Type hints in production

a.k.a. static type checking

Angela Santin

angela.d.santin@gmail.com

SMARKETS

Overview

- Why?
- What’s needed?
- How are we using it in production?

Why?

Our Problems (1)

Understanding the code

```
def batch_process(members):
    ...
    for member in members:
        member.process()
    ...
```

Our Problems (1)

Understanding the code

```
def batch_process(members):
    ...
    for member in members:
        member.process()
    ...
```

```
>>> grep ‘def process’ | wc -l
56
```

Our Problems (2)

Refactoring code

```
class Employee():
    ...
    def process(self):
        # Modify interface of method
```

Our Problems (2)

Refactoring code

```
class Employee():
    ...
    def process(self):
        # Modify interface of method
```

A. >>> grep ‘process()’

B. >>> grep ‘Employee()’

Types

A type is a set of values that share some structural property

Types

A type is a set of values that share some structural property

```
>>> ‘a’ + 1
```

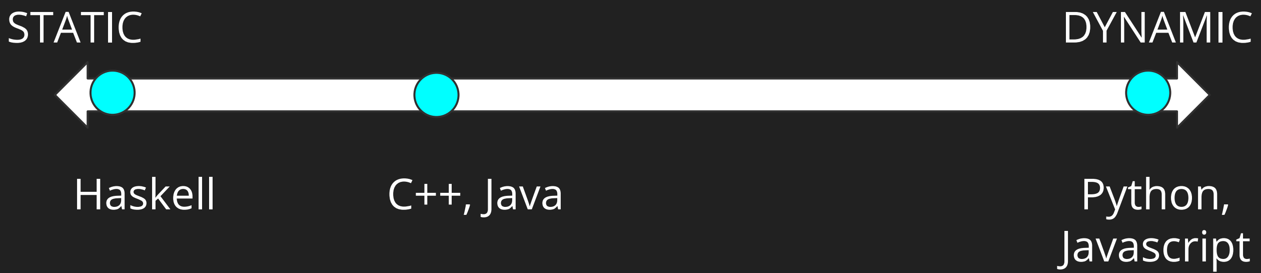
Types

A type is a set of values that share some structural property

```
>>> ‘a’ + 1
Traceback (most recent call last):
  File “<stdin>”, line 1, in <module>
TypeError: cannot concatenate ‘str’ and ‘int’ objects
```

Static type checking

Static type checking



What is required for static type checking?

1. Set of rules that assign types to expressions

2. Annotations to specify types explicitly

3. A type checking tool

I will never use static type checking because...

I will never use static type checking because...

I can:

- write code faster
- write concise code
- benefit from flexibility

I can’t live without static type checking because...

Without static type checking I can:

- write code faster
- write less wordy code
- benefit from flexibility

It helps me by:

- Catching errors before runtime
- Providing support from IDEs
- Allowing compiler to optimize code
- Documenting the code

I can’t live without static type checking because...

Without static type checking I can:

- write code faster
- write less wordy code
- benefit from flexibility

It helps me by:

- Catching errors before runtime
- Providing support from IDEs
- Allowing compiler to optimize code
- Documenting the code

Solving our problems (1)

Understanding code

```
def batch_process(members: List[Customer]) -> None:
    ...
    for member in members:
        member.process()
    ...
```

Solving our problems (1)

Understanding code

```
def batch_process(members: List[Customer]) -> None:
    ...
    for member in members:
        member.process()
    ...
>>> grep ‘class Customer’ | wc -l
1
```

Solving our problems (2)

Refactoring

```
class Employee():
    ...
    def process(self, number: int) -> None:
        # Modify interface of method
```

Solving our problems (2)

Refactoring

```
class Employee():
    ...
    def process(self, number: int) -> None:
        # Modify interface of method
```

```
>>> mypy program_files
```

Solving our problems (2)

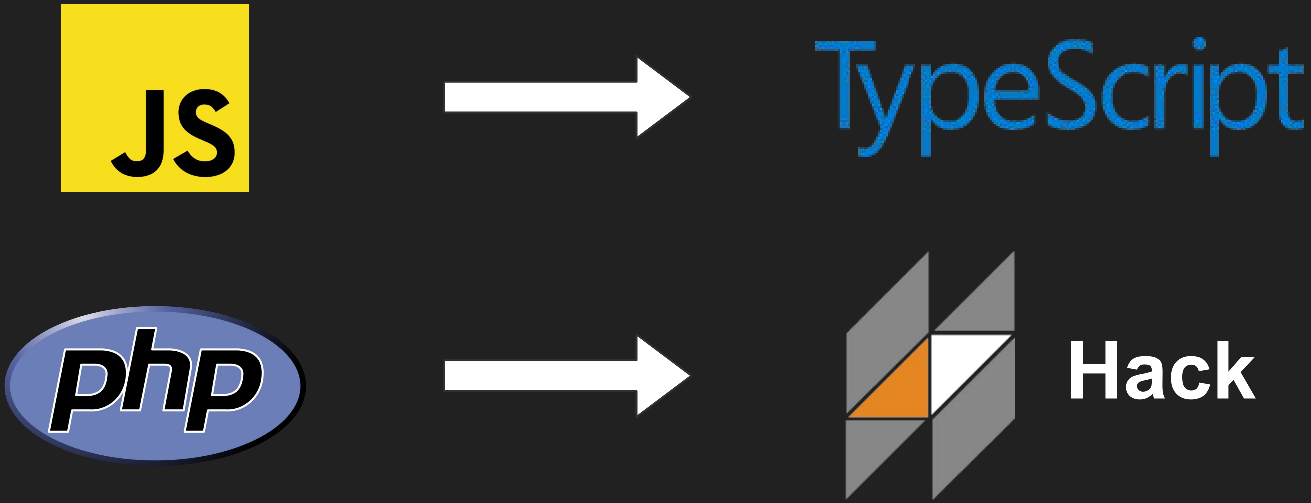
Refactoring

```
class Employee():
    ...
    def process(self, number: int) -> None:
        # Modify interface of method
```

```
>>> mypy program_files
services/user/redirect.py   TypeError
services/reauth/session.py       TypeError
....
```

TODO list

When scripts mature into programs...



What about Python?

1. Rules: PEP 483/484
2. Annotations: PEP 3107
3. Type checker: PEP 484

Syntax for type annotations (PEP 3107)

Python 3:

```
def process(numbers: List[int], name: str) -> bool:

        ....
```

Syntax for type annotations (PEP 3107)

Python 3:

```
def process(numbers: List[int], name: str) -> bool:
    ....


numbers = []        # type: List [int]
```

Syntax for type annotations (PEP 3107)

Python 2:

```
def process(numbers, name):
    # type: (List[int], str) -> bool
    ....
```

The type system: supported types

User-defined classes and built-ins

```
class EmailSender():
    def __init__(self)->None:
        ....
```

User-defined classes and built-ins

```
class EmailSender():
    def __init__(self)->None:
        ....
```

```
import EmailSender

def send(sender: EmailSender) -> bool:
    ....
```

Typing module: Iterable

```
from typing import Iterable
```

Typing module: Iterable

```
from typing import Iterable


def greet_all(names: Iterable[str]) -> None:
    for name in names:
        ....
```

Typing module: Iterable

```
from typing import Iterable


def greet_all(names: Iterable[str]) -> None:
    for name in names:
        ....
```

Other abstract base classes: Sequence[bool], Dict[str, int]...

Any

```
a = None        # type: Any
a.split()
```

Union, None, Optional, TypeVar, Generics ...

http://mypy.readthedocs.io/en/latest/kinds_of_types.html

Static type checking tool

- PyCharm
- Pylint
- Pytype
- Mypy

Installation

```
>>> pip install mypy-lang
>>> mypy program.py
```

- Type checking works for both Python 2.7 and 3.2+ code
- However, mypy must be run from Python 3
- Running mypy 0.4.4

When to do static checking?

Part of continuous integration (CI):

- Unit and integration tests
- Flake8, Pylint
- mypy → “linter on steroids”

Critical to ensure trust in type annotations

Will it slow down CI?

Mypy is fast

-- incremental: reuse cached results

-- fast-parser: faster parser

Gradual typing

Total Python LOC	> 300,000
Annotated LOC	> 20,000
% annotated LOC	5 %
Functions annotated	991 functions

Gradual typing

Total Python LOC	> 300,000
Annotated LOC	> 20,000
% annotated LOC	5 %
Functions annotated	991 functions

```
def subtract(a: Any, b: Any) -> Any:
    return a - b
```

=

```
def subtract(a, b):
    return a - b
```

What about third-party libraries?

- First, start by ignoring imported files mypy --silent-imports
- Can incorporate stub files (.pyi files) incrementally
- As you incorporate stub files mypy --almost-silent

What about duck typing?

```
def quack_processing(input):
    input.quack()
    ...
```

What about duck typing?

```
def quack_processing(input):
    input.quack()
    ...
```

Solutions?

- Do not annotate at all
- Any type
- Union of types

Use an Abstract Base Classes

```
def quack_processing(input: Duck) -> None:
    input.quack()
    ...


class Duck(metaclass=ABCMeta):
    @abstractmethod
    def quack(self) -> None:
        pass
```

Can we avoid subclassing?

```
def quack_processing(input: SupportsQuacking) -> None:
    input.quack()
    ...


class SupportsQuacking(Protocol):
    @abstractmethod
    def quack(self) -> None:
        pass
```

https://github.com/python/typing/issues/11

Issues? Returning None

```
def greeting(speak: bool) -> Optional[str]:
    if speak:  return ‘Hello friend!’
```

Issues? Returning None

```
def greeting(speak: bool) -> Optional[str]:
    if speak:  return ‘Hello friend!’
greeting(False).split()          # Runtime Type Error
```

Issues? Returning None

```
def greeting(speak: bool) -> Optional[str]:
    if speak:  return ‘Hello friend!’
greeting(False).split()          # Runtime Type Error
```

```
>>>  mypy hello.py
```

Issues? Returning None

```
def greeting(speak: bool) -> Optional[str]:
    if speak:  return ‘Hello friend!’
greeting(False).split()              # Runtime Type Error
```

```
>>>  mypy hello.py

>>> mypy --strict-optional hello.py

hello.py: 3: error: Some element of union has no attribute "split"
```

Ignoring mypy?

# type: ignore	105 times
Any	327 times

2% of the annotated lines (21,362)

Testimonials

“It’s a mystery how we were able to be productive without static type checking”

Testimonials

“It’s a mystery how we were able to be productive without static type checking”

...

Testimonials

“It’s a mystery how we were able to be productive without static type checking”

...

“I hate static type checking. Leave me alone.”

Conclusion

- Improved understanding/refactoring
- Easy to incorporate into CI
- Help catching some errors before runtime
- Not perfect, but improving quickly

Thank you!

To learn more

Mypy:

http://mypy-lang.org/

https://mypy.readthedocs.io/en/latest/

https://github.com/python/mypy/

PEPs: 3107, 482, 484, 483

Typeshed: https://github.com/python/typeshed

Pycon 2016 Pytype talk - https://www.youtube.com/watch?v=IDm_YIQihhs

Class vs. type vs. metaclass

Instance::Class

Class::metaclass

→ the default metaclass is type

→ The type metaclass is not the same as the types in the typing module

Covariance, contravariance, invariance…?

```
class Base():
    ....

class Derived(Base):
    ....
```

Covariance, contravariance, invariance...

```
def fun_invariant(arg: Derived):          # only accepts type specified
    ....
```

Covariance, contravariance, invariance...

```
def fun_invariant(arg: Derived):        # only accepts type specified

    ....

def fun_contravariant(arg: Derived)     # also accepts more generic types (i.e. Base)

    ....
```

Covariance, contravariance, invariance

```
def fun_invariant(arg: Derived):          # only accepts type specified
    ....
def fun_contravariant(arg: Derived)       # also accepts more generic types (i.e. Base)
    ....
def fun_covariant(arg: Base)              # also accepts derived types (i.e. Derived)
    ....
```

Covariance, contravariance, invariance

```
def fun_invariant(arg: Derived):           # only accepts type specified
    ....
def fun_contravariant(arg: Derived)        # also accepts more generic types (i.e. Base)
    ....
def fun_covariant(arg: Base)               # also accepts derived types (i.e. Derived)
    ....
```

https://github.com/python/mypy/issues/2034

Automatic generation of annotations

Mypy: Can be done, but to .pyi -- we have not tried it

Pytype: does have functionality to merge it back in

How good is mypy’s type inferencing?

Type inferencing: automatic deduction of the type of an expression

```
i = 1                  # mypy infers type as int
some_list = [1, 2]     # mypy infers type as List[int]
```

No technical document exists describing algorithms used

See: https://github.com/python/mypy/issues/1994