Type hints cheat sheet

This document is a quick cheat sheet showing how to use type annotations for various common types in Python.

Variables

Technically many of the type annotations shown below are redundant, since mypy can usually infer the type of a variable from its value. See Type inference and type annotations for more details.

```
# This is how you declare the type of a variable
age: int = 1

# You don't need to initialize a variable to annotate it
a: int  # Ok (no value at runtime until assigned)

# Doing so can be useful in conditional branches
child: bool
if age < 18:
    child = True
else:
    child = False
```

Useful built-in types

```
# For most types, just use the name of the type in the annotation
# Note that mypy can usually infer the type of a variable from its value,
# so technically these annotations are redundant
x: int = 1
x: float = 1.0
x: bool = True
x: str = "test"
x: bytes = b"test"

# For collections on Python 3.9+, the type of the collection item is in brackets
x: list[int] = [1]
x: set[int] = {6, 7}

# For mappings, we need the types of both keys and values
x: dict[str, float] = {"field": 2.0}  # Python 3.9+

# For tuples of fixed size, we specify the types of all the elements
x: tuple[int, str, float] = (3, "yes", 7.5)  # Python 3.9+

# For tuples of variable size, we use one type and ellipsis
x: tuple[int, ...] = (1, 2, 3)  # Python 3.9+

# On Python 3.8 and earlier, the name of the collection type is
# capitalized, and the type is imported from the 'typing' module
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

# On Python 3.10+, use the | operator when something could be one of a few types
x: list[int | str] = [3, 5, "test", "fun"]  # Python 3.10+
# On earlier versions, use Union
x: list[Union[int, str]] = [3, 5, "test", "fun"]

# Use Optional[X] for a value that could be None
# Optional[X] is the same as X | None or Union[X, None]
x: Optional[str] = "something" if some_condition() else None
if x is not None:
    # Mypy understands x won't be None here because of the if-statement
    print(x.upper())
# If you know a value can never be None due to some logic that mypy doesn't
# understand, use an assert
assert x is not None
print(x.upper())
```

Functions

```
from typing import Callable, Iterator, Union, Optional

# This is how you annotate a function definition
def stringify(num: int) -> str:
    return str(num)

# And here's how you specify multiple arguments
def plus(num1: int, num2: int) -> int:
    return num1 + num2

# If a function does not return a value, use None as the return type
# Default value for an argument goes after the type annotation
def show(value: str, excitement: int = 10) -> None:
    print(value + "!" * excitement)

# Note that arguments without a type are dynamically typed (treated as Any)
# and that functions without any annotations not checked
def untyped(x):
    x.anything() + 1 + "string"  # no errors

# This is how you annotate a callable (function) value
x: Callable[[int, float], float] = f
def register(callback: Callable[[str], int]) -> None: ...

# A generator function that yields ints is secretly just a function that
# returns an iterator of ints, so that's how we annotate it
def gen(n: int) -> Iterator[int]:
    i = 0
    while i < n:
        yield i
```

```
        i += 1

# You can of course split a function annotation over multiple lines
def send_email(address: Union[str, list[str]],
               sender: str,
               cc: Optional[list[str]],
               bcc: Optional[list[str]],
               subject: str = '',
               body: Optional[list[str]] = None
               ) -> bool:
    ...

# Mypy understands positional-only and keyword-only arguments
# Positional-only arguments can also be marked by using a name starting with
# two underscores
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"

# This says each positional arg and each keyword arg is a "str"
def call(self, *args: str, **kwargs: str) -> str:
    reveal_type(args)  # Revealed type is "tuple[str, ...]"
    reveal_type(kwargs)  # Revealed type is "dict[str, str]"
    request = make_request(*args, **kwargs)
    return self.do_api_query(request)
```

Classes

```
class BankAccount:
    # The "__init__" method doesn't return anything, so it gets return
    # type "None" just like any other method that doesn't return anything
    def __init__(self, account_name: str, initial_balance: int = 0) -> None:
        # mypy will infer the correct types for these instance variables
        # based on the types of the parameters.
        self.account_name = account_name
        self.balance = initial_balance

    # For instance methods, omit type for "self"
    def deposit(self, amount: int) -> None:
        self.balance += amount

    def withdraw(self, amount: int) -> None:
        self.balance -= amount

# User-defined classes are valid as types in annotations
account: BankAccount = BankAccount("Alice", 400)
def transfer(src: BankAccount, dst: BankAccount, amount: int) -> None:
    src.withdraw(amount)
    dst.deposit(amount)

# Functions that accept BankAccount also accept any subclass of BankAccount!
class AuditedBankAccount(BankAccount):
    # You can optionally declare instance variables in the class body
```

```
    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)  # type checks!

# You can use the ClassVar annotation to declare a class variable
class Car:
    seats: ClassVar[int] = 4
    passengers: ClassVar[list[str]]

# If you want dynamic attributes on your class, have it
# override "__setattr__" or "__getattr__"
class A:
    # This will allow assignment to any A.x, if x is the same type as "value"
    # (use "value: Any" to allow arbitrary types)
    def __setattr__(self, name: str, value: int) -> None: ...

    # This will allow access to any A.x, if x is compatible with the return type
    def __getattr__(self, name: str) -> int: ...

a.foo = 42  # Works
a.bar = 'Ex-parrot'  # Fails type checking
```

When you’re puzzled or when things are complicated

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

# To find out what type mypy infers for an expression anywhere in
# your program, wrap it in reveal_type().  Mypy will print an error
# message with the type; remove it again before running the code.
reveal_type(1)  # Revealed type is "builtins.int"

# If you initialize a variable with an empty container or "None"
# you may have to help mypy a bit by providing an explicit type annotation
x: list[str] = []
x: Optional[str] = None

# Use Any if you don't know the type of something or it's too
# dynamic to write a type for
x: Any = mystery_function()
# Mypy will let you do anything with x!
x.whatever() * x["you"] + x("want") - any(x) and all(x) is super  # no errors
```

```
# Use a "type: ignore" comment to suppress errors on a given line,
# when your code confuses mypy or runs into an outright bug in mypy.
# Good practice is to add a comment explaining the issue.
x = confusing_function()  # type: ignore  # confusing_function won't return None here because ...

# "cast" is a helper function that lets you override the inferred
# type of an expression. It's only for mypy -- there's no runtime check.
a = [4]
b = cast(list[int], a)  # Passes fine
c = cast(list[str], a)  # Passes fine despite being a lie (no runtime check)
reveal_type(c)  # Revealed type is "builtins.list[builtins.str]"
print(c)  # Still prints [4] ... the object is not changed or casted at runtime

# Use "TYPE_CHECKING" if you want to have code that mypy can see but will not
# be executed at runtime (or to have code that mypy can't see)
if TYPE_CHECKING:
    import json
else:
    import orjson as json  # mypy is unaware of this
```

In some cases type annotations can cause issues at runtime, see Annotation issues at runtime for dealing with this.

See Silencing type errors for details on how to silence errors.

Standard “duck types”

In typical Python code, many functions that can take a list or a dict as an argument only need their argument to be somehow “list-like” or “dict-like”. A specific meaning of “list-like” or “dict-like” (or something-else-like) is called a “duck type”, and several duck types that are common in idiomatic Python are standardized.

```
from typing import Mapping, MutableMapping, Sequence, Iterable

# Use Iterable for generic iterables (anything usable in "for"),
# and Sequence where a sequence (supporting "len" and "__getitem__") is
# required
def f(ints: Iterable[int]) -> list[str]:
    return [str(x) for x in ints]

f(range(1, 3))

# Mapping describes a dict-like object (with "__getitem__") that we won't
# mutate, and MutableMapping one (with "__setitem__") that we might
def f(my_mapping: Mapping[int, str]) -> list[int]:
    my_mapping[5] = 'maybe'  # mypy will complain about this line...
    return list(my_mapping.keys())

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

def f(my_mapping: MutableMapping[int, str]) -> set[str]:
    my_mapping[5] = 'maybe'  # ...but mypy is OK with this.
    return set(my_mapping.values())

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

```
import sys
from typing import IO

# Use IO[str] or IO[bytes] for functions that should accept or return
# objects that come from an open() call (note that IO does not
# distinguish between reading, writing or other modes)
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
```

You can even make your own duck types using Protocols and structural subtyping.

Forward references

```
# You may want to reference a class before it is defined.
# This is known as a "forward reference".
def f(foo: A) -> int:  # This will fail at runtime with 'A' is not defined
    ...

# However, if you add the following special import:
from __future__ import annotations
# It will work at runtime and type checking will succeed as long as there
# is a class of that name later on in the file
def f(foo: A) -> int:  # Ok
    ...

# Another option is to just put the type in quotes
def f(foo: 'A') -> int:  # Also ok
    ...

class A:
    # This can also come up if you need to reference a class in a type
    # annotation inside the definition of that class
    @classmethod
    def create(cls) -> A:
        ...
```

See Class name forward references for more details.

Decorators

Decorator functions can be expressed via generics. See Declaring decorators for more details.

```
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]:
    ...
```

Coroutines and asyncio

See Typing async/await for the full detail on typing coroutines and asynchronous code.

```
import asyncio

# A coroutine is typed like a normal function
async def countdown(tag: str, count: int) -> str:
    while count > 0:
        print(f'T-minus {count} ({tag})')
        await asyncio.sleep(0.1)
        count -= 1
    return "Blastoff!"
```