



# Static Typing in Python

Alex Molas, Senior Data Scientist @ Wallapop



# About Me

Physicist

Data Scientist - Wallapop (Search Team)

Blogger ([www.alexmolass.com](http://www.alexmolass.com))






# Setup

1. Clone the repo (<https://github.com/alexmolas/python-static-typing>)
2. Install a virtual environment with **python 3.12**
3. Install the requirements.txt



# What's typing?

- **Dynamic typing:** Python is a dynamically typed language. This means it does type checking only as code runs, and that the type of a variable can change over its lifetime.



```
if False:
    1 + "two" # This line never runs, so no TypeError is raised
else:
    1 + 2
```



# What's typing?

- **Static typing:** type checks are performed before running the program. Usually when the program is compiled.



```
String thing;  
thing = "Hello";
```



# What's typing?

- **Duck typing:** “if it walks like a duck and it quacks like a duck, then it must be a duck”. This means that in Python you don’t check the type of an object, you only check if it implements the methods that are needed.

```
class Duck:
    def swim(self):
        print("Duck swimming")
    def fly(self):
        print("Duck flying")

class Whale:
    def swim(self):
        print("Whale swimming")

for animal in [Duck(), Whale()]:
    animal.swim()
```



# What's typing?

- **Dynamic typing:** Python is a dynamically typed language. This means it does type checking only as code runs, and that the type of a variable can change over its lifetime.
- **Static typing:** type checks are performed before running the program. Usually when the program is compiled.
- **Duck typing:** “if it walks like a duck and it quacks like a duck, then it must be a duck”. This means that in Python you don't check the type of an object, you only check if it implements the methods that are needed.



# Types in Python

Python has some builtin types

- *int, float, complex, list, tuple, range, bool, string, set, dict, etc.*

Python also allows you to define your own types via classes.

```
class Duck:
    def swim(self):
        print("Duck swimming")
    def fly(self):
        print("Duck flying")
```






# Type Hinting

Even if Python is a **dynamically typed** language it allows for **type hints**.

Type hinting consists in **annotating** variables, methods, function arguments, and return values with types.



```
def function(arg_1: type_1, arg_2: type_2 = default) -> return_type:
    ...

# example
def add(a: int, b: int = 2) -> int:
    return a + b
```



## 1st exercise!



1. Open `1.1-circumference-no-types.py`
2. Read it carefully.
3. Run it.
4. Add type hints.



## Why are type hints useful?

- Type hints help the IDE to autofill or to recommend completions.
- Work as documentation. You know which type is supposed to be used.
- Can be used to type check your code. Check that types are consistent. Using **mypy**

```
def foo(lst: list) -> None:
    lst.ap
```

A screenshot of a code editor showing a function definition `def foo(lst: list) -> None:` and a line of code `lst.ap` with a cursor. A dropdown menu shows a suggestion for `append` with a small cube icon to its left.

```
mypy python_program.py
```

A screenshot of a terminal window with three colored window control buttons (red, yellow, green) at the top left. The command `mypy python_program.py` is entered in the terminal.



## 2nd exercise

1. Open **2.1-headline-no-types.py** and read it.
2. Run **mypy** over **2.1-headline-no-types.py**. Since it doesn't have types no error is shown.
3. Execute **2.1-headline-no-types.py**. There are some errors 🤖
4. Add types to **2.1-headline-no-types.py**.
5. Run again **mypy**. Does it show some error? Why?
6. Fix **2.1-headline-no-types.py**



# Complex types

- Python allows you to type arguments with complex types.

```
list[TYPE] # type the elements of a list. All the elements have the same type

tuple[TYPE1, TYPE2, etc] # type the elements of a tuple. Can have different types

tuple[TYPE, ...] # if it has multiple elements of the same type

dict[KEY_TYPE, VALUE_TYPE] # to type dicts

int | float | str # union of different types
```




## 3rd exercise

1. Open `3.1-deck-game-no-types.py`
2. Read it and understand it.
3. Run it.
4. Add types.



## Complex types: aliases

- To reduce verbosity we can use define our own types and type aliases



```
type Card = tuple[str, str]
type Deck = list[Card]

def create_deck(shuffle: bool = False) -> list[Card]:
    ...
```



## 4th exercise

1. Open `3.1-deck-game-no-types.py`
2. Define your own type aliases and reduce the verbosity of the code





## Complex types: typing module

- There are some types that are not available in the builtins
- We can import them from the typing module
- For example we can type methods with *Callable[[input\_types], output\_type]*

```
from typing import Callable, Sequence, Any

def select_first_element(x: Sequence[Any]) -> Any:
    # We use Sequence when we don't care if it's a list, tuple, string, etc
    # A Sequence contains items arranged in order and can be accessed by their index
    return x[0]

method: Callable[[Sequence[Any]], Any] = select_first_element
```



## 5th exercise

1. Open `4.1-process-data.py`
2. Read it and understand it.
3. Run it.
4. Add types.



# Pydantic: how to type external data

- How to read a JSON and add type safety?
- Welcome to Pydantic!

```
from pydantic import BaseModel
import json

class Student(BaseModel):
    name: str
    age: int

class Classroom(BaseModel):
    students: list[Student]

# data.json is
# {"students": [{"name": "John", "age": 19}, {"name": "Jane", "age": 25}]}
```

```
with open("data.json", "r") as f:
    json_data = json.load(f)
data = Classroom(**json_data)
```



## Pydantic: complex validations

```
class User(BaseModel):  
    name: str  
    email: str  
  
    @field_validator("email")  
    def validate_email(cls, v):  
        if "@" not in v:  
            raise ValueError("Email must contain @")  
        return v
```



## 6th exercise

1. Open `5.1-json-no-types.py`
2. Read it and understand it.
3. Run it. Why is it failing?
4. Add types and Pydantic.



# Generic

- Sometimes you want to type classes or method for generic types.
- Allow reuse of the same class with different type combinations.

```
class GenericClass[K, V]:  
  def get_value(self, key: K) -> V:  
    ...  
  
instance: GenericClass[int, str] = GenericClass()
```



## 7th exercise

1. Open `6.1-dict-no-types.py`
2. Read and understand it
3. Run it
4. Run mypy over it
5. What could go wrong?
6. Add types using generic



# Your own types

You can define you own types using classes

```
class Animal:
    ...

class Dog(Animal):
    ...

class Cat(Animal):
    ...

def run(animal: Animal):
    ...

dog = Dog()
cat = Cat()
run(dog) # Works
run(cat) # Works
```





# Your own types

Be careful of the Liskov substitution principle

> An object (such as a class) may be replaced by a sub-object (such as a class that extends the first class) without breaking the program

```
class Bird:
    def fly(self) -> str:
        ...

class Pigeon(Bird):
    def fly(self) -> str:
        return "A pigeon can fly"

class Penguin(Bird):
    def fly(self) -> None: # Wrong! Subtypes must return str
        raise ValueError("Penguins do not fly")
```



## 8th exercise

1. Open and read `7.1-animals-types.py`



## More on duck typing: Protocols

- You can define structural subtypes.
- You don't care about subclassing, just about implementing the needed methods.

```
from typing import Protocol

class FlyingAnimal(Protocol):
    def fly(self) -> str:
        ...

class Pigeon:
    def fly(self) -> str:
        return "I'm a pigeon and I can fly"

class FlyingSquirrel:
    def fly(self) -> str:
        return "I'm a flying squirrel and I can fly"

def make_animal_fly(animal: FlyingAnimal):
    animal.fly()

make_animal_fly(Pigeon())
make_animal_fly(FlyingSquirrel())
```



## Advanced examples

- In the folder *advanced* we have some advanced exercises.
- Play with them and have fun!



## Conclusions

- We learned how to add type hints to python.
- Useful for code security, IDE completion, and documentation.
- mypy is a very useful tool to detect bugs in our code.
- Pydantic can help to add hints and data validation.
- Classes can be used as types.
- You can define structural subtypes using Protocols.



## Q&A

If you have any doubt feel free to write me

- [alexmolasmartin@gmail.com](mailto:alexmolasmartin@gmail.com)
- Twitter: @MolasAlex
- Blog: [www.alexmolas.com](http://www.alexmolas.com)



# Sources

- Basic examples:
  - <https://realpython.com/python-type-checking/>
- Advanced examples:
  - <https://wickstrom.tech/2024-05-23-statically-typed-functional-programming-python-312.html>