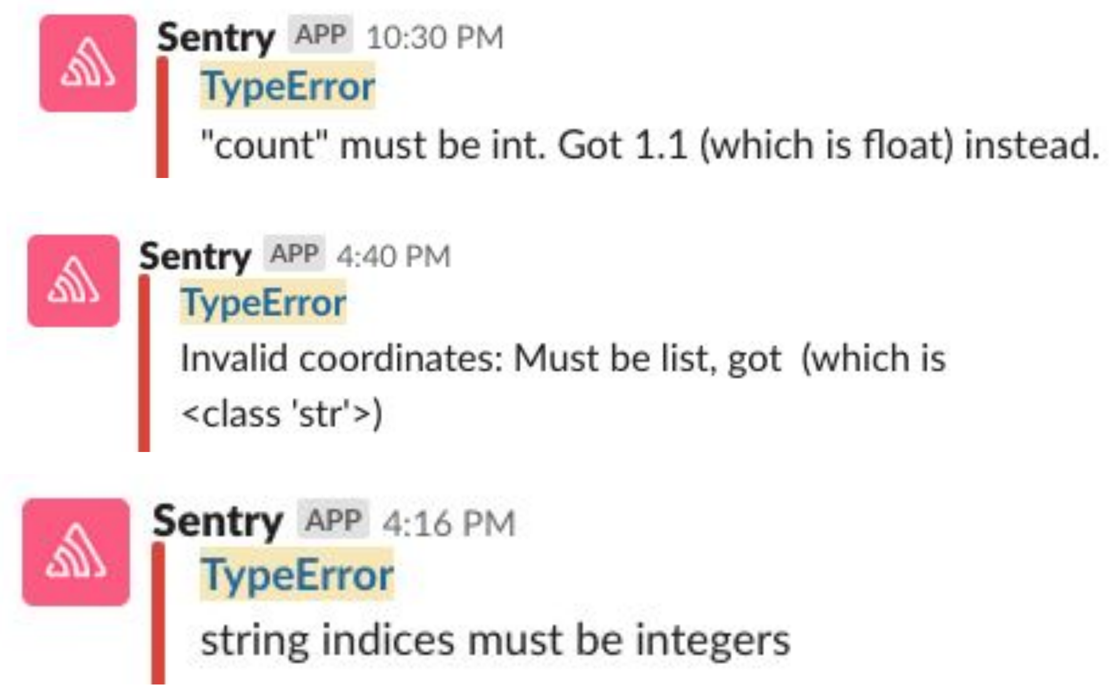
Sleep better with type-safe Python

By Jerry Pussinen

This is Jerry

- Staff Engineer and Competence Lead of Python at Wolt
- jerry-git
- @JerryPussinen
- Blog https://blog.jerrycodes.com

Motivation - wrong type

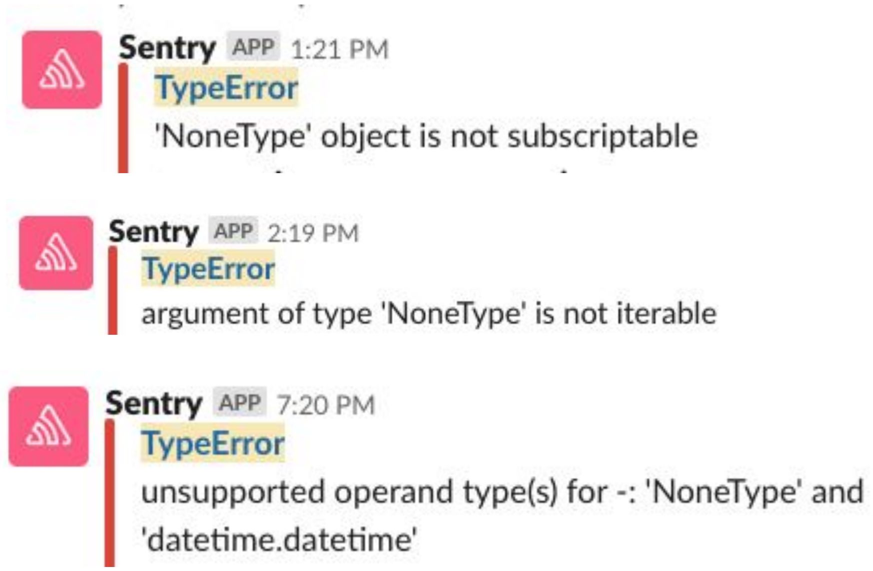

Sentry APP 10:30 PM
TypeError
"count" must be int. Got 1.1 (which is float) instead.

Sentry APP 4:40 PM
TypeError
Invalid coordinates: Must be list, got (which is <class 'str'>)

Sentry APP 4:16 PM
TypeError
string indices must be integers

Wolt

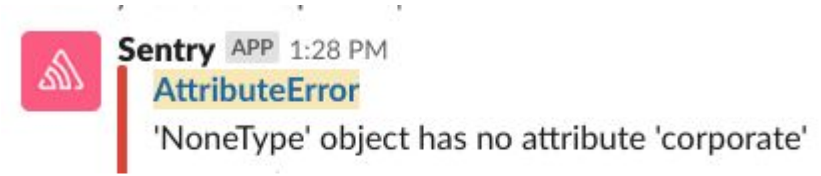
Motivation - None



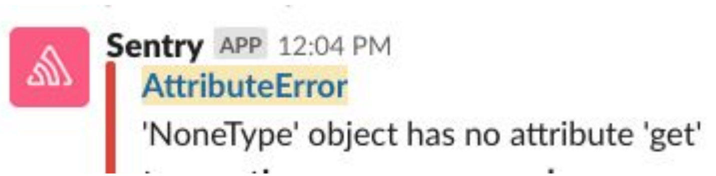
Sentry APP 1:21 PM
TypeError
'NoneType' object is not subscriptable

Sentry APP 2:19 PM
TypeError
argument of type 'NoneType' is not iterable

Sentry APP 7:20 PM
TypeError
unsupported operand type(s) for -: 'NoneType' and 'datetime.datetime'

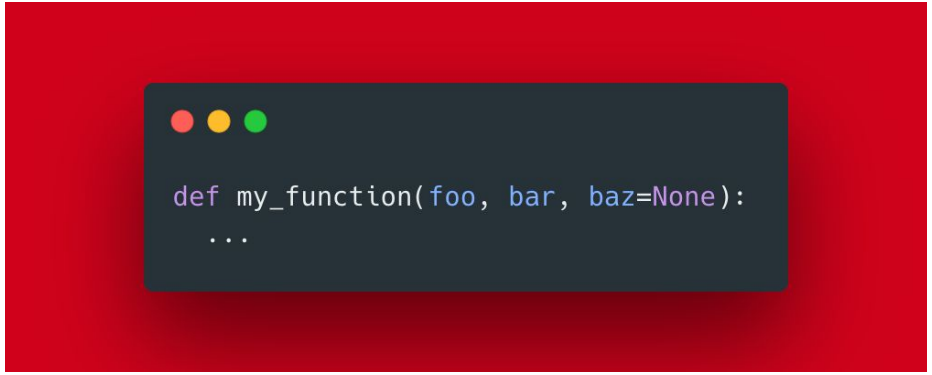


Sentry APP 1:28 PM
AttributeError
'NoneType' object has no attribute 'corporate'

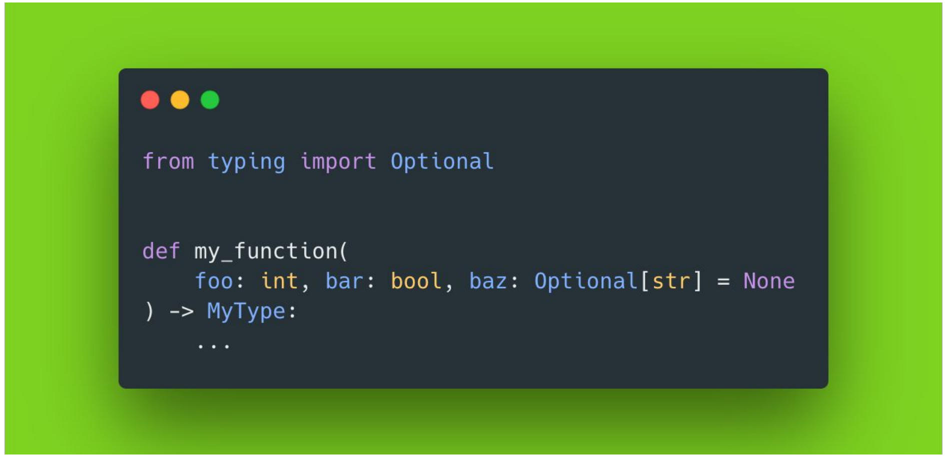


Sentry APP 12:04 PM
AttributeError
'NoneType' object has no attribute 'get'

Wolt



```python
def my_function(foo, bar, baz=None):
    ...
```

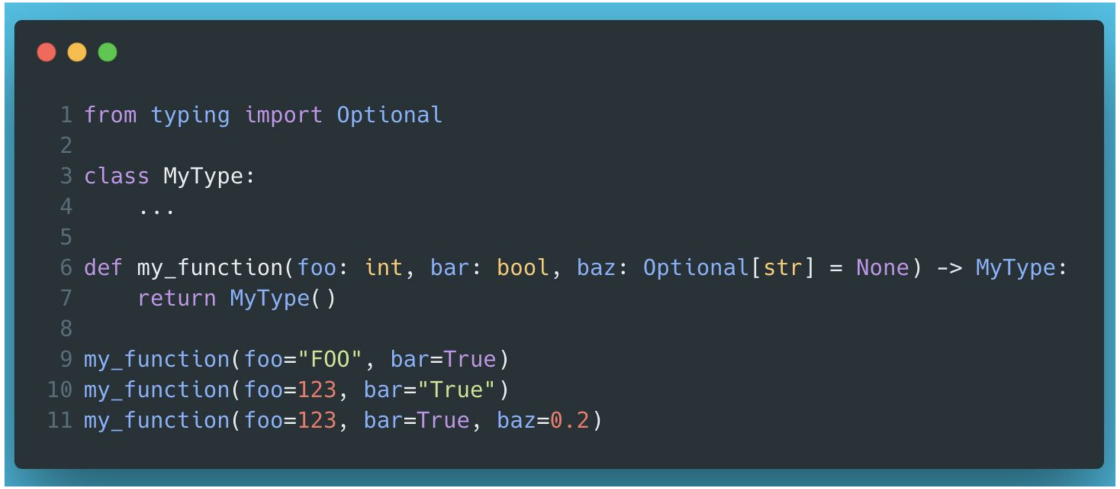


```python
from typing import Optional


def my_function(
    foo: int, bar: bool, baz: Optional[str] = None
) -> MyType:
    ...
```

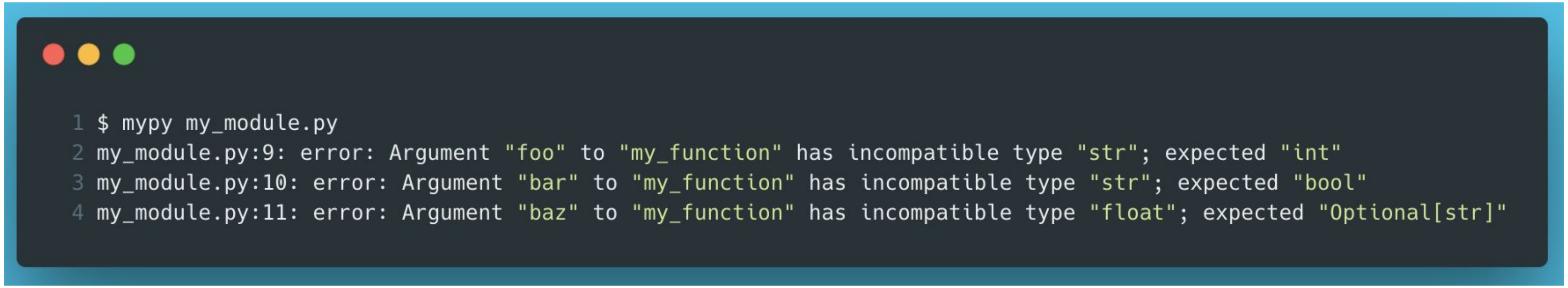
Static type checkers

- Mypy
- pyright (Microsoft, requires Node)
- pytype (Google, comparison with mypy)
- Pyre (Facebook, optimised for perf)

Type hints + static type checker = 💪



```python
from typing import Optional

class MyType:
    ...

def my_function(foo: int, bar: bool, baz: Optional[str] = None) -> MyType:
    return MyType()

my_function(foo="FOO", bar=True)
my_function(foo=123, bar="True")
my_function(foo=123, bar=True, baz=0.2)
```



```
$ mypy my_module.py
my_module.py:9: error: Argument "foo" to "my_function" has incompatible type "str"; expected "int"
my_module.py:10: error: Argument "bar" to "my_function" has incompatible type "str"; expected "bool"
my_module.py:11: error: Argument "baz" to "my_function" has incompatible type "float"; expected "Optional[str]"
```

How about my dependencies?

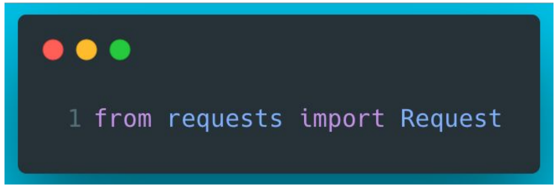
Most popular projects rollout type hints sooner or later

Some examples:

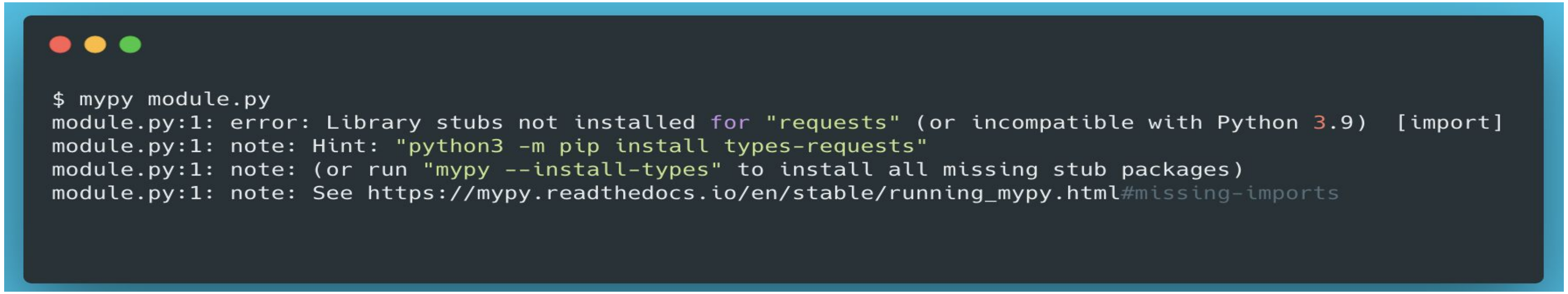
- Flask added in 2.0.0
- Pytest added in 6.0.0

Separate stubs

- For example: https://github.com/python/typeshed
- If you are brave enough: https://pypi.org/search/?q=types
- Mypy is also often helpful:



```python
from requests import Request
```

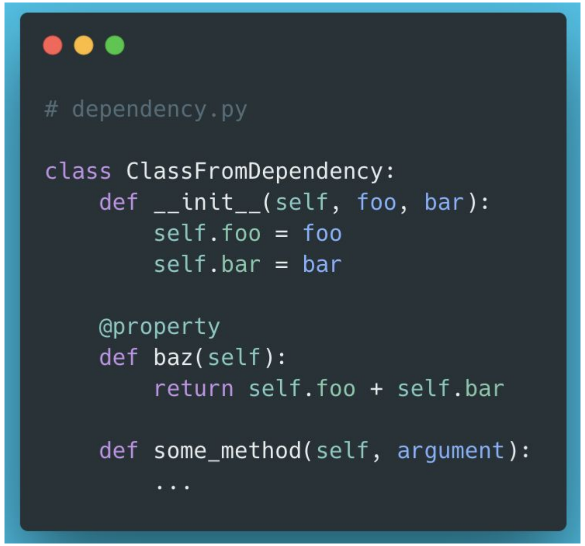


```
$ mypy module.py
module.py:1: error: Library stubs not installed for "requests" (or incompatible with Python 3.9)  [import]
module.py:1: note: Hint: "python3 -m pip install types-requests"
module.py:1: note: (or run "mypy --install-types" to install all missing stub packages)
module.py:1: note: See https://mypy.readthedocs.io/en/stable/running_mypy.html#missing-imports
```

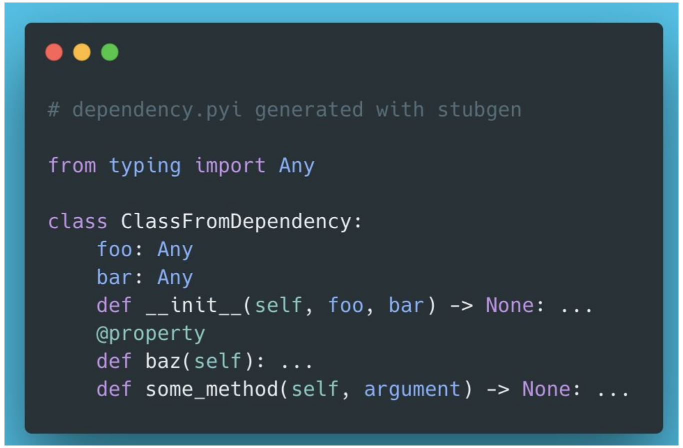
Create stubs yourself

- Stubgen (comes with mypy) can be helpful
- Usually you are using only a tiny part of each dependency -> not much manual work needed
- Consider open sourcing your stubs to help others 🙂

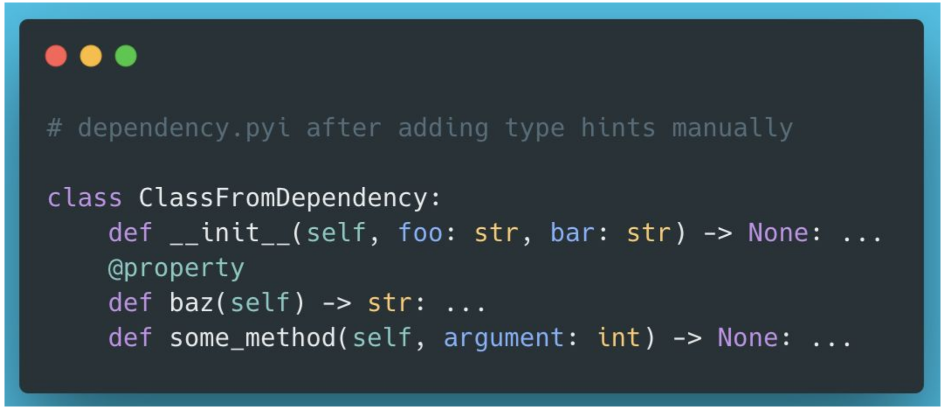
Wolt



```python
# dependency.py

class ClassFromDependency:
    def __init__(self, foo, bar):
        self.foo = foo
        self.bar = bar

    @property
    def baz(self):
        return self.foo + self.bar

    def some_method(self, argument):
        ...
```



```python
# dependency.pyi generated with stubgen

from typing import Any

class ClassFromDependency:
    foo: Any
    bar: Any
    def __init__(self, foo, bar) -> None: ...
    @property
    def baz(self): ...
    def some_method(self, argument) -> None: ...
```



```python
# dependency.pyi after adding type hints manually

class ClassFromDependency:
    def __init__(self, foo: str, bar: str) -> None: ...
    @property
    def baz(self) -> str: ...
    def some_method(self, argument: int) -> None: ...
```

I have an existing project, is it too late?

Mypy supports gradual typing

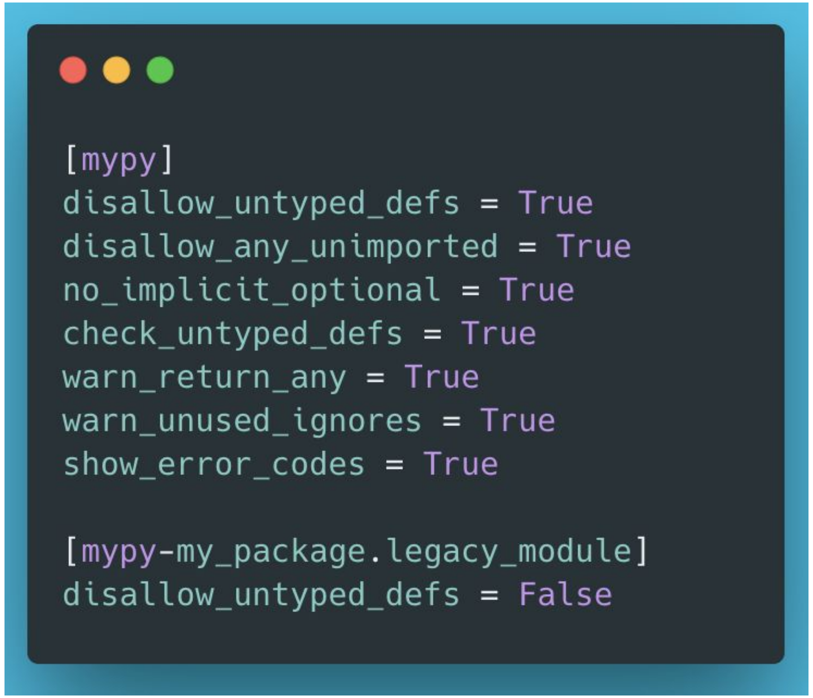
Tips for adding type hints gradually

- We did it for a 100k+ LOC project at Wolt
- Main learning: *Strict configuration by default, loose when needed.*
- Read more from my recent blog post:

Professional-grade mypy configuration

Strict by default

Loose when needed



```
[mypy]
disallow_untyped_defs = True
disallow_any_unimported = True
no_implicit_optional = True
check_untyped_defs = True
warn_return_any = True
warn_unused_ignores = True
show_error_codes = True

[mypy-my_package.legacy_module]
disallow_untyped_defs = False
```

My project is mature and has 100% test coverage. Should I still bother?

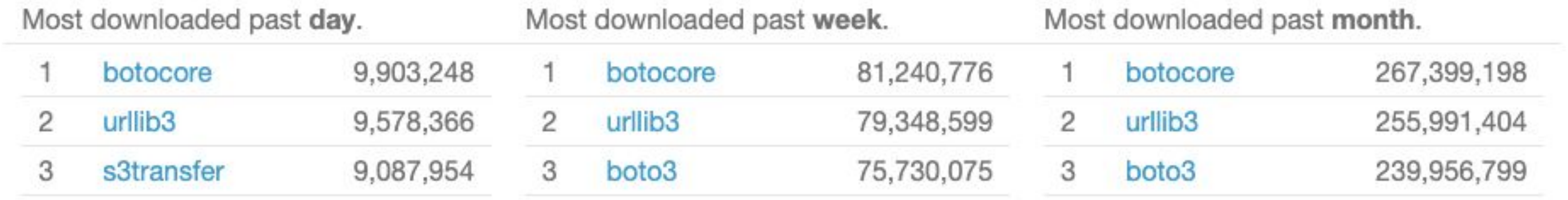
If others depend on your project,
then definitely yes!

Motivation

- See a recent blog post from urllib3 folks:

  Tests aren't enough: Case study after adding type hints to urllib3

- Urllib3 is definitely mature, and they already had 100% test coverage before adding type hints.



Most downloaded past **day**.

1	botocore	9,903,248
2	urllib3	9,578,366
3	s3transfer	9,087,954

Most downloaded past **week**.

1	botocore	81,240,776
2	urllib3	79,348,599
3	boto3	75,730,075

Most downloaded past **month**.

1	botocore	267,399,198
2	urllib3	255,991,404
3	boto3	239,956,799

from https://pypistats.org/top

Get a head start with MonkeyType

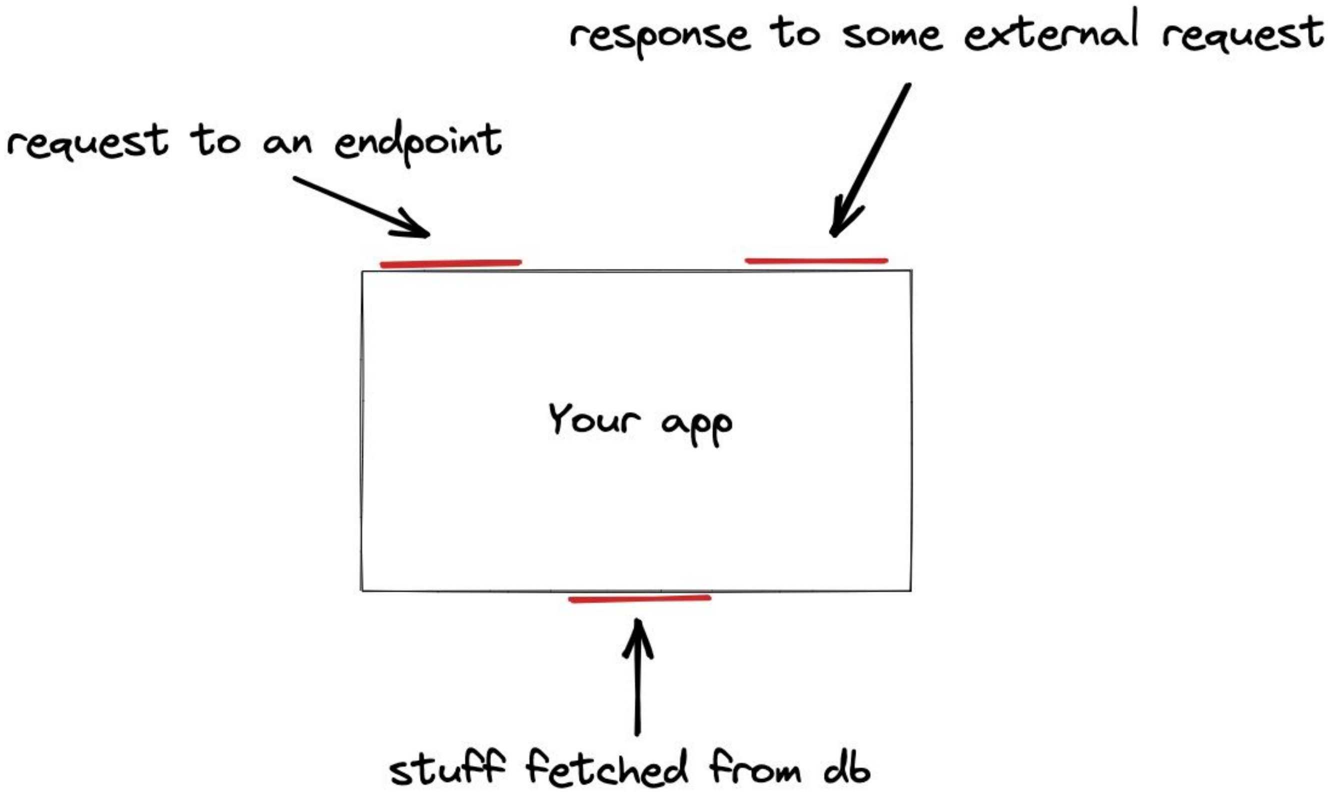
- MonkeyType (from Instagram):

  "*MonkeyType collects runtime types of function arguments and return values, and can automatically generate stub files or even add draft type annotations directly to your Python code based on the types collected at runtime.*"

- So, if you have 100% test coverage, you can auto-generate the type hints just by running your test suite

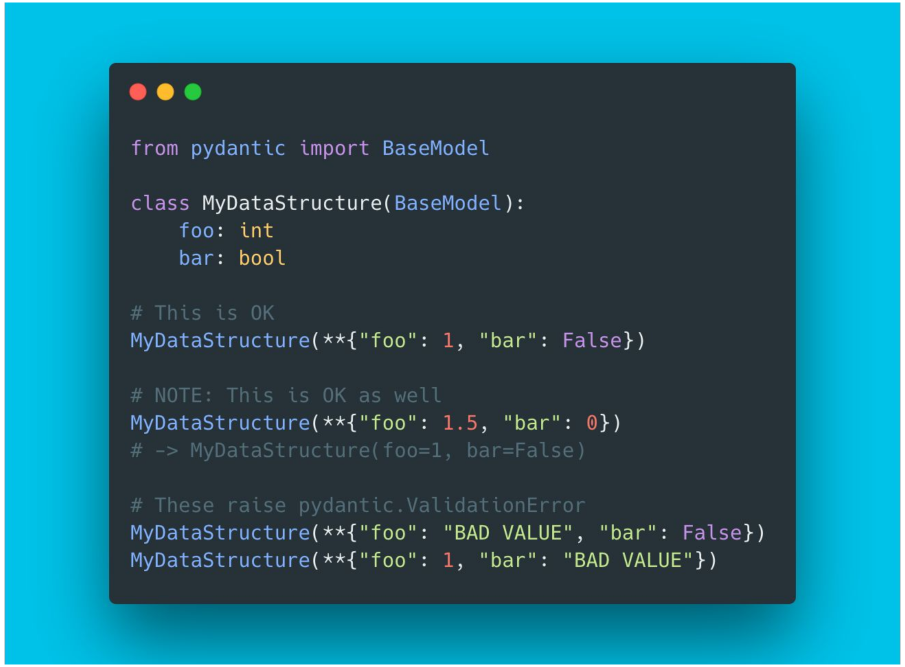
Quite often you'll also need some runtime type checks

Check types during runtime “on the edges”

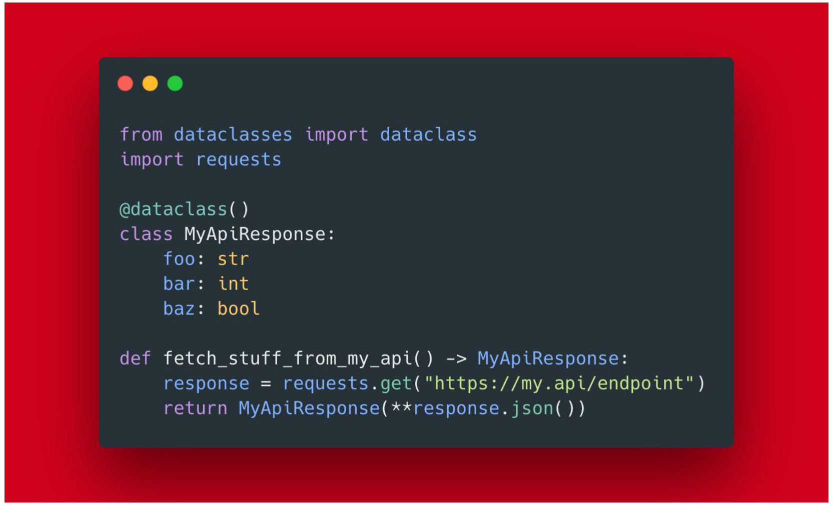


Luckily there are tools which can help

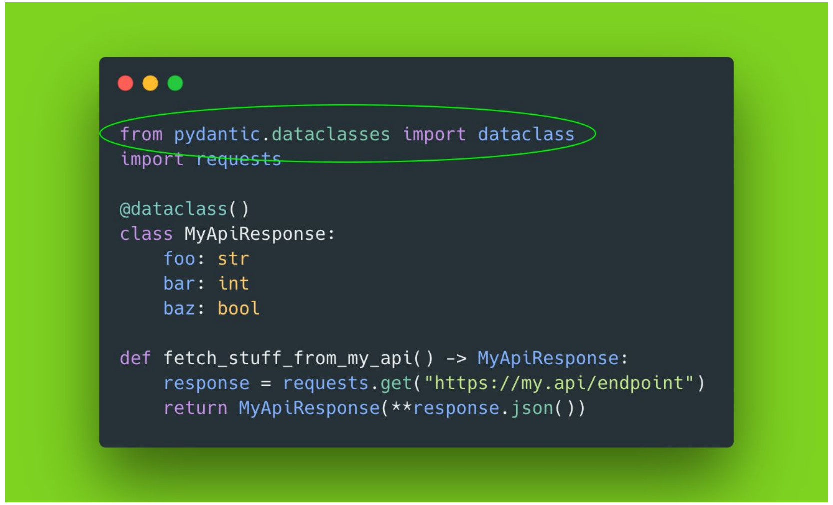
pydantic



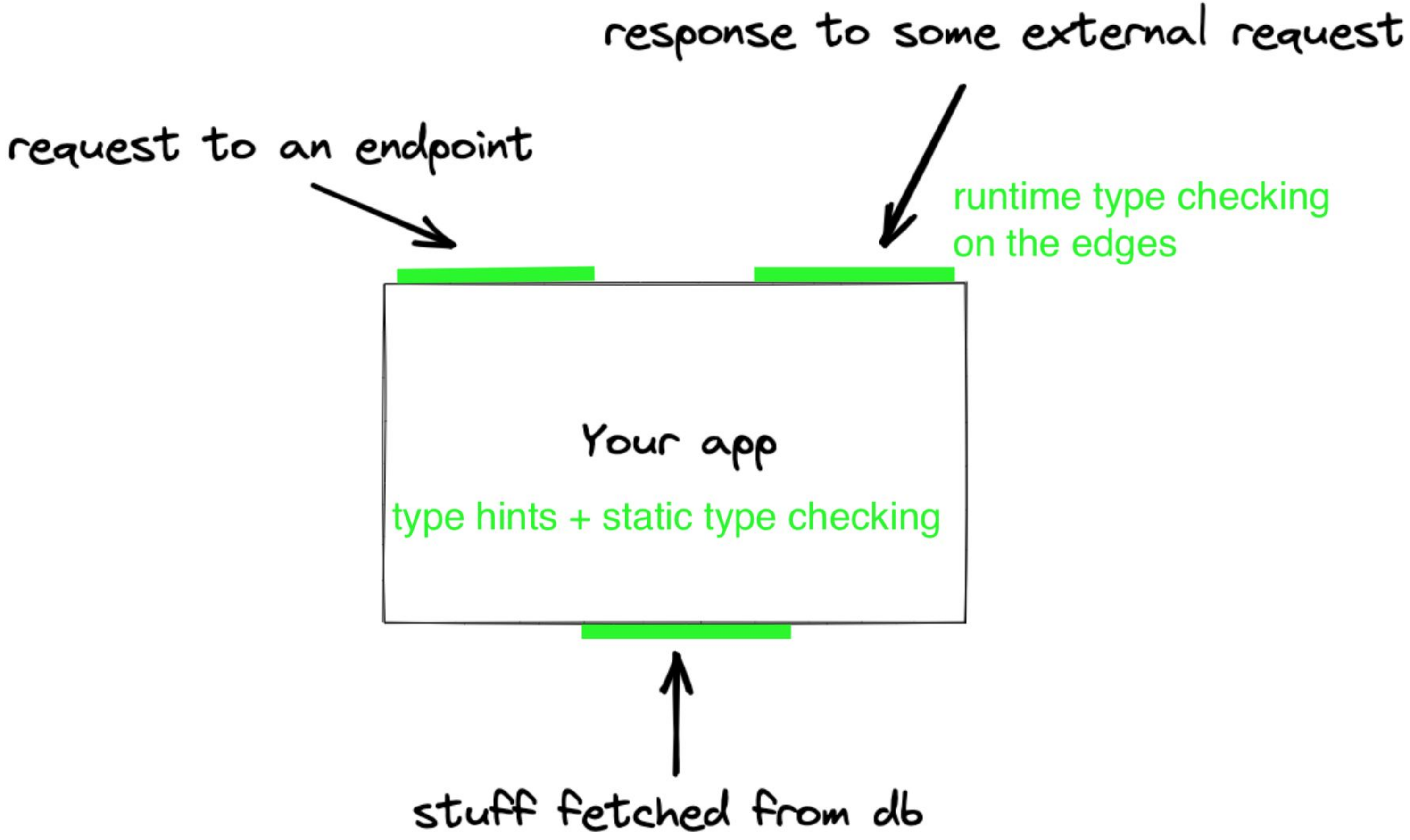
```python
from pydantic import BaseModel

class MyDataStructure(BaseModel):
    foo: int
    bar: bool

# This is OK
MyDataStructure(**{"foo": 1, "bar": False})

# NOTE: This is OK as well
MyDataStructure(**{"foo": 1.5, "bar": 0})
# -> MyDataStructure(foo=1, bar=False)

# These raise pydantic.ValidationError
MyDataStructure(**{"foo": "BAD VALUE", "bar": False})
MyDataStructure(**{"foo": 1, "bar": "BAD VALUE"})
```



```
from dataclasses import dataclass
import requests

@dataclass()
class MyApiResponse:
    foo: str
    bar: int
    baz: bool

def fetch_stuff_from_my_api() -> MyApiResponse:
    response = requests.get("https://my.api/endpoint")
    return MyApiResponse(**response.json())
```



```
from pydantic.dataclasses import dataclass
import requests

@dataclass()
class MyApiResponse:
    foo: str
    bar: int
    baz: bool

def fetch_stuff_from_my_api() -> MyApiResponse:
    response = requests.get("https://my.api/endpoint")
    return MyApiResponse(**response.json())
```


response to some external request
request to an endpoint
runtime type checking
on the edges
Your app
type hints + static type checking
stuff fetched from db

Not convinced yet?

IDEs get super powers

Easier to onboard newcomers

Less testing needs

Less documentation needs

Wolt

Thank you!

Wolt