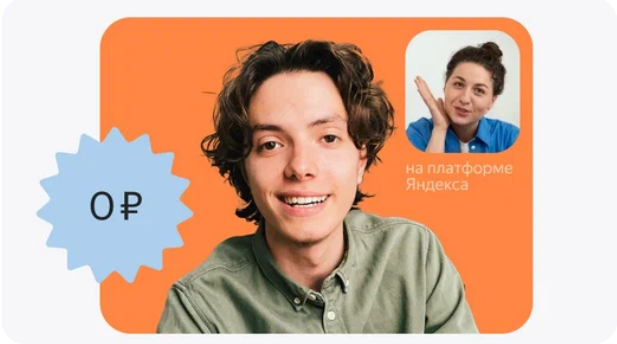


ХОЧУ ПОМОЧЬ
ПРОЕКТУ

Поддержка аннотации типов в Python



practicum.yandex.ru

РЕКЛАМА · 18+

Бесплатное занятие английским в Яндекс Практикуме

Полноценное занятие с преподавателем, а не презентация курсов

Узнать больше

Статья / Поддержка аннотации типов в Python

Внимание! Python не проверяет и не принимает во внимание аннотации типов функций и переменных. Их инструменты, такие как средства проверки типов, [IDE](#), линтеры и т. д.

Модуль `typing` предоставляет поддержку выполнения аннотации типов. Наиболее фундаментальная поддержка состоит из `typing.List`, `typing.Tuple`, `typing.Callable`, `typing.TypeVar` и `typing.Generic`.

Для упрощенного введения в аннотации типов смотрите материал "[Аннотации типов в функциях Python](#)".

Функция ниже принимает и возвращает строку и аннотируется следующим образом:

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

В функции `greeting()` ожидается, что имя аргумента будет иметь тип `str` и возвращаемый тип ожидается `str`. Подтипы принимаются в качестве аргументов типов. Например переменная `lst`, состоящий из значений `int` будет аннотироваться как `lst: list[int]`.

Примечание. Модуль `typing` определяет несколько типов, которые являются подклассами уже существующих классов стандартной библиотеки, и которые также расширяют `typing.Generic` для поддержки типов переменных внутри `[]` скобок. С версии Python 3.9, [классы стандартной библиотеки были расширены](#) для поддержки синтаксиса `[]` и эти типы стали избыточными.

Избыточные типы устарели в Python 3.9, но интерпретатор не будет выдавать предупреждений `DEPRECATED`. Ожидается, что средства проверки типов будут отмечать устаревшие типы, когда проверяемый код нацелен на версию Python 3.9 или новее.

Устаревшие типы будут удалены из модуля `typing` в первой версии Python, выпущенной через 5 лет после выпуска *Python 3.9.0*.

Содержание:

- [Использование псевдонимов типов;](#)
- [Аннотация отдельных подтипов `typing.NewType`;](#)
- [Аннотация функций обратного вызова `typing.Callable`;](#)
- [Аннотация универсальных типов;](#)
 - [Аннотация универсальных типов, определяемые пользователем;](#)
- [Применение аннотации `typing.Any`;](#)
- [Номинальные и структурные подтипы.](#)

Использование псевдонимов типов в модуле `typing`.

Псевдоним типа определяется путем присвоения типа псевдониму. В этом примере `Vector` и `list[float]` будут рассматриваться как взаимозаменяемые синонимы:

```
Vector = list[float]

def scale(scalar: float, vector: Vector) -> Vector:
    return [scalar * num for num in vector]

# Тип будет проверяться;
```

```
# список значений `float` квалифицируется как вектор.
new_vector = scale(2.0, [1.0, -4.2, 5.4])
```

Псевдонимы типов полезны для упрощения сигнатур сложных типов. Например:

```
from collections.abc import Sequence

ConnectionOptions = dict[str, str]
Address = tuple[str, int]
Server = tuple[Address, ConnectionOptions]

def broadcast_message(message: str, servers: Sequence[Server]) -> None:
    ...

# Средство проверки статического типа будет рассматривать
# подпись предыдущего типа как точно эквивалентную этой.
def broadcast_message(
    message: str,
    servers: Sequence[tuple[tuple[str, int], dict[str, str]]]) -> None:
```

Обратите внимание, что указание типа None является особым случаем и заменяется type(None).

Аннотация отдельных типов typing.NewType.

Используйте вспомогательный класс `typing.NewType()` для создания отдельных типов:

```
from typing import NewType

UserId = NewType('UserId', int)
some_id = UserId(524313)
```

Средство проверки статического типа будет рассматривать новый тип, как если бы он был подклассом исходного типа. Такое поведение полезно для выявления логических ошибок:

```
def get_user_name(user_id: UserId) -> str:
    ...

# Тип будет проверяться
user_a = get_user_name(UserId(42351))

# тип не проверяется; `int` не является UserId
user_b = get_user_name(-1)
```

По-прежнему можно выполнять все операции с `int` с переменной типа `UserId`, но результат всегда будет иметь тип `int`. Это позволяет передать `UserId` везде, где можно ожидать `int`, но предотвратит случайное создание `UserId` недопустимым способом:

```
# 'output' имеет тип 'int', а не 'UserId'
output = UserId(23413) + UserId(54341)
```

Обратите внимание, что эти проверки выполняются только средством проверки типов. Во время выполнения оператор `Derived = NewType('Derived', Base)` сделает `Derived` функцией, которая немедленно возвращает любой переданный ей параметр. Это означает, что выражение `Derived(some_value)` не создает новый класс и не вводит никаких накладных расходов, помимо обычных вызовов функции.

Точнее, выражение `some_value is Derived(some_value)` всегда истинно во время выполнения.

Это также означает, что невозможно создать подтип `Derived`, т. к. во время выполнения это функция идентификации, а не фактический тип:

```
from typing import NewType

UserId = NewType('UserId', int)

# 

Вверх

тает во время выполнения и не проверяет тип
class InUserId(UserId): pass
```

Однако можно создать `typing.NewType()` на основе производного `NewType`:

```
from typing import NewType

UserId = NewType('UserId', int)

ProUserId = NewType('ProUserId', UserId)
```

и проверка типов для `ProUserId` будет работать так, как ожидалось.

Примечание. Напомним, что использование псевдонима типа объявляет, что два типа эквивалентны друг другу. Выполнение `Alias = Original` заставит средство проверки статического типа обрабатывать псевдоним как полностью эквивалентный оригиналу во всех случаях. Это полезно, когда необходимо упростить сигнатуры сложных типов.

Напротив, `typing.NewType` объявляет один тип подтипом другого. Выполнение `Derived = NewType('Derived', Original)` заставит средство проверки аннотации типов рассматривать `Derived` как подкласс `Original`, это означает, что значение типа `Original` не может использоваться в местах, где ожидается значение типа `Derived`. Такое поведение полезно, когда необходимо предотвратить логические ошибки с минимальными затратами времени выполнения.

Изменено в версии 3.10: `NewType` теперь является классом, а не функцией. При вызове `NewType` вместо обычной функции возникают некоторые дополнительные затраты времени выполнения. В Python 3.11.0 эти затраты будут снижены.

Аннотация функций обратного вызова `typing.Callable`.

Платформы, ожидающие функций обратного вызова для определенных сигнатур, могут создать аннотацию типа с помощью `Callable[[Arg1Type, Arg2Type], ReturnType]`.

Например:

```
from collections.abc import Callable

def feeder(get_next_item: Callable[[], str]) -> None:
    # тело функции

def async_query(on_success: Callable[[int], None],
                on_error: Callable[[int, Exception], None]) -> None:
    # тело функции
```

Можно объявить возвращаемый тип вызываемого объекта без указания сигнатуры вызова, подставив буквальное многоточие вместо списка аргументов в аннотации типа: `Callable[..., ReturnType]`.

Вызываемые объекты, которые принимают другие вызываемые объекты в качестве аргументов, могут указывать на то, что их типы параметров зависят друг от друга с помощью [typing.ParamSpec](#). Кроме того, если этот вызываемый объект добавляет или удаляет аргументы из других вызываемых объектов, то может использоваться оператор [typing.Concatenate](#). Они принимают форму `Callable[ParamSpecVariable, ReturnType]` и `Callable[Concatenate[Arg1Type, Arg2Type, ..., ParamSpecVariable], ReturnType]` соответственно.

Изменено в версии 3.10: `Callable` теперь поддерживает `ParamSpec` и `Concatenate`.

Также документацию для [typing.ParamSpec](#) и `typing.Concatenate` [typing.Concatenate](#), в которой приведены примеры использования в `Callable`.

Аннотация универсальных типов.

Так как информация о типах объектов, хранящихся в контейнерах, не может быть статически представлена универсальным способом, для обозначения ожидаемых типов элементов контейнера, абстрактные базовые классы были расширены.

```
from collections.abc import Mapping, Sequence

def notify_by_email(employees: Sequence[Employee],
                   overrides: Mapping[str, str]) -> None: ...
```

Универсальные шаблоны типов можно параметризовать с помощью новой фабрики, доступной для ввода, которая называется [typing.TypedDict](#).

```
from collections.abc import Sequence
from typing import TypeVar

# Объявляем переменную типа
T = TypeVar('T')

# Универсальная функция
def first(l: Sequence[T]) -> T:
    return l[0]
```

Аннотация типов, определяемых пользователем.

Пользовательский класс может быть определен как Generic - универсальный класс.

```
from typing import TypeVar, Generic
from logging import Logger

T = TypeVar('T')

class LoggedVar(Generic[T]):
    def __init__(self, value: T, name: str, logger: Logger) -> None:
        self.name = name
        self.logger = logger
        self.value = value

    def set(self, new: T) -> None:
        self.log('Set ' + repr(self.value))
        self.value = new

    def get(self) -> T:
        self.log('Get ' + repr(self.value))
        return self.value

    def log(self, message: str) -> None:
        self.logger.info('%s: %s', self.name, message)
```

Generic[T] как базовый класс определяет, что класс LoggedVar принимает единственный параметр типа T. Это также делает T действительным как тип в теле класса.

Базовый класс [typing.Generic](#) определяет `__class_getitem__()`, так что LoggedVar[t] действителен как тип:

```
from collections.abc import Iterable

def zero_all_vars(vars: Iterable[LoggedVar[int]]) -> None:
    for var in vars:
        var.set(0)
```

Универсальный тип может иметь любое количество типов переменных, а переменные типа могут быть ограничены:

```
from typing import TypeVar, Generic
...

T = TypeVar('T')
S = TypeVar('S', int, str)

class StrangePair(Generic[T, S]):
    ...
```

Все аргументы переменной типа для typing.Generic должны быть разными. Таким образом, следующее неверно:

```
from typing import TypeVar, Generic
...

T = TypeVar('T')
```

Вверх

```
class Pair(Generic[T, T]):    # НЕБЕРНО
    ...
```

C [typing.Generic](#) можно использовать множественное наследование :

```
from РЕКЛАМАcollections.abc import Sized
from typing import TypeVar, Generic

T = TypeVar('T')

class LinkedList(Sized, Generic[T]):
    ...
```

При наследовании от универсальных классов некоторые переменные типа могут быть исправлены:

```
from collections.abc import Mapping
from typing import TypeVar

T = TypeVar('T')

class MyDict(Mapping[str, T]):
    ...
```

В этом случае MyDict имеет единственный параметр T.

Использование универсального класса без указания параметров типа предполагает [typing.Any](#) для каждой позиции. В следующем примере MyIterable не является универсальным, а неявно наследуется от Iterable[Any]:

```
from collections.abc import Iterable

# Такой же как `Iterable[Any]`
class MyIterable(Iterable):
```

Также поддерживаются определенные пользователем псевдонимы универсального типа. Примеры:

```
from collections.abc import Iterable
from typing import TypeVar, Union

S = TypeVar('S')
Response = Union[Iterable[S], int]

# Возвращаемый тип здесь такой же,
# как `Union[Iterable[str], int]`
def response(query: str) -> Response[str]:
    ...

T = TypeVar('T', int, float, complex)
Vec = Iterable[tuple[T, T]]

# Такой же как `Iterable[tuple[T, T]]`
def inproduct(v: Vec[T]) -> T:
    return sum(x*y for x, y in v)
```

Изменено в Python 3.7: typing.Generic больше не имеет собственного метакласса.

Определяемые пользователем универсальные типы для выражений параметров, также поддерживаются через переменные спецификации параметров в форме Generic[P]. Поведение согласуется с описанными выше переменными типа, т.к. спецификации параметров переменных обрабатываются модулем ввода как специализированная переменная типа. Единственным исключением из этого является то, что список типов можно использовать для замены [typing.ParamSpec](#):

```
>>> from typing import Generic, ParamSpec, TypeVar

>>> T = TypeVar('T')
>>> P = ParamSpec('P')

>> Вверх Z(Generic[T, P]): ...
...

```



```
>>> Z[int, [dict, float]]
# __main__.Z[int, (<class 'dict'>, <class 'float'>)]
```

Кроме того, универсальный шаблон с только одной переменной спецификации параметра будет принимать списки параметров в формах `X[[Type1, Type2, ...]]`, а также `X[Type1, Type2, ...]` по эстетическим соображениям. Внутри последние преобразуются в первые и, таким образом, эквивалентны:

```
>>> class X(Generic[P]): ...
...
>>> X[int, str]
# __main__.X[(<class 'int'>, <class 'str'>)]
>>> X[[int, str]]
# __main__.X[(<class 'int'>, <class 'str'>)]
```

Обратите внимание, что универсальные типы с [typing.ParamSpec](#), в некоторых случаях, могут не иметь правильных `__parameters__` после подстановки, потому что они предназначены в первую очередь для проверки статического типа.

Изменено в версии 3.10: Generic теперь можно параметризовать с помощью выражений ParamSpec.

Определенный пользователем универсальный класс может иметь ABC в качестве базовых классов без конфликта метаклассов. Универсальные метаклассы не поддерживаются. Результат параметризации универсальных шаблонов кэшируется, и большинство типов в модуле типизации являются хешируемыми и сопоставимыми по равенству.

Применение аннотации typing.Any.

Особый тип аннотации [typing.Any](#). Средство проверки статического типа будет рассматривать каждый тип как совместимый с Any и Any как совместимый с каждым типом.

Это означает, что можно выполнить любую операцию или вызов метода для значения типа Any и присвоить его любой переменной:

```
from typing import Any

a = None      # type: Any
a = []        # OK
a = 2         # OK

s = ''        # type: str
s = a         # OK

def foo(item: Any) -> int:
    # Проверка типов; 'item' может быть
    # любого типа, и этот тип может
    # иметь метод 'bar'
    item.bar()
    ...
```

Обратите внимание, что при присвоении значения типа Any более точному типу, проверка типов выполняться не будет. Например, средство проверки аннотации не сообщило об ошибке при присвоении a параметру s, даже если s был объявлен как имеющий тип str и получил значение int во время выполнения!

Кроме того, все функции без возвращаемого типа или типов параметров неявно по умолчанию будут использовать Any:

```
def legacy_parser(text):
    ...
    return data

# Статическая проверка типов будет
# рассматривать вышеприведенное
# как имеющее ту же сигнатуру, что и:
def legacy_parser(text: Any) -> Any:
    ...
    return data
```

Такое поведение позволяет использовать [typing.Any](#) в качестве аварийного выхода, когда необходимо смешивать динамически и статически типизированный код.

Сравните поведение `typing.Any` с поведением [object](#): Подобно `Any`, каждый тип является подтипом [object](#). Однако, в отличие от `Any`, обратное неверно: `object` не является подтипом любого другого типа.

Это означает, что, когда типом значения является объект, то средство проверки типов отклоняет почти все операции с ним, РЕКЛАМА присвоение его переменной (или использование в качестве возвращаемого значения) более специализированного типа является ошибкой типа. Например:

```
def hash_a(item: object) -> int:
    # Не проходит; у `object` нет `magic` метода.
    item.magic()
    ...

def hash_b(item: Any) -> int:
    # Проверка типа прошла успешно
    item.magic()
    ...

# Проверка прошла, поскольку `ints` и
# `strs` являются подклассами объекта
hash_a(42)
hash_a("foo")

# Проверка прошла, т.к. `Any`
# совместим со всеми типами
hash_b(42)
hash_b("foo")
```

Используйте `object`, для указания значения любого типа безопасным способом, а `typing.Any`, для динамически типизируемых значений.

Номинальные и структурные подтипы.

Первоначально определи систему статических типов Python, как использование номинальных подтипов. Это означает, что класс A разрешен там, где ожидается класс B тогда и только тогда, когда A является подклассом B.

Это требование ранее также применялось к абстрактным базовым классам, таким как `Iterable`. Проблема с этим подходом заключается в том, что класс должен быть явно помечен для их поддержки, что не является питоническим и отличается от того, что обычно делают в идиоматическом динамически типизированном коде Python. Например:

```
from collections.abc import Sized, Iterable, Iterator

class Bucket(Sized, Iterable[int]):
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...
```

Однако пользователи могут писать приведенный выше код без явных базовых классов в определении класса, что позволяет средствам проверки статических типов неявно рассматривать `Bucket` как подтип `Sized` и `Iterable[int]`. Это называется структурным подтипом или статической [утиной типизацией](#):

```
from collections.abc import Iterator, Iterable

# Примечание: нет базовых классов
class Bucket:
    ...
    def __len__(self) -> int: ...
    def __iter__(self) -> Iterator[int]: ...

def collect(items: Iterable[int]) -> int: ...
result = collect(Bucket()) # Проходит проверку типа
```

Более того, создавая подклассы для специального класса `Protocol`, пользователь может определять новые настраиваемые протоколы, чтобы в полной мере насладиться структурным подтипом.

- [КРАТКИЙ ОБЗОР МАТЕРИАЛА.](#)
- [Аннотация Any модуля typing](#)
- [Аннотации Never и NoReturn модуля typing](#)
- РЕКЛАМА [...](#)
- [Тип аннотации TypeAlias модуля typing](#)
- [Аннотация LiteralString модуля typing](#)
- [Аннотация Self модуля typing](#)
- [Аннотации Required и NotRequired модуля typing](#)
- [Тип аннотации Union модуля typing](#)
- [Тип аннотации Optional модуля typing](#)
- [Тип аннотации Tuple\(\) модуля typing](#)
- [Тип аннотации Callable\(\) модуля typing](#)
- [Тип аннотации Concatenate модуля typing](#)
- [Класс ParamSpec модуля typing](#)
- [Тип аннотации TypeGuard модуля typing](#)
- [Класс Type\(\) модуля typing](#)
- [Тип аннотации Literal модуля typing](#)
- [Тип аннотации ClassVar модуля typing](#)
- [Тип аннотации Final\(\) модуля typing](#)
- [Тип аннотации Annotated модуля typing](#)
- [Тип аннотации Generic модуля typing](#)
- [Тип аннотации TypeVar модуля typing](#)
- [Аннотация TypeVarTuple модуля typing](#)
- [Тип аннотации Unpack модуля typing](#)
- [Тип аннотации AnyStr модуля typing](#)
- [Тип аннотации Protocol\(\) модуля typing](#)
- [Декоратор @runtime checkable модуля typing](#)
- [Тип аннотации NamedTuple модуля typing](#)
- [Класс NewType модуля typing](#)
- [Тип аннотации TypedDict\(\) модуля typing](#)
- [Типы аннотаций коллекций модуля typing](#)
- [Аннотация абстрактных базовых классов](#)
- [Функции и декораторы модуля typing](#)
- [Помощники самоанализа модуля typing](#)
- [ParamSpecArgs и ParamSpecKwargs модуля typing](#)