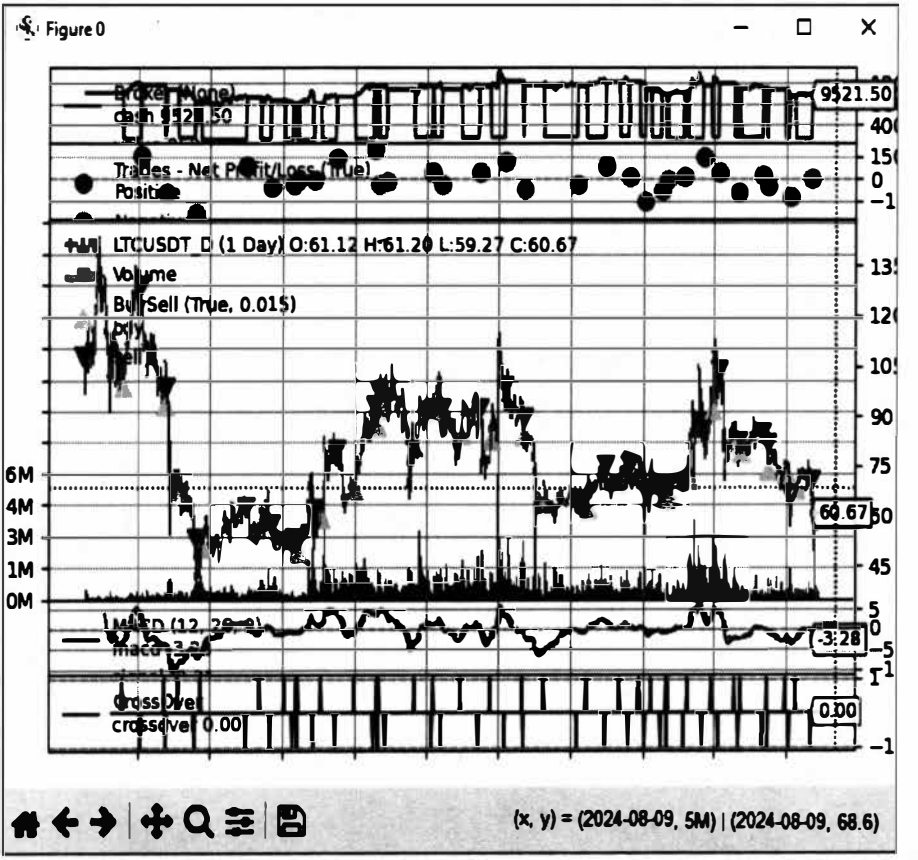
Ниже представлены выходные данные для BTC и LTC.

```
Результат бэктеста BTC:
Стартовый капитал: 10000
Конечный капитал: 17912.76923246284
Чистая прибыль: 7912.769232462841
```



Изображение 4.19.

```
Результат бэктеста LTC:
Стартовый капитал: 10000
Конечный капитал: 9521.504870921259
Чистая прибыль: -478.49512907874123
```



Изображение 4.20.

Обратите внимание, что на Плоте появились дополнительные части с показателем индикатора MACD и местами пересечений сигнальной линии.

4.20. Линии Боллинджера

Завершим тестирование классических индикаторов моим любимым инструментом. Линии Боллинджера были созданы Джоном Боллинджером в 1984 году. Впоследствии он выстроил успешный бизнес на своем изобретении.

Полосы Боллинджера просто рассчитывают показатели стандартного отклонения и линий скользящей средней.

Создайте новый файл BOLLINGERBacktest.py и перенесите код из листинга 4.6.

Листинг 4.6

```
from os.path import abspath # Функция для работы с путями
from datetime import datetime # Объект для работы со временем
from backtrader import Cerebro, Strategy, sizers # Компоненты БТ
from backtrader.feeds import GenericCSVData # Основной класс для CSV
# Импорт индикаторов
from backtrader.indicators import BollingerBands, CrossUp, CrossDown

class BollingerBandStrategy(Strategy): # Наследуемся от класса Стратегий
    params = ( # Параметры индикатора Боллинджера
        "bperiod": 36, # Количество дней для расчёта
        "dev": 2.25, # Коэффициент отклонения
    )

    def __init__(self): # Инициализация стратегии и временных рядов
        # Инициализируем линии Боллинджера
        self.boll = BollingerBands(
            period=self.params.bperiod,
            devfactor=self.params.dev,
            plot=True,
            plotname="Bollinger Band",
            subplot=False) # Не рисуем отдельный плот

        # Добавляем точки пересечения индикатора
        # Пересечение нижней границы снизу вверх сигнал на покупку
        self.buy_signal = CrossUp(
            self.data, # Указываем источник данных
            self.boll.lines.bot, # Указываем место пересечения
            plotname="Сигнал на покупку", # Имя плота
            plot=True, # Отрисовываем плот
        )

        # Пересечение верхней границы  сверху вниз сигнал на продажу
        self.sell_lsignal = CrossDown(
            self.data, # Указываем источник данных
```

```
            self.boll.lines.top, # Указываем место пересечения
            plotname="Сигнал на продажу", # Имя плота
            plot=True, # Отрисовываем плот
        )

    def next(self): # Появление новой свечи и логика стратегии
        # Если НЕ в позиции и произошло пересечение низа
        if not self.position:
            if self.buy_signal > 0:
                self.buy() # Покупаем
        # Если в позиции и произошло пересечение верха
        elif self.sell_lsignal > 0:
            self.sell() # Продаём

# Подготовка источника свечных данных
# MyCSVData основной класс для загрузки CSV файла
class MyCSVData(GenericCSVData):
    # В параметрах указываем какой столбец CSV файла
    # за какие данные отвечает
    params = (
        "dtformat": "%Y-%m-%d", # Формат времени для парсинга
        "datetime": 0,
        "open": 1,
        "high": 2,
        "low": 3,
        "close": 4,
        "volume": 5,
        "openinterest": -1, # -1 это последний столбец
    )

csv_file_path = abspath("./data/LTCUSDT_D.csv") # Путь к источнику данных
data = MyCSVData( # Объект с источником свечных данных
    dataname=csv_file_path,
    fromdate=datetime(2022, 1, 24),
    todate=datetime(2024, 8, 10),
    reverse=False)

# Подключение источника данных, аналитика и запуск бэктеста
cerebro = Cerebro() # Создаём объект Церебро. Мозг Бектрейдера
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data
cerebro.addstrategy(BollingerBandStrategy) # Добавляем стратегию
cerebro.broker.setcash(10000.0) # Устанавливаем стартовый баланс
cerebro.addsizer(sizers.PercentSizer, percents=85) # Позиция в %
cerebro.broker.setcommission(commission=0.0003) # Комиссия брокера
start_capital = cerebro.broker.getvalue() # Фиксируем баланс до старта
cerebro.run() # Запускаем бэктест
```

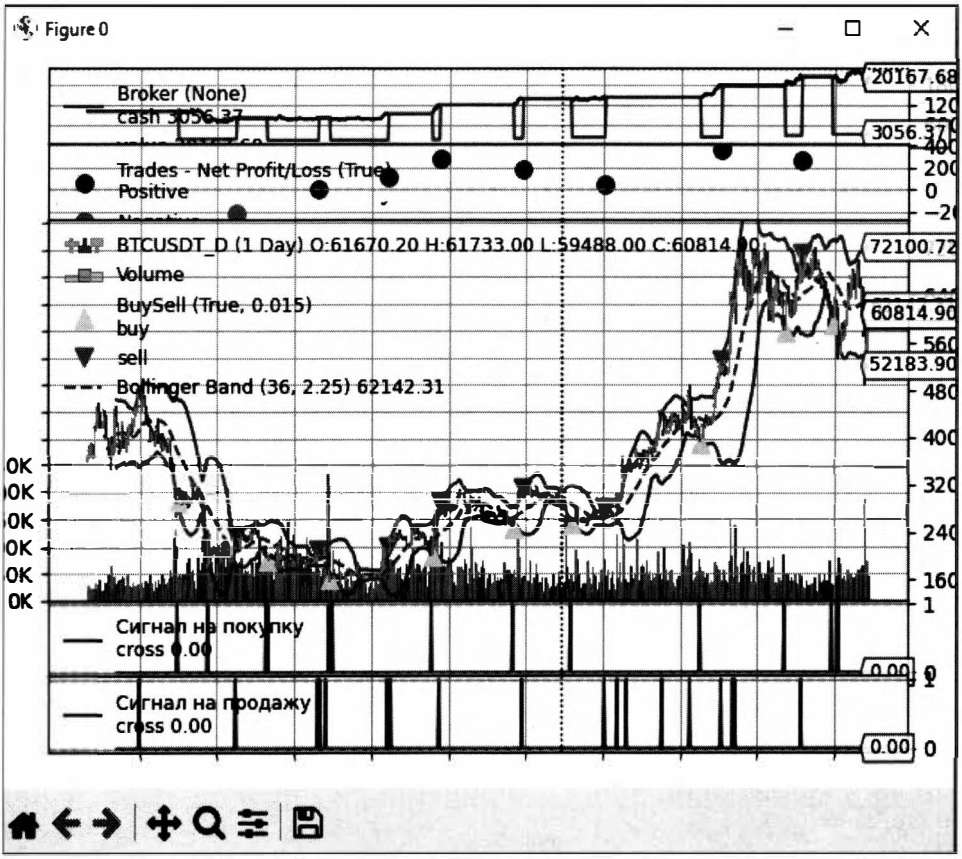
```
print("Конечный капитал", cerebro.broker.getvalue())
print("Чистая прибыль", cerebro.broker.getvalue() - start_capital)
cerebro.plot(style="candle") # Рисуем свечной график
```

После импорта всех необходимых библиотек, объявления класса и набора параметров стратегии мы, как всегда, инициализируем свойства.

- В методе __init__() при инициализации индикатора Боллинджера мы передали чуть больше аргументов, чем обычно, а именно явно указали имя для индикатора и запретили отрисовку отдельного плота.
- Затем в свойствах self.buy_signal и self.sell_lsignal указали условия для входа и выхода из сделки. Благодаря индикаторам CrossUp и CrossDown можно не только отследить пересечение сигнальных линий, но и определить направление, в котором пересечение произошло.
- Пересечение нижней границы Боллинджера снизу вверх — открываем позицию.
- Пересечение верхней границы Боллинджера сверху вниз — закрываем.
- В том числе мы приказали Бэктрейдеру, чтобы при срабатывании сигнала были отрисованы соответствующие данные в качестве дополнительной линии на Плоте.

Ниже представлены выходные данные для BTC и LTC.

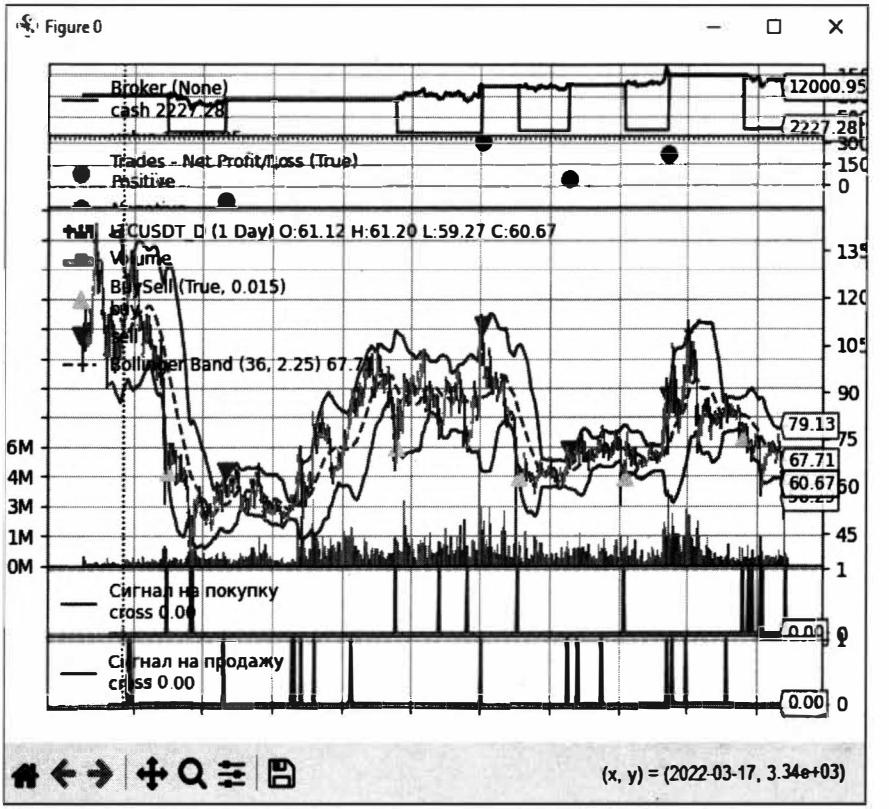
```
Результат бэктеста BTC:
Стартовый капитал: 10000
Конечный капитал: 20167.681486044465
Чистая прибыль: 10167.681486044465
```



Изображение 4.21.

```
Результат бэктеста LTC:
Стартовый капитал: 10000
Конечный капитал: 12000.945230463522
Чистая прибыль: 2000.9452304635215
```

Воспользуйтесь индикаторами CrossUp и CrossDown как для Стохастика, так и для RSI.



Изображение 4.22.

4.21. Уровни поддержи/сопротивления. Точки Пивот. Работа с мультитаймреймами

В Бэктрейдер уже встроено всё необходимое для поиска линий поддержки/сопротивления, а также для определения потенциальных разворотов рынка (точки Пивот). Для этого просто воспользуемся индикатором PivotPoint, который рассчитает уровни и определит точки Пивот по старшему ТФ.

Создайте файл PIVOTBacktest.py и перепишите код.

Листинг 4.7

```
from os.path import abspath # Функция для работы с путями
from datetime import datetime # Объект для работы со временем
from backtrader import Cerebro, Strategy, TimeFrame, sizers # Компоненты
from backtrader.feeds import GenericCSVData # Основной класс для CSV
from backtrader.indicators import PivotPoint # Импорт индикаторов

class PivotPoitStrategy(Strategy): # Наследуемся от класса Стратегий
    params = ( # Параметры индикатора Пивот
        "plot_on_daily": False
    )

    def __init__(self): # Инициализация стратегии и временных рядов
        # Подключаем Пивот к недельным свечам. Находятся в self.data1
        self.pivot_point = PivotPoint(self.data1,
        _autoplot=self.params.plot_on_daily)
        self.close = self.data.close # Цены закрытия текущей свечи
        self.order = None # Храним ордер
        self.r1_price_for_take = None # Храним цену сопротивления
        self.s1_price_for_stop = None # Храним цену поддержки
        self.executed_price = None # Цена исполненного ордера

    def notify_order(self, order):
        # Срабатывает когда изменяется статус ордера
        # Если ордер принят то ничего не делаем
        if order.status in [order.Submitted, order.Accepted]:
            return # Выходим и дальше не продолжаем

        if order.status == order.Completed:
            # Если ордер исполнен и произошла покупка
            if order.isbuy():
                # Фиксируем цену входа
                self.executed_price = order.executed.price
        # Если произошла отмена ордера или недостаток маржи
        elif order.status in [order.Rejected, order.Margin]:
            print("Ордер отменён")
        self.order = None # Сбрасываем ордер

    def next(self): # Появление новой свечи и логика стратегии
        # По индексу 0 лежит Пивот текущего дня
        print("Точка pivot", self.pivot_point[0])
        print("Поддержка s1", self.pivot_point.lines.s1[0])
        print("Поддержка s2", self.pivot_point.lines.s2[0])
        # R1 это откуда обычно выходят из сделки
        print("Сопротивление r1", self.pivot_point.lines.r1[0])
        print("Сопротивление r2", self.pivot_point.lines.r2[0])
```

```
        if self.order: # Если есть не исполненный ордер
            return # Выходим и дальше не продолжаем

        if not self.position: # Если позиции нет
            # Если цена закрытия текущей свечи ниже
            # или равна точке разворота
            is_enter = self.data_close <= self.pivot_point[0]
            if is_enter: # Входим в позицию и фиксируем тейк и стоп
                print("Входим в позицию")
                # Фиксируем точки тейка и стопа
                self.r1_price_for_take = self.pivot_point.lines.r1[0]
                self.s1_price_for_stop = self.pivot_point.lines.s1[0]
                self.order = self.buy() # Покупаем
                return # Завершаем работу метода next()
        else: # Проверяем сигналы на выход из позиции
            # Если цена закрытия текущей свечи ниже
            # или равна цене поддержки
            is_stop = self.close <= self.s1_price_for_stop
            if is_stop: # Обрабатываем стоп
                print("Сработал стоп")
                self.order = self.sell() # Продаём
                return # Завершаем работу метода next()

            # Если цена закрытия текущей свечи выше
            # или равна сопротивлению
            is_exit = self.data_close >= self.r1_price_for_take
            if is_exit: # Фиксируем прибыль / убыток
                print("Выход из позиции")
                self.order = self.sell() # Продаём
                return # Завершаем работу метода next()

# Подготовка источника свечных данных
# MyCSVData основной класс для загрузки CSV файла
class MyCSVData(GenericCSVData):
    # В параметрах указываем какой столбец CSV файла
    # за какие данные отвечает
    params = (
        ("dtformat": "%Y-%m-%d", # Формат времени для парсинга
        ("datetime": 0,
        ("open": 1,
        ("high": 2,
        ("low": 3,
        ("close": 4,
        ("volume": 5,
        ("openinterest": -1, # -1 это последний столбец
    )
```

```
csv_file_path = abspath("./data/LTCUSDT_D.csv") # Путь к источнику данных
data = MyCSVData( # Объект с источником свечных данных
    dataname=csv_file_path,
    fromdate=datetime(2022, 1, 24),
    todate=datetime(2024, 8, 10),
    reverse=False)

# Подключение источника данных, аналитика и запуск бэктеста
cerebro = Cerebro() # Создаём объект Церебро. Мозг Бектрейдера
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data

# С помощью resampledata() сжимаем до недель. Хранится в self.data1
cerebro.resampledata(data, timeframe=TimeFrame.Weeks)
cerebro.addstrategy(PivotPoitStrategy) # Добавляем стратегию
cerebro.broker.setcash(10000.0) # Устанавливаем стартовый баланс
cerebro.addsizer(sizers.PercentSizer, percents=90) # Позиция в %
cerebro.broker.setcommission(commission=0.0003) # Комиссия брокера
start_capital = cerebro.broker.getvalue() # Фиксируем баланс до старта
cerebro.run() # Запускаем бэктест

print("Конечный капитал", cerebro.broker.getvalue())
print("Чистая прибыль", cerebro.broker.getvalue() - start_capital)
cerebro.plot(style="bar") # Рисуем барный график
```

Данный пример стоит разобрать с конца исходника. Особого внимания требуют строки:

```
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data
cerebro.resampledata(data, timeframe=TimeFrame.Weeks)
# Сжимаем до недель в self.data1
```

После подключения основного источника данных, который будет доступен в классе стратегии в свойстве self.data, мы заставили Бэктрейдер "сжать" дневные свечи до недельных, при помощи метода cerebro.resampledata().

Благодаря этому у нас появился новый источник свечных данных в свойстве self.data1.

В методе _init__() мы указали недельные свечи для построения уровней и точек Пивот:

```
self.pivot_point = PivotPoint(self.data1,
_autoplot=self.params.plot_on_daily)
```

и при этом приказали, что НЕ нужно рисовать графические данные в разделе со свечами.

В методе next() у нас появилось несколько свойств, а именно:

- *self.pivot_point[0]* — по индексу 0 лежит точка Пивот текущего дня.
- *self.pivot_point.lines.s1[0]* — ближняя линия поддержки.
- *self.pivot_point.lines.s2[0]* — дальняя линия поддержки.
- *self.pivot_point.lines.r1[0]* — ближняя линия сопротивления.
- *self.pivot_point.lines.r2[0]* — дальняя линия сопротивления.

Свойство R1 (ближняя линия сопротивления) — это место, откуда обычно выходят из сделки.

В строке `is_enter = self.close <= self.pivot_point[0]` мы проверяем, закрылась ли текущая свеча на точке Пивот или ниже её. Если условие верно, фиксируем ближайшую линию поддержки в качестве стопа, а ближайшую линию сопротивления — в качестве тейка.

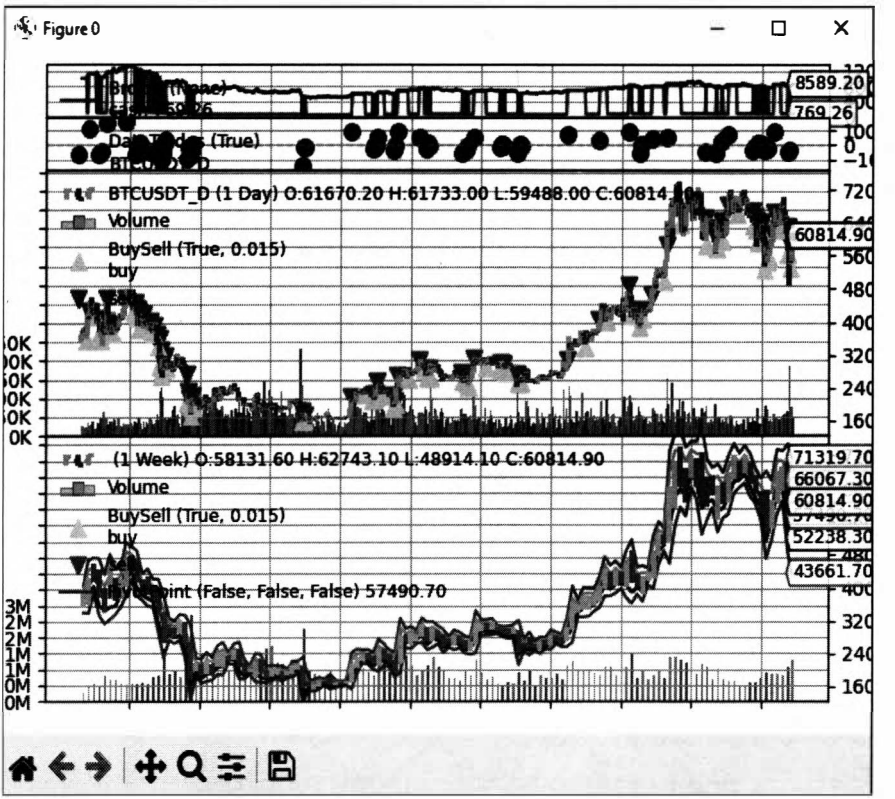
Получается, что мы используем старший таймфрейм для работы с младшим. При этом не забывайте соблюдать порядок подключения данных. Более низкий ТФ должен быть подключен ДО компрессии к старшему ТФ.

Объект TimeFrame, который мы импортировали в начале исходника и использовали для сжатия свечей, имеет ещё несколько свойств:

- *TimeFrame.Days* — сжимает до дневных свечей (значение по умолчанию).
- *TimeFrame.Weeks* — сжимает до недельных свечей.
- *TimeFrame.Months* — сжимает до месячных свечей.
- *TimeFrame.Years* — сжимает до годовых свечей.

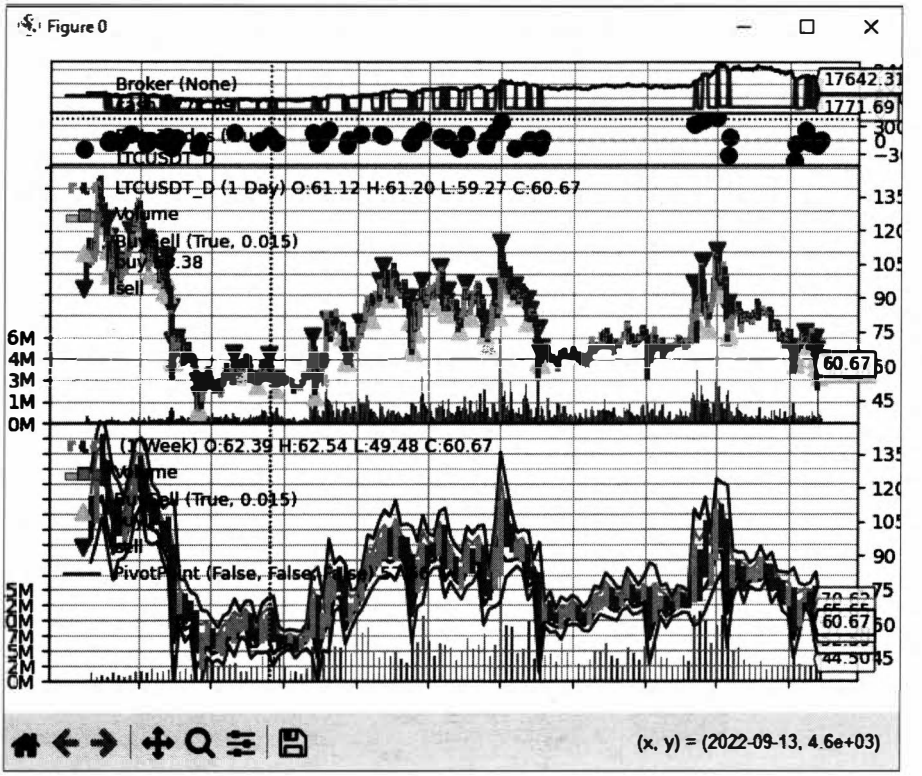
Ниже представлены выходные данные для BTC и LTC.

```
Результат бэктеста BTC:
Начальный капитал: 10000
Конечный капитал: 8589.200617446684
Чистая прибыль: -1410.7993825533158
```



Изображение 4.23.

```
Результат бэктеста LTC:
Начальный капитал: 10000
Конечный капитал: 17642.310570161386
Чистая прибыль: 7642.310570161386
```



Изображение 4.24.

4.22. Используем числа Фибоначчи для расчёта уровней и точек Пивот

Числа Фибоначчи выстраивают последовательность чисел, основанную на сумме двух предыдущих.

Числа Фибоначчи — любимый инструмент институционального трейдера, поскольку с их помощью можно очень точно рассчитать линии поддержки/сопротивления и определить потенциальные развороты рынка.

По сути, именно в зонах уровней расположена основная ликвидность, необходимая для открытия позиции институциональными трейдерами.

Создайте новый файл FIBOTest.py и внесите код.

Листинг 4.8

```
from os.path import abspath # Функция для работы с путями
from datetime import datetime # Объект для работы со временем
from backtrader import Cerebro, Strategy, TimeFrame # Компоненты BT
from backtrader.feeds import GenericCSVData # Класс для CSV
from backtrader.indicators import FibonacciPivotPoint # Индикаторы

class FiboPivotStrategy(Strategy): # Наследуемся от класса Стратегий
    def __init__(self): # Инициализация стратегии и временных рядов
        # Подключаем Пивот к недельным свечам. Находятся в self.data1
        self.pivot_point = FibonacciPivotPoint(self.data1)

    def next(self): # Появление новой свечи и логика стратегии
        print("Точка pivot", self.pivot_point[0])
        print("Сопротивление s1", self.pivot_point.lines.s1[0])
        print("Сопротивление s2", self.pivot_point.lines.s2[0])
        print("Сопротивление s3", self.pivot_point.lines.s2[0])
        print("Поддержка r1", self.pivot_point.lines.r1[0])
        print("Поддержка r2", self.pivot_point.lines.r2[0])
        print("Поддержка r2", self.pivot_point.lines.r2[0])

# Подготовка источника свечных данных
# MyCSVData основной класс для загрузки CSV файла
class MyCSVData(GenericCSVData):
    # В параметрах указываем какой столбец CSV файла
    # за какие данные отвечает
    params = (
        "dtformat": "%Y-%m-%d", # Формат времени для парсинга
        "datetime": 0,
        "open": 1,
        "high": 2,
        "low": 3,
        "close": 4,
        "volume": 5,
        "openinterest": -1, # -1 это последний столбец
    )

csv_file_path = abspath("./data/LTCUSDT_D.csv") # Путь к источнику данных
data = MyCSVData( # Объект с источником свечных данных
    dataname=csv_file_path,
    fromdate=datetime(2022, 1, 24),
```

```
    todate=datetime(2024, 8, 10),
    reverse=False)

# Подключение источника данных, аналитика и запуск бэктеста
cerebro = Cerebro() # Создаём объект Церебро. Мозг Бектрейдера
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data

# Сжимаем до недель. Хранится в self.data1
cerebro.resampledata(data, timeframe=TimeFrame.Weeks)
cerebro.addstrategy(FiboPivotStrategy) # Добавляем стратегию
cerebro.run() # Запускаем бэктест
cerebro.plot(style="bar") # Барный плоттинг
```

После импорта всех библиотек объявляем класс стратегии. В методе __init()___ объявляем только одно свойство, в котором храним точки Пивот, рассчитанные на основе месячных свечей и чисел Фибоначчи. Теперь вам доступно по 3 линии поддержи и сопротивления. Соответствующие значения уровней и сама точка Пивот выводятся в методе next().

Изображение 4.25.

4.23. Индикатор ATR. Определяем волатильность рынка в несколько потоков

Показатель волатильности рынка ATR имеет широкое распространение в узких кругах профессиональных трейдеров. Считается, что данный индикатор не подходит в качестве сигнального инструмента для входа в рынок. Зато благодаря ему можно определять состояние флета и многое другое.

Давайте протестируем этот индикатор сразу на двух источниках свечных данных. Создайте файл ATRTest.py и перепишите следующий код.

Листинг 4.9

```
from os.path import abspath # Функция для работы с путями
from datetime import datetime # Объект для работы со временем
from backtrader import Cerebro, Strategy, TimeFrame # Компоненты БТ
from backtrader.feeds import GenericCSVData # Класс для CSV
from backtrader.indicators import ATR # Импорт индикатора ATR

class ATRStrategy(Strategy):
    def __init__(self): # Инициализация стратегии и временных рядов
        # Свечи LTC в  self.data. Аналог self.datas[0]
        # Свечи BTC в self.data1. Аналог self.datas[1]
        self.ltc_atr = ATR(self.data, plot = True) # Подключаем ATR к LTC
        self.btc_atr = ATR(self.data1, plot = True) # Подключаем ATR к BTC

    def next(self):
        # Работаем с данными LTC ATR за последние 3 дня
        LTC_atr1 = self.ltc_atr.lines.atr[0] # Сегодняшний ATR у LTC
        LTC_atr2 = self.ltc_atr.lines.atr[-1] # Вчерашний ATR у LTC
        LTC_atr3 = self.ltc_atr.lines.atr[-2] # Позавчерашний ATR у LTC
        print("LTC ATR сегодня", LTC_atr1)
        print("LTC ATR вчера", LTC_atr2)
        print("LTC ATR позавчера", LTC_atr3)

        # Если сегодняшний LTC ATR выше чем вчерашний
        # и вчерашний выше чем позавчера
        if LTC_atr1 > LTC_atr2 and LTC_atr2 > LTC_atr3:
            print("Волатильность LTC растёт",
                  "Допустимы взлёты и зарождение тренда")
        # Если сегодняшний ATR ниже чем вчерашний
        # и вчерашний ниже чем позавчера
```

```
        elif LTC_atr1 < LTC_atr2 and LTC_atr2 < LTC_atr3:
            print("Волатильность LTC падает")

        # Отслеживаем максимальный и минимальный
        # показатели волатильности LTC
        if LTC_atr1 >= 6.5: # Если сегодняшний LTC ATR в районе максимума
            print("LTC в районе максимума. Возможна коррекция или флет")
        elif LTC_atr1 <= 3.5: # Если сегодняшний LTC ATR в районе минимума
            print("LTC в районе минимума. Слабый интерес. Вероятно флет")

        # Работаем с данными BTC ATR за последние 3 дня
        BTC_atr1 = self.btc_atr.lines.atr[0] # Сегодняшний ATR у BTC
        BTC_atr2 = self.btc_atr.lines.atr[-1] # Вчерашний ATR у BTC
        BTC_atr3 = self.btc_atr.lines.atr[-2] # Позавчерашний ATR у BTC
        print("BTC ATR сегодня", BTC_atr1)
        print("BTC ATR вчера", BTC_atr2)
        print("BTC ATR позавчера", BTC_atr3)

        # Если сегодняшний BTC ATR выше чем вчера
        # и вчера выше чем позавчера
        if BTC_atr1 > BTC_atr2 and BTC_atr2 > BTC_atr3:
            print("Волатильность BTC растёт",
                  "Допустимы взлёты и зарождение тренда")
        elif BTC_atr1 < BTC_atr2 and BTC_atr2 < BTC_atr3:
            print("Волатильность BTC падает")

        # Отслеживаем максимальный и минимальный
        # показатель волатильности BTC
        if BTC_atr1 >= 3000: # Если сегодняшний BTC ATR в районе максимума
            print("BTC в районе максимума. Возможна коррекция или флет")
        elif BTC_atr1 <= 1500:
            # Если сегодняшний BTC ATR в районе минимума
            print("BTC в районе минимума. Слабый интерес. Вероятно флет")

# Подготовка источника свечных данных
# MyCSVData основной класс для загрузки CSV файла
class MyCSVData(GenericCSVData):
    # В параметрах указываем какой столбец CSV файла
    # за какие данные отвечает
    params = (
        ("dtformat", "%Y-%m-%d"), # Формат времени для парсинга
        ("datetime", 0),
        ("open", 1),
        ("high", 2),
        ("low", 3),
        ("close", 4),
        ("volume", 5),
```

```
        "openinterest": -1, # -1 это последний столбец
    }

# Указываем источники данных
datapath_ltc = abspath("./data/LTCUSDT_D.csv") # Путь к свечам LTC
datapath_btc = abspath("./data/BTCUSDT_D.csv") # Путь к свечам BTC

data_ltc = MyCSVData( # Объект с источником свечных данных LTC
    dataname=datapath_ltc,
    fromdate=datetime(2022, 1, 24),
    todate=datetime(2024, 8, 10),
    reverse=False)

data_btc = MyCSVData( # Объект с источником свечных данных BTC
    dataname=datapath_btc,
    fromdate=datetime(2022, 1, 24),
    todate=datetime(2024, 8, 10),
    reverse=False)

cerebro = Cerebro() # Создаём объект Церебро. Мозг Бектрейдера
cerebro.adddata(data_ltc) # Добавляем поток LTC. Хранятся в self.data
cerebro.adddata(data_btc) # Добавляем поток BTC. Хранятся в self.data1
cerebro.addstrategy(ATRStrategy) # Добавляем стратегию
cerebro.run() # Запускаем бэктест
cerebro.plot(style="bar") # Барный плоттинг
```

Мы подключили два источника данных, которые хранятся в свойствах self.data и self.data1.

В методе next() мы написали некоторое подобие сигнальной версии этого индикатора.

- Если волатильность находится в районе минимума и при этом начинает расти, возможно возобновление активности и тренда.
- Если ATR на максимальных значениях, то возможны коррекция или флет.

Считается, что данный индикатор не подходит в качестве сигнального инструмента для входа в рынок. Зато благодаря ему можно значительно улучшить точки входа и выхода своей стратегии.

4.24. Свечи Хайкин-Аши. Специфические свечи и фильтрация графиков

Свечи Хайкин-Аши были созданы в Японии для усреднения цены на продовольствие на разных биржах. Они позволяют отсеять рыночный шум, которому подвержены классические японские свечи.

Благодаря свечам Хайкин-Аши можно в более удобном виде отслеживать направление тренда, его замедление и даже моменты разворота рынка.

Давайте посмотрим, насколько эффективными могут быть эти свечи в торговле, благодаря бэктесту, проведенному в следующем исходнике.

Создайте новый проект HaikinAshiBacktest.py и перепишите листинг 4.10.

Листинг 4.10

```
from os.path import abspath # Функция для работы с путями
from datetime import datetime # Объект для работы со временем
from backtrader import Cerebro, Strategy, sizers, filters # Компоненты
from backtrader.feeds import GenericCSVData # Класс для CSV

class HaikinAshiStrategy(Strategy):
    def __init__(self):  # Инициализация стратегии и временных рядов
        # Просто сокращаем внешний вид свечных данных
        self.open = self.datas[0].open
        self.close = self.datas[0].close

    def next(self): # Появление новой свечи и логика стратегии
        if not self.position: # Если позиции нет
            # Если закрытия текущей свечи выше открытия предыдущей
            is_signal_buy = self.close[0] > self.open[-1]
            if is_signal_buy:
                self.buy() # Открываем сделку
        else:
            # Если цена закрытия текущей свечи меньше
            # чем цена открытия предыдущей
```

```
            is_signal_cell = self.close[0] < self.open[-1]
            if is_signal_cell:
                self.sell() # Выходим из сделки

# Подготовка источника свечных данных
# MyCSVData основной класс для загрузки CSV файла
class MyCSVData(GenericCSVData):
    # В параметрах указываем, какой столбец CSV файла
    # за какие данные отвечает
    params = (
        "dtformat": "%Y-%m-%d", # Формат времени для парсинга
        "datetime": 0,
        "open": 1,
        "high": 2,
        "low": 3,
        "close": 4,
        "volume": 5,
        "openinterest": -1) # -1 это последний столбец

csv_file_path = abspath("./data/LTCUSDT_D.csv") # Путь к источнику данных
data = MyCSVData( # Объект с источником свечных данных
    dataname=csv_file_path,
    fromdate=datetime(2022, 1, 24),
    todate=datetime(2024, 8, 10),
    reverse=False)
data.addfilter(filters.HeikinAshi) # Добавляем свечной фильтр HeikinAshi

# Подключение источника данных, аналитика и запуск бэктеста
cerebro = Cerebro() # Создаём объект Церебро. Мозг Бектрейдера
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data
cerebro.addstrategy(HaikinAshiStrategy) # Добавляем стратегию
cerebro.broker.setcash(10000.0) # Устанавливаем стартовый баланс
cerebro.addsizer(sizers.PercentSizer, percents=85) # Размер позиции в %
cerebro.broker.setcommission(commission=0.0003) # Комиссия брокера
start_capital = cerebro.broker.getvalue() # Фиксируем баланс до старта
cerebro.run() # Запускаем бэктест

print("Конечный капитал", cerebro.broker.getvalue())
print("Чистая прибыль", cerebro.broker.getvalue() - start_capital)
cerebro.plot(style="candle") # Рисуем свечной график
```

В начале файла мы импортировали все необходимые подбиблиотеки, в том числе новую под названием *filters*.

В методе __init__() мы привели данные о свечах в более читабельный вид. В методе next() мы описали стратегию входа и выхода из позиции.

Если закрытие текущей свечи выше открытия предыдущей — открываем сделку на 90% от депозита. Если цена закрытия текущей свечи меньше, чем открытия предыдущей — выходим из позиции.

Ниже представлены выходные данные для BTC и LTC.

```
Результат бэктеста BTC:
Начальный капитал: 10000
Конечный капитал: 18.1737.67568053212
Чистая прибыль: 17.1737.67568053212

Результат бэктеста LTC:
Начальный капитал: 10000
Конечный капитал: 186484.69256072253
Чистая прибыль: 176484.69256072253
```

Несмотря на впечатляющие показатели бэктеста, к сожалению, для автоматизации данная стратегия не подходит. Свечи Хайкин-Аши отражают усредненные показатели свечей, которые не отвечают реальным ценам.

Тем не менее, для визуализации ручной торговли свечи Хайкин-Аши — потрясающий инструмент, который я активно использую. Они отрабатывают НЕ на всех активах, но даже визуально можно отследить, что сигнальная разворотная дневная свеча Хайкин-Аши отлично срабатывает (например, на LTC) в комбинации с уровнями или выходом из зоны перекупленности осцилляторов.

4.25. Умный риск-менеджмент. Локальные мин/макс. Отложенные тейк/стоп-ордера

В данном примере реализована достаточно сложная логика, которой, на мой субъективный взгляд, можно избежать (но это не точно).

Также используется установка отложенных стоп-лоссов по показателям локального минимума. Лично мне по душе простая фиксация цены входа и отслеживание просадки от этой точки.

При таком подходе биржа НЕ видит, где расположены ваши стопы, и у неё НЕ появится соблазна нарисовать тень на свече, чтобы поглотить вашу позицию.

С другой стороны, рынок меняется, и отслеживание локальных минимумов и максимумов избавит от некоторой головной боли.

Тем не менее, данный пример отлично демонстрирует всю гибкость Бэктрейдера и охватывает практически все НЕ изученные до этого момента возможности. Я рекомендую переписать этот пример, а затем разделить его на более мелкие части в тестовых проектах. Так вы сможете более уверенно овладеть каждой новой для себя конструкцией, а это руководство не будет избыточным.

Создайте файл FULLRSIStrategy.py и перепишите листинг 4.11.

Листинг 4.11

```
from os.path import abspath # Функция для работы с путями
from datetime import datetime # Объект для работы со временем

# Компоненты Бэктрейдер
from backtrader import Cerebro, Strategy, Order, analyzers, sizers
from backtrader.feeds import GenericCSVData # Класс для загрузки CSV

# Импорт индикаторов
from backtrader.indicators import RSI_EMA, CrossDown
from backtrader.indicators import CrossUp, Lowest, Highest

class RsiStrategy(Strategy):
    params = ( # Параметры индикатора RSI
        "rsi_period": 14, # Период RSI
        "stop_period": 7 # Период для поиска локального минимума
    )

    def __init__(self): # Инициализация стратегии и временных рядов
        self.rsi = RSI_EMA(period=self.params.rsi_period) # RSI
        # Индикатор для сигналов на вход на базе RSI:
        # 1 - входим лонг, 0 - ждём
        self.enter_long = CrossUp(self.rsi, 20) # Пересечение снизу вверх

        # Индикатор для сигналов на выход из лонга на базе RSI:
        # 1 - выходим, 0 - ждём
        self.exit_long = CrossDown(self.rsi, 80) # Пересечение сверху вниз
```

```
        # Расчёт значений локального минимума за период stop_period
        self.local_min = Lowest(self.data.low,
            period=self.params.stop_period)
        self.local_max = Highest(self.data.high,
            period=self.params.stop_period)
        self.long_order = None # Храним лонговый ордер
        self.stop_loss_order = None # Храним стоповый ордер
        self.take_profit_order = None # Храним тейк ордер
        self.entry_price = None # Цена входа
        self.stop_loss_price = None # Цена стоп-лосса
        self.profit_trades = [] # Список прибыльных сделок
        self.losing_trades = [] # Список убыточных сделок

    def notify_order(self, order):
        # Срабатывает когда изменяется статус ордера
        if order.status == order.Completed:
            # Если ордер принят и размещён
            # Размещаем отложенный стоп-лосс Ордер после входа
            if order.info.name in ("long"):
                print("NOTIFY_ORDER. Вошли в", order.info.name)
                print("Позиция", self.position.size)
                print("Цена", order.executed.price)
                print("Стоп-лосс", self.stop_loss_price)

                # Параметры стоп-лосс ордера в виде словаря
                stop_loss_params = {
                    "exectype": Order.Stop, # Тип ордера Stop
                    "size": order.executed.size, # Размер
                    "price": self.stop_loss_price, # Цена стопа
                    "name": "stop_loss", # В имени - тип ордера
                    "oco": self.take_profit_order} # Отмена тейка

                if order.isbuy(): # Если ордер был лонговым
                    # Ставим стоп
                    self.stop_loss_order = self.sell(**stop_loss_params)

                # Оповещаем, что стоп-лосс установлен
                print("NOTIFY_ORDER. Отправлен Ордер стоп-лосс для",
                    order.info.name)
                print("Позиция", self.stop_loss_order.size)
                print("Цена", self.stop_loss_price)

            # На случай, если ордер стоповый или тейковы.
            # Т.е. НЕ для входа в рынок
            elif order.info.name in ("stop_loss", "take_profit"):
                print("Вышли из позиции. Сработал", order.info.name)
                print("Позиция", self.position.size)
                print("Цена", order.executed.price)
```

```
    def next(self): # Появление новой свечи и логика стратегии
        if self.position.size: # Если есть открытая позиция
            # Проверяем сигналы на выход
            # Если текущее состояние индикатора выхода из лонга
            # и размер позиции больше 0
            if self.exit_long[0]:
                print("NEXT. Сигнал на выход!")
                # Создаём ордер для закрытия позиции
                # и помечаем его как take_profit
                self.take_profit_order = self.close(name="take_profit",
                    oco=self.stop_loss_order) # Тейкаем и отменяем стоп
                print("NEXT. Отправлен Ордер take_profit")
        else: # Проверяем сигналы на вход в лонг
            if self.enter_long[0] > 0: # Если сигнал на вход больше 1
                print("NEXT. Сигнал на вход в лонг!")
                self.entry_price = self.data.high[0] # Цена входа
                self.stop_loss_price = self.local_min[0] # Стоп цена

                # Открываем лонговый, лимитный ордер по цене,
                # на которой пришёл сигнал
                self.long_order = self.buy(exectype=Order.Limit,
                    price=self.entry_price, name="long") # Лонг лимит

                print("NEXT. Отправлен Ордер на вход в long")
                print("Кэш:", self.broker.getcash())
                print("Позиция:", self.long_order.size)
                print("Цена:", self.long_order.price)

# Подготовка источника свечных данных
# MyCSVData основной класс для загрузки CSV файла
class MyCSVData(GenericCSVData):
    # В параметрах указываем какой столбец CSV файла
    # за какие данные отвечает
    params = (
        "dtformat": "%Y-%m-%d", # Формат времени для парсинга
        "datetime": 0,
        "open": 1,
        "high": 2,
        "low": 3,
        "close": 4,
        "volume": 5,
        "openinterest": -1)# -1 это последний столбец

csv_file_path = abspath("./data/LTCUSDT_D.csv") # Путь к источнику данных
data = MyCSVData( # Объект с источником свечных данных
    dataname=csv_file_path,
    fromdate=datetime(2022, 1, 24),
```

```
    todate=datetime(2024, 8, 10),
    reverse=False)

# Подключение источника данных, аналитика и запуск бэктеста
cerebro = Cerebro() # Создаём объект Церебро. Мозг Бектрейдера
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data
cerebro.addstrategy(RsiStrategy) # Добавляем стратегию
cerebro.broker.setcash(10000.0) # Устанавливаем стартовый баланс
cerebro.addsizer(sizers.PercentSizer, percents=95) # Позиция в %
cerebro.broker.setcommission(commission=0.0003) # Комиссия брокера
start_capital = cerebro.broker.getvalue() # Фиксируем баланс до старта
cerebro.run() # Запускаем бэктест

print("Конечный капитал", cerebro.broker.getvalue())
print("Чистая прибыль", cerebro.broker.getvalue() - start_capital)
cerebro.plot(style="candle") # Рисуем свечной график
```

В методе __init__() мы, как всегда, инициализируем свойства, которые понадобятся для реализации стратегии и её логики.

В данном примере мы НЕ станем приводить свечи к сохранённому виду и оставим как есть. В свойствах self.enter_long и self.exit_long хранятся сигналы на вход и выход в зависимости от направлений пересечений.

В свойствах self.local_min и self.local_max хранятся локальные минимум и максимум за период, указанный в параметре params.stop_period.

Эти показатели используются для выставления "умных" стоп-лоссов и тейк-профитов. Мы воспользуемся только свойством self.local_min для выставления отложенного стоп-лосса.

В свойствах self.losing_trades и self.profit_trades мы храним суммы успешных и убыточных сделок.

В завершение инициализации свойств мы сохраняем информацию на лонг-ордер, стоп-ордер и тейк-ордер.

Устанавливаем параметры стоп-лосса, цену тейк-профита, цену входа и цену, по которой сработает стоп-лосс.

> В методе notify_order() отслеживаем состояние ордера. Если ордер принят и размещён, самое время установить стоп-лосс.

Как вы уже заметили, каждому созданному ордеру могут быть присвоены собственный идентификатор в виде имени.

В order.info.name хранится идентификатор ордера. В нашем случае это один из трех вариантов: *long*, *stop_loss, take_profit*.

Если в order.info.name содержится имя *long*, начинаем работу по установке стопа. В переменной словаря *stop_loss_params* хранятся параметры стоп-лосс ордера.

Благодаря заполнению параметра exectype=Order.Stop, Бэктрейдер понимает, что ордер не для входа в рынок, а стоповый (т.е. отложенный).

Далее, проверяем является ли ордер лонговым. Если условие верно, то благодаря строке

```
self.stop_loss = self.sell(**stop_loss_params)
```

будут распакованы параметры стоп-ордера.

Затем происходит проверка elif: order.info.name in ("stop_loss", "take_profit"). Она необходима, чтобы уведомить о срабатывании стоп-лосс ордера или тейк-профит ордера.

> Метод notify_trade() просто раскидывает по спискам суммы успешных и неуспешных сделок.

В методе next() первым делом проверяем, есть ли позиция. Если позиция есть, ищем сигналы на выход. Для этого проверяем состояние пересечения индикаторов, соответствующих выходу из лонг-позиции.

Если условия для выхода есть, в строке

```
self.take_profit = self.close()
```

создаём ордер для закрытия текущего ордера.

Строка

```
self.take_profit_order = self.close(name="take_profit",
oco=self.stop_loss_order)
```

закроет текущую позицию по тейк-профиту и отменит активный стоплосс ордер.

> Метод self.close() — это альтернатива self.sell().

Не забудьте, если вы создадите свойство self.close, в котором хранятся цены закрытия свечей, как в предыдущих примерах, то при использовании метода self.close() у вас будет ошибка совпадения имен.

Если позиции нет, ищем точки входа по текущему состоянию индикаторов пересечения.

В строке `if self.enter_long[0] > 0:` проверяем, находится ли индикатор входа в положении выше 0.

Если условие удовлетворено, в строке

```
self.long_order = self.buy(exectype=Order.Limit,
price=self.entry_price, name="long")
```

сохраняем лимитный ордер на покупку, исполненный НЕ по рынку, а по цене self.entry_price.

> В self.entry_price хранится цена закрытия текущей свечи из self.data.high[0].

Подведем итог. В этом примере мы узнали, что открывать можно лимитные ордера по необходимой цене благодаря строке:

```
self.buy(exectype=Order.Limit, price=self.entry_price, name="long")
```

В том числе узнали, что ордеру можно дать имя или идентификатор.

Благодаря строке

```
self.local_min = Lowest(self.data.low, period=self.params.stop_period)
```

мы смогли определить локальный минимум и создать отложенный стоп-ордер в строках:

```
stop_loss_params = {
    "exectype": Order.Stop, # Тип ордера Stop, т.е. отложенный
    "size": order.executed.size, # Размер лота для закрытия
    "price": self.stop_loss_price, # Цена стоп-лосса
    "name": "stop_loss", # В имени помечаем тип ордера
    "oco": self.take_profit_order
}

if order.isbuy(): # Если ордер был лонговым
    # Устанавливаем стоп-лосс
    self.stop_loss_order = self.sell(**stop_loss_params)
```

Научились закрывать активные позиции с помощью метода self.close().

Научились выставлять тейк-ордеры в строке:

```
self.take_profit_order = self.close(name="take_profit",
oco=self.stop_loss_order)
```

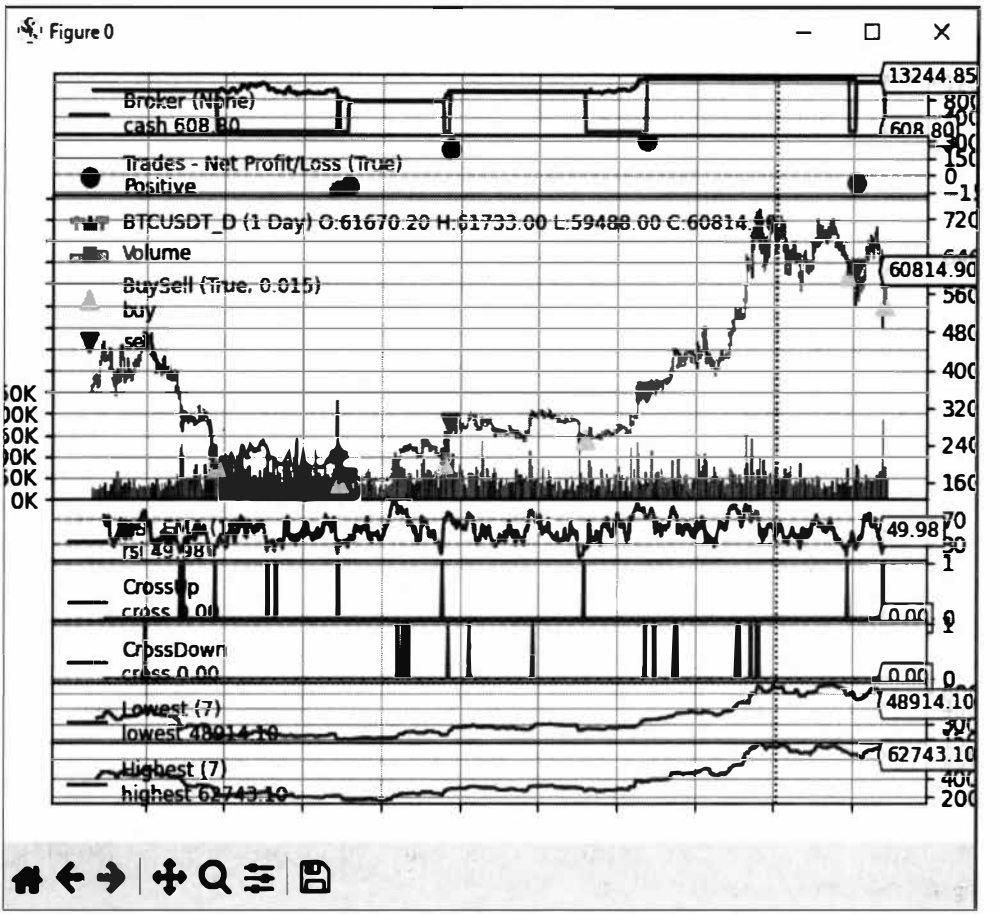
Если сработает тейк-ордер, то стоп-ордер будет отменен в параметре *oco=self.stop_loss_order*.

Полный список всех возможных типов ордеров:

Order.Market, Order.Close, Order.Limit, Order.Stop, Order.StopLimit, Order.StopTrail, Order.StopTrailLimit, Order.Histstorical.

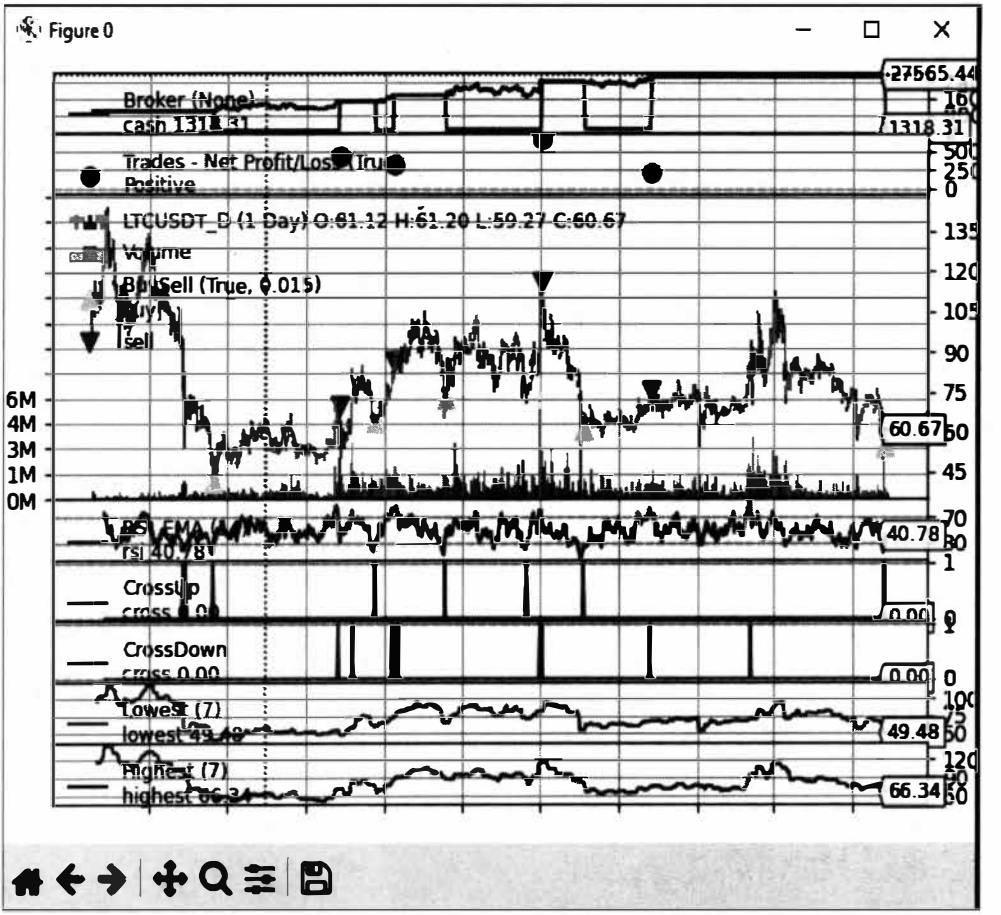
Ниже представлены выходные данные для BTC и LTC.

```
Результат бэктеста BTC.
Стартовый капитал: 10000
Конечный капитал: 13244.852944995908
Чистая прибыль: 3244.852944995908
```



Изображение 4.26.

```
Результат бэктеста LTC.
Стартовый капитал: 10000
Конечный капитал: 27565.437523264034
Чистая прибыль: 17565.437523264034
```



Изображение 4.27.

4.26. Коротко о трейлинг-стопах

Для Order.StopTrail, скользящего вверх:

```
self.buy(size=10, exectype=Order.StopTrail, trailamount=0.25) # или
self.buy(size=10, exectype=Order.StopTrail, price=13.50,
trailamount=0.21)
```

Для StopTrail, скользящего вниз:

```
self.sell(size=1, exectype=Order.StopTrail, trailamount=0.25) # или
self.sell(size=1, exectype=Order.StopTrail, price=10.50,
trailamount=0.21)
```

Прочтите документацию для подробного ознакомления по адресу:

https://backtrader.ru/docu/orders/trail/stoptrail/

4.27. Оптимизация торговой стратегии по одной SMA

Несмотря на то, что наши стратегии уже показывают потенциальную прибыль для растущего рынка и уберегают от убытков во время слома тренда, хочется выяснить, какой-же показатель SMA наиболее прибыльный.

Бэктрейдер позволяет автоматически найти наилучший показатель прибыли при использовании индикаторов, избавляя нас от рутинного перебора параметров.

Во-первых, на всякий случай наберите в терминале команду:

```
pip install pandas
```

Вам необходимо установить новую зависимость, если ещё этого не сделали. Библиотека Pandas позволяет организовать табличный вывод и расчёты, аналогичные Excel. С его помощью мы отсортируем данные на каждый из вариантов SMA и найдем лучший.

Во-вторых, сделайте копию предыдущего исходника и приведите его к листингу 4.12.

Листинг 4.12

```
from os.path import abspath # Функция для работы с путями
from datetime import datetime # Объект для работы со временем
```

```
from backtrader import Cerebro, Strategy, analyzers, sizers # Компоненты
from backtrader.feeds import GenericCSVData # Класс для загрузки CSV

# Импорт индикаторов
from backtrader.indicators import SimpleMovingAverage, CrossOver
from pandas import DataFrame # Объект сводной таблицы Pandas

class SingleSma(Strategy):
    params = ( # Параметры индикатора SMA
        "SMAperiod": 19
    ) # Период для быстрой скользящей средней

    def __init__(self): # Инициализация стратегии и временных рядов
        # Инициализируем две скользящие средние
        self.close = self.data.close
        self.order = None # Храним ордер
        self.sma = SimpleMovingAverage(self.data,
            period=self.params.SMAperiod)

    def next(self): # Появление новой свечи и логика стратегии
        if not self.position: # Если нет открытой позиции
            # Цена закрылась выше СМА
            is_signal_buy = self.close[0] > self.sma[0]
            if is_signal_buy: # открываем сделку
                print("Совершаю сделку")
                self.order = self.buy()
        else: # Если позиция есть
            # Цена закрылась ниже СМА
            is_signal_sell = self.close[0] < self.sma[0]
            if is_signal_sell:
                print("Совершаю продажу")
                self.order = self.sell()

    def stop(self): # Завершение запуска торговой системы
        print("Результат бектеста")
        print("Параметр индикатора SMA", self.params.SMAperiod)
        print("Конечный капитал", self.broker.getvalue())
        print()

# Подготовка источника свечных данных
# MyCSVData основной класс для загрузки CSV файла
class MyCSVData(GenericCSVData):
    # В параметрах указываем какой столбец CSV
    # за какие данные отвечает
    params = (
        "dtformat": "%Y-%m-%d", # Формат времени для парсинга
        "datetime": 0,
        "open": 1,
```

```
        "high": 2,
        "low": 3,
        "close": 4,
        "volume": 5,
        "openinterest": -1, }# -1 это последний столбец

csv_file_path = abspath("./data/LTCUSDT_D.csv") # Путь к источнику данных
data = MyCSVData( # Объект с источником свечных данных
    dataname=csv_file_path,
    fromdate=datetime(2022, 1, 24),
    todate=datetime(2024, 8, 10),
    reverse=False)

# Подключение источника данных, аналитика и запуск бэктеста
cerebro = Cerebro() # Создаём объект Церебро. Мозг Бектрейдера
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data

#  Запуск стратегии на оптимизацию
cerebro.optstrategy(SingleSma, SMAperiod=range(19, 40))
cerebro.broker.setcash(10000.0) # Устанавливаем стартовый баланс
cerebro.addsizer(sizers.PercentSizer, percents=95) # Позиция в %
cerebro.broker.setcommission(commission=0.0003) # Комиссия брокера

# Прикручиваем аналитические данные с псевдонимами в аргументе _name
# Годовой возврат инвестиций
cerebro.addanalyzer(analyzers.Returns, _name="returns")

# Максимальная просадка
cerebro.addanalyzer(analyzers.DrawDown, _name="drowd")

# Показатель эффективности
cerebro.addanalyzer(analyzers.SharpeRatio, _name="sharpe")

# Анализатор трейда
cerebro.addanalyzer(analyzers.TradeAnalyzer, _name="TradeAnalyzer")

results = cerebro.run(maxcpus=1) # Запускаем бэктест на 1 ядре
formating_list = [] # Список для результатов [[1,2,3], [1,2,3], [1,2,3]]

for result in results: # Перебор всех бектестов
    try: # Пытаемся получить результат трейда
        balance = result[0].analyzers.TradeAnalyzer.get_analysis()
["pnl"]["net"]["total"]
    except: # Если трейдов не было и ключи отсутствуют
        # Обрабатываем ошибку и записываем 0 в баланс
        balance = 0 # Не было трейдов
```

```
    # Вытягиваем только небходимые показатели для таблицы Pandas
    prepare_data = [
        result[0].params.SMAperiod,
        result[0].analyzers.returns.get_analysis()["rnorm100"],
        result[0].analyzers.drowd.get_analysis()["max"]["drawdown"],
        result[0].analyzers.sharpe.get_analysis()["sharperatio"],
        balance]

    # Отправляем отформатированные данные в список
    formating_list.append(prepare_data)

# Подготавливаем столбцы для сводной таблицы
pands_dataframe = DataFrame(formating_list, columns = ["SMAperiod",
"RETUNR%", "DROWD", "SHARPE", "BANK"])
print(pands_dataframe.sort_values("RETUNR%")) # Сортируем по RETUNR%
```

Обратите внимание, что после функции next() добавлена функция stop(). Эта функция выведет результат по завершении бэктестинга.

Далее мы подключаем необходимые анализаторы и даём им соответствующие псевдонимы. Анализаторы следующие: возврат инвестиций с псевдонимом RETUNR%, максимальная просадка с псевдонимом DROWD, RATIO для показателя эффективности инвестиций и TradeAnalyzer, чтобы получить окончательную сумму бэктеста. Изучайте документацию и найдете больше примеров выборки аналитических данных.

Теперь в переменной RETURNS находится список списков. Для того чтобы передать данные в Pandas, их необходимо подготовить. В строке `formating_list = []` мы создали пустой список для выборки данных.

Далее запустили цикл по показателям всех диапазонов бэктеста. На каждый бэктест был создан собственный список, который хранится в промежуточной переменной `prepare_data`.

В каждом списке бэктеста хранятся аналитические данные, которые мы попытаемся извлечь по ключу. Если трейдов не было и мы обратимся к ключу `result[0].analyzers.TradeAnalyzer.get_analysis()["pnl"]["net"]["total"]`, то получим ошибку, и программа упадет. Для отслеживания критически важных частей кода есть конструкция TRY/EXCEPT.

В блок TRY помещается критически важный кусок кода, если происходит ошибка, то срабатывает блок EXCEPT. Тем самым мы убережём программу от падения. В данном примере мы просто сохраним баланс по ключу PNL, дабы перехватить баланс на момент окончания бэктеста. В случае если трейдов не было, мы просто присваиваем переменной *balance* значение 0.

Затем мы поочередно выбираем аналитические данные по псевдонимам, созданным во время подключения анализаторов, и добавляем подготовленные данные в список **prepare_data**.

Сначала вытаскиваем текущий показатель SMA, затем RETURNS — годовой возврат инвестиций, DDOWN — максимальная просадка, RATIO — коэффициент возврата инвестиций и финальный баланс.

Далее просто отправляем подготовленный список в общий список для всех бэктестов и на выходе получаем отформатированный двумерный список.

Далее, вместо cerebro.addstrategy() мы использовали cerebro.opt_strategy() с функцией range(). Благодаря функции range() мы передаём НЕ одно значение, а диапазон целых чисел. Теперь в параметр SMA будет поочередно подставляться каждое значение от 8 до 64, а Pandas покажет конечный результат в виде сводной таблицы.

Далее мы используем Пандас, для того чтобы отобразить табличный вывод результатов в консоли. Пандас позволяет элегантно структурировать, отсортировать и вывести данные.

Мы используем конструктор объекта DataFrame(), который импортировали в начале файла, и передаём список *formating_list*, подготовленный ранее.

Далее в columns = ["SMAperiod", "RETUNR%", "DROWD", "SHARPE", "BANK"]) мы указываем, какие колонки за что отвечают.

Соответственно, значение списка formating_list по индексу 0 попадёт в колонку SMAperiod, значение по индексу 1 попадёт в колонку RETUNR%, значение по индексу 2 — в колонку DROWD, значение по индексу 3 — в колонку SHARPE, значение по индексу 4 — в колонку BANK.

Строка `pands_dataframe.sort_values("RETUNR%")` сортирует данные по указанной колонке. Лучший показатель по прогону в диапазоне, указанном в функции range(), окажется последним в списке.

Вывод в консоли для BTC должен выглядеть примерно так:

```
SMAperiod     RETUNR%         DROWD        SHARPE          BANK
20          39 -28.540069   74.074790  -3.401786      -7102.652537
19          38 -25.655081   70.002258  -3.408834      -6647.514817
42          58 6689.739332  49.850290  -0.684692      16689.403892
```

В конце исходника в строке `results = cerebro.run(maxcpus=1)` добавлен аргумент maxcpus, чтобы указать, сколько процессоров использовать для бэктеста. Не на всех компьютерах получается корректно запустить бэктест на нескольких ядрах, по этой причине лучше указать 1 ядро.

При бэктесте на одном ядре скорость выполнения будет ниже, зато вы НЕ столкнетесь с дополнительными ошибками. Запустите бэктест.

Теперь мы можем точно быть уверенными, что SMA с периодом 42 потенциально может дать 58 процентов годовых.

В качестве самостоятельного задания попробуйте добавить в стратегию ещё одну SMA и отследить наилучший показатель по пересечениям двух скользящих.

4.28. Оптимизация торговой стратегии по пересечению двух SMA

Для наилучшего понимания работы оптимизатора стратегий давайте заставим Бэктрейдер найти для нас наилучшие показатели пересечения двух скользящих.

Ниже приведён пример кода с комментариями.

Листинг 4.15

```
from os.path import abspath # Функция для работы с путями
from datetime import datetime # Объект для работы со временем
from backtrader import Cerebro, Strategy, analyzers, sizers # Компоненты
from backtrader.feeds import GenericCSVData # Класс для загрузки CSV

# Импорт индикаторов
from backtrader.indicators import SimpleMovingAverage, CrossOver
from pandas import DataFrame # Объект свобной таблицы Pandas

class SMACrossover(Strategy):
    params = { # Параметры индикатора быстрой и медленной SMA
        "fast_length": 10, # Период для быстрой скользящей средней
        "slow_length": 41, # Период для медленной скользящей средней
    }

    def __init__(self): # Инициализация стратегии и временных рядов
        # Инициализируем две скользящие средние
        self.sma_fast = SimpleMovingAverage(self.datas[0],
            period=self.params.fast_length)
        self.sma_slow = SimpleMovingAverage(self.datas[0],
            period=self.params.slow_length)

        # Отслеживаем, находится ли быстрая SMA выше медленной SMA
        self.crossover = CrossOver(self.sma_fast, self.sma_slow)

    def next(self): # Появление новой свечи и логика стратегии
        # Купить, если быстрая SMA пересекается над медленной SMA
        if not self.position:  # Если нет позиции на рынке
            if self.crossover > 0:  # Если быстрое превышает медленное
                self.buy() # Открываем позицию

        # Продать, если быстрая SMA пересекается ниже медленной SMA
        elif self.crossover < 0:  # Если быстрое пересекает медленное
            self.close()  # Закрываем позицию

    def stop(self): # Завершение запуска торговой системы
        print("Результат бектеста")
        print("Параметр индикатора fast_length SMA",
            self.params.fast_length)
        print("Параметр индикатора slow_length SMA",
            self.params.slow_length)
        print("Конечный капитал", self.broker.getvalue())

# Подготовка источника свечных данных
# MyCSVData основной класс для загрузки CSV файла
```

```
class MyCSVData(GenericCSVData):
    # В параметрах указываем какой столбец CSV файла
    # за какие данные отвечает
    params = (
        "dtformat": "%Y-%m-%d", # Формат времени для парсинга
        "datetime": 0,
        "open": 1,
        "high": 2,
        "low": 3,
        "close": 4,
        "volume": 5,
        "openinterest": -1, # -1 это последний столбец
    )

csv_file_path = abspath("./data/LTCUSDT_D.csv") # Путь к источнику данных
data = MyCSVData( # Объект с источником свечных данных
    dataname=csv_file_path,
    fromdate=datetime(2022, 1, 24),
    todate=datetime(2024, 8, 10),
    reverse=False)

# Подключение источника данных, аналитика и запуск бэктеста
cerebro = Cerebro() # Создаём объект Церебро. Мозг Бектрейдера
cerebro.adddata(data) # Добавляем поток данных. Хранится в self.data
cerebro.optstrategy(SMACrossover, fast_length=range(19, 22),
    slow_length=range(19, 25)) #  Оптимизация по двум параметрам

cerebro.broker.setcash(10000.0) # Устанавливаем стартовый баланс
cerebro.addsizer(sizers.PercentSizer, percents=95) # Размер позиции в %
cerebro.broker.setcommission(commission=0.0003) # Комиссия брокера

# Прикручиваем аналитические данные с псевдонимами в аргументе _name
# Годовой возврат инвестиций
cerebro.addanalyzer(analyzers.Returns, _name="returns")

# Максимальная просадка
cerebro.addanalyzer(analyzers.DrawDown, _name="drowd")
# Показатель эффективности
cerebro.addanalyzer(analyzers.SharpeRatio, _name="sharpe")

# Анализатор трейда
cerebro.addanalyzer(analyzers.TradeAnalyzer, _name="TradeAnalyzer")

 # Запускаем бэктест на 1 ядре и сохраняем результат
results = cerebro.run(maxcpus=1)
formating_list = [] # Список для результатов [[1,2,3], [1,2,3], [1,2,3]]

for result in results: # Перебор всех бектестов
```

```
    try: # Пытаемся получить результат трейда
        balance = result[0].analyzers.TradeAnalyzer.get_analysis()["pnl"]
["net"]["total"]
    except: # Если трейдов не было и ключи отсутствуют
        # Обрабатываем ошибку и записываем 0 в баланс
        balance = 0 # Не было трейдов

    # Вытягиваем только небходимые показатели для таблицы Pandas
    prepare_data = [
        result[0].params.fast_length,
        result[0].params.slow_length,
        result[0].analyzers.returns.get_analysis()["rnorm100"],
        result[0].analyzers.drowd.get_analysis()["max"]["drawdown"],
        result[0].analyzers.sharpe.get_analysis()["sharperatio"],
        balance]

    # Отправляем отформатированные данные в список
    formating_list.append(prepare_data)

# Подготавливаем столбцы для сводной таблицы
pands_dataframe = DataFrame(formating_list, columns = ["fast_length",
"slow_length", "RETUNR%", "DROWD", "SHARPE", "BANK"])
print(pands_dataframe.sort_values("RETUNR%")) # Сортируем по BANK
```

С помощью Пандас мы научились красиво и просто выводить сводные данные на экран и находить наилучшие результаты бэктеста. На эти данные можно смело полагаться в ручной торговле. Для алгоритмической торговли (на мой субъективный взгляд) эта комбинация слишком запаздывает.

Результат для BTC будет примерно следующий:

fast_length	slow_length		RETUNR%	DROWD	SHARPE	BANK
90	12	37	16.782931	35.705088	0.602682	7684.486115
52	11	35	17.007754	35.958007	0.596129	7809.910800
53	11	36	17.697007	34.971125	0.636669	8198.468202

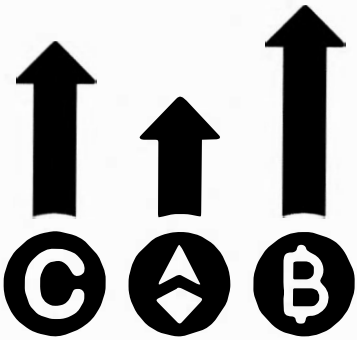
4.29. Итоги раздела Бэктестинг

Мы протестировали все самые популярные классические индикаторы и стратегии, которые работают уже несколько десятков лет и будут продолжать работать дальше.

Полный список изученных индикаторов: SMA, RSI, STOHASTIC, MACD, BOLLINGER BANDS, PIVOT, FIBO, CROSSOVER, CROSSUP, CROSSDOWN, LOWEST, HIGHEST, ATR, свечи Хайкин-Аши.

Также мы научились добавлять специфические индикаторы, такие как измеритель волатильности рынка ATR и свечи Хайкин-Аши. Научились определять уровни поддержки/сопротивления и находить разворотные точки рынка. Научились определять локальные минимумы и максимумы для умных стопов и отложенных тейк-профитов.

В том числе научились комбинировать индикаторы и множественные тайм-фреймы. Научились подключать несколько потоков свечных данных. Провели немало успешных бэктестов и в целом вооружились полным набором институционального трейдера. Можно сказать, что единственная тема, которой мы не коснулись в рамках данного руководства, — это создание собственных индикаторов Бэктрейдера.



Глава 5.

Живая торговля. Пенсионный фонд своими руками

Содержание главы:

5.1. Ваш первый торговый робот. Размещение сделок на бирже

Уверен, вы НЕ знаете, что 90% всех средств, которыми управляет многотриллионный фонд BlackRock, — это пенсионные накопления американцев. Несмотря на то, что эти ребята действительно эффективно управляют средствами своих вкладчиков, они активно участвуют в создании так называемых цифровых ошейников CBDC.

Я считаю, что инвестировать в собственные разработки, самостоятельно нести ответственность за средства, контролировать стратегию инвестирования и наслаждаться автоматизацией куда увлекательнее.

К текущему моменту вы более чем достаточно понимаете, как устроен Бэктрейдер, какие существуют части внутри его системы, и готовы выстроить собственную торговую систему.

Цель финального раздела этой книги создать запредельно простую и надежную систему для накопительной части своих финансов. Что-то, что обгонит по доходности банковский депозит, отобьет инфляцию, сможет работать на падающем рынке и позволит при этом иметь постоянный доступ к своим средствам.

Многие ошибочно полагают, что банковский депозит или облигация являются исключительно надежной, консервативной стратегией инвестирования. Я с этим в корне не согласен. Нет гарантий того, что предприятие или банк, одолжившие у вас деньги, не разорятся.

В самой простой схеме банк возьмет ваши средства, купит на них облигации, а разницу в процентах заберет себе. Получается, что, инвестируя в консервативные на первый взгляд инструменты, мы рискуем так же, как и

размещая свои деньги на бирже. Просто банк делает это за нас. Зачем нам этот посредник?

Безусловно, держать средства на криптовалютной бирже — это риск, но никто не мешает запустить 10 ботов на 10 биржах или создать сигнального бота по исходнику, представленному ниже, покупать и продавать монеты вручную. Главное — не забывать выводить прибыль на холодный или кастодиальный кошелёк.

Мечта любого трейдера — иметь возможность торговать прибыльно в каждой фазе рынка. Можно обложиться десятками индикаторов, пытаться торговать сотнями инструментов, но так и не добиться желаемого результата, забывая, что главная цель — не потерять средства.

Мы постараемся выстроить торговую систему таким образом, чтобы вся сумма депозита работала практически каждый день, но при этом риск потерять всё был сведен к минимуму. Т.е. мы будем считать успехом ситуацию, в которой получим прибыль 31% годовых при просадке 29% на падающем рынке.

Как ни странно, для получения до гениальности простого решения по созданию прибыльной стратегии мы вернемся в самое начало, когда пользовались сделками исключительно по состоянию открытия и закрытия дневных свечей.

Как уже упоминалось, в рамках данного руководства для снятия нагрузки мы будем использовать открытие сделок только на дневных таймфреймах.

Стратегия, по которой мы будем открывать сделки, предельно проста и звучит так:

> *"Если цена закрытия текущей дневной свечи ниже, чем цена открытия вчерашней — открываем сделку".*

```
is_signal_buy = self.data.close[0] < self.data.open[-1]
if is_signal_buy:
    self.buy() # Открываем сделку
```

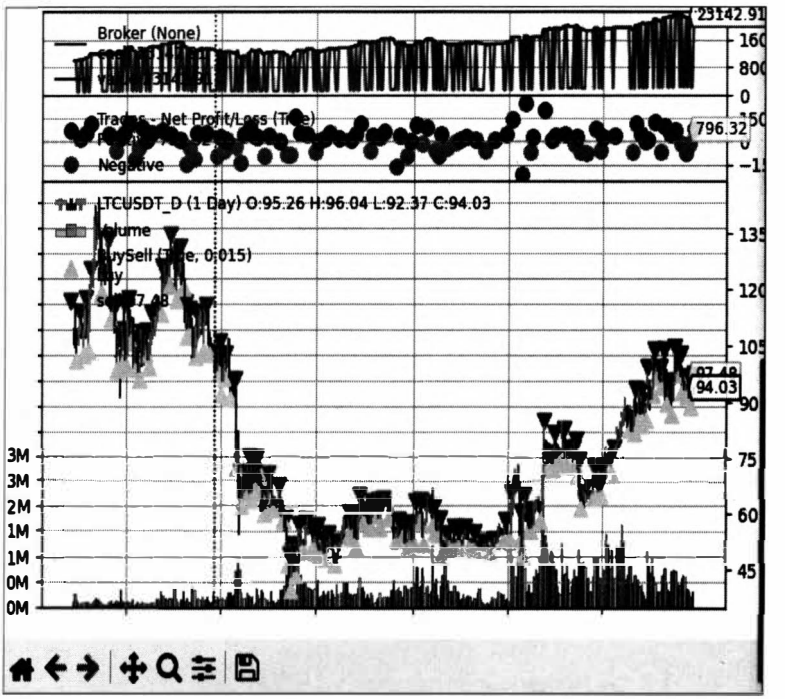
Как только цена закрытия текущей свечи окажется выше цены открытия предыдущей, мы закрываем позицию.

```
is_signal_cell = self.data.close[0] > self.data.open[-1]
if is_signal_cell: # Если в is_signal_cell лежит True
    self.close() # Закрываем сделку
```

Данная стратегия позволяет прибыльно торговать даже на падающем рынке. Там, где при удержании позиции после покупки на пике цены большинство получит -50%, мы извлечем прибыль около 70% годовых.

Результат бэктеста стратегии BakalovD для LTC вы можете увидеть ниже.

```
Конечный капитал: 23140.2912318247218
Годовой возврат инвестиций: 69.88645071589382 %
Максимальная просадка: 30.500473818764565 %
Чистая прибыль: 13140.2912318247218
Количество прибыльных сделок: 66
Количество убыточных сделок: 34
Всего сделок: 100
Коэффициент побед: 66.0%
Максимальная прибыль 2529.0795848372127
Максимальный убыток -2141.676363245345
```



Изображение 5.1.

В конце концов, после достижения максимальных показателей цены робота можно просто отключить, дождавшись коррекции.

Мы подобрались к финальному разделу, в котором автоматизируем торговлю на бирже. Есть две новости, одна хорошая, другая плохая.

На момент написания книги существует буквально две криптобиржи, которые позволяют автоматизировать свою торговлю через Бэктрейдер напрямую.

Первая — это Бинанс, и она закрыта для пользователей РФ, вторая — Bybit, которую мы активно используем.

Bybit — плохая биржа. Я не люблю Bybit. Но они неплохо справляются с проблемами внутри компании, а на их свечах бэктесты показывают наилучшие результаты по прибыли.

Bybit пытаются создать дополнительное препятствие для ботоводов и для доступа к API через Бэктрейдер требуют предварительного пополнения баланса на 550 долларов. Чуть позже будет описан способ это обойти.

Хотя эти 550 долларов и являются возвратными средствами, но не каждый готов на старте внести их на биржу.

Хорошая новость, что мы можем автоматизировать нашу стратегию напрямую через API биржи без использования Бэктрейдера. Это позволит протестировать стратегию в реальном времени и, если результат нас удовлетворит, заказать адаптацию библиотеки на фрилансе под любую биржу. Я думаю, что это будет стоить в районе 300 долларов.

Также в чате, посвященном этой книге, стоит следить за новостями, вероятно, мы общими силами добьемся выпуска библиотеки для одной из бирж, лояльных к российской аудитории, и при этом по более приятным комиссиям, чем Bybit.

Для тех, кто планирует свою торговлю на МОС-бирже или других фондовых российских биржах, существует достаточно много расширений, список которых с удовольствием предоставим по личному запросу.

5.2. Получаем приватные ключи Bybit

Приватный доступ позволяет размещать и отменять сделки на бирже, управлять позицией, следить за балансом, управлять переводом средств на другие аккаунты, создавать саб-аккаунты и многое, многое другое.

Чтобы НЕ платить ренту за подключение Backtrader Bybit, необходимо создать аккаунт с реферальным кодом **KXLXXE%230**.

Это код создателя библиотеки:

www.github.com/WISEPLAT/backtrader_bybit

Если у вас уже есть аккаунт на Bybit, вы можете создать новый с указанной выше реферальной ссылкой и перенести свой KYC на новый аккаунт.

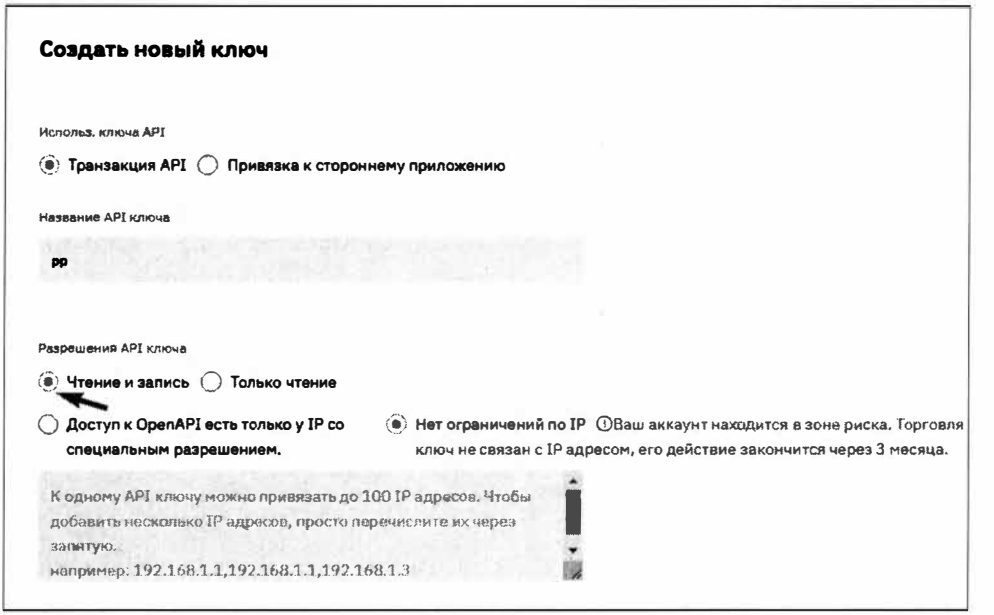
Я рассчитываю на то, что вы в состоянии самостоятельно зарегистрироваться на бирже Байбит и перенести верификацию аккаунта. Поисковики вам помогут.

Вам останется получить ключи доступа, которые мы будем использовать для получения исторических данных, а также для реальной торговли.

1. Логинимся в аккаунте и включаем двухфакторную авторизацию с помощью Google.
2. Заходим по ссылке:

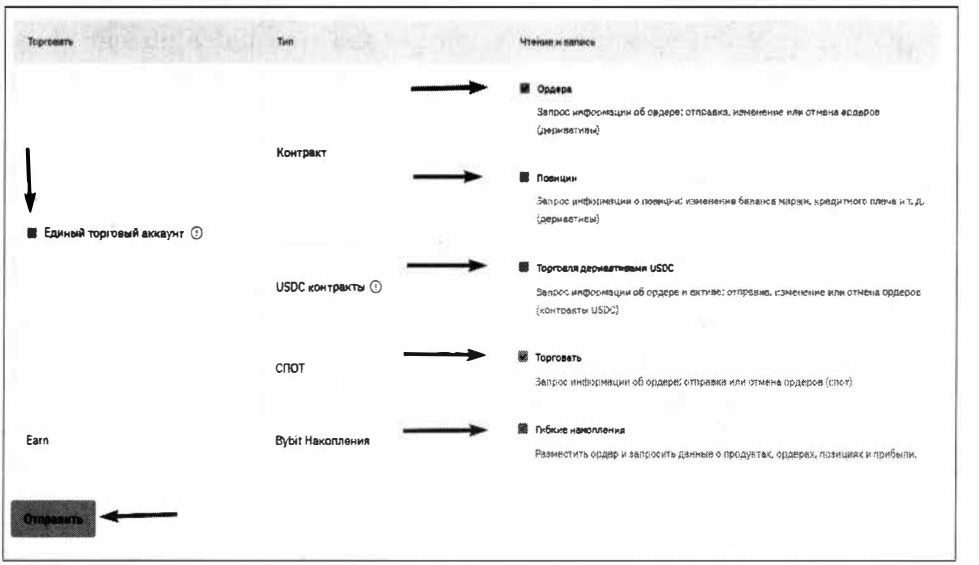
   https://www.bybit.com/app/user/api-management

3. Нажимаем на кнопку **Создать новый ключ**.
4. Выбираем **Транзакция API**, указываем имя ключа и устанавливаем **Чтение и запись**.



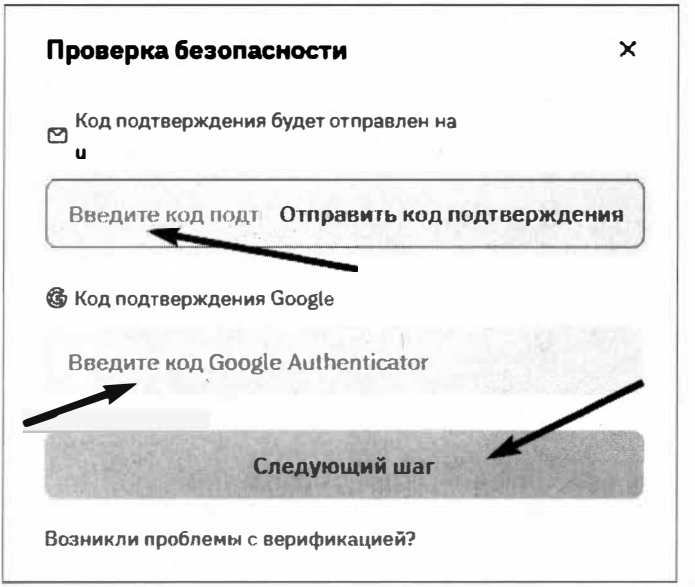
Изображение 5.2.

5. Пролистаем чуть ниже, проставляем все галочки напротив Единого Торгового Аккаунта и нажимаем кнопку **Отправить**.



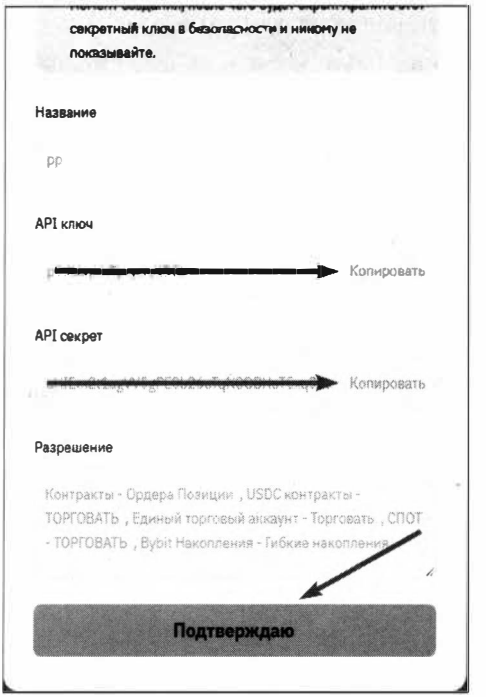
Изображение 5.3.

6. Подтверждаем проверочные коды.



Изображение 5.4.

7. Копируем и сохраняем API-ключ и API-секрет.



Изображение 5.5.

Всё готово. По истечении 3 месяцев вам необходимо будет перевыпустить ключи. Также можете узнать внешний IP вашего сервера и вписать его в список исключений. В таком случае ключи заново выпускать не придётся.

5.3. Робот на BACKTRADER LIVE

В первую очередь установите библиотеку backtrader_bybit:

```
pip install backtrader_bybit
```

Если всё хорошо, в терминале вы увидите:

```
Downloading backtrader_bybit-2.0.9-py3-none-any.whl (18 kB)
Installing collected packages: backtrader_bybit
Successfully installed backtrader_bybit-2.0.9
```

Поскольку стратегия родилась во время написания этой книги и в Интернете, а также среди моих товарищей, профессиональных трейдеров, подобной стратегии мы НЕ обнаружили, назовем её моим именем.

Создайте проект BakalovDStrategy.py и перенесите заготовку будущей стратегии, как представлено в листинге 5.1.

Листинг 5.1

```
from datetime import datetime, timedelta # Объекты для работы со временем
from backtrader import Cerebro, Strategy, TimeFrame # Компоненты БТ
from backtrader_bybit import BybitStore # Для работы с хранилищем Bybit

BYBIT_API_KEY = "lt1DIs456zVCBX4rXm"
BYBIT_API_SECRET = "7J8690QsHs8HDpIow5wWQq6ZCVtpuVm"
BYBIT_ACCOUNT_TYPE = "UNIFIED"  # UNIFIED or CONTRACT

class BakalovD(Strategy):
    params = (  # Параметры торговой системы.
        "coin_target": "",
        "stop_percent": -2.5, # Максимальная просадка от цены входа
```

```
}

def __init__(self):
    # Инициализация стратегии
    order = None # Храним ордер в свойстве
    self.executed_price = None # Цена исполненного ордера

def notify_order(self, order):
    # Срабатывает когда изменяется статус ордера
    # Если заявка исполнена
    if order.status in [order.Completed]:
        if order.isbuy(): # Заявка была покупку
            self.executed_price = order.executed.price # Цена открытия
        elif order.issell(): # Заявка была на продажу
            self.executed_price = None # Сбрасываем цену входа
        self.order = None # Очещаем отработавший ордер
        self.executed_price = None # Цена исполненного ордера

def next(self): # Срабатывает при поступлении новой свечи
    status = self.data._state  # 0 - Live данные, 1 - История, 2 - None
    _interval = self.broker._store.get_interval(self.data._timeframe,
        self.data._compression)
    ticker_name = self.data._name  # Извлекаем имя тикера

    if status in [0, 1]:
        if status: _state = "False - History data" # Однострочный IF
        else: _state = "True - Live data" # Однострочный ELSE

        print(
            self.data._name,
            _interval,  # Ticker timeframe
            self.data.open[0],
            self.data.close[0],
            _state)

        if status == 0:  # Live торговля
            aval_balance = self.broker.getcash()  # Свободный баланс
            print("Свободный баланс", aval_balance)
            get_90_percents = (aval_balance * 90) / 100 # Получаем 90%
            print("90% от баланса в USDT", get_90_percents)
            size = get_90_percents / cur_price
            size = round(size, 1) # Необходимо округлять до целого
            print("Размер позиции", size)

            cur_price = self.data.close[0] # Получаем текущую цену

            if not self.position: # Если позиции нет
```

```
                    # Если закрытие текущей свечи ниже
                    # чем открытия предыдущей
                    is_signal_buy = self.data.close[0] < self.data.open[-1]
                    if is_signal_buy:
                        print("Покупаю")
                        self.order = self.buy(data=data, # Открываем ордер
                            exectype=Order.Market, # Рыночный ордер
                            size=size) # Размер позиции
                else: # Если позиция уже есть
                    print("Цена открытия сделки:", self.executed_price)
                    print("Цена закрытия текущей свечи:",
                        self.data.close[0])
                    # Если закрытие текущей свечи выше
                    # чем открытие предыдущей
                    is_signal_cell = self.data.close[0] > self.data.open[-1]
                    if is_signal_cell: # Если в is_signal_cell лежит True
                        print("Продаю")
                        self.order = self.close() # Закрываем сделку
                        return # Дальше не идём, ждём следующей свечи

                    # Проверяем стоп-лосс
                    down_percent = (self.data,close[0]-self.executed_price)
                                                        / cur_price * 100
                    print("Разница цены в процентах:", down_percent)

                    if down_percent <= self.params["stop_percent"]:
                    # Максимальная просадка
                        print("Сработал стоп при разнице процентов:",
                            down_percent)
                        self.close() # Продаём сделку с убытком
                        return # Дальше не идём, ждём следующей свечи

# Исторические/live бары тикера
cerebro = Cerebro(quicknotify=True) # Создём объект Cerebro
coin_target = "USDT"  # Тикер в котором будут осуществляться расчёты
symbol = "LTC" + coin_target  # Тикер которым торгуем
accountType = BYBIT_ACCOUNT_TYPE # Тип аккаунта Bybit

session = BybitStore( # Объект/хранилище для работы с Bybit
    api_key=BYBIT_API_KEY,
    api_secret=BYBIT_API_SECRET,
    coin_target=coin_target,
    testnet=False,
    accountType=accountType)

broker = session.getbroker() # Получаем объект с информацией от брокера
cerebro.setbroker(broker) # Устанавливаем брокера
```

```
# Берём данные за последние 2 дня
from_date = datetime.now() - timedelta(hours=24*2)

# В каком ТФ подгружаем историю и live данные
data = session.getdata(timeframe=TimeFrame.Days,
    compression=1, # До какого размера компрессируем свечи
    dataname=symbol,
    start_date=from_date,
    from_date=from_date,
    LiveBars=True)

# Добавляем LIVE данные. Доступны в self.data вместе с историей
cerebro.adddata(data)

# Добавляем торговую систему
cerebro.addstrategy(BakalovD, coin_target=coin_target)
cerebro.run()  # Запуск торговой системы
#cerebro.plot()  # НЕ Рисуем график. В лайв режиме может упасть
```

Пройдемся с низа.

- В строке `cerebro = Cerebro(quicknotify=True)` создаём объект Cerebro с быстрыми уведомлениями.
- В `coin_target` находится тикер, в котором будут осуществляться расчёты.
- В `symbol` находится тикер, которым торгуем. В нашем случае LTC.
- В объекте session = BybitStore(...) находится объект/хранилище для работы Bybit. После подключения к брокеру в строке `broker = session.getbroker()` получаем объект с информацией о брокере.
- В строке `cerebro.setbroker(broker)` устанавливаем параметры брокера.
- В переменной *from_date* с помощью функции timedelta() берем диапазон за последние 2 дня. В объект **data**, который заполнил метод session.getdata(), получаем данные об исторических и LIVE-свечах.
- В параметре *compression=1* указываем, до какого количества сжимаем свечи.

- В строке `cerebro.adddata(data)` добавляем исторические и LIVE-данные. В классе Стратегии доступны в self.data.
- В строке `cerebro.addstrategy(BakalovD,coin_target=coin_target)` добавляем стратегию с заполненным параметром *coin_target*.
- В финале cerebro.run() запускаем торговую систему.
- В методе next() описали условия входа в рынок и выхода.

За одним исключением, что отныне за поступлением новых свечей и за сделками будет следить Бэктрейдер.

После запуска, если всё в порядке, вы увидите примерно следующие данные:

```
739334.125 LTCUSDT D 91.44 95.58 90.8 93.8 211335.95805 False -
History data
739335.125 LTCUSDT D 93.8 94.78 91.88 94.38 114611.84001 False -
History data
Live started for ticker: LTCUSDT
```

Данный код лучше оформить в виде сервиса. Пример находится в конце главы.

Если при запуске робота вы получили ошибку: "*ValueError: could not convert string to float:*", значит библиотеку backtrader_bybit всё ещё НЕ актуализировали под новое API Bybit.

Чтобы исправить ситуацию:

1. Зайдите в исходный код файла: C:\Users\Samsung\AppData\Roaming\Python\Python313\site-packages\backtrader_bybit\bybit_store.py

2. Найдите функцию get_asset_balance()

3. Внутри неё отыщите код:

```
balance[_coin["coin"]] = {"free": _coin['availableToWithdraw'],
                          "usdValue": _coin['usdValue'],
                          "locked": locked, }
```

4. Замените на представленный ниже:

```
response = self.bybit_session.get_transferable_amount(coinName="USDT")
balance[_coin["coin"]] = {"free": response["result"]["availableWithdrawal"],
                          "usdValue": response["result"]["availableWithdrawal"],
                          "locked": locked, }
```

5. Сохраните файл и перезапустите робота.

К сожалению в нашем приватном направлении, устаревание библиотек не редкость и придётся учиться решать подобные проблемы самостоятельно. Также не забывайте обращаться за помощью в чат книги.

5.4. Торговый робот на HTTP-запросах

Напишем HTTP-версию бота, которую легко можно адаптировать под любую другую биржу, где нет библиотеки Бэктрейдер Лайв.

Этапы реализации примерно следующие:

1. Настроить время сервера на работу биржи или подстроить Cron. По МСК это 3 утра.

2. Перезапускать скрипт в 23:59:50. В это время мы получим закрытие текущей дневки.

3. Сравниваем закрытие текущей свечи с открытием предыдущей.

4. Если совпало по стратегии, открываем сделку по рынку.

5. Цену исполненного ордера сохраняем в глобальной переменной.

8. Если скрипт перезапустился, а ордер ещё не исполнен, проверяем условия для тейка.

9. Если условий для тейка нет, проверяем условия для стопа.

10. На всех этапах присылаем уведомления в Телеграм.

В листинге 5.2 по старинке представлены заготовки функций Робота.

Листинг 5.2

```
from pybit.unified_trading import HTTP  # Класс Bybit
from pprint import pprint # Продвинутый print()
from requests import post # Для POST запросов
from time import sleep # Для сна

KEY = "" # Bybit-ключ
SECRET = "" # Bybit-секрет
TGBOT = "" # TG BOT секрет
CHATID = "" # Your chat id
TGLINK = "https://api.telegram.org/bot"
start_balance = 10000 # Стартовый баланс

current_position_confirm_price = None # Цена, по которой зашли
current_position_size = None # Размер позиции в LTC
current_position_balance = None # Текущий баланс позиции

session = HTTP( # Объект для работы с Bybit
    testnet=False,
    api_key=KEY,
    api_secret=SECRET)

def send_message(msg="LTC ONLY CANDELS NOTIFY"):
    # Функция отправляет сообщение пользователю
    # Принимает один аргумент msg (сообщение пользователю)
    # Ничего не отдаёт
    url = TGLINK+TGBOT+"/sendMessage?chat_id="+CHATID+"&text="+msg
    response = post(url)
    return # Завершаем функцию

def time_formater(unix=1731196800000):
    # Форматируем юникс в объект даты
    # Принимаем один аргумент по умолчанию
    # Отдаём объект дейтайм
    timestamp = unix / 1000
    dt_object = datetime.fromtimestamp(timestamp)
    return dt_object # Завершаем функцию

def get_balance_info()
    # Получаем текущий баланс на бирже
    # Ничего не принимаем
    # Отдаём баланс текущего пользователя
    pass

def get_symbol_current_price(symbol="LTCUSDT"):
    # Получаем текущую стоимость монеты
    # Принимает один аргумент. Со значением по умолчанию LTCUSDT.
```

```
    # Возвращает текущую цену
    # Получаем последнюю свечу
    # Цена закрытия любой текущей свечи — это актуальная цена
    pass

def get_candles(symbol="LTCUSDT", interval="D", limit=2):
    # Получаем свечи по тикеру, интервалу и лимиту.
    # Принимает три аргумента со значениями по умолчанию
    # Для большинства случаев достаточно будет последних двух свечей
    ''' Формат отдаваемых данных
    only_klines = [['1731196800000', '73.99', '77.67','73.7', '77.32',
    '512852.7', '39005660.315'],
    ['1731110400000', '72.79', '74.19', '71.57', '73.99', '423841.7',
    '30931005.826']]
    '''
    pass

def get_current_position(symbol="LTCUSDT"):
    # Принимает один аргумент. Со значением по умолчанию LTCUSDT.
    # Просто получаем сумму открытой позиции
    # Если сумма равна 0, значит позиции нет, поэтому возвращаем False
    pass

def calculate_position():
    # Функция, которая рассчитывает текущую позицию.
    # Ничего не принимает, ничего не отдаёт
    # Берем текущий баланс в usdt и получаем 90 процентов
    # Затем переводим в размер позиции и открываем ордер по рынку
    # Возвращает сумму, на которую входим в номинале монеты
    pass

def create_or_cancel_position(symbol="LTCUSDT",
   side="Buy", order_type="market", quantity="0.1"):
    # Функция открывает сделку либо закрывает уже существующую
    # Функция принимает четыре аргумента со значениями по умолчанию
    # Ничего не отдаёт
    pass

def check_stop_loss():
    # Функция для проверки условия стопа
    # Функция ничего не принимает и не отдаёт
    # Если функция запущена, она берет текущую цену актива
    # Смотрит цену открытия ордера
    # Если цена упала больше чем на 3 процента,
    # вызываем закрытие позиции
    pass
```

```
def run():
    # Функция для запуска бота. Ничего не принимает и не отдаёт.
    # Сначала функция проверяет, есть ли открытый ордер.
    # Если есть получаем последние дневные свечи.
    # Проверяем условие для закрытия ордера.
    # Если есть условие, закрываем позицию
    # и пишем в ТГ что произошёл трейд.
    # Пытаемся высчитать прибыль или убыток.
    # Если условие для закрытия не сработало пишем,
    # что позиция по прежнему открыта и запускаем проверку стопа.

    # Если ордера нет, проверяем, есть ли условие для входа.
    # Если условие подходит, рассчитываем сумму сделки.
    # и заходим в позицию.
    # Отправляем сумму и цену по которой ордер исполнился,
    # в сообщения Телеграм.
    # Если нет условия для входа, пишем в Телеграм,
    # что условия для входа сегодня нет.
    pass
```

Обратите внимание: если у переменной аргумента symbol="LTCUSDT" есть значение по умолчанию и при вызове функции ничего не передаём в неё, то переменная аргумента заполнится значением LTCUSDT.

Давайте поочередно реализуем каждую функцию Робота.

5.5. Реализация функции get_balance_info()

Приведите функцию get_balance_info() к следующему виду:

```
def get_balance_info():
    # Получает текущий баланс на бирже
    # Ничего не принимает
    # Отдаёт баланс текущего пользователя
    response = session.get_transferable_amount(coinName="USDT")
    aval_bal = response["result"]["availableWithdrawal"]
    # Формируем уведомление
    msg = "Доступный баланс USDT: " + str(aval_bal)
    send_message(msg) # Уведомляем в ТГ
    return float(aval_bal) # Отдаём баланс
```

Если всё написано правильно, вы получите уведомление в Телеграме о состоянии своего баланса.

Функция get_balance_info() не принимает никаких аргументов.

Метод session.get_wallet_balance() отдаёт доступный баланс с различных кошельков биржи Bybit.

В полученном ответе выбираем баланс USDT, доступный для вывода.

Уведомляем в Телеграм о доступном балансе. В последней строке в качестве завершения работы функции отдаём сумму текущего доступного баланса.

5.6. Реализация функции get_candles()

Приведите функцию get_candles() к следующему виду:

```
def get_candles(symbol="LTCUSDT", interval="D", limit=2):
    # Получаем свечи по тикеру, интервалу и лимиту.
    # Для большинства случаев достаточно будет последних двух свечей
    # Для индикаторов 14-19 дней
    ''' ФОРМАТ СВЕЧЕЙ ОТВЕТА
    only_klines = [['1731196800000', '73.99', '77.67', '73.7',
    '77.32', '512852.7',
    '39005660.315']] '''

    print("Получаю данные на вчерашние свечи")
    response = session.get_kline(
        category="linear",
        symbol=symbol,
        interval=interval,
        limit = limit)

    # В 0 элементе будет текущая свеча
    only_klines = response["result"]["list"]

    for kline in only_klines:
        print("Open time", time_formater(int(kline[0])))
        print("Open price", kline[1])
        print("High price", kline[2])
        print("Low price", kline[3])
        print("Close price", kline[4])
        print("Volume", kline[5])
        print("Turnover", kline[6])
        print()
```

```
    return only_klines # Отдаём свечи ввиде списка списков

get_candles() # Сразу запускаем на тест
```

Если всё в порядке, вы увидите информацию о последних двух дневных свечах.

Функция get_candles() получает свечи по тикеру, интервалу и лимиту.

Метод session.get_kline() отдаёт свечи на указанный тикер. Для наших задач достаточно последних двух свечей, что отражено в аргументе limit=2 значением по умолчанию.

Свечи приходят в виде списка списков:

```
only_klines = [["1731196800000', '73.99', '77.67', '73.7',
'77.32', '512852.7', '39005660.315'],
['1731110400000', '72.79', '74.19', '71.57', '73.99', '423841.7',
'30931005.826']]
```

Каждая свеча имеет следующую структуру:

```
['1731196800000', '73.99', '77.67', '73.7', '77.32', '512852.7']
```

Наконец, мы отдаём свечи в качестве работы функции и останавливаем её выполнение.

5.7. Реализация функции get_current_symbol_price()

Приведите функцию get_symbol_current_price() к следующему виду:

```
def get_current_symbol_price(symbol="LTCUSDT"):
    # Получаем текущую стоимость монеты
    # Принимает один аргумент. Со Значением по умолчанию LTCUSDT
    # Возвращает текущую цену

    # Извлекаем последнюю свечу
    last_candle = get_candles(symbol=symbol, limit=1)
```

```
    # Цена закрытия текущей свечи = стоимость
    current_price = float(last_candle[0][4])
    print("Текущая цена", symbol, current_price)
    return current_price # Отдаём цену

get_symbol_current_price() # Сразу запускаем на тест
```

Если всё в порядке, вы увидите актуальную цену LTC.

Функция get_symbol_current_price() получает текущую стоимость монеты, принимает всего один аргумент *symbol* со значением по умолчанию LTCUSDT.

С помощью уже реализованной нами функции get_candles() получаем последнюю свечу.

Далее извлекаем из текущей дневной свечи цену закрытия. Она всегда отражает актуальную стоимость монеты. В конце отдаём текущую цену и завершаем функцию.

5.8. Реализация функции get_current_position()

Реализуйте функцию get_current_position(), как описано ниже.

```
def get_current_position(symbol="LTCUSDT"):
    # Импортируем глобальные переменные
    global current_position_confirm_price, current_position_size
    global current_position_balance
    # Просто получаем сумму открытой позиции
    # Если сумма равна 0, значит позиции нет и возвращаем False
    try:
        # Безопасно обращаемся к данным на открытую позицию
        response = session.get_positions(category="linear",
            symbol=symbol)
    except Exception as error:
        msg = "Ошибка при проверке позиции: " + str(error.message)
        print(msg)
        send_message(msg) # Уведомляем в Tr
        return # Аварийно завершаем функцию
```

```
    # Извлекаем данные на открытую позицию
    pos_balance = response["result"]["list"][0]["positionBalance"]
    size = response["result"]["list"][0]["size"]
    enter_price = response["result"]["list"][0]["avgPrice"]

    if size == "0":
        msg = "Нет открытых позиций"
        print(msg)
        send_message(msg) # Уведомляем в ТГ
        return False # Завершаем функцию
    else:
        current_position_balance =  pos_balance
        current_position_confirm_price = enter_price
        current_position_size = size
        print("Есть позиция на сумму:", pos_balance)
        print("Объём в LTC:", size)
        print("Цена входа:", enter_price)

        msg1 = "Позиция на сумму: " + str(pos_balance)
        msg2 = "/ Объём в LTC: " + str(size)
        msg3 = "/ Цена входа:" + str(enter_price)
        final_msg = msg1 + msg2 + msg3
        send_message(final_msg) # Отправляем в ТГ
        return True # Завершаем функцию

get_current_position() # Запускаем на тест
```

В переменную `current_position_balance` извлекли сумму позиции, в переменную `current_position_size` её размер, а в переменную `current_position_confirm_price` цену, по которой ордер был открыт.

Если в переменной `current_position_size` 0 позиций, значит открытых нет. Уведомляем пользователя и завершаем работу функции тем, что отдаём логическое *False*. Это позволит легко отслеживать статус наличия позиции в других функциях.

Если позиции есть, собираем данные на неё в глобальные переменные, уведомляем о полном состоянии позиции и завершаем функцию, вернув True, что также позволяет легко отслеживать статус позиции в других функциях.

5.9. Реализация функции calculate_position()

Реализуйте функцию calculate_position(), как описано ниже.

```
def calculate_position():
    # Функция рассчитывает позицию.
    # Берём текущий баланс в usdt и получаем 90 процентов
    # Затем переводим в размер позиции и открываем ордер по рынку
    # Возвращает сумму на которую входим в номинале монеты
    aval_balance = get_balance_info() # Получаем текущий баланс
    get_90_percents = (aval_balance * 90) / 100
    print("Расчитываю позицию ордера")
    print("90% от баланса в USDT", get_90_percents)

    cur_price = get_current_symbol_price() # Получаем текущую цену
    final_quote = get_90_percents / cur_price # Финальный размер
    final_quote = round(final_quote, 1) # Округляем до целого числа
    msg = "Размер позиции " + str(final_quote)
    print(msg)
    send_message(msg) # Отпрвляем в ТГ
    return str(final_quote) # Отдаём размер позиции

calculate_position() # Сразу запускаем на тест
```

Если всё в порядке, вы получите размер позиции в Телеграм.

Функция calculate_position() рассчитывает сумму будущей позиции, не принимает аргументов.

С помощью уже написанной нами функции get_balance_info() берём текущий доступный баланс в **usdt**. От полученной суммы получаем 90 процентов и показываем на экране доступную для трейда сумму.

Рассчитываем финальный объём позиции в LTC и отправляем уведомление в Телеграм. Возвращаем финальный объём в количестве монет.

5.10. Реализация функции create_or_cancel_position()

Опишите функцию create_or_cancel_position(), как описано ниже.

Листинг 5.3

```
def create_or_cancel_position(symbol="LTCUSDT", side="Buy",
order_type="market", quantity="0.7"):
    global start_balance # Импорт глобальной переменной
    if side == "Buy":
        msg = "Приступаю к покупке на " + str(quantity)
        print(msg)
        send_message(msg) # Уведомляем в ТГ
    else:
        msg = "Приступаю к продаже на " + str(quantity)
        print(msg)
        send_message(msg) # Уведомляем в ТГ
    try: # Размещаем ордер в безопасном режиме
        response = session.place_order(
            category="linear",
            symbol=symbol,
            side=side,
            orderType=order_type,
            qty=quantity,
            timeInForce="GTC",)
        sleep(1) # Засыпаем на секунду
    except Exception as error:
        msg = "Ошибка при создании ордера " + str(error.message)
        print(msg)
        send_message(msg) # Уведомляем в ТГ
        return # Завершаем функцию ничего не отдав

    if response["retMsg"] == "OK" and side=="Buy":
        # Если биржа разместила ордер и направление Покупка
        print("Подтверждён ордер на покупку",
            "устанавливаю цену входа и объём")
        msg = "Подтверждён ордер на покупку"
        send_message(msg) # Уведомляем в ТГ
        get_current_position() # Получаем текущую позицию
    elif response["retMsg"] == "OK" and side=="Sell":
        # Если биржа разместила ордер и направление Продажа
        print("Подтверждена продажа и фиксация прибыли/убытка")
        print("Начальный баланс", start_balance)
        end_balance = get_balance_info()
        freeze = start_balance = end_balance
        print("Разница в валюте:", freeze)
        msg1 = "Подтверждена продажа. Текущий баланс: "
        msg2 = str(end_balance)
        send_message(msg1+msg2) # Уведомляем в ТГ
        get_current_position() # Получаем текущую позицию
```

```
    else: # Если биржа вернула ошибку в ответе
        msg = "Ордер не исполнен, разруливай вручную"
        print(msg)
        send_message(msg) # Уведомляем в ТГ
    return response # Возвращаем ответ от Байбит

create_or_cancel_position() # Запускаем тест
```

- Функция `create_or_cancel_position()` открывает сделку либо закрывает уже существующую.
- Функция принимает четыре переменные аргумента (`symbol="LTCUSDT", side="Buy", order_type="market", quantity="0.1"`) со значениями по умолчанию.
- Первым делом импортирует глобальную перемеренную *start_balance*, чтобы рассчитать текущую прибыль (убыток). Проверяем направление для покупки или продажи. Если ордер на покупку, то уведомляем пользователя Телеграм. Аналогично поступаем в случае ордера на продажу.
- Затем в блоке TRY: безопасно обращаемся к объекту *session* и его методу place_order().
- Если всё хорошо, размещаем ордер и сохраняем данные на него в переменной *response*.
- Если была ошибка при обращении к бирже и сработал блок EXCEPT:, уведомляем пользователя и завершаем функцию, ничего не возвращая.
- Далее подробно разбираем ответ от биржи, который хранится в переменной *response*.
- Биржа возвращает ответ в формате json, а это по сути обычный словарь.
- Если response["retMsg"] равен "OK" и направление сделки *side* равно "Buy" (сервер вернул успешный ответ и направление "на покупку"), уведомляем пользователя в Телеграм, что ордер успешно размещён.
- Если response["retMsg"] равен "OK" и направление сделки *side* равно "Sell" (сервер вернул успешный ответ и направление "на продажу"), уведомляем, что совершена успешная продажа, и фиксируем прибыль или убыток.

Во всех остальных случаях в блоке ELSE: уведомляем, что ордер не смог открыться и ситуацию необходимо разрешать вручную.

5.11. Реализация функции stop_loss_monitoring()

Реализуйте функцию stop_loss_monitoring(), как описано ниже:

```
def stop_loss_monitoring():
    global current_position_confirm_price
    print("Запускаю функцию мониторинга стопа")
    cur_price = get_current_symbol_price()
    sleep(1)
    print("current_position_confirm_price", current_position_confirm_price)
    cur_pos =  current_position_confirm_price # Для сокращения
    down_percent = (cur_price-float(cur_pos)) / cur_price * 100

    if down_percent <= -2.5:
        # Максимальная просадка при профитных сделках
        msg = "Сработал стоп: " + str(down_percent)
        send_message(msg) # Уведомляем в ТГ
        create_or_cancel_position(side="Sell",
            quantity=current_position_size) # Закрываем позицию
        return "STOP_LOSS" # Дальше не идём, ждём следующей свечи
    else:
        msg = "Нет условий для срабатывания стопа"
        send_message(msg) # Уведомляем в ТГ
        return # Завершаем функцию ничего не отдав
```

Предпоследняя функция check_stop_loss() для мониторинга за стопом работает следующим образом. Если функция запущена, она берет текущую цену актива и цену открытия ордера. Если цена упала больше чем на 2,5 процента, вызываем закрытие позиции.

5.12. Реализация функции run(). Запускаем HTTP-робота

Самая важная из всех функций — это run(). Заранее опишем её работу на псевдокоде и лишь затем реализуем.

Я постарался сохранить вложенность конструкций.

Функция run() для запуска бота. Ничего не принимает и не отдаёт.
Сначала функция проверяет, есть ли открытый ордер.

Если ордер есть:
Получаем последние дневные свечи.
Проверяем условие для выхода из сделки:
Если есть условие для выхода из сделки:
Закрываем позицию.
Пытаемся высчитать прибыль или убыток.
Пишем в ТГ, что произошёл трейд и был получен доход.
Если условие для закрытия позиции не сработало:
Пишем в ТГ, что позиция по-прежнему открыта.
Запускаем проверку стопа.
Если стоп не сработал, завершаем функцию.

Если ордера нет:
Проверяем, есть ли условие для входа:
Если есть условие для входа:
Рассчитываем сумму сделки и заходим в позицию.
Отправляем сумму и цену, по которой ордер исполнился, в ТГ.
Если нет условия для входа:
Пишем в ТГ, что условия для входа сегодня нет.
Завершаем работу скрипта.

Листинг 5.4

```
def run():
    check_current_position = get_current_position() # Текущая поза
    latest_candels = get_candles() # Последние свечи
    current_close= latest_candels[0][4] # Текущая цена закрытия
    previous_open = latest_candels[-1][1] # Прошлая открытие

    if check_current_position:
        # Если есть позиция, проверяем условие выхода
        print("Позиция есть, делаю проверку на выход")
```

```
        is_signal_cell = current_close > previous_open
        if is_signal_cell:
            # Тут фиксируем цену, по которой открылась сделка
            msg = "Продаём по тейк профиту"
            print(msg)
            send_message(msg)
            create_or_cancel_position(side="Sell",
                quantity=current_position_size) # Продажа
            return # Просто завершаем функцию до следующего запуска
        else: # Если позиция есть, но нет сигнала на выход
            msg = "Нет условия для выхода"
            print(msg)
            send_message(msg) # Уведомляем в ТГ
            stop_loss_monitoring() # Запускаем мониторинг стопа
            return # Останавливаем функцию run()
    else: # Если позиции нет, проверяем на условие входа
        is_signal_buy = current_close < previous_open
        if is_signal_buy: # Если есть условия для входа
            position_size = calculate_position() # Рассчитываем позицию
             # Создаём ордер
            create_or_cancel_position(quantity=position_size)
            return # Останаваливаем run()
        else: # Если нет условий для входа
            msg = "Нет условий для входа"
            print(msg)
            send_message(msg) # Уведомляем в ТГ
            return # Завершаем функцию run() ничего не отдав

run() # Запускаем на тест
```

Всё готово! Осталось обернуть код в виде сервиса и настроить запуск. Не забудьте убрать тестовый запуск предыдущих функций.

5.13. Создаём LINUX-сервис

Линукс-сервисы позволяют фоново запускать скрипты, а также перезапускать их в случае падения сервера.

1. Заходим на сервер Jino.ru.

2. В директории home создаём файл: nano ltcbot.py.

3. Копируем код с локального компьютера в файл nano ltcbot.py.

4. Переходим в папку для создания сервисов: cd /etc/systemd/system/.

5. Создаём файл: nano ltcbot.service.

6. В ltcbot.service добавляем следующее содержимое:

```
  [Unit]
Description=LTC Bot
After=network.target

[Service]
WorkingDirectory=/home

User=crypto-monitor

Restart=always
TimeoutStopSec=1

ExecStart=/bin/python3 /home/ghost/ltcbot.py

[Install]
WantedBy=multi-user.target
```

7. Затем включаем автозапуск вместе с системой:

```
systemctl enable crypto-monitor
```

8. Запускаем сервис:

```
systemctl start ltcbot
```

9. Рестартим его скрипт с помощью *cron -e*.

```
59 23 * * *  /usr/bin/systemctl restart ltcbot >>> log.txt
```

Поздравляю, вы создали и запустили своего торгового HTTP-Робота! Наслаждайтесь автоматизацией финансов на Python!

5.14. Перезапуск Ботов с помощью bash-скрипта

Если у вас возникли трудности с созданием линукс-сервиса, вы можете использовать bash-скрипт. Он гарантированно остановит предыдущее выполнение ваших скриптов и перезапустит их заново.

1. Перейдите в домашнюю директорию с помощью команды *cd /home/ghost*

2. С помощью команды *nano restart-ltc-bot.sh* создайте bash-скрипт со следующим содержимым:

```
#!/bin/bash
pkill -f "python3 /home/ghost/ltcbot.py"
python3 /home/ghost/ltcbot.py &
```

3. Нажмите сочетание клавиш Ctrl+O чтобы сохранить файл.

4. Наберите команду прав доступа: *chmod +x /home/ghost/restart-ltc-bot.sh*

5. Наберите *crontab -e*

6. Добавьте строку *59 23 * * * /home/ghost/restart-ltc-bot.sh >> ltclog.txt*

7. Нажмите сочетание клавиш Ctrl+O чтобы сохранить файл.

Для точности и надёжности вам нужно отследить время вашего сервера. Обратите внимание, что строка 59 23 перезапустит скрипт в 23 часа 59 минут.

Есть шанс, что время сервера будет по часовому поясу МСК.

В таком случае перезапуск скрипта должен выглядеть так:

```
59 2 * * * /home/ghost/restart-ltc-bot.sh >> ltclog.txt
```

На этом моя книга подходит к концу, надеюсь, что она будет вам полезна.

Послесловие

Если вдуматься, машины НЕ сделали нашу жизнь заметно счастливее.

10 из 10 сотрудников офиса занимаются тем, что обслуживают документы Эксель. 9.9 из 10 приложений которыми мы пользуемся НЕ в состоянии даже запуститься без нажатия кнопки человеком.

Даже считающимися передовыми технологии нейросетей, НЕ сдвинутся с места, пока их не пнёт человек. Мало того, нейросети обучают лгать человеку! Да, безусловно, генерировать контент стало проще. Другой вопрос – а действительно ли этот контент нужен людям?

Финансовая грамотность и автоматизация – это первое, где, на мой взгляд, стоит применять навык программиста, чтобы действительно подчинить своим интересам машину и освободить пространство для творчества.

Благодаря доступности средств автоматизации на финансовых рынках и анонимизации доходов с помощью криптовалют, мы получили больше свободы. Конечно, НЕ каждый сможет выйти на полноценный автоматизированный доход за счёт торговых роботов, но, надеюсь, эта книга Вам поможет создать дополнительный источник дохода.

Дерзайте и Удачи!

Список использованных источников информации:

- *https://www.iconfinder.com/icons/7829129/bitcoin_cryptocurrency_stocks_stocks_exchange_icon*
- *github.com*
- *backtrader.ru*
- *backtrader.com*
- *https://www.rbc.ru/quote/news/article/628ccd3a9a79474ed43db40f*
- *https://www.rbc.ru/quote/news/article/6307b2a49a7947a84188be2d*
- *https://www.rbc.ru/quote/news/article/6318aec49a7947448608ecc2*

Издательство «Наука и Техника» выпускает книги более 25 лет!

Уважаемые авторы!

Приглашаем к сотрудничеству по созданию книг
по **IT-технологиям, электронике, электротехнике, медицине, педагогике.**

Наши преимущества:

- являемся одним из ведущих технических издательств страны;
- выпускаем книги большими тиражами, что положительно влияет на гонорар авторов;
- регулярно переиздаем тиражи, автоматически выплачивая гонорар за *каждый* тираж;
- применяем индивидуальный подход в работе с каждым автором;
- работаем профессионально: от корректуры до авторских дизайн-проектов;
- проводим политику доступной цены;
- имеем собственные каналы сбыта: от федеральных сетей, крупнейших книжных магазинов РФ, ведущих маркетплейсов OZON, Wildberries, Яндекс-Маркет и др. до ведущих библиотек вузов, ссузов.

Ждем Ваши предложения:

- **тел. (812) 412-70-26**
- **эл. почта: nitmail@nit.com.ru**

Будем рады сотрудничеству!

Для заказа книг:

- **интернет-магазин: nit.com.ru**
  - более 3000 пунктов выдачи на территории РФ, доставка 3–5 дней
  - более 300 пунктов выдачи в Санкт-Петербурге и Москве, доставка 1–2 дня
  - тел. (812) 412-70-26
  - эл. почта nitmail@nit.com.ru
- **магазин издательства:** г. Санкт-Петербург, пр. Обуховской обороны, д. 107
  - метро Елизаровская, 200 м за ДК им. Крупской
  - ежедневно с 10.00 до 18.00
  - справки и заказ: тел. (812) 412-70-26
- **книжные сети и магазины**
  - «Читай-город» - сеть магазинов тел. +7 (495) 424-84-44
  - «Буквоед» - сеть магазинов тел. +7 (812) 601-0-601
  - Московский дом книги – сеть магазинов тел. +7 (495) 789-35-91
  - ТД «БиблиоГлобус» тел. +7 (495) 781-19-12
  - «Амиталь» — сеть магазинов тел. +7 (473) 223-00-02
  - Дом книги, г. Екатеринбург тел. +7 (343) 289-40-45
  - Дом книги, г. Нижний Новгород тел. +7 (831) 246-22-92
  - Приморский торговый Дом книги тел. +7 (423) 263-10-54
- **маркетплейсы OZON, Wildberries, Яндекс-Маркет, Myshop и др.**

Бакалов Д. А.

Программируем финансы на Python

Криптовалюта, биржа, торговые и Телеграм боты

Группа подготовки издания:

Зав. редакцией компьютерной литературы: *Е.В. Финков*

Редактор: *О.С. Петрунич, Н.В. Жерлов*

Корректор: *А.В. Громова*

Изображение на обложке использовано с ресурсов freepik.com и vecteezy.com

ООО "Издательство Наука и Техника"

ОГРН 1217800116247, ИНН 7811763020, КПП 781101001

192029, г. Санкт-Петербург, пр. Обуховской обороны, д. 107, лит. Б, пом. 1-Н

Подписано в печать 09.06.2025. Формат 70x100 1/16.

Бумага офсетная. Печать офсетная. Объем 20 п.л.

Тираж 1500. Заказ 13937.

Отпечатано с готового оригинал-макета

ООО «Принт-М», 142300, М.О., г.Чехов, ул. Полиграфистов, д.1

Программируем финансы на Python | криптовалюта, биржа, торговые и Телеграм боты

Эта книга поможет Вам стать реально богаче используя автоматизацию финансов с помощью **Python:** Вы не только создадите прототип биржи и собственные скринеры в **Телеграм**, но и научитесь грабить популярные биржи с помощью ими же предоставленных средств.

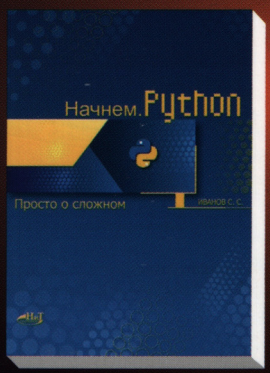
Вам НЕ потребуются знания в области программирования, т.к. всё необходимое для базового навыка программиста Вы получите в первых двух главах, в которых мы разработаем биржу бинарных опционов, реализуем **Биткойн** и **Альткойн**, а также освоим объектно-ориентированное программирование, без которого невозможно эффективно пользоваться библиотеками для бэкстестинга и торговли.

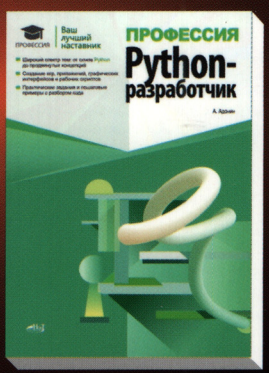
Далее мы разработаем вспомогательные инструменты для работы трейдера (к примеру, перенесем биржу опционов из первой главы в **Телеграм**), научим Вас создавать одновременно простых и полезных чат-ассистентов, которые будут мониторить рынки и сигнализировать о его состоянии. Отдельное внимание уделим тестированию торговых стратегий и, получив в первых главах навыки программирования, разработаем автоматизированную торговую систему со средним потенциалом возврата инвестиций до 70% годовых на падающем рынке.

Автор приложил много усилий для простоты и чистоты кода, чтобы код примеров было легко читать даже новичкам (например, Телеграм скринеры удалось реализовать без использования специализированных библиотек, используя лишь простые **HTTP-запросы** и **JSON-файлы**), ну а опытные разработчики смогут оценить, насколько просто такой код поддерживать и получат рабочие прототипы за считанные часы (а не недели (!)).

Приведённые пошаговые примеры имеют прикладной характер и используются в ботах, которые работают на автора ежедневно, при этом примеры могут быть легко адаптированы под Ваши собственные цели и стратегии.

издательство НАУКА и ТЕХНИКА Санкт-Петербург





Наши книги — ваши инвестиции в будущее

ISBN 978-5-907592-79-7

«Издательство Наука и Техника» г. Санкт-Петербург
Для заказа книг: т. (812) 412-70-26
E-mail: nitmail@nit.com.ru
Сайт: nit.com.ru

nit.com.ru