Оглавление

1.1 - Базовая аннотация типов данных

Библиотека **mypy**(для проверки корректности типизации в коде):

```
# Установка
pip install mypy

# Проверка нашего файла в корне проекта:
mypy .\имя файла
```

Аннотация переменной(редко юзается):

```
x: int
x = 'gh'

# или

x: int = 'gh'
```

Пример типизации параметра и возвращаемого объекта в функции:

```
def func(n: int) -> int:  # n: int = 5 - для именнованного параметра
    return sum(range(1, n + 1))
```

У функций при аннотации появляется магический атрибут **__annotations__**, который по сути является словарем, в котором указаны все аннотированные параметры и **return**:

```
def func(n: int = 5) -> int:
    return sum(range(1, n + 1))

print(func.__annotations__)
```

Если функция не возвращающая, то для **return** указываем **None** :

```
def func() -> None:
    print('Буги-Вуги')
```

Если наша функция подразумевает прием или возврат разных типов, для этого используется модуль **typing**, оттуда импортируется тип **Union**:

```
from typing import Union


def func(x: Union[int, float], y: Union[int, float]) -> Union[int, float]:
    return x + y
```

Начиная с версии Python 3.10 необходимость этого импорта отпала, теперь объединять типы можно так:

```
def func(x: int | float, y: int | float) -> int | float:
    return x + y
```

Типы аннотаций можно сохранять в переменные, обычно это делается когда аннотация немного сложнее чем обычно:

```
from typing import Union

z = Union[int, float]
al = int | float


def func(x: z, y: al) -> int:
    return x + y
```

Если параметр или возвращаемое значение подразумевает какой-то тип или **None**, то используется тип **Optional** из модуля **typing**:

```
from typing import Optional


def func(x: list, y: list) -> Optional[list]: # list | None
    match x, y:
        case list(), list():
            return x + y
    return None # здесь без явного возврата None муру ругается
```

Соответственно можно использовать **Union[list, None]**, но это менее удобный вариант, поэтому и собственно был придуман
Optional. Но вполне себе удобный вариант => **list | None.**

Если мы не можем определиться какой тип данных необходимо принять функции или вернуть, то в таком случае используется
тип **Any** из модуля **typing**:

```
from typing import Any


def func(*args: Any) -> list:
    return list(args)
```

Если вы хотите создать переменную-константу, значение которой менять нельзя можно использовать тип **Final** модуля **typing**:

```
from typing import Final

password: Final = 'qwerty'
password = 'пароль'
```

При аннотировании списков мы можем указать какие-типы данных могут быть внутри:

```
lst: list[int | str] = [1, 2, 3, '4']
```

До версии Python 3.9 коллекции также импортировались из модуля **typing** как типы: **List, Tuple, Dict, Set**. Поэтому этот код можно переписать так:

```
from typing import List

lst: List[int | str] = [1, 2, 3, '4']
```

Для кортежей мы должны указывать тип для каждого его элемента, почему так? Дело в том, что те же списки обычно используются для объединения объектов одного типа, а кортежи для разных - отсюда и различия:

```
tpl_1: tuple[int, str] = (1, 'Альфа')
```

Но если все же у нас будут объекты одного типа, то это можно записать так:

```
tpl_2: tuple[int, ...] = (1, 2, 3)
```

Если мы хотим аннотировать словари, то в квадратных скобках нужно указать какие типы данных мы ожидаем от ключей и значений, через запятую:

```
dct: dict[int, str] = {1: 'Аня', 2: 'Никита'}
```

Для множеств также как и для списков указывают один тип для всех элементов:

```
st: set[int] = {1, 3, 5, 9}
```

Пример вложенности аннотаций:

```
from typing import Any


def func(data: list[str | int] | dict[int, Any]) -> set[int |
str]:
    return set(data)
```

Если необходимо указать конкретное значение типа, то нужно использовать **Literal**:

```
from typing import Literal


def func(tpl: tuple[int, Literal[5]]) -> None:
    pass


print(func((3, 5)))
```

Если мы хотим использовать вызываемый тип данных, например функция или метод, то используется тип **Callable** модуля **collections.abc**, в нем в **[]** мы можем первым аргументом указать параметр(тоже в **[]**), затем возвращаемый тип(без **[]**):

```
from collections.abc import Callable

def func(f: Callable[[str], int], lst: list[str]) -> list[int]:
    if lst is None:
        return []
    return [f(i) for i in lst]
```

```
print(func(len, ['Abc', 'Bcdef']))
```

2.1 - Области видимости переменных

В примере ниже переменные **args**, **summ**, **i** являются локальными, их областью видимости является тело функции:

```
def func(*args):
    summ = 0
    for i in args:
        summ += i
    return summ
```

Здесь мы добавили переменную **x** и немного изменили функцию, функция из локальной области видимости использует значение глобальной переменной, т.к. в своей области имен одноименная переменная не объявлялась. Т.е. в любой вложенной области имен, если есть запрос на переменную и она в данной области отсутствует, то происходит поиск данной переменной областью выше:

```
x = 100


def func(*args):
    summ = 0
    for i in args:
        summ += i
    return summ + x


print(func(5, 5))
```

Если одноименная переменная существует во всех областях видимости, то берется значение из той же области в которой происходит запрос на данную переменную:

```
x = 100


def func(*args):
    summ = 0
    x = 20
    for i in args:
        summ += i
    return summ + x


print(func(5, 5))
```

Если вы все же хотите изменять значения переменных глобальной области из вложенной области, то нужно использовать ключевое слово **global**:

```
x = 100
y = 500


def func(*args):
    global x, y # здесь можно перечислять переменные
    x = 20
    summ = 0
    for i in args:
        summ += i
    return summ + x


print(func(5, 5), x)
```

Пример в котором показано уже 3 области видимости, в каждой из них есть переменная **x** и все они независимы:

```
x = 1


def func_1():
    x = 2

    def func_2():
        x = 3
        print('x_func_2 =', x)
    func_2()
    print('x_func_1 =', x)


func_1()
print('global x =', x)
```

Если наша задача влиять из локальной области имен на переменные локальной области уровнем выше, то необходимо использовать ключевое слово **nonlocal**:

```
x = 1


def func_1():
    x = 2

    def func_2():
        nonlocal x
        x = 3
        print('x_func_2 =', x)
    func_2()
    print('x_func_1 =', x)


func_1()
print('global x =', x)
```

Из локальной области можно влиять на глобальные переменные и без **global**. Если внести изменения на изменяемый объект, например список. Т.к. в данном случае не происходит замена объекта как такового, происходит только его изменение:

```
lst = [1, 2, 3]


def func():
    lst.append(4)


func()
print(lst)
```

2.2 - lambda-функции

Простой пример **lambda** функции:

```
x = lambda a, b: a + b
print(x(3, 7))
```

где **a**, **b** - параметры функции
a + b - **return**

lambda функция может быть без аргументов, но как правило так она не используется. Самый простой вариант:

```
print((lambda: 5)())
```

Пример функции, которая принимает необязательным аргументом **lambda** функцию, которая выполняет роль фильтра:

```
def func(lst, filtr=None):
    if filtr is None:
        return lst
    new_lst = [i for i in lst if filtr(i)]
    return new_lst


print(func([1, 2, 3, 4, 5, 12, 66], filtr=lambda x: 9 < x < 100))
# x % 2 == 0
```

В **lambda** функции можно использовать моржовый оператор:

```
z = lambda x: x + (y := 5)
print(z(8))
```

lambda функции также поддерживают все виды аргументов, что и обычные функции:

```
f1 = lambda x, y, z: x + y + z
f2 = lambda x, y, z=3: x + y + z
f3 = lambda *args: sum(args)
```

```
f4 = lambda **kwargs: sum(kwargs.values())
```

2.3 - Замыкание функций

В примере ниже представлено как внешняя функция объявляет и возвращает внутреннюю и когда казалось бы внешняя функция выполнила свою работу и больше никак не связана с внутренней, всё равно возвращенная функция пользуется пространством имен внешней:

```
def func_1(name):
    def func_2():
        print(f'Привет, {name}!')
    return func_2


x = func_1('Артем')
x()
```

Это поведение называется замыканием функций. Пространство имен внешней функции существует пока к нему сохраняется хоть одна ссылка. Цепочка вызовов: переменная **x** -> вложенная функция -> пространство имен внешней функции. Пока вложенная функция пользуется пространством имен внешней функции(переменная **name**) до тех пор существует связь между внешней и вложенной функцией, даже если внешняя функция отработала и вернула внутреннюю.

Пример в котором пространство имен внешней функции является своего рода буфером для внутренней функции. Ниже представлена функция счетчика:

```
def count(number):
    def step():
        nonlocal number
        number += 1
        return number
    return step


x = count(5)
print(x(), x(), x())
```

Здесь переменная **number** "паркуется" во внешнем пространстве имен, а внутренняя функция имеет доступ к ней с возможностью изменения при помощи **nonlocal**. Казалось бы мы пользуемся результатом внешней функции, т.е. функцией **step**, но как и было сказано некое пространство имен еще существует уровнем выше и оттуда и вытаскиваются необходимые данные.

Пример создания кэша максимум до **10** элементов во внешней области в виде списка:

```
def cash_lst():
    lst = []
```

```
    def add_cash(obj=None):
        if obj is None:
            return lst
        (lst.pop(0), lst.append(obj)) if len(lst) == 10 else
lst.append(obj)
        return lst

    return add_cash


x = cash_lst()
print(x(1))
print(x(2))
print(x(3))
print(x(4))
print(x(5))
print(x(6))
print(x(7))
print(x(8))
print(x(9))
print(x(10))
print(x(11))
print(x(12))
```

Кэш как и всегда расположен в пространстве имен внешней функции. Вызов внешней функции подразумевает создание внутренней, у которой есть доступ к пустому списку. Первые 10-ть вызовов с аргументом подразумевают пополнение данного списка и его **return**. Далее происходит пополнение с одновременным удалением первого элемента.

2.4 - Декораторы

Ниже пример создания декоратора, который применяется к функции и добавляет ей новый функционал в виде дополнительного вывода перед запуском функции и после:

```
def decorator(func):
    def wrapper():
        print('Начало ф-ии')
        func()
        print('Конец ф-ии')
    return wrapper


def funcx():
    print('Ф-ия func в работе...')
```

Чтобы применить декоратор к функции можно использовать несколько вариантов, самый распространенный:

```
def decorator(func):
```

```
    def wrapper():
        print('Начало ф-ии')
        func()
        print('Конец ф-ии')
    return wrapper


@decorator
def funcx():
    print('Ф-ия func в работе...')


funcx()
```

В данном варианте любой запуск функции **funcx** подразумевает запуск декорированного варианта.

Тоже самое, но в ручном варианте:

```
def decorator(func):
    def wrapper():
        print('Начало ф-ии')
        func()
        print('Конец ф-ии')
    return wrapper


def funcx():
    print('Ф-ия func в работе...')


funcx = decorator(funcx)
funcx()
```

Преимущество данного способа в том, если наша задача будет использовать оба варианта функции - с декоратором и без, здесь это возможно. Просто переменную **funcx** меняем на другую:

```
def decorator(func):
    def wrapper():
        print('Начало ф-ии')
        func()
        print('Конец ф-ии')
    return wrapper


def funcx():
    print('Ф-ия func в работе...')


funcx_ = decorator(funcx)
funcx_()
print('----------------------')
funcx()
```

Если декорируемая функция предполагает наличие атрибутов, то это записывается так:

```
def decorator(func):
    def wrapper(*args):
        print('Начало ф-ии')
        func(*args) # тут распаковка
        print('Конец ф-ии')
    return wrapper


@decorator
def funcx(name):
    print(f'Ф-ия func с атрибутом {name} в работе...')


funcx('abc')
```

Т.е. во **wrapper** мы прокидываем аргументы, которые собираются в кортеж, а при вызове функции он распаковывается. С **args** декоратор становится универсальным. Соответственно для именованных параметров мы будем использовать **kwargs**.

Пример применения декоратора для измерения времени работы функции:

```
import time


def time_decorator(func):
    def wrapper(*args, **kwargs):
        x_1 = time.time()
        f = func(*args, **kwargs)
        print(f'Время работы ф-ии: {time.time() - x_1}')
        return f
    return wrapper


# @time_decorator
def funcx(lst):
    s = 0
    for i in lst:
        s += i**3
    return s


funcx = time_decorator(funcx)    # эта конструкция аналогична
@time_decorator
print(funcx(range(1, 19999999)))
```

Декорирование функции возможно сразу несколькими функциями-декораторами:

```
def decor_1(func):
    def wrapper(*args):
        return f'-----------\n{func(*args)}\n-----------'
    return wrapper
```

```
def decor_2(func):
    def wrapper(*args):
        return f'**********\n{func(*args)}\n**********'
    return wrapper


@decor_2
@decor_1
def func_x(s):
    return s


print(func_x('Бармалей'))
```

Сначала функция декорируется декоратором decor_1, затем декорированная функция декорируется декоратором decor_2. Обратите внимание на порядок декорации в примере. В ручном варианте это выглядит так:

```
x = decor_2(decor_1(func_x))
print(x("Бармалей"))
```

2.5 - Декораторы с параметрами

Есть небольшая проблема, связанная с декорированием функций, оригинальная функция при декорировании теряет свое имя и документацию, если она есть:

```
def decorator(func):
    def wrapper(*args, **kwargs):
        print(f'Начало работы ф-ии {func}')
        return func(*args, **kwargs)
    return wrapper


# @decorator
def func(it):
    """
    Эта ф-ия аналог sum
    :param it: iterable
    :return: int
    """
    return sum(it)


print(func, func.__doc__, func.__name__, sep='\n')
```

В данном примере функция еще не задекорирована. При стандартном выводе мы видим упоминание имени. Также имя функции можно вернуть с помощью специальной переменной **__name__**, документацию соответственно при помощи переменной **__doc__**. Теперь раскомментируйте декоратор и убедитесь, что изначальные данные потеряны.

Есть два способа исправить ситуацию. В первом случае, так как декоратор возвращает обертку, а она в свое время тоже функция, мы переменные имени и документации обертки присваиваем к переменным функции:

```
def decorator(func):
    def wrapper(*args, **kwargs):
        print(f'Начало работы ф-ии {func}')
        return func(*args, **kwargs)
    wrapper.__name__ = func.__name__
    wrapper.__doc__ = func.__doc__
    return wrapper


@decorator
def func(it):
    """
    Эта ф-ия аналог sum
    :param it: iterable
    :return: int
    """
    return sum(it)


print(func, func.__doc__, func.__name__, sep='\n')
```

Второй(чаще используемый) воспользоваться специальным декоратором **wraps** из библиотеки **functools**:

```
from functools import wraps

def decorator(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        print(f'Начало работы ф-ии {func}')
        return func(*args, **kwargs)
    return wrapper


@decorator
def func(it):
    """
    Эта ф-ия аналог sum
    :param it: iterable
    :return: int
    """
    return sum(it)


print(func, func.__doc__, func.__name__, sep='\n')
```

Технически этот декоратор делает тоже самое, что и в первом случае.

Декоратор с параметром очень похож на обычный. Технически это тот же декоратор, только вложенный в еще одну функцию, суть которой принять параметр и "протащить" его в обертку:

```
def decor_param(typ):
    def decor(func):
        def wrapper(*args):
            if len([i for i in args if type(i) == typ]) !=
len(args):
                return f'Аргументами должны быть объекты типа -
{typ}'
            return func(*args)
        return wrapper
    return decor


@decor_param(str)
def func(a, b):
    return a + b


print(func('8', '5'))

# x = decor_param(int)(func)
# print(x(8, 5))
```

Здесь декоратор служит в качестве валидатора и является неким переключателем между типами, которые могут быть параметрами функции.

2.6 - Сортировки с ключом key

Прежде чем мы с вами поговорим о параметре **key**, напомню что функция **sorted** и метод **list.sort** отличаются тем, что в первой возвращается новый объект, а во втором видоизменяется список к которому был применен метод. А объединяет их этот самый параметр. Т.е. в зависимости от потребности используется какая-то из этих 2-ух сортировок, но принцип сортировки в них идентичный, в том числе и с параметром **key**.

Пример сортировки по модулю:

```
lst = [-5, 5, 3, -7, -2, 23]
print(sorted(lst, key=abs))
```

Принцип сортировки в том, что к каждому объекту нашей коллекции применяется функция из аргумента **key**, в данном случае функция **abs**, и уже получившиеся результаты сравниваются между собой. Но сортируются изначальные объекты, а не те которые мы получили после применения функции из параметра **key**.

Пример сортировки сначала по четным числам - потом по нечетным:

```
lst = [-5, 5, 3, -7, 24, 23, 6]
print(sorted(lst, key=lambda x: (x % 2, x)))
```

Это пример двойной сортировки, сначала применяется **x % 2**, в результате которой мы получаем **0** и **1**. Значения с **0** - четные числа и поэтому они попадут в начало коллекции(мы же сортируем по возрастающей). Второй аргумент в кортеже необходим, когда внутри группы четных или нечетных нужна дополнительная сортировка. В данном случае я хочу чтобы четные отсортировались между собой по возрастающей поэтому я и прописал просто **x**. Тоже самое происходит и внутри группы нечетных.

Пример сортировки по методу:

```
lst = ['B', 'a', 'd', 'C', 'r', 'o']
print(sorted(lst, key=str.lower))  # lambda x: x.lower()
```

С **lambda** это все таки понятнее выглядит.

Сортировка по длине слов и алфавиту:

```
lst = ['ремонт', 'арбалет', 'рыбий хрящь', 'эвакуатор', 'брикет',
'стирка']
lst.sort(key=lambda x: (len(x), x))
print(lst)
```

Пример сортировки вложенных коллекций по индексу элемента:

```
lst = [['Nintendo64', 2500],
       ['ps1', 4300],
       ['Dreamcast', 3700],
       ['Dendy', 1300]
       ]

print(sorted(lst, key=lambda x: x[1]))
```

Начну с того, что если мы сравниваем коллекции, а не простые объекты, то сравнение всегда происходит по первому элементу, попробуйте запустить код без ключа key. В нашем случае мы применяем функцию, которая возвращает второй элемент коллекции, а значит и сравнение происходит по нему. Если прописать **-x[1]**, то сравнение будет по убывающей.

3.4 - Виртуальное окружение

Команды:

```
# Создать виртуальное окружение в папке проекта

python -m venv venv


# Посмотреть список установленных интепретаторов

py --list           - для Windows
whereis python3     - для Linux и MacOs


# Создать виртуальное окружение с интерпретатором определенной
версии в папке проекта

py -3.7 -m venv venv - для Windows, где 3.7 версия python
python3.7 -m venv venv - для Linux и MacOs


# Активация виртуального окружения из папки проекта

venv\Scripts\activate.bat - для Windows
source venv/bin/activate - Для Mac и Linux


# Выйти из виртуального окружение

deactivate
```