Introduction to Python Typing

Dynamic Typing in Python

- Python is **dynamically typed**.
- Variable types are determined at runtime.
- No need to declare types when writing code.

```
x = 5     # int
x = "hello"  # now a str
```

- This flexibility makes Python quick for prototyping and writing concise code.

Static Typing - Overview

Static Typing in Other Languages - In contrast, languages like Java, C++, and Go are **statically typed**. - Types are explicitly declared and checked at compile time.

```
int x = 5;
x = "hello";  // Error: Type mismatch
```

- **Benefits**:
  - Type checking at compile time (catch errors early).
  - Better performance (optimized machine code).
- **Drawbacks**:
  - Less flexible (more verbose code).
  - Slower for prototyping.

Dynamic vs. Static Typing - Comparison

Dynamic vs. Static Typing: Pros and Cons

Dynamic Typing	Static Typing
Flexible and easy to write	Catches errors earlier
Faster for prototyping	Better performance
Potential runtime errors	Type-safe, fewer bugs
Less clear for large codebases	More predictable behavior

Introduction to Type Hints

What are Type Hints?

- Type hints allow you to annotate Python code with types.

- Introduced in **PEP 484** (Python 3.5) and evolved in later versions.
- **Syntax**:

```python
def greet(name: str) -> str:
    return "Hello " + name
```

- They don't change how the code runs but help with static analysis and readability.

Python stores the annotations in the `__annotations__` attribute of the object.

```
```python
>>> def greet(name: str) -> str:
...     return 'Hello, ' + name

>>> greet.__annotations__
```
```
{'name': str, 'return': str}
```

Real-World Use Case

Real-World Use Case: Large Codebases

- Type hints help maintain clarity in large projects with many contributors.
- Example: API development.

```python
def fetch_user(user_id: int) -> dict[str, str]:
    # Fetch user data as a dictionary
    return {"name": "Alice", "id": str(user_id)}
```

- Type hints help other developers know the expected types without needing to read through the entire function.

Evolution of Type Hints

Evolution of Type Hints in Python

- **PEP 484**: Introduced basic type annotations (Python 3.5).
- **PEP 526**: Variable annotations (Python 3.6).
- **PEP 563**: Postponed evaluation of annotations (Python 3.7).
- **PEP 585**: Built-in generics like `list`, `dict`, and `tuple` (Python 3.9).
- **PEP 604**: Simplified Union types (`int | str` instead of `Union[int, str]`) in Python 3.10.

Basic Type Annotations

Basic Type Annotations in Python

- You can annotate functions, variables, and class attributes.

```python
# Function with type hints
def add(a: int, b: int) -> int:
    return a + b

# Variable annotations
```
```

```python
count: int = 0
name: str = "Alice"

from typing import ClassVar

# Class annotations
class Person:
    species: ClassVar[str] = 'human'
    def __init__(self, name: str, age: int) -> None:
        self.name = name
        self.age = age
```

- Common annotations:
  - **int**, **float**, **str**, **bool**
  - Collections: **list**, **dict**, **tuple**

Benefits of Type Hints

1. **Readability**: Makes it easier for others to understand the code.
2. **Tooling Support**: Static analyzers (e.g., `mypy`) can catch bugs early.
3. **Documentation**: Serves as in-line documentation for expected types.
4. **Refactoring**: Type hints make large refactorings safer and easier.

Static Analysis Tools - Real-World Use Case

Real-World Use Case: Static Analysis with mypy

- `mypy` helps catch type errors in large-scale projects before runtime.
- Example:

```
$ mypy script.py
script.py: error: Argument 1 to "add" has incompatible type "str"; expected "int"
```

- Without running the program, `mypy` catches a mismatch between expected and actual types.

```python
# Example in VSCode
import pandas as pd

def add_name_col(df_: pd.DataFrame, col: str, name: str) -> pd.DataFrame:
    return df_.assign(**{col: name})

def remove_name_col(df_, col):
    return df_.drop(columns=[col])


# look at tab completion after a period
add_name_col(pd.DataFrame(), 'name', 'John')

remove_name_col(pd.DataFrame(), 'name')
```

Core Python Typing Constructs

Annotating Functions and Variables

- **Function signatures** allow you to specify the expected types for parameters and return values.
- Syntax for function annotations:

```
def greet(name: str) -> str:
    return "Hello " + name
```

- The parameter `name` is expected to be a `str`, and the function returns a `str`.

Variable Annotations

- Variable annotations specify the type of a variable, improving code clarity.

```
# Without annotation
x = 42  # type inferred

# With annotation
x: int = 42

# Without value
y: str
```

Typing Complex Data Structures

- Python's typing module provides support for complex data structures like `List`, `Dict`, and `Tuple`.
- If you are using Python 3.9+, you can use built-in generics.

```
# A list of integers
numbers: list[int] = [1, 2, 3]

# A dictionary with string keys and integer values
user_ages: dict[str, int] = {"Alice": 30, "Bob": 25}

# A tuple containing a string and an integer
person: tuple[str, int] = ("Alice", 30)
```

Generics in Typing

- The `typing` module allows for generics, which let you define reusable types for different data structures.
- Also useful when you need to refer to a class/type inside of its definition.
- Key point is that it remembers the type of the input and matches the output type.

```
from typing import TypeVar, List

T = TypeVar('T')
```

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

- **Generics** make functions more flexible, allowing them to work with various data types while maintaining type safety.

Optional and Union Types

- `Optional` is a shorthand for a type that can either be a given type or `None`.
  - In Python 3.10+, you can use `str | None` instead of `Optional[str]`.
- `Union` allows for a type to be one of several specified types.
  - In Python 3.10+, you can use `int | str` instead of `Union[int, str]`.

```python
# Optional type
def get_name(user_id: int|None) -> str|None:
    if user_id is None:
        return None
    return "Alice"

# Union type
def parse_data(data: str|int) -> str:
    if isinstance(data, int):
        return str(data)
    return data
```

Real-World Use Case: Handling Optional Values

- In web applications, optional types are useful for handling missing or null data.

```python
def get_user_email(user_id: int) -> str|None:
    # Fetch email if exists, otherwise return None
    return database.get(user_id, {}).get('email', None)
```

Add Types to Sk-Stepwise

Change:

```python
    def fit(self, X, y):
```

to:

```python
    def fit(self, X: pd.DataFrame, y: pd.Series) -> StepwiseHyperoptOptimizer:
```

This fails in VSCode, 'StepwiseHyperoptOptimizer' not defined. So, add:

```python
from typing import TypeVar

SHO = TypeVar('SHO', bound='StepwiseHyperoptOptimizer')

...

    def fit(self, X: pd.DataFrame, y: pd.Series) -> SHO:
```

In Python 3.11, you can use PEP 673 (Self Type) to avoid the need for the TypeVar:

```python
from typing import Self
    def fit(self, X: pd.DataFrame, y: pd.Series) -> Self:
```

Advanced Python Typing

Type Aliases and Custom Types

- Type aliases allow you to create descriptive names for types, improving readability.

```python
# Define a type alias for a list of strings
Usernames = list[str]

def get_usernames() -> Usernames:
    return ["alice", "bob", "charlie"]
```

`TypedDict` for Custom Types

- **`TypedDict`** allows you to define a dictionary with fixed keys and value types.

```python
from typing import TypedDict

class User(TypedDict):
    name: str
    age: int

user: User = {"name": "Alice", "age": 30}
```

- Useful for working with JSON-like data structures where specific keys are expected.

Working with Callable and Any

- `Callable` allows you to define the expected signature of a function.

```python
from typing import Callable

# A function that accepts a callable with specific signature
def process(func: Callable[[int, int], int]) -> int:
    return func(1, 2)
```

- This ensures that only functions with the matching signature can be passed in.

Using `Any`

- `Any` allows for any type to be passed in, bypassing type checking.

```python
from typing import Any

def process_data(data: Any) -> None:
    print(data)
```

- **Caution**: `Any` disables static type checking, so use it sparingly.
- Best used for functions working with highly dynamic data (e.g., deserialized JSON).

Type Variables and Generics

- `TypeVar` enables the creation of generic types for functions or classes.

```
from typing import TypeVar, List

T = TypeVar('T')

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

- Allows the function to operate on a list of any type while maintaining type safety.

Applying Constraints to `TypeVar`

- You can apply constraints to `TypeVar` to limit the allowed types.

```
from typing import TypeVar

# Constrained to int and float
T = TypeVar('T', int, float)

def add(x: T, y: T) -> T:
    return x + y
```

- This ensures that only specific types can be passed into the function.

Note that `typing` includes a `AnyStr` type variable that can represent both `str` and `bytes`.

Annotating Class Methods and Attributes

- You can annotate class attributes and methods to provide clear expectations of their types.

```
from typing import ClassVar

class Employee:
    version: ClassVar[str] = "1.0"

    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age
```

- Helps ensure that the class is used correctly and enforces consistency.

Function and Generator Type Hints

- You can use `Callable` to type hint functions that accept other functions as arguments or return functions.

```
from typing import Callable
```

```python
def make_adder(x: int) -> Callable[[int], int]:
    return lambda y: x + y
```

- `Callable[[str], int]` means a function takes an `str` and returns an `int`.

Generator and Iterator Annotations

- For generator functions, use `Iterator` from `typing`.

```python
from typing import Iterator

def countdown(n: int) -> Iterator[int]:
    while n > 0:
        yield n
        n -= 1
```

- This tells the type checker that the function returns an iterator over `int`.

Sk-stepwise `clean_int_params` method

Change:

```python
    def clean_int_params(self, params):
        int_vals = ['max_depth', 'reg_alpha']
        return {k: int(v) if k in int_vals else v for k, v in params.items()}
```

to:

```python
    def clean_int_params(self, params: dict) -> dict:
```

or better.

```python
PARAM = int|float|str|bool
...
    def clean_int_params(self, params: dict[str, PARAM]) -> dict[str, PARAM]:
```

Sk-stepwise `objective` method

Change:

```python
    def objective(self, params):
```

to:

```python
    def objective(self, params: dict[str, PARAM]) -> float:
```

`predict` and `score`

Make sure that you define `MatrixLike` as a type with a `TypeAlias`! This is needed when you use a custom type.

```python
from typing import TypeAlias
from scipy.sparse import spmatrix
MatrixLike: TypeAlias = np.ndarray | pd.DataFrame | spmatrix
ArrayLike: TypeAlias = numpy.typing.ArrayLike
```

```python
    def predict(self, X: pd.DataFrame) -> ArrayLike:
        return self.model.predict(X)

    def score(self, X: pd.DataFrame, y: pd.Series) -> float:
        return self.model.score(X, y)
```

Results from mypy

```
% uv run mypy  --ignore-missing-imports src tests/test_basic.py --strict
src/sk_stepwise/__init__.py:15: error: Missing type parameters for generic type "ndarray"  [type-arg]
src/sk_stepwise/__init__.py:26: error: Function is missing a type annotation for one or more arguments  [no-untyped-def]
src/sk_stepwise/__init__.py:35: error: Class cannot subclass "BaseEstimator" (has type "Any")  [misc]
src/sk_stepwise/__init__.py:35: error: Class cannot subclass "MetaEstimatorMixin" (has type "Any")  [misc]
src/sk_stepwise/__init__.py:42: error: Function is missing a type annotation  [no-untyped-def]
src/sk_stepwise/__init__.py:68: error: Returning Any from function declared to return "float"  [no-any-return]
src/sk_stepwise/__init__.py:104:  error:  Returning  Any  from  function  declared  to  return  "Buffer  |  _SupportsArray[dtype[Any]]  |
_NestedSequence[_SupportsArray[dtype[Any]]] | bool | int | float | complex | str | bytes | _NestedSequence[bool | int | float | complex
| str | bytes]"  [no-any-return]
src/sk_stepwise/__init__.py:107: error: Returning Any from function declared to return "float"  [no-any-return]
tests/test_basic.py:5: error: Function is missing a return type annotation  [no-untyped-def]
tests/test_basic.py:5: note: Use "-> None" if function does not return a value
tests/test_basic.py:7: error: Need type annotation for "rounds" (hint: "rounds: list[<type>] = ...")  [var-annotated]
tests/test_basic.py:8: error: Call to untyped function "StepwiseHyperoptOptimizer" in typed context  [no-untyped-call]
tests/test_basic.py:11: error: Function is missing a return type annotation  [no-untyped-def]
tests/test_basic.py:11: note: Use "-> None" if function does not return a value
tests/test_basic.py:15: error: Non-overlapping equality check (left operand type: "Literal['matt']", right operand type: "Literal['fred']")
[comparison-overlap]
tests/test_basic.py:18: error: Function is missing a type annotation  [no-untyped-def]
tests/test_basic.py:22: error: Function is missing a return type annotation  [no-untyped-def]
tests/test_basic.py:22: note: Use "-> None" if function does not return a value
tests/test_basic.py:25: error: Need type annotation for "rounds" (hint: "rounds: list[<type>] = ...")  [var-annotated]
tests/test_basic.py:26: error: Call to untyped function "StepwiseHyperoptOptimizer" in typed context  [no-untyped-call]
tests/test_basic.py:32: error: Function is missing a return type annotation  [no-untyped-def]
tests/test_basic.py:32: note: Use "-> None" if function does not return a value
Found 18 errors in 2 files (checked 2 source files)
```

Structural Typing and Protocols

- **Structural Typing** is based on the shape of an object rather than its explicit inheritance from a class.
- If an object has the required attributes and methods, it satisfies the type.

```python
class Dog:
    def speak(self) -> str:
        return "Woof"

class Cat:
    def speak(self) -> str:
        return "Meow"

def make_sound(animal: Dog) -> None:
    print(animal.speak())

# Even though Cat doesn't inherit from Dog, it works
make_sound(Cat())  # Output: Meow
```

- In **structural typing**, it doesn't matter that `Cat` isn't a subclass of `Dog`; as long as it has a `speak` method, it is compatible.

Real-World Use Case: Flexible APIs

- Structural typing is useful in building flexible APIs.
- APIs can accept any object that adheres to a "protocol" (i.e., provides the required methods or attributes), regardless of its actual class.

```
class Printer:
    def print(self) -> str:
        return "Printing..."

class Logger:
    def print(self) -> str:
        return "Logging..."

def output_device(device) -> None:
    print(device.print())

output_device(Printer())  # Output: Printing...
output_device(Logger())   # Output: Logging...
```

- The `output_device` function works with any object that has a `print` method.

Introducing Protocols

- **`Protocol`** is a type introduced in **PEP 544** (structural subtyping (static duck typing)) that allows you to define a structural contract for an object.
- A class that implements the specified methods or attributes satisfies the `Protocol`, even without inheritance.
- I heard that best practice is for protocols to have one method. If you need more, create multiple protocols and compose them. But not a way to compose right now.
- Use the ... ellipsis to indicate a method definition.
- In large projects, you may define protocols to standardize how certain interfaces work across various classes without requiring inheritance.

```
from typing import Protocol

class _Connectable(Protocol):
    def connect(self) -> str:
        ...

class SQLDatabase:
    def connect(self) -> str:
        return "Connected to SQL"

class NoSQLDatabase:
    def connect(self) -> str:
        return "Connected to NoSQL"

def connect_to_db(db: _Connectable) -> None:
    print(db.connect())
```

```
connect_to_db(SQLDatabase())   # Output: Connected to SQL
connect_to_db(NoSQLDatabase()) # Output: Connected to NoSQL
```

- Protocols allow different types of databases to work with the same `connect_to_db` function without explicit inheritance.

Add Protocol to sk-stepwise

Add:

```
from typing import Protocol
class _Fitable(Protocol):
    def fit(self, X: MatrixLike, y: ArrayLike) -> Self:
        ...
    def predict(self, X: MatrixLike) -> ArrayLike:
        ...
    def set_params(self, **params) -> Self:
        ...
    def score(self, X: MatrixLike, y: ArrayLike) -> float:
        ...

from hyperopt import hp
from collections.abc import Callable
from hyperopt.pyll.base import SymbolTable

class StepwiseHyperoptOptimizer(BaseEstimator, MetaEstimatorMixin):
    def __init__(self, model: _Fitable,
                 param_space_sequence: list[dict[str, PARAM | SymbolTable]],
                 max_evals_per_step:int=100,
                 cv:int=5,
                 scoring:str|Callable[[ArrayLike, ArrayLike], float]='neg_mean_squared_error',
                 random_state:int=42) -> None:
```

Typing `*args` and `**kwargs`

- Don't use `tuple` or `dict` for `*args` and `**kwargs`
- Use the type of a single element instead
- MS recommends `Unpack[TypedDict]` for `**kwargs`

https://github.com/microsoft/pyright/issues/3002#issuecomment-1046100462

Update my `set_params` method

```
    def set_params(self, **params: PARAM) -> Self:
        ...
```

Type Checking with `mypy`

Introduction to mypy

- `mypy` is a static type checker for Python.
- It checks for type inconsistencies without running the code.

Why use `mypy`?

1. **Catch type errors early**: Prevent runtime type errors.
2. **Improve code clarity**: Type hints make code easier to understand and maintain.
3. **Support for gradual typing**: Works with partially annotated codebases, allowing you to add type hints incrementally.

Setting up `mypy` in Your Python Project

```
uv install --dev mypy
```

Run `mypy` on your code:

```
uv run mypy your_script.py
```

Basic `mypy` Command-Line Usage

- You can run `mypy` on a single file or an entire project.

```
# Checking a single file
uv run mypy script.py

# Checking an entire project
uv run mypy src/
```

Configuring mypy for Flexibility

- You can configure `mypy` to enforce stricter or looser type checking.

```
# Enable strict mode (more comprehensive checks)
uv run mypy --strict script.py
```

- **Strict mode** includes checks for:
  1. Missing type hints
  2. Inconsistent return types

Ignoring Errors

- `# type: ignore` allows you to bypass error
- `# type: ignore [attr-defined]` silences `attr-defined` errors.
- Using `Any` for type hints also bypasses type checking.
- Add `# mypy: ignore-errors` at top to ignore all errors in a file.
- Decorate a function with `@typing.no_type_check` to disable type checking for that function.

Customizing `mypy` Configuration with `pyproject.toml`

- You can configure `mypy` using a `pyproject.toml` file.

```
[tool.mypy]
strict = true
ignore_missing_imports = true
warn_unused_ignores = true
```

- Common options:
  - `ignore_missing_imports`: Avoids errors for missing stubs in third-party libraries.
  - `warn_unused_ignores`: Warns if a `# type: ignore` comment is unnecessary.

Ignoring Errors with Config Files

You can specify module rules in the `mypy` configuration file to ignore errors for specific modules.

Note that ini and `pyproject.toml` syntax is different.

For `pyproject.toml`:

```
[tool.mypy]
strict = true
ignore_missing_imports = true

[[tool.mypy.overrides]]
module = 'tests.*'
ignore_errors = true
```

Understanding `reveal_type`

- **`reveal_type`** is a special function in `mypy` that helps you inspect and debug the inferred type of any expression.
- Insert `reveal_type` into your code to see what type `mypy` infers for a variable or expression.

```
def process(value: int|str):
    reveal_type(value)  # Reveal the inferred type of "value"
    if isinstance(value, int):
        reveal_type(value)  # The type narrows to "int"
    else:
        reveal_type(value)  # The type narrows to "str"
```

Output:

```
$ uv run mypy script.py
script.py:3: note: Revealed type is 'Union[int, str]'
script.py:5: note: Revealed type is 'int'
script.py:7: note: Revealed type is 'str'
```

Why use `reveal_type`? 1. **Debugging**: Helps you understand what type `mypy` is inferring, useful for debugging complex types. 2. **Type Narrowing**: Confirms how `mypy` narrows types after `if` conditions, `isinstance` checks, etc. 3. **Learning Tool**: Useful for learning and improving your understanding of Python's type system.

- **Note**: `reveal_type` is not part of Python's runtime, and it won't affect your code. It only works during type checking with `mypy`.

Ignoring Type Errors with `# type: ignore`

- If a type error is unavoidable or irrelevant, use `# type: ignore` to bypass it.

```python
# A case where type checking might be bypassed
def process_data(data: Any) -> None:
    print(data)

process_data(123)  # type: ignore
```

- **Caution**: Use sparingly to avoid masking real issues.
- Adding `# type: ignore` silences `mypy` warnings for that specific line.

Create a Stub File

- **Stub files** (`.pyi`) are Python files that contain type hints without any implementation.
- Move type hints to a stub file to separate them from the actual code.
- Only include type hints in the stub file, not the implementation. Place `...` for functions and methods.

Running mypy in CI Pipelines

- Add `mypy` to your continuous integration (CI) pipeline to enforce type checking on every commit.
- Example: GitHub Actions

```yaml
# .github/workflows/mypy.yml
name: uv run mypy check

on: [push, pull_request]

jobs:
  type-check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: "3.10"
      - name: Install dependencies
        run: |
          uv install --dev mypy
      - name: Run mypy
        run: |
          uv run mypy src/
```

- **Benefit**: Automatically catch type issues before merging code into production.