**Теоретические вопросы к экзамену**

1. **Понятие виртуальной сети. Виды виртуальных сетевых компонентов.**

Виртуальная сеть - совокупность технологий, позволяющих обеспечить одно или несколько сетевых соединений поверх другой сети. Виртуальные сети представляют собой самые обычные виртуальные коммутаторы. К каждому виртуальному коммутатору могут присоединяться как сетевые интерфейсы виртуальных машин, так и физические сетевые интерфейсы сервера. Виртуальные сети (то есть виртуальные коммутаторы) бывают трех типов: External, Internal и Private.

**External** – виртуальная сеть, имеющая выход «во внешний мир».

**Internal** – внутренняя виртуальная сеть, к которой могут подключаться только виртуальные интерфейсы – виртуальных машин и хостовой ОС.

**Private** – то же самое, что и Internal, за исключением того, что к такой сети могут подключаться только виртуальные машины. Сеть типа Private не имеет доступа ни ко «внешнему миру», ни к хостовой ОС.

***Виртуальные сетевые компоненты:***

***Виртуальный сетевой адаптер*** (**Virtual network adapter) -** программный эмулятор сетевой карты, устанавливаемый на гостевую ОС.

***Виртуальный адаптер хоста*** (**Host virtual adapter)-** виртуальный Ethernet-адаптер, устанавливаемый на хостовую ОС. Это виртуальное устройство служит для взаимодействия ВМ с хост-компьютером и включается в состав ВМ, когда для нее задается тип сетевого подключения **Host Only**

***Мост*** (**Bridge)-** программно реализованный сетевой мост, он соединяет виртуальный сетевой адаптер с физическим Ethernet-адаптером хост-компьютера.

***Виртуальные коммутаторы*** (**Virtual switches)-** эти устройства, подобно физическим коммутаторам, обеспечивают соединение между собой различных узлов сети.

***Устройство преобразования сетевых адресов*** (**NAT Device)-** позволяет подключать ВМ к внешней сети, когда ВМ невозможно выделить собственный IP-адрес и приходится использовать IP-адрес, назначенный хост-компьютеру.

***DHCP-сервер*** - программный компонент, обеспечивающий назначение сетевых IP-адресов виртуальным машинам в сети, в которой не используются подключения через мост

1. **Трансляция сетевых адресов. Виды NAT.**

**Трансляция сетевых адресов** (NAT, Network Address Translation) позволяет нескольким машинам локальной сети иметь доступ к Интернет через одно подключение и один реальный внешний IP-адрес.

**NAT** - механизм в сетях TCP/IP, позволяющий преобразовывать IP адреса транзитных пакетов.

Функции NAT:

· Экономия IP адресов

· Ограничение доступа извне

· Сокрытие внутренних сервисов

Для того, чтобы компьютера локальной сети могли устанавливать соединения с серверами сети Интернет, нужно, чтобы:

• IP-пакеты, адресованные серверу в Интернет, смогли его достигнуть;

• ответные IP-пакеты, идущие от сервера Интернет на машину в локальной сети, также смогли ее достигнуть.

### *Виды NAT:*

**Статический NAT** — отображение незарегистрированного IP-адреса на зарегистрированный IP-адрес на основании один к одному.

**Динамический NAT** — отображает незарегистрированный IP-адрес на зарегистрированный адрес из группы зарегистрированных IP-адресов. Динамический NAT также устанавливает непосредственное отображение между незарегистрированными и зарегистрированными адресами.

**Перегруженный NAT** — форма динамического NAT, который отображает несколько незарегистрированных адресов в единственный зарегистрированный IP-адрес, используя различные порты. Известен также как [PAT](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_%D0%BF%D0%BE%D1%80%D1%82-%D0%B0%D0%B4%D1%80%D0%B5%D1%81) (Port Address Translation). При перегрузке каждый компьютер в частной сети транслируется в тот же самый адрес, но с различным номером порта.

1. **Использование Linux для разработки. Стандартные программные средства.**

## *Почему стоит использовать linux для программирования?*

### Простота настройки среды

### Полноценный терминал

### Среда близкая к производственной

### Контроль над именами файлов

*Стандартные программные средства:*

* компиляторы C и C++ - gcc и g++
* какой-либо пакетный менеджер (apt, packman, yum)
* интерпретатор perl

*То, что скорее всего будет в дистрибутиве:*

* nano - простой текстовый редактор
* make и cmake - сборщик исходников для C-подобных языков программирования
* vim - более сложный текстовый редактор, есть возможность добавлять плагины
* интерпретатор python
* snap или flatpack (дистрибуция приложений в контейнерах)

1. **Интерпретатор Python. Использование, версии. Понятие виртуального окружения, настройка, использование.**

Интерпретатор Python после установки располагается, обычно, по пути /usr/local/bin/python. Кто установил интерпретатор на машину Unix, потенциально имеют поддержку библиотеки GNU readline, обеспечивающей усовершенствованное интерактивное редактирование и сохранение истории. Интерпретатор ведёт себя сходно шеллу Unix: если он вызван, когда стандартный ввод привязан к устройству tty — он считывает и выполняет команды в режиме диалога; будучи вызванным с именем файла в качестве аргумента или с файлом, назначенным на стандартный ввод — он читает и выполняет сценарий из этого файла.

У каждого проекта должен быть свой интерпретатор Python, со своей собственной изолированной директорией site-packages. Это и есть основная идея, стоящая за виртуальными окружениями. *Виртуальное окружение* — это самостоятельная копия интерпретатора со своими пакетами.

***Настройка:***

Virtualenv можно установить с использованием менеджера pip, либо скачать исходные коды проекта и установить приложение вручную.

pip install virtualenv

***Виртуальное окружение создается следующей командой:***

virtualenv \*имя окружения\*

После выполнения данной команды, в текущем каталоге будет создан новый каталог с именем окружения.

***Для активации виртуального окружения:***

source \*имя\*/bin/activate

Команда source выполняет bash-скрипт без запуска второго bash-процесса.

После завершения работы деактивировать виртуальную среду можно с помощью команды deactivate.

Введите ее и приставка \*имя\* пропадет. Вы вернетесь к использованию глобально версии python.

***Удаление виртуальной среды:***

rm -rf \*имя\*

1. **Структура проекта на Python. Организация модулей. Файл зависимостей.**

Под модулем в *Python* понимается файл с расширением *.py*. Модули предназначены для того, чтобы в них хранить часто используемые функции, классы, константы и т.п. Можно условно разделить модули и программы: программы предназначены для непосредственного запуска, а модули для импортирования их в другие программы. Стоит заметить, что модули могут быть написаны не только на языке *Python*, но и на других языках.

При импорте модуля Python выполняет весь код в нём, который не находится в функциях. Условие if \_\_name\_\_ == “\_\_main\_\_” необходимо для того, чтоб код в нем не выполнялся при импорте файла в качестве модуля, НО при непосредственном вызове файла через интерпретатор Python отрабатывался.

***Порядок поиска файлов, содержащих модули:***

**1**.каталог, где находится сама программа, даже если она вызывается из другого каталога;

**2**.пути к каталогам из переменной окружения PYTHONPATH, если она определена;

**3**.пути к каталогам стандартной библиотеки языка Python - они определяются на этапе установки Python.

В каждом языке программирования есть свои файлы помогающие правильно установить окружение приложений, в python таким файлом является requirements.txt - файл зависимостей приложения, для автоматической установки пакетов python с помощью утилиты pip

Данный формат является обычным текстовым файлом, где указано название пакета python, его версия и условие, больше, меньше, равно.

При дистрибуции проекта настоятельно рекомендуется использовать файл зависимостей **requirements.txt**, который содержит в себе все библиотеки, которые используются текущим проектом

Чаще всего файл зависимостей используют с обособленным виртуальном окружением проекта, реже с глобальным окружением.

Для того, чтоб *экспортировать названия всех библиотек текущего окружения* в **requirements.txt**, используют команду:

pip3 freeze > **requirements.txt**

Для того, *чтоб поставить все пакеты из файла* **requirements.txt**, используют команду:

pip3 install -r **requirements.txt**

1. **Системы контроля версий. Примеры, назначение, общие понятия.**

**Система контроля версиями** — программное обеспечение для облегчения работы с изменяющейся информацией. Система контроля версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

*Все они делятся на два типа:*

* распределенные (Git, Mercurial) – изменения хранятся в локальных хранилищах компьютеров и оттуда синхронизируются с другими компьютерами;
* централизованные (CVS, SVN) – все данные хранятся на центральном сервере, и оттуда каждый локальный компьютер получает обновленные данные.

*Обычно вся работа над проектом со стороны разработчика сводится к трем простым операциям:*

· удаление;

· добавление;

· изменение.

*С приходом VCS работать стало гораздо проще:*

· можно откатывать изменения, если смысла в их внедрении нет;

· можно быстро и безболезненно восстанавливать поврежденные файлы;

· можно определить, кто из команды писал определенный блок кода;

· можно следить за процессом, даже если в один и тот же момент над одним модулем работает несколько разработчиков или даже команд по всему миру.

1. **Общий алгоритм работы с СКВ Git. Инициализация репозитория, добавление файлов, коммиты.**

**GIT** – удобная и быстрая децентрализованная система совместной разработки (репозитория). Рассмотрим несколько приёмов работы с git.

***Предварительная настройка Git:***

Можно задать глобальную идентификацию:

git config --global user.name "You Name"

git config --global user.email you\_name@you\_domain.com

Это задаст ваше имя пользователя и почтовый адрес которыми будут подписываться ваши коммиты.

***Распределённая разработка проекта:***

Условия:

Есть общий сервер git-server (которым, например, может выступать GitHub). Каждый разработчик работает с репозиторием проекта локальный и регулярно обмениваться информацией с общим сервером.

***Инициализация:***

Перейдите в папку, где будет размещаться ваш проект и выполните клонирование проекта с сервера:

git clone ssh://you\_name@git-server.com/~/you\_project

Протокол доступа к репозиторию зависит от настроек сервера. После выполнения создастся папка проекта you\_project и в последующем проводить инициализацию больше не нужно.

***Разработка:***

*Перед началом разработки необходимо создать локальную ветку и связать её с удалённой*:

git checkout -b my\_branch origin/my\_branch

*При создании новых файлов необходимо их пометить на добавление в репозиторий:*

git add myfile.txt myfile2.txt myfile3

*После внесения изменений их необходимо зафиксировать в коммит:*

Указываем какие файлы нужно добавить в коммит:

git commit myfile.txt myfile2.txt myfile3

после чего потребуется ввести описание изменений или задать описание сразу (использоваться опцию -m):

git commit myfile.txt myfile2.txt myfile3 -m "описание изменений"

Чтобы закоммитить все произведённые изменения репозитория можно выполнить (использоваться опцию -a):

git commit -a -m "описание изменений"

*Оправка изменений на сервер*:

Для отправки изменений на общий сервер выполняем:

git push

*Получение последних изменений с сервера:*

Когда нужно забрать изменения с общего сервера:

git pull

Для просмотра состояния репозитория:

git status

1. **Работа с ветвлением в Git. Назначение веток. Создание, переключение, объединение веток.** **Разрешение конфликтов слияния.**

**Ветки** — это специальные указатели на коммиты, которые перемещаются с каждым новым коммитом. **Ветвление** — это способ изолировать одни изменения от других. Ветвление происходит, когда мы произвели два разных изменения, основываясь на одном и том же состоянии.

**HEAD**– так называемый курсор Git. На какой коммит указывает **HEAD**– из того коммита и загружаются файлы в рабочую директорию.

**git branch <имя ветки> -** Создает новую ветку.

**git checkout -b <имя ветки> -** Создает новую ветку и сразу переключается на нее.

**git branch <ключи> -** По умолчанию выводит список локальных веток.

**git checkout <ключи> <имя ветки> -** Переключает пользователя на другую ветку

**git status –** Просматривает статус файлов ветки

**git branch –delete/-d <имя ветки> -** Удаляет указанную ветку.

**git merge <branch>** - Слияние данной ветки в текущую

**Разрешение конфликтов слияния.**

При работе с ветками бывает такое, что возникают конфликты слияния. Это происходит потому, что в разных ветках были разные коммиты, связанные с одним и тем же файлом. В этом случае git поругается, что ничего сделать не может и далее вам придется вручную править файл для разрешения конфликта слияния, после чего сделать коммит.

1. **Работа с удаленными репозиториями. Клонирование и форк репозиториев. Отправка и получение изменений в удаленный репозиторий.**

Удалённые репозитории представляют собой версии вашего проекта, сохранённые в интернете или ещё где-то в сети. У вас может быть несколько удалённых репозиториев, каждый из которых может быть доступен для чтения или для чтения-записи. Взаимодействие с другими пользователями предполагает управление удалёнными репозиториями, а также отправку и получение данных из них. Управление репозиториями включает в себя как умение добавлять новые, так и умение удалять устаревшие репозитории, а также умение управлять различными удалёнными ветками, объявлять их отслеживаемыми или нет и так далее.

Для того, чтобы просмотреть список настроенных удалённых репозиториев, вы можете запустить команду **git remote**.

Для того, чтобы добавить удалённый репозиторий и присвоить ему имя (shortname), просто выполните команду **git remote add <shortname> <url>**

Для получения данных из удалённых проектов, следует выполнить: **git fetch [remote-name]**. Команда git fetch забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент

**git clone <url>** – Команда для клонирования репозитория

**git pull <remote>** – Команда для получения информации об изменениях

**git push [remote] [branch]** - Команда для выгрузки содержимого локального репозитория в удаленный репозиторий.

1. **Современные методологии работы с Git в командном проекте. GitFlow.**

Наиболее популярной СКВ для организации совместной разработки является Git. Ветвление и слияние обычно считаются опасными из-за конфликтов слияния и потому проводятся как можно реже.

Но с Git эти действия становятся исключительно простыми и дешёвыми, и потому на деле они становятся центральными элементами обычного *ежедневного* рабочего процесса.

Благодаря своей простоте и предсказуемости, ветвление и слияние больше не являются действиями, которых стоит опасаться. Теперь инструменты управления версиями способны помочь в ветвлении и слиянии больше, чем какие-либо другие.

**Gitflow**— это модель, которая, по сути, является просто набором процедур, которые исполняет каждый член команды, чтобы все вместе могли достичь высокой управляемости процесса разработки.

Она предполагает выстраивание строгой модели ветвления вокруг релиза проекта, которая дает надежную схему управления крупными проектами.

Кроме feature-веток в ней используются отдельные ветки для подготовки, поддержки и записи релиза.

Gitflow использует собственный набор инструментов git-flow, который легко интегрируется с Git, добавляя новые команды Git.

**Начало работы**

Набор инструментов git-flow нужно установить отдельно. После установки git-flow необходимо выполнить команду git flow init, она является расширением стандартной команды git init и ничего не меняет в вашем репозитории, кроме того, что создает ветки.

**Ключевые идеи, которые нужно запомнить о Gitflow:**

· Данная модель отлично подходит для организации рабочего процесса на основе релизов.

· Gitflow предлагает создание отдельной ветки для исправлений ошибок в продуктовой среде.

1. **Понятие сетевого сокета. Применение, виды, схема взаимодействия.**

**Сокеты** — название программного интерфейса для обеспечения обмена данными между процессами. Процессы при таком обмене могут исполняться как на одной ЭВМ, так и на различных, связанных между собой сетью. **Сокет** — абстрактный объект, представляющий конечную точку соединения.

**Применение сокетов:**

Каждый процесс может создать серверный сокет и привязать его к какому-нибудь порту операционной системы. Серверный процесс обычно находится в цикле ожидания, то есть просыпается при появлении нового соединения. Каждый сокет имеет свой адрес. ОС семейства UNIX могут поддерживать много типов адресов, но обязательными являются INET-адрес и UNIX-адрес. Обычно клиент явно подсоединяется к слушателю, после чего любое чтение или запись через его файловый дескриптор будут передавать данные между ним и сервером.

**Виды:**

*Синхронные* (блокируемые) и *асинхронные* (неблокируемые). Синхронные сокеты задерживают управление на время выполнения операции, а асинхронные возвращают его немедленно, продолжая выполнение в фоновом режиме, и, закончив работу, уведомляют об этом вызывающий код.

**Схемы взаимодействия:**

*Общие:*

Socket – Создать новый сокет и вернуть файловый дескриптор

*Пример на Python:*

import socket

# Создание объекта сокета.

sock\_obj = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM, 0)

# AF\_INET, SOCK\_STREAM и 0 используются по умолчанию при создании сокета.

# Поэтому можно просто писать:

sock\_obj = socket.socket()

Send – Отправить данные по сети

send, sendto - отправка данных.

*Пример на Python:*

IP = '192.168.1.100'

PORT = 8080

sock\_obj.send('Hello World!')

sock\_obj.sendto('Hello World!', (IP, PORT))

Receive – Получить данные из сети

recv, recvfrom - чтение данных из сокета.

*Пример на Python:*

BUFFER\_SIZE = 1024

data = conn.recv(BUFFER\_SIZE)

data, sender\_addr = conn.recvfrom(BUFFER\_SIZE)

SOCK\_STREAM vs SOCK\_DGRAM¶

Close – Закрыть соединение

*Серверные:*

Bind – Связать сокет с IP-адресом и портом

*Пример на Python:*

server\_address = ('localhost', 8080)

sock\_obj.bind(server\_address) # Привязка

Listen – Объявить о желании принимать соединения. Слушает порт и ждет, когда будет установлено соединение

*Пример на Python:*

sock\_obj.listen(5) # Ждем соединение клиента.

Accept – Принять запрос на установку соединения

*Пример на Python:*

conn, addr = sock\_obj.accept() # Установление соединения с кл

*Клиентские:*

Connect – Установить соединение с сервером

*Пример на Python:*

server\_address = ('192.168.1.100', 8080)

sock\_obj.connect(server\_address)

1. **Блокирующие операции при обмене через сокеты. Возможные ошибки. Таймауты.**

В Python сокет может быть помещён как в блокируемый, так и в не блокируемый режим. При не блокируемом режиме если вы ваш некий вызов API, скажем, send() или recv(), сталкивается с любой проблемой, будет распространена какая- то ошибка. Однако, если это происходит при режиме с блокировкой, это не остановит данную операцию. Можем вызвать setblocking(1) для установки блокировки, или setblocking(0) для её снятия. В конце концов, мы прикрепляем данный сокет к некоторому определённому порту и прослушиваем входящие соединения. Мы включаем блокирование некоторого сокета устанавливая значение 1 в своём методе setblocking(). Аналогично мы можем сбросить значение в 0 в данном методе, чтобы сделать сокет не блокированным.

Библиотека Python socket располагает неким элегантным методом обработки ошибок посредством исключения socket.error. Давайте предположим, что есть ряд блоков кода try-except и там умещено по одной потенциальной ошибке в каждом блоке.

В таком примере было применено четыре блока try-except. Все блоки применяли исключение socket.error за исключением, наапример, второго, который применял socket.gaierror. Оно используется для ошибок, относящихся к адресации. Имеются два других типа исключений - socket.herror применяется для наследуемого API C, а если вы применяете в некотором сокете метод settimeout(), будет распространена ошибка socket.timeout по достижению некоторого таймаута в данном сокете.

*Установка и получение значения таймаута сокета по умолчанию:*

Сделать некий экземпляр какого-то объекта socket и вызвать метод gettimeout() для получения установленного по умолчанию значения таймаута и метод settimeout() для установки определённого значения таймаута. Это очень полезно при разработке индивидуальных серверных приложений.

Вначале мы создадим некий объект socket внутри своей функции test\_socket\_timeout(). Затем мы можем применять методы экземпляра getter/setter для обработки значений таймаута.

#!/usr/bin/env python

import socket

def test\_socket\_timeout():

s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

print ("Default socket timeout: %s" %s.gettimeout())

s.settimeout(100)

print ("Current socket timeout: %s" %s.gettimeout())

if \_\_name\_\_ == '\_\_main\_\_':

test\_socket\_timeout()

После выполнения предыдущего сценария вы поймёте, как это изменяет значение таймаута сокета по умолчанию:

$ python 1\_6\_socket\_timeout.py

Default socket timeout: None

Current socket timeout: 100.0

В данном фрагменте кода мы вначале создаём некий объект socket, передавая имеющееся семейство сокета и тип сокета в качестве первого и второго параметров своего конструктора сокета. Затем вы можете получить значение таймаута сокета вызвав gettimeout() и изменить это значение путём вызова метода settimeout(). Передаваемое в метод settimeout() значение может быть в секундах или None. Данный метод применяется для работы с операциями, блокирующими сокет. Установка некоторого таймаута в значение None отключает таймауты в операциях с сокетом.

1. **Транспортные протоколы TCP и UDP. Принципы работы, сравнение.**

**Транспортный уровень (Transport Layer)** – обеспечивает приложениям или верхним уровням стека – прикладному, представления и сеансовому – передачу данных с той степенью надежности, которая им требуется.

**Протокол UDP (User Datagram Protocol)** является простейшим дейтаграммным протоколом, который используется в том случае, когда задача надежного обмена данными либо вообще не ставится, либо решается средствами более высокого уровня – прикладным уровнем или пользовательскими приложениями. Единица данных протокола UDP называется UDP-пакетом или пользовательской дейтаграммой. Каждая дейтаграмма переносит отдельное пользовательское сообщение. Это приводит к естественному ограничению: длина дейтаграммы UDP не может превышать длины поля данных протокола IP. Поэтому если UDP-буфер переполняется, то данные приложения отбрасываются. UDP не занимается контролем потока, контролем ошибок, повторной передачей после приема испорченного сегмента. Его функции сводятся к мультиплексированию и демультиплексированию данных между сетевым и прикладным уровнями. Для процессов, которым хочется управлять потоком, контролировать ошибки и временные интервалы, протокол UDP – это как раз то, что нужно.

**Протокол TCP (Transmission Control Protocol)** обеспечивает надежную транспортировку данных между прикладными процессами путем установления логического соединения. Информация, поступающая к протоколу TCP от протоколов более высокого уровня, рассматривается протоколом TCP как неструктурированный поток байтов. Поступающие данные буферизируются средствами TCP. Для передачи на сетевой уровень из буфера «вырезается» некоторая непрерывная часть данных, которая называется *сегментом*. Сегмент состоит из фиксированного 20-байтного заголовка, за которой могут следовать байты данных. Размер сегментов определяется программным обеспечением TCP. Оно может объединять в один сегмент данные, полученные в результате нескольких операций записи, или, наоборот, распределять результата одной записи между несколькими сегментами.

***Сравнение****:*

*Надежность:* в этом случае предпочтительнее будет протокол TCP, за счет подтверждения получения данных, повторной отправки в случае необходимости, а также использованию такого инструмента как тайм-аут. Протокол UDP такого инструментария не имеет, а потому при получении отправленные данные могут приходить не полностью;

*Упорядоченность:* снова будет предпочтительнее TCP, поскольку этот протокол гарантирует передачу пакетов данных именно в том порядке, в котором они были отправлены. В случае с UDP такой порядок не соблюдается;

*Скорость:* здесь уже лидировать будет UDP, так как более тяжеловесному TCP-протоколу будет требоваться больше времени для установки соединения, подтверждения получения, повторной отправки данных и т.д.;

*Метод передачи данных:* в случае с TCP данные передаются потоково, границы фрагментов данных не имеют обозначения. В случае с UDP данные передаются в виде датаграмм – проверка пакетов на целостность осуществляется принимающей стороной только в случае получения сообщения. Также пакеты данных имеют определенные обозначения границ.

1. **Клиент-серверное взаимодействие.**

*Клиент-серверное взаимодействие* — это обмен данными между клиентом и сервером. Клиенты и серверы обмениваются сообщениями в шаблоне запрос-ответ. Клиент отправляет запрос, а сервер возвращает ответ. Этот обмен сообщениями является примером межпроцессного взаимодействия. Для взаимодействия компьютеры должны иметь общий язык, и они должны следовать правилам, чтобы и клиент, и сервер знали, чего ожидать. Язык и правила общения определены в протоколе связи.

Сервер может получать запросы от множества различных клиентов за короткий период времени. Компьютер может выполнять только ограниченное количество задач в любой момент и полагается на систему планирования для определения приоритетов входящих запросов от клиентов для их удовлетворения.

Клиент — это программная оболочка, с которой взаимодействует пользователь. А сервер — это та часть программного обеспечения, которая выполняет все основные функции. Другими словами, пользователь видит программу, которая, допустим, работает с какими-то данными, которые хранятся в базе данных, тем самым он видит всего лишь интерфейс этой программы, а все самое основное выполняет сервер, и процесс, когда пользователь оперирует данными через интерфейс программы, при котором клиентская часть взаимодействует с серверной, и называется *Клиент-Сервер*.

*Преимущества:*

Низкие требования к компьютерам клиента, так как вся нагрузка должна возлагаться на сервер и серверную часть приложения, в некоторых случаях можно значительно сэкономить затраты на приобретение вычислительной техники в организациях;

Многопользовательский режим. Ресурсами сервера могут пользоваться неограниченное число пользователей, при том, что данные располагаются в одном месте;

Целостность данных. Вывести из строя компьютер клиента гораздо проще, и чаще встречается, чем компьютер, который выполняет роль сервера.

Внесение изменений. Проще внести изменения один раз в серверной части, чем вносить их на каждом клиенте.

*Недостатки:*

Для быстродействия требуется приобрести достаточно мощный сервер, но как было уже сказано выше, это может и окупится, за счет компьютеров пользователей;

Выход из строя серверной части прекратит работу всех клиентов, в связи с этим возникает необходимость постоянного мониторинга серверной части.

1. **Реализация сокетов в языке Python. Модуль socket.**

**Сокет** — название программного интерфейса для обеспечения обмена данными между процессами. Существуют клиентские и серверные сокеты. Серверный сокет прослушивает определенный порт, а клиентский подключается к серверу. После того, как было установлено соединение начинается обмен данными. Сокет на языке Python позаимствован из языка программирования C и операционной системы Unix. Для работы с сокетами нужен клиент(ы) и сервер.

В Python для работы с сокетами используется модуль socket:

1. Подключение библиотеки для работы с сокетами. import socket

2. Создание сокета: sock = socket.socket()

3. Определяемся с хостом и портом сервера sock.bind(‘localhost’,6789)

4. Указываем количество клиентов, которые будет слушать наш сервер. Sock.listen(5)

5. Client, addr = sock.accept()

Sock.recv – получает данные

Sock.send – отправляет данные.

Мы можем построить наши сервера как по udp, так и по tcp протоколу. UDP отправляет сообщения, но их размер ограничен и не гарантируется, что они достигнут места назначения. UDP не гарантирует в каком порядке придут сообщения. TCP вместо сообщений отправляет потоки байтов. Нельзя сказать, сколько байтов отправит или получит система с каждым вызовом. TCP доставляет данные в том порядке, в котором они были отправлены.

1. **Понятие программного потока. Процессы и потоки.**

*Программный поток* – поток, в котором выполняются задачи программы. Это мощный инстру­мент, когда приложению необходимо выполнять сразу несколько действий. Все задачи выполняются последовательно. С появлением многоядерных процессоров стала общеупотребительной практикой - распространять нагрузку на все доступные ядра. Существует два основных подхода в распределении нагрузки: использование процессов и потоков. Использование нескольких процессов фактически означает использование нескольких программ, которые выполняются независимо друг от друга.

*Процесс* - отдельная подпрограмма(процесс) от текущей программы. Понятие *процесса* характеризует некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов и текущего момента его выполнения, находящуюся под управлением ОС.

*Поток* - подпрограмма, которая выполняется внутри текущей (в текущем процессе). *Потоки* – последовательности команд. Поток можно определить как часть процесса, включающую управляющую последовательность команд и использующую системные ресурсы этого процесса.

Thread — это отдельный поток выполнения. Это означает, что в вашей программе могут работать две и более подпрограммы одновременно. Запуск потоков (threading) похожа на эту идею, но ваши программы работает только на одном процессоре. Различные задачи, внутри потоков выполняются на одном ядре, а операционная система управляет, когда ваша программа работает с каким потоком. В Python за это отвечают модули subprocessing и multiprocessing.

Для увеличения скорости работы программы используются потоки и процессы. Если нужно, чтобы ваше приложение выполняло несколько задач в одно и то же время, то можете воспользоваться потоками (threads). Потоки позволяют приложениям выполнять в одно и то же время множество задач. Многопоточность (multi-threading) важна во множестве приложений, от примитивных серверов до современных сложных и ресурсоёмких игр. За многопоточность отвечает модуль Threading. Создание отдельного потока:

import Threading

def foo(n):

print(n)

threading.thread(target=foo, args=10).start

1. **Асинхронное программирование. Основные понятия. Параллелизм и конкуррентность.**

В синхронных операциях задачи выполняются друг за другом. В асинхронных задачи могут запускаться и завершаться независимо друг от друга. Одна асинхронная задача может запускаться и продолжать выполняться, пока выполнение переходит к новой задаче. Асинхронные задачи не блокируют операции и обычно выполняются в фоновом режиме.

Асинхронный код убирает блокирующие операции из основного потока. *Асинхронный код* – это набор абстрактных парадигм, которые вместе дают асинхронное (конкурентное) выполнение. Асинхронность достигается с помощью Event Loop – цикл событий. В Event Loop кладутся coroutines (сопрограммы или корутины), которые выполняются в нем. Coroutines способны приостанавливаться в тот момент, когда Event Loop прикажет это сделать.

Task (задача) – позволяет запускать coroutine на фоне.

Future (футура) – будущий результат выполнения coroutine.

Когда использовать асинхронный код:

- Микросервисы (I/O-bound, не CPU-bound)

- Долгоживущие соединения (websocket, раздача файлов)

- Экономия ресурсов серверов

*Параллелизм* – когда несколько задач выполняются одновременно(параллельно).

*Конкурентность* – когда несколько задач выполняются совместно, но не одновременно. Такую форму использует асинхронность. Задача может разбиваться на несколько подзадач.

Параллелизм подразумевает конкурентность. Но конкурентность не всегда подразумевает параллелизм.

1. **Блокирующие и неблокирующие операции.**

В синхронном коде каждая операция ожидает окончание предыдущей, поэтому программа может не отвечать пока выполняется какая-то большая операция.

*Блокирующие операции* – операции, которые мешают выполнению других операций.

Решение проблем блокировки:

- *Процесс* - отдельная подпрограмма от текущей программы. Требует большие затраты на создание, требует очень много памяти, сложно связать с основной программой, оверхед со стороны ОС (user-space to scheduling, context switching)

- *Поток* - подпрограмма, которая выполняется внутри текущей (в текущем процессе). Требует большие затраты на создание. При работе в высоконагруженных систем выполнения могут произойти, например race conditional, dedlock, starvation, да еще и GIL.

- Асинхронность – когда программа работает, используя Event Loop (цикл событий), где Event это подпрограммы (coroutine), а Loop место, где происходит их хранение и выполнение путем прохода по всем событиям.

Рассмотрим данные операции на примере сокетов. Сокеты могут работать в одном из двух режимов: блокирующем или неблокирующем. Сокет обычно указывается блокирующим или неблокирующим при помощи функций fcntl() или ioctl().

Блокирующий сокет не возвращает контроль, пока не отправит или пока не получит все данные, указанные для операции. На время выполнения операции с блокирующим сокетом программа блокируется.

Неблокирующие сокеты – инструмент для работы из одного потока с несколькими сокетами сразу или с несколькими операциями над одним сокетом сразу.

Стандартные операции над сокетом, переведенным в неблокирующий режим, никогда не приводят к блокировке. Вместо этого они завершаются со специальным кодом ошибки, который означает, что операция не может быть выполнена в данный момент. Таким образом, программа не лишается управления при временной невозможности выполнить операцию, а лишь информируется об этом, и может повторить ее успешно позже.

1. **Алгоритмы, ограниченные процессором и вводом-выводом. Основные характеристики, особенности выполнения и распараллеливания.**

Процессы можно классифицировать как те, которые ограничены скоростью ввода-вывода (I/O-bound), и те, которые ограничены скоростью процессора(processor-bound). К первому типу относятся процессы, которые большую часть своего времени выполнения тратят на отправку запросов на ввод-вывод информации и на ожидание ответов на эти запросы. Следовательно, такие процессы часто готовы к выполнению, но могут выполняться только в течение короткого периода времени, так как в конце концов они блокируются в ожидании выполнения ввода-вывода.

Процессы, ограниченные скоростью процессора, наоборот, большую часть времени исполняют программный код. Такие процессы обычно выполняются до того момента, пока они не будут вытеснены, так как эти процессы не блокируются в ожидании на запросы ввода-вывода. Поскольку такие процессы не влияют на скорость ввода-вывода, то для обеспечения нормальной скорости реакции системы не требуется, чтобы они выполнялись часто. Стратегия планирования процессов, ограниченных скоростью процессора, поэтому предполагает, что такие процессы должны выполняться реже, но более продолжительный период времени.

*Распараллеливание программ* — процесс адаптации алгоритмов, записанных в виде программ, для их эффективного исполнения на вычислительной системе параллельной архитектуры. Заключается либо в переписывании программ на специальный язык, описывающий параллелизм и понятный трансляторам целевой вычислительной системы, либо к вставке специальной разметки. Распараллеливание может быть ручным, автоматизированным и полуавтоматизированным. Для оценки эффективности его качества применяются следующие критерии:

*Ускорение* – время исполнения распараллеленной программы на p процессорах, T1— время исполнения исходной программы. В идеальном случае равна p.

*Загруженность*, показывающая долю использования процессоров. В идеальном случае равна 1, или 100 %.

1. **Особенности реализации многопоточности в Python. Модуль threading.**

В современной операционной системе, даже не выполняющей ничего особенного, могут одновременно работать несколько процессов. Функции для управления процессами можно найти в стандартном модуле os языка Python. Потоки управления образуются и работают в рамках одного процесса. В однопоточном приложении имеется только один поток управления. В любой момент времени интерпретатор Python знает, какую команду исполнить следующей. После исполнения команды становится известно, какой команде передать управление. Эта ниточка непрерывна в ходе выполнения программы и обрывается только по ее завершении. Теперь можно представить себе, что в некоторой точке программы ниточка раздваивается, и каждый поток идет своим путем. Каждый из образовавшихся потоков может в дальнейшем еще несколько раз раздваиваться. Одним из классических средств согласования потоков являются объекты, называемые *семафорами*. Семафоры не допускают выполнения некоторого участка кода несколькими потоками одновременно. Самый простой семафор - замок (lock) или mutex.

Поддержка многопоточности в языке Python доступна через использование ряда модулей. В стандартном модуле threading определены нужные для разработки многопоточной (multithreading) программы классы: несколько видов семафоров и другие механизмы взаимодействия между потоками, класс Timer для запуска функции по прошествии некоторого времени. Модуль Queue реализует очередь, которой могут пользоваться сразу несколько потоков. Для создания и управления потоками в стандартном модуле thread определен класс Thread.

*Пример модуля threading:*

import threading

def proc(n):

print “Процесс”, n

p1 = threading.Thread(target=proc, name=“t1”, args={“1”})

p2 = threading.Thread(target=proc, name=“t2”, args={“2”})

p1.start()

p2.start()

В модуле threading есть функции, позволяющие получить информацию о потоках:

* activeCount() Возвращает количество активных в настоящий момент экземпляров класса Thread.
* currentThread() Возвращает текущий объект-поток, то есть соответствующий потоку управления, который вызвал эту функцию.
* enumerate() Возвращает список активных потоков.
* join([timeout]) Поток, который вызывает этот метод, приостанавливается, ожидая завершения потока, чей метод вызван.
* getName() Возвращает имя потока.
* setName(name) Присваивает потоку имя name.
* isAlive() Возвращает истину, если поток работает.
* isDaemon() Возвращает истину, если поток имеет признак демона. Программа на Python завершается по завершении всех потоков, не являющихся демонами. Главный поток демоном не является.

1. **Особенности организации многопроцессорной программы в Python. Модуль multiprocessing.**

Multiprocessing входит в стандартную библиотеку Python, позволяет запускать задачи в разных процессах. Процессы управляются операционной системой. Каждый процесс имеет свою копию интерпретатора и всех ресурсов. Multiprocessing позволяет получить прирост производительности на многоядерных системах

*Создание нескольких процессов:*

import os

from multiprocessing import Process

def doubler(number):

result = number \* 2

proc = os.getpid()

print('{0} doubled to {1} by process id: {2}'.format(

number, result, proc))

if \_\_name\_\_ == '\_\_main\_\_':

numbers = [5, 10, 15, 20, 25]

procs = []

for index, number in enumerate(numbers):

proc = Process(target=doubler, args=(number,))

procs.append(proc)

proc.start()

for proc in procs:

proc.join()

Модуль multiprocessing позволяет вам создавать процессы таким же образом, как при создании потоков при помощи модуля threading. Пакет multiprocessing также включает ряд API, которых вообще нет в модуле threading.

*Особенности организации многопроцессорной программы в Python:*

Это использование двух или более процессорных блоков в одной компьютерной системе. Это лучший способ получить полный потенциал от нашего оборудования, используя полное количество процессорных ядер, доступных в нашей компьютерной системе.

Под многопроцессорной обработкой понимается обработка нескольких процессов одновременно несколькими процессорами. Это позволяет параллельную обработку. Меньше времени уходит на обработку работ. Это способствует гораздо более эффективному использованию устройств компьютерной системы.

При работе с параллельными приложениями в Python есть ограничение, называемое *GIL*. GIL никогда не позволяет нам использовать несколько ядер CPU, и поэтому мы можем сказать, что в Python нет настоящих потоков. GIL препятствует параллельному выполнению кода Python несколькими потоками. Блокировка может удерживаться только одним потоком за раз, и, если мы хотим выполнить поток, он должен сначала получить блокировку. Используя многопроцессорность, мы можем эффективно обойти ограничение, ведь мы используем возможности нескольких процессов и, следовательно, мы используем несколько экземпляров GIL. В связи с этим нет ограничений на выполнение байт-кода одного потока в наших программах одновременно.

1. **Асинхронное программирование в Python. Использование asyncio.**

Асинхронное программирование является некоторой моделью программирования, которая сосредоточена на координации различных задач в некотором приложении. Её цель состоит в предоставлении гарантии того, что данное приложение завершит исполнять такие задачи за наименьший возможный промежуток времени. С этой точки зрения асинхронное программирование состоит в переключении с одной задачи на другую, когда это соответствует созданию совпадению между временем ожидания и временем обработки, а следовательно, сокращению общего времени, требуемого для завершения данной программы в целом.

*Основные понятия:*

корутины — специальные функции, похожие на генераторы python, от которых ожидают (await), что они будут отдавать управление обратно в цикл событий. Необходимо, чтобы они были запущены именно через цикл событий

task в asyncio – это объект, который оборачивает coroutine, предоставляя методы для контроля ее выполнения и запроса ее статуса. task может быть создан с помощью asyncio.create\_task() или asyncio.gather().

В asyncio event loop управляет планированием и передачей ожидаемых объектов. Каждая программа asyncio имеет как минимум один event loop.

Любой объект, который можно ожидать прерывание своего процесса выполнения, называется awaitable. Ключевое слово await приостанавливает выполнение текущей подпрограммы (coroutine) и вызывает указанное ожидание awaitable.

*Цикл обработки событий:*

event loop выполняется в потоке;

получает данные из очереди;

каждая задача вызывает следующий шаг сопрограммы:

если сопрограмма вызывает другую сопрограмму (await<имя\_сопрограммы>), текущая сопрограмма приостанавливается, и происходит переключение контекста. Контекст текущей сопрограммы сохраняется и загружается контекст вызванной сопрограммы;

если сопрограмма встречает блокирующий код (I/O, sleep), текущая сопрограмма приостанавливается, и управление возвращается в event loop;

event loop получает следующие задачи из очереди 2, ...n, затем event loop возвращается к задаче 1, с которой он был прерван.

1. **Параллельное программирование. Достоинства и недостатки.**

*Параллельное программирование* - техника программирования, при которой используются все преимущества многоядерных или многопроцессорных систем. При этом параллельное программирование является одним из методов параллелизма, наравне с распределенным программированием.

*Преимущества:*

· Более быстрое время исполнения: Одним из основных преимуществ параллельного программирования является достигаемое ускорение. Отдельные потоки в одной и той же программе могут исполняться совместно или параллельно, когда они достаточно независимы друг от друга.

· Быстрота отклика

· Эффективность в потреблении ресурсов: программы могут совместно разделять одни и те же ресурсы и осуществлять к ним доступ. Вследствие этого параллельные программы могут обслуживать и обрабатывать множество запросов клиентов к данным для совместной обработки, используя значительно меньше ресурсов.

*Минусы:*

· Крушения: отдельная недопустимая операция может отрицательно сказываться на имеющейся обработке программы и может вызывать в результате крушение всей программы целиком.

· Синхронизация

1. **Понятие потокобезопасности. Причины, проблематика, способы обеспечения.**

Несколько потоков одновременно работают с критическими ресурсами без неоднозначности данных, что означает, что этот поток является потокобезопасным. *Потокобезопасность* — это концепция программирования, применимая к многопоточным программам. Код потокобезопасен, если он функционирует исправно при использовании его из нескольких потоков одновременно. В частности, он должен обеспечивать правильный доступ нескольких потоков к разделяемым данным.

Для обеспечения безопасности потоков мы используем синхронизацию и взаимное исключение. Синхронизация предназначена для управления разумным доступом к критическим ресурсам (время можно контролировать), а взаимное исключение — это единственный доступ к критическим ресурсам одновременно. В любое время взаимное исключение гарантирует, что существует один и только один поток выполнения, который входит в критическую область для работы с критическими ресурсами, которые обычно защищают ресурсы улиц.

У потокобезопасности есть и проблематика – взаимоблокировка. Взаимоблокировка возникает из-за разблокировки после блокировки, что приводит к зависанию программы.  
***4 необходимых условия для тупика:***

* Взаимоисключающие условия (другие не могут работать, когда я работаю)
* Неотъемлемое состояние (моя блокировка не может быть снята другими)
* Условия запроса и удержания (когда поток выполнения блокируется из-за запроса ресурсов, оставьте существующие ресурсы на месте)
* Условия ожидания цикла (формируется ситуация взаимного ожидания)

1. **Алгоритм выполнения многопоточной программы. Блокировка потоков.**

Потоки управления (threads) образуются и работают в рамках одного процесса. В однопоточной программе используется только 1 главный поток (некоторая “нить” последовательно исполняемых операторов, которая начинается при запуске программы, является непрерывной и обрывается только по завершении программы). При использовании многопоточности, «нить» в некоторой точке программы делится на две, три и т.д. нити, которые в дальнейшем так же могут порождать другие потоки от себя. Но при этом важно знать, что один из потоков всегда остается главным, и его завершение означает завершение всей программы.

В каждый момент времени интерпретатор знает, какую команду какой поток должен выполнить, и уделяет кванты времени каждому потоку. Несмотря на внешнюю простоту такого механизма, деятельность потоков при этом должна быть полностью согласованна. Одновременная попытка изменения двумя и более потоками объекта скорее всего приведет к нарушению его целостности и другим ошибкам исполнения программы. Классическим средством согласования потоков являются семафоры. Самый простой – lock(замок) или mutex(взаимоисключения). Если в некоторой части программы нам необходимо сохранить целостность данных согласовав потоки, то логично использовать замки. Принцип их работы можно описать как эстафету с палочкой. Пока данными пользуется один поток, он забирает «эстафетную палочку», в то время как следующий поток ожидает передачи «палочки» от первого потока, чтобы потом начать использовать те же данные.

1. **Доступ к общим ресурсам в многопоточной программе. Механизмы блокировки ресурсов модуля threading.**

В стандартном модуле threading определены нужные для разработки многопоточной (multithreading) программы классы: несколько видов семафоров (классы замков Lock, RLock и класс Semaphore) и другие механизмы взаимодействия между потоками (классы Event и Condition), класс Timer для запуска функции по прошествии некоторого времени. А также модуль Queue реализует очередь, которой могут пользоваться сразу несколько потоков.

Простейший замок может быть реализован на основе класса Lock модуля threading. Замок имеет два состояния: он может быть или открыт, или заперт. В последнем случае им владеет некоторый поток. Объект класса Lock имеет следующие методы:

· acquire([blocking=True]) Делает запрос на запирание замка.

· release() Запрос на отпирание замка.

· locked() Возвращает текущее состояние замка ( True - заперт, False - открыт).

Имеется еще один вариант замка - threading.RLock, который отличается от threading.Lock тем, что некоторый поток может запрашивать его запирание много раз. Отпирание такого замка должно происходить столько же раз, сколько было запираний.

Замки позволяют ограничивать вход в некоторую область программы одним потоком. Замки могут потребоваться для обеспечения целостности структуры данных. В случае сомнений, конечно, лучше перестраховаться и поставить замки, однако следует минимизировать общее время действия замка, так как замок останавливает другие потоки, пытающиеся попасть в ту же область программы. Отсутствие замка в критической части программы, работающей над общими для двух и более потоков ресурсами, может привести к случайным, трудноуловимым ошибкам.

Семафоры являются более общим механизмом синхронизации потоков, нежели замки. Семафоры могут допустить в критическую область программы сразу несколько потоков. Семафор имеет счетчик запросов, уменьшающийся с каждым вызовом метода acquire() и увеличивающийся при каждом вызове release(). Счетчик не может стать меньше нуля, поэтому в таком состоянии потокам приходится ждать, как и в случае с замками, пока значение счетчика не увеличится. Конструктор класса threading.Semaphore принимает в качестве аргумента начальное состояние счетчика. Методы acquire() и release() действуют аналогично описанным выше одноименным методам у замков.

Еще одним способом коммуникации между объектами являются события. Экземпляры класса threading.Event могут быть использованы для передачи информации о наступлении некоторого события от одного потока одному или нескольким другим потокам. Объекты-события имеют внутренний флаг, который может находиться в установленном или сброшенном состоянии. При своем создании флаг события находится в сброшенном состоянии. Методы экземпляров класса threading.Event:

· set() Устанавливает внутренний флаг, сигнализирующий о наступлении события.

· clear() Сбрасывает флаг.

· isSet() Возвращает состояние флага.

· wait([timeout]) Переводит поток в состояние ожидания, если флаг сброшен, и сразу возвращается, если флаг установлен.

Более сложным механизмом коммуникации между потоками является механизм условий. Условия представляются в виде экземпляров класса threading.Condition и, подобно только что рассмотренным событиям, оповещают потоки об изменении некоторого состояния. Конструктор класса threading.Condition принимает необязательный параметр, задающий замок класса threading.Lock или threading.RLock. По умолчанию создается новый экземпляр замка класса threading.RLock. Методы объекта-условия:

· acquire(...) Запрашивает замок.

· release() Снимает замок.

· wait([timeout]) Переводит поток в режим ожидания. Этот метод может быть вызван только в том случае, если вызывающий его поток получил замок.

· notify() Выводит из режима ожидания один из потоков, ожидающих данные условия. Метод можно вызвать, только овладев замком, ассоциированным с условием.

· notifyAll() Этот метод аналогичен методу notify(), но прерывает ожидание всех ждущих выполнения условия потоков.

Очередь. Один процесс имеет значение, достойное отдельного модуля. Такой модуль в стандартной библиотеке языка Python есть, и он называется Queue. Помимо исключений - Queue.Full (очередь переполнена) и Queue.Empty (очередь пуста) - модуль определяет класс Queue, заведующий собственно очередью.

1. **Работа с файловой системой в Python. Основные операции.**

Модуль OS в python предоставляет функции для взаимодействия с операционной системой. OS, поставляется под стандартные служебные модули Python. Этот модуль предоставляет портативный способ использования функциональных возможностей, зависящих от операционной системы. Модули \* os \* и \* os.path \* включают в себя множество функций для взаимодействия с файловой системой.

Основные операции: монтирование и размонтирование файловых систем, мониторинг свободного места.

Для использования какой-либо файловой системы в операционной системе она должна быть смонтирована (подключена) к ней. *Монтирование* — это многоэтапный процесс, включающий проверку файловой системы на ошибки, загрузку драйверов для конкретной файловой системы, подключение новой файловой системы к точке монтирования - какому-либо существующему каталогу. Все это можно произвести всего одной командой - mount.

Некоторые файловые системы монтируются автоматически, при загрузке ОС. Их список вместе с параметрами монтирования указан в специальном конфигурационном файле /etc/fstab. Чтобы вручную смонтировать файловую систему нужно ввести команду mount с двумя обязательными параметрами - именем файла-устройства с нужным разделом диска и имя каталога, который послужит точкой монтирования.

Для отключения файловой системы используется команда размонтирования - umount. Обычно операционная система выполняет демонтирование всех смонтированных файловых систем автоматически при перезагрузке или выключении компьютера. При демонтировании файловой системы все ее кэшированные данные, находящиеся в оперативной памяти, сбрасываются на диск. Файловые системы можно демонтировать и вручную. Более того, вы должны делать это при извлечении доступных для записи съемных устройств, таких как дискеты или USB-накопители.

1. **Понятие веб-технологий. Основные характеристики, история, назначение.**

*Веб-технологии* – это коммуникационные, информационные и иные технологии и сервисы, основываясь на которые осуществляется деятельность в Интернете или с помощью него. В первую очередь, это, конечно, сайты, а также: чаты, почта, Интернет-магазины, форумы и т.д. *Веб-технологии* – это логическая составляющая Интернет-технологий, которые включают в себя:

1. Интернет-сервисы:

* WWW – всемирная паутина

1. Работы в Интернет:

* Браузеры
* Поисковые системы
* Просмотр страниц в браузере

1. Информационные ресурсы Интернет:

* Веб-страниц, интернет-магазины, интернет-порталы
* URL и протоколы передачи данных, адресация
* Создание сайтов
* Языки веб-программирования

**Основные характеристики:**

Работает WWW по принципу клиент-серверы. То есть клиент делает запрос на серверы, серверы выдают клиенту гипермедийный текст.

Основные составляющие при работе с WWW: HTML (специальный язык разметки гипертекста), URL (адресная ссылка на ресурсы и их части: протокол, хост, порт, путь до ресурса, запрос), HTTP (протокол передачи гипертекста).

**История:**

С 1962года министерство обороны США активно вкладывало деньги в разработки технологий взаимодействия между компьютерами. Благодаря этим исследованиям, в 1969году, американскими военными был создана локальная сеть - предшественница Интернета. С 1972 года функционирует всеми нами любимая электронная почта.

Переломным этапом в развитии интернет-технологий стали 90-е годы. С изобретением в 1993 году первого браузера, предполагаемые возможности Интернета стали приобретать глобальный характер. Появились многочисленные телекоммуникационные операторы, компании электронной торговли, разработчики программного обеспечения и т.п.

**Назначение:**

Позволяет пользователю разместить, найти, обрабатывать практически любую информацию в сети Интернет.

1. **Программное обеспечение, используемое для веб-технологий. Виды, назначение, примеры.**

Создание web-сайтов и других web-технологий, а также их поддержка и развитие осуществляется с помощью специализированного ПО.

**Серверные ОС**

Назначение серверной операционной системы: под управлением этих ОС выполняются приложения, обслуживающие всех пользователей корпоративной сети, а нередко и внешних пользователей. К таким приложениям относятся современные системы управления базами данных, средства управления сетями и анализа событий в сети, службы каталогов, средства обмена сообщениями и групповой работы, Web-серверы, почтовые серверы, серверы приложений разнообразного назначения.

*Примеры серверных ОС:* Solaris, Linux, MS Windows Server, HP UX.

**Web-браузер**

Должно быть ПО, которое сможет запросить информацию, получить ее, обработать и отобразить на экран пользователя. Именно этим и занимается браузер. Браузер - окно просмотра, программа для просмотра текстового и графического содержания web-ресурсов Интернет, проигрывания мультимедийной информации, передачи управляющей, почтовой, форм-интерактивной и иных форм информации от пользователя Интернет, поиска информации в Интернет с помощью поисковых систем.

Происходит это с помощью http запросовк серверу и получения от него данных, которые обрабатываются по специальным утвержденным стандартам и таким образом формируется веб-страница.

Браузер представляет собой компьютерную программу, - так называемый "тонкий клиент", - в системе клиент-серверной технологии современного сетевого обмена информацией, когда "тонкому клиенту" делегируется лишь незначительная часть полномочий по управлению информационным процессом, не позволяя влиять на жизненно важные его элементы, от которых зависит работоспособность информационной системы.

*Примеры web-браузера:* Firefox, Chrome, Safari, MSIE, Opera.

**Web-сервер**

Чтобы опубликовать веб-сайт, нужен либо статический, либо динамический веб-сервер.

Как он работает: простыми словами, когда браузеру нужен файл, размещенный на веб-сервере, браузер запрашивает его через HTTP. Когда запрос достигает нужного веб-сервера, сервер HTTP (ПО) передает запрашиваемый документ обратно, также через HTTP.

1. При получении запроса HTTP сервер сначала проверяет существует ли ресурс по данному URL.
2. Если это так, веб-сервер отправляет содержимое файла обратно в браузер. Если нет, сервер приложений создает необходимый ресурс.
3. Если это невозможно, веб-сервер возвращает сообщение об ошибке в браузер, чаще всего «404 Not Found».

*Примеры web-сервера:* Apache, Nginx, Microsoft-IIS.

**Серверы приложений**

Предназначены выполнять обширный набор специализированных функций — от обеспечения гибких средств электронных коммуникаций до управления реляционными базами данных.

*Примеры серверов приложений:* WebSphere Application Server, Tomcat.

1. **Понятие URL: назначение, применение, состав.**

**URL (Uniform Resource Locator)** – это адрес документа или файла. Он определяет местонахождение ресурса либо документа в сети, его сетевой локатор, ссылка на него. Чтобы осуществлять взаимодействие в глобальной сети, обмениваться данными, иметь доступ к искомому URL веб-ресурса, используют протоколы – так называется свод правил и очередности действий, который позволяет устанавливать соединение устройств в Интернете. Практически мгновенная предоставление информации посредством Всемирной паутины обеспечивается благодаря поддержке протоколами TCP/IP, ATM, MPLS и прочими. Самым распространенным среди них является набор протоколов передачи данных TCP/IP.

В их числе наибольшей популярностью в сети пользуются:

* HTTP — это протокол передачи гипертекста. Он применяется при обмене данными с серверов – подключенному к Интернету компьютеру пользователя по его запросу;
* FTP — протокол, назначение которого — передача документов с сервера на персональное устройство пользователя. FTP позволяет обмениваться документами любым компьютерам сети.

Каждый URL состоит из определенной последовательности символов. Его составляющие в ниже представленной последовательности могут быть следующими:

* способ обмена данными с ресурсом или тип сетевого протокола;
* логин, то есть то имя учетной записи пользователя для обращения к ресурсу;
* пароль для входа в учетную запись;
* хост или целиком вписанное доменное имя;
* порт для подключения в пределах обозначенного хоста;
* URL-путь – содержит конкретизирующую информацию о заданных характеристиках места расположения ресурса;
* строка запроса, содержащая передаваемые на сервер параметры;
* «якорь», в функцию которого входит возможность ссылаться на определенную часть документа, страницы.

Такая последовательность указывает компьютерной системе, где конкретно хранится необходимый пользователю ресурс и путь к нему.

Выглядит следующим образом: <схема>://<логин>:<пароль>@<хост>:<порт>/<путь>?<парам>#<якорь>

Если на заре создания URL служил для определения пути к определенным файлам, то в настоящее время посредством ссылок обозначается расположение любого типа информации:

* интернет-ресурсов;
* веб-страниц;
* изображений;
* видео.

Применение идентификатора URL дает возможность создавать комфортные, понятные для восприятия пользователя конструкции символов. Благодаря чему в этом случае исчезает необходимость использовать программные коды.

Вместе с тем помимо этих достоинств существуют некоторые ограничения. В URL разрешается пользоваться определенным набором знаков:

* латиница (или символы английского алфавита);
* цифры длиной до 225 символов;
* определенные знаки пунктуации (дефис, нижнее подчеркивание);
* специальные символы (в частности, «/» — слеш, «\_» — нижнее подчеркивание).

1. **Понятие веб-сервера. Цели, принцип работы.**

*Веб-сервер* — это сложный программный комплекс, реализующий огромный набор функций и поддерживающий все нюансы современных версий протокола HTTP. Это программа, которая принимает входящие HTTP-запросы, обрабатывает эти запросы, генерирует HTTP-ответ и отправляет его клиенту. – это и есть **принцип работы**. *Веб-сервер (web-server)* – это сервер, отвечающий за прием и обработку запросов от клиентов к веб-сайту. В качестве клиентов обычно выступают различные веб-браузеры. В ответ веб-сервер выдает клиентам HTTP-ответы, в большинстве случаев – вместе с HTML-страницей, которая может содержать: всевозможные файлы, изображения, медиа-поток или любые другие данные. Также веб-сервер выполняет функцию исполнения скриптов.

Все основные и дополнительные цели веб-сервера:

* + Прием запросов от веб-браузеров по протоколу стандарта HTTP с использованием сетевых протоколов TCP/IP;
  + Выполнение поиска и отсылки файлов с гипертекстом или каких-либо документов в браузер по протоколу HTTP;
  + Обслуживание и обработка запросов, типа: mailto, FTP, Telnet и т. п.;
  + Запуск прикладных программ на веб-сервере с последующей передачей и возвратом параметров обработки через стандарт интерфейса CGI;
  + Работа и обслуживание навигационных карт изображений (Image map);
  + Загрузка Java-приложений;
  + Администрация и оперативное управление сервером;
  + Авторизация пользователей и их аутентификация;
  + Ведение регистрационного журнала обращений пользователей к различным ресурсам;
  + Автоматизированная работа веб-страниц;
  + Поддержка страниц, которые генерируются динамически;
  + Поддержка работы протокола HTTPS для защищенных соединений с клиентами.

Принцип работы веб-сервера:

Веб-браузеры поддерживают связь с веб-серверами с помощью протокола передачи гипертекстовых сообщений (HypertextTransferProtocol, HTTP). Это простой протокол запросов и ответов для пересылки информации с использованием протокола TCP/IP. Веб-сервер получает запрос, обнаруживает файл, посылает его браузеру, а затем разрывает соединение. Графическая информация, которая имеется на странице, обрабатывается таким же образом. Далее настает очередь веб-браузера – вывести на монитор пользователя загруженный из сети HTML-документ. Кроме HTML-страниц и графики, веб-серверы могут хранить любые файлы, в том числе текстовые документы, документы текстовых процессоров, видеофайлы и аудиоинформацию. На веб-сервере также выполняют свою работу различные приложения, наибольшую популярность среди которых получили поисковики и средства связи с базами данных.

Базовые функции поиска помогают пользователям отсортировывать нужную им информацию, а утилиты для связи с базами данных предоставляют пользователям веб-браузеров доступ к этой информации.

1. **Протокол HTTP. Принцип работы, назначение, основные понятия.**

*HTTP (HyperText Transfer Protocol)* – это протокол [прикладного уровня](https://en.wikipedia.org/wiki/Application_Layer) для передачи гипермедиа документов, таких как HTML. Он был разработан для связи между веб-браузерами и веб-серверами, но также может использоваться для других целей. HTTP следует классической [модели клиент-сервер:](https://en.wikipedia.org/wiki/Client%E2%80%93server_model) клиент открывает соединение, чтобы сделать запрос, а затем ждет, пока не получит ответ.

Ключевые понятия HTTP – **URL** и **URI**. URL (Uniform Resource Locator) выглядит следующим образом: <схема>://<логин>:<пароль>@<хост>:<порт>/<путь>?<парам>#<якорь> . URI (Uniform Resource Identifier) выглядит так: /<путь>?<парам>#<якорь> .

Типичные задачи, которые решает **HTTP протокол**: протокол HTTP осуществляет доступ к веб-ресурсам и обмен данными между пользовательскими приложениями.

Передача данных по HTTP протоколу осуществляется через TCP/IP соединение. Машина, которая выступает в роли сервера, использует восьмидесятый TCP порт или порт 8080.

Структура протокола определяет, что каждое HTTP-сообщение состоит из трёх частей, которые передаются в следующем порядке:

1. Запрос — определяет тип сообщения;
2. Поля запроса — характеризуют тело сообщения, параметры передачи и прочие сведения;
3. Тело сообщения (англ. Message Body) — непосредственно данные сообщения. Обязательно должно отделяться от заголовков пустой строкой.

*Метод HTTP* — последовательность из любых символов, кроме управляющих и разделителей, указывающая на основную операцию над ресурсом. Обычно метод представляет собой короткое английское слово, записанное заглавными буквами. Названия метода чувствительны к регистру.

HTTP [метод](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods) , как правило , имеет вид - [GET](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET), [POST](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST) , [OPTIONS](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/OPTIONS) или [HEAD](https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/HEAD) что определяет операцию клиент хочет выполнить. Как правило, клиент хочет получить ресурс (используя GET) или опубликовать значение [HTML-формы](https://developer.mozilla.org/en-US/docs/Web/Guide/HTML/Forms) (используя POST), хотя в других случаях может потребоваться больше операций.

Код состояния информирует клиента о результатах выполнения запроса и определяет его дальнейшее поведение. Набор кодов состояния является стандартом, и все они описаны в соответствующих документах RFC.

Каждый код представляется целым трехзначным числом. Первая цифра указывает на класс состояния, последующие - порядковый номер состояния. За кодом ответа обычно следует краткое описание на английском языке.

1. **Настройка веб-сервера. Основные конфигурационные файлы, понятия.**

Для начала установка: apt install apache2. После завершения установки открываем браузер и переходим по ссылке “http://[ip адрес сервера]”

Далее рассмотрим создание тестовой страницы. По умолчанию корневым каталогом для размещения сайта является директория “/var/www/html”, именно там находится страница приветствия. Создадим отдельную директорию “/var/www/sites” для размещения виртуальных хостов и вложенную папку “/var/www/sites/site1” с индексной страницей тестового сайта.

*Создание:*

cd /var/www/

mkdir -p sites/site1

echo “<H1>Welcome</H1>” > sites/site1/index.

Сейчас рассмотрим конфигурацию Apache-сервера. Конфигурационные файлы сайтов находятся в каталоге “/etc/apache2/sites-available/”. Создадим конфигурационный файл для нового виртуального хоста взяв за основу конфигурацию по умолчанию из файла “000-default.conf”

cd /etc/apache2/sites-available/

cp 000-default.conf site1.conf

Откроем файл “site1.conf” и изменим параметр “DocumentRoot”. В качестве значения нужно указать путь к новому сайту, в нашем случае это “/var/www/sites/site1”. На данном этапе нам не требуется настройка одновременной работы нескольких сайтов, поэтому отключим сайт по умолчанию и включим новый сайт. Для применения изменений перезагружаем конфигурацию сервера:

a2dissite 000-default

a2ensite site1

systemctl reload apache2

Снова переходим по ссылке “http://[ip\_адрес\_сервера]” и убеждаемся, что вместо стандартной страницы приветствия отображается наша новая страница. Настройка HTTP-сервера завершена.

В зависимости от версии Apache и вашего дистрибутива, **конфигурационные файлы** Apache могут находиться в следующих каталогах: /etc/apache, /etc/apache2, /etc/httpd или /etc/httpd2. Основные конфигурационные файлы называются httpd.conf, httpd2.conf или apache.conf и apache2.conf.

Первым делом откройте конфигурационный файл и найдите директиву: **#ServerName new.host.name** . Нужно ее раскомментировать и указать имя сервера, которое будут задавать пользователи в строке браузера. Данное имя должно быть зарегистрировано в DNS-сервере вашей сети. Обычно здесь указывается имя компьютера, например: **ServerName user-desktop.** После этого можно будет обращаться к серверу по адресу <http://user-desktop/>.

*Основные директивы и их описание:*

**ServerName** имя - задает имя Web-сервера, имя должно быть зарегистрированным на DNS-сервере, то есть обычно это доменное имя сервера

**ServerAdmin** e-mail - задает e-mail администратора сервера

**ServerRoot** каталог - определяет каталог с конфигурационными файлами сервера

**DocumentRoot** каталог - Позволяет задать каталог, в котором хранятся документы Web-сервера - это корневой каталог документов.

**ErrorLog файл** - задает журнал ошибок

**User –** пользователь

**Group –** группа

1. **Виртуальные хосты. Применение, настройка.**

*Веб хостинг* – услуга, предоставление ресурсов для размещения информации на сервере, постоянно, находящихся в сети – размещение сайтов и приложений. Базовая часть, которая отвечает за отдельный сайт или домен называется *виртуальным хостом*. Эта система позволяет использовать один сервер, чтобы раздавать несколько сайтов используя один интерфейс или IP. Каждый настроенный соответствующим образом домен будет направлять пользователя к определенной директории сервера.

*Настройка*:

*Подготовительная работа*:

# создание директорий

$ mkdir -p /var/www/example.com/public\_html

$ sudo mkdir -p /var/www/test.com/public\_html

# назначение прав

$ sudo chown -R $USER:$USER /var/www/example.com/public\_html

$ sudo chown -R $USER:$USER /var/www/test.com/public\_html

$ sudo chmod -R 755 /var/www

# создать соответствующие файлы сайтов

*Создание файлов виртуального хоста:*

# /etc/apache2/sites-available/example.com.conf

<VirtualHost \*:80>

ServerAdmin [admin@example.com](mailto:admin@example.com)

ServerName example.com

ServerAlias [www.example.com](http://www.example.com)

DocumentRoot /var/www/example.com/public\_html

ErrorLog ${APACHE\_LOG\_DIR}/error.log

CustomLog ${APACHE\_LOG\_DIR}/access.log combined

</VirtualHost>

*Включение новых виртуальных хостов:*

sudo a2ensite example.com.conf

sudo a2ensite test.com.conf

sudo a2ensite 000-default.conf

sudo systemctl restart apache2

*Настройка файла локальных хостов:*

$ sudo nano /etc/hosts

127.0.0.1 localhost

127.0.1.1 guest-desktop

111.111.111.111 example.com

111.111.111.111 test.com

1. **Понятие прокси-сервера. Настройка сервера nginx.**

*Рroxy-сервер* – это дополнительная служба в сети интернет, позволяющая пользователям выполнять косвенные запросы к различным веб-ресурсам и сетевым службам. Данный ресурс выступает в качестве посредника, причем на оперативности и качестве работы это отражается только в лучшую сторону, за счет использования отдельного оптоволокна (выделенного канала) с максимально возможной скоростью обмена данными.

*Прокси-сервер* — удаленный компьютер, который, при подключении к нему вашей машины, становится посредником для выхода абонента в интернет. Прокси передает все запросы программ пользователя в сеть, и, получив ответ, отправляет его обратно.

Каждому компьютеру, подключенному к интернет, присваивается уникальный ip-адрес, который несет информацию о стране и регионе абонента, номере его провайдера и номере компьютера в сети.

Прокси-сервер также имеет свой ip-адрес. Подключившись к прокси, вы передаете все запросы в интернет через него, при этом проверка показывает ip прокси-сервера, а вы остаетесь анонимным.

*Далее рассмотрим настройку nginx:* Nginx доступен в стандартных репозиториях ubuntu, а установить его мы сможем при помощи пакетов APT. Для начала обновим список пакетов apt, а потом установим nginx:

sudo apt update && sudo apt install nginx

Далее необходимо настроить файрвол. При установке Nginx регистрируется в сервисе файрвола ufw. Поэтому настройка доступа осуществляется достаточно просто. Для вывода настроек доступа введем команду:

sudo ufw app list

В результате будет выведен список профилей приложений:

Nginx Full

Nginx HTTP

Nginx HTTPS

Выбираем нужный, и пишем команду:

sudo ufw allow 'Nginx HTTP'

Далее ubuntu запустит nginx автоматически. Убедимся, введя команду systemctl status nginx . Таким образом, сервер запущен и работает. Тем не менее, лучше проверить его работоспособность путем запроса веб-страницы.

Для остановки, старта, и перезапуска сервера необходимо ввести следующие команды:

sudo systemctl stop nginx

sudo systemctl start nginx

sudo systemctl restart nginx

1. **Основные принципы криптографии. Шифры. Исторические шифры.**

Использование криптографической системы помогает зашифровать изначальный текст (открытый текст), чтобы в итоге получилась совершенно другой текст (шифртекст или криптограмма), смысл которого сможет понять только тот, для кого этот текст предназначался. Причем, получатель должен полученный текст дешифровать, чтобы понять смысл исходного текста. Кроме того, если шифртекст попадет в руки третьего лица, который не должен был ее получить, то он должен быть не способен ее дешифровать. *Надежность систем защиты информации* — это очень сложный и запутанный вопрос. Проблема заключается в том, что почти невозможно проверить степень защищенности информации. Кроме того, проблема криптографии в том, что порой на дешифрование необходимо затратить намного больше ресурсов и усилий, нежели на процесс шифрования.

**Задачи:**

* Обеспечение секретности
* Аутентификация сторон
* Обеспечение целостности

*Шифрование*– процесс криптографическогопреобразования текста на основе определенного параметра (ключа) и алгоритма. *Расшифрование*– криптографическоепреобразование шифрованного текста в исходный. *Шифр*– совокупность обратимых преобразований исходных данных в скрытый текст.

**Виды шифра:**

Симметричное шифрование (шифрование с закрытым ключом)

Есть ключ шифрования. С его помощью данные шифруются по какому-то алгоритму. Тот, кто обладает ключом и знает алгоритм, может расшифровать сообщение.

Асимметричное шифрование (шифрование с открытым ключом)

Асимметричное шифрование — это метод шифрования данных, предполагающий использование двух ключей — открытого и закрытого. Открытый (публичный) ключ применяется для шифрования информации и может передаваться по незащищенным каналам. Закрытый (приватный) ключ применяется для расшифровки данных, зашифрованных открытым ключом.

Электронные подписи (MAC, HMAC)

Электронная подпись — это асимметричное шифрование наоборот: вы зашифровываете закрытым ключом, а расшифровать может кто угодно с помощью открытого ключа, который доступен всем.

Хэширование (MD5, SHA1, SHA256)

Криптографические хэши используются везде, от хранения паролей до систем проверки файлов. Основная идея состоит в том, чтобы использовать детерминированный алгоритм (алгоритмический процесс, который выдает уникальный и предопределенный результат для задачи входных данных), который принимает один вход и создает строку фиксированной длины каждый раз.

**Исторические шифры:**

Шифр Цезаря

Классический шифр Цезаря предполагает смещение каждой буквы текста на следующую за ней через три. Последние буквы смещаются в начало алфавита по кольцу.

Шифр Вижинера

Фатальный недостаток шифра Цезаря - то, что каждый символ текста преобразуется в один и тот же символ шифротекста. Это можно исправить, используя ключ длиной не в одно число, а в несколько. Тогда первый символ смещается на первое число в ключе, второй - на второе и так далее. Дополнительно, такой ключ тоже можно воспринимать как строку - выполнив преобразование из массива чисел в массив символов.

Такой шифр имеет абсолютную криптографическую стойкость, если ключ равен по длине тексту и используется только один раз. На практике это очень неудобно и приходится использовать ключ гораздо короче, чем сам текст. В таком случае ключ просто “размножается” до нужной длины.

Шифр Энигмы

Энигма — это машина, которая использовалась нацистами во времена Второй Мировой для шифрования сообщений.

Есть несколько колёс и клавиатура. На экране оператору показывалась буква, которой шифровалась соответствующая буква на клавиатуре. То, какой будет зашифрованная буква, зависело от начальной конфигурации колес.

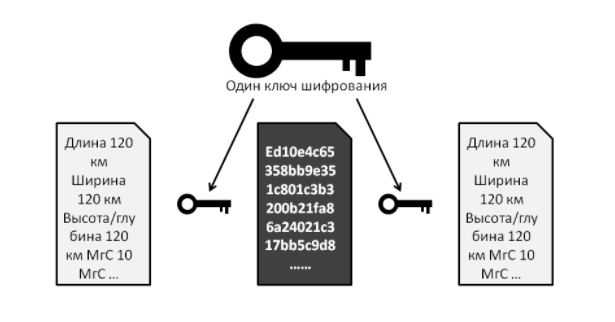
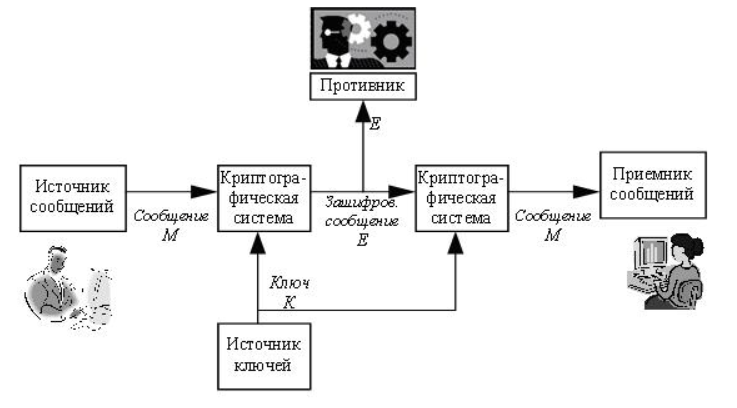
Существовало более ста триллионов возможных комбинаций колёс, и со временем набора текста колеса сдвигались сами, так что шифр менялся на протяжении всего сообщения.

1. **Симметричное шифрование. Примеры алгоритмов, общая схема, виды.**

*Симметричное шифрование* — это метод шифрования, при котором для защиты информации используется ключ, зная который любой может расшифровать или зашифровать данные. Симметричное шифрование остаётся самым актуальным и криптографически гарантированными методом защиты информации. В симметричном шифровании, основанном на использовании составных ключей, идея состоит в том, что секретный ключ делится на две части, хранящиеся отдельно. Каждая часть сама по себе не позволяет выполнить дешифрование.

Основным недостатком симметричного шифрования является то, что секретный ключ должен быть известен и отправителю, и получателю. Это создает проблему распространения ключей. Получатель на основании наличия зашифрованного и расшифрованного сообщения не может доказать, что он получил это сообщение от конкретного отправителя, поскольку такое же сообщение он мог сгенерировать самостоятельно.

*Общая схема шифрования выглядит следующим образом:*

Классическими примерами таких алгоритмов являются симметричные криптографические алгоритмы:

* Простая перестановка

Сообщение записывается в таблицу по столбцам. После того, как открытый текст записан колонками, для образования шифртекста он считывается по строкам. Для использования этого шифра отправителю и получателю нужно договориться об общем ключе в виде размера таблицы. Объединение букв в группы не входит в ключ шифра и используется лишь для удобства записи несмыслового текста.

* Одиночная перестановка по ключу

Он отличается от предыдущего лишь тем, что колонки таблицы переставляются по ключевому слову, фразе или набору чисел длиной в строку таблицы.

* Двойная перестановка

Повторно шифруется сообщение, которое уже было зашифровано. Для этого размер второй таблицы подбирают так, чтобы длины её строк и столбцов отличались от длин в первой таблице. Лучше всего, если они будут взаимно простыми. Кроме того, в первой таблице можно переставлять столбцы, а во второй строки. Наконец, можно заполнять таблицу зигзагом, змейкой, по спирали или каким-то другим способом. Такие способы заполнения таблицы если и не усиливают стойкость шифра, то делают процесс дешифрования гораздо более занимательным.

* Перестановка «Магический квадрат»

Магическими квадратами называются квадратные таблицы со вписанными в их клетки последовательными натуральными числами от 1, которые дают в сумме по каждому столбцу, каждой строке и каждой диагонали одно и то же число. Подобные квадраты широко применялись для вписывания шифруемого текста по приведенной в них нумерации. Если потом выписать содержимое таблицы по строкам, то получалась шифровка перестановкой букв. На первый взгляд кажется, будто магических квадратов очень мало. Тем не менее, их число очень быстро возрастает с увеличением размера квадрата. Так, существует лишь один магический квадрат размером 3 х 3, если не принимать во внимание его повороты. Магических квадратов 4 х 4 насчитывается уже 880, а число магических квадратов размером 5 х 5 около 250000. Поэтому магические квадраты больших размеров могли быть хорошей основой для надежной системы шифрования того времени, потому что ручной перебор всех вариантов ключа для этого шифра был немыслим.

***Виды симметричных шифров:***

блочные шифры:

* AES (англ. Advanced Encryption Standard) — американский стандарт шифрования
* ГОСТ 28147-89 — советский и российский стандарт шифрования, также является стандартом СНГ
* DES (англ. Data Encryption Standard) — стандарт шифрования данных в США
* 3DES (Triple-DES, тройной DES)
* RC2 (Шифр Ривеста (Rivest Cipher или Ron’s Cipher))
* RC5
* Blowfish
* Twofish
* NUSH
* IDEA (International Data Encryption Algorithm, международный алгоритм шифрования данных)
* CAST (по инициалам разработчиков Carlisle Adams и Stafford Tavares)
* CRAB
* 3-WAY
* Khufu и Khafre
* Kuznechik

потоковые шифры:

* RC4 (алгоритм шифрования с ключом переменной длины)
* SEAL (Software Efficient Algorithm, программно-эффективный алгоритм)
* WAKE (World Auto Key Encryption algorithm, алгоритм шифрования на автоматическом ключе)

1. **Асимметричное шифрование. Примеры алгоритмов, общая схема, преимущества и недостатки.**

Асимметричное шифрование — это метод шифрования данных, предполагающий использование двух ключей — открытого и закрытого. Открытый (публичный) ключ применяется для шифрования информации и может передаваться по незащищенным каналам. Закрытый (приватный) ключ применяется для расшифровки данных, зашифрованных открытым ключом. Асимметричное шифрование используется для защиты информации при ее передаче, также на его принципах построена работа электронных подписей.

***Принцип действия асимметричного шифрования:***

Схема передачи данных между двумя субъектами (А и Б) с использованием открытого ключа выглядит следующим образом:

* Субъект А генерирует пару ключей, открытый и закрытый (публичный и приватный).
* Субъект А передает открытый ключ субъекту Б. Передача может осуществляться по незащищенным каналам.
* Субъект Б шифрует пакет данных при помощи полученного открытого ключа и передает его А. Передача может осуществляться по незащищенным каналам.
* Субъект А расшифровывает полученную от Б информацию при помощи секретного, закрытого ключа.

***Наиболее распространенные алгоритмы асимметричного шифрования:***

* RSA (Rivest, Shamir и Adelman) — алгоритм, в основе которого лежит вычислительная сложность факторизации больших чисел. Применяется в защищенных протоколах SSL и TLS, стандартах шифрования. Используется и для шифрования данных, и для создания цифровых подписей.
* DSA (Digital Signature Algorithm) — алгоритм, основанный на сложности вычисления дискретных логарифмов. Используется для генерации цифровых подписей. Является частью стандарта DSS (Digital Signature Standard).
* Схема Эль-Гамаля — алгоритм, основанный на сложности вычисления дискретных логарифмов. Лежит в основе DSA и устаревшего российского стандарта ГОСТ 34.10–94. Применяется как для шифрования, так и для создания цифровых подписей.
* ECDSA (Elliptic Curve Digital Signature Algorithm) — алгоритм, основанный на сложности вычисления дискретного логарифма в группе точек эллиптической кривой. Применяется для генерации цифровых подписей, в частности для подтверждения транзакций в криптовалюте Ripple.

***Общая схема ассиметричного шифрования:***



Преимущества и недостатки асимметричного шифрования:

Плюс таких алгоритмов в том, что для передачи зашифрованных сообщений можно использовать открытый канал связи. Даже если злоумышленник перехватит сообщение, он не сможет прочитать его без секретного ключа. Но чтобы всё было именно так, нужно, чтобы ключ был достаточно длинный — 1024 бит и выше.

Минус асимметричного шифрования очевиден — оно работает только в одну сторону. Чтобы такое общение было двусторонним, каждый должен предоставить другому свой открытый ключ.

1. **Алгоритмы хэширования. Примеры, назначение.**

*Хэширование* – это преобразование входного массива данных определенного типа и произвольной длины в выходную битовую строку фиксированной длины. Такие преобразования также называются хеш-функциями или функциями свертки, а их результаты называют хешем, хеш-кодом, хеш-таблицей или дайджестом сообщения.

Изначально алгоритм SHA (алгоритм безопасного хеширования) был создан NSA и NIST с целью генерации хэшей или уникальных кодов на основе стандарта. В 1993 году родился первый протокол SHA, также названный SHA-0, но он почти не использовался и не оказал большого влияния. Пару лет спустя был выпущен более надежный и безопасный улучшенный вариант SHA-1, который в течение многих лет использовался для подписания цифровых сертификатов SSL/TLS для миллионов веб-сайтов. Спустя несколько лет был создан SHA-2, который имеет четыре варианта в зависимости от количества выходных битов: SHA2-224, SHA2-256, SHA2-384 и SHA2-512. В настоящее время из соображений безопасности SHA1 больше не используется, но настоятельно рекомендуется использовать SHA2 или SHA3.

Алгоритмы хеширования работают только в одном направлении: мы можем сгенерировать хэш любого контента или отпечаток пальца, но с помощью хеша или отпечатка пальца нет возможности сгенерировать исходный контент.

Среди множества различных способов создания хэшей алгоритм SHA2-256 является одним из наиболее часто используемых благодаря своему балансу между безопасностью и скоростью, это очень эффективный алгоритм и имеет высокую устойчивость к коллизиям, что очень важно для поддержания безопасности этого алгоритма хеширования. Чтобы алгоритм хеширования был безопасным, не должно быть известно о коллизиях.

***Характеристики различных типов SHA2:***

* Выходной размер: это размер символов, которые образуют хеш.
* Размер внутреннего состояния: это внутренняя хеш-сумма после каждого сжатия блока данных.
* Размер блока: размер блока, обрабатываемого алгоритмом.
* Максимальное сообщение size: это максимальный размер сообщения, к которому мы применяем алгоритм.
* Word длина: это длина в битах операции, применяемой алгоритмом в каждом раунде.
* Взаимодействия или раунды: это количество операций, которые алгоритм выполняет для получения окончательного хеша.
* Поддерживаемые операции: это операции, выполняемые алгоритмом для получения окончательного хеша.

***SHA3*** ​​— это новейший алгоритм хеширования семейства SHA. Этот новый алгоритм хеширования основан на «Конструкция из губки.» Конструкция этой губки основана на случайной функции или случайной перестановке данных, она позволяет вводить любой объем данных и генерировать любой объем данных, кроме того, функция является псевдослучайной по отношению ко всем предыдущим записям. Это позволяет SHA-3 иметь большую гибкость, цель состоит в том, чтобы заменить SHA2 в типичном TLS или [VPN](https://itigic.com/ru/tag/vpn/) протоколы, которые используют этот алгоритм хеширования для проверки целостности данных и их подлинности.

Хэш-функции имеют разнообразные применения при проведении статистических экспериментов, при тестировании логических устройств, при построении алгоритмов быстрого поиска и проверки целостности записей в базах данных. Основным требованием к хэш-функциям является равномерность распределения их значений при случайном выборе значений аргумента.

***Назначение хэширования***: проверка целостности файлов (отпечаток файла), хеширование паролей на сайтах в БД.

1. **Протокол TLS/SSL. Общая схема взаимодействия, назначение.**

Протоколы SSL и TLS обеспечивают аутентификацию сервера и клиента и шифрование соединений. **Протокол TLS преследует три цели:**

* обеспечение безопасности соединения
* идентификация сторон
* обеспечение надежной передачи данных.

Для работы TLS/SSL использует комбинацию открытого сертификата и закрытого ключа. Закрытый ключ хранится на сервере и не разглашается. SSL-сертификат используется открыто и доступен всем пользователям, запрашивающим контент.

***Схема работы SSL/TLS:***

Протокол реализуется на базе стека TCP/IP. Устанавливает алгоритмы шифрования и ключи на обоих сторонах и прокидывает шифрованный туннель, по которому могут передаваться другие протоколы.

Принцип работы SSL состоит из двух фаз: фаза рукопожатия и фаза передачи данных. Во время фазы рукопожатия клиент и сервер используют шифрование открытым ключом для того, чтобы определить параметры секретного ключа, используемого клиентом и сервером для шифрования во время фазы передачи данных. Клиент инициирует рукопожатие посылая “hello”-сообщение серверу. Такое сообщение содержит список алгоритмов симметричного шифрования, поддерживаемых клиентом. Сервер отвечает похожим “hello”-сообщением, выбрав при этом наиболее подходящий алгоритм шифрования из полученного списка. Далее сервер отправляет сертификат, который содержит его публичный ключ. *Сертификат* — это набор данных, который подтверждает подлинность. Подтвержденная третья сторона, известная как центр сертификации (CA), генерирует сертификат и проверяет его подлинность. Чтобы получить сертификат сервер должен использовать безопасные каналы для отправки своего публичного ключа в центр сертификации. Он генерирует сертификат, а также центр сертификации создает отпечаток сертификата. Далее центр сертификации создает подпись сертификата, которая формируется путем шифрования отпечатка сертификата приватным ключом центра сертификации. Для проверки сертификата сервера клиент использует публичный ключ центра сертификации для расшифровки подписи. Затем клиент самостоятельно считает отпечаток сертификата сервера и сверяет с расшифрованным. Если они не совпадают, то сертификат был подделан. Естественно, для расшифровки подписи у клиента должен быть публичный ключ центра авторизации. Поэтому клиент хранит у себя список публичных ключей подтвержденных центров сертификации. По факту многие браузерные приложения имеют подобный список, находящийся непосредственно в их коде. Когда клиент установил подлинность сервера, сервер использует шифрование открытым ключом для определения секретного ключа для обмена информацией. Фаза рукопожатия завершается отправкой “finished”-сообщений, как только обе стороны готовы начать использование секретного ключа. Начинается фаза передачи данных, в ходе которой каждая сторона разбивает исходящие сообщения на фрагменты и прикрепляет к ним коды авторизации сообщений MAC (message authentication code). Код авторизации сообщения — это зашифрованный отпечаток, вычисленный на основе содержимого сообщений. Из соображений безопасности он не совпадает с секретным ключом и вычисляется вместе с секретным ключом на стадии рукопожатия. Для получения полноценного SSL пакета каждая из сторон объединяет данные фрагмента, код авторизации сообщения, заголовки сообщения и шифруют с использованием секретного. При получении пакета каждая из сторон расшифровывает его и сверяет полученный код авторизации сообщения со своим. Если они не совпадают, то пакет был подделан.

Протокол TLS делится на два слоя: TLS Record и TLS Handshake.

**Подтверждение связи (handshake):**

1. Клиент посылает сообщение ClientHello, указывающее версию SSL или TLS и поддерживаемые клиентом методы шифрования. Это сообщение также содержит случайное число (набор байт), которое используется в последующих вычислениях. Протокол также позволяет указать поддерживаемые клиентом методы сжатия данных.
2. Сервер отвечает сообщением ServerHello, которое содержит метод шифрования, выбранный сервером из списка, предложенного клиентом, а также идентификатор сессии и еще одно случайное число. Также сервер посылает свой цифровой сертификат. Если серверу нужен сертификат для аутентификации клиента, на этом шаге он может послать клиенту запрос такого сертификата.
3. Клиент проверяет сертификат сервера.
4. Клиент отправляет случайное число, которое клиент и сервер используют для шифрования последующих сообщений. Сама строка из байт шифруется публичным ключом сервера.
5. Если сервер потребовал у клиента сертификат, клиент отсылает набор байт, зашифрованный его секретным ключом, и свой цифровой сертификат, или оповещение об отсутствии сертификата.
6. Сервер проверяет сертификат клиента.
7. Клиент и сервер отправляют друг другу сообщение ChangeCipherSpec, объявляя об изменении режима передачи данных с незащищенного на защищенный.
8. Клиент отправляет сообщение Finished, зашифрованное секретным ключом, и таким образом завершает подтверждение связи со своей стороны.
9. Аналогичные действия производит сервер.
10. На протяжении данной сессии клиент и сервер могут обмениваться сообщениями, зашифрованными секретным ключом.

**Возобновление сессии:**

1. Клиент посылает сообщение ClientHello, используя ID сессии, которую нужно возобновить.
2. Сервер проверяет, есть ли у него в кэше соответствующий идентификатор. Если есть и сервер способен возобновить сессию, он отсылает клиенту сообщение ServerHello с этим же ID сессии. Если нет, сервер генерирует новый ID сессии и выполняет процедуру handshake с клиентом.
3. Клиент и сервер обмениваются сообщениями ChangeCipherSpec, а затем Finished.
4. Передача данных по защищенному каналу возобновляется.

**Протокол записи (TLS Record):**

Этот слой защищает данные с помощью ключей, полученных при подтверждении связи, и проверяет целостность и источник входящих сообщений. Он выполняет следующие функции:

* Разбиение исходящих сообщений на блоки нужного размера и "склеивание" входящих сообщений.
* Сжатие исходящих сообщений и распаковку входящих (используется не всегда).
* Применение кода аутентификации к исходящим сообщениям и проверку входящих с помощью MAC.
* Шифрование исходящих сообщений и дешифровку входящих.

После обработки протоколом TLS Record зашифрованные данные передаются на слой TCP для передачи.

**Назначение:**

Сетевые протоколы SSL и TLS являются криптографическими протоколами, обеспечивающими аутентификацию и защиту от несанкционированного доступа, нарушения целостности передаваемых данных. Протоколы SSL/TLS предназначены для исключения подмены идентификатора на клиентской или серверной стороне, раскрытия или искажения данных.

1. **Понятие SSL-сертификата. Назначение. Самоподписанные сертификаты. Центры сертификации.**

***SSL-сертификат (Secure Sockets Layer)*** – это цифровой сертификат, удостоверяющий подлинность веб-сайта и позволяющий использовать зашифрованное соединение. Компаниям и организациям необходимо добавлять SSL-сертификаты на веб-сайты для защиты онлайн-транзакций и обеспечения конфиденциальности и безопасности клиентских данных. SSL обеспечивает безопасность интернет-соединений и не позволяет злоумышленникам считывать или изменять информацию, передаваемую между двумя системами. Если в адресной строке рядом с веб-адресом отображается значок замка, значит этот веб-сайт защищен с помощью SSL.

***Зачем нужен SSL-сертификат:***

SSL-сертификаты сайтов требуются для обеспечения безопасности данных пользователей, подтверждения прав собственности на сайт, предотвращения создание поддельной версии сайта злоумышленниками и обеспечения доверия со стороны пользователей.

Если использование веб-сайта предполагает вход в систему, ввод личных данных, таких как номера кредитных карт, или просмотр конфиденциальной информации, такой как данные медицинской страховки, или финансовой информации, то важно сохранить конфиденциальность этих данных. SSL-сертификаты помогают сохранить конфиденциальность онлайн-транзакций и гарантируют пользователям, что веб-сайт является подлинным и безопасным для ввода личных данных.

*Самоподписанный сертификат SSL* — это сертификат, подписанный лицом, которое его создало, а не доверенным центром сертификации. Самоподписанные сертификаты могут иметь тот же уровень шифрования, что и доверенный CA-подписанный SSL-сертификат. Самоподписанные сертификаты, признанные действительными в любом браузере. Самоподписанные сертификаты в основном используются для целей тестирования или внутреннего использования. Чтобы создать новый самоподписанный сертификат SSL, используйте openssl req команду.

*Центр сертификации (CA)* — это организация, которая обладает правом выдачи цифровых сертификатов. Она производит проверку данных, содержащихся в CSR, перед выдачей сертификата. В самых простых сертификатах проверяется только соответствие доменного имени, в самых дорогих производится целый ряд проверок самой организации, которая запрашивает сертификат.

Разница между самоподписными бесплатным и платными сертификатами, выданными центром сертификации как раз, и заключается в том, что данные в сертификате проверены центром сертификации и при использовании такого сертификата на сайте ваш посетитель никогда не увидит огромную ошибку на весь экран.

***Центров сертификации существует достаточно много, вот перечень самых популярных:***

Comodo — работает с 1998 штабквартира в Jersey City, New Jersey, США.

Geotrust — основан в 2001, в 2006 продан Verisign, штабквартира Mountain View, California, США

Symantec — бывший Verisign в состав которого входит и Geotrust. Купил всех в 2010 году.

Thawte — основан в 1995, продан Verisign в 1999.

Trustwave — работает с 1995, штабквартира Chicago, Illinois, США.

Как видим самый крупный игрок на рынке SSL сертификатов это Symantec, который владеет тремя крупнейшими центрами сертификации — Thawte, Verisgin и Geotrust.

**Примерные практические задания к экзамену**

1. **Напишите программу, которая создает нить. Родительская и вновь созданная нити должны распечатать десять строк текста.**

*from threading import Thread  
def Print(name):  
 for i in range(1, 11):  
 print(f"{name}: строка {i}\n")  
p = Thread(target=Print, name='t1', args=['Thread 1'])  
p.start()  
Print("Main")*

1. **Напишите простой эхо-сервер, использующий неблокирующие сокеты и клиент к нему.**

import socket  
#server  
sock = socket.socket()  
sock.bind('', 9090)  
sock.listen(1)  
conn, addr = sock.accept()  
conn.setblocking(False)  
while True:  
 try:  
 msg = conn.recv(1024)  
 if not msg:  
 break  
 conn.send(msg)  
 except socket.error:  
 pass  
conn.close()  
#client  
sock = socket.socket()  
sock.connect(('localhost', 9090))  
while True:  
 msg = input()  
 sock.send(msg.encode())  
 if not msg:  
 break  
 msgg = sock.recv(1024)  
 print(msgg.decode())  
sock.close()

1. **Напишите простой многопоточный загрузчик URL. Список URL скрипт принимает как аргументы командной строки.**

import requests  
from threading import Thread  
import sys  
URL = sys.argv[1:]  
#URL = ['https://python-scripts.com/requests' , 'https://github.com/EkaterinaSimachkova/5\_FTP\_server']  
def load(url):  
 response = requests.get(url)  
 print(f"{url}: {response.text}")  
for url in URL:  
 p = Thread(target=load, args=[url])  
 p.start()

1. **Реализуйте простой HTTP-клиент. Он принимает один параметр командной строки - URL. Клиент делает запрос по указанному URL и выдает тело ответа на терминал как текст.**

import urllib.request  
import sys  
url = sys.argv[1]  
#url = 'https://docs-python.ru/standart-library/modul-urllib-request-python/'  
req = urllib.request.Request(url)  
try:  
 request = urllib.request.urlopen(req)  
 print(request.read().decode())  
except urllib.error.HTTPError as e:  
 print("Code: ", e.code)  
 print("Message: ", e.read().decode())

1. **Напишите программу, которая вычисляет число Пи при помощи ряда Эйлера. Количество потоков программы должно определяться параметром командной строки.**

import threading  
import sys  
import math  
n = sys.argv[1]  
#n = 10  
s = 0  
def count(i):  
 global s  
 s += 1/(i\*\*2)  
i = 1  
while i < 10000:  
 if threading.active\_count() < n:  
 threading.Thread(target=count, args=[i]).start()  
 i += 1  
print(math.sqrt(s\*6))

1. **Дана функция calculate(x, y). Напишите программу, которая создает пул из 5 процессов и распределяет в этом пуле вычисление функции на промежутке х от 0 до 1 с шагом 0,1. у равняется 2 всегда.**

from multiprocessing import Pool  
def calculate(x, y):  
 return x + y  
if \_\_name\_\_ == '\_\_main\_\_':  
 pool = Pool(5)  
 args = [(x/10, 2) for x in range(11)]  
 res = pool.starmap(calculate, args)  
 for i in res:  
 print(i)

1. **Напишите программу, которая проверяет все числа от 0 на простоту и выводит простые числа на экран по мере нахождения. Числа должны проверяться в различных потоках (или процессах, по выбору студента) Программа должна работать до тех пор, пока ее не остановит пользователь.**

import threading  
def check(n):  
 k = 0  
 for i in range(2, n // 2 + 1):  
 if (n % i == 0):  
 k = k + 1  
 if (k == 0):  
 print(n)  
def Stop():  
 global Work  
 stop = input()  
 Work = False  
print("Нажмите enter для остановки")  
n = 0  
max\_thread = 100  
Work = True  
threading.Thread(target=Stop).start()  
while Work:  
 if threading.active\_count() < max\_thread:  
 threading.Thread(target=check, args=[n]).start()  
 n += 1

1. **Напишите программу, которая обходит все файлы в директории, переданной ей как параметр и выводит на экран имена тех, чей размер задан как второй параметр. Реализовать рекурсивный обход поддиректорий.**

import os  
import sys  
dir = sys.argv[1]  
size = sys.argv[2]  
#dir = "C:\WorkPlace"  
#size = 12  
for dirpath, dirnames, filenames in os.walk(dir):  
 for file in filenames:  
 if os.stat(f"{dirpath}\{file}").st\_size == size:  
 print(file)

1. **Напишите программу, которая выводит на экран список номеров открытых портов на данной машине. Использовать команду netstat.**

import subprocess  
p = subprocess.run('netstat -tulpn', shell=True, capture\_output=True, text=True)  
print(p.stdout)  
# -l или --listening - посмотреть только прослушиваемые порты;  
# -p или --program - показать имя программы и ее PID;  
# -t или --tcp - показать tcp порты;  
# -u или --udp показать udp порты;  
# -n или --numeric показывать ip адреса в числовом виде.

1. **Напишите программу, которая копирует файл с удаленного хоста в текущую папку по SSH. Имя файла и адрес хоста принимать как параметры.**

import os

import sys

def main():

# Если при запуске не были указаны параметры:

if len(sys.argv) != 3:

# Ввод первой части

auth\_and\_ip = input(

"Введите имя пользователя и ip адрес (например, root@127.0.0.1)\n->"

)

# Ввод названия файла

file\_path = input(

"Введите путь до файла для копирования с удалённого хоста (например, /root/test.txt)\n->"

)

# Если были указаны параметры

else:

auth\_and\_ip = sys.argv[1]

file\_path = sys.argv[2]

# Формирование итоговой строки

command\_result = f"scp -r {auth\_and\_ip}:{file\_path} ."

# Вывод на экран

print(f"Сформировали команду: '{command\_result}'")

r = os.system(command\_result)

# Если все норм

if r == 0:

print("Успешно скопировали файл")

else:

print("Что-то пошло не так при копировании файла")

if \_\_name\_\_ == "\_\_main\_\_":

main()