Сыны голубой планеты

Нам, советским людям, строившим воинством, выпала честь первым представить в космос. Победы в освоении космоса нам суждено не только достоянием нашего народа, но и всего человечества. Мы с радостью своей им на службу всем народам, освободить прогресс, счастье и блага всех людей на Земле. Нашим достижениям и открытиям мы ставим не на службу войне, а на службу миру и безопасности народов.
Облетев Землю в корабле-спутнике, я увидел, как прекрасна наша планета. Люди, будем хранить и приумножать эту красоту, а не разрушать её!
WAR IS STUPID

SPACE

IS WONDERFUL
GRADUAL TYPING OF PRODUCTION APPLICATIONS

ŁUKASZ LANGA
ŁUKASZ LANGA

ambv @ #python
fb.me/ambv
@llanga
lukasz@langa.pl
WHAT WE’RE GOING TO TALK ABOUT

Intro to PEP 484
- Why annotate?
- What’s the syntax?
- Types are more than classes
- The typing module
- Generics
- Type inference
- Runtime errors are hard to debug

Gradual typing
- What’s the problem?
- What tools exist?
- Where to start?
- Continuous integration
- Fighting regressions
- Metrics of success
- Typeshed

Common gotchas
- Which functions are typed
- Unions
- Out of order definitions
- Invariance
- Self-types
- Bugs in tooling
WHY ANNOTATE PYTHON CODE WITH TYPES?
WHY ANNOTATE PYTHON CODE WITH TYPES?

def process_all(items):
    for item in items:
        item.children.process()
WHY ANNOTATE PYTHON CODE WITH TYPES?

• Help readers understand the code at hand
WHY ANNOTATE PYTHON CODE WITH TYPES?

• Help readers understand the code at hand
• Validate assumptions programmers make while writing
WHY ANNOTATE PYTHON CODE WITH TYPES?

• Help readers understand the code at hand
• Validate assumptions programmers make while writing
• Minimize amount of state developers need to keep in their heads
WHY ANNOTATE PYTHON CODE WITH TYPES?

• Help readers **understand** the code at hand
• **Validate assumptions** programmers make while writing
• **Minimize amount of state** developers need to keep in their heads
• Find existing bugs, prevent future ones
def f(argument):
    ...

def f(argument: T) -> R:
    ...

```python
def f(argument: T) -> R:
...

def f(argument: str)
def f(argument: int)
def f(argument: list)
def f(argument: MediaType)
...
```
def f(argument: T) -> R:
...

def f() -> str: ...
def f() -> int: ...
def f() -> None: ...
```python
example.py

def gcd(a: int, b: int) -> int:
    while b:
        a, b = b, a % b
    return a

print(gcd(10, 15))
```
```python
def gcd(a: int, b: int) -> int:
    while b:
        a, b = b, a % b
    return a

print(gcd('x', 15))
```
TYPE CHECK ERROR VS. RUNTIME ERROR

Type error:
.local/example.py:7:6: error: Argument 1 to "gcd" has incompatible type "str"; expected "int"

Runtime error:
Traceback (most recent call last):
  File ".local/example.py", line 7, in <module>
    print(gcd('x', 15))
  File ".local/example.py", line 3, in gcd
    a, b = b, a % b
TypeError: not all arguments converted during string formatting
```python
class Point:
    def __init__(self, x: int, y: int) -> None:
        self.x = x
        self.y = y

p = Point(0, 1)
print(p.x, p.y, p.z)
reveal_type(p.x)
```
THERE’S MORE TO TYPES THAN JUST CLASSES
THERE’S MORE TO TYPES THAN JUST CLASSES

- What if a function returns anything?
- What if a function only accepts lists with string elements?
- What if a function accepts a string or None?
- What if a function accepts instances of TypeA or TypeB?
- What if a function returns a value of the same type as passed as an argument?
- What if a function accepts an int but it’s really expecting only fbids?
- What if a function accepts a class, not an object?
WHAT IF A FUNCTION RETURNS \textbf{ANYTHING}?

def f() -> Any:
    ...
WHAT IF A FUNCTION ONLY ACCEPTS LISTS WITH STRING ELEMENTS?

```python
from typing import List

def f(l: List[str]) -> None:
    ...
```
WHAT IF A FUNCTION ACCEPTS A STRING **OR** NONE?

```python
def f(s: Optional[Optional[str]]) -> None:
    ...
```
WHAT IF A FUNCTION ACCEPTS A STRING OR NONE?

def f(s: Optional[str]) -> None:
...

def g(s: str = None) -> None:
...
WHAT IF A FUNCTION ACCEPTS TYPE A OR TYPE B?

```python
from typing import Union
def f(s: Union[A, B]) -> None:
    ...
```
WHAT IF A FUNCTION RETURNS A VALUE OF THE SAME TYPE AS PASSED AS AN ARGUMENT?

```python
from typing import TypeVar
T = TypeVar('T')
def f(s: T) -> T:
    ...
```
WHAT IF A FUNCTION RETURNS A VALUE OF THE SAME TYPE AS PASSED AS AN ARGUMENT?

from typing import List, TypeVar

T = TypeVar('T')
def f(s: List[T]) -> T:
    ...

WHAT IF A FUNCTION ACCEPTS AN INT BUT IT’S REALLY EXPECTING **ONLY FBIDS**?

```python
going from typing import NewType
FBID = NewType('FBID', int)
def unixname(id: FBID) -> str:
    ...
```
WHAT IF A FUNCTION ACCEPTS A CLASS, NOT AN OBJECT?

```python
from typing import Type

def factory(t: Type[django.Model]) -> django.Model:
    ...
```
WHAT IF A FUNCTION ACCEPTS A CLASS, NOT AN OBJECT?

```python
from typing import TypeVar, Type
M = TypeVar('M', bound=django.Model)

def factory(t: Type[M]) -> M:
    ...
```
TYPE INFERENCE
TYPE INFERENCE

• The first assignment to a variable defines its type

```python
i = 1           # reveal_type(i) -> int
l = ['a', 'b']  # reveal_type(l) -> list[str]
```
TYPE INFERENCE

• The first assignment to a variable defines its type

```python
i = 1           # reveal_type(i) -> int
l = ['a', 'b']  # reveal_type(l) -> list[str]
```

• If this assignment is an empty collection or None, inference fails

```python
i = None        # reveal_type(i) -> None
l = []          # error: need type annotation for variable
                # reveal_type(l) -> Any
```
TYPE INFERENCE

- We need generics for collections to infer what elements they contain

```python
def f(l: List[str]) -> None:
    s = l.pop() # reveal_type(s) -> str
```
TYPE INFERENCE

• We need generics for collections to infer what elements they contain

```python
def f(l: List[str]) -> None:
    s = l.pop() # reveal_type(s) -> str
```

• We can improve inference by constraining the flow paths of the function

```python
def f(message: str = None) -> None:
    if message is None:
        return
    reveal_type(message) # -> str
```

• Same goes for using `assert isinstance(...)` and similar
TYPE INFERENCEx

• We can use **variable annotations** for the non-obvious cases

```python
def f() -> None:
    l: List[str] = []
    m: str  # no need for initial value
```

• This requires **Python 3.6**

• In Python 3.5 and older, you had to use type comments for the same effect

```python
def f() -> None:
    l = []  # type: List[str]
    m = ...  # type: str  # `...` means no initial value
```
• When everything else fails, you can cast:

```python
from typing import cast

some_int = function_returning_ints()
fbid = cast(FBID, some_int)
```
GRADUAL TYPING
GRADUAL TYPING
GRADUAL TYPING
```python
# from typing import List

def process_all():
    return process_strings("string1", "string2")

def process_strings(to_repeat, num_repeats):
    
    """Sorts and repeats the strings."""
    return process_input(to_repeat) * num_repeats

def process_input(input):
    return input.sort()
```

from typing import List

def process_all():
    return process_strings("string1", "string2")

def process_strings(to_repeat, num_repeats):
    """Sorts and repeats the strings."""
    return process_input(to_repeat) * num_repeats

def process_input(input: List[str]) -> List[str]:
    return input.sort()
from typing import List

def process_all() -> None:
    return process_strings("string1", "string2")

def process_strings(to_repeat, num_repeats):
    """Sorts and repeats the strings."""
    return process_input(to_repeat) × num_repeats

def process_input(input: List[str]) -> List[str]:
    return input.sort()
from typing import List

def process_all() -> None:
    return process_strings("string1", "string2")

def process_strings(to_repeat: List[str], num_repeats: int) -> List[str]:
    """Sorts and repeats the strings."""
    return process_input(to_repeat) * num_repeats

def process_input(input: List[str]) -> List[str]:
    return input.sort()
GRADUAL TYPING
GRADUAL TYPING
WORKFLOW SUGGESTION #1

Find the most critical functions and start typing those first.
Enable file-level linter type checking early.
• You will use types and functions from other modules, too, by importing them.

```
# a.py
from b import B
from f import function

# b.py
from base import Model
from f import function

def f():
    ...

# f.py
from base import Model
from f import function

# g.py
from base import Model
from utils import x

# base.py
class Model:
    ...

# z.py
from f import function
import g

# utils.py
```
FILE-LEVEL VS. FULL PROGRAM ANALYSIS

- You will use types and functions from other modules, too, by importing them.
- `mypy` performs full program analysis

$ pip install mypy
file-level vs. full program analysis

- You will use types and functions from other modules, too, by importing them.
- **mypy** performs full program analysis *which makes it slow (so far)*
- If mypy cannot find an import or is configured to ignore imports, it assumes Any types everywhere.
- This hides usage errors: imported classes and functions are all treated as Any.
- **flake8-mypy** does this intentionally *to make checks fast*
GRADUAL TYPING
from typing import Optional, Union, Any, Iterable, Mapping, MutableMapping, Tuple, IO, TypeVar

from .models import Response

_ParmsMappingValueType = Union[Text, bytes, int, float, Iterable[Union[Text, bytes, int, float]]]
_ParmsData = Union[None, bytes, MutableMapping[Text, Text], IO]

def request(method: str, url: str, **kwargs) -> Response: ...

def get(url: Union[Text, bytes], params: Optional[
    Union[Mapping[Union[Text, bytes, int, float], _ParmsMappingValueType],
            Union[Text, bytes],
            Tuple[Union[Text, bytes, int, float], _ParmsMappingValueType],
            Mapping[Text, _ParmsMappingValueType],
            Mapping[bytes, _ParmsMappingValueType],
            Mapping[int, _ParmsMappingValueType],
            Mapping[float, _ParmsMappingValueType]]] = None,
    **kwargs) -> Response: ...

def options(url: Union[str, Text], **kwargs) -> Response: ...

def head(url: Union[str, Text], **kwargs) -> Response: ...

def post(url: Union[str, Text], data: Data, **kwargs) -> Response: ...
Enable continuous integration to fight regressions.
Only annotated functions are type checked.
A clean run of the type checker doesn’t prove there’s no errors.
New annotations can discover errors in other functions.
flake8-mypy only presents type errors related to the current file and the standard library.
Do full codebase checks with mypy as part of continuous integration.
WORKFLOW SUGGESTION #4

Measure function coverage and the number of TypeError/AttributeError exceptions in production.
def gcd(a: int, b: int) -> int:
    while b:
        a, b = b, a % b
    return a

def f():
a = gcd(1, 2)
reveal_type(a)  # -> Any!
ONLY ANNOTATED FUNCTIONS ARE TYPED

def convert_to_str(obj):
    return obj

def to_upper(exc: Exception) -> str:
    return convert_to_str(exc).upper()  # no error!

At runtime:
AttributeError: 'ValueError' object has no attribute 'upper'
UNIONS LIMIT ALLOWED BEHAVIOR

def f(arg: Union[str, int]) -> None:
    lowercase_str = arg.lower()
    bitlength_of_int = arg.bit_length()
UNIONS LIMIT ALLOWED BEHAVIOR

def f(arg: Union[str, int]) -> None:
    lowercase_str = arg.lower()
    bitlength_of_int = arg.bit_length()

:2:20: error: Some element of union has no attribute "lower"
:3:23: error: Some element of union has no attribute "bit_length"
UNIONS LIMIT ALLOWED BEHAVIOR

def template() -> Union[bytes, str]:
    return "s"

def g(arg: str) -> str:
    return template().format(arg)
def template() -> Union[bytes, str]:
    return "s"

def g(arg: str) -> str:
    return template().format(arg)

:5:11: error: Some element of union has no attribute "format"
OUT OF ORDER DEFINITIONS

class A:
    def f() -> 'B':
        ...

class B:
    ...

from __future__ import annotations

class A:
    def f() -> B:
        ...

class B:
    ...

OUT OF ORDER DEFINITIONS IN PYTHON 3.7
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from module import B

class A:
    def f() -> 'B':
        ...

IMPORT CYCLES
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from module import B  # noqa

class A:
    def f() -> 'B':
        ...

IMPORT CYCLES
from __future__ import annotations
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from module import B

class A:
    def f() -> B:
        ...

IMPORT CYCLES IN PYTHON 3.7
from __future__ import annotations

if __typing__:
    from module import B

class A:
    def f() -> B:
        ...

def f(l: List[Optional[str]]) -> None:
    ...

def g(l: List[str]) -> None:
    f(l)
MUTABLE COLLECTIONS ARE INVARIANT

def f(l: List[Optional[str]]) -> None:
    ...

def g(l: List[str]) -> None:
    f(l)  # error: arg 1 to f() incompatible type
    # List[str]; expected List[Optional[str]]
CRAZY IDEAS

- Static type inference
- Runtime type inference
- Runtime type *checking*
- Duck typing + type checking
  - See PEP 544: Protocols
  - Performance optimizations
WHERE TO GET SUPPORT?

• Read first:
  • comprehensive mypy docs: http://mypy.readthedocs.io/en/latest/
  • #typing on Gitter for real-time support
• For PEP 484/PEP 526/PEP 544/PEP 563 issues:
  • github.com/python/typing
• For type checker issues:
  • github.com/python/mypy
• For issues with standard library and third-party annotations:
  • github.com/python/typeshed