Алексей Голобурдин “Диджитализируй!”

ТИПИЗИРОВАННЫЙ РУТНОП

Типизированный Python для профессиональной разработки

Алексей Голобурдин,
команда Диджитализируй!

обложка — Васильев Никита, nikita.vasiliev@math.msu.ru

Цель этой книги — помочь тебе научиться писать более красивые, надёжные и легко сопровождаемые программы на Python. То, о чём мы здесь будем говорить, это не начальный уровень владения языком, предполагается, что ты уже минимально умеешь программировать, но хочешь научиться делать это лучше.

И это — отличная цель, к которой мы вместе будем двигаться на протяжении ближайших часов!

Этот материал есть также в видео формате на моём YouTube — «Диджитализируй!».

Также обращаю внимание, что на момент написания этих строк готовится перезапуск моего авторского курса «Основы компьютерных и веб-технологий на Python» course01.to.digital, запуск планируется в июне 2022, если ты читаешь этот материал позже, то вполне вероятно, что на курс уже снова можно записаться.

Итак!

Часто в учебниках и курсах по Python не уделяют должного внимания типизации и некоторым структурам, в то время как они очень важны и могут значительно, просто драматически улучшить твой код.

В ревью кода начинающих разработчиков часто видны результаты того, что в учебных материалах не уделяется отдельное внимание вопросам типизации. В коде не используются подсказки типов, используются неправильно, не лучшим образом выбираются типы для разных данных в приложении и так далее. Качество программы и её надёжность страдают — а это гораздо более важные параметры, чем многие поначалу думают. Поначалу кажется, что я написал программу, она в моих идеальных условиях работает и этого достаточно. Но нет, этого недостаточно.

Наличие функциональности это одно, а надёжность этой функциональности и качество реализации этой функциональности это совсем другое. Наличие функциональности это когда вы видите обувь и думаете — о, отлично, можно её

надеть и пойти в ней куда-то. А надёжность и качество реализации этой функциональности это про то, что у вас не треснет подошва где-то на улице, в обувь не будет попадать вода, обувь не будет натирать вам ноги, она не потеряет быстро приличный вид, а также это про то, что обувь легка в эксплуатации, её можно легко протереть, её можно ремонтировать и многое другое.

То, что мы написали программу и она имеет функциональность — это вовсе не означает, что программа действительно хороша. В этой небольшой книге мы поговорим о том, как разрабатывать, думая не только о функциональности, но и о качестве и надёжности её реализации.

Мы поговорим о типизации в Python, поговорим о нескольких структурах и встроенных типах:

- `NamedTuple`,
- `dataclass`,
- `TypedDict`,
- `Enum`,
- `Literal`,
- `Union`, `Optional`,
- `Iterable`, `Sequence`,
- `Callable`,
- `TypeVar` и др.

Напишем приложение погоды, используя эти типы и поясняя по ходу некоторые архитектурные моменты выбора того или иного подхода. Смотри видео версию этой книги на YouTube и читай обязательно до конца.

Обещаю, что после проработки этого материала твой код больше никогда не будет прежним. Буквально — драматическое улучшение кода гарантировано. Как пишут в англоязычных книжках, *dramatic improvement*!

Поднимаемые вопросы актуальны, кстати, не только для Python, говорить мы будем о нём, но аналогичные подходы применимы и к PHP, TypeScript и тд. Подходы к написанию качественного ПО схожи для разных языков программирования, выступающих просто инструментом реализации задумок автора кода.

Говорить мы здесь будем о версии Python 3.10. В предыдущих версиях Python некоторые аспекты работают чуть иначе (нужно импортировать некоторые типы из `typing`, например), но это не столь критично.

> ✓ **Опчки! Время подписаться!**
>
> YouTube / Telegram / VK

Начать нужно с разговора о самой типизации и о том, почему этому нужно уделять тщательное внимание. Итак, подсказки типов Python или, что то же самое, type hinting.

Type hinting

Что делает любая программа? Оперирует данными, то есть какие-то данные принимает на вход, какие-то данные отдаёт на выход, а внутри данные как-то трансформирует, обрабатывает и передаёт в разные функции, классы, модули и так далее. И весь вопрос в том, в каком виде и формате программа внутри себя эти данные передаёт! То есть — какие типы данных для этого используются. Часто одни и те же данные можно передавать внутри приложения строкой, списком, кортежем, словарём и массой других способов.

Как все мы знаем, Python это язык с динамической типизацией. Что означает динамическая типизация? Что тип переменной определяется не в момент создания переменной, а в момент присваивания значения этой переменной. Мы можем сохранить в переменную строку, потом число, потом список, и это будет работать. Фактически интерпретатор Python сам выводит типы данных и мы их нигде их не указываем, вообще не думаем об этом — просто используем то, что нам нужно в текущий момент.

```
user = "Пётр"
user = 120560
user = {
    "name": "Пётр",
    "username": "petr@email.com",
    "id": 120560
}
user = ("Пётр", "petr@email.com", 120560)
```

Так зачем же вводить type hinting в язык с динамической типизацией? А я напомню, что в Python сейчас есть type hinting, то есть подсказки типов, они же есть в PHP, а в JS даже разработали TypeScipt, отдельный язык программирования, который является надстройкой над JS и вводит типизацию. Зачем это всё делается, для чего? Вроде скриптовые языки, не надо писать типы, думать о них, и всё прекрасно, а тут раз — и вводят какие-то типы данных.

Зачем в динамически типизированном языке вводить явное указание типов?

Раннее выявление ошибок

Первое — это ранее выявление ошибок. Есть у нас некая программа и есть в этой некой программе ошибка. Когда мы можем её выявить? Мы можем выявить её на этапе написания программы, мы можем выявить её на этапе подготовки программы

к разворачиванию на сервере, или мы можем выявить её на этапе runtime, то есть когда программа уже опубликована на сервере, ей пользуются пользователи.

Как вы думаете, на каком этапе лучше выявлять ошибки? Очевидно — чем раньше, тем лучше. Если ошибки долетают до пользователей, то это максимально плохо. Почему?

Во-первых, потому что пользователи недовольны, а раз пользователи недовольны, то много денег мы с нашим программным продуктом не заработаем, так как люди не будут охотно его покупать и рекомендовать другим. К тому же очень неприятно, что мы занимаемся любимым делом, активно трудимся, реализуем сложные алгоритмы, а результатом нашего труда пользователи недовольны. И винить-то объективно некого, кроме нас самих. Непорядочек, непорядочек!

Во-вторых, недовольные пользователи обращаются в техподдержку, создают тикеты, которые спускаются потом на разработку — это всё тратит деньги компании. Если ничего не ломается, то обращений в поддержку мало, тикетов мало, а разработчики заняты разработкой новых фичей продукта, а не постоянными правками отвалившейся старой логики. Постоянные поломки это постоянные финансовые потери.

В-третьих, из-за ошибок, которые видят пользователи, компания несёт репутационные потери. Пользователи пишут негативные отзывы, они легко гуглятся другими потенциальными пользователями, СМИ, инвесторами, всё это в конечном итоге негативно влияет и на капитализацию компании, и на возможности привлечения инвестиций, и на чистую прибыль компании, если она вообще есть.

Если мы хотим быть профессиональными высокооплачиваемыми специалистами, то наша задача — генерировать через результаты нашей работы радость и прибыль, а не поток проблем и убытков.

Поэтому важнейшая задача для нас — сделать так, чтобы до пользователей не доходило ни одной ошибки. Для достижения этой цели нужен системный подход, одной внимательности в процессе программирования мало. Нужна выверенная система, алгоритм действий, инструментарий, который не позволит ошибкам дойти до пользователей.

Какой это может быть инструментарий? Это могут быть автотесты. Однако первый принцип тестирования гласит, что *тестирование может показать наличие дефектов в программе, но не доказать их отсутствие*. Тесты это хорошо, но не на одних только тестах всё держится. Чем больше капканов для разных видов ошибок расставлено, тем надёжнее программа и крепче сон разработчика. А что, в конце концов, может быть важнее крепкого, здорового сна разработчика?

Помимо автотестов (и ручного тестирования людьми) можно проверять корректность использования типов специальными инструментами. Например, компилятор Rust — прооосто красавчик! Он на этапе компиляции выявляет огромное количество проблем и попросту не даёт скомпилировать программу, если видит в ней места с ошибками. Какая-то функция может вернуть успешный результат или условный `null` и вызывающая функция не обрабатывает сценарий с `null`? Вот тебе потенциальная серьёзная ошибка. Компилятор об этом скажет и вам придется сделать всё красиво, обработать все такие моменты и они не дойдут до рантайма.

Штука в том, что в случае с динамически типизированными языками вроде Python, очень сложно написать инструментарий, который бы выполнял проверки по типам, потому что в каждый конкретный момент времени непонятно какой тип в переменной. И для того, чтобы этому инструментарию помогать, вводят подсказки типов в Python, PHP или типизацию в JavaScript в виде отдельного языка TypeScript, компилирующегося в JavaScript. Это то, что помогает выявлять ошибки на этапе до runtime. Либо на этапе подготовки сборки программы, либо даже на этапе написания программы непосредственно в редакторе кода.

Инструмент видит, что вот здесь такой-то тип данных, и если он используется некорректно, то инструмент покажет ошибку и эта ошибка не уйдёт в рантайм и пользователи не словят эту ошибку, а мы как разработчик не схлопочем по макушке от руководства. Прекрасно? Прекрасно!

То есть, ещё раз, первое, для чего вводят подсказки типов — как можно более ранее выявление ошибок, в идеале в редакторе кода в вашей IDE, либо хотя бы на этапе проверки программы перед её сборкой и публикацией на сервер.

На википедии есть прекрасная страница про системы типов, Type system:

> **Wikipedia**
>
> The main purpose of a type system is to reduce possibilities for bugs in computer programs by defining interfaces between different parts of a computer program, and then checking that the parts have been connected in a consistent way.

То есть главной целью системы типов является уменьшение вероятности ошибок в компьютерных программах путём определения интерфейсов между разными частями программы и последующей проверки, что эти части соединены друг с другом правильным образом через эти интерфейсы. Под интерфейсами тут подразумеваются собственно типы данных, например, какие-то конкретные классы, которые описывают конкретные поля и методы.

Допустим, у нас есть вот такая функция:

```
def validate_user(user):
    """Проверяет юзера, райзит исключение, если с ним что-то не так"""
    validate_user_on_server(user)
    check_username(user)
    check_birthday(user)
```

Под `user` тут подразумевается объект юзера, например, ORM объект, то есть запись из базы данных, преобразованная в специальный Python объект. Человек, написавший код, это знает. В момент написания кода знает. Через месяц он об этом совершенно точно забудет, а человек, не писавший этот код, об этом знать вовсе не может. По сигнатуре функции `def validate_user(user)` нельзя понять, какой тип данных ожидается в `user`, но при этом очень легко сделать предположение об этом типе — и ошибиться.

Спустя пол года дописывается какой-то новый код и функция `validate_user` в нём внезапно начинает вызываться с аргументом `user`, который равен не ORM объекту, а числу — просто потому что совсем неочевидно, что в `user` на самом деле подразумевается не число:

```
user_id = 123
validate_user(user_id)
```

Этот код упадёт только в рантайме. Потому что IDE или статический анализатор кода не смогут понять, что тут есть какая-то ошибка.

Как сделать так, чтобы мы узнали об ошибке до этапа рантайма? Явным образом указать тип для атрибута `user`, например, если это экземпляр датакласса `User`, то так (о датаклассах мы поговорим подробнее дальше):

```
from dataclasses import dataclass
import datetime

@dataclass
class User:
    username: str
    created_at: datetime.datetime
    birthday: datetime.datetime | None

def validate_user(user: User):
```

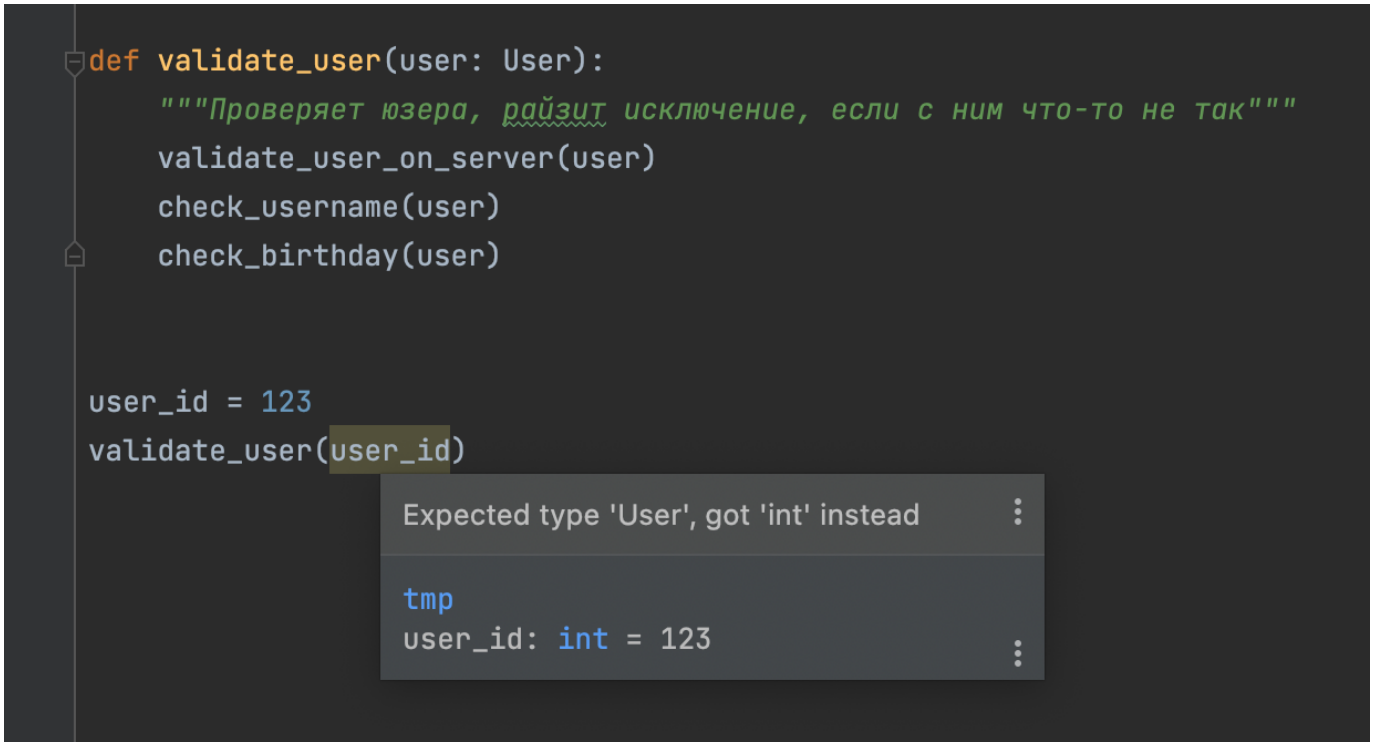
```
    """Проверяет юзера, райзит исключение, если с ним что-то не так"""
    validate_user_on_server(user)
    check_username(user)
    check_birthday(user)
```

Датакласс определяет просто структуру данных с полями `username`, `created_at` и `birthday`, причём тип поля `username` — строка, тип `created_at` — `datetime`, а `birthday` хранит `None` или значение типа `datetime`.

И теперь такой код:

```
user_id = 123
validate_user(user_id)
```

подкрасится ошибкой уже в IDE на этапе написания кода (например, в PyCharm или настроенном VS Code или `nvim`), а также код упадёт с ошибкой в статическом анализаторе кода, который запустится при сборке проекта перед его публикацией на сервер.



Получается, что наша программа стала надёжнее, так как пользователи не увидят многие ошибки, они будут выявлены и исправлены на ранних этапах. Да, при этом надо писать типы, вводить их, думать, но кому сейчас легко:). Нам платят деньги за качественный софт, а качественный софт это в первую очередь надёжный софт, то

есть софт, который не падает в рантайме с непонятными пользователю ошибками вроде `AttributeError`.

> **Важно**
>
> Цена исправления ошибки в программе тем меньше, чем раньше этап, на котором эта ошибка выявлена. Главная причина введения системы типов — уменьшение вероятности возникновения ошибок в рантайме, то есть при эксплуатации приложения.

Читаемость, понятность и поддерживаемость кода

Вернёмся к нашей функции:

```
def validate_user(user):
    """Проверяет юзера, райзит исключение, если с ним что-то не так"""
    validate_user_on_server(user)
    check_username(user)
    check_birthday(user)
```

Представим, что ты пока не очень опытный программист, который только пришел в компанию, и тебе дали задачу добавить ещё одну проверку по юзеру, чтобы валидацию проходили только пользователи, созданные вчера или раньше. Ты этот код очевидно не писал, видишь его впервые и как тут всё работает ещё не знаешь.

Тут `user` — это что? Это словарь key-value? Это ORM объект? Это Pydantic модель? У этого юзера тут есть поле `created_at`, дата создания, или нет? Оно нам в нашей задаче ведь нужно будет.

Как ответить на эти вопросы? Перелопачивать код, который вызывает эту нашу функцию `validate_user`. А там тоже непонятно, что в `user` лежит. Там 100500 функций выше, и где и когда там появляется `user` и что в нём лежит — большой-большой вопрос; плюс мы нашли 2 сценария, в одном наша функция вызывается с `dict`'ом, то есть `user` это словарь, а в другом сценарии функция вызывается с ORM моделью, и возможно еще какой-то код вызывает еще как-то иначе нашу горе-функцию `validate_user`. Вот как с этим жить? Вам может понадобиться конкретно перелопатить весь проект, чтобы понять, как добавить абсолютно простейшую проверку.

А если бы здесь был такой код — то все вопросы решились бы мгновенно:

```python
from dataclasses import dataclass
import datetime

@dataclass
class User:
    username: int
    created_at: datetime.datetime
    birthday: datetime.datetime | None

def validate_user(user: User):
    """Проверяет юзера, райзит исключение, если с ним что-то не так"""
    validate_user_on_server(user)
    check_username(user)
    check_birthday(user)
```

Тут понятно, чем является структура `user`. Тут очевидно, что это класс и у него есть такие-то атрибуты, дата создания юзера, юзернейм и дата рождения юзера. Причем даты хранятся тут не как строки, а как datetime, то есть все вопросы у нас мгновенно снимаются.

Чтение кода значительно облегчилось. Нам понятно, что за данные в `user`, у нас больше нет вопросов, как их обработать. Если вы хотите, чтобы вашим кодом было приятно пользоваться — подсказки типов это обязательный инструмент для вас. Что код принимает на вход? Что он отдаёт на выход? На эти вопросы отвечают подсказки типов.

> **Важно**
>
> Подсказки типов значительно улучшают читаемость кода и облегчают его сопровождение и поддержку.

Помощь IDE при разработке

Если мы пользуемся подсказками типов, то наша IDE уже на этапе написания кода будет подсказывать нам места с потенциальными ошибками, о чем говорилось выше, то есть мы на самом самом раннем этапе некоторые типы ошибок предотвратим, что хорошо, но еще и разрабатывать нам будет удобнее, потому что будет работать автодополнение и все плюшки нашей IDE. В реальности использовать код, который написан с типами, гораздо проще, приятнее, удобнее и быстрее, чем код без явно указанных типов, так как, если типы известны нашей IDE, то она

всячески будет помогать нам, делать автодополнения, предлагать сразу правильные методы, выполнять проверки и так далее.

> **Важно**
>
> При использовании подсказок типов IDE в значительной степени помогает писать код, в полной мере работают подсказки, автодополнения и тд. Чтобы IDE действительно помогала нам писать код, пользоваться подсказками типов просто необходимо.

Zen of Python

Явное лучше неявного. Когда у нас явно и конкретно указан тип данных — это хорошо и это соответствует дзену. Когда же нам приходится неявно домысливать, что тут, наверное, (*наверное!*) строка или `int` — это плохо и это не соответствует дзену.

Простое лучше сложного. Подсказки типов это просто, во всяком случае точно проще, чем попытки описать тип данных в докстринге функции в том или ином формате.

Удобочитаемость важна. Опять же, обсудили выше в примере с юзером, в котором непонятно какая структура без явной типизации.

Должен быть один — и желательно всего один — способ это сделать. Это как раз механизм type hinting. Не надо описывать типы в докстрингах или в формате какого-то внешнего инструмента с птичьим языком. Есть родной механизм в языке программирования, который решает задачу описания типов данных.

Интерпретатор не проверяет подсказки типов

Важно! Здесь стоит отметить, что подсказки типов именно что подсказки. Они не проверяются интерпретатором Python. Они читаются людьми, это подсказка для людей, они читаются IDE, это подсказка для IDE, они могут читаться специальными средствами статического анализа кода вроде `mypy`, это подсказка для них. Но сам интерпретатор не проверяет типы. Если вы укажете type hinting для атрибута функции как `int`, а сами передадите строку — интерпретатор не свалит здесь ошибку, для него в этом не будет проблемы. Имейте это в виду.

```python
def plus_two(num: int):
    print("мы в функции plus_two")
    return num + 2

print(plus_two(5))
# мы в функции plus_two
# 7
```

Мы имеем функцию `plus_two`, которая к переданному аргументу `num` типа `int` прибавляет число `2` и возвращает результат. В этом примере приведено правильное использование этой функции, она вызывается с целочисленным аргументом `5`. Программа работает корректно.

Теперь вызовем функцию с неправильным типом аргумента:

```python
print(plus_two("5"))
# мы в функции plus_two
# TypeError: can only concatenate str (not "int") to str
```

С точки зрения проверки типов эта программа некорректна и на строке, где мы неправильным образом вызываем функцию `plus_two`, у нас покажется ошибка в нашем редакторе кода, также эту ошибку типов покажет и статический анализатор кода вроде `mypy`.

Но интерпретатор именно эту ошибку не заметит. Он не проверяет типы, указанные в type hinting. Обратите внимание — несмотря на то, что функция вызывается явно с неправильным типом данных аргумента `num`, она всё равно запускается, так как `print("мы в функции plus_two")` срабатывает. Функция запускается и «падает» уже тогда, когда мы пытаемся сложить строку `"5"` и число `2`.

Python — это по-прежнему язык с динамической типизацией, а подсказки типов являются именно что подсказками для разработчика, IDE и анализатора кода, эти подсказки призваны упростить жизнь разработчику и снизить количество ошибок в рантайме. Интерпретатор на подсказки типов внимания не обращает.

Итак, подводя промежуточный итог: подсказки типов это очень важно и вам точно следует их изучить и ими пользоваться. А как подсказками типов пользоваться и какие есть варианты — поговорим подробнее дальше.

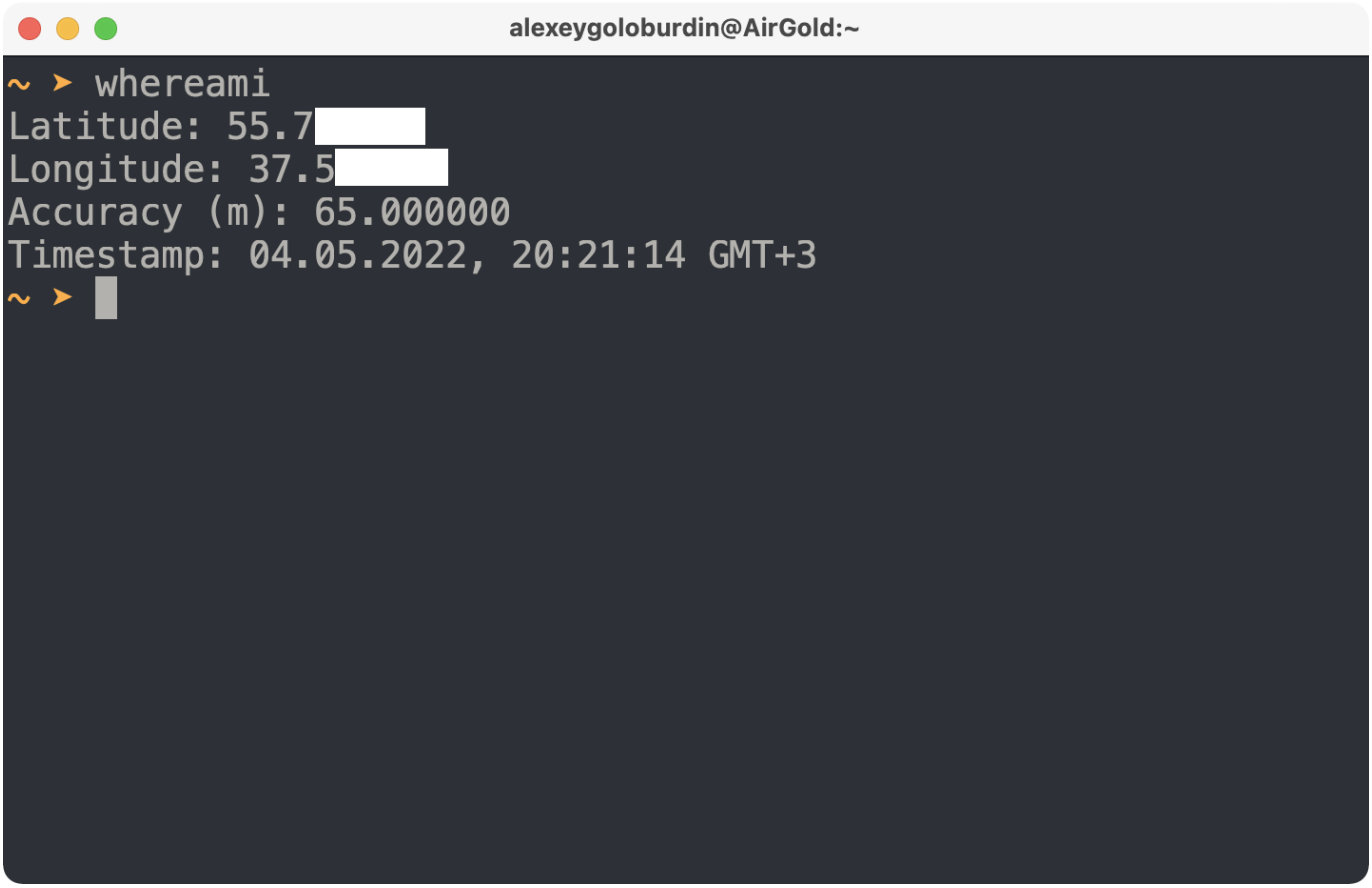
Пишем программу погоды

Подготовка

Итак, давайте напишем консольную программу, которая будет показывать текущую погоду по нашим координатам. Чтоб не по IP адресу как-то пытаться неточно вычислять местоположение, а именно по текущим реальным GPS координатам. Чтобы программа показывала температуру за бортом, идёт ли там дождь/снег и время восхода-заказа солнца. Для съёмки видео важно понимать, во сколько сегодня восход или закат солнца, чтобы ориентироваться на освещённость за окном.

Итак, в первую очередь, нам надо понять, как получить доступ к текущим координатам, есть ли такая возможность. Решение будет для MacBook, гуглим: `python mac get gps coordinates`. Первая ссылка говорит о программе whereami, которая печатает текущие координаты в консоль

```
whereami
```



Отлично, теперь мы можем получать наши текущие координаты, отправить их в какой-то сервис погоды через API, получить оттуда погоду и отобразить её.

Команда работает по аналогии с `whoami` — та показывает, «кто я», а вот команда `whereami` показывает, «где я»:).

Давайте найдём какой-то сервис погоды. Поисковый запрос `API прогноз погоды` привёл меня на проект OpenWeather. У них есть бесплатный доступ. Еще есть Яндекс погода в России, Gismeteo, но там, насколько я понял, для получения API ключа надо куда-то писать на почту, для наших целей это слишком долго. Воспользуемся OpenWeather.

Запрос на получение погоды по примерно моим координатам:

```
http https://api.openweathermap.org/data/2.5/weather\?
lat\=55.7\&lon\=37.5\&appid\=7549b3ff11a7b2f3cd25b56d21c83c6a\&lang\=ru\&unit
s\=metric
```

`httpie` — это удобная утилита работы с веб-сервисами, такая вариация на тему `curl`, можно установить на Mac OS с помощью brew командой `brew install httpie`. Она выводит в раскрашенном виде JSON, например, что удобно.

API ключ, использующийся в запросе, получается сразу после регистрации, но активируется в течение, может быть, минут десяти.

Результат запрос:


```
{
    "base": "stations",
    "clouds": {
        "all": 61
    },
    "cod": 200,
    "coord": {
        "lat": 55.7,
        "lon": 37.5
    },
    "dt": 1651521003,
    "id": 529334,
    "main": {
        "feels_like": 9.26,
        "grnd_level": 993,
        "humidity": 74,
        "pressure": 1013,
```

```
        "sea_level": 1013,
        "temp": 10.25, <!-- температура в градусах Цельсия -->
        "temp_max": 12.01,
        "temp_min": 8.55
    },
    "name": "Moscow", <!-- твоё место -->
    "sys": {
        "country": "RU",
        "id": 47754,
        "sunrise": 1651455877, <!-- восход в Unix time -->
        "sunset": 1651511306, <!-- закат в Unix time -->
        "type": 2
    },
    "timezone": 10800,
    "visibility": 10000,
    "weather": [
        {
            "description": "облачно с прояснениями",
            "icon": "04n",
            "id": 803, <!-- тип погоды, справочник
https://openweathermap.org/weather-conditions#Weather-Condition-Codes-2 -->
            "main": "Clouds"
        }
    ],
    "wind": {
        "deg": 180,
        "gust": 8.08,
        "speed": 2.69
    }
}
```

Так, отлично, мы умеем находить текущие координаты и умеем по ним получать температуру, состояние погоды — дождь/снег/облака, а также получать время восхода и заката солнца.

Давайте напишем программу для этого! Чтобы запускаешь её и она писала наше местоположение и выводила эти данные — температуру, характеристику погоды (снег/облака/туман) и время восхода заката солнца.

Накидываем структуру приложения

Итак, первое, что надо сделать — подумать, из каких слоёв будет состоять наше приложение. Бросаться писать код сразу не надо. Давайте подумаем, что будет делать наша программа, вот просто перечислим, не думая пока о коде, функциях,

классах, о том, как именно мы будем это реализовывать, а просто подумаем, что будет делать программа, из каких функциональных блоков она будет состоять.

Итак, наша программа погоды должна:

- уметь получать текущие координаты устройства
- запрашивать по этим координатам где-то погоду, в нашем случае на OpenWeather, но потенциально было бы здорово, если бы была возможность потом подцепить и какой-то другой сервис, если понадобится
- результаты работы этого погодного сервиса надо распарсить, то есть разобрать, чтобы выдернуть оттуда нужные нам данные
- и, наконец, наши данные надо отобразить в терминале.

Получается, 4 блока тут есть, причём второй и третий функции мы можем объединить в один верхнеуровневый слой получения погоды из внешнего сервиса. Итого мы имеем следующие слои работы приложения:

1. Слой, запускающий приложение и связывающий остальные слои
2. Получение текущих координат
3. Получение по координатам погоды
4. Печать погоды

Отлично.

Создаём директорию и накидываем туда слои нашего приложения. Сразу создаём структуру. У нас есть 4 слоя нашего приложения, создадим под них сразу Python модули, чтобы логика каждого слоя лежала сразу в них.

- `weather` — входная точка приложения, сделаем её исполнимым файлом без расширения `.py`, чтобы можно было запускать её без указания интерпретатора
- `gps_coordinates.py` — получение текущих GPS координат ноутбука
- `weather_api_service.py` — работа с внешним сервисом прогноза погоды
- `weather_formatter.py` — форматирование погоды, то есть «сборка» строки с данными погоды (например, для последующей печати этой строки в терминале)

Создаём директорию для проекта, в моём случае `weather-yt`, переходим в неё и создаём в ней пустой файл `weather`, добавляем этому файлу права на выполнение с помощью `chmod`, и затем открываем этот файл в редакторе кода, в моём случае в `nvim`:

```
mkdir weather-yt && cd $_
true > weather
chmod +x weather
nvim weather
```

Зададим заглушку в файле `weather`:

```
#!/usr/bin/env python3.10
print("Hello world")
```

Первая строка называется шебанг, при помощи чего будет запускаться текущий файл. В нашем случае файл будет запускаться с помощью интерпретатора `python3.10`. Убедитесь, что у вас есть такой интерпретатор в системе, что путь к нему добавлен в переменную окружения `PATH`, убедитесь, что `python3.10` успешно запускается. Мы будем использовать здесь возможности актуальной на сегодня версии Python — 3.10. Проверим работу приложения:

```
./weather
```

Отлично! Точка входа в приложение готова. Теперь сделаем, чтобы она запускалась откуда угодно из системы, прокинув симлинк (ярлык) на этот исполнимый файл в `/usr/local/bin/`:

```
sudo ln -s $(pwd)/weather /usr/local/bin/
```

Отлично, теперь мы можем узнавать погоду (запускать `weather`) из любой директории в системе.

Создаём остальные модули приложения:

```
true > gps_coordinates.py
true > weather_api_service.py
true > weather_formatter.py
ls -l
```

Итак, у нас есть структура приложения, начинаем накидывать функционал.

Пишем каркас приложения

Итак, накидывать функционал можно по-разному. Есть множество подходов. Например, есть TDD, *Test Driven Development*, согласно которому надо сначала написать много-много тестов, они все сначала падают с ошибками, а потом просто постепенно мы пишем код, который заставляет эти тесты постепенно работать.

Мы по TDD сейчас не пойдём и тесты писать не будем, но в целом можете иметь в голове такой подход. Мы начнём постепенно с реализации, думая каждый раз, в каком слое должна лежать реализуемая сейчас функция или класс.

Файл `weather`:

```
#!/usr/bin/env python3.10
from coordinates import get_coordinates
from weather_api_service import get_weather
from weather_formatter import format_weather


def main():
    coordinates = get_coordinates()
    weather = get_weather(coordinates)
    print(format_weather(weather))


if __name__ == "__main__":
    main()
```

То есть фактическая реализация логики будет инкапсулирована, то есть заключена в отдельные Python модули `gps_coordinates`, `weather_api_service` и `weather_formatter`, а модуль `weather` будет просто точкой входа в приложение, запускающей логику.

Обратите внимание — при таком подходе у нас изначально не получится ситуации, что вся логика написана в одной каше, например, вообще без функций или в одной длинной километровой функции. Мы подумали о слоях нашего приложения, для каждого слоя создали отдельное место, в нашем случае Python модуль, но впоследствии можно расширить до Python пакета, и теперь будем создавать функции, в которые ляжет бизнес-логика нашего приложения.

Файл `gps_coordinates.py`:

```
def get_gps_coordinates():
    """Returns current coordinates using MacBook GPS"""
    pass
```

И вот тут было бы неплохо подумать, какой формат данных вернёт эта чудесная функция. Она очевидно должна вернуть координаты. Координаты это широта и долгота, то есть два числа. Какие есть варианты?

Самый простой вариант — просто tuple с двумя float числами:

```
def get_gps_coordinates() -> tuple[float, float]:
    """Returns current coordinates using MacBook GPS"""
    pass
```

Так, хорошо... А широта это нулевой элемент кортежа, а долгота это первый элемент, да? Или наоборот? Насколько хорошо, что приходится додумывать или читать внутренний код функции, чтобы понять, что она возвращает? В этом нет ничего хорошего. Надо явным образом прописать тип, чтобы разночтений не было и все типы были понятны по сигнатуре функции.

NamedTuple — именованный кортеж

Можно воспользоваться именованным кортежем `NamedTuple`. В Python есть именованные кортежи в составе пакета `collections` и в составе `typing`. Чтобы можно было указать полям кортежа типы мы, конечно, воспользуемся импортом из `typing`:

```
from typing import NamedTuple

class Coordinates(NamedTuple):
    latitude: float
    longitude: float

def get_gps_coordinates() -> Coordinates:
    """Returns current coordinates using MacBook GPS"""
    return Coordinates(10, 20)
```

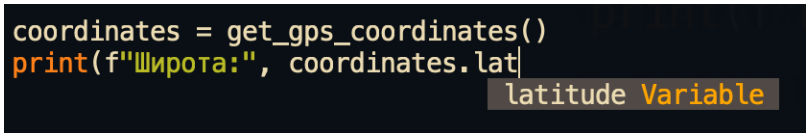
Именованные кортежи — такие же кортежи, как и обычные `tuple`, но каждый элемент кортежа имеет имя, по которому мы можем к нему обращаться.

Обращаться по имени ведь проще, чем по индексу. Индекс `0` нам мало что говорит о данных, которые лежат в этом индексе, а имя `longitude` прямо говорит нам, что тут хранится географическая долгота.

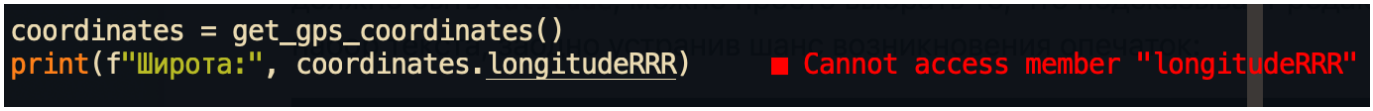
Теперь пользоваться кодом проще и разночтений никаких нет:

```
coordinates = get_gps_coordinates()
print(f"Широта:", coordinates.latitude)  # Печать широты
print(f"Долгота:", coordinates.longitudeRRR)  # IDE подсветит ошибку опечатки
```

В редакторе кода срабатывает автокомплит (*autocomplete*), то есть автодополнение кода. Мы начинаем набирать `coordinates.lat` и редактор подсказывает нам, что здесь должно быть `latitude`, можно просто выбрать то, что подсказывает редактор и ускорить набор текста, заодно устранив шанс возникновения опечаток:



А ещё, если по какой-то причине опечатки всё же возникли, то редактор подсветит нам места с такими проблемами:



При этом такой именованный кортеж по-прежнему является кортежем, то есть им можно пользоваться и так, с распаковкой:

```
latitude, longitude = get_gps_coordinates()
```

А также, как и в случае с обычным кортежем, нельзя изменять значения элементов кортежа:

```
coordinates = get_gps_coordinates()
coordinates.latitude = 10  # IDE подсветит ошибку тут
```

Обычный словарь dict

Вторым вариантом структуры, которой тут можно воспользоваться — это словарь, просто обычный `dict`:

```
# Совсем плохо! Что за dict, что внутри в нём?
def get_gps_coordinates() -> dict:
    return {"longitude": 10, "latitude": 20}

# Так лучше, хотя бы прописаны типы для ключей и значений
def get_gps_coordinates() -> dict[str, float]:
    return {"longitude": 10, "latitude": 20}

coords = get_gps_coordinates()
print(coords["longitude"])  # IDE не покажет опечатку в `longitude`
```

Как видно, при вводе ключа словаря `longitude` IDE нам не подсказывает и нет никакой проверки на опечатки. Если мы опечатаемся в ключе словаря, то эта ошибка может дойти до рантайма и уже в рантайме упадёт ошибка `KeyError`. Хочется, чтобы IDE и статический анализатор кода вроде `mypy`, о котором поговорим позднее, помогали нам, а чтобы они нам помогали, надо чётко прописывать типы данных и `dict` это не то, что нам нужно.

Словарь с Literal ключами

Плюс в описанном выше словаре ключом является строка, получается — любая строка? Но нет, в реальности не любая, а только одна из двух строк — `longitude` или `latitude`. Это можно отразить в type hinting с помощью `Literal`:

```
from typing import Literal

def get_gps_coordinates() -> dict[Literal["longitude"] | Literal["latitude"],
float]:
    return {"longitude": 10, "latitude": 20}

print(
    get_gps_coordinates["longitude"]
)

print(
    get_gps_coordinates["longitudeRRR"]  # Тут IDE покажет ошибку!
)
```

`Literal` позволяет указать не просто какой-то тип вроде `str`, а позволяет указать конкретное значение этого типа. В данном случае у нас ключом может быть либо строка со значением `"longitude"`, либо строка со значением `"latitude"`.

Вот эта вертикальная черта обозначает ИЛИ, то есть или тип слева от черты, или тип справа от черты. Это синтаксис Python 3.10, в предыдущих версиях Python нужно было импортировать из `typing` специальный тип `Union`, который делал то же самое. Сейчас можно просто пользоваться вертикальной чертой для того, чтобы задать несколько возможных типов для переменной.

Кстати, *literally* — по-русски означает «буквально». То есть, когда нам надо буквально задать конкретные значения в типе, мы можем это сделать при помощи типа `Literal`.

В целом в таком формате использовать словарь здесь можно, но мне больше нравится вариант с именованным кортежем из этих двух вариантов.

TypedDict

Есть еще специальный типизированный `dict`. Если почему-то хочется иметь доступ к данным именно как к словарю, а не как к классу (то есть писать `coordinates["latitude"]` вместо `coordinates.latitude`), то можно воспользоваться типизированным словарём:

```
from typing import TypedDict

class Coordinates(TypedDict):
    longitude: float
    latitude: float

c = Coordinates(longitude=10, latitude=20)
print(c["longitude"])  # Работает автодополнение в IDE
print(c["longitudeRRR"])  # IDE покажет ошибку
```

Я на практике не вижу большого смысла пользоваться именно типизированным словарём, но в целом можно найти какие-то сценарии для его использования. Например, уже есть много кода, который использует нашу структуру как словарь, но мы хотим добавить в эту структуру типизацию и при этом не переписывать код, который уже использует эту структуру. В таком сценарии как раз имеет смысл воспользоваться `TypedDict`.

А когда мы пишем новый код, то именованные кортежи или датаклассы это наиболее часто используемые варианты. О датаклассах мы как раз сейчас и поговорим!

Dataclass

Ещё один вариант задания структуры — `dataclass`:

```
from dataclasses import dataclass

@dataclass
class Coordinates:
    longitude: float
    latitude: float

def get_gps_coordinates() -> Coordinates:
    return Coordinates(10, 20)
```

Это обычный класс, это не именованный кортеж, распаковывать его как кортеж уже нельзя, и также он не ведет себя как кортеж с точки зрения изменения каждого элемента. Это обычный класс.

С ним работают проверки в IDE, автодополнения — это, пожалуй, самая часто используемая структура для таких задач:

```
print(get_gps_coordinates().latitude)  # Автодополнение IDE для атрибута
print(get_gps_coordinates().latitudeRRR)  # IDE подсветит опечатку
```

Когда использовать `NamedTuple`, когда `dataclass`? Как мы поймём чуть позже, сценарий именованных кортежей — это сценарий распаковки. Когда нам нужно использовать структуру именно как кортеж, тогда стоит задать её как `NamedTuple`. В остальных сценариях имеет смысл предпочесть `dataclass`.

Давайте сравним количество памяти, которое занимает в оперативке именованный кортеж и датакласс. Для того, чтобы узнать, сколько памяти занимает переменная, воспользуемся библиотекой Pympler.

```
from dataclasses import dataclass
from typing import NamedTuple

from pympler import asizeof

@dataclass
class CoordinatesDT:
    longitude: float
    latitude: float

class CoordinatesNT(NamedTuple):
    longitude: float
    latitude: float


coordinates_dt = CoordinatesDT(longitude=10.0, latitude=20.0)
coordinates_nt = CoordinatesNT(longitude=10.0, latitude=20.0)

print("dataclass", asizeof.asized(coordinates_dt).size)  # 328 bytes
print("namedtuple:", asizeof.asized(coordinates_nt).size)  # 104 bytes
```

То есть, как видим, именованный кортеж занимает значительно меньше памяти в оперативке, чем `dataclass`, в данном примере в 3 раза. Это понятно, то как по своей сути это более простая структура данных, её нельзя менять и потому именованный кортеж можно эффективно хранить в памяти.

В то же время, если мы используем `dataclass` просто как фиксированную структуру для хранения неизменяемых данных, то можно сделать и его более эффективным:

```
from dataclasses import dataclass
from pympler import asizeof


@dataclass(slots=True, frozen=True)
class CoordinatesDT2:
    longitude: float
    latitude: float

coordinates_dt2 = CoordinatesDT2(longitude=10.0, latitude=20.0)
print("dataclass with frozen and slots:",
asizeof.asized(coordinates_dt2).size)
# dataclass with frozen and slots: 96 bytes
```

Обрати внимание — такая структура неизменна, как и кортеж (благодаря флагу `frozen=True`), то есть не получится после определения экземпляра класса изменить его атрибуты. Флаг `slots=True` автоматически добавляет `__slots__` нашему датаклассу (более быстрый доступ к атрибутам и более эффективное хранение в памяти).

Таким образом, как мы видим по нашему тесту, по памяти такой `dataclass` получается даже эффективнее кортежа. Кортеж можно использовать, если вам важно использовать его с распаковкой, например, таким образом:

```
latitude, longitude = coordinates_nt
```

Экземпляр датакласса, очевидно, с распаковкой работать не будет, так как это не кортеж.

Alias для типа

После нашего отступления о структурах продолжим накидывать каркас приложения.

В `gps_coordinates.py` оставим структуру `dataclass` с параметрами `slots` и `frozen`, потому что не предусматривается изменение координат, которые вернёт нам GPS датчик на ноутбуке.

```
from dataclasses import dataclass

@dataclass(slots=True, frozen=True)
class Coordinates:
    longitude: float
    latitude: float

def get_coordinates() -> Coordinates:
    return Coordinates(longitude=10, latitude=20)
```

Возвращаемое значение вставили пока, просто чтобы не ругались проверки в редакторе. Потом напишем реализацию, которая запросит координаты у команды `whereami`, распарсит её результаты и вернёт как результат функции `get_coordinates`.

Составим `weather_api_service.py`:

```python
from coordinates import Coordinate

def get_weather(coordinates: Coordinate):
    """Requests weather in OpenWeather API and returns it"""
    pass
```

Так, какой тип у погоды будет возвращаться? Тут главное не смотреть на формат данных в API сервисе, потому что сервис и формат данных в нём вторичны, первичны наши потребности. Какие данные нам нужны? Нам нужна температура за бортом, наше место, общая характеристика погоды — ясно/неясно/снег/дождь и тп, а также мне лично ещё интересно, во сколько сегодня восход солнца и закат солнца. Вот эти данные нам нужны, их пусть функция `get_weather` и возвращает. В каком формате?

Так, ну давайте думать. Просто `tuple`? Точно нет. Вообще есть мнение, что если мы хотим использовать `tuple`, то стоит использовать `NamedTuple`, потому что в нём явно данные будут названы. Поэтому возможно `NamedTuple`.

Просто `dict`? Точно нет. Не будет нормальных проверок в IDE и статическом анализаторе, не будет подсказок, и читателю кода непонятно, что там внутри словаря. `TypedDict`? Лучше, но мне нравится доставать данные как атрибуты класса, а не как ключи словаря. Поэтому `TypedDict` тоже не будем брать.

Может `dataclass`? Можно.

Итого `NamedTuple` или `dataclass`? Оба варианта ок, можно выбрать любой вариант, я, пожалуй, тут выберу `dataclass` с параметрами `frozen` и `slots` просто потому что распаковывать структуру как кортеж нам незачем, а по памяти `dataclass` с такими параметрами даже эффективнее кортежа.

```python
from dataclasses import dataclass
from datetime import datetime

from coordinates import Coordinate

Celsius = int

@dataclass(slots=True, frozen=True)
class Weather:
    temperature: Celsius
    weather_type: str  # Подумаем, как хранить описание погоды
    sunrise: datetime
    sunset: datetime
    city: str

def get_weather(coordinates: Coordinate):
    """Requests weather in OpenWeather API and returns it"""
    pass
```

Обратите внимание, как я обошёлся с температурой. Можно было прописать тип напрямую `int`, но я сделал *alias*, то есть псевдоним, для `int` с названием `Celsius` и теперь понятно, что у нас температура тут будет именно в градусах Цельсия, а не Фаренгейта или Кельвина.

Также, если какая-то функция будет принимать на вход или возвращать температуру, то мы тоже укажем для температуры там конкретный тип `Celsius`, а не общий непонятный `int`.

Enum

Дальше, как быть с полем `weather_type`? Что за строка там будет? Хочется, чтобы там была не просто любая строка, а строго один из заранее заданных вариантов. Тут мы будем хранить описание погоды — ясно, туманно, дождливо и тп. Для этих целей существует структура `Enum`. Её название происходит от слова *Enumeration*, *перечисление*. Когда нам нужно перечислить какие-то заранее заданные варианты значений, то `Enum` это та самая структура, которая нам нужна.

Давайте создадим структуру типов погоды, отнаследовав её от `Enum` и заполнив всеми возможными типами погоды, которые мы возьмём из справочника с OpenWeather:

```python
from datetime import datetime
from enum import Enum

class WeatherType(Enum):
    THUNDERSTORM = "Гроза"
    DRIZZLE = "Изморось"
    RAIN = "Дождь"
    SNOW = "Снег"
    CLEAR = "Ясно"
    FOG = "Туман"
    CLOUDS = "Облачно"

@dataclass(slots=True, frozen=True)
class Weather:
    temperature: Celsius
    weather_type: WeatherType
    sunrise: datetime
    sunset: datetime
    city: str
```

Каждый конкретный тип погоды описывается через атрибут `WeatherType`:

```python
print(WeatherType.CLEAR)  # WeatherType.CLEAR
print(WeatherType.CLEAR.value)  # Ясно
print(WeatherType.CLEAR.name)  # CLEAR
```

В чём фишка `Enum`? Зачем наследовать наш класс от `Enum`, почему бы просто не сделать класс с такими же атрибутами класса? Допустим, у нас есть функция `print_weather_type`, которая печатает погоду:

```python
from enum import Enum

class WeatherType(Enum):
    THUNDERSTORM = "Гроза"
    DRIZZLE = "Изморось"
    RAIN = "Дождь"
    SNOW = "Снег"
    CLEAR = "Ясно"
    FOG = "Туман"
    CLOUDS = "Облачно"

def print_weather_type(weather_type: WeatherType) -> None:
    print(weather_type.value)
```

```
print_weather_type(WeatherType.CLOUDS)  # Облачно
```

Как видите, тип для аргумента функции `weather_type` указан как `WeatherType`. А передаём туда мы не экземпляр `WeatherType`, а `WeatherType.CLOUDS`, при этом наш «проверятор» кода в IDE не ругается, ему всё нравится. Дело в том, что:

```
print(
    isinstance(WeatherType.CLOUDS, WeatherType)
)  # True
```

То есть `WeatherType.CLOUDS` является экземпляром самого типа `WeatherType`, и это позволяет нам таким образом использовать этот класс в подсказке типов. В функцию `print_weather_type` можно передать только то, что явным образом перечислено в `Enum` структуре `WeatherType` и ничего больше.

Если мы уберём наследование от `Enum`, то IDE сразу скажет нам о несоответствии типов:

```
from enum import Enum

class WeatherType:  # Убрали наследование от Enum
    THUNDERSTORM = "Гроза"
    DRIZZLE = "Изморось"
    RAIN = "Дождь"
    SNOW = "Снег"
    CLEAR = "Ясно"
    FOG = "Туман"
    CLOUDS = "Облачно"

def print_weather_type(weather_type: WeatherType) -> None:
    print(weather_type)  # Вместо weather_type.value

print_weather_type(WeatherType.CLOUDS)  # IDE подсветит ошибку типов
```

Здесь `WeatherType.CLOUDS` — это обычная строка со значением `"Облачно"`, тип `str`, а не `WeatherType`. Тип `str` и тип `WeatherType` — разные, поэтому IDE определит и подсветит эту ошибку несоответствия типов.

В этом особенность `Enum`. Цель этой структуры — задавать перечисление возможных вариантов значений.

Ещё по `Enum` можно итерироваться в цикле, что иногда может быть удобно:

```
for weather_type in WeatherType:
    print(weather_type.name, weather_type.value)
```

И, конечно, `Enum` структуру можно разбирать с помощью новых возможностей Python — Pattern Matching:

```
def what_should_i_do(weather_type: WeatherType) -> None:
    match weather_type:
        case WeatherType.THUNDERSTORM | WeatherType.RAIN:
            print("Уф, лучше сиди дома")
        case WeatherType.CLEAR:
            print("О, отличная погодка")
        case _:
            print("Ну так, выходить можно")

what_should_i_do(WeatherType.CLOUDS)  # Ну так, выходить можно
```

Но нам здесь это пока не нужно.

Также часто полезным бывает отнаследовать класс перечисления от `Enum` и от `str`. Тогда значение можно использовать как строку без обращения к `.value` атрибуту:

```
# Наследование от str и Enum
class WeatherTypeStrEnum(str, Enum):
    FOG = "Туман"
    CLOUDS = "Облачно"

# Вариант без наследования от str
class WeatherTypeEnum(Enum):
    FOG = "Туман"
    CLOUDS = "Облачно"

print(WeatherTypeStrEnum.CLOUDS.upper())  # ОБЛАЧНО
print(WeatherTypeEnum.CLOUDS.upper())  # AttributeError
print(WeatherTypeEnum.CLOUDS.value.upper())  # ОБЛАЧНО

print(WeatherTypeStrEnum.CLOUDS == "Облачно")  # True
print(WeatherTypeEnum.CLOUDS == "Облачно")  # False
print(WeatherTypeEnum.CLOUDS.value == "Облачно")  # True

print(f"Погода: {WeatherTypeStrEnum.CLOUDS}")  # Погода: Облачно
```

```
print(f"Погода: {WeatherTypeEnum.CLOUDS}")  # Погода: WeatherTypeEnum.CLOUDS
print(f"Погода: {WeatherTypeEnum.CLOUDS.value}")  # Погода: Облачно
```

При этом тип `WeatherTypeStrEnum` и `str` — это всё же разные типы. Если аргумент функции ожидает `WeatherTypeStrEnum`, то передать туда `str` не получится. Типизация работает как надо:

```
def make_something_great_with_weather(weather: WeatherTypeStrEnum): pass

smth("Туман")  # Не пройдёт проверку типов
smth(WeatherTypeStrEnum.FOG)  # Ок, всё в порядке
```

Какие еще варианты для использования Enum можно придумать? Например, перечисление полов, мужской/женский. Перечисление статусов запросов, ответов, каких-то операций. Перечисление статусов заказов, например, если эти статусы зашиты в приложении, а не берутся из справочника БД. Перечисление дней недели (понедельник, вторник и тд).

Итак, полный код `weather_api_service.py` на текущий момент:

```python
from datetime import datetime
from dataclasses import dataclass
from enum import Enum

from coordinates import Coordinate

Celsius = float

class WeatherType(str, Enum):
    THUNDERSTORM = "Гроза"
    DRIZZLE = "Изморось"
    RAIN = "Дождь"
    SNOW = "Снег"
    CLEAR = "Ясно"
    FOG = "Туман"
    CLOUDS = "Облачно"

@dataclass(slots=True, frozen=True)
class Weather:
    temperature: Celsius
    weather_type: WeatherType
    sunrise: datetime
    sunset: datetime
    city: str

def get_weather(coordinates: Coordinate) -> Weather:
    """Requests weather in OpenWeather API and returns it"""
    return Weather(
       temperature=20,
       weather_type=WeatherType.CLEAR,
       sunrise=datetime.fromisoformat("2022-05-04 04:00:00"),
       sunset=datetime.fromisoformat("2022-05-04 20:25:00"),
       city="Moscow"
   )
```

Обратите внимание, как всё чётенько! Мы читаем описание функции `get_weather` и у нас не может быть непониманий, что эта функция принимает на вход и в каком формате, а также что она возвращает на выход и опять же в каком формате. Если в будущем мы будем работать не с OpenWeather API, а с каким-то другим сервисом погоды, то мы просто заменим слой общения с этим внешним сервисом, но пока наша функция `get_weather` будет возвращать структуру `Weather`, весь остальной, внешний по отношению к этой функции, код будет работать без изменений. Мы прописали интерфейс коммуникации функции `get_weather` с внешним миром и пока этот интерфейс поддерживается — неважно как и откуда получаются данные внутри

этой функции, главное, чтобы они просто на выходе преобразовались в нужный нам формат.

Отлично, осталось реализовать заглушку для принтера, который будет печатать нашу погоду, `weather_formatter.py`:

```
from weather_api_service import Weather

def format_weather(weather: Weather) -> str:
    """Formats weather data in string"""
    return "Тут будет печать данных погоды из структуры weather"
```

Отлично, каркас приложения готов. Прописаны основные типы данных, что функции принимают на вход и возвращают. По этим функциям и типам уже сейчас понятно, как будет работать приложение, хотя мы ещё по сути ничего не реализовали.

Заполним полученный скелет приложения теперь реализацией.

Реализация приложения — получение GPS координат

Реализуем в первую очередь получение GPS координат, `get_gps_coordinates.py`:

```
from dataclasses import dataclass
from subprocess import Popen, PIPE

from exceptions import CantGetCoordinates

@dataclass(slots=True, frozen=True)
class Coordinates:
    longitude: float
    latitude: float

def get_gps_coordinates() -> Coordinates:
    """Returns current coordinates using MacBook GPS"""
    process = Popen(["whereami"], stdout=PIPE)
    (output, err) = process.communicate()
    exit_code = process.wait()
    if err is not None or exit_code != 0:
        raise CantGetCoordinates
    output_lines = output.decode().strip().lower().split("\n")
    latitude = longitude = None
    for line in output_lines:
        if line.startswith("latitude:"):
            latitude = float(line.split()[1])
```

```
            if line.startswith("longitude:"):
                longitude = float(line.split()[1])
        return Coordinates(longitude=longitude, latitude=latitude)

    if __name__ == "__main__":
        print(get_gps_coordinates())
```

Хочу обратить внимание тут вот на что. Если что-то пошло не так с процессом получения координат — мы не возвращаем какую-то ерунду вроде `None`. Мы возбуждаем (*райзим*, от англ. *raise*) исключение. Причём исключение не какое-то системное вроде `ValueError`, а наш собственный тип исключения, который мы назвали `CantGetCoordinates` и положили в специальный модуль, куда мы будем класть исключения `exceptions.py`:

```
class CantGetCoordinates(Exception):
    """Program can't get current GPS coordinates"""
```

Почему не `ValueError`, а свой тип исключений? Чтобы разделять обычные питоновские `ValueError` от конкретно нашей ситуации с невозможностью получить координаты. Явное лучше неявного.

Почему исключение, а не возврат `None`? Потому что если у функции есть нормальный сценарий работы и ненормальный, то есть исключительный, то исключительный сценарий должен использовать исключения, а не возвращать какую-то ерунду вроде `False`, `0`, `None`, `tuple()`. Исключительная ситуация должна возбуждать исключение, и уже на уровне выше нашей функции мы должны решить, что с этой исключительной ситуацией делать. Код, который будет вызывать нашу функцию `get_gps_coordinates`, решит, что делать с исключительной ситуацией, на каком уровне и как эта ситуация должна быть обработана.

Отлично. Функция отдаёт сейчас точные координаты, которые я не хочу раскрывать, давайте введём в приложение конфиг `config.py` и в нём зададим, использовать точные координаты или примерные. Я буду использовать примерные координаты. Погода от этого не изменится, просто в другой район города попаду.

`config.py`:

```
USE_ROUNDED_COORDS = True
```

`get_gps_coordinates.py`:

```
from dataclasses import dataclass
from subprocess import Popen, PIPE

import config
from exceptions import CantGetCoordinates

@dataclass(slots=True, frozen=True)
class Coordinates:
    longitude: float
    latitude: float

def get_gps_coordinates() -> Coordinates:
    """Returns current coordinates using MacBook GPS"""
    process = Popen(["whereami"], stdout=PIPE)
    output, err = process.communicate()
    exit_code = process.wait()
    if err is not None or exit_code != 0:
        raise CantGetCoordinates
    output_lines = output.decode().strip().lower().split("\n")
    latitude = longitude = None
    for line in output_lines:
        if line.startswith("latitude:"):
            latitude = float(line.split()[1])
        if line.startswith("longitude:"):
            longitude = float(line.split()[1])
    if config.USE_ROUNDED_COORDS:  # Добавили округление координат
        latitude, longitude = map(lambda c: round(c, 1), [latitude,
longitude])
    return Coordinates(longitude=longitude, latitude=latitude)

if __name__ == "__main__":
    print(get_gps_coordinates())
```

Отлично. Обратите внимание — мы не полагаемся здесь на то, на какой строке будет значение широты и долготы в выдаче команды `whereami`. Мы ищем нужную строку во всех возвращаемых строках, не полагаясь на то, будут это первые строки или нет. Получается более надёжное решение на случай смены порядка строк в `whereami`.

Теперь проведём рефакторинг, поделив большую, делающую слишком много всего функцию `get_gps_coordinates` на несколько небольших простых функций:

```python
from dataclasses import dataclass
from subprocess import Popen, PIPE
from typing import Literal

import config
from exceptions import CantGetCoordinates

@dataclass(slots=True, frozen=True)
class Coordinates:
    latitude: float
    longitude: float

def get_gps_coordinates() -> Coordinates:
    """Returns current coordinates using MacBook GPS"""
    coordinates = _get_whereami_coordinates()
    return _round_coordinates(coordinates)

def _get_whereami_coordinates() -> Coordinates:
    whereami_output = _get_whereami_output()
    coordinates = _parse_coordinates(whereami_output)
    return coordinates

def _get_whereami_output() -> bytes:
    process = Popen(["whereami"], stdout=PIPE)
    output, err = process.communicate()
    exit_code = process.wait()
    if err is not None or exit_code != 0:
        raise CantGetCoordinates
    return output

def _parse_coordinates(whereami_output: bytes) -> Coordinates:
    try:
        output = whereami_output.decode().strip().lower().split("\n")
    except UnicodeDecodeError:
        raise CantGetCoordinates
    return Coordinates(
        latitude=_parse_coord(output, "latitude"),
        longitude=_parse_coord(output, "longitude")
    )

def _parse_coord(
        output: list[str],
        coord_type: Literal["latitude"] | Literal["longitude"]) -> float:
    for line in output:
        if line.startswith(f"{coord_type}:"):
            return _parse_float_coordinate(line.split()[1])
    else:
```

```
        raise CantGetCoordinates

def _parse_float_coordinate(value: str) -> float:
    try:
        return float(value)
    except ValueError:
        raise CantGetCoordinates

def _round_coordinates(coordinates: Coordinates) -> Coordinates:
    if not config.USE_ROUNDED_COORDS:
        return coordinates
    return Coordinates(*map(
        lambda c: round(c, 1),
        [coordinates.latitude, coordinates.longitude]
    ))


if __name__ == "__main__":
    print(get_gps_coordinates())
```

Кода стало больше, функций стало больше, но код стал проще читаться и будет проще сопровождаться. Если бы мы сейчас писали тесты, то убедились бы ещё и в том, что этот код легче обложить тестами, чем предыдущий вариант с одной большой функцией, делающей всё подряд.

Функции, имена которых начинаются с подчёркивания — не предназначены для вызова извне модуля, то есть они вызываются только соседними функциями модуля `get_gps_coordinates.py`.

Почему много коротких функций это лучше, чем одна большая функция? Потому что для того, чтобы понять, что происходит внутри функции на 50 строк, надо прочитать 50 строк. А если эти 50 строк разбить на пару меньших функций и понятным образом эти пару функций назвать, то нам понадобится прочесть всего пару строк с вызовами этой пары функций и всё. Прочесть пару строк легче, чем 50. А если нам нужны детали реализации какой-то из этих меньших функций, мы всегда можем в неё провалиться и посмотреть, что внутри.

Функция `get_gps_coordinates` тут максимально проста — она получает координаты и затем округляет их и возвращает, всё. Два вызова понятно названных функций вместо длинного сложного кода, как было раньше.

Также обратите внимание — абсолютно все функции типизированы, все принимаемые аргументы функций типизированы и все возвращаемые значения тоже типизированы. Причём типизированы максимально конкретными типами.

Эта логика реализована без классов, на обычных функциях. Это нормально. Не нужно использовать ООП просто для того, чтобы у вас были классы. От того, что мы обернём несколько описанных здесь функций в класс — никакого нового полезного качества в нашем коде не появится, просто вместо функций будет класс. В таком случае вовсе не нужно использовать классы.

Обратите внимание также, как в функции `_parse_float_coordinate` обработана ошибка `ValueError`, которая может возникать, если вдруг координаты не получается привести из строки к типу `float` — мы возбуждаем (райзим) исключение своего типа `CantGetCoordinates`. В любой ситуации, когда нам не удалось получить координаты из результатов команды `whereami` мы получаем такое исключение и можем обработать (или не обрабатывать) его в коде, который будет вызывать нашу верхнеуровневую функцию `get_gps_coordinates`. Про работу с исключениями более подробно мы поговорим в отдельном материале.

Реализация приложения — получение погоды с API OpenWeather

Отлично, у нас реализована структура и скелет приложения, а также полностью реализована логика получения текущих GPS координат — в точном или округлённом варианте. Реализуем теперь получение по этим координатам значения погоды с использованием API сервиса OpenWeather. Добавим шаблон URL для получения погоды в `config.py`:

```
USE_ROUNDED_COORDS = True
OPENWEATHER_API = "7549b3ff11a7b2f3cd25b56d21c83c6a"
OPENWEATHER_URL = (
    "https://api.openweathermap.org/data/2.5/weather?"
    "lat={latitude}&lon={longitude}&"
    "appid=" + OPENWEATHER_API + "&lang=ru&"
    "units=metric"
)
```

Значения широты и долготы будем потом подставлять в этот шаблон. Если нам понадобится изменить однажды этот шаблон URL для получения данных, мы сможем не искать его где-то глубоко в приложении, он лежит в конфиге. Все данные, которые предполагаются как конфигурационные, имеет смысл выносить в отдельное место, которое можно назвать конфигом или настройками приложения.

API ключ для сервиса OpenWeather лучше сохранить в переменной окружения и не хранить в исходном коде проекта (тогда значение константы будет получаться как-

то так: `os.getenv("OPENWEATHER_API_KEY")`, но сейчас мы этого делать не будем для упрощения запуска приложения.

Итак, реализация работы с сервисом погоды OpenWeather, `weather_api_service.py`:

```
from datetime import datetime
from dataclasses import dataclass
from enum import Enum
import json
from json.decoder import JSONDecodeError
import ssl
from typing import Literal
import urllib.request
from urllib.error import URLError

from coordinates import Coordinates
import config
from exceptions import ApiServiceError

Celsius = float

class WeatherType(str, Enum):
    THUNDERSTORM = "Гроза"
    DRIZZLE = "Изморось"
    RAIN = "Дождь"
    SNOW = "Снег"
    CLEAR = "Ясно"
    FOG = "Туман"
    CLOUDS = "Облачно"

@dataclass(slots=True, frozen=True)
class Weather:
    temperature: Celsius
    weather_type: WeatherType
    sunrise: datetime
    sunset: datetime
    city: str

def get_weather(coordinates: Coordinates) -> Weather:
    """Requests weather in OpenWeather API and returns it"""
    openweather_response = _get_openweather_response(
        longitude=coordinates.longitude, latitude=coordinates.latitude)
    weather = _parse_openweather_response(openweather_response)
    return weather

def _get_openweather_response(latitude: float, longitude: float) -> str:
    ssl._create_default_https_context = ssl._create_unverified_context
```

```
    url = config.OPENWEATHER_URL.format(
        latitude=latitude, longitude=longitude)
    try:
        return urllib.request.urlopen(url).read()
    except URLError:
        raise ApiServiceError

def _parse_openweather_response(openweather_response: str) -> Weather:
    try:
        openweather_dict = json.loads(openweather_response)
    except JSONDecodeError:
        raise ApiServiceError
    return Weather(
        temperature=_parse_temperature(openweather_dict),
        weather_type=_parse_weather_type(openweather_dict),
        sunrise=_parse_sun_time(openweather_dict, "sunrise"),
        sunset=_parse_sun_time(openweather_dict, "sunset"),
        city=_parse_city(openweather_dict)
    )

def _parse_temperature(openweather_dict: dict) -> Celsius:
    return round(openweather_dict["main"]["temp"])

def _parse_weather_type(openweather_dict: dict) -> WeatherType:
    try:
        weather_type_id = str(openweather_dict["weather"][0]["id"])
    except (IndexError, KeyError):
        raise ApiServiceError
    weather_types = {
        "1": WeatherType.THUNDERSTORM,
        "3": WeatherType.DRIZZLE,
        "5": WeatherType.RAIN,
        "6": WeatherType.SNOW,
        "7": WeatherType.FOG,
        "800": WeatherType.CLEAR,
        "80": WeatherType.CLOUDS
    }
    for _id, _weather_type in weather_types.items():
        if weather_type_id.startswith(_id):
            return _weather_type
    raise ApiServiceError

def _parse_sun_time(
        openweather_dict: dict,
        time: Literal["sunrise"] | Literal["sunset"]) -> datetime:
    return datetime.fromtimestamp(openweather_dict["sys"][time])

def _parse_city(openweather_dict: dict) -> str:
```

```
    return openweather_dict["name"]

if __name__ == "__main__":
    print(get_weather(Coordinates(latitude=55.7, longitude=37.6)))
```

Как и ранее, следуем подходу небольших функций, каждая из которых делает одно небольшое действие, а общий результат достигается за счёт компоновки этих небольших функций. Логику парсинга каждой нужной нам единицы информации выносим в отдельные небольшие функции — отдельно парсинг температуры, отдельно парсинг типа погоды и времени восхода и заката. Каждая функция названа глагольным словом — получить, распарсить и тд. Напомню, что функция это ни что иное как именованный блок кода, этот блок кода что-то делает и потому его имеет смысл называть именно глаголом, который опишет это действие.

Тут стоит отметить, что для парсинга и одновременно валидации JSON данных удобно использовать библиотеку Pydantic. О ней было видео на канале «Диджитализируй!». Здесь мы не стали её использовать из-за возможно некоторой её избыточности для нашей простой задачи, а также чтобы ограничиться только стандартной библиотекой Python.

Осталось реализовать «принтер», который выведет нужные нам значения погоды в консоль!

Реализация приложения — принтер погоды

Итак, файл `weather_formatter.py`:

```
from weather_api_service import Weather

def format_weather(weather: Weather) -> str:
    """Formats weather data in string"""
    return (f"{weather.city}, температура {weather.temperature}°C, "
            f"{weather.weather_type}\n"
            f"Восход: {weather.sunrise.strftime('%H:%M')}\n"
            f"Закат: {weather.sunset.strftime('%H:%M')}\n")

if __name__ == "__main__":
    from datetime import datetime
    from weather_api_service import WeatherType
    print(format_weather(Weather(
        temperature=25,
        weather_type=WeatherType.CLEAR,
        sunrise=datetime.fromisoformat("2022-05-03 04:00:00"),
        sunset=datetime.fromisoformat("2022-05-03 20:25:00"),
```

```
            city="Moscow"
        )))
```

Обратите внимание на печать типа погоды — `weather.weather_type`. Так можно, потому что мы отнаследовали `WeatherType` от `str` и `Enum`, а не только от `Enum`. Если бы мы отнаследовали `WeatherType` только от `Enum`, то для получения строкового значения нужно было бы напрямую обратиться к атрибуту `value`, вот так: `weather.weather_type.value` .

При необходимости выводить на печать значения как-то иначе, всегда можно это реализовать в одном месте приложения. Как всегда обратите внимание, здесь реализован блок `if __name__ == "__main__":`, который позволяет тестировать код при непосредственно прямом вызове этого файла `python3.10 weather_formatter.py`. При импорте функции `format_weather` код в этом блоке выполнен не будет.

Обработка исключений

В процессе работы приложения могут возникать 2 вида исключений, которые мы заложили в приложении — что-то может пойти не так с `whereami`, через который мы получаем текущие GPS координаты. Его может не быть в системе или по какой-то причине он может выдать результат не того формата, что мы ожидаем. В таком случае возбуждается исключение `CantGetCoordinates`.

Также что-то может пойти не так при запросе погоды по координатам. Тогда возбуждается исключение `ApiServiceError`. Обработаем и его. Файл `weather`:

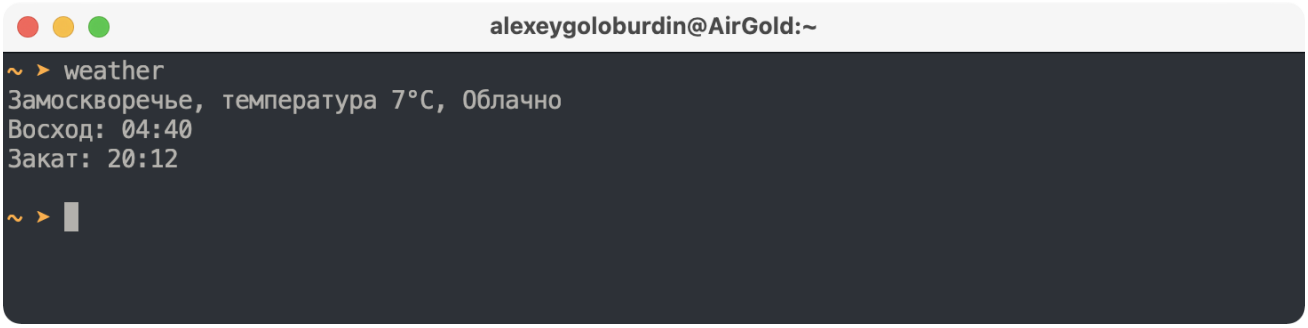
```python
#!/usr/bin/env python3.10
from exceptions import ApiServiceError, CantGetCoordinates
from coordinates import get_gps_coordinates
from weather_api_service import get_weather
from weather_formatter import format_weather


def main():
    try:
        coordinates = get_gps_coordinates()
    except CantGetCoordinates:
        print("Не смог получить GPS координаты")
        exit(1)
    try:
        weather = get_weather(coordinates)
    except ApiServiceError:
        print("Не смог получить погоду в API сервиса погоды")
        exit(1)
    print(format_weather(weather))


if __name__ == "__main__":
    main()
```

Проверяем работу приложения

Всё готово, вжух! Проверяем работу приложения:



Отлично!

Использование интерфейсов и протоколов

В теории объектно-ориентированного программирования есть понятия интерфейсов и абстрактных классов. Эти классы созданы для того, чтобы быть отнаследованными в других классах. Интерфейс и абстрактный класс созданы для того, чтобы показать, какими свойствами и методами должны обладать все их дочерние классы. Разница интерфейса и абстрактного класса в том, что интерфейс не содержит реализации, а абстрактный класс может помимо абстрактных методов содержать и часть реализованных методов.

Использование интерфейсов и абстрактных классов — хорошая затея, если мы хотим заложить на будущее возможность замены компонентов системы на другие. Расширяемость системы это хорошо.

Например, допустим, мы хотим реализовать сохранение истории всех запросов погоды. Чтобы при каждом запуске нашей программы куда-то сохранялись её результаты, и в будущем можно было проанализировать эту информацию.

Куда мы можем сохранить эту информацию? В плоский txt файл. В файл JSON. В базу данных SQL. В NoSQL базу данных. Отправить куда-то по сети в какой-то веб-сервис. Вариантов много и потенциально в будущем возможно нам захочется заменить текущий выбранный вариант на какой-то другой. Давайте реализуем модуль `history.py`, который будет отвечать за сохранение истории:

```python
from weather_api_service import Weather

class WeatherStorage:
    """Interface for any storage saving weather"""
    def save(self, weather: Weather) -> None:
        raise NotImplementedError

def save_weather(weather: Weather, storage: WeatherStorage) -> None:
    """Saves weather in the storage"""
    storage.save(weather)
```

Здесь `WeatherStorage` — это интерфейс в терминах объектно-ориентированного программирования, этот интерфейс который описывает те методы, которые обязательно должны присутствовать у любого хранилища погоды. Собственно говоря, у любого хранилища погоды должен быть как минимум метод `save`, который принимает на вход погоду, которую он должен сохранить.

В интерфейсе `WeatherStorage` нет реализации (на то он и интерфейс), он только объявляет метод `save`, который должен быть определён в любом классе, реализующем этот интерфейс.

Функция `save_weather` будет вызываться более высокоуровневым управляющим кодом для сохранения погоды в хранилище. Эта функция принимает на вход погоду `weather`, которую надо сохранить, и реальный экземпляр хранилища `storage`, которое реализует интерфейс `WeatherStorage`.

Чтобы показать, что метод `save` интерфейса не реализован, мы возбуждаем в нём исключение `NotImplementedError`, эта ошибка говорит о том, что вызываемый метод не реализован. Таким образом, если мы создадим хранилище, отнаследованное от этого интерфейса, не реализуем в нём метод `save` и вызовем его, то у нас упадёт в рантайме исключение `NotImplementedError`:

```python
class PlainFileWeatherStorage(WeatherStorage):
    pass

srorage = PlainFileWeatherStorage()
storage.save()  # Тут в runtime упадёт ошибка NotImplementedError
```

Проблема такого подхода в том, что ошибка, относящаяся к проверке типов (все ли методы интерфейса реализованы в наследующем его классе) падает только в рантайме. Хотелось бы, чтобы такая проверка выполнялась в IDE и статическим

анализатором кода, а не падала в рантайме. Наша задача, напомню, сделать так, чтобы до рантайма ошибки не доходили.

Какой есть ещё вариант определения интерфейсов в Python? Есть вариант с использованием встроенного модуля ABC (документация), созданного как раз для работы с такими абстрактными классами и интерфейсами:

```
from abc import ABC, abstractmethod

class WeatherStorage(ABC):
    """Interface for any storage saving weather"""
    @abstractmethod
    def save(self, weather: Weather) -> None:
        pass
```

Экземпляр класса, наследующего таким образом объявленный интерфейс, не получится создать без явной реализации всех методов, объявленных с декоратором `@abstractmethod`. То есть вот такой код в runtime упадёт сразу в момент создания экземпляра такого класса:

```
class PlainFileWeatherStorage(WeatherStorage):
    pass

# Тут упадет ошибка в рантайме, так как в PlainFileWeatherStorage
# не определен метод save
storage = PlainFileWeatherStorage()
```

Опять же — код падает в runtime, пользователи видят ошибку, плохо. Как перенести проверку на корректность использования интерфейсов и абстрактных классов на IDE и статический анализатор кода?

Способ появился в Python 3.8 благодаря PEP 544, и он называется протоколами, `Protocol`:

```
from typing import protocol

class WeatherStorage(Protocol):
    """Interface for any storage saving weather"""
    def save(self, weather: Weather) -> None:
        pass

class PlainFileWeatherStorage:
    def save(self, weather: Weather) -> None:
        print("реализация сохранения погоды...")

def save_weather(weather: Weather, storage: WeatherStorage) -> None:
    """Saves weather in the storage"""
    storage.save(weather)
```

Воу! Класс `PlainFileWeatherStorage` никак не связан с `WeatherStorage`, не отнаследован от него, хотя и реализует его интерфейс в неявном виде, то есть просто определяет все функции, которые должны быть реализованы в этом интерфейсе. Сам интерфейс `WeatherStorage` отнаследован от класса `typing.Protocol`, что делает его так называемым протоколом. В функции `save_weather` тип аргумента `storage` по-прежнему установлен в этот интерфейс `WeatherStorage`.

Получается, что класс `PlainFileWeatherStorage` неявно реализует протокол/интерфейс `WeatherStorage`. Если вы работали с языком программирования Go — в нём интерфейсы реализованы схожим образом, это так называемая структурная типизация.

Почему использование такого подхода в приоритете? Потому что проверкой корректности использования интерфейсов занимается IDE и статический анализатор кода вроде `mypy`. Речь идёт уже не о проверке в runtime, речь идет о проверке корректности реализации до этапа, в котором участвуют пользователи программы. Это то, что нам нужно!

Таким образом, наш модуль `history.py` принимает следующий вид:

```python
from datetime import datetime
from pathlib import Path
from typine import Protocol

from weather_api_service import Weather
from weather_formatter import format_weather

class WeatherStorage(Protocol):
    """Interface for any storage saving weather"""
    def save(self, weather: Weather) -> None:
        raise NotImplementedError

class PlainFileWeatherStorage:
    """Store weather in plain text file"""
    def __init__(self, file: Path):
        self._file = file

    def save(self, weather: Weather) -> None:
        now = datetime.now()
        formatted_weather = format_weather(weather)
        with open(self._file, "a") as f:
            f.write(f"{now}\n{formatted_weather}\n")

def save_weather(weather: Weather, storage: WeatherStorage) -> None:
    """Saves weather in the storage"""
    storage.save(weather)
```

`PlainFileWeatherStorage` это реализованное хранилище, отнаследованное от нашего интерфейса, то есть реализующее его методы. Помимо метода `save` этот класс реализует ещё конструктор, который сохраняет в поле `self._file` путь до файла, в который будет записываться информация о погоде.

Для перевода объекта погоды типа `Weather` в строку используется функция `format_weather`, которую мы реализовали ранее в модуле `weather_formatter`.

Этот код — абсолютно валиден с точки зрения проверки системы типов.

Вызовем теперь логику сохранения погоды в главном файле `weather`:

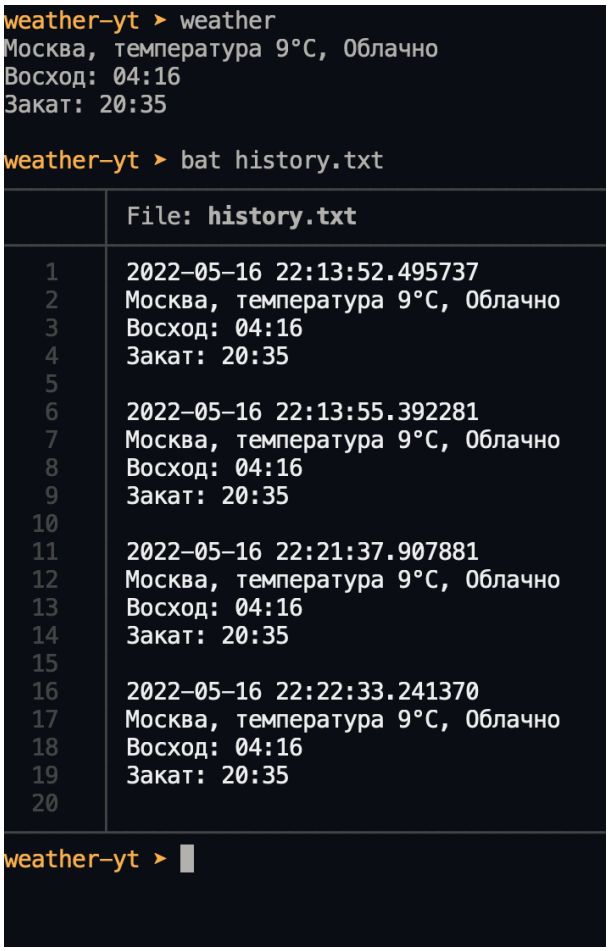
```
#!/usr/bin/env python3.10
from pathlib import Path

from exceptions import ApiServiceError, CantGetCoordinates
from coordinates import get_gps_coordinates
from history import PlainFileWeatherStorage, save_weather
from weather_api_service import get_weather
from weather_formatter import format_weather


def main():
    try:
        coordinates = get_gps_coordinates()
    except CantGetCoordinates:
        print("Не смог получить GPS координаты")
        exit(1)
    try:
        weather = get_weather(coordinates)
    except ApiServiceError:
        print("Не смог получить погоду в API сервиса погоды")
        exit(1)
    save_weather(
        weather,
        PlainFileWeatherStorage(Path.cwd() / "history.txt")
    )
    print(format_weather(weather))


if __name__ == "__main__":
    main()
```

Здесь мы создаём экземпляр объекта `PlainFileWeatherStorage` и передаём его на вход функции `save_weather`. Всё работает!



Для вывода содержимого текстового файла на скриншоте вместо `cat` использовался `bat` — продвинутый вариант:)

Теперь, если мы захотим изменить хранилище, мы можем создать новое хранилище, например, JSON хранилище, реализовав в нём все методы интерфейса `WeatherStorage`, и передать это новое хранилище в `save_weather`. Всё продолжит работать и будет корректно с точки зрения типов. Причём нам не придётся ничего менять в функции `save_weather`, так как она опирается только на интерфейс, определённый в классе `WeatherStorage`.

`history.py`, добавленный код:

```
import json
from typing import Protocol, TypedDict


class HistoryRecord(TypedDict):
    date: str
    weather: str

class JSONFileWeatherStorage:
    """Store weather in JSON file"""
    def __init__(self, jsonfile: Path):
        self._jsonfile = jsonfile
        self._init_storage()

    def save(self, weather: Weather) -> None:
        history = self._read_history()
        history.append({
            "date": str(datetime.now()),
            "weather": format_weather(weather)
        })
        self._write(history)

    def _init_storage(self) -> None:
        if not self._jsonfile.exists():
            self._jsonfile.write_text("[]")

    def _read_history(self) -> list[HistoryRecord]:
        with open(self._jsonfile, "r") as f:
            return json.load(f)

    def _write(self, history: list[HistoryRecord]) -> None:
        with open(self._jsonfile, "w") as f:
            json.dump(history, f, ensure_ascii=False, indent=4)
```

Здесь мы воспользовались структурой `TypedDict`, типизированным словарём. Это удобно для нашего сценария, так как каждая запись погоды в JSON файл будет представлять собой как раз структуру словаря, состоящую из двух полей — date для `даты` и времени получения погоды и `weather` для описания погоды. Метод `_read_history` предназначен для чтения данных погоды из JSON файла и он возвращает не `list[dict]`, а `list[HistoryRecord]`, максимально конкретный тип данных. Аналогично метод `_write` принимает в качестве аргумента не `list[dict]`, а

тоже `list[HistoryRecord]`. Везде используем максимально точную конкретную структуру данных.

`weather`, изменённый код:

```python
from history import JSONFileWeatherStorage, save_weather


def main():
    # пропущено....
    save_weather(
        weather,
        JSONFileWeatherStorage(Path.cwd() / "history.json")
    )
    print(format_weather(weather))


if __name__ == "__main__":
    main()
```

Всё работает:

```
weather-yt ➤ bat history.json
       File: history.json
   1   [
   2       {
   3           "date": "2022-05-16 22:42:55.936445",
   4           "weather": "Москва, температура 8°C, Облачно\nВосход: 04:16\nЗакат: 20:35\n"
   5       },
   6       {
   7           "date": "2022-05-16 22:46:15.076943",
   8           "weather": "Москва, температура 8°C, Облачно\nВосход: 04:16\nЗакат: 20:35\n"
   9       },
  10       {
  11           "date": "2022-05-16 22:47:04.889282",
  12           "weather": "Москва, температура 8°C, Облачно\nВосход: 04:16\nЗакат: 20:35\n"
  13       },
  14       {
  15           "date": "2022-05-16 22:47:09.204298",
  16           "weather": "Москва, температура 8°C, Облачно\nВосход: 04:16\nЗакат: 20:35\n"
  17       }
  18   ]
weather-yt ➤
```

В процессе сохранения файла тоже может возникнуть ошибка. Например, директория может быть закрыта для записей и тд. Такие ошибки тоже нужно обработать. Напишите эту обработку самостоятельно в качестве тренировки!

Анализ получившейся архитектуры кода

Давайте посмотрим свежим взглядом на получившуюся архитектуру кода.

Мы имеем 4 слоя приложения:

1. Модуль `weather`, запускающий приложение и связывающий остальные слои. Важно обратить внимание: этот файл не содержит никакой логики реализации, никакой бизнес-логики. Это точка входа в приложение. Она не знает ничего о деталях реализации всех остальных нижележащих слоёв приложения.

2. Модуль `gps_coordinates` отвечает за получение координат из внешней команды `whereami`. Сюда инкапсулирована вся логика по работе с координатами. Эта логика ничего не знает о том, для чего эти координаты будут использованы затем в приложении. Модуль определяет структуру данных для хранения и передачи в приложение координат.

   Если нам понадобится получать координаты откуда-то иначе — мы перепишем логику этого модуля, никак не затронув все остальные модули приложения. Связь этого модуля с остальными — слабая, и это хорошо.

3. Модуль `weather_api_service` инкапсулирует в себе логику получения погоды по координатам. Он не знает, откуда были получены координаты, поступившие в этот модуль. Он не знает, как будет использоваться погода дальше в приложении. В этом модуле определена структура для хранения и передачи данных погоды в приложение.

   Если нам понадобится получать погоду в другом API сервисе — мы заменим логику этого модуля и это никак не затронет остальные модули приложения. Связь этого модуля с остальными — слабая, и это хорошо.

4. Модуль `weather_formatter` отвечает за преобразование погоды в строку. Он ничего не знает о том, откуда погода была получена, была ли она получена по координатам GPS или по названию населённого пункта или как-то иначе, он не знает ничего кроме того, как преобразовать данные погоды в строку, всё. Связь этого модуля с остальными слабая, и это хорошо. В любой момент мы можем изменить логику форматирования данных погоды (добавив в неё иконки погоды, например), никак не затронув при этом все остальные модули приложения.

5. Модуль `history` инкапсулирует в себе логику по сохранению истории погоды. Этот модуль также независим от остальных модулей. Более того, реализована

гибкая схема смены хранилища на любое другое через механизм интерфейсов. Ответственность за то, какое хранилище будет использовано для сохранения данных, лежит вовне этого модуля. Можно, например, данные погоды днём сохранять в текстовый плоский TXT файл, а данные ночной погоды — в JSON. Для этого не придётся ничего менять в самом модуле `history`. Чем меньше поводов менять код какого-то модуля, класса, функции — тем лучше.

Получается, что мы реализовали всё приложение в виде слабозависимых друг от друга модулей. При этом эти модули могут использоваться и в составе других приложений, они *reusable*, то есть переиспользуемые. Скажем, модуль получения GPS координат может использоваться в программе вычисления расстояния от текущей точки, где мы находимся, до Рима. Почему нет. Для этого не понадобится изменять этот модуль. Отлично!

Если мы откроем код любого модуля, любой функции — нам сразу станет понятно, какие данные принимаются на вход и какие возвращаются на выход, причём понятно максимально точно. Это:

1. Облегчает чтение кода — все типы данных в явном виде и максимально конкретно прописаны, не надо их предугадывать.
2. Гарантируется отсутствие ошибок использования типов — IDE подсветит, если мы что-то используем не так, как нужно; также на ошибки укажет инструмент статического анализа вроде `mypy`, о котором мы поговорим ниже.
3. IDE поможет писать код всем, использующим наши разработанные модули. Будет работать автодополнение по полям и методам классов с учетом типов, которые мы указали.

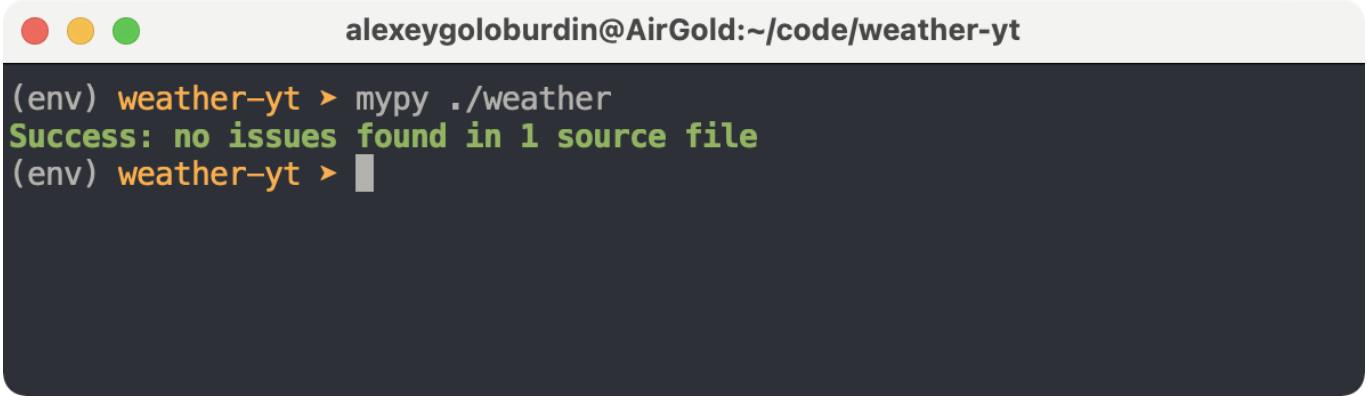
Финальный вариант исходного кода программы размещён на Github:
https://github.com/alexey-goloburdin/weather

Статический анализатор mypy

`mypy` это инструмент, который устанавливается отдельно как pip пакет и запускается в проекте как часть тестов или CI/CD процесса. Перед сборкой и раскаткой приложения на сервер запускается проверка исходного Python кода с `mypy` и если `mypy` находит ошибки, то процесс останавливается, разработчики исправляют найденные ошибки и процесс повторяется. Это приводит к тому, что до продакшн, то есть до рантайма и до живых пользователей соответственно ошибок долетает меньше, потому что многое выявляется на более ранних этапах.

В директории проекта создадим и активируем виртуальное окружение, установим в него `mypy` и запустим проверку нашего кода:

```
python3.10 -m venv env
. ./env/bin/activate
pip install mypy
mypy ./weather
```



Как видим, `mypy` не нашёл проблем в нашем коде. Внесём специально ошибку в код и убедимся, что `mypy` её найдёт:

Запуск `mypy` можно встроить в процесс CI/CD, чтобы процесс разворачивания приложения на серверах не запускался, если проверки `mypy` не прошли. Таким образом до runtime не смогут дойти ошибки, связанные с некорректным использованием типов данных, и это здорово — надёжность приложения значительно возрастает!

И ещё важно отметить, что используя mypy, вы можете проверять корректность своих тайп хинтингов, которые вы указали. Пока учишься могут быть вопросы, правильно ли указан тип — вот можно указать тип у параметра функции, вызвать эту функцию с данными и посмотреть, как поведёт себя проверятор типов, встроенный в IDE, и как поведёт себя `mypy`.

Ещё о практических аспектах типизации

Опциональные данные

Для указания опциональных данных можно пользоваться вертикальной чертой:

```
def print_hello(name: str | None=None) -> None:
    print(f"hello, {name}" if name is not None else "hello anon!")
```

Здесь параметр `name` функции `print_hello` является опциональным, что отражено а) в type hinting (напомню, вертикальная черта в подсказках типов означает ИЛИ) б) задано значение по умолчанию None.

Контейнеры — Iterable, Sequence, Mapping и другие

Как указать тип для контейнера с данными, например, для списка юзеров?

```
from datetime import datetime
from dataclasses import dataclass

@dataclass
class User:
    birthday: datetime

users = [
    User(birthday=datetime.fromisoformat("1988-01-01")),
    User(birthday=datetime.fromisoformat("1985-07-29")),
    User(birthday=datetime.fromisoformat("2000-10-10"))
]

def get_younger_user(users: list[User]) -> User:
    if not users: raise ValueError("empty users!")
    sorded_users = sorted(users, key=lambda x: x.birthday)
    return sorded_users[0]

print(get_younger_user(users))
# User(birthday=datetime.datetime(1985, 7, 29, 0, 0))
```

До python3.10 список для указания типа надо было импортировать из `typing`, но сейчас можно list не импортировать и просто сразу использовать, что удобно. То

есть Python продолжает движение в сторону ещё более простого и удобного использования подсказок типов.

Обратите внимание — технически можно указать просто `users: list`, но тогда IDE и статический анализатор кода вроде `mypy` не будут знать, что находится внутри этого списка, и это нехорошо. Мы же изначально знаем, что там именно тип данных `User`, объекты класса `User`, и, значит, это надо в явном виде указать.

Так, отлично, а давайте подумаем, а обязательно ли функция поиска самого молодого юзера должна принимать на вход именно список юзеров? Ведь по сути главное, чтобы просто можно было проитерироваться по пользователям. Может, мы захотим потом передать сюда не список пользователей, а кортеж с пользователями, или еще что-то? Если мы передадим вместо списка кортеж — будет ошибка типов сейчас:

```
from datetime import datetime
from dataclasses import dataclass

@dataclass
class User:
    birthday: datetime

users = (  # сменили на tuple
    User(birthday=datetime.fromisoformat("1988-01-01")),
    User(birthday=datetime.fromisoformat("1985-07-29")),
    User(birthday=datetime.fromisoformat("2000-10-10"))
)

def get_younger_user(users: list[User]) -> User:
    """Возвращает самого молодого пользователя из списка"""
    sorded_users = sorted(users, key=lambda x: x.birthday)
    return sorded_users[0]


print(get_younger_user(users))  # тут видна ошибка в pyright!
```

Код работает (повторимся, что интерпретатор не проверяет типы в type hinting), но проверка типов в редакторе (и `mypy`) ругается, это нехорошо.

Если мы посмотрим документацию по функции `sorted`, то увидим, что первый элемент там назван *iterable*, то есть итерируемый, то, по чему можно проитерироваться. То есть мы можем передать любую итерируемую структуру:

```python
from typing import Iterable

def get_younger_user(users: Iterable[User]) -> User | None:
    if not users: return None
    sorded_users = sorted(users, key=lambda x: x.birthday)
    return sorded_users[0]
```

И теперь всё в порядке. Мы можем передать любую итерируемую структуру, элементами которой являются экземпляры `User`.

А если нам надо обращаться внутри функции по индексу к элементам последовательности? Подойдёт ли `Iterable`? Нет, так как `Iterable` подразумевает возможность итерироваться по контейнеру, то есть обходить его в цикле, но это не предполагает обязательной возможности обращаться по индексу. Для этого есть `Sequence`:

```python
from typing import Sequence

def get_younger_user(users: Sequence[User]) -> User | None:
    """Возвращает самого молодого пользователя из списка"""
    if not users: return None
    print(users[0])
    sorded_users = sorted(users, key=lambda x: x.birthday)
    return sorded_users[0]
```

Теперь всё в порядке. В `Sequence` можно обращаться к элементам по индексу.

Ещё один важный вопрос тут. А зачем использовать `Iterable` или `Sequence`, если можно просто перечислить разные типы контейнеров? Ну их же ограниченное количество — там `list`, `tuple`, `set`, `dict`. Для чего нам тогда общие типы `Iterable` и `Sequence`?

На самом деле таких типов контейнеров, по которым можно итерироваться, вовсе не ограниченное число. Например, можно создать свой контейнер, по которому можно будет итерироваться, но при этом этот тип не будет наследовать ничего из вышеперечисленного типа `list`, `dict` и тп:

```python
from typing import Sequence

class Users:
    def __init__(self, users: Sequence[User]):
        self._users = users

    def __getitem__(self, key: int) -> User:
        return self._users[key]

users = Users((  # сменили на tuple
    User(birthday=datetime.fromisoformat("1988-01-01")),
    User(birthday=datetime.fromisoformat("1985-07-29")),
    User(birthday=datetime.fromisoformat("2000-10-10"))
))

for u in users:
    print(u)
```

Способов создать такую структуру, по которой можно итерироваться или обращаться по индексам, в Python много, это один из способов. Важно просто понимать, что если вам надо показать структуру, по которой, например, можно итерироваться, то не стоит ограничивать набор таких структур простым перечислением списка, кортежа и чего-то ещё. Используйте обобщённые типы, созданные специально для этого, например, `Iterable` или `Sequence`, потому что они покроют действительно всё, в том числе и свои кастомные (самописные) реализации контейнеров.

Ну и напоследок — как определить тип словаря, ключами которого являются строки, а значениями, например, объекты типа `User`:

```python
some_users_dict: dict[str, User] = {
    "alex": User(birthday=datetime.fromisoformat("1990-01-01")),
    "petr": User(birthday=datetime.fromisoformat("1988-10-23"))
}
```

И также, если нет смысла ограничиваться именно словарём и подойдёт любая структура, к которой можно обращаться по ключам — то есть обобщённый тип `Mapping`:

```
from typing import Mapping

def smth(some_users: Mapping[str, User]) -> None:
    print(some_users["alex"])

smth({
    "alex": User(birthday=datetime.fromisoformat("1990-01-01")),
    "petr": User(birthday=datetime.fromisoformat("1988-10-23"))
})
```

Пару слов стоит сказать про кортежи, если размер кортежа важен и мы хотим его прямо указать в типе, то это можно сделать так:

```
three_ints = tuple[int, int, int]
```

Если количество элементов неизвестно — можно так:

```
tuple_ints = tuple[int, ...]
```

Дженерики

Что если мы хотим написать обобщённую функцию, которая принимает на вход итерируемую структуру, то есть структуру, по которой можно итерироваться, и возвращает результат первой итерации?

```
from typing import TypeVar, Iterable

T = TypeVar("T")

def first(iterable: Iterable[T]) -> T | None:
    for element in iterable:
        return element

print(first(["one", "two"]))  # one
print(first((100, 200)))  # 200
```

Как видите, типом данных в этой итерируемой структуре `iterable` могут быть любые данные, а наши type hinting в функции `first` говорят буквально, что функция

принимает на вход итерабельную структуру данных, каждый элемент которой имеет тип `T`, и функция возвращает тот же тип `T`. Тип `T` при этом может быть любым.

Это так называемые дженерики, то есть обобщённые типы.

Причём имя `T` здесь это пример, он часто используется именно так, `T`, от *Type*, но название типа может быть и любым другим.

Помимо дженериков можно сохранять отдельные типы для лучшей читаемости кода и подсказки читателю, что именно за данные здесь хранятся:

```
from dataclasses import dataclass

Phone = str

@dataclass
class User:
    user_id: int
    phone: Phone

def get_user_phone(user: User) -> Phone:
    return user.phone
```

Мы уже использовали это в коде приложения погоды, когда задавали псевдоним для города и градусов Цельсия.

Вызываемые объекты

Как известно функции в Python это обычные объекты, которые можно передавать в другие функции, возвращать из других функций и тп, поэтому для них тоже есть свой тип `Callable`:

```
from typing import Callable

def mysum(a: int, b: int) -> int:
    return a + b


def process_operation(operation: Callable[[int, int], int],
                      a: int, b: int) -> int:
    return operation(a, b)

print(process_operation(mysum, 1, 5))  # 6
```

Здесь для аргумента `operation` функции `process_operation` проставлен тип `Callable[[int, int], int]`. Здесь `[int, int]` — это типы аргументов функции `operation`, получается, что у этой функции должно быть два аргумента и они оба должны иметь тип `int`. Последний `int` в определении типа `Callable[[int, int], int]` обозначает тип возвращаемого функцией значения.

Stub файлы и работа с нетипизированными библиотеками

Важно понимать, что type hinting работает не только для аргументов функций и возвращаемых значений. Мы можем просто создать переменную и указать ей тип:

```
book: str = "Тополек мой в красной косынке"
```

В таком сценарии это избыточно — IDE и статический анализатор кода и так видят, что в переменной `book` хранится значение типа `str`. Однако в таком сценарии:

```
book: str = find_book_in_library("Тополек мой в красной косынке")
```

функция поиска книги `find_book_in_library` может быть не нашей функцией, а функцией какой-то внешней библиотеки, которая не использует подсказки типов. То есть для функции может быть не проставлен тип возвращаемого значения. Чтобы IDE и статический анализатор знали, что тип данных, который будет храниться в `book`, это именно `str`, можно таким образом подсказать инструментам о верном типе. Иногда это бывает очень полезно, когда библиотека не использует подсказки типов, а возвращаемый тип данных какой-то сложный и мы хотим, чтобы IDE и mypy нам помогали анализировать наш код и типы.

В то же время в Python существует механизм так называемых стаб-файлов, которые позволяют типизировать в том числе внешние библиотеки. Например, для django есть пакет в pip, который называется `django-stubs`. О стаб-файлах есть видео на канале Диджитализируй!.

Если вы используете нетипизированную библиотеку — можно поискать готовые стаб файлы для неё, чтобы воспользоваться преимуществами типизированного Python.

Подсказки типов нужны только в функциях?

В чате Telegram канала задали отличный вопрос — подсказки типов имеет смысл ставить только для аргументов функций и возвращаемых значений или вообще для всех переменных?

И действительно. Как лучше?

В большинстве сценариев подсказок типов достаточно только для аргументов и результатов функций. Если в нашем коде все функции типизированы таким образом, то получается, что IDE и статический анализатор кода понимают тип любой переменной в коде и могут выполнять все проверки.

Объявляя переменную, мы либо задаём её значение в явном виде (и тогда тип переменной равен типу значения):

```
age = 33
user = User(username="Иннокентий")
```

Здесь тип переменной `age` равен типу значения `33`, то есть `int`, а тип переменной `user` равен `User`.

Второй вариант создания переменной — присваивание ей значения, которое возвращается функцией:

```
user = get_user_by_username("Иннокентий")
```

Если все функции в нашем коде типизированы, в том числе и функция `get_user_by_username`, то и в таких сценариях тип переменной очевиден. Какой тип данных функция возвращает, такой тип данных у переменной и будет.

Получается, что если все функции, используемые в коде, типизированы, то как правило нет смысла проставлять типы для обычных переменных.

Однако, иногда бывает так, что мы используем внешнюю библиотеку, функции которой нетипизированы. Если функция `get_user_by_username` в примере выше это функция внешней библиотеки и она нетипизирована, то IDE и статический анализатор кода не знают, какой тип данных вернёт эта функция и потому не знают, какой тип будет у переменной `user`. Тогда можно подсказать инструментам, явно указав тип:

```
user: User = get_user_by_username("Иннокентий")
```

Теперь IDE и статический анализатор будут знать тип переменной и смогут выполнять все проверки. Отлично!

Ещё один сценарий, при котором полезно задать переменной тип — когда мы инициализируем переменную пустым значением, но хотим указать, данные какого конкретно типа там будут.

Например, в конструкторе класса мы инициализируем атрибут с начальным значением `{}`, то есть пустой словарь, но указываем, что в этом словаре ключами будут строки, значениями числа. То есть мы уточняем тип данных, сужаем его с просто словаря до словаря с конкретным типом ключей и значений:

```
class SomeClass:
    def __init__(self):
        self._some_dict: dict[str, int] = {}

    def some_method(self):
        self._some_dict["some_key"] = 123  # Всё ок по типам
        self._some_dict[123] = "some_key"  # Ошибка типов!
```

Резюме

Грамотное использование type hinting и осознанный выбор классов отделяет код новичка от кода растущего профессионала. Пользуйся подсказками типов, продумывай структуру твоего приложения и используемые типы данных, и тогда твои решения будут красивыми, приятно читаемыми, легко поддерживаемыми и надёжными.

Ты дочитал досюда и разобрался с материалом? Отлично, молодец! Думаю, тебе стоит прислать нам резюме на почту: join@to.digital

Контакты

У нас много хороших материалов в YouTube, если вдруг ты ещё не подписан — YouTube канал Диджитализируй!

Много оперативной и текстовой полезной информации — в Telegram канале!

Также, конечно, есть VK группа и Дзен.

Об образовательной программе Диджитализируй!

Сейчас готовится к перезапуску курс Основы компьютерных и веб-технологий с Python. Курс — отличный, на Stepik Awards был признан лучший платным курсом 2021. Подписывайся, чтобы не пропустить старт. Новое издание курса будет на 30% больше и полезнее!

А если ты читаешь это позднее июня 2022, то вполне вероятно, что курс уже вышел. Беги по ссылке!