GeekBrains

Автоматизация тестирования консольных приложений Linux на Python

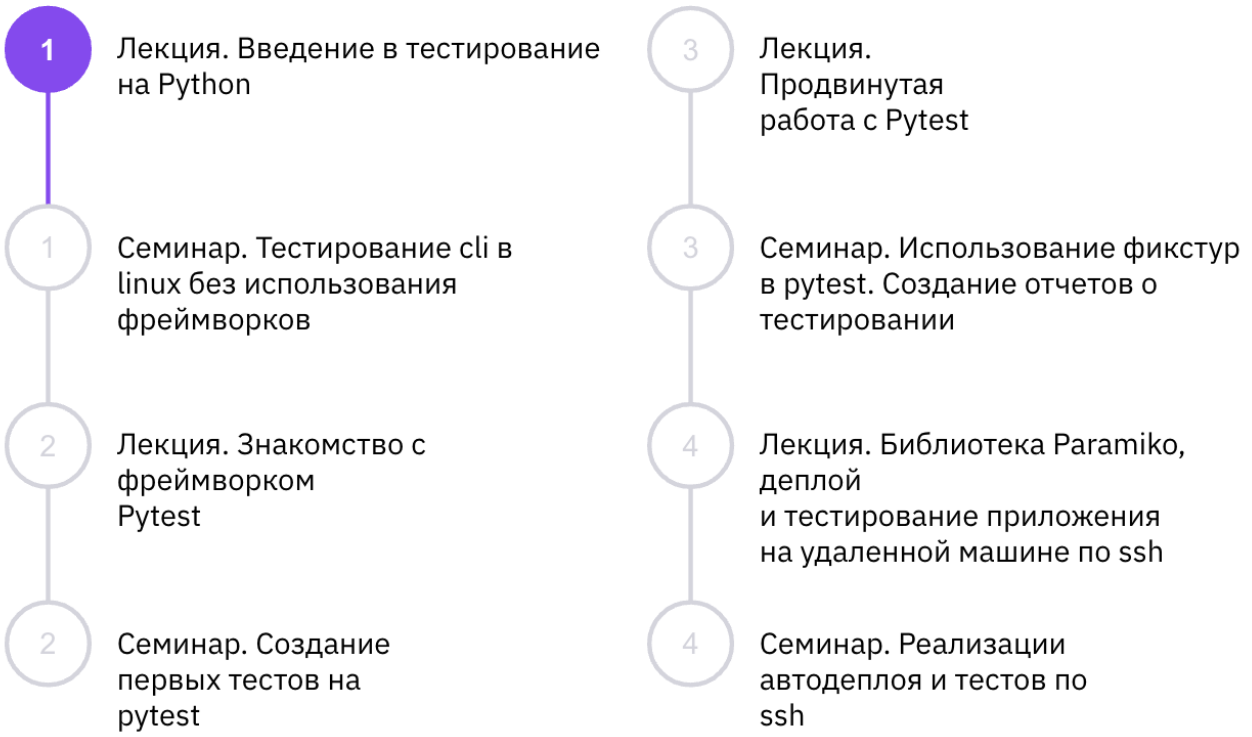
Лекция 1
Введение в тестирование на Python

Оглавление

Введение

Добро пожаловать на курс «Автоматизация тестирования консольных приложений Linux на Python»!

Курс состоит из 4 лекций и 4 семинаров, на которых мы научимся решать задачи автоматизации тестирования с использованием языка Python.



Освоив этот курс, вы на реальных примерах получите практические навыки решения основных задач в области автоматизации тестирования CLI-приложений на серверах с Linux.

На этом уроке вы узнаете:

- Зачем писать автотесты.
- Почему их нужно писать на Python.
- Какие виды тестов можно автоматизировать.
- Зачем в тестировании нужен ООП.
- Что такое bash.
- Что такое автодеплой.
- Как вызывать из Python команды операционной системы.

Термины, используемые в лекции

Автотесты — это тесты, которые выполняет компьютер, а не человек.

Модульное тестирование (блочное тестирование, unit-тестирование) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки.

Функциональное тестирование — тестирование ПО в целях проверки реализуемости функциональных требований.

Разработка тестов методом белого ящика (white box test design technique) — процедура разработки или выбора тестовых сценариев на основании анализа внутренней структуры компонента или системы.

Разработка тестов методом чёрного ящика (black box test design technique) — процедура создания и/или выбора тестовых сценариев, основанная на анализе функциональной или нефункциональной спецификации компонента или системы без знания внутренней структуры.

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Bash — усовершенствованная и модернизированная вариация командной оболочки Bourne shell. Одна из наиболее популярных современных разновидностей командной оболочки UNIX.

Cli (Command line interface) — интерфейс командной строки.

Что такое автотесты?

Автотесты — это тесты, которые выполняет компьютер, а не человек. По сути, это программа, которая тестирует, то есть проверяет работу другой программы.

В этом курсе мы научимся писать автотесты на Python. У вас могут возникнуть закономерные вопросы:

1. Зачем писать автотесты?
2. Почему их стоит писать именно на Python?

В ходе лекции мы на них ответим.

Зачем писать автотесты?

С точки зрения менеджера проекта: важно выпустить качественный продукт, минимизировать сроки и затраты. Автотесты помогают достичь этих целей.

- ✅ Автотесты находят дефекты (баги), то есть позволяют улучшить качество продукта.
- ✅ Они делают это гораздо быстрее ручных тестировщиков, то есть сокращают сроки тестирования.
- ✅ Сокращение сроков и отказ от ручного тестирования в областях, покрытых автотестами, сокращают затраты на проект.

С точки зрения программиста: автотесты помогают находить проблемы и исправлять ошибки.

- ✅ Автотесты находят ошибки на первых этапах разработки, когда исправить их проще всего.
- ✅ Хорошо написанные автотесты указывают на причину проблемы, помогают локализовать её.

С точки зрения тестировщика: автоматизируют рутинные операции и помогают справиться с большим количеством проверок.

Рассмотрим на примере консольной утилиты архиватора 7zip для Windows.

💡 Для более детального понимания последовательности работы с кейсом посмотрите видеолекцию.

Чтобы узнать возможности утилиты, откроем командную строку и вызовем 7zip.exe с ключом **-h** (help — справка). Откроется справка по поддерживаемым командам (Commands) и ключами (Switches).

```
c:\7Zip>7z.exe -h

7-Zip 22.01 (x64) : Copyright (c) 1999-2022 Igor Pavlov : 2022-07-15

Usage: 7z <command> [<switches>...] <archive_name> [<file_names>...] [@listfile]

<Commands>
  a : Add files to archive
  b : Benchmark
  d : Delete files from archive
  e : Extract files from archive (without using directory names)
  h : Calculate hash values for files
  i : Show information about supported formats
  l : List contents of archive
  rn : Rename files in archive
  t : Test integrity of archive
  u : Update files to archive
  x : eXtract files with full paths
```

```
<Switches>
  -- : Stop switches and @listfile parsing
  -ai[r[-|0]]{@listfile|!wildcard} : Include archives
  -ax[r[-|0]]{@listfile|!wildcard} : eXclude archives
  -ao{a|s|t|u} : set Overwrite mode
  -an : disable archive_name field
  -bb[0-3] : set output log level
  -bd : disable progress indicator
  -bs{o|e|p}{0|1|2} : set output stream for output/error/progress line
  -bt : show execution time statistics
  -i[r[-|0]]{@listfile|!wildcard} : Include filenames
  -m{Parameters} : set compression Method
    -mmt[N] : set number of CPU threads
    -mx[N] : set compression level: -mx1 (fastest) ... -mx9 (ultra)
  -o{Directory} : set Output directory
  -p{Password} : set Password
  -r[-|0] : Recurse subdirectories for name search
  -sa{a|e|s} : set Archive name mode
  -scc{UTF-8|WIN|DOS} : set charset for for console input/output
  -scs{UTF-8|UTF-16LE|UTF-16BE|WIN|DOS|{id}} : set charset for list files
  -scrc[CRC32|CRC64|SHA1|SHA256|*] : set hash function for x, e, h commands
```

```
  -sdel : delete files after compression
  -seml[.] : send archive by email
  -sfx[{name}] : Create SFX archive
  -si[{name}] : read data from stdin
  -slp : set Large Pages mode
  -slt : show technical information for l (List) command
  -snh : store hard links as links
  -snl : store symbolic links as links
  -sni : store NT security information
  -sns[-] : store NTFS alternate streams
  -so : write data to stdout
  -spd : disable wildcard matching for file names
  -spe : eliminate duplication of root folder for extract command
  -spf : use fully qualified file paths
  -ssc[-] : set sensitive case mode
  -sse : stop archive creating, if it can't open some input file
  -ssp : do not change Last Access Time of source files while archiving
  -ssw : compress shared files
  -stl : set archive timestamp from the most recently modified file
```

```
  -stm{HexMask} : set CPU thread affinity mask (hexadecimal number)
  -stx{Type} : exclude archive type
  -t{Type} : Set type of archive
  -u[-][p#][q#][r#][x#][y#][z#][!newArchiveName] : Update options
  -v{Size}[b|k|m|g] : Create volumes
  -w[{path}] : assign Work directory. Empty path means a temporary directory
  -x[r[-|0]]{@listfile|!wildcard} : eXclude filenames
  -y : assume Yes on all queries

c:\7Zip>
```

Из справки видно, что при запуске утилиты мы можем указать одну из 11 команд. Самое интересное начинается при использовании ключей. Их очень много. И самое страшное — их можно комбинировать.

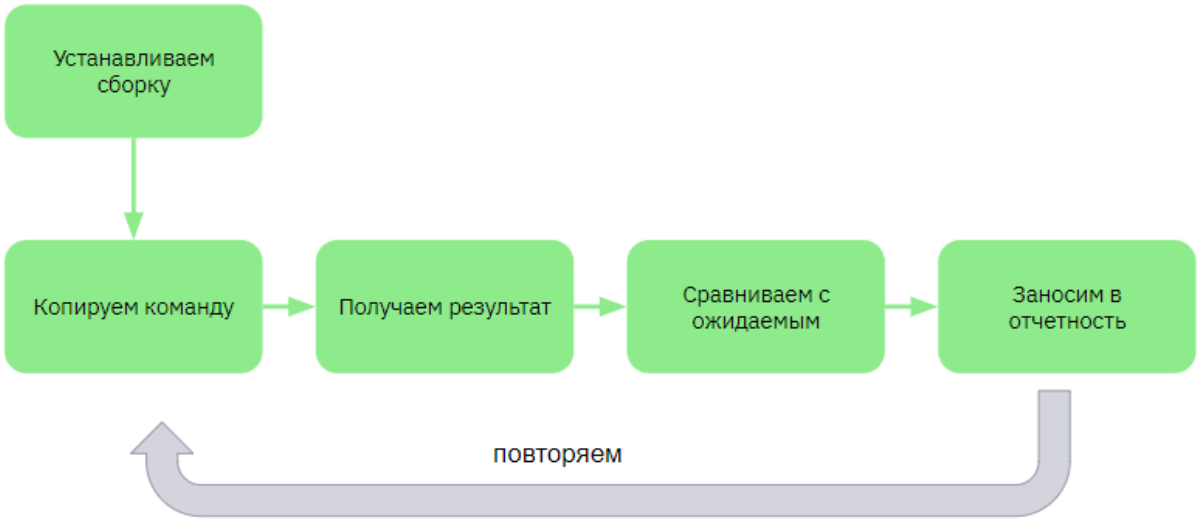
Число комбинаций всех ключей огромно — 2^n (где n — число ключей). Провести ручное тестирование перебором всех комбинаций — невыполнимая задача. Такое количество комбинаций сложно проверить даже с автотестами.

Мы не будем ставить себе невыполнимых задач. Допустим, нам нужно проверить работу каждой команды с каждым (одним!) ключом. Мы насчитали 47 ключей, при этом некоторые из них содержат параметры. Упрощённо предположим, что есть 60 вариантов ключей. Значит, нам нужно выполнить 11 (количество команд) * 60 (количество ключей) = 660 проверок. Понятно, что некоторые ключи могут быть несовместимы с некоторыми командами, поэтому реальное число проверок будет меньше.

Теперь представьте себе ручного тестировщика, которому нужно выполнить 600 таких проверок. Что он почувствует? Сначала ему станет грустно, а потом он захочет облегчить себе жизнь, автоматизировав рутинные операции.

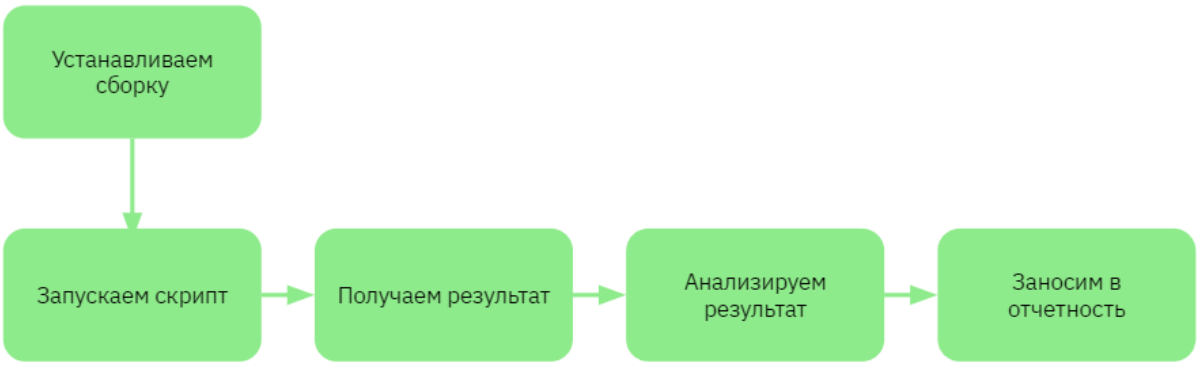
Как тестировщик может упростить работу?

Чтобы не писать команды каждый раз, можно сохранить их в текстовый файл (с использованием копирования и замены) и вставлять их в терминал. Это будет первый шаг на пути к автоматизации.

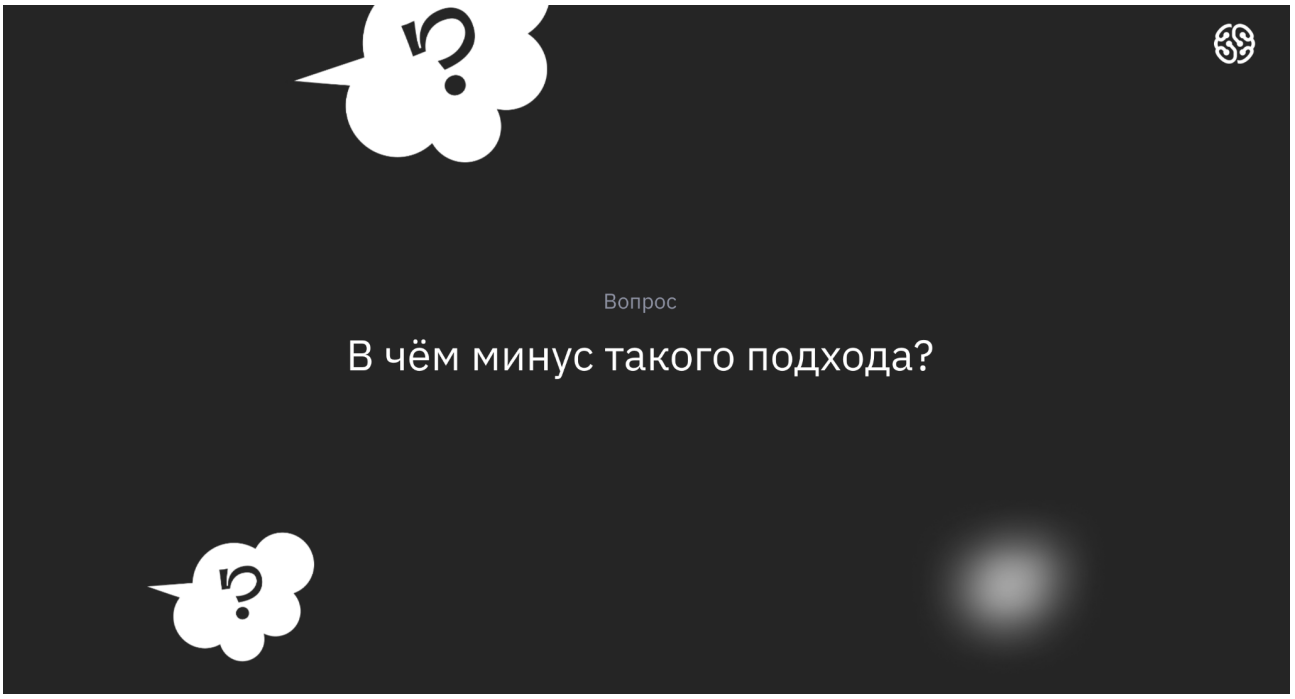


Но копировать и вставлять всё ещё нужно руками. Возникает логичное желание автоматизировать и это.

Раз мы всё равно работаем в командной строке, можно написать bat-файл, который последовательно выполняет все команды.

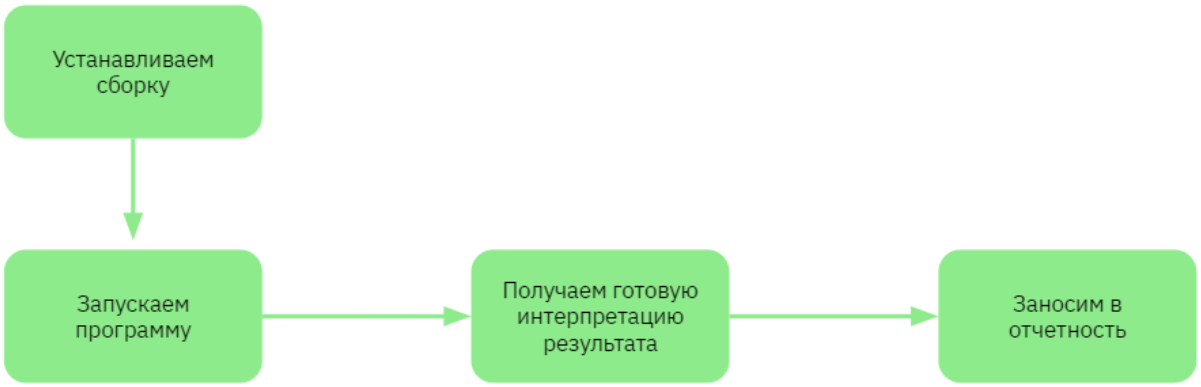


Уже лучше — копировать ничего не надо. Но есть существенные недостатки.



Нам не хватает проверки результатов.

- ➔ Можно проверять их вручную, наблюдая за тем, какую информацию выводит скрипт. Но мы уже вошли во вкус и хотим дальнейшей автоматизации.
- ➔ Можно добавить проверки в bat-файл, но становится ясно, что командной строки нам начинает не хватать. На самом деле в bat-файле можно написать очень многое, но тесты разрастаются, начинают выглядеть очень сложно и поддерживать их тоже непросто.
- ➔ Логично начать использовать более мощный и гибкий инструмент — язык программирования (почему Python мы обсудим чуть позже).

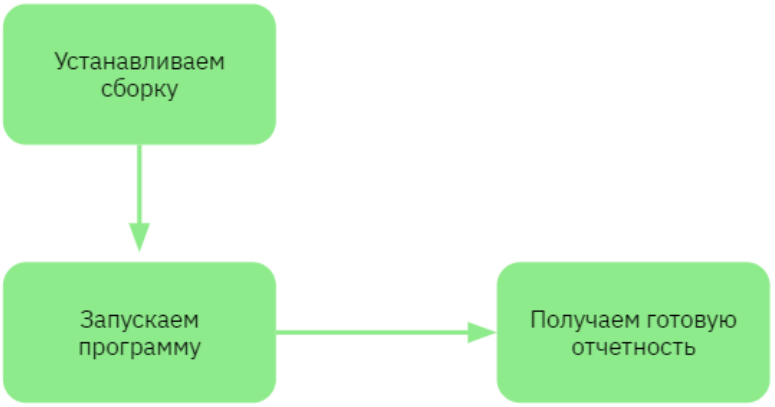


Но и это ещё не всё. Нам хочется формировать отчётность автоматически. С использованием языка программирования мы сможем сформировать практически любую отчётность — помогут специальные подключаемые библиотеки.

Можно ли автоматизировать ещё больше?

Давайте взглянем на то, что у нас уже есть, и разберёмся, как это работает.

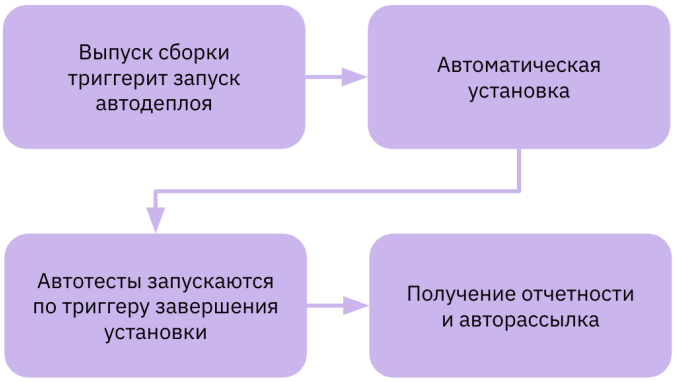
1. Разработчики выпускают новую сборку программы и передают её в тестирование.
2. Тестировщик устанавливает программу, запускает автотесты и получает отчёт.



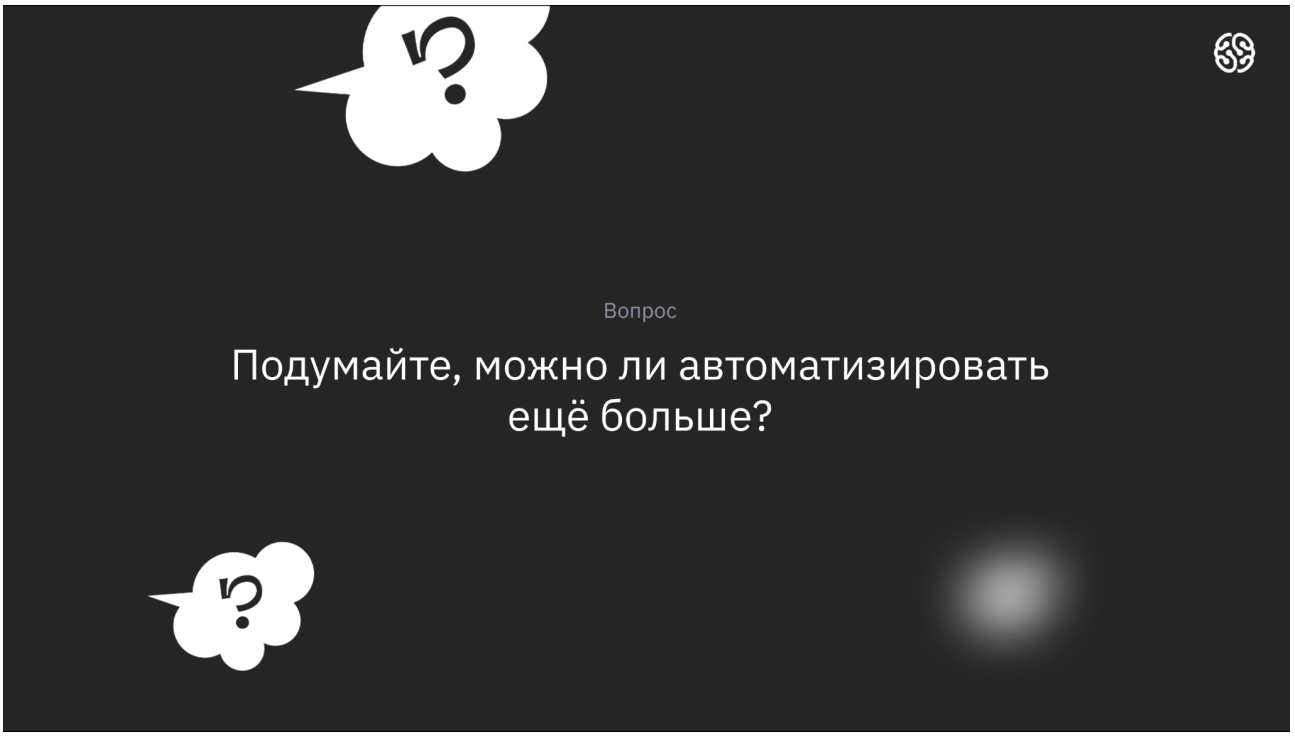
Обычно автоматизаторы — люди «энергосберегающие». Их цель — работать как можно меньше, а в идеале — не работать вообще. У нас остались ещё две ручные операции: установить сборку и запустить автотесты. Можно ли их автоматизировать? Можно.

Процесс этот выглядит так: после завершения сборки сборочная система посылает сигнал нашей тестовой системе о том, что сборка завершена и файлы дистрибутива выложены в хранилище. Наша тестовая система берёт файлы из хранилища, сама устанавливает их (это называется автодеплой), запускает автотесты, получает отчёт о результатах, размещает отчёт в системе и/или отправляет его по почте или в мессенджер всем заинтересованным лицам.

Дальнейшая автоматизация



Теперь мы достигли цели и руками делать вообще ничего не надо!



А теперь перейдём ко второму вопросу раздела.

Почему автоматизировать тесты лучше на Python?

1. **У Python простой синтаксис и низкий порог вхождения.**

   Он прощает мелкие ошибки разработчика. На нём легко освоить азы программирования и начать писать простые скрипты. У него большое и отзывчивое комьюнити, на любой вопрос можно довольно быстро найти ответ в сети.

   Ниже — пример кода, который нужно написать на Java и на Python, чтобы вывести на экран Hello World. Простота Python очевидна:

Источник: Denidenx.com

   У каждого языка свои достоинства и недостатки. На Python мы можем писать тесты проще, быстрее и лаконичнее. Java сложнее, но он выигрывает в скорости выполнения программы и кроссплатформенности.

2. **Python — один из самых популярных языков высокого уровня, и он активно развивается.**

Крупнейшие корпорации используют Python в разработке. А с учётом того, что идеальное сочетание — разработка и автотестирование на одном языке, Python — правильный выбор. Хотя я, например, провожу функциональное тестирование при помощи Python программ, написанных на C++, и это вполне работоспособный вариант.

3. **Python — универсальный язык.**

   У него есть библиотеки и для мобильных приложений, и для десктопных. На нём можно писать скрипты автоматизации для серверов и веб-приложений.

4. **Код программы на Python всегда доступен, не требует компиляции и в него можно быстро вносить изменения.**

   Открытость кода может быть и минусом, но, поскольку автотесты применяются внутри компании, то закрывать код нет нужды.

Подходы к тестированию

Благодаря гибкости, Python (не зря он назван в честь змеи!) можно использовать в автоматизации самых разных видов тестирования: модульного (юнит), функционального, нагрузочного, приёмочного, интеграционного и других.

Но чаще всего его применяют для юнит- и функционального тестирования. Почему?

При приёмочном и интеграционном тестировании важно учесть особенности взаимодействия программы с конечным пользователем, поэтому полностью их автоматизировать нельзя. Для нагрузочного тестирования Python можно использовать, но есть ряд узкоспециализированных инструментов, которые лучше заточены именно под него: например, Apache JMeter.

Сравним оставшиеся виды: unit-тестирование и функциональное тестирование.

Unit-тестирование и функциональное тестирование

Модульное тестирование (блочное тестирование, unit-тестирование) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы, наборы из одного или более программных модулей вместе с соответствующими управляющими данными, процедурами использования и обработки.

Функциональное тестирование — тестирование, которое направлено на проверку соответствия функциональных требований ПО к его реальным характеристикам.

Фактор	Unit-тесты	Функциональные тесты
Цели	Убедиться, что отдельные модули работают должным образом	Убедиться, что программное приложение работает должным образом
Кто пишет	Как правило, программисты	Команда тестирования
Стадия	Цикл разработки	Цикл тестирования
Техники	Техника белого ящика	Техника чёрного ящика

Как выглядит unit-тестирование на Python?

Упрощённо процесс можно представить так: из основной программы в тестовую импортируется тестовый класс, ему передаются тестовые данные, затем проверяется: то, что возвращают методы класса с передачей им тестовых данных, соответствует ожидаемому значению.

В unit-тестировании активно задействован ООП, и нужно хорошо знать структуру тестируемых классов. Поэтому этот вид тестов, как правило, пишут сами разработчики. Код юнит-тестов можно написать без использования дополнительных фреймворков, просто сравнивая возвращаемое значение с ожидаемым и выполняя какую-то логику в зависимости от результата, применяя условие if. Но это не очень удобно.

В стандартную поставку Python включен модуль **unittest**, предназначенный для такого тестирования. Код теста с использованием этого фреймворка выглядит таким образом:

```
import unittest
from Calculator import Calculator
class TestCalculator(unittest.TestCase):
  #setUp method is overridden from the parent class TestCase
  def setUp(self):
    self.calculator = Calculator()
  #Each test method starts with the keyword test_
  def test_add(self):
    self.assertEqual(self.calculator.add(4,7), 11)
  def test_subtract(self):
    self.assertEqual(self.calculator.subtract(10,5), 5)
  def test_multiply(self):
    self.assertEqual(self.calculator.multiply(3,7), 21)
  def test_divide(self):
    self.assertEqual(self.calculator.divide(10,2), 5)
# Executing the tests in the above test case class
if __name__ == "__main__":
  unittest.main()

```

Здесь для сравнения используются специальные функции **assert** (одна из них — **assertEqual**, которая просто проверяет равенство).

Очевидно, что удобнее всего проводить на Python unit-тестирование кода, который тоже написан на Python. Иначе могут возникнуть сложности с импортов классов.

Как уже сказано, чаще всего unit-тесты пишут программисты, которые и создавали программу. Для тестировщиков более актуально функциональное тестирование, которое мы и рассмотрим в этом курсе.

В функциональных требованиях описывается ожидаемое поведение системы.

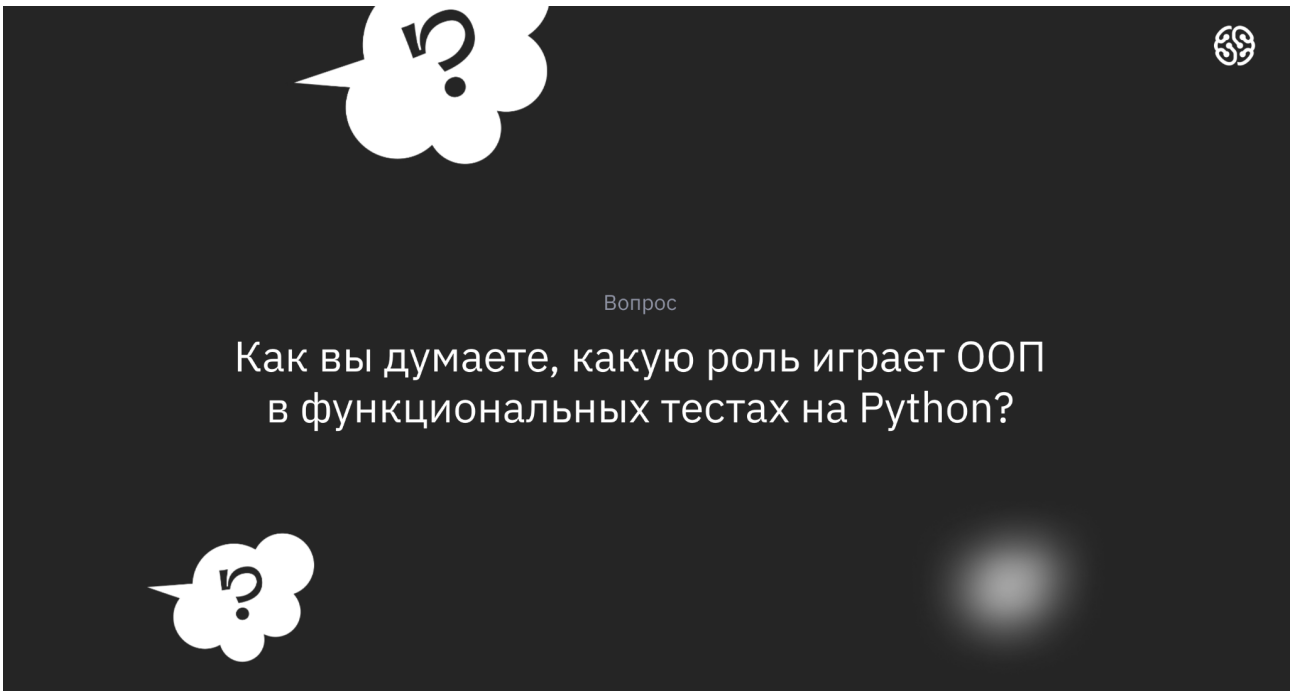
Таким образом, в ходе функционального тестирования мы работаем уже не с кодом, а с развёрнутым ПО. При автоматизированном функциональном тестировании мы автоматически из кода выполняем некоторые операции, команды и анализируем результат выполнения.

Например, вы ранее тестировали API при помощи SOAPUI или Postman. Суть тестирования сводилась к отправке запросов и обработке их ответов. Но запросы можно отправлять и через Python (для этого у него есть специальные библиотеки). При этом нам в программу вернётся http-ответ, который мы можем автоматически проанализировать. Использование Python даёт нам возможность отправлять запросы с параметрами, запросы, зависящие друг от друга, выполнять какое-либо действие в зависимости от результата запроса и многое другое.

Вернёмся к примеру с архиватором 7z. В Python есть модуль для взаимодействия с операционной системой, который позволяет запускать команды, получать их вывод, а также получать доступ к файлам. Этого достаточно, чтобы написать функциональные автотесты для архиватора.

Как мы видели, для unit-тестов активно использовался ООП.

Роль ООП в функциональных тестах на Python



Без ООП в Python не обходится ничего. Все данные в Python — объекты (даже просто переменная). Но, в отличие от unit-тестирования, при функциональном тестировании мы используем не объекты тестируемой программы, а уже развёрнутое ПО (методика чёрного ящика — black box). Поэтому сложных структур в

нашем коде не будет. Тем не менее писать код для тестов с нуля без фреймворков — плохая идея.

Фреймворки дают нам массу удобств и дополнительных возможностей. Далее в курсе мы будем использовать фреймворк **PyTest**: он более продвинутый, чем unittest, и подходит в том числе для функционального тестирования. Подробнее мы рассмотрим его в следующих занятиях, а сейчас заметим, что при его использовании не обойтись без знания ООП, но сложные конструкции и классы мы использовать не будем.

Тестирование для Linux, bash, распараллеливание тестов

Этот курс посвящён тестированию консольных приложений для Linux. Почему выбрана именно эта тема?

На самом деле Linux более распространён, чем вы можете себе представить. В сегменте серверных ОС он доминирует. Например, Linux обеспечивает работу серверов, на которых работает более 80% из первого миллиона доменов в мире (Usage share of operating systems — Wikipedia)

Никто не знает, сколько всего существует дистрибутивов Linux, ведь каждый может сделать собственный дистрибутив. По данным сайта DistroWatch.com, активно развивающихся дистрибутивов около 300.

В десктопном сегменте популярность Linux растёт, в том числе и вследствие проводимой в России политики импортозамещения. Разработан ряд отечественных дистрибутивов Linux, например Astra Linux, Alt Linux, RedOS и другие.

Почему же рассматривается именно тестирование консольных приложений?

Дело в том, что в Linux графическая подсистема не является обязательной (на серверах обычно её вообще нет) и вся работа с ПО ведётся через консоль или API. Даже в приложениях, у которых графический интерфейс всё же есть, зачастую он просто дублирует функции, которые доступны посредством командной строки. Это упрощает разработку автотестов, поскольку взаимодействовать с графическим интерфейсом в автотестах намного сложнее.

Что нам нужно, чтобы написать простейший автотест в Linux, пока даже без использования Python? Нужно уметь выполнять команды, получать их вывод и код возврата, а также уметь проверять наличие текста в выводе.

Для этого нам нужно познакомиться с основами Bash.

Что такое Bash?

Bash — усовершенствованная и модернизированная вариация командной оболочки Bourne shell. Одна из наиболее популярных современных разновидностей командной оболочки UNIX. Можно сказать, что это аналог командной строки в Windows. Возможности Bash огромны, по нему написаны книги. Но не волнуйтесь, нам понадобятся только некоторые основные команды:

cd <путь> — смена каталога (в Linux пути разделяются прямым слешем: /home/user)
ls — просмотр содержимого текущего каталога
touch <имя файла> — создание файла
echo <текст> — вывод текста (на экран или перенаправить в файл, добавив >> имя файла)
rm <путь> — удаление
cat <путь к файлу> — вывод содержимого файла на экран
nano — текстовый редактор

Выполнение любой программы завершается:

- возвратом кода возврата — 0, если программа завершена без ошибок,
- не 0 (раньше это был код ошибки, сейчас это не всегда верно), если программа завершена с ошибкой.

Вывести на экран код возврата последней выполненной в терминале программы можно командой **$?**.

Выше мы рассмотрели пример командного сценария для Windows. Аналог bat-сценариев Windows в Linux — скрипты с расширением sh. Это текстовый командный файл, который должен начинаться со строки **#!/bin/bash**, а далее идут команды bash. Команды разделяются переводом строки либо точкой с запятой.

Для нас важно, что в скриптах можно использовать переменные и условия. Переменные задаются при помощи оператора присваивания **=**. Их тип указывать не нужно. Обратиться к переменной можно, указав знак **$** перед её именем.

В автотестах нам часто нужно будет сохранить вывод какой-нибудь программы в переменную. Это можно сделать при помощи оператора вида **result=$(команда)**.

Нам также будут интересны условный оператор и логические операции.

Условный оператор в bash выглядит следующим образом:

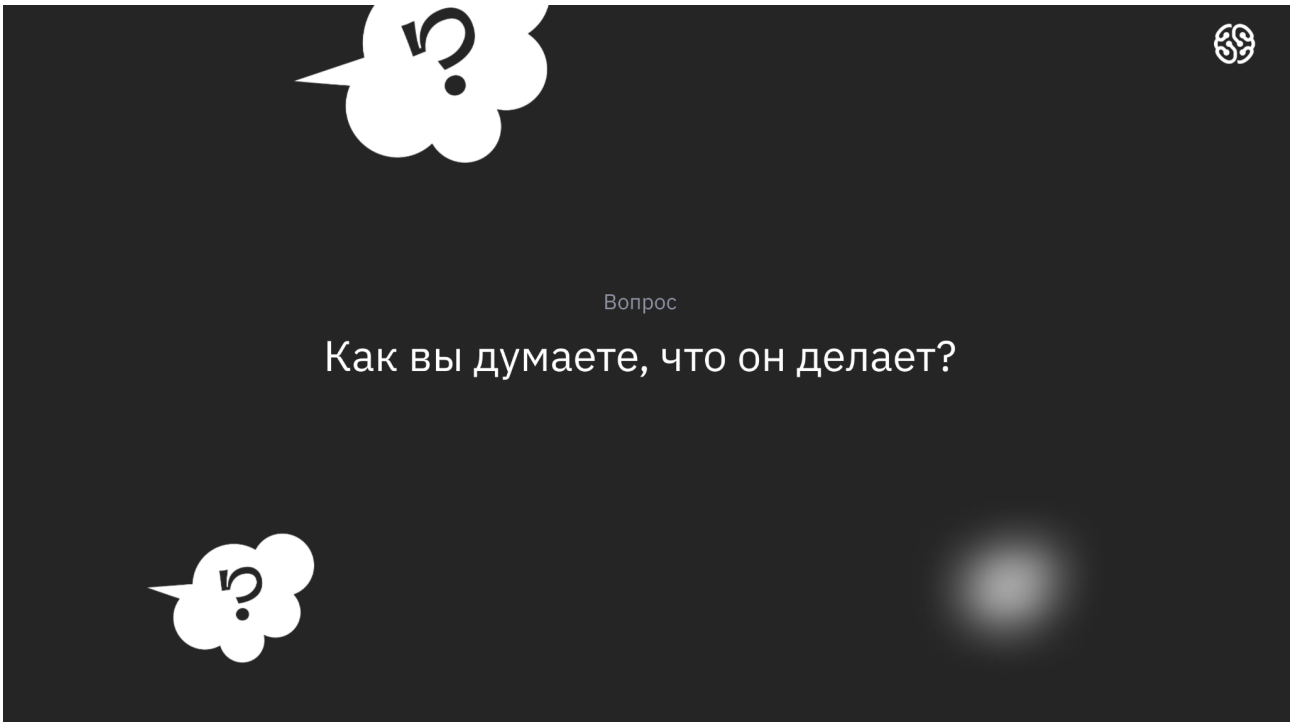
```
if [[ условие ]];
then оператор1
else оператор2
fi
```

Для проверки равенства можно использовать оператор **==**. Для конструирования составного условия — операторы **&&** (и) и **||** (или). Для проверки вхождения подстроки в строку — регулярные выражения. Например, условие **$RESULT == *"TEST"*** (здесь * обозначает любое количество любых символов) будет верным, если строка, хранящаяся в переменной RESULT, содержит подстроку TEST.

> 💡 Чтобы скрипт можно было выполнить, его нужно сделать исполняемым, выполнив команду **chmod +x <путь к скрипту>**. После этого скрипт можно будет выполнять, просто набрав в терминале его имя (или полный путь, если пользователь находится в другом каталоге).

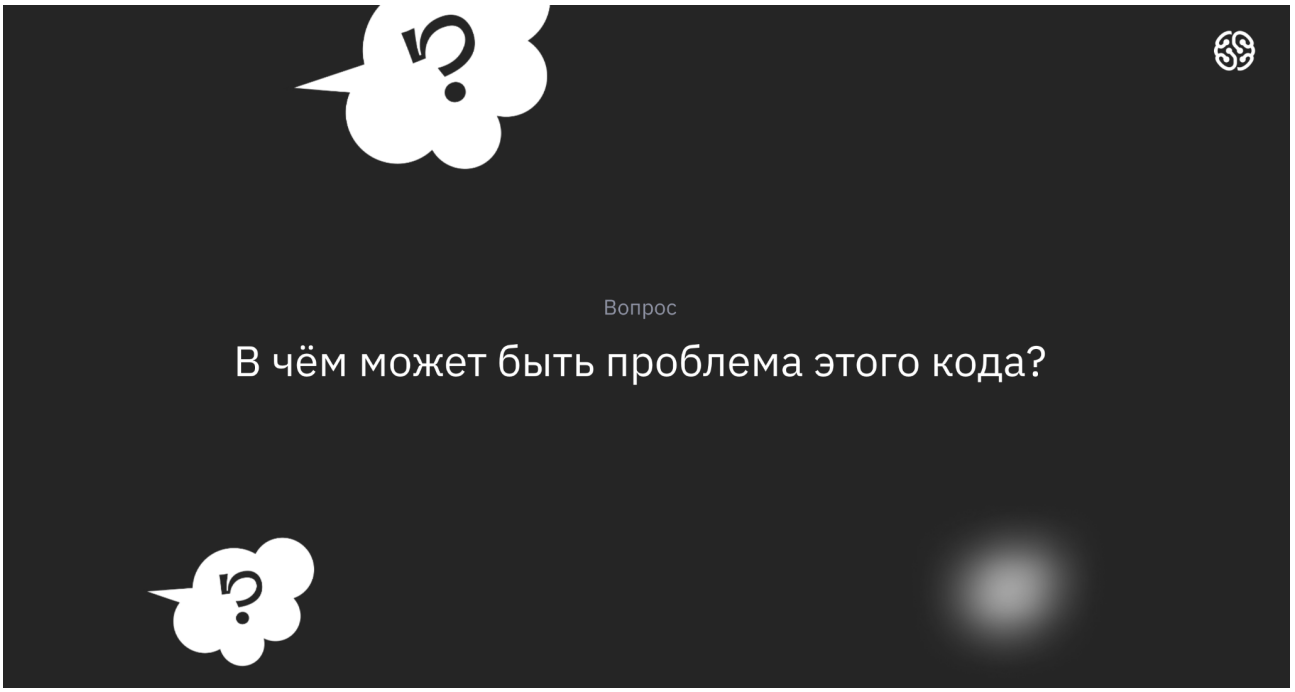
Теперь мы знаем достаточно, чтобы написать наш первый автотест на bash. Вот его код:

```
1 #!/bin/bash
2 RESULT=$(ls /etc)
3 if [[ $RESULT == *"apt"* && $? == 0 ]];
4 then echo "SUCCESS"
5 else echo "FAIL"
6 fi
```



- В переменную RESULT сохраняется список (это строка, не путайте со списком Python) подкаталогов и файлов каталога /etc.
- Проверяется, есть ли подстрока apt. Если есть, выводится SUCCESS, иначе FAIL.

Предполагается, что тест проверяет наличие подкаталога apt в каталоге /etc.



Проблема в том, что apt может содержаться в имени другого файла или каталога. Нужно аккуратнее обрабатывать вывод. В bash это можно сделать с использованием более сложной обработки вывода, но мы исправим эту проблему позже уже в тесте на Python.

Чтобы переписать этот тест на Python, нам нужно уметь как минимум две вещи: получить код возврата и вывод команды. Для этого мы будем использовать модуль Python под названием **subprocess**.

🔥 Выполнить команду внутри операционной системы можно и при помощи команды os.system() модуля os. Однако этот способ считается устаревшим, небезопасным и менее гибким. Рекомендуется использовать библиотеку subprocess.

Нам понадобится команда **subprocess.run()**, имеющая следующий вид:

```
import subprocess

subprocess.run(args, *, stdin=None, input=None, stdout=None,
               stderr=None, capture_output=False, shell=False,
               cwd=None, timeout=None, check=False,
               encoding=None, errors=None, text=None,
               env=None, universal_newlines=None)
```

Функция **run()** модуля subprocess запускает команду/скрипт/программу с аргументами, ждёт завершения команды, а затем возвращает экземпляр CompletedProcess() с результатами работы.

Разберём параметры функции. Аргументы функции, за исключением args, не обязательны и, следовательно, задаются только ключевыми словами.

- **args** — запускаемая программа с аргументами,
- **stdin=None** — поток данных, отправляемых в процесс,
- **input=None** — поток данных, отправляемых в процесс,
- **stdout=None** — поток вывода программы,
- **stderr=None** — поток ошибок программы,
- **capture_output=False** — захват вывода stdout и stderr,
- **shell=False** — используйте True, если программа и её аргументы представлены как одна строка,
- **cwd=None** — путь к рабочему каталогу запускаемой программы,
- **timeout=None** — максимально допустимое время выполнения запущенной программы,
- **check=False** — вызывает исключение при завершении запущенной программы с ошибками,
- **encoding=None** — кодировка, если input — строка,
- **errors=None** — обработчик ошибок кодировки,
- **text=None** — текстовый режим для stdin, stdout и stderr,
- **env=None** — переменные среды для нового процесса,
- **universal_newlines=None** — то же, что и параметр text.

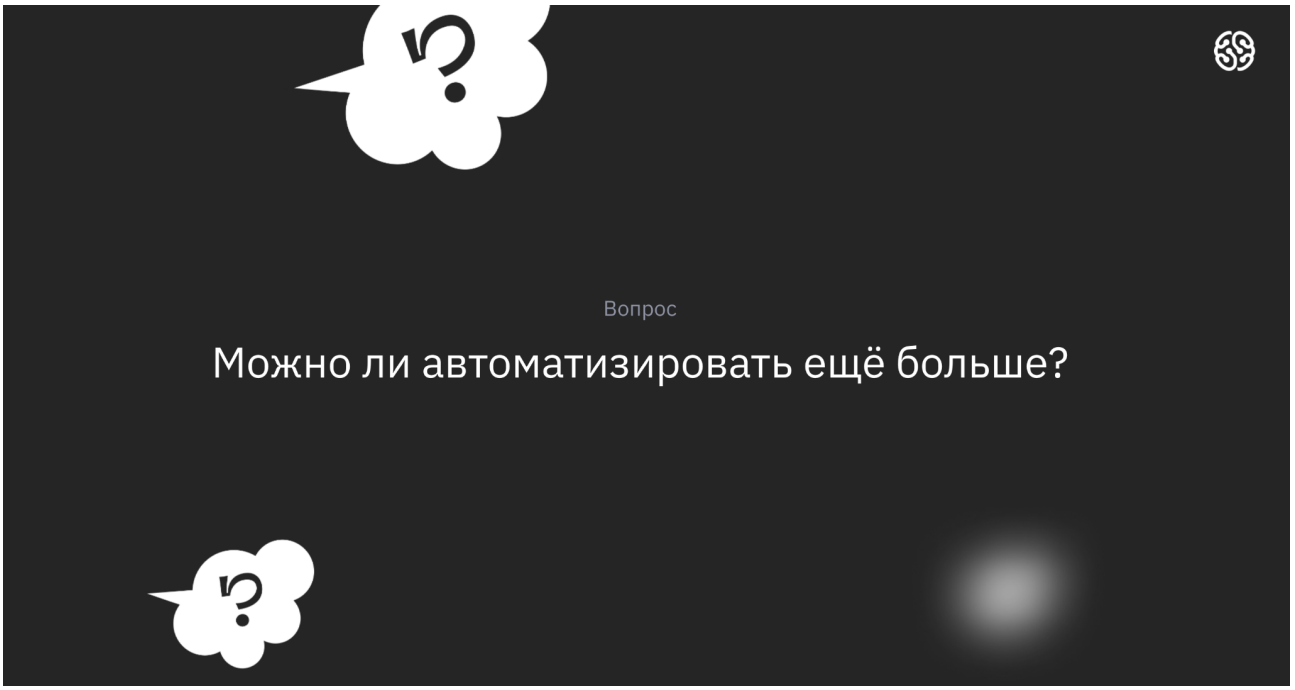
Мы будем использовать эту функцию для получения кода возврата команды и её вывода следующим образом:

```python
import subprocess
result = subprocess.run('ls /etc', shell=True, stdout=subprocess.PIPE, encoding='utf-8')
result.stdout
result.returncode
```

🔥 Если не указать параметр **encoding**, то результат получим как байтовую строку.

На семинаре мы напишем наш первый автотест на Python с использованием модуля subprocess.

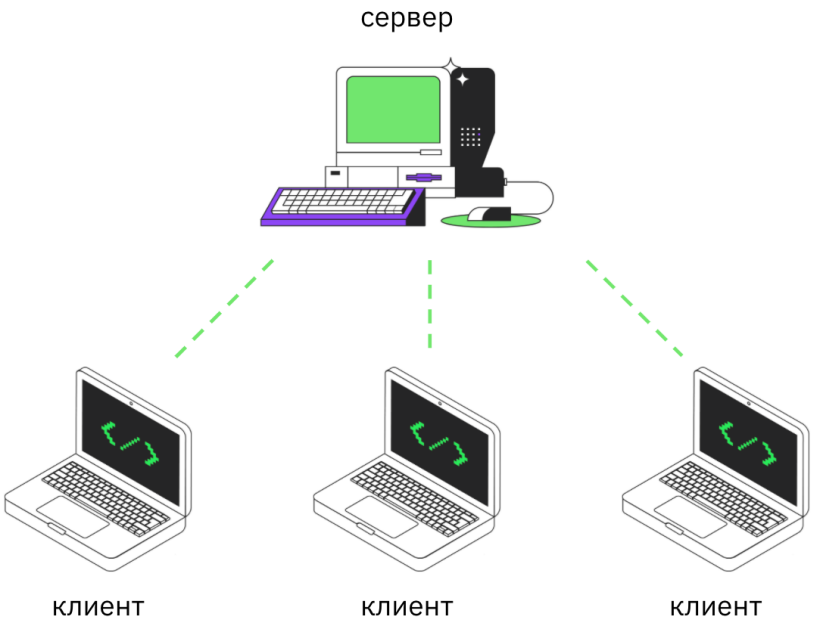
Теперь вернёмся к вопросу, который собирались обсудить в конце лекции.



Ответ — да. Если мы говорим про современное тестирование под Linux, то тестировщику приходится работать с огромным многообразием дистрибутивов Linux.

Поскольку ядро Linux распространяется как свободное программное обеспечение, каждый желающий может создать собственный дистрибутив Linux. Этих дистрибутивов очень много, даже если рассматривать только созданные крупными компаниями. Зачастую при разработке программного обеспечения заявляется поддержка нескольких десятков различных дистрибутивов. И на каждом из них нужно протестировать работу ПО.

Дальнейшее развитие идеи автотестов состоит в том, что тесты запускаются параллельно на нескольких машинах с разными операционными системами. Для этого нужно иметь сервер автотестов и машины-клиенты с различными дистрибутивами Linux. Сервер устанавливает на клиентах сборку, запускает на них автотесты, собирает результат и формирует сводный отчёт.



Домашнее задание

На семинаре мы перейдем к практике, для этого необходимо сделать следующие пункты:

- ☐ Скачать VirtualBox для вашей системы с Downloads – Oracle VM VirtualBox и установить его.

- [ ] Скачать образ из ubuntu.ova и импортировать его в VirtualBox.

Что можно почитать ещё?

1. Основы BASH. Часть 1 — здесь описаны основы bash.
2. Автоматизация функционального тестирования: что это такое и как это может быть полезно — а здесь автоматизация функционального тестирования.
3. Приручаем JMeter — статья про нагрузочное тестирование с использованием Apache Jmeter.

Используемая литература

1. Unit, Integration, and Functional Testing — про модульное, интеграционное и функциональное тестирование (на английском).
2. Subprocess management – документация модуля subprocess.
3. Топ 10 языков программирования 2022 | Digital Academy.