**ТРЕБОВАНИЯ К РАЗРАБОТКЕ КОДА НА PYTHON**

**PEP 8 и лучшие практики code-style**

<https://pep8.org/>

Код необходимо разрабатывать таким образом:

* чтобы его можно было отредактировать;
* чтобы другой разработчик мог легко в нём разобраться, не свернув шею;
* чтобы была возможность отладить прямо на сервере;
* чтобы не стыдно было показывать другим.

Здесь перечислены основные ограничения по code-style.

1. **Именование**
   1. Длина значимой части названия не должна превышать 10 символов. Значимой частью названия является, к примеру, “Student” в StudentBuilder, “AuthService” в AuthServiceError.
   2. Имена должны отражать область ответственности переменной внутри пространства имён, функции, класса, метода. Например, название функции “make\_file” некорректно, поскольку может быть понято необнозначно.
   3. Любой класс должен называться camel-case. Student, StudentBuilder, StudentMetaclass.
   4. Переменная должна называться как объект данных, результат преобразования, но не как действие.
      1. Локальная переменная должна называться snake-case:
      2. Глобальная переменная должна называться screaming-snake case, не дублируя имя стандартного пакета.
   5. Наследник класса Exception должен называться только camel-case, однако в конце должен встречаться постфикс Exception/Error.
2. **Отступы**
   1. Внутри метода/функции/скрипта/класса между логическими блоками необходимо ставить один enter.
   2. Внутри модуля между логическими блоками необходимо ставить два enter.
   3. Подпространство имëн отделяется четырьмя пробелами.
3. **Импорт пакетов и модулей** должен происходить в порядке встроенные, установленные, свои.
4. **Длина строки** не должна превышать 79 символов.
   1. Функция/метод с количеством переменных больше 5, включая self, должна быть оформлена соответствующем образом:. Еë вызов должен осуществляться с позиционными аргументами.
   2. Большие однострочники должны быть разделены на элементарные.
5. **Комментарии** должны позволять оценить область ответственности объекта
   1. **docstring**
      1. В начале функции/метода должна присутствовать docstring.
      2. В начале модуля должна присутствовать docstring.
      3. В начале класса должна присутствовать docstring.
   2. однострочными комментариями желательно пренебрегать, если они не являются частью документации.

**«Сущности не должны умножаться сверх необходимого»**

<https://github.com/faif/python-patterns>

1. Класс имеет право на существование только в тех случаях, когда:
   1. Является наследником класса предка.
   2. Является метаклассом.
   3. Содержит специфические только для него атрибуты.
   4. Является пространством имëн.
   5. Является шаблоном для класса-наследника.
   6. Преобразуется метаклассом/декоратором в более сложный объект.
   7. Если является датаклассом.
2. Функция имеет право на существование, когда:
   1. Содержит алгоритм более чем одну элементарную операцию. Излишняя вложенность вызовов замедляет python. Такой объект функции лучше заменить на property.
   2. Является сложным однострочником.
   3. Является локальной функцией для выполнения замыкания.
   4. Не использует внутри тела глобальные переменные.
3. Глобальная переменная имеет право на своё существование тогда и только тогда, когда является конфигурацией.
4. Логическая операция с объектом имеет право на существование, когда еë нельзя заменить на аналогичную с использованием встроенных объектов python. Например, not Entity is None не имеет смысла, поскольку есть Entity is not None – отрицание Entity медленнее, чем отрицание None.
5. Значение по умолчанию допустимо только в том случае, если не является массивом. Питон постоянно инициализирует новые спискоаые переменные, чтобы избежать переполнение памяти, необходимо использовать if var is None: var = [].
6. Список по возможности необходимо менять на tuple/set/generator.

**Работа с файловой системой**

1. Необходимо использовать pathlib.Path, дабы приложение свободно могло запускаться в любом окружении: windows, macOS, linux.
2. При чтении файла необходимо обработать ошибку, когда файла может не оказаться на сервере. С этим может помочь метод pathlib.Path.exists.
3. Рекомендуется использовать относительные пути.

**Менеджер пакетов**

<https://pipenv.pypa.io/en/latest/>

pipenv представляет собой менеджер пакетов, который управляет зависимостями, в том числе решает конфликты, а также создает виртуальное окружение. Кроме этого pipenv умеет:

* искать необходимый интерпретатор python;
* подгружать переменные среды из .env;
* запускать вспомогательные скрипты для разработки;
* проверять наличие уязвимостей в пакетах,

однако нас больше интересует pipenv как средство разрешения конфликтов зависимостей между пакетами, поскольку данная проблема является, пожалуй, самой критичной при разработке. pip и conda имеют такую возможность, однако являются менее строгими по отношению к конфликтам. pipenv предоставляет великолепную поддержку для управления прямыми зависимостями, при этом хранит полный список версий зависимостей, которые гарантированно работают правильно, и всегда поддерживает среду в актуальном состоянии.

**Установка**

Чтобы установить pip, необходимо выполнить:

| sudo apt install software-properties-common python-software-properties  sudo add-apt-repository ppa:pypa/ppa  sudo apt update  sudo apt install pipenv  python –m pip install ––user pipenv |
| --- |

Также возможна установка на конкретную версию:

pipenv install --python 3.6

Далее можно просмотреть версию пакетного менеджера с помощью:

pipenv version

Далее необходимо зайти в вашу IDE и сменить интерпретатор на pipenv. Обычно, после полноценной установки IDE автоматически обнаруживает pipenv, однако если этого не произошло, на сайте разработчиков сред для разработки можно найти информацию о том, как выполнить настройку. Для Pycharm: <https://www.jetbrains.com/help/pycharm/pipenv.html>, в другом случае достаточно выполнить pipenv shell и конфигурировать запуск.

При работе с юпитером, необходимо установить jupyter в pipenv:

| pipenv install jupyter  pipenv run jupyter notebook |
| --- |

**Использование**

После того, как пакетный менеджер pipenv был активирован в среде разработке, вы увидете файлы с именами Pipfile, Pipfile.lock.

Pipfile содержит имена установленных пакетов с разделами packages и dev-packages. В dev-packages обычно прописываются версии, которые не нужны разработчику сейчас. Чтобы установить с dev-packages при наличии Pipfile, запустите pipenv install -d или pipenv sync -d.

Установите всё, что раньше писалось в requirements:

pipenv install -r requirements.txt

Далее выполните:

pipenv requirements > requirements.txt

Загляните в файл и вы увидите все зависимости вашего проекта с указанием версий python.

**Вэндоринг**

После установки запустите

pip download -d ./path/to/dependencies -r requirements.txt

Аналогично работает команда:

pip wheel -r requirements.txt -w ./path/to/dependencies

У вас скачаются все пакеты, находящиеся в requirements.txt.

**Установка в Pipenv**

**Подготовка сервера pypi**

mkdir private\_pypi && cd private\_pypi

pipenv install

pipenv install pypiserver

mkdir packages

pipenv run pypi-server -p 8080 ./packages

pipfile

[[source]]

url = "http://hostname:8080/simple"

verify\_ssl = false

name = "pypi"

…

[packages]

wheel1 = {version="\*", index="logical\_name"}

wheel2 = {version="0.41.1", index="logical\_name"}

public\_project = "\*"

…

**Установка в pip**

pip install --no-index --find-links /path/to/deps/ -r requirements.txt

**ВСЕ ПАКЕТЫ ДОЛЖНЫ БЫТЬ СКАЧАНЫ И ДОСТУПНЫ ДЛЯ ДЕПЛОЯ РАЗРАБОТЧИКУ, КОТОРЫЙ ЗАНИМАЕТСЯ ДОСТАВКОЙ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ НА ПРОДУКТОВЫЙ СЕРВЕР!**