



Hints are labels for readers and tools: Python stores them but never checks them, a checker such as mypy or pyright does. Each box shows the imports it needs.

BASICS

```
count: int = 0
```

 a variable with its type

```
def greet(name: str, n: int = 1) -> str:
    return name * n
```

arguments, defaults, return type

```
def log(msg: str) -> None:
    print(msg)
```

None: returns nothing useful

```
names: list[str] = []
```

label empty containers: the checker can't guess

```
class User:
    name: str
    age: int
```

declares attributes, sets no values

BUILT-IN GENERICS

```
list[int]
```

 a list of ints

```
dict[str, int]
```

 str keys, int values

```
set[str]
```

 also frozenset[str]

```
tuple[int, str]
```

 exactly two items, in this order

```
tuple[int, ...]
```

 any length, all ints

```
type[User]
```

 the class itself, not an instance

NONE AND EITHER-OR

```
from typing import Optional, Union
```

```
def find(id: int) -> str | None: ...
```

may return None: check before use

```
x: int | str = 5
```

 either type (3.10+)

```
Optional[int]
Union[int, str]
```

older spellings of int | None, int | str

FUNCTIONS AS VALUES

```
from collections.abc import Callable
```

```
Callable[[int, str], bool]
```

takes (int, str), returns bool

```
Callable[..., None]
```

any arguments, returns nothing

```
def run(f: Callable[[int], int]) -> int:
    return f(1)
```

pass a function in

ACCEPT BROAD, RETURN EXACT

```
from collections.abc import (
    Iterable, Iterator, Mapping, Sequence)
```

```
def total(xs: Iterable[float]) -> float:
    return sum(xs)
```

anything a for loop can walk

```
def first(xs: Sequence[str]) -> str:
    return xs[0]
```

indexable: list, tuple, str

```
def show(d: Mapping[str, int]) -> None:
    print(d)
```

a read-only dict-like argument

```
def nums() -> Iterator[int]:
    yield 1
```

a generator function

RECORDS

```
from typing import NamedTuple, TypedDict
from dataclasses import dataclass
```

```
class Movie(TypedDict):
    title: str
    year: int
```

a dict with known keys

```
m: Movie = {"title": "Up", "year": 2009}
```

still a plain dict at run time

```
@dataclass
class Item:
    name: str
    qty: int = 0
```

hints become the __init__ arguments

```
Item("pen")
```

 → Item(name='pen', qty=0)

```
class Point(NamedTuple):
    x: float
    y: float
```

a tuple with named, typed fields

NARROW CHOICES

```
from typing import (
    ClassVar, Final, Literal, NewType)
```

```
def act(m: Literal["r", "w"]): ...
```

only these exact values

```
MAX: Final = 100
```

the checker flags any reassignment

```
class Cfg:
    count: ClassVar[int] = 0
```

shared by the class, not per instance

```
UserId = NewType("UserId", int)
uid = UserId(42)
```

an int the checker won't mix up

GENERICS

```
from typing import TypeVar
```

```
def first[T](xs: list[T]) -> T:
    return xs[0]
```

3.12+: the return type follows the input

```
class Box[T]:
    def __init__(self, item: T) -> None:
        self.item = item
```

a generic class (3.12+)

```
T = TypeVar("T")
def last(xs: list[T]) -> T:
    return xs[-1]
```

the pre-3.12 spelling

```
def biggest[N: float](xs: list[N]) -> N:
    return max(xs)
```

a bound: N must be float or a subtype

```
type Pair = tuple[int, int]
```

a type alias (3.12+)

DUCK TYPING

```
from typing import Protocol
```

```
class Closer(Protocol):
    def close(self) -> None: ...
```

any object with close() fits

```
def shut(x: Closer) -> None:
    x.close()
```

no inheritance needed

METHODS AND OVERLOADS

```
from typing import NoReturn, Self, overload
```

```
class Q:
    def where(self, s: str) -> Self:
        return self
```

Self: subclasses keep their own type

```
def grow(self) -> "Tree": ...
```

quote a class that isn't defined yet

```
@overload
def dbl(x: int) -> int: ...
@overload
def dbl(x: str) -> str: ...
def dbl(x):
    return x * 2
```

different input, different return type

```
def fail(msg: str) -> NoReturn:
    raise ValueError(msg)
```

never returns: always raises

ANY VS OBJECT

```
from typing import Any
```

```
def f(x: Any) -> None: ...
```

turns checking off for x

```
def g(x: object) -> None: ...
```

accepts anything, allows almost nothing

AT RUN TIME

```
from typing import get_type_hints
def double(x: int) -> int:
    return x * 2
```

```
double("ab")
```

 → 'abab'
hints are not enforced: no error

```
get_type_hints(double)["x"]
```

 → <class 'int'>
read them back

```
reveal_type(x)
```

mypy and pyright print what they infer

WATCH OUT

```
from typing import Optional
```

```
def f(x: int = None): ...
```

write x: int | None = None

```
def f(x: Optional[int]): ...
```

Optional means may be None; it is still required

```
def up(s: str | None) -> str:
    return s.upper()
```

checker error: rule out None first (if s is None)

```
def avg(xs: list[float]): ...
avg(ints)
```

list[int] is rejected: lists are invariant. Use Sequence[float]

```
isinstance([1], list)
```

 → True
isinstance(x, list[int]) raises TypeError

```
from typing import List, Dict
```

old: use list and dict (3.9+)

```
x: int = "5"
```

runs fine: only the checker complains