# Typed Python

*Jan Mayer - 2023*

# What we will cover

- Static type checking with `mypy`
- Structures to handle data
- Runtime input validation with `pydantic`

# What is the problem here?

```
def important_business_logic(bar, baz):
    # Way too many lines
    # of badly written code
    # ...
    return foo
```

# Ahh, this is better!

```
def get_vehicle(vin: str, volumes: list[int]) -> Optional[Vehicle]:
    # Way too many lines
    # of badly written code
    # ...
    return foo
```

# Type Hints are Documentation

```
# test.py
x: str = 1  # Will still work
print(x)
print(type(x))
```

Python will ignore annotations

```
$ python test.py
1
<class 'int'>
```

# Type Hints are Documentation

> "It should also be emphasized that Python will remain a dynamically typed language, and the authors have no desire to ever make type hints mandatory, even by convention. (PEP484)"

# Type hints are More than Documentation

- help write shorter and verifiable documentation
- make coupling more explicit
- force being explicit about data structures
- simplify your code
- **improve the developer experience**

# Verifying Type Hints

```
$ pip install mypy
```

```
# test.py
x: str = 1  # Note: Variables are rarely annotated this way
print(x)
```

```
$ mypy test.py
test.py:1: error: Incompatible types in assignment (expression has type "int", variable has type "str")
Found 1 error in 1 file (checked 1 source file)
```

# Requirements

- Python 3.9+
  `list[str]`
- With some minor changes: Python 3.5+
  ```
  from typing import List
  List[str]
  ```
- Python 2.x, and <3.5 somewhat possible ...
  - Stop it, get some help

# Types

# Standard types

- Usual suspects: `str`, `int`, `float`, `bool`
- Collections:

```
list[int] = [1, 2, 4, 4, 5, 4, 1]
set[int] = {1, 2, 4}
dict[int, str] = {1: "Hello", 2: "There"}
tuple[int, str, float] = (1, "Pi", 3.1415)
```

- More Generic: `Sequence` and `Iterable` (includes `str`!), `Mapping` (e.g. dicts)

# Nothing

- `None` : I return, but the thing I return is nothing.

```
def returns_nothing(x: str) -> None:
    print(x)
```

- `NoReturn` : I will not return, ever.

```
def always_throws() -> NoReturn:
    raise RuntimeError('nope')
```

# Union

```
from typing import Union

# Parameter needs to be int OR float
def new_divide(i: Union[int, float]) -> float:
  return i/2
```

Shorter, alternative syntax (3.10+):

```
def new_divide(i: int | float) -> float:
  return i/2
```

# Optional

Documents that a function might or might not return a value. Mypy enforces that the None case is handled.

```
from typing import Optional

def get_vehicle_id(vin: str) -> Optional[str]:
    # Might return a string or None

must_pass_a_string(get_vehicle_id("abc"))
```

```
mypy.py:11: error: Argument 1 to "must_pass_a_string"
   has incompatible type "Optional[str]"; expected "str"
```

# Any

`Any` indicates that any type is allowed. Use sparingly.

- To silence mypy
  - if unknown at the moment
  - if you don't care / if it doesn't matter
  - temporarily, if hard to figure out or too large
- To actually indicate that all types are allowed

# Literals

```
from typing import Literal

BrandStatus = Literal["OK", "SKIPPED", "UNSUPPORTED_BRAND"]


def brand_check(...) -> BrandStatus:
    if ... :
        return "SKIPPED"
    if ... :
        return "UNSUPPORTED_BRAND"
    return "OK"
```

Bonus: Will get picked up by swagger-generators.

# Aliases

```
VIN = str
Vector = list[int]
Matrix = list[Vector]
BrandStatus = Literal["OK", "SKIPPED", "UNSUPPORTED_BRAND"]
```

Note: Does not check for conversions between aliases

# NewType

```
from typing import NewType

VIN = NewType(VIN, str)


def get_vehicle(vin: VIN) -> Vehicle:
    ...

vehicle_a = get_vehicle(VIN("BAU1"))
vehicle_b = get_vehicle("BAU1")
```

```
mypy.py:10: error: Argument 1 to "get_vehicle" has
    incompatible type "str"; expected "VIN"
```

# TypeVar

```
T = TypeVar('T')  # Can be anything
S = TypeVar('S', bound=str)  # Can be any subtype of str
A = TypeVar('A', str, bytes)  # Must be exactly str or bytes
```

```
from typing import TypeVar

T = TypeVar('T')


def get_first(inputs: list[T]) -> T:
    return inputs[0]
```

# Callable

```
from typing import Callable

def stupid_example(a: int, b: int, fun: Callable[[int,int],int]) -> None:
    print(fun(a,b))

def minus(x: int, y: int) -> int:
    return y - x

def plus(x: int, y: int) -> int:
    return y + x


stupid_example(1, 2, minus)
stupid_example(1, 2, plus)
```

Also for returning functions / lambdas

# ABC: Abstract Base Classes

Roughly equivalent to `Interface` in Java.

```
from abc import ABC, abstractmethod

class Pricing(ABC):
    @abstractmethod
    def get_offerings(self, tenant: str, brand: str) -> str:
        raise NotImplementedError

class SomeClassThatNeedsToKnowTheInterface(Pricing):
    # OK, Implements the abstract method
    def get_offerings(self, tenant: str, brand: str) -> str:
        # ...
```

# Protocol

In duck typing, an object is of a given type if it has all methods and properties required by that type.

```
from typing import Protocol

class Pricing(Protocol):
    def get_offerings(self, tenant: str, brand: str) -> str:
        ...  # Not a comment typically ... is used

class SomeClassFromSomewhere:  # Does not inherit from anything!
    # OK, has the required function
    def get_offerings(self, tenant: str, brand: str) -> str:
        # ...
```

# ABC VS Protocol

- ABC-Interfaces:
  - Error on class instantiation
  - Subclasses belong to the interface
    - Explicit dependencies
- Protocols:
  - "Belongs" to where the classes are used
    - Can apply to third party libraries
    - Allow splitting the interface
  - Reduces coupling

# Common Problems

- Type hints artificially limit your code
  - ... but thats good!
- Not all libraries have type hints

```
pip install types-requests
import isodate  # type: ignore
```

- Sometimes, mypy needs some help

```
def get_expire_time(rate_limit_data: func.DocumentList) -> datetime.datetime:
    last_access = datetime.datetime.fromisoformat(rate_limit_data[0].get("timestamp"))
    duration: datetime.timedelta = isodate.parse_duration(os.environ.get("RATE_LIMIT"))
    return last_access + duration
```

# Dataframes

```
from pyspark.sql import DataFrame

def generate_data_product(df: DataFrame) -> DataFrame:
    ...
```

Sadly, does not help you with dataframes at all.

Mypy can not tell what the schema of the dataframes are.

```
def something(df: DataFrame, *, time: datetime, things: list[str]) -> DataFrame:
    ...
```

# Typing TLDR

- Use Types. Types very good.
- You know what types your function expects.
  - Just write them down.
  - Thank me later.

# Storing Data

# Tuple / namedtuple / NamedTuple

```
x: tuple[int, float, str] = 5, 4.20, "TWTR"
print(x[2])


from collections import namedtuple

T: namedtuple("T", ["a", "b", "c"])
x = T(5, 4.20, "TWTR")
print(x.b)


from typing import NamedTuple

class T(NamedTuple):
    a: int
    b: float
    c: str

x = T(5, 4.20, "TWTR")
print(x.b)
```

# Dict / Typed Dict

```
a = {'x': 1, 'y': 2, 'label': 'good'}

# Type?
dict[str, int | str ]  # Meh.
```

```
from typing import TypedDict

class Point2D(TypedDict):
    x: int
    y: int
    label: str

a: Point2D = {'x': 1, 'y': 2, 'label': 'good'}  # OK
b: Point2D = {'z': 3, 'label': 'bad'}           # Fails type check
```

# P(I)ain Class

```
class Comment:
    def __init__(self, id: int, text: str):
        self.id: int = id
        self.text: str = text

    def __repr__(self):
        return "{}(id={}, text={})".format(self.__class__.__name__, self.id, self.text)

    def __eq__(self, other):
        if other.__class__ is self.__class__:
            return (self.id, self.text) == (other.id, other.text)
        else:
            return NotImplemented

    def __ne__(self, other):
        result = self.__eq__(other)
        if result is NotImplemented:
            return NotImplemented
        else:
            return not result

    def __hash__(self):
        return hash((self.__class__, self.id, self.text))

    def __lt__(self, other):
        if other.__class__ is self.__class__:
            return (self.id, self.text) < (other.id, other.text)
        else:
            return NotImplemented

    def __le__(self, other):
        if other.__class__ is self.__class__:
            return (self.id, self.text) <= (other.id, other.text)
        else:
            return NotImplemented

    def __gt__(self, other):
        if other.__class__ is self.__class__:
            return (self.id, self.text) > (other.id, other.text)
        else:
            return NotImplemented

    def __ge__(self, other):
        if other.__class__ is self.__class__:
            return (self.id, self.text) >= (other.id, other.text)
        else:
            return NotImplemented
```

# Dataclass

```python
from dataclasses import dataclass, field

@dataclass(frozen=True, order=True, slots=True)  # Note: Freezing adds hash function
class Comment:
    id: int
    text: str = ""
    # replies list[int] = [] <-- Common error!
    replies: list[int] = field(default_factory=list)  # Also to change behavior
```

# Pydantic

Pydantic is a very opinionated library for **parsing**

- will try to convert types
- will check types at runtime
- can be wasteful
- extremely powerful

# Example

```
def get_aircraft_metadata(ac_tail_no):
    response = requests.get(
        f"{BASE_URL}/aircrafts/{ac_tail_no}",
        headers={"Accept": "application/json"},
    )
    response.raise_for_status()
    return response.json()
```

# Example

```
class Aircraft(BaseModel):
    ac_tail_no: Field(regex="^[A-Z]{2}-[A-Z0-9]{4}$")
    status: Literal["IN-SERVICE", "GROUNDED", "IN-REPAIR"]
    birthdate: datetime.date

    @validator("birthdate", pre=True)
    def parse_birthdate(cls, value: str) -> datetime.date:
        return datetime.strptime(value,"%d/%m/%Y").date()


def get_aircraft_metadata(ac_tail_no: str) -> Aircraft:
    response = requests.get(
        f"{BASE_URL}/aircrafts/{ac_tail_no}",
        headers={"Accept": "application/json"},
    )
    response.raise_for_status()
    return Aircraft(**response.json())
```

# TLDR: Storing Data

- Use `dataclass`es and `pydantic.BaseModel`s
- Only use raw dicts for simple mappings
  - something you would iterate over
  - something with limited scope