

facebook

whoami

- Luka Sterbic
- Software Eng @ Facebook London
- <https://github.com/sterbic>
- @lsterbic
- ✈️ #AvGeek 🌎 #PointsAndMiles



Agenda

1. Why is typing important
2. Typing 101
3. Types in the real world
4. Lessons learnt

Why should I even care about typing?

What jobs?

```
def submit(jobs):
    for job in jobs:
        try:
            submit_job(job, priority=job.meta.priority)
            log(job.name, success=True)
        except queue.Full:
            log(job.name, success=False)
```

What jobs?

```
def submit(jobs):
    for job in jobs:
        try:
            submit_job(job, priority=job.meta.priority)
            log(job.name, success=True)
        except queue.Full:
            log(job.name, success=False)
```

What jobs?

```
def submit(jobs):
    for job in jobs:
        try:
            submit_job(job, priority=job.meta.priority)
            log(job.name, success=True)
        except queue.Full:
            log(job.name, success=False)
```

What jobs?

```
def submit(jobs):
    for job in jobs:
        try:
            submit_job(job, priority=job.meta.priority)
            log(job.name, success=True)
        except queue.Full:
            log(job.name, success=False)
```

- Grep for it?

```
[luka:~/fbcodes:0 (e7090bd|remote/master)]$ grep 'class Job' | wc -l
1376
```

What jobs?

```
def submit(jobs):
    for job in jobs:
        try:
            submit_job(job, priority=job.meta.priority)
            log(job.name, success=True)
        except queue.Full:
            log(job.name, success=False)
```

```
import scheduler
scheduler.submit()
```

```
from scheduler import submit
submit()
```

What jobs?

```
def submit(jobs):
    for job in jobs:
        try:
            submit_job(job, priority=job.meta.priority)
            log(job.name, success=True)
        except queue.Full:
            log(job.name, success=False)
```

```
from scheduler import submit
submit(create_jobs_10_modules_away())
```

What jobs?

```
def submit(jobs):
    for job in jobs:
        print(type(job).__name__)
        print([
            name for name in dir(job)
            if not name.startswith("__")
        ])
    try:
        submit_job(job, priority=job.meta.priority)
        log(job.name, success=True)
    except queue.Full:
        log(job.name, success=False)
```

What jobs?

```
from typing import Iterable
from fb.scheduler.jobs import Job

def submit(jobs: Iterable[Job]) -> None:
    for job in jobs:
        try:
            submit_job(job, priority=job.meta.priority)
            log(job.name, success=True)
        except queue.Full:
            log(job.name, success=False)
```

Scale my team @FB

- Bootcamp 
- New hires do tasks for multiple teams
 - Team decision after ~6 weeks
 - Engineers can pick any team with headcount 
-  eng productive -->  likely to join
- Applies to Open Source too!

Are Python types Pythonic?

Are Python types Pythonic?

Yes!

> **import** this

...

Explicit is better than implicit.

...

Readability counts.

...

In the face of ambiguity, refuse the temptation to guess.

- If nothing else, because Guido said so

.

Is Python turning into Java?

No

- Inspired by types in other languages
- Typing is still optional
- Has no impact at runtime (*)
- Typing errors can be ignored

Typing 101

How did it start?

- PEP 3107 - Function Annotations (2006)
 - By Collin Winter and Tony Lownds
- PEP 484 - Type Hints (2014)
 - By Guido van Rossum, Jukka Lehtosalo, Łukasz Langa

```
def func(arg: awesome_thing) -> new_thing():  
    ...
```

What needs to be annotated?

```
def my_print(message):  
    print(f"< {message} >")
```

```
class Square:  
    def __init__(self, a):  
        self.__a = a
```

```
def my_print(message: str) -> None:  
    print(f"< {message} >")
```

```
class Square:  
    def __init__(self, a: int) -> None:  
        self.__a = a
```

What needs to be annotated?

```
def my_print(message):  
    print(f"< {message} >")
```

```
class Square:  
    def __init__(self, a):  
        self.__a = a
```

```
def my_print(message: str) -> None:  
    print(f"< {message} >")
```

```
class Square:  
    def __init__(self, a: int) -> None:  
        self.__a = a
```

What needs to be annotated?

- In some cases, variables

```
dictionary = {}  
# complex code  
# nested if-else blocks  
# function calls  
func(dictionary) # expects str keys
```

What needs to be annotated?

- In some cases, variables

```
dictionary: Dict[str, int] = {}  
# complex code  
# nested if-else blocks  
# function calls  
func(dictionary) # expects str keys
```

- For Python < 3.6

```
dictionary = {} # type: Dict[str, int]
```

Collections

```
from typing import List, Set, Dict, Tuple

def active_users_per_country(
    users: List[Tuple[int, str]], # user_id / country pairs
    test_users: Set[int],        # don't count test users
) -> Dict[str, int]:
    counter = defaultdict(int)
    for user_id, country in users:
        if user_id not in test_users:
            counter[country] += 1
    return counter
```

Collections

```
from typing import List, Set, Dict, Tuple

def active_users_per_country(
    users: List[Tuple[int, str]], # user_id / country pairs
    test_users: Set[int],        # don't count test users
) -> Dict[str, int]:
    counter = defaultdict(int)
    for user_id, country in users:
        if user_id not in test_users:
            counter[country] += 1
    return counter
```

Collections

```
from typing import List, Set, Dict, Tuple

def active_users_per_country(
    users: List[Tuple[int, str]], # user_id / country pairs
    test_users: Set[int],        # don't count test users
) -> Dict[str, int]:
    counter = defaultdict(int)
    for user_id, country in users:
        if user_id not in test_users:
            counter[country] += 1
    return counter
```

Collections

```
from typing import List, Set, Dict, Tuple

def active_users_per_country(
    users: List[Tuple[int, str]], # user_id / country pairs
    test_users: Set[int],        # don't count test users
) -> Dict[str, int]:
    counter = defaultdict(int)
    for user_id, country in users:
        if user_id not in test_users:
            counter[country] += 1
    return counter
```

Unions

```
from typing import Union

def get_by_id(id: int) -> Union[User, Page]:
    pass
```

```
luka: User = get_by_id(1451268501)
PyConZA: Page = get_by_id(952938054729683)
probably_none = get_by_id(-1)
```

Unions

```
from typing import Union

def get_by_id(id: int) -> Union[User, Page, None]:
    pass
```

```
luka: User = get_by_id(1451268501)
PyConZA: Page = get_by_id(952938054729683)
probably_none = get_by_id(-1)
```

Unions

```
from typing import Union, Optional

def get_by_id(id: int) -> Optional[Union[User, Page]]:
    pass
```

```
luka: User = get_by_id(1451268501)
PyConZA: Page = get_by_id(952938054729683)
probably_none = get_by_id(-1)
```

```
from typing import Type

class BaseClass:
    pass

class DerivedClass(BaseClass):
    pass

def factory(clazz: Type[BaseClass]) -> BaseClass:
    return clazz()

base = factory(BaseClass)  # BaseClass
derived = factory(DerivedClass)  # BaseClass
```

TypeVars

```
from typing import Type, TypeVar

T = TypeVar("T", bound=BaseClass)

def factory(clazz: Type[T]) -> T:
    return clazz()

base = factory(BaseClass) # BaseClass
derived = factory(DerivedClass) # DerivedClass
```

The Type Checker

```
# test.py
import math
ten = math.sqrt(100)
ten = math.sqrt("100")
```

```
$ mypy test.py
```

test.py:4: error:
Argument 1 to "sqrt" has incompatible type "str"; expected "float"

- Pyre --> <https://github.com/facebook/pyre-check>

Typing in the real world

The Easy Bit

New projects

1. 100% typed coverage from Day 1
2. Type checker in CI
3. Profit!

The Hard Part

Existing (Large) Projects

1. ... first understand gradual typing

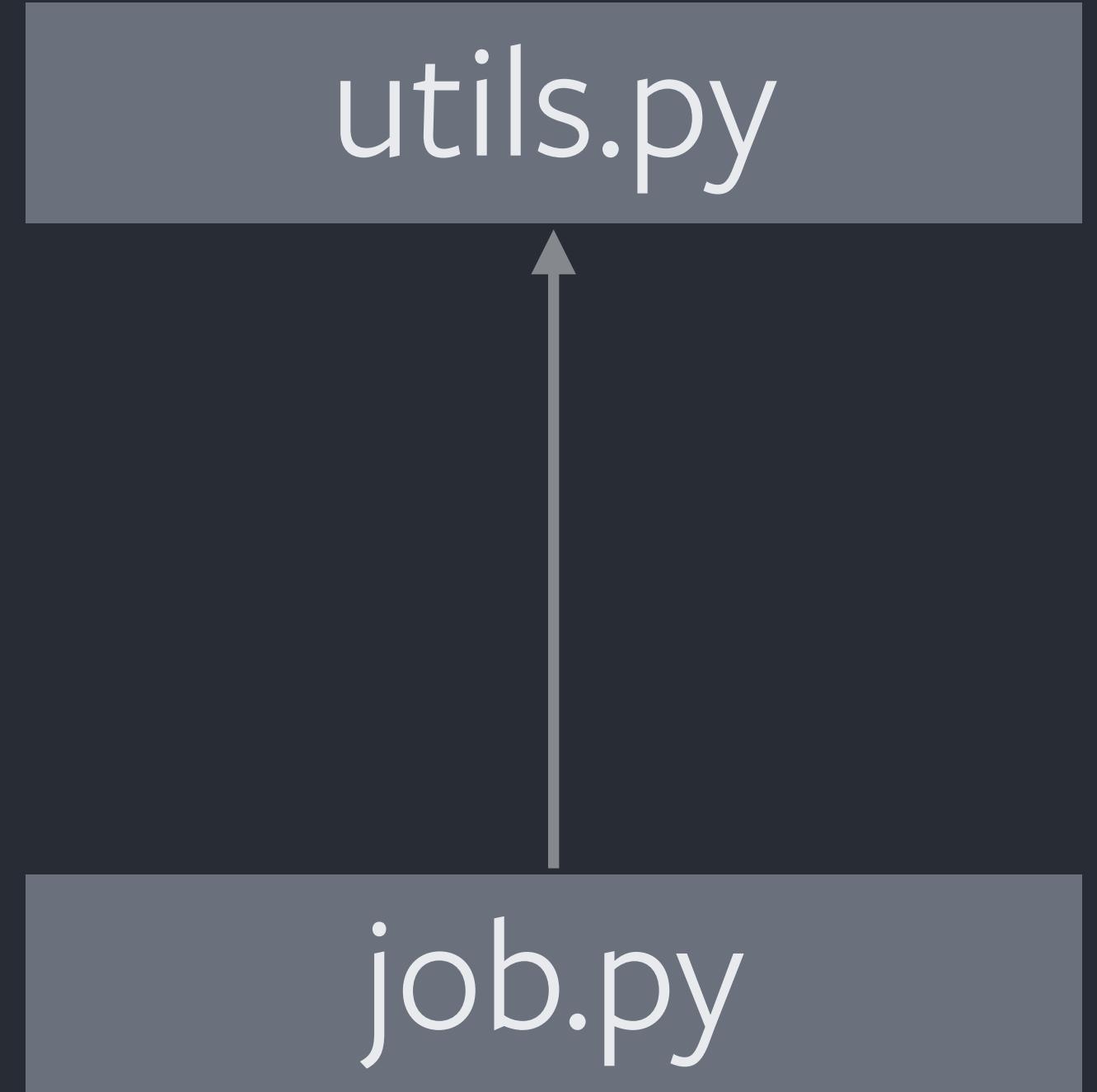
utils.py



job.py

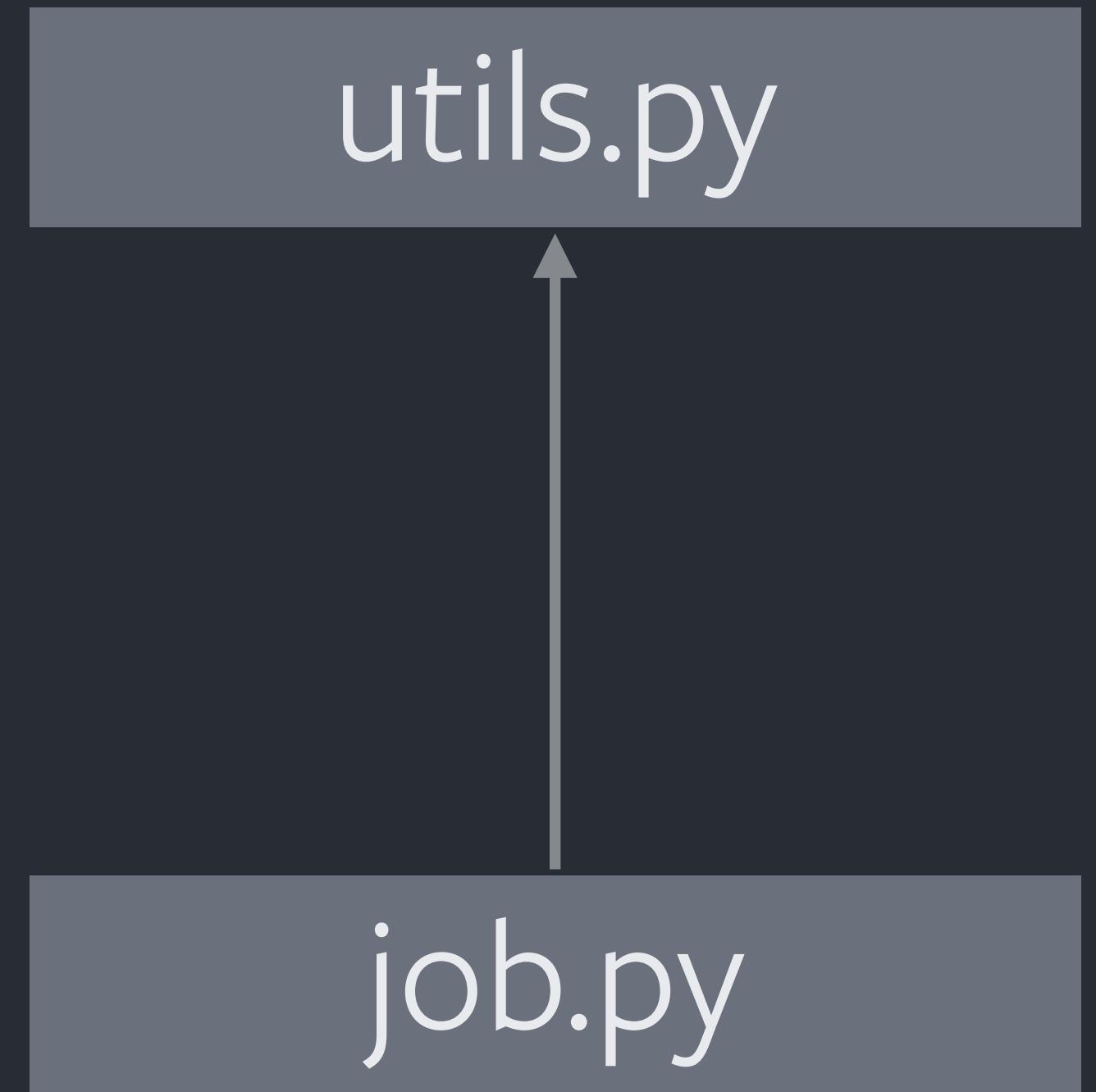
```
# utils.py
import json
def pretty_print(job: Job) -> None:
    print(json.dumps(job, indent=4))
```

```
# job.py
from utils import pretty_print
def create_job():
    job = Job()
    pretty_print("job created!")
    return job
```



```
# utils.py
import json
def pretty_print(job: Job) -> None:
    print(json.dumps(job, indent=4))
```

```
# job.py
from utils import pretty_print
def create_job() -> Job:
    job = Job()
    pretty_print("job created!")
    return job
```



error: Argument 1 to "pretty_print" has incompatible type "str"; expected "Job"

The Hard Part Bit

Existing (Large) Projects

1. Type common libraries --> utils.py
2. Try running the type checker
3. Type checker in CI
4. Ask people to add types to new code
5. Use MonkeyType

MonkeyType

<https://github.com/lnstagram/MonkeyType>

1. Collects types at runtime and logs them
2. Generates type stubs
3. Can even apply back type information to your code

Stubs

```
1 # Stubs for math
2 # See: http://docs.python.org/2/library/math.html
3
4 from typing import Tuple, Iterable, SupportsFloat, SupportsInt
5
6 import sys
7
8 e = ... # type: float
9 pi = ... # type: float
10 if sys.version_info >= (3, 5):
11     inf = ... # type: float
12     nan = ... # type: float
13 if sys.version_info >= (3, 6):
14     tau = ... # type: float
15
16 def acos(x: SupportsFloat) -> float: ...
17 def acosh(x: SupportsFloat) -> float: ...
18 def asin(x: SupportsFloat) -> float: ...
19 def asinh(x: SupportsFloat) -> float: ...
20 def atan(x: SupportsFloat) -> float: ...
21 def atan2(y: SupportsFloat, x: SupportsFloat) -> float: ...
22 def atanh(x: SupportsFloat) -> float: ...
```

MonkeyType

<https://github.com/lnstagram/MonkeyType>

```
$ pip install MonkeyType
```

```
$ monkeytype run main.py
```

```
$ monkeytype stub utils
def pretty_print(job: str) -> None: ...
```

```
$ monkeytype apply utils
```

Lessons learnt

Where did the ducks end up?

Duck typing in the typed world

```
def duck_things(duck: Duck) -> None:  
    duck.quack()  
    duck.swim()
```

```
class PharaohDuck:  
    def quack(self) -> str:  
        return "quaaack"  
    def swim(self) -> str:  
        return "🦆"
```

```
class GeckoDuck:  
    def quack(self) -> str:  
        return "quack??"  
    def swim(self) -> str:  
        return "Gecko doesn't swim"
```



Protocols

```
duck_things(PharaohDuck())
```

error: Argument 1 to "duck_things" has incompatible type
"PharaohDuck"; expected "Duck"

```
from typing_extensions import Protocol

class Duck(Protocol):
    def quack(self) -> str: ...
    def fly(self) -> str: ...
```

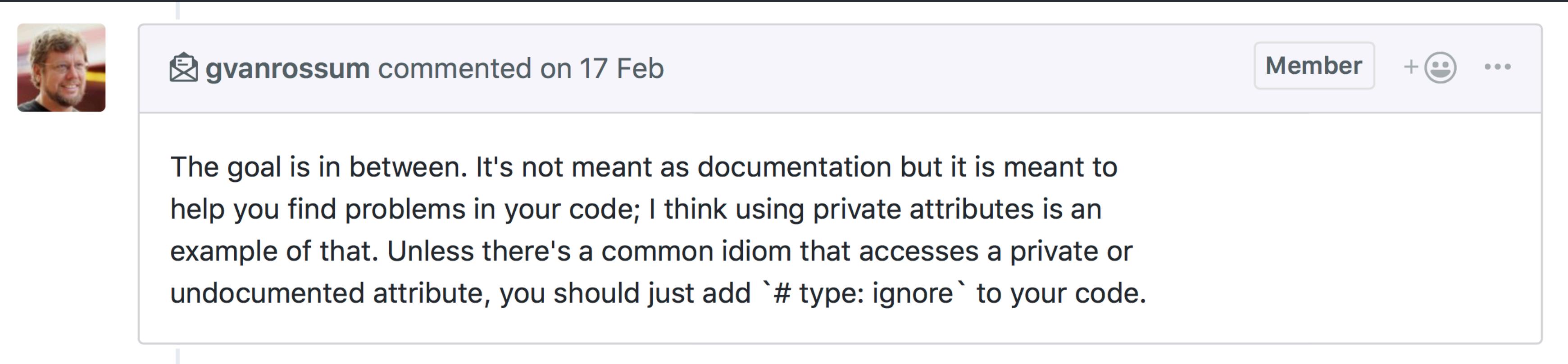


The Typeshed

- Typing for the standard library is not part of it ?!
- Separate repo: <https://github.com/python/typeshed>
- Bundled with your type checker
- Still far from perfect --> send a PR!

type: ignore

- Silences typing error on one statement
- Not necessarily evil
- <https://github.com/python/typeshed/pull/1885>



A screenshot of a GitHub comment card. The profile picture of the user gvanrossum is on the left, followed by the text "gvanrossum commented on 17 Feb". To the right are buttons for "Member", a smiley face icon, and three dots for more options. The main text area contains a quote from gvanrossum: "The goal is in between. It's not meant as documentation but it is meant to help you find problems in your code; I think using private attributes is an example of that. Unless there's a common idiom that accesses a private or undocumented attribute, you should just add `# type: ignore` to your code."

```
async_queue = queue.Queue()

def unblock_queue_callback() -> None:
    with async_queue.all_tasks_done:
        async_queue.all_tasks_done.notify_all()

timeout = MyTimeout(60, callback=unblock_queue_callback)

# wait for all async logging jobs to finish
try:
    with timeout:
        async_queue.join()
except MyTimeoutError as ex:
    log()
```


JSON

```
blob = {  
    "x": null,  
    "numbers": [1, 2, 3],  
    "object": {"y": 2}  
}
```

```
JSONPrimitive = Union[int, float, str, None]  
JSONList = List[JSONPrimitive]  
JSONObject = Dict[str, Union[JSONPrimitive, JSONList, "JSONObject"]]  
  
JSON = Dict[str, Any]
```



NamedTuple

```
from typing import *

class Blob(NamedTuple):
    x: Optional[float]
    numbers: List[int]
    object: Dict[str, int]

    @classmethod
    def from_json(cls, blob: Dict[str, Any]) -> "Blob":
        return cls(blob["x"], blob["numbers"], blob["object"])

blob = {
    "x": null,
    "numbers": [1, 2, 3],
    "object": {"y": 2}
}
```

Summary

Summary

- Type annotations are Pythonic
- Start using them for the readability win
- Get the extra safety of the type checker for free
- The easy bit, new projects should be 100% typed
- The hard part, existing large codebases
 - Start small with gradual typing
 - Use MonkeyType
 - Avoid JSON blobs and use NamedTuple instead

<https://github.com/Sterbic/PyConZA-2019>

Luka Šterbić

Software Engineer @ Facebook London

luka@fb.com

github.com/sterbic

facebook