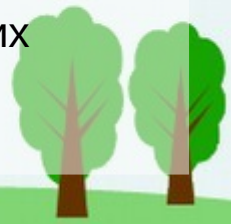
Разработка и инструменты

Часть 1

Инструменты

- Терминал — исполнитель текстовых команд
- Система контроля версий — хранит информацию об изменения содержания проекта, важен для командной разработки.
- Интерпретатор Python — выполнение кода на языке python
  - Отладчик кода — пошаговое выполнение кода с просмотром промежуточного состояния программы
  - Интерактивная среда Python — выполнение инструкций python в интерактивном режиме
- Пакетный менеджер pip — выполняет управление установкой необходимы библиотек (установка, обновление, удаление)
- Виртуальное окружение venv — облегчённая виртуальная среда
- Тестирование кода — проверка работы кода на некоторые условия (без гарантий работоспособности всей системы)
- Редактор кода — удобная программа для написания кода, может интегрировать все выше перечисленные, но дополняет их удобной работой с текстом программ: подсветкой синтаксиса, подсказками, форматирование и т. д.
- Онлайн редактор кода — как правило более ограниченное решение в плане работы с большими проектами, но позволяющие обойтись без локально установленных инструментов
- Линтер - отладчик или предупреждающее средство против гейзенбагов (плавающих ошибок)



- Терминал

Терминал

Программы Python могут запускаться в терминалах в различных командных оболочках:

- Bash (sh, ksh, zsh, …) - терминалы Linux, MacOS
- PowerShell, command line — терминалы Windows

Но для Windows есть подходы ввести linux-окружение...

GitBash vs WSL vs VirtualBox (для пользователей Windows)

1. GitBash
https://gitforwindows.org/

Git BASH
Git for Windows provides a BASH emulation used to run Git from the command line. *NIX users should feel right at home, as the BASH emulation behaves just like the "git" command in LINUX and UNIX environments.

2. Установка Linux в Windows с помощью WSL
https://learn.microsoft.com/ru-ru/windows/wsl/install

Разработчики могут одновременно получить доступ к возможности Windows и Linux на компьютере с Windows. Подсистема Windows для Linux (WSL) позволяет разработчикам устанавливать дистрибутив Linux (например, Ubuntu, OpenSUSE, Kali, Debian, Arch Linux и т. д.) и использовать приложения Linux

3. Если ресурсы у компьютера хорошие можно использовать VirtualBox
https://www.virtualbox.org/

Powerful open source virtualization

Команды системы

Самые частые команды:

Текущее местонахождение

```
$ pwd
```

Перемещение по папкам:
вверх на один уровень,
в корневую папку,
по абсолютному пути,
по относительному пути...

```
$ cd ..
$ cd /
$ cd /usr/bin
$ cd ./c_projects
$ cd ../py_projects
```

Просмотр содержимого директории,
стоит вводить опции «уточнители» и
маску-фильтр

```
$ ls
$ ls -la *.txt
```

Команды системы

Просто полезные приёмы работы с командной строкой:

```
$ date --help
Использование: date [ПАРАМЕТР]… [+ФОРМАТ]
       или:    date [-u|--utc|--universal] [ММДДччмм[[ВВ]ГГ][.сс]]
Выводит текущее время в заданном ФОРМАТЕ, или изменяет время в системе.
...


$ man date
$ man man
```

Почти у каждой команды есть короткая справка

У команд системы есть подробная справка с поиском и т. д. (man - доступ к системным справочным страницам)

```
DATE(1)                          User Commands                          DATE(1)

NAME
       date - print or set the system date and time

SYNOPSIS
       date [OPTION]... [+FORMAT]
       date [-u|--utc|--universal] [MMDDhhmm[[CC]YY][.ss]]

DESCRIPTION
       Display  the  current  time  in  the given FORMAT, or set the system
       date.

 Manual page date(1) line 1 (press h for help or q to quit)
```

Для страниц справки (man) откроется окно

Справка ;)

Команды системы

Просто полезные приёмы работы с командной строкой:

```
$ cd ~


$ touch README.md
$ > README.md
$ >> README.md


$ history
...
 1397  cd ~
 1398  ls -ls
 1399  ls -h
 1400  ls --help
 1401  cd Документы/
 1402  history


$ mc
```

Переход в домашний каталог

Создание файла

История команд, полезна, когда подзабыли команду со множеством параметров

Midnight Commander
https://midnight-commander.org

- Автозавершения команд с помощью клавиши Tab
- Двойное нажатие Tab — получение списка возможных продолжений команды или имени файла

Команды системы

Потоки и каналы:

```
 > для отправления в поток;
 < для получения из потока;
>> для добавления в поток;
 | канал, перенаправление вывода одной команды
   на следующую команду.
```

Пример использования:

```
$ diff <(ls dirA) <(ls dirB)

$ sudo yum -y update >> yum_update.log

$ pip install not_existsts > stdout.txt 2>stderr.txt

$ pip freeze > requirements.txt

$ history | tail
```

Команды системы

Полезные команды для работы с файлами:

```
$ cat requirements.txt

$ less /var/log/syslog
```

Для относительно коротких файлов, показать содержимое

считывает текст не полностью, а небольшими фрагментами

```
more — для длинных файлов (поэкранный вывод)

head – вывод первых строк файла

tail – вывод последних строк файла

tail –f — используется для просмотра растущего файла в окне интерактивного запуска кода
```

Windows vs Linux

	Windows	Linux
Кодировка по умолчанию	cp1251	UTF8
Учёт регистра букв в названиях файлов	нет	да
Окончание строк в текстовых файлах	окончание строки и перенос CR+LF ASCII 0x0D 0x0A	окончание строки LF ASCII 0x0A

Поэтому при открытии файла, рекомендуется явно указывать кодировку, обычно UTF8

Поэтому в названиях файлов рекомендуется использование snake_case

```
#!/bin/bash
sed -i 's/\r//g' $1
```

Файл для изменения окончания строк текстовых файлов с Windows, на Linux

```
#!/bin/bash
sed -i 's/$/\r/' $1
```

Файл для изменения окончания строк текстовых файлов с Linux, на Windows

- Система контроля версий

Система контроля версий

Система контроля версий (от англ. Version Control System, VCS) — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления 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.

Ежедневный цикл работы:

- Обновление рабочей копии
- Модификация проекта
- Фиксация изменений

Локальная копия разработчика стареет, что повышает риск возникновения конфликтных изменений

Локальные изменения

Завершив очередной этап работы над заданием, разработчик фиксирует свои изменения, передавая их на сервер

https://ru.wikipedia.org/wiki/Система_управления_версиями

Система контроля версий, Git
Обновление рабочей копии

Скачивает всю историю из удалённого репозитория

Вносит изменения из ветки удалённого репозитория в текущую ветку локального репозитория

```
$ git fetch [удалённый репозиторий]

$ git merge [удалённый репозиторий]/[ветка]

$ git pull
```

Загружает историю из удалённого репозитория и объединяет её с локальной. pull = fetch + merge

Шпаргалка по git
https://training.github.com/downloads/ru/github-git-cheat-sheet/

Система контроля версий, Git
Обновление рабочей копии

История коммитов для текущей ветки

История изменений конкретного файла, включая его переименование

```
$ git log

$ git log --follow [файл]


$ git diff [первая ветка]...[вторая ветка]


$ git show [коммит]
```

Показывает разницу между содержанием коммитов двух веток

Выводит информацию и показывает изменения в выбранном коммите

Шпаргалка по git
https://training.github.com/downloads/ru/github-git-cheat-sheet/

Система контроля версий, Git
Модификация проекта

Проверка статуса изменений

Различия по внесённым изменениям в ещё не проиндексированных файлах

Варианты добавления изменений для фиксации (add)

Часто некоторые руководства говорят, что добавить все файлы, но это не так

```
$ git status

$ git diff


$ git add [файл]

$ git add .

$ git add *.py


$ git commit –m "[сообщение с описанием]"
```

Фиксация изменений

Шпаргалка по git
https://training.github.com/downloads/ru/github-git-cheat-sheet/

Система контроля версий, Git
Модификация проекта

Удаляет конкретный файл из рабочей директории и индексирует его удаление

Убирает конкретный файл из контроля версий, но физически оставляет его на своём месте

```
$ git rm [файл]

$ git rm --cached [файл]

$ git mv [оригинальный файл] [новое имя]
```

Перемещает и переименовывает указанный файл, сразу индексируя его для последующего коммита

Отменяет все коммиты после заданного, оставляя все изменения в рабочей директории

```
$ git reset [коммит]

$ git reset --hard [коммит]
```

Сбрасывает всю историю вместе с состоянием рабочей директории до указанного коммита.

Шпаргалка по git
https://training.github.com/downloads/ru/github-git-cheat-sheet/

Система контроля версий, Git
Модификация проекта

Многие команды требуют понимания! (ну чтобы проблем не было), например:

Как работает данная команда?

```
$ git commit --amend -m "Текст комментария"
```

Шпаргалка по git
https://training.github.com/downloads/ru/github-git-cheat-sheet/

Система контроля версий, Git
Модификация проекта

Многие команды требуют понимания! (ну чтобы проблем не было), например:

Как работает данная команда?

```
$ git commit --amend -m "Текст комментария"
```

Не меняет сообщение коммита, а откатывает последний и делает новый, с новым комментарием.
Не стоит применять эту команду, если последний коммит уже удалённом на сервере

Шпаргалка по git
https://training.github.com/downloads/ru/github-git-cheat-sheet/

Система контроля версий, Git
Фиксация изменений

Загружает все изменения локальной ветки в удалённый репозиторий

```
$ git push [удалённый репозиторий] [ветка]
```

Голый/Чистый/bare репозиторий в Git — это репозиторий, который не имеет рабочего каталога. Обычно используется в качестве удалённого репозитория

Каталог .git содержит все данные по изменениям.
По факту являясь Базой Данных!

Тогда Рабочим каталогом можно считать выборку из базы данных плюс, возможно, не зафиксированные изменения.

Шпаргалка по git
https://training.github.com/downloads/ru/github-git-cheat-sheet/

Система контроля версий, Git

Файл **.gitignore**

A collection of .gitignore templates

В частности для проектов на python

Для ленивых:
https://github.com/github/gitignore/tree/main
https://github.com/github/gitignore/blob/main/Python.gitignore

Но лучше явно игнорировать то, что есть у разработчика локально в проекте, но это не нужно в основном репозитории. Например:

- файлы БД (SQLite)
- venv
- IDE конфиг
- *.pyc
- секреты

Однако должен быть файл README.rst, где прописана версия питона и зависимостей

Вместо самой базы данных стоит в репозиторий положить набор данных, которым можно заполнить тестовую базу данных

Лучше положить конфиг как пример

Байт-код

Если это отдельный файл, но в коде тоже не стоит прописывать секреты, даже на ранних этапах разработки

Система контроля версий, Git

Git for Windows: работа с параметром core.autocrlf
https://git-scm.com/book/ru/v2/Настройка-Git-Конфигурация-Git

Windows при создании файлов использует для обозначения переноса строки два символа «возврат каретки» и «перевод строки», в то время как Mac и Linux используют только один — «перевод строки». Это незначительный, но невероятно раздражающий факт кроссплатформенной работы; большинство редакторов в Windows молча заменяют переносы строк вида LF на CRLF или вставляют оба символа, когда пользователь нажимает клавишу ввод.

Git может автоматически конвертировать переносы строк CRLF в LF при добавлении файла в индекс и наоборот — при извлечении кода.

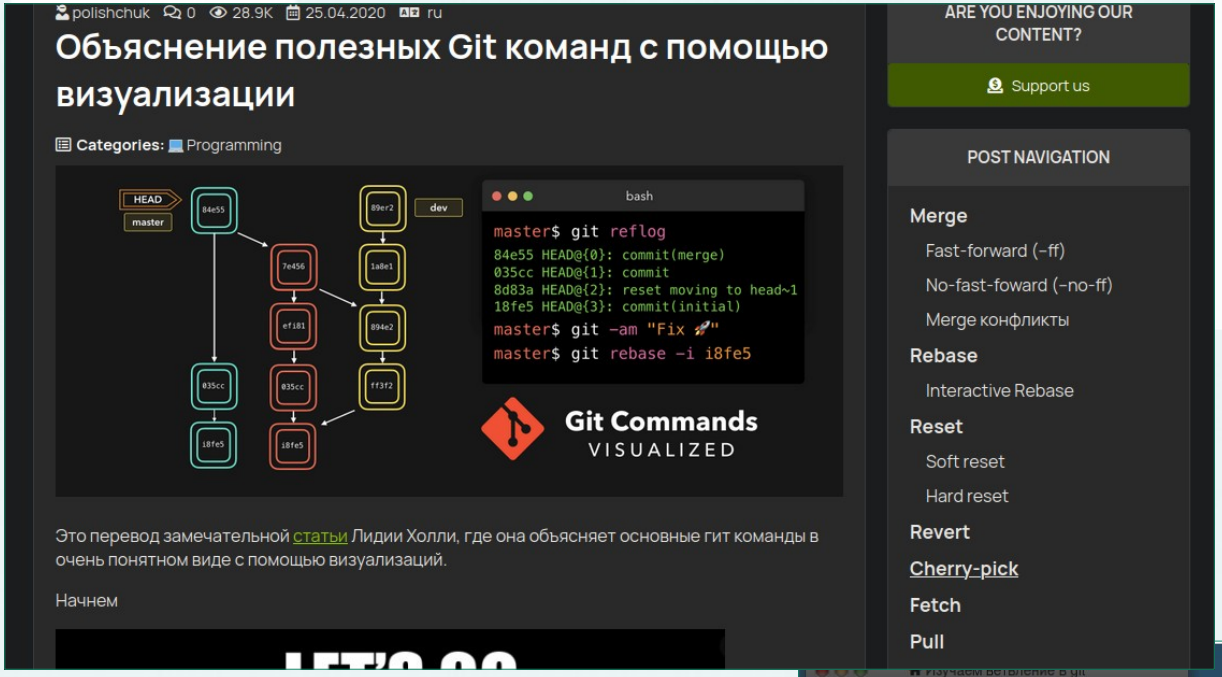
```
$ git config --global core.autocrlf true

$ git config --global core.autocrlf input

$ git config --global core.autocrlf false
```

Если у вас Windows, то установите значение true — при извлечении кода LF окончания строк будут преобразовываться в CRLF

Linux или Mac, если файл с CRLF окончаниями строк случайно попал в репозиторий, то Git может его исправить

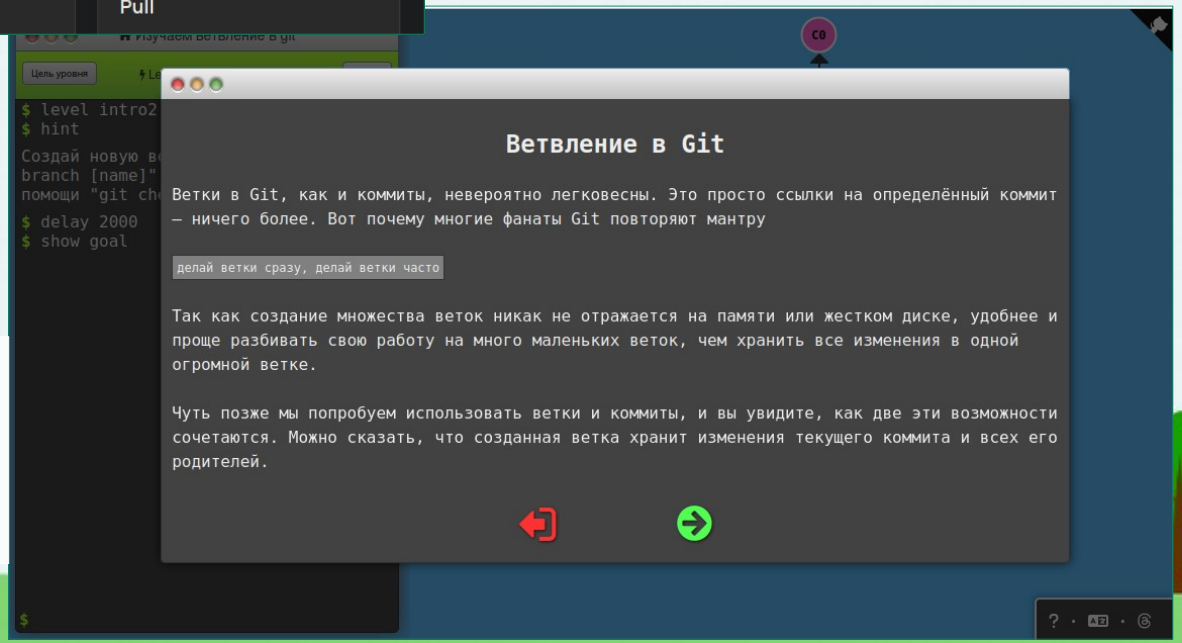
Windows, вы можете отключить описанный функционал задав значение false, сохраняя при этом CR символы в репозитории

Система контроля версий, Git



Объяснение полезных Git команд с помощью визуализации

https://bool.dev/blog/detail/vizualizatsiya-poleznykh-git-komand



LearnGitBranching!

https://learngitbranching.js.org/?locale=ru_RU

- Интерпретатор Python

Интерпретатор Python

Python способен функционировать в 2-х режимах:

- Интерактивный;
- Пакетный.

```
$ python3
Python 3.10.12 (main, Sep 11 2024, 15:47:36) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 123
>>> a
123
>>> print(a)
123
>>>
```

Интерактивный режим,
запущен в консоли bash

Ввод символа конца файла (Control-D в Unix, Control-Z в Windows) в главном приглашении приводит к выходу интерпретатора с нулевым статусом выхода. Если это не сработает, вы можете выйти из интерпретатора, введя следующую команду: quit()

```
$ python3 hello.py
Привет! Как дела?
```

Пакетный режим.
Выполняется код в файле.

Интерпретатор Python, отладка

```
 1  from random import randint
 2  import pdb
 3
 4  def summ(a, b):
 5      return a + b
 6
 7  x = randint(1, 1000)
 8  y = randint(1, 1000)
 9  pdb.set_trace()
10  print(sum(x, y))
11
```

Отладчик python-программ

Остановить программу здесь (breakpoint)

```
$ python3 pdb_ex.py
> /home/.../pdb_ex.py(10)<module>()
-> print(summ(x, y))
(Pdb) x
458
(Pdb) y
72
(Pdb) x + y
530
(Pdb)
```

Интерпретатор Python, отладка

```
 1  from random import randint
 2  # import pdb
 3
 4  def summ(a, b):
 5      return a + b
 6
 7  x = randint(1, 1000)
 8  y = randint(1, 1000)
 9  # pdb.set_trace()
10  print(summ(x, y))
```

Закомментировали

Запустили через команду

Остановились автоматом на первой строке

```
$ python -m pdb pdb_ex.py
> /home/.../pdb_ex.py(1)<module>()
-> from random import randint
(Pdb) b 5
Breakpoint 1 at /home/.../pdb_ex.py:5
(Pdb) c
> /home/.../pdb_ex.py(5)summ()
-> return a + b
(Pdb) locals()
{'a': 506, 'b': 408}
(Pdb) c
914
The program finished and will be restarted
```

Установили breakpoint на 5 строке

Продолжили выполнение кода

Остановились на точке останова (breakpoint)

Посмотрели локальные переменные

Интерпретатор Python, отладка

Встроенная справка

```
(Pdb) h

Documented commands (type help <topic>):
========================================
EOF     c          d        h         list      q        rv       undisplay
a       cl         debug    help      ll        quit     s        unt
alias   clear      disable  ignore    longlist  r        source   until
args    commands   display  interact  n         restart  step     up
b       condition  down     j         next      return   tbreak   w
break   cont       enable   jump      p         retval   u        whatis
bt      continue   exit     l         pp        run      unalias  where

Miscellaneous help topics:
==========================
exec  pdb

(Pdb)
```

Аналогичный инструмент существует
для Jupyter Notebook: ipdb

Debugging in Google Colab notebook
https://zohaib.me/debugging-in-google-collab-notebook/

- Пакетный менеджер pip

Пакетный менеджер pip

Версия

```
$ pip --version
pip 24.2 from /home/.../.venv/lib/python3.12/site-packages/pip (python 3.12)
```

Обновление, первый вариант более предпочтителен (безопасен)

```
$ python -m pip install -U pip
$ pip install --upgrade pip
Requirement already satisfied: pip in ./.venv/lib/python3.12/site-packages (24.3.1)
```

Установка наиболее свежей подходящей версии, установка точно указанной версии и установка пакетов, перечисленных в файле

```
$ pip install requests
$ pip install Django==4.2.16
$ pip install -r requirements.txt
...
```

Вывод списка установленных пакетов

```
$ pip list
Package  Version
-------- -------
asgiref  3.8.1
Django   4.2.16
pip      24.2
sqlparse 0.5.1
```

Вывод списка установленных пакетов в другом формате

```
$ pip freeze
asgiref==3.8.1
Django==4.2.16
sqlparse==0.5.1
```

«заморозка» пакетов в файле

```
$ pip freeze > requirements.txt
```

Пакетный менеджер pip

Получить список устаревших пакетов

```
$ pip list -outdated
Package Version Latest Type
------- ------- ------ -----
Django  4.2.16  5.1.3  wheel
pip     24.2    24.3.1 wheel

[notice] A new release of pip is available: 24.2 -> 24.3.1
[notice] To update, run: pip install --upgrade pip


$ pip uninstall requests
$ pip uninstall requests -y
$ pip uninstall -y -r <(pip freeze)
Found existing installation: certifi 2024.8.30
Uninstalling certifi-2024.8.30:
  Successfully uninstalled certifi-2024.8.30
...
```

Удалить конкретный пакет,
удалить пакет без подтверждения,
удалить все пакеты...

- Виртуальное окружение venv

Виртуальное окружение venv

С версии 3.3 — это стандартный модуль python, не нужно использовать сторонние решения

- Модуль venv поддерживает создание облегченных «виртуальных сред», каждая из которых имеет собственный независимый набор пакетов Python.
- При использовании из виртуальной среды обычные инструменты установки, такие как pip, установят пакеты Python в виртуальную среду без необходимости явно указывать им делать это.

Виртуальное окружение venv

- Виртуальная среда (помимо прочего):
  - Используется для хранения определенного интерпретатора Python, а также библиотек программного обеспечения и двоичных файлов, которые необходимы для поддержки проекта (библиотеки или приложения). По умолчанию они изолированы от программного обеспечения в других виртуальных средах и интерпретаторов и библиотек Python, установленных в операционной системе.
  - Содержится в каталоге, традиционно называемом .venv или venv в каталоге проекта.
  - Рассматривается как одноразовое — его должно быть просто удалить и создать заново с нуля. Вы не помещаете никакой код проекта в среду.
  - Не рассматривается как перемещаемое или копируемое — вы просто воссоздаете ту же среду в целевом месте.

https://docs.python.org/3/library/venv.html

Легко иметь проекты для разных версий python и специфичных версий библиотек

Если стандартно назовём, то IDE скорее всего автоматически его распознает

Это то, что не попадает в репозиторий, это пересоздаваемая директория на основе необходимой версии python и файла requirements.txt

При перемещении проекта в другую директорию, требуется пересоздание

Виртуальное окружение venv

```
$ python -m venv /path/to/new/virtual/environment


$ cd /path/to/project
$ python -m venv .venv
```

Создание виртуального окружения, общий вид

Создание виртуального окружения, типичный сценарий

Далее виртуальную среду следует «активировать» с помощью скрипта.

Платформа	shell	Команда активации виртуального окружения
POSIX	bash/zsh	$ source <venv>/bin/activate
	fish	$ source <venv>/bin/activate.fish
	csh/tcsh	$ source <venv>/bin/activate.csh
	pwsh	$ <venv>/bin/Activate.ps1
Windows	cmd.exe	C:\> <venv>\Scripts\activate.bat
	PowerSell	PS C:\> <venv>\Scripts\Activate.ps1

Активация добавит каталог в ваш PATH, что приведёт к запуску python с помощью интерпретатора Python виртуального окружения, что даст возможность запускать установленные скрипты без необходимости использовать их полный путь.

https://docs.python.org/3/library/venv.html

Команды системы + venv

```
$ python3 bash_it.py

$ bash_it.py
```

Обычно, в командной строке код запускаем так

Но можно и так

Для этого в файле нужна специальная первая строка.
Не обычный комментарий, а «шебанг»:
«#!» + «путь к интерпретатору»

```
1  #!/usr/bin/env python
2  print(
3      'Привет! Я могу исполнится без ввода полной команды: \n'
4      '«python3 <имя файла>»\n'
5      '1. В системе должен быть установлен интерпретатор python.\n'
6      '2. Пользователю разрешено исполнить файл.\n')
7
```

```
$ chmod u+x ./bash_it.py
```

Права на выполнение файла изменяем командой

https://ru.wikipedia.org/wiki/Шебанг_(Unix)

- Тестирование кода

Тестирование кода

- Ожидаемый результат (ОР) — то, как сервис должен работать согласно требованиям.
  - Например, пункт чек-листа «При выборе (нажатии на dropbox) ингредиента появится список ингредиентов».
  - При делении на 0 появится «Traceback ... ZeroDivisionError: division by zero»
- Фактический результат (ФР) — то, как сервис работает на самом деле.

Тесты проверяют, что ожидаемый результат совпадает с фактическим

Тестирование, python

- Doc Test (стандартная)
- Unittest (стандартная)
- Pytest (наиболее популярная)

+ инструкции assert

Тестирование, python, assert

7.3. The `assert` statement

Assert statements are a convenient way to insert debugging assertions into a program:

```
assert_stmt ::=  "assert" expression ["," expression]
```

The simple form, `assert expression`, is equivalent to

```
if __debug__:
    if not expression: raise AssertionError
```

The extended form, `assert expression1, expression2`, is equivalent to

```
if __debug__:
    if not expression1: raise AssertionError(expression2)
```

Можно отключить инструкции assert.
В текущей реализации встроенная переменная __debug__ имеет значение True при обычных обстоятельствах и False при запросе оптимизации (параметр командной строки -O)

https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement

Тестирование, python, assert

Пример использования assert:

```
1  hello_str = 'Привет! Как дела?'
2  print(hello_str)
3  assert hello_str == 'Привет! Мир!', (
4      'Строка не равна ожидаемой! '
5      'Тест не пройден.'
6  )
7
```

Длинный текст ошибки следует оформлять так

Тут будет неравенство строк, соответственно произойдёт выброс исключения AssertionError

Тестирование, python, assert

```
$ python hello_assert.py
Привет! Как дела?
Traceback (most recent call last):
  File "/home/.../hello_assert.py", line 4, in <module>
    assert hello_str == 'Привет! Мир!', (
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
AssertionError: Строка не равна ожидаемой! Тест не пройден.



$ python -O hello_assert.py
Привет! Как дела?
```

Тест не пройден из-за разницы ожидаемой строки и реально выводимой

Благодаря опции -O при запуске, инструкции assert отключены

https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement

Тестирование, python, doctest

```
1  def some_func(n):
2      """Положительный 'удвоятель' не любит число 13
3      и думает о Смысле жизни при числе 42.
4      >>> [some_func(n) for n in range(-3, 3, 1)]
5      [6, 4, 2, 0, 2, 4]
6      >>> some_func(3)
7      6
8      >>> some_func(42)
9      84
10     >>> some_func(13)
11     Traceback (most recent call last):
12         ...
13     ValueError: 13
14     """
15     # рукотворный bug – не удваивает
16     if n == 42:
17         n = 24
18     # не bug – не любит число 13
19     if n == 13:
20         raise ValueError('13')
21     return n * 2 if n >= 0 else -n * 2
22
23
24 if __name__ == "__main__":
25     import doctest
26     print('Используй -v для вывода результатов тестов.')
27     doctest.testmod()
```

«>>>» команда

Ожидаемый результат

Ожидаем удвоение

Ожидаем исключение

Ожидаемый результат для исключения

Думать о Смысле жизни
!=
правильно не работать

Задокументировано исключение для работы алгоритма

Тестирование, python, doctest

Запуск файла

```
$ python doctest_sample.py
Используй -v для вывода результатов тестов.
**********************************************************************
File
"/home/serzhook/00_mirea/assembly_testing_verivication_soft/practic/
for_lection/lection1/doctest_sample.py", line 9, in __main__.some_func
Failed example:
    some_func(42)
Expected:
    84
Got:
    48
**********************************************************************
1 items had failures:
   1 of   4 in __main__.some_func
***Test Failed*** 1 failures.
```

Тестирование, python, doctest

```
$ python -m doctest -v doctest_sample.py
Trying:
    [some_func(n) for n in range(-3, 3, 1)]
Expecting:
    [6, 4, 2, 0, 2, 4]
ok
Trying:
    some_func(3)
Expecting:
    6
ok
Trying:
    some_func(42)
Expecting:
    84
**********************************************************************
File "/home/.../doctest_sample.py", line 9, in doctest_sample.some_func
Failed example:
    some_func(42)
Expected:
    84
Got:
    48
Trying:
    some_func(13)
Expecting:
    Traceback (most recent call last):
        ...
    ValueError: 13
ok
1 items had no tests:
    doctest_sample
**********************************************************************
1 items had failures:
   1 of   4 in doctest_sample.some_func
4 tests in 2 items.
3 passed and 1 failed.
***Test Failed*** 1 failures.
```

Запуск через модуль doctest
Плюс «говорливый»режим

Выведена информация и по тестам, которые прошли

Тестирование кода, Unittest

- Тест — это специальный класс, унаследован от unittest.TestCase (обычно каждый класс создают в отдельном файле)
- Методы класса, начинающиеся с «test_» - это отдельные тесты (один тест — одна проверка).
- Вместо assert для проверки результата используются специальные методы, начинающиеся с «assert» (например, assertEqual(a, b), assertIsInstance(a, b), assertIsNotNone(x), assertRaises(exc, fun, *args, **kwds))

Тестирование кода, Unittest

```
1  """Пример unittest. Тестируем площадь прямоугольника."""
2  # импортируем саму библиотеку
3  import unittest
4  # импортируем функцию, которую будем тестировать
5  from rectangle import calc_perimeter
6
7  class TestRectangle(unittest.TestCase):
8      """Тест модуля rectangle."""
9      def test_calc_perimeter(self):
10         """Правильно ли считается периметр."""
11         self.assertEqual(calc_perimeter(2, 3), 10)
12
13 if __name__ == "__main__":
14     unittest.main()  # Запуск тестов
```

Тестирование кода, Unittest

Запускаем файл из командной строки

```
$ python test_rectangle.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
$ python test_rectangle.py -v
test_calc_perimeter (__main__.TestRectangle.test_calc_perimeter)
Правильно ли считается периметр. ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Тестирование кода, Unittest

Запускаем файл из командной строки (здесь важно, что файлы с тестами начинались с "test")

```
$ python -m unittest
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
$ python -m unittest -v
test_calc_perimeter (test_rectangle.TestRectangle.test_calc_perimeter)
Правильно ли считается периметр. ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Тестирование кода, Unittest

Если файлов тестов больше чем один:

```
tests/
├── test_one.py
├── test_two.py
└── test_three.py
```

Мы можем запустить тесты выборочно:

```
python -m unittest           # Запуск всех тестов (из всех трёх фалов).
python -m unittest test_one  # Запуск тестов одного файла.
python -m unittest test_one test_two   # Запуск тестов выбранных файлов.
python -m unittest test_one.TestClass  # Запуск отдельного класса с тестами.
python -m unittest test_one.TestClass.test_method  # Запуск отдельного теста.
```

Тестирование кода, Unittest

Команда, достойная отдельного слайда! СПРАВКА.

```
$ python -m unittest -h

usage: python -m unittest [-h] [-v] [-q] [--locals] [--durations N] [-f] [-c] [-b] [-k
TESTNAMEPATTERNS] [tests ...]


positional arguments:

  tests                 a list of any number of test modules, classes and test methods.


options:

  -h, --help            show this help message and exit

  -v, --verbose         Verbose output

  -q, --quiet           Quiet output

  --locals              Show local variables in tracebacks

  --durations N         Show the N slowest test cases (N=0 for all)

  -f, --failfast        Stop on first fail or error

.............
```

Тестирование кода, Unittest

Прошли ли тесты без ошибки?

run_test — это специально написанный bash-скрипт, который может понять статус завершения тестов

```
$ ./run_test "python -m unittest"

======================================================================
FAIL: test_calc_perimeter (test_rectangle.TestRectangle.test_calc_perimeter)
Правильно ли считается периметр.
----------------------------------------------------------------------
Traceback (most recent call last):
  File ".../test_rectangle.py", line 15, in test_calc_perimeter
    self.assertEqual(calc_perimeter(2, 3), 101)
AssertionError: 10 != 101


----------------------------------------------------------------------
Ran 1 test in 0.000s


FAILED (failures=1)


Были ошибки в тестах
```

Тестирование кода, Unittest

Прошли ли тесты без ошибки?
Содержимое файла run_test:

Исполнить с помощью bash

```
#!/usr/bin/env bash
$1
exit_code=$?
echo $exit_code
if [ $exit_code -ne 0 ]; then
    echo $'\n'"Были ошибки в тестах"$'\n'
fi
exit $exit_code
```

В параметр $1 передаётся строка: «python -m unittest»

В $? будет статус завершения программы, который присвоим переменной exit_code

Сравним значение статуса завершения с 0

== 0 — успешное завершение
!= 0 — были какие-то ошибки (стоит изучить справку)

Тестирование кода, Unittest

Проверка нескольких схожих входных параметров (критических значений)

```
 1  def test_calc_perimeter_more(self):
 2      """Правильно ли считается периметр на нескольких данных."""
 3      values_results = (
 4          ((2, 2), 8),
 5          ((1, 2), 6),
 6          ((2, 1), 6),
 7      )
 8      for params, expected in values_results:
 9          with self.subTest():
10              result = calc_perimeter(*params)
11              self.assertEqual(result, expected)
```

Благодаря контекст-менеджеру subTest(), будут проверены все варианты входных данных. Без него, если бы, например, первый assertEqual() показал не верный результат, то дальше бы проверки прервались

Тестирование кода, паттерн AAA

Паттерн AAA (Arrange-Act-Assert)

Arrange - Секция подготовки. Она может состоять из нескольких строк. В этой секции мы определяем все необходимые элементы для начала тестирования: создаем объекты, стабы, моки и т.д.

Act - Секция действия. Она всегда состоит из одной строки. Собственно это то самое выражение, которое мы хотим проверить. Если выражение получается в несколько строк, то можно смело утверждать об утечке деталей имплементации, и как следствие хрупкости теста.

Assert - Секция проверки. Эта секция может состоять из нескольких утверждений. Однако если они разноплановые, то стоит задуматься о хорошем дизайне метода или системы под тестированием, возможно разбить большой тест, на более мелкие.

https://tenchat.ru/media/97620-pattern-aaa-arrangeactassert-i-yego-antipatterny

Тестирование кода, паттерн AAA

Множественные секции AAA. Один тест должен содержать по одному из каждой секции: одна подготовка, одно действие, одна проверка

Команды if в тестах. Тест должен представлять простую последовательность шагов без ветвлений.

Объединять позитивные и негативные сценарии, например если тест проверяет равенство, то проверять на неравенство следует в другом тесте.

Секция действий из нескольких строк, 2 и более. Это нарушение логической целостности, которое решается инкапсуляцией: несколько assert на один тест/подтест

Подготовка данных внутри теста.

https://tenchat.ru/media/97620-pattern-aaa-arrangeactassert-i-yego-antipatterny

Тестирование, Unittest, фикстуры

```
1  from circle import Circle
2  class TestCircle(unittest.TestCase):
3      """Тест модуля circle."""
4      def test_calc_perimeter(self):
5          """Правильно ли считается периметр."""
6          circle = Circle(3)
7          self.assertAlmostEqual(circle.calc_perimeter(), 18.8495, places=3)
8
9      def test_calc_perimeter_more(self):
10         """Правильно ли считается периметр на нескольких данных."""
11         radiuses = [-1, 0]
12         circle = Circle(3)
13         for radius in radiuses:
14             with self.subTest(value=radius):
15                 with self.assertRaises(ValueError):
16                     circle.radius = radius
```

Инструкция повторяется в каждом тесте

Тестирование, Unittest, фикстуры

Фикстуры – это не только специальные методы и функции, которые выполняются перед запуском теста и подготавливают для него исходные данные, такие как:

- содержимое базы данных,
- подготовленные файлы с определённым содержанием,
- программные объекты: словари/списки и т.п.

Но и функции и методы, которые после тестирования удаляют всю информацию, созданную для проведения тестов.

Тестирование, Unittest, фикстуры

`setUp()` – метод, который автоматически вызывается перед запуском каждого теста в классе.

`SetUpClass()` - метод, запускающийся однократно перед запуском начала тестирования класса

`TearDown()` - метод, который автоматически вызывается после запуска каждого теста в классе.

`TearDownClass()` - метод, запускающийся однократно после прохождения всего тестирования класса

https://docs.python.org/3/library/unittest.html

Небольшое, но важное, замечание из документации

Note

The order in which the various tests will be run is determined by sorting the test method names with respect to the built-in ordering for strings

Тестирование, Unittest, фикстуры

```
 1  class TestCircle2(unittest.TestCase):
 2      """Тест модуля circle."""
 3
 4      def setUp(self):
 5          self.circle = Circle(3)
 6
 7      def test_calc_perimeter(self):
 8          """Правильно ли считается периметр."""
 9          self.assertAlmostEqual(self.circle.calc_perimeter(), 18.8495, places=3)
10
11      def test_calc_perimeter_more(self):
12          """Правильно ли считается периметр на нескольких данных."""
13          radiuses = [-1, 0]
14          for radius in radiuses:
15              with self.subTest(value=radius):
16                  with self.assertRaises(ValueError):
17                      self.circle.radius = radius
```

Важно. Перед каждым тестом — это новый экземпляр

Тестирование, Unittest, покрытие

```
$ pip install coverage
$ python -m coverage run -m unittest
.........
----------------------------------------------------------
Ran 9 tests in 0.001s

OK
python -m coverage report -m
Name                 Stmts   Miss  Cover   Missing
------------------------------------------------
rectangle.py             4      0   100%
test_rectangle.py       16      1    94%   35
------------------------------------------------
TOTAL                   20      1    95%
```

Для замера нужно установить библиотеку coverage

Не все строки покрыты тестами

Тестирование, Unittest, покрытие

Coverage for **test_rectangle.py**: 94%

16 statements | 15 run | 1 missing | 0 excluded

« prev ^ index » next coverage.py v7.6.1, created at 2024-09-19 21:01 +0300

```
1  """Пример unittest. Тестируем площадь прямоугольника
2
3  # импортируем саму библиотеку
4  import unittest
5
6  # импортируем функцию, которую будем тестировать
7  from rectangle import calc_perimeter
8
9
10 class TestRectangle(unittest.TestCase):
11     """Тест модуля rectangle."""
12
13     def test_calc_perimeter(self):
14         """Правильно ли считается периметр."""
15         self.assertEqual(calc_perimeter(2, 3), 10)
```

Coverage report: 95%

Files | Functions | Classes

coverage.py v7.6.1, created at 2024-09-19 21:01 +0300

File ▲	statements	missing	excluded	coverage
rectangle.py	4	0	0	100%
test_rectangle.py	16	1	0	94%
Total	**20**	**1**	**0**	**95%**

coverage.py v7.6.1, created at 2024-09-19 21:01 +0300

Даёт ли гарантию, что тесты хорошо проверили код, если покрытие будет 100%?

```
27             ((2, 1), 6),
28         )
29         for params, expected in values_results:
30             with self.subTest(value=params):
31                 result = calc_perimeter(*params)
32                 self.assertEqual(result, expected)
33
34 if __name__ == "__main__":
35     unittest.main()  # Запуск тестов
```

« prev ^ index » next coverage.py v7.6.1, created at 2024-09-19 21:01 +0300

Эта строка не выполняется при импорте

- Редактор кода

- Онлайн редактор кода

- Линтер

Вопросы