

Type hints cheat sheet

변수

기술적으로 아래에 표시된 타입 어노테이션의 대부분은 중복입니다. mypy는 일반적으로 변수의 값으로부터 타입을 추론할 수 있기 때문입니다. 자세한 내용은 [Type inference and type annotations](#)을 참조하세요.

변수의 타입을 선언하는 방법입니다

```
age: int = 1
```

변수를 어노테이션하기 위해 초기화할 필요는 없습니다

```
a: int # 0k (할당되기 전까지는 런타임에 값이 없음)
```

이렇게 하는 것은 조건부 분기에서 유용할 수 있습니다

```
child: bool
if age < 18:
    child = True
else:
    child = False
```

유용한 내장 타입

대부분의 타입에 대해서는 어노테이션에 타입의 이름을 사용하면 됩니다.

따라서 기술적으로 이 어노테이션들은 중복입니다

```
x: int = 1
x: float = 1.0
x: bool = True
x: str = "test"
x: bytes = b"test"
```

Python 3.9+에서 컬렉션의 경우, 컬렉션 아이템의 타입은 괄호 안에 넣습니다

```
x: list[int] = [1]
x: set[int] = {6, 7}
```

매핑의 경우, 키와 값 모두의 타입이 필요합니다

```
x: dict[str, float] = {"field": 2.0} # Python 3.9+
```

고정 크기 튜플의 경우, 모든 요소의 타입을 지정합니다

```
x: tuple[int, str, float] = (3, "yes", 7.5) # Python 3.9+
```

가변 크기 튜플의 경우, 하나의 타입과 생략 부호를 사용합니다

```
x: tuple[int, ...] = (1, 2, 3) # Python 3.9+
```

Python 3.8 이전 버전에서는 컬렉션 타입의 이름이 대문자로 시작하며, 타입은 'typing' 모듈에서 가져옵니다

```
from typing import List, Set, Dict, Tuple
x: List[int] = [1]
x: Set[int] = {6, 7}
x: Dict[str, float] = {"field": 2.0}
x: Tuple[int, str, float] = (3, "yes", 7.5)
x: Tuple[int, ...] = (1, 2, 3)
```

from typing import Union, Optional

Python 3.10+에서는 무언가가 몇 가지 타입 중 하나일 수 있을 때 | 연산자를 사용합니다

```
x: list[int | str] = [3, 5, "test", "fun"] # Python 3.10+
# 이전 버전에서는 Union을 사용합니다
x: list[Union[int, str]] = [3, 5, "test", "fun"]
```

None일 수 있는 값에 대해서는 Optional[X]를 사용합니다

Optional[X]는 X | None 또는 Union[X, None]과 같습니다

```
x: Optional[str] = "something" if
some_condition() else None
if x is not None:
    # Mypy는 if문 때문에 여기서 x가 None이 아닐
    # 것임을 이해합니다
    print(x.upper())
```

mypy가 이해하지 못하는 로직으로 인해 값이 절대 None이 될 수 없다는 것을 안다면, # assert를 사용하세요

```
assert x is not None
print(x.upper())
```

함수

from typing import Callable, Iterator, Union, Optional

함수 정의를 어노테이션하는 방법입니다

```
def stringify(num: int) → str:
    return str(num)
```

여러 인수를 지정하는 방법입니다

```
def plus(num1: int, num2: int) → int:
    return num1 + num2
```

함수가 값을 반환하지 않는다면, 반환 타입으로 None을 사용합니다

```
# 인수의 기본값은 타입 어노테이션 뒤에 옵니다
def show(value: str, excitement: int = 10) → None:
    print(value + "!" * excitement)
```

타입이 없는 인수는 동적으로 타입이 지정되며 (Any로 처리됨)

```
# 어노테이션이 없는 함수는 확인되지 않습니다
def untyped(x):
    x.anything() + 1 + "string" # 오류 없음
```

호출 가능한(함수) 값을 어노테이션하는 방법입니다

```
x: Callable[[int, float], float] = f
def register(callback: Callable[[str], int]) → None: ...
```

int를 생성하는 제네레이터 함수는 사실상 int의 이터레이터를 반환하는 함수이므로,

그렇게 어노테이션합니다

```
def gen(n: int) → Iterator[int]:
    i = 0
    while i < n:
        yield i
        i += 1
```

물론 함수 어노테이션을 여러 줄로 나눌 수 있습니다

```
def send_email(address: Union[str, list[str]],
                sender: str,
                cc: Optional[list[str]],
                bcc: Optional[list[str]],
                subject: str = '',
                body: Optional[list[str]]
                ) → bool:
    ...
```

Mypy는 위치 전용 및 키워드 전용 인수를 이해합니다

위치 전용 인수는 두 개의 밑줄로 시작하는 이름을 사용하여 표시할 수도 있습니다

```
def quux(x: int, /, *, y: int) → None:
    pass
```

```
quux(3, y=5) # Ok
```

```
quux(3, 5) # error: Too many positional arguments for "quux"
```

```
quux(x=3, y=5) # error: Unexpected keyword argument "x" for "quux"
```

이것은 각 위치 인수와 각 키워드 인수가 "str"임을 의미합니다

```
def call(self, *args: str, **kwargs: str) → str:
```

```
    reveal_type(args) # 드러난 타입은 "tuple[str, ...]"
```

```
    reveal_type(kwargs) # 드러난 타입은 "dict[str, str]"
```

```
    request = make_request(*args, **kwargs)
```

```
    return self.do_api_query(request)
```

클래스

```
class BankAccount:
```

```
    # "__init__" 메서드는 아무것도 반환하지 않으므로,
```

```
    # 아무것도 반환하지 않는 다른 메서드와 마찬가지로 반환 타입 "None"을 가집니다
```

```
    def __init__(self, account_name: str, initial_balance: int = 0) → None:
```

```
        # mypy는 매개변수의 타입을 기반으로 이 인스턴스 변수들의
```

```
        # 올바른 타입을 추론합니다.
```

```
        self.account_name = account_name
        self.balance = initial_balance
```

```
        # 인스턴스 메서드의 경우, "self"의 타입은 생략합니다
```

```
        def deposit(self, amount: int) → None:
```

```
            self.balance += amount
```

```
        def withdraw(self, amount: int) →
```

```
None:
```

```

        self.balance -= amount

# 사용자 정의 클래스는 어노테이션에서 타입으로
# 유효합니다
account: BankAccount =
BankAccount("Alice", 400)
def transfer(src: BankAccount, dst:
BankAccount, amount: int) → None:
    src.withdraw(amount)
    dst.deposit(amount)

# BankAccount를 받는 함수는 BankAccount의
# 모든 서브클래스도 받습니다!
class AuditedBankAccount(BankAccount):
    # 클래스 본문에서 인스턴스 변수를 선택적
    # 로 선언할 수 있습니다
    audit_log: list[str]

    def __init__(self, account_name:
str, initial_balance: int = 0) → None:
        super().__init__(account_name,
initial_balance)
        self.audit_log: list[str] = []

    def deposit(self, amount: int) →
None:
        self.audit_log.append(f"Deposited
{amount}")
        self.balance += amount

    def withdraw(self, amount: int) →
None:
        self.audit_log.append(f"Withdrew
{amount}")
        self.balance -= amount

audited = AuditedBankAccount("Bob", 300)
transfer(audited, account, 100) # 타입
# 검사 통과!

# ClassVar 어노테이션을 사용하여 클래스 변수
# 를 선언할 수 있습니다
class Car:
    seats: ClassVar[int] = 4
    passengers: ClassVar[list[str]]

# 클래스에서 동적 속성을 원한다면,
# "__setattr__" 또는 "__getattr__"을 오버

```

```

라이드하세요
class A:
    # 이것은 x가 "value"와 같은 타입인 경우
    # 모든 A.x에 대한 할당을 허용합니다
    # (임의의 타입을 허용하려면 "value:
Any"를 사용하세요)
    def __setattr__(self, name: str,
value: int) → None: ...

    # 이것은 x가 반환 타입과 호환되는 경우 모
    # 든 A.x에 대한 액세스를 허용합니다
    def __getattr__(self, name: str) →
int: ...

a = A()
a.foo = 42 # 작동함
a.bar = 'Ex-parrot' # 타입 검사 실패

```

혼란스럽거나 복잡한 경우

```

from typing import Union, Any, Optional,
TYPE_CHECKING, cast

```

```

# 프로그램의 어느 곳에서든 표현식에 대해 mypy
# 가 추론하는 타입을 알아보려면,
# reveal_type()으로 감싸세요. Mypy는 타입과
# 함께 오류 메시지를 출력합니다;
# 코드를 실행하기 전에 다시 제거하세요.
reveal_type(1) # 드러난 타입은
"builtins.int"

```

```

# 빈 컨테이너 또는 "None"으로 변수를 초기화하
# 는 경우
# 명시적 타입 어노테이션을 제공하여 mypy를 약
# 간 도와야 할 수 있습니다
x: list[str] = []
x: Optional[str] = None

```

```

# 무언가의 타입을 모르거나 타입을 작성하기에는
# 너무 동적인 경우 Any를 사용합니다
x: Any = mystery_function()
# Mypy는 x로 무엇이든 할 수 있게 해줍니다!
x.whatever() * x["you"] + x("want") -
any(x) and all(x) is super # 오류 없음

```

```

# 코드가 mypy를 혼란스럽게 하거나 mypy의 완전
# 한 버그에 부딪힐 때,
# 특정 줄의 오류를 억제하려면 "type: ignore"
# 주석을 사용하세요.

```

```

# 문제를 설명하는 주석을 추가하는 것이 좋은 관
# 행입니다.

```

```

x = confusing_function() # type: ignore
# confusing_function은 여기서 None을 반환
# 하지 않을 것입니다. 왜냐하면 ...

```

```

# "cast"는 표현식의 추론된 타입을 오버라이드
# 할 수 있게 해주는 도우미 함수입니다.
# 이것은 mypy만을 위한 것이며 -- 런타임 검사
# 는 없습니다.
a = [4]
b = cast(list[int], a) # 문제없이 통과
c = cast(list[str], a) # 거짓말임에도 불
# 구하고 문제없이 통과 (런타임 검사 없음)
reveal_type(c) # 드러난 타입은
"builtins.list[builtins.str]"
print(c) # 여전히 [4]를 출력 ... 객체는 런
# 타임에 변경되거나 캐스팅되지 않습니다

```

```

# mypy가 볼 수 있지만 런타임에 실행되지 않는
# 코드를 원한다면 "TYPE_CHECKING"을 사용하세요
# (또는 mypy가 볼 수 없는 코드를 위해)
if TYPE_CHECKING:

```

```

    import json
else:
    import orjson as json # mypy는 이것
# 을 인식하지 못합니다
# 일부 경우에 타입 어노테이션이 런타임에 문제를 일으
# 킬 수 있습니다. 이를 처리하려면 Annotation issues
# at runtime을 참조하세요.

```

```

오류를 무시하는 방법에 대한 자세한 내용은 Silencing
type errors를 참조하세요.

```

표준 "덕 타입"

```

일반적인 Python 코드에서 list나 dict를 인수로 받을
# 수 있는 많은 함수들은 인수가 어떻게든 "list-like"하거
# 나 "dict-like"하기만 하면 되며, 관용적인 Python에서
# 흔한 여러 덕 타입이 표준화되어 있습니다.

```

```

from typing import Mapping,
MutableMapping, Sequence, Iterable

```

```

# 일반적인 반복 가능한 객체("for"에서 사용할
# 수 있는 모든 것)에는 Iterable을 사용하고,
# 시퀀스("len"과 "__getitem__"을 지원하는)
# 가 필요한 곳에는 Sequence를 사용합니다
def f(ints: Iterable[int]) → list[str]:
    return [str(x) for x in ints]

```

```

f(range(1, 3))

```

```

# Mapping은 변경하지 않을 dict-like 객체
# ("__getitem__"이 있는)를 설명하고,
# MutableMapping은 변경할 수 있는 객체
# ("__setitem__"이 있는)를 설명합니다
def f(my_mapping: Mapping[int, str]) →
list[int]:
    my_mapping[5] = 'maybe' # mypy는 이
# 줄에 대해 불평할 것입니다...
    return list(my_mapping.keys())

```

```

f({3: 'yes', 4: 'no'})

```

```

def f(my_mapping: MutableMapping[int,
str]) → set[str]:
    my_mapping[5] = 'maybe' # ...하지만
# mypy는 이것은 문제없다고 합니다.
    return set(my_mapping.values())

```

```

f({3: 'yes', 4: 'no'})

```

```

import sys
from typing import IO

```

```

# open() 호출에서 나오는 객체를 받거나 반환해
# 야 하는 함수에는
# IO[str] 또는 IO[bytes]를 사용합니다 (IO는
# 읽기, 쓰기 또는 기타 모드를 구분하지 않음)
def get_sys_IO(mode: str = 'w') →
IO[str]:
    if mode == 'w':
        return sys.stdout
    elif mode == 'r':
        return sys.stdin
    else:
        return sys.stdout
# 자신만의 덕 타입을 만들 수도 있습니다.

```

전방 참조

```

# 클래스가 정의되기 전에 참조하고 싶을 수 있습
# 니다.
# 이것을 "전방 참조"라고 합니다.
def f(foo: A) → int: # 이것은 런타임에
# 'A' is not defined로 실패할 것입니다
    ...

```

Last updated: 2026-02-11

```
# 하지만 다음과 같은 특별한 임포트를 추가하면:
from __future__ import annotations
# 런타임에 작동하고 파일 뒤쪽에 그 이름의 클래스가 있는 한 타입 검사가 성공합니다
def f(foo: A) → int: # Ok
    ...
    ({tag}')
    await asyncio.sleep(0.1)
    count -= 1
    return "Blastoff!"
```

```
# 또 다른 옵션은 타입을 따옴표로 묶는 것입니다
def f(foo: 'A') → int: # 이것도 ok
    ...
```

```
class A:
    # 이것은 해당 클래스의 정의 내부에서 타입 어노테이션에 클래스를 참조해야 하는 경우에도 발생할 수 있습니다
    @classmethod
    def create(cls) → A:
        ...
```

자세한 내용은 Class name forward references를 참조하세요.

데코레이터

데코레이터 함수는 제네릭을 통해 표현할 수 있습니다. 자세한 내용은 Declaring decorators를 참조하세요.

```
from typing import Any, Callable, TypeVar

F = TypeVar('F', bound=Callable[... , Any])
```

```
def bare_decorator(func: F) → F:
    ...
```

```
def decorator_args(url: str) → Callable[[F], F]:
    ...
```

코루틴과 asyncio

코루틴 타이핑 및 비동기 코드에 대한 전체 세부 사항은 Typing async/await를 참조하세요.

```
import asyncio

# 코루틴은 일반 함수처럼 타입이 지정됩니다
async def countdown(tag: str, count: int) → str:
    while count > 0:
        print(f'T-minus {count}')
```