Typing Annotation

Python Typing Annotation: A Beginner's Guide

Typing annotations in Python are a way to indicate the types of variables, function parameters, and return values. This can make code more readable and help with debugging, though the types are not enforced at runtime.

1. Basic Syntax

Python's typing annotations are introduced using a colon (`:`) to specify the type of a variable or function parameter, and an arrow (`->`) to specify the return type of a function.

Variable Annotations

```
x: int = 5
y: str = "Hello"
z: float = 3.14
```

In this example:

- `x` is an integer.
- `y` is a string.
- `z` is a float.

Function Annotations

```
def add(a: int, b: int) -> int:
    return a + b
```

Here, `add` is a function that:

- Takes two arguments `a` and `b`, both of type `int`.
- Returns an `int`.

2. Common Built-in Types

You can use common built-in types as annotations:

- `int`
- `str`
- `float`
- `bool`
- `list`
- `dict`
- `set`
- `tuple`

Example:

```
def greet(names: list[str]) -> None:
    for name in names:
        print(f"Hello, {name}")
```

3. Typing Module

The `typing` module provides more complex types for annotations:

- **List, Dict, Tuple, Set**: Use these for more precise annotations.
- **Union**: When a variable can be of multiple types.
- **Optional**: When a value can be of a certain type or `None`.
- **Any**: When you don't want to specify the type.

Example:

```
from typing import List, Dict, Tuple, Union, Optional

def process_data(data: List[Dict[str, Union[int, str]]]) -> O
ptional[str]:
    if not data:
        return None
    return "Processed"
```

Here:

- `data` is a list of dictionaries where the keys are strings, and the values can be either integers or strings.
- The function returns a `str` or `None`.

4. Type Aliases

You can create type aliases for readability or reuse.

Example:

```
from typing import Dict

Person = Dict[str, str]

def get_person() -> Person:
    return {"name": "Alice", "age": "30"}
```

Here, `Person` is a type alias for a dictionary with string keys and string values.

5. Custom Classes

You can also use custom classes in annotations:

```
class Car:
    def __init__(self, make: str, model: str):
        self.make = make
        self.model = model
```

```
def describe_car(car: Car) -> str:
    return f"{car.make} {car.model}"
```

6. Generics

Generics allow you to define classes or functions that can operate on any type. Use `TypeVar` to declare a generic type.

Example:

```
from typing import TypeVar, List

T = TypeVar('T')

def first_item(items: List[T]) -> T:
    return items[0]
```

Here, `first_item` can take a list of any type and return the first item of that type.

7. Callable

Use `Callable` to annotate functions or methods passed as arguments.

Example:

```
from typing import Callable

def execute(func: Callable[[int, int], int], a: int, b: int)
-> int:
    return func(a, b)

def multiply(x: int, y: int) -> int:
    return x * y

result = execute(multiply, 3, 4)
```

In this example, `func` is a callable that takes two integers and returns an integer.

8. Forward References

When referencing a class that hasn't been defined yet, use a string for a forward reference.

Example:

```
class Node:
    def __init__(self, value: int, next_node: 'Node' = None):
        self.value = value
        self.next_node = next_node
```

9. Final

Use `Final` to mark variables that should not be reassigned.

```
from typing import Final

PI: Final = 3.14159
```

10. Literal Types

`Literal` allows specifying exact values that a variable can take.

Example:

```
from typing import Literal

def get_status(status: Literal["open", "closed"]) -> str:
    return f"The status is {status}"

get_status("open")  # Valid
get_status("pending")  # Error: Not allowed
```

11. Type Checking with `mypy`

To enforce type checks, you can use `mypy`, a static type checker for Python.

1. Install `mypy`:

```
pip install mypy
```

2. Run `mypy` on your Python files:

```
mypy script.py
```

This will report any type inconsistencies without running your code.

Summary

Python's type annotations help clarify what types your functions and variables expect, making your code easier to understand and maintain. While not enforced at runtime, tools like `mypy` can be used for static type checking to catch errors early.

`TypedDict` and `Annotated` are advanced features in Python's type annotations that offer more granular control and clarity over data structures and constraints.

1. `TypedDict`

`TypedDict` is part of the `typing` module and allows you to define a dictionary with a specific structure, where the keys have associated types. This is useful when you want to ensure that dictionaries have a fixed set of keys with particular value types.

Basic Usage

```
from typing import TypedDict

class Point(TypedDict):
    x: int
    y: int

point: Point = {'x': 10, 'y': 20}  # Valid
invalid_point: Point = {'x': 10}  # Error: Missing key 'y'
```

In this example:

- `Point` is a `TypedDict` with two keys: `x` (an integer) and `y` (also an integer).
- If you miss a key or use an incorrect type, type checkers like `mypy` will raise an error.

Optional Keys in `TypedDict`

You can also define optional keys using `NotRequired` (in Python 3.11 and later) or `Optional` (in earlier versions).

```
from typing import TypedDict, NotRequired

class Point(TypedDict):
    x: int
    y: NotRequired[int]

point1: Point = {'x': 10}  # Valid
point2: Point = {'x': 10, 'y': 20}  # Also valid
```

Here, the key `y` is optional.

TypedDict Inheritance

You can create `TypedDict` classes that inherit from each other to extend or refine the structure:

```
class Point(TypedDict):
    x: int
    y: int

class ColoredPoint(Point):
    color: str

colored_point: ColoredPoint = {'x': 10, 'y': 20, 'color': 're
d'}
```

This allows for reusable and extendable dictionary structures.

2. `Annotated`

`Annotated` is used to add metadata or constraints to types. This metadata can be used by type checkers or runtime validators. It's part of the `typing` module (introduced in Python 3.9 and enhanced in Python 3.10).

Basic Usage

```
from typing import Annotated

def process_age(age: Annotated[int, "Must be a positive integ
er"]) -> None:
    print(f"Processing age: {age}")
```

Here, `Annotated` adds a description that `age` must be a positive integer. While this metadata is not enforced by Python, it can be utilized by static analyzers or runtime validation tools.

Combining Annotations

You can add multiple annotations to a type by chaining them.

```
from typing import Annotated
from math import sqrt

PositiveInt = Annotated[int, "positive"]
OddInt = Annotated[int, "odd"]

def square_root(value: PositiveInt) -> float:
    return sqrt(value)
```

In this example, `PositiveInt` indicates that the value should be a positive integer.

Use Cases

`Annotated` types are useful in scenarios where you want to attach metadata for:

- **Validation**: Indicate constraints like ranges, lengths, or custom conditions.
- **Documentation**: Provide additional context to developers using the API.
- **Custom Tools**: Leverage the metadata in custom tools that consume annotations.

Example with Pydantic

For runtime validation, libraries like `Pydantic` can use `Annotated` metadata:

```
from pydantic import BaseModel, Field
from typing import Annotated

class User(BaseModel):
    age: Annotated[int, Field(ge=0, le=120)]  # Age must be b
etween 0 and 120

user = User(age=25)  # Valid
invalid_user = User(age=150)  # Raises validation error
```

In this example, `Annotated` provides a way to specify constraints (e.g., `ge=0`, `le=120`) that are enforced at runtime by `Pydantic`.

Summary

- `TypedDict`: Use this when you need dictionaries with specific key-value types, and you want to enforce this structure with type checkers.
- `Annotated`: Use this to add metadata to types, which can be utilized by static analyzers, documentation tools, or runtime validators. It's especially powerful in combination with libraries like `Pydantic` for adding constraints or validation rules.