Типичная структура каталогов пакета Python

Сначала обратим внимание на структуру директорий проекта, на именование файлов и организацию модулей. Уныло смотрится проект, в структуре которого нет никакого порядка из-за того, что его папки и файлы просто расположены по алфавиту, в соответствии с правилами сортировки объектов в IDE.

Представим структуру некоторого проекта Python.

<project>/

├── <package>/

│ ├── \_\_init\_\_.py

│ ├── module.py

├── tests/

│ ├── \_\_init\_\_.py

│ ├── test\_module.py

├── setup.py

* **project/** - это корневой каталог вашего проекта на Python. Он содержит все файлы и подкаталоги, связанные с вашим проектом.
* **package/** - это подкаталог, содержащий код для пакета Python. Пакет - это набор модулей, которые можно импортировать и использовать в других программах на Python.
* **\_\_init\_\_.py** - это специальный файл, который сообщает Python, что этот каталог следует рассматривать как пакет.
* **module.py** - это модуль Python, содержащий код, который может использоваться другими частями вашего проекта.
* **tests/** - это подкаталог, содержащий тестовый код для вашего проекта.
* **test\_module.py** — модуль Python, содержащий код для выполнения тестирования модуля module.py. Тестовые модули должны содержать в своём названии слово test (допускается писать в начале или в конце) и название тестируемого модуля.
* **setup.py** это скрипт на Python, который используется для установки вашего проекта. Он определяет метаданные о проекте, такие как его имя, версия и зависимости, и определяет точки входа для вашего проекта.

В данном случае, структура проекта организована таким способом, чтобы была возможность выполнить сборку проекта в отдельную библиотеку. Если проект не предполагает быть инсталлируемым файл setup.py не создаётся.

Помимо указанных директорий в проекте могут находиться и другие папки для агрегации дополнительных файлов. Например /config для агрегации файлов конфигурации программы; /scripts для скриптов запуска приложения, файлов, которые запускают основную функцию проекта или служат точкой входа в приложение; /extensions для агрегации расширений написанных на других языках.

Именование файлов

**Правило №1: тут нет файлов**

Во-первых — в Python нет таких сущностей, как «файлы», и я заметил, что это — главный источник путаницы для новичков.

Если вы находитесь в директории, содержащей файл \_\_init\_\_.py, то это — директория, включающая в себя модули, а не файлы.

Рассматривайте каждый модуль, как пространство имён.

Я говорю о «пространстве имён», так как нельзя сказать с уверенностью — имеется ли в модуле множество функций и классов, или только константы. В нём может присутствовать практически всё что угодно, или лишь несколько сущностей пары видов.

**Правило №2: если нужно — держите сущности в одном месте**

Совершенно нормально, когда в одном модуле имеется несколько классов. Так и стоит организовывать код (но, конечно, только если классы связаны с модулем).

Выделяйте классы в отдельные модули только в том случае, если модуль становится слишком большим, или если его разные части направлены на решение различных задач.

Часто встречается мнение, что это — пример неудачного приёма работы. Те, кто так считают, находятся под влиянием опыта, полученного после использования других языков программирования, которые принуждают к другим решениям (например — это Java и C#).

**Правило №3: давайте модулям имена, представляющие собой существительные во множественном числе**

Давая модулям имена, следуйте общему правилу, в соответствии с которым эти имена должны представлять собой существительные во множественном числе. При этом они должны отражать особенности предметной области проекта.

Правда, у этого правила есть и исключение. Модули могут называться core, main.py или похожим образом, что указывает на то, что они представляют собой некую единичную сущность. Подбирая имена модулей, руководствуйтесь здравым смыслом, а если сомневаетесь — придерживайтесь вышеприведённого правила.

### Именование классов, функций и переменных

Некоторые программисты считают, что давать сущностям имена — это непросто. Но если заранее определиться с правилами именования, эта задача становится уже не такой сложной.

Имена функций и методов должны быть глаголами. Функции и методы представляют собой действия, или нечто, выполняющее действия.

Функция или метод — это не просто нечто «существующее». Это — нечто «действующее».

Действия чётко определяются глаголами.

Вот — несколько удачных примеров из реального проекта, над которым я раньше работал:

def get\_orders():

...

def acknowledge\_event():

...

def get\_delivery\_information():

...

def publish():

...

А вот — несколько неудачных примеров:

def email\_send():

...

def api\_call():

...

def specific\_stuff():

...

Тут не очень ясно — возвращают ли функции объект, позволяющий выполнить обращение к API, или они сами выполняют какие-то действия, например — отправку письма.

Имена переменных и констант должны быть существительными

Имена переменных и констант всегда должны быть существительными и никогда — глаголами (это позволяет чётко отделить их от функций).

Вот примеры удачных имён:

plane = Plane()

customer\_id = 5

KEY\_COMPARISON = "abc"

Вот — неудачные имена:

fly = Plane()

get\_customer\_id = 5

COMPARE\_KEY = "abc"

А если переменная или константа представляют собой список или коллекцию — им подойдёт имя, представленное существительным во множественном числе:

planes: list[Plane] = [Plane()]

customer\_ids: set[int] = {5, 12, 22}

**Имена классов должны говорить сами за себя, но использование суффиксов — это нормально**

Отдавайте предпочтение именам классов, понятным без дополнительных пояснений. При этом можно использовать и суффиксы, вроде Service, Strategy, Middleware, но — только в крайнем случае, когда они необходимы для чёткого описания цели существования класса.

Всегда давайте классам имена в единственном, а не во множественном числе. Имена во множественном числе напоминают имена коллекций элементов (например — если я вижу имя orders, то я полагаю, что это — список или итерируемый объект). Поэтому, выбирая имя класса, напоминайте себе, что после создания экземпляра класса в нашем распоряжении оказывается единственный объект.

**Классы представляют собой некие сущности**

Классы, представляющие нечто из бизнес-среды, должны называться в соответствии с названиями связанных с ними сущностей (и имена должны быть существительными!). Например — Order, Sale, Store, Restaurant и так далее.

**Пример использования суффиксов**

Представим, что надо создать класс, ответственный за отправку электронных писем. Если назвать его просто Email, цель его существования будет неясна.

Кто-то может решить, что он может олицетворять некую сущность:

email = Email()

email.title = "Title"

email.body = create\_body()

email.send\_to = "guilatrova.dev"

send\_email(email)

Такой класс следует назвать EmailSender или EmailService.

Соглашения по именованию сущностей

Следуйте этим соглашениям по именованию сущностей:

|  |  |  |  |
| --- | --- | --- | --- |
| **Тип** | **Public** | **Protected** | **Private** |
| Пакеты (директории) | lower\_with\_under | — | - |
| Модули (файлы) | lower\_with\_under.py | — | - |
| Классы | CapWords | — | - |
| Функции и методы | lower\_with\_under() | \_lower\_with\_under() | \_\_lower\_with\_under() |
| Константы | ALL\_CAPS\_UNDER | \_ALL\_CAPS\_UNDER | \_\_ALL\_CAPS\_UNDER |

Создание модулей и точки входа в приложение

У каждого приложения есть точка входа.

То есть — имеется единственный модуль (другими словами — файл), который запускает приложение. Это может быть как отдельный скрипт, так и большой модуль.

Когда бы вы ни создавали точку входа в приложение — обязательно добавьте в код проверку на то, что этот код выполняется, а не импортируется:

def execute\_main():

...

if \_\_name\_\_ == "\_\_main\_\_": # Добавьте это условие

execute\_main()

Сделав это, вы обеспечите то, что импорт этого кода не приведёт к его случайному выполнению. Выполняться он будет только в том случае, если будет запущен явным образом.

Тестирование в Python

Юнит-тестом называется автоматизированный тест, который проверяет правильность работы небольшого фрагмента кода (также называемого юнитом),  
делает это быстро, поддерживая изоляцию от другого кода, целью которого является обеспечение стабильного роста проекта.

Возможность покрытия кода тестами — помогает выявить низкокачественный код с относительно высокой точностью. Если вдруг обнаружится, что код трудно протестировать, это верный признак того, что код нуждается в улучшении. Плохое качество обычно проявляется в сильной связности кода — это означает, что части кода недостаточно четко изолированы друг от друга, что в свою очередь создает сложности с их раздельным тестированием. Однако, тот факт, что код проекта легко тестируется, еще не означает, что этот код написан хорошо. Качество кода может быть плохим даже в том случае, если он не страдает сильной связностью.

Просто писать тесты недостаточно: правильный подход к тестированию позволит максимизировать качество и скорость разработки проекта и сохранить темп разработки со временем - поэтому, очень важно научиться отличать хорошие тесты от плохих и научиться рефакторить тесты для повышения их качества.

Плохо написанные тесты на первых порах помогают замедлить ухудшение качества кода (по сравнению с ситуацией, в которой тестов нет вообще), однако, в будущем поспособствуют растяжению срока и затрат на разработку ПО, т.к. могут:

* увеличить количество ошибок,
* нарушать работоспособность кода,
* давать много ложных срабатываний,
* создавать сложности с сопровождением ПО,
* работать медленно,
* не помогать выявлять баги.

Тест с точки зрения разработчика ПО

Тесты — это тоже код. Их следует рассматривать как часть кодовой базы, предназначенную для решения конкретной проблемы: обеспечения правильности приложения.

Юнит-тесты, как и любой другой код, также подвержены ошибкам. При написании тестов, необходимо учитывать как пользу этих тестов, так и затраты на их сопровождение.

Составляющая затрат на сопровождение определяется количеством времени, ушедшего на:

* рефакторинг теста при рефакторинге кода, покрытым этим тестом;
* выполнение теста при каждом изменении кода;
* отвлечение на ложные срабатывания теста;
* затраты на чтение теста при попытке понять, как работает код, покрытый этим тестом.

Успешный набор тестов обладает следующими свойствами:

* он [интегрирован в цикл разработки](https://psbank.digital/step/747);
* он проверяет только самые важные части вашего кода;
* он дает максимальную защиту от багов с минимальными затратами на сопровождение

Выполнение одного теста никоим образом не должно влиять на выполнение другого теста.

Тесты должны полностью игнорировать друг друга:

* Если один из тестов не срабатывает, это значит, что в программе присутствует одна проблема.
* Если не срабатывают два теста, значит, в программе присутствуют две проблемы.
* Если тесты изолированы друг от друга, значит, порядок их выполнения не имеет значения.

Именование юнит-тестов

Существует множество рекомендаций по наименованию юнит-тестов: использование простых фраз или строго типизированных шаблонов, однако, если в компании приняты определенные соглашения, то необходимо придерживаться их.

В текущем курсе для проведения тестирования будем использовать библиотеку unittest. Для корректной работы библиотеки все методы тестового модуля должны начинаться с префикса test\_.

По [ссылке](https://www.geeksforgeeks.org/unit-testing-python-unittest/) вы можете найти некоторую дополнительную информацию касательно написания тестов с использованием unittest.

Рекомендация по юнит-тестам

В попытке написать тесты для модуля можно задаться вопросом: «Что собственно нужно проверять?». Не стоит писать в тестах всё подряд, если тестировать любой, первый попавшийся вам в голову пример, эта информация не скажет ничего о надёжности написанного модуля.

Подумайте, какие ограничения существуют у написанной вами функции. Это могут быть ограничения принимаемых аргументов, типов данных. Подобные ограничения определённо имеют свои допуски и граничные условия, в таком случае мы можем гарантировать выполнение кода, если он корректно отрабатывается на границе допустимого и корректно обрабатывает случаи выхода за граничные условия (выводит соответствующее сообщение об ошибке).

Грамотно составленный тест будет работать всегда одинаково, то есть не зависит от запускаемого устройства, времени его выполнения и т.д., а зависит лишь от переменных окружения тестовой среды.

Задание на лабораторную работу №2

Подготовка

Сделать реструктуризацию репозитория:

* В корне репозитория создать две папки: first\_lab\_work и data\_analysis
* Все файлы первой лабораторной работы перенести в папку first\_lab\_work
* Файлы и папки git оставить в корне.
* Дальнейшую работу вести в папке data\_analysis
* В папке data\_analysis поднять виртуальное окружение.
* Создать конфигурационный файл flake8 и установить максимальную длину строки равной 90 символов.

**Задание.** Проанализировать временной ряд своего домена данных (согласно варианту). Для выполнения работы в проекте написать python пакет (библиотеку) обработки данных с возможностью дальнейшей сборки инсталлируемого модуля. В пакете должны также находиться юнит-тесты. Для работы с пакетом в проекте создать notebook.ipynb файл.

В пакете должен быть реализован функционал скользящего среднего, вычисления дифференциала временного ряда, нахождения автокорреляции и точек экстремума. С использованием данного функционала сделать вывод о сезонности временного ряда, тренде.

Все промежуточные результаты должны сохраняться в excel документ, название документа должно отражать очередность проделанной операции, последнюю проделанную операцию и источник данных.

Варианты задания

|  |  |  |
| --- | --- | --- |
| **№** | **Домен данных** | **Рекомендуемые библиотеки** |
| **1** | Цены на акции | yfinance |
| **2** | Погода | meteostat |
| **3** | Поисковые тренды | pytrends |

Требование к лабораторным работам

* Код должен правильно работать.
* Тестирование завершается успехом.
* Ноутбук файл запускается последовательно без ошибок.
* В проекте присутствует структура.
* В пакете содержится только код анализа данных.
* Именование переменных, функций, методов и классов соблюдается.
* Код читабелен (осмысленное название переменных и функций, прослеживается логика компоновки).
* Соблюдается форматирование кода.
* Отсутствует дублирование кода / логики.
* Отсутствует мусор (закомментированных строк, лишних переменных и т.д.).
* В коде присутствует документация.
* В github репозитории нет лишних файлов / папок.