Static Typing in Python

EuroPython 2020

@di_codes

Hi, I'm Dustin

- Developer Advocate @ Google
- PyTexas (Austin, TX, Oct 24-25th 2020)
- Python Package Index

Pop quiz:

Is Python *dynamically* or *statically* typed?

Answer:

Dynamically typed... but can optionally be statically typed.

Steps to understand that:

- Types in Python
- Type systems in general
- Dynamic typing in Python
- Static typing in Python

Once we understand that:

- How to use static typing
- When you should use static typing
- When you *shouldn't* use static typing

Let's talk about

types (and `type`)

```
>>> type(42)
<class 'int'>
```

```
>>> type(42)
<class 'int'>
>>> type(42.0)
<class 'float'>
```

```
>>> type(42)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('foo')
<class 'str'>
```

```
>>> type(42)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('foo')
<class 'str'>
>>> type(['foo', 'bar'])
<class 'list'>
```

```
>>> a = 42
42
```

```
>>> a = 42
42
>>> float(42)
42.0
```

```
>>> a = 42
42
>>> float(42)
42.0
>>> str(float(42))
'42.0'
```

```
>>> a = 42
42
>>> float(42)
42.0
>>> str(float(42))
'42.0'
>>> list(str(float(42)))
['4', '2', '.', '0']
```

```
>>> type(42) is int
True
>>> int
<class 'int'>
>>> isinstance(42, int)
True
```

```
>>> type(None)
<class 'NoneType'>
>>> def func():
...     pass
...
>>> type(func)
<class 'function'>
>>> type(...)
<class 'ellipsis'>
```

```
>>> import types
```

```
>>> import types
>>> dir(types)
['AsyncGeneratorType', 'BuiltinFunctionType',
'BuiltinMethodType', 'ClassMethodDescriptorType',
'CodeType', 'CoroutineType', 'DynamicClassAttribute',
'FrameType', 'FunctionType', 'GeneratorType',
'GetSetDescriptorType', 'LambdaType',
'MappingProxyType', 'MemberDescriptorType',
'MethodDescriptorType', 'MethodType',
'MethodWrapperType', 'ModuleType', 'SimpleNamespace',
'TracebackType', 'WrapperDescriptorType',
...
```

Dynamic typing

Variables can be *any* type

```
>>> import random
>>> a = random.choice([42, 42.0, '42'])
>>> type(a)
```

```
>>> import random
>>> a = random.choice([42, 42.0, '42'])
>>> type(a)  # Could be str, int, float
```

Dynamic typing

Arguments and return values of functions can be *any* type

```python
def frobnicate(a, b, c):
    "Frobnicates the bizbaz"
    return a + b + c
```

```
>>> def frobnicate(a, b, c):
...     return a + b + c
```

```
>>> def frobnicate(a, b, c):
...     return a + b + c
>>> frobnicate(1, 2, 3)
6
```

```
>>> def frobnicate(a, b, c):
...     return a + b + c
>>> frobnicate(1, 2, 3)
6
>>> frobnicate('hi', ' ', 'there')
'hi there'
```

```
>>> def frobnicate(a, b, c):
...     return a + b + c
>>> frobnicate(1, 2, 3)
6
>>> frobnicate('hi', ' ', 'there')
'hi there'
>>> frobnicate(1, 2, 'foo')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in frobnicate
TypeError: unsupported operand type(s) for +: 'int'
and 'str'
```

```python
def frobnicate(a, b, c):
    """Frobnicates the bizbaz

    Args:
        a (int): The first parameter.
        b (int): The second parameter.
        c (int): The third parameter.

    Returns:
        int: The bizbaz
    """
    return a + b + c
```

```python
def frobnicate(a, b, c):
    "Frobnicates the bizbaz"
    assert type(a) is int
    assert type(b) is int
    assert type(c) is int
    bizbaz = a + b + c
    assert type(bizbaz) is int
    return bizbaz
```

Duck typing

If it walks like a duck and it quacks like a duck...

```
foo = [f(x) for x in bar]

foo = bar > 0

foo = bar(...)
```

Static typing

As in, defined and not changing

```
int frobnicate(int a, int b, int c) {
    return a + b + c;
}
```

```java
public static int frobnicate(int a, int b, int c) {
    return a + b + c;
}
```

```
fn frobnicate(a: u8, b: u8, c: u8) -> u8 {
    return a + b + c;
}
```

```
function frobnicate(a: number, b: number, c:number): number {
    return a + b + c;
}
```

Dynamic

- Python
- Ruby
- Clojure
- JavaScript

Static

- C/C++
- Rust
- Java
- TypeScript

Dynamic

- Python*
- Ruby
- Clojure
- JavaScript

Static

- C/C++
- Rust
- Java
- TypeScript

---

* Kinda.

@di_codes

Python is dynamically typed

But can optionally be statically typed

Our journey to type checking 4

blogs.dropbox.com/tech/2019/09/our-journey-to-type-checking-4-million-lines-of-python/

Dropbox

Topics ▾ Subscribe ▾ Dropbox blogs ▾

Our journey to type checking 4 million lines of Python

Jukka Lehtosalo | September 5, 2019

0 0

Dropbox is a big user of Python. It's our most widely used language both for backend services and the desktop client app (we are also heavy users of Go, TypeScript, and Rust). At our scale—millions of lines of Python—the dynamic typing in Python made code needlessly hard to understand and started to seriously impact productivity. To mitigate this, we have been gradually migrating our code to static type checking using mypy, likely the most popular standalone type checker for Python. (Mypy is an open source project, and the core team is employed by Dropbox.)

Dropbox has been one of the first companies to adopt Python static type checking at this scale. These days thousands of projects use mypy, and things are quite battle tested. It has been a long journey for us to get to this point, and there were a bunch of false starts and failed experiments along the way. This post tells the story of Python static checking at Dropbox, from the humble beginnings as part of my academic research project, to the

@di_codes

PEP 3107

Function Annotations

```python
def frobnicate(a, b, c):
    "Frobnicates the bizbaz"
    return a + b + c
```

```python
def frobnicate(a: 'x', b: 5 + 6, c: []) -> max(2, 9):
    "Frobnicates the bizbaz"
    return a + b + c
```

```
>>> def frob(a: 'x', b: 5 + 6, c: []) -> max(2, 9):
...     return a + b + c
...
>>> frob.__annotations__
{'a': 'x', 'b': 11, 'c': [], 'return': 9}
```

- **Providing typing information**
  - **Type checking**
  - **Let IDEs show what types a function expects/returns**
  - **Function overloading / generic functions**
  - **Foreign-language bridges**
  - **Adaptation**
  - **Predicate logic functions**
  - **Database query mapping**
  - **RPC parameter marshaling**

- **Other information**
  - **Documentation for parameters and return values**

```
>>> def frobnicate(a: int, b: int, c: int) -> int:
...     return a + b + c
...
>>> frobnicate.__annotations__
{'a': int, 'b': int, 'c': int, 'return': int}
```

Jukka Lehtosalo

University of Cambridge

@di_codes

Unification

Of statically typed and dynamically typed languages

Using the same language

For tiny scripts and sprawling, multi-million line codebases

Gradual growth

from an untyped prototype to a statically typed product

@di_codes

stop.dvi

pdfs.semanticscholar.org/3979/6704851a76709671e7c3e10538ba4dd856fe.pdf

Language with a Pluggable Type System and Optional Runtime Monitoring of Type Errors

Jukka Lehtosalo and David J. Greaves

University of Cambridge Computer Laboratory
firstname.lastname@cl.cam.ac.uk

Abstract. Adding a static type system to a dynamically-typed language can be an invasive change that requires coordinated modification of existing programs, virtual machines and development tools. Optional pluggable type systems do not affect runtime semantics of programs, and thus they can be added to a language without affecting existing code and tools. However, in programs mixing dynamic and static types, pluggable type systems do not allow reporting runtime type errors precisely. We present *optional runtime monitoring of type errors* for tracking these errors without affecting execution semantics. Our Python-like target language Alore has a nominal optional type system with *bindable interfaces* that can be bound to existing classes by clients to help the safe evolution of programs and scripts to static typing.

1 Introduction

Dynamic typing enables high productivity for scripting, but it does not scale well to large-scale software development. Adding an optional static type system that allows gradually evolving a dynamically-typed program to a statically-typed one has been proposed as a solution to this problem [15–18].

Several factors make adding static type checking to a mature dynamically-typed language such as Python challenging. Adding the type system is an invasive change that affects the language in fundamental ways. All the tooling from virtual machines, compilers, debuggers to integrated debugging environments needs to be updated to be aware of the static type system.

This objection can be dealt with, in part, by using an *optional pluggable type sys-*

@di_codes

"Adding a static type system to a dynamically-typed language can be an invasive change that requires coordinated modification of existing programs, virtual machines and development tools."

– Jukka Lehtosalo

"Optional pluggable type systems do not affect runtime semantics of programs, and thus they can be added to a language without affecting existing code and tools."

– Jukka Lehtosalo

Presentation: Mypy: Optional S

Python Software Foundation [US] | us.pycon.org/2013/schedule/presentation/166/

Change the future

LOG IN | SIGN UP

About Events Speaking Sponsors Venue News PSF Registration

Mypy: Optional Static Typing for Python

Jukka Lehtosalo

Audience level: Intermediate
Category: Core Python (Language, Stdlib)

Description

Mypy is an experimental Python variant that supports seamless mixing of dynamic and static typing. The implementation can type check programs with optional type annotations and translate them to readable Python 3. The long-term goal of the project is to develop an ahead-of-time compiler that generates efficient native code.

Abstract

Mypy is an experimental variant of Python that supports writing programs that seamlessly mix dynamic and static typing. Mypy lets you add optional type annotations to Python code, type check your programs and translate them to readable Python 3 for execution.

I will give an informal overview of mypy and dynamic and static typing, and explain why having both dynamic and static typing in a programming language can be useful for Python developers. Static typing can, for example, make projects with multiple developers easier to maintain and refactor, it can improve efficiency and enable powerful IDE features such as precise code completion. I will also discuss what kinds of projects are likely to get the biggest benefits from static typing.

The mypy implementation is in development, but it is already self-hosting: the type checker and translator is implemented in mypy. The long-term goal of the project is to develop an ahead-of-time compiler that generates efficient native code and a new VM that supports efficient multi-threading without the GIL.

I will also contrast mypy with earlier projects with similar goals, such as PyPy and Cython.

Sponsors

Diamond

heroku
Google

Platinum

redhat
ebay local
New Relic
caktus GROUP
Eventbrite
newcars.com
SurveyMonkey

@di_codes

"Mypy is an experimental variant of Python that supports writing programs that seamlessly mix dynamic and static typing."

– Jukka Lehtosalo

```
int fib(int n):
    if n <= 1:
        return n
    else:
        return fib(n - 1) + fib(n - 2)
```

"I eventually presented my project at the PyCon 2013 conference in Santa Clara, and I chatted about it with Guido van Rossum, the BDFL of Python. He convinced me to drop the custom syntax and stick to straight Python 3 syntax."

– Jukka Lehtosalo

PEP 483

The Theory of Type Hints

Optional typing

Only gets in your way if you want it to get in your way

Gradual typing

Let's not try to do this all at once

Variable annotations

For annotating more than just functions

```python
def frobnicate(a: int, b: int, c: int) -> int:
    bizbaz = a + b + c
    return bizbaz
```

```python
def frobnicate(a: int, b: int, c: int) -> int:
    bizbaz = a + b + c  # type: int
    return bizbaz
```

Type hinting for Python 2

Because even those stuck in the past deserve static typing

@di_codes

```python
# Python 3
def frobnicate(a: int, b: int, c: int) -> int:
    return a + b + c

# Python 2
def frobnicate(a, b, c):
    # type: (int, int, int) -> int
    return a + b + c
```

Special type constructs

Fundamental building blocks we need to do static typing

- **Existing types:** `int`, `float`, `str`, `NoneType`, **etc.**
- **New types: (**`from typing import ...`**)**
  - `Any`: **consistent with any type**
  - `Union[t1, t2, ...]`: **at least one of** `t1`, `t2`, **etc.**
  - `Optional[t1]`: **alias for** `Union[t1, NoneType]`
  - `Tuple[t1, t2, ...]`: **tuple whose items are** `t1`, **etc.**
  - `Callable[[t1, t2, ...], tr]`: **a function**

```python
def frobnicate(
    a: int, b: int, c: Union[int, float]
) -> Union[int, float]:
    return a + b + c
```

Container types

For defining types inside container classes

```python
users = []  # type: List[int]
users.append(42)  # OK
users.append('Some Guy')  # fails

examples = {}  # type: Dict[str, int]
examples['Some Guy'] = 42  # OK
examples[2] = None  # fails
```

Generic types

For when a class or function behaves in a generic manner

```python
from typing import Iterable

class Task:
    ...

def work(todo_list: Iterable[Task]) -> None:
    ...
```

Type aliases

To be more succinct

```python
from typing import Union
from decimal import Decimal

Number = Union[int, float, complex, Decimal]

def frob(a: Number, b: Number, c: Number) -> Number:
    "Frobnicates the bizbaz"
    return a + b + c
```

PEP 484

Type Hints

Python 3.5

Released: September 13, 2015

PEP 526

Syntax for Variable Annotations

```python
# 'primes' is a list of integers
primes = []  # type: List[int]

# 'captain' is a string (initial value is a problem!)
captain = ...  # type: str

class Starship:
    # 'stats' is a class variable
    stats = {}  # type: Dict[str, int]
```

```python
# 'primes' is a list of integers
primes: List[int] = []

# 'captain' is a string (initial value is a problem!)
captain = ...  # type: str

class Starship:
    # 'stats' is a class variable
    stats = {}  # type: Dict[str, int]
```

```python
# 'primes' is a list of integers
primes: List[int] = []

# 'captain' is a string
captain: str  # Note: no initial value!

class Starship:
    # 'stats' is a class variable
    stats = {}  # type: Dict[str, int]
```

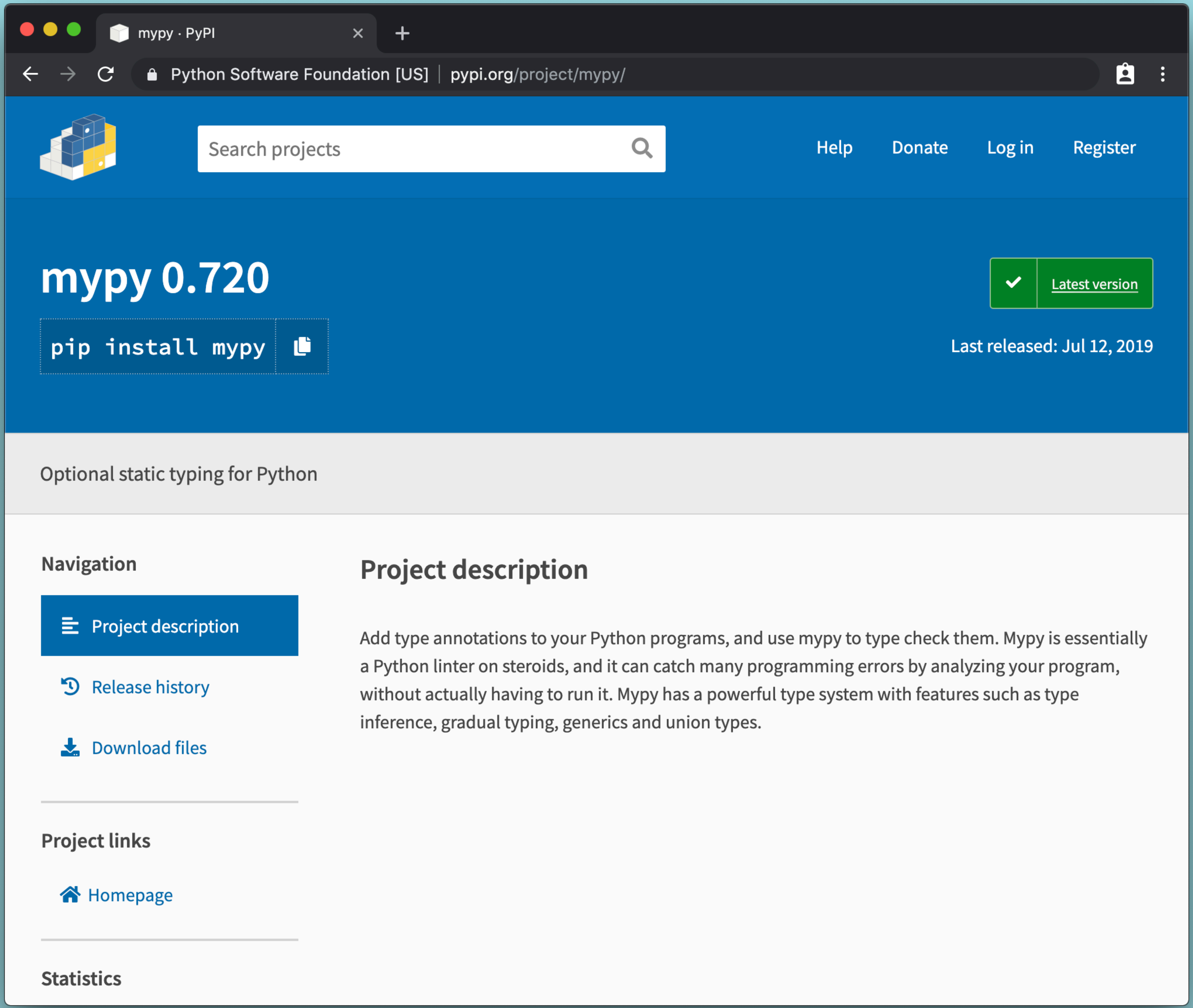
```python
# 'primes' is a list of integers
primes: List[int] = []

# 'captain' is a string
captain: str  # Note: no initial value!

class Starship:
    # 'stats' is a class variable
    stats: ClassVar[Dict[str, int]] = {}
```

Python 3.6

Released: December 23, 2016

Type checkers

Static vs. dynamic


mypy · PyPI
Python Software Foundation [US] | pypi.org/project/mypy/
Search projects
Help
Donate
Log in
Register
mypy 0.720
pip install mypy
Latest version
Last released: Jul 12, 2019
Optional static typing for Python
Navigation
Project description
Release history
Download files
Project links
Homepage
Statistics
Project description
Add type annotations to your Python programs, and use mypy to type check them. Mypy is essentially a Python linter on steroids, and it can catch many programming errors by analyzing your program, without actually having to run it. Mypy has a powerful type system with features such as type inference, gradual typing, generics and union types.

@di_codes

```
$ pip install mypy
...

$ cat frob.py
def frobnicate(a: int, b: int, c: int) -> int:
    return a + b + c

frobnicate('hi', ' ', 'there')

$ mypy frob.py
frob.py:4: error: Argument 1 to "frobnicate" has incompatible type
"str"; expected "int"
frob.py:4: error: Argument 2 to "frobnicate" has incompatible type
"str"; expected "int"
frob.py:4: error: Argument 3 to "frobnicate" has incompatible type
"str"; expected "int"
```

- **Static**
  - `mypy` **(Dropbox)**
  - `pytype` **(Google)**
  - `pyre` **(Facebook)**
  - `pyright` **(Microsoft)**
  - **PyCharm,** `$YOUR_EDITOR`
- **Dynamic**
  - `enforce`, `typeguard`, `typo`, `ducktype`, `strictconf`, **etc.**

Differences between `mypy` and `pytype`

Cross-function inference, runtime lenience

```python
# example.py

def f():
    return "EuroPython"

def g():
    return f() + 2020

g()
```

@di_codes

```
$ python example.py
Traceback (most recent call last):
  File "example.py", line 5, in <module>
    g()
  File "example.py", line 4, in g
    return f() + 2020
TypeError: can only concatenate str (not "int") to str
```

```
$ mypy example.py
```

```
$ mypy example.py

$
```

```
$ mypy example.py

$ pytype example.py
```

```
$ mypy example.py

$ pytype example.py
Computing dependencies
Analyzing 1 sources with 0 local dependencies
[1/1] check test
FAILED: /tmp/.pytype/pyi/example.pyi pytype-single --imports_info
/tmp/.pytype/imports/test.imports --module-name test -V 3.7 -o
/tmp/.pytype/pyi/test.pyi --analyze-annotated --nofail --quick
/tmp/example.py
File "/tmp/example.py", line 4, in g: unsupported operand type(s)
for +: 'str' and 'int' [unsupported-operands]
  Function __add__ on str expects str

For more details, see
https://google.github.io/pytype/errors.html#unsupported-operands.
```

```python
# example.py

from typing import List

def f() -> List[str]:
    lst = ["PyCon"]
    lst.append(2020)
    return [str(x) for x in lst]

print(f())
```

```
$ python example.py
['PyCon', '2020']
```

```
$ pytype example.py
Computing dependencies
Analyzing 1 sources with 0 local dependencies
ninja: Entering directory `/private/tmp/.pytype'
[1/1] check example
Success: no errors found
```

```
$ pytype example.py
Computing dependencies
Analyzing 1 sources with 0 local dependencies
ninja: Entering directory `/private/tmp/.pytype'
[1/1] check example
Success: no errors found

$ mypy example.py
example.py:7: error: Argument 1 to "append" of "list"
has incompatible type "int"; expected "str"
```

y tho

When (and why) we should use static typing

When you *shouldn't* use static typing

Basically never

Static typing:

Not a replacement for unit tests

When you *should* use static typing

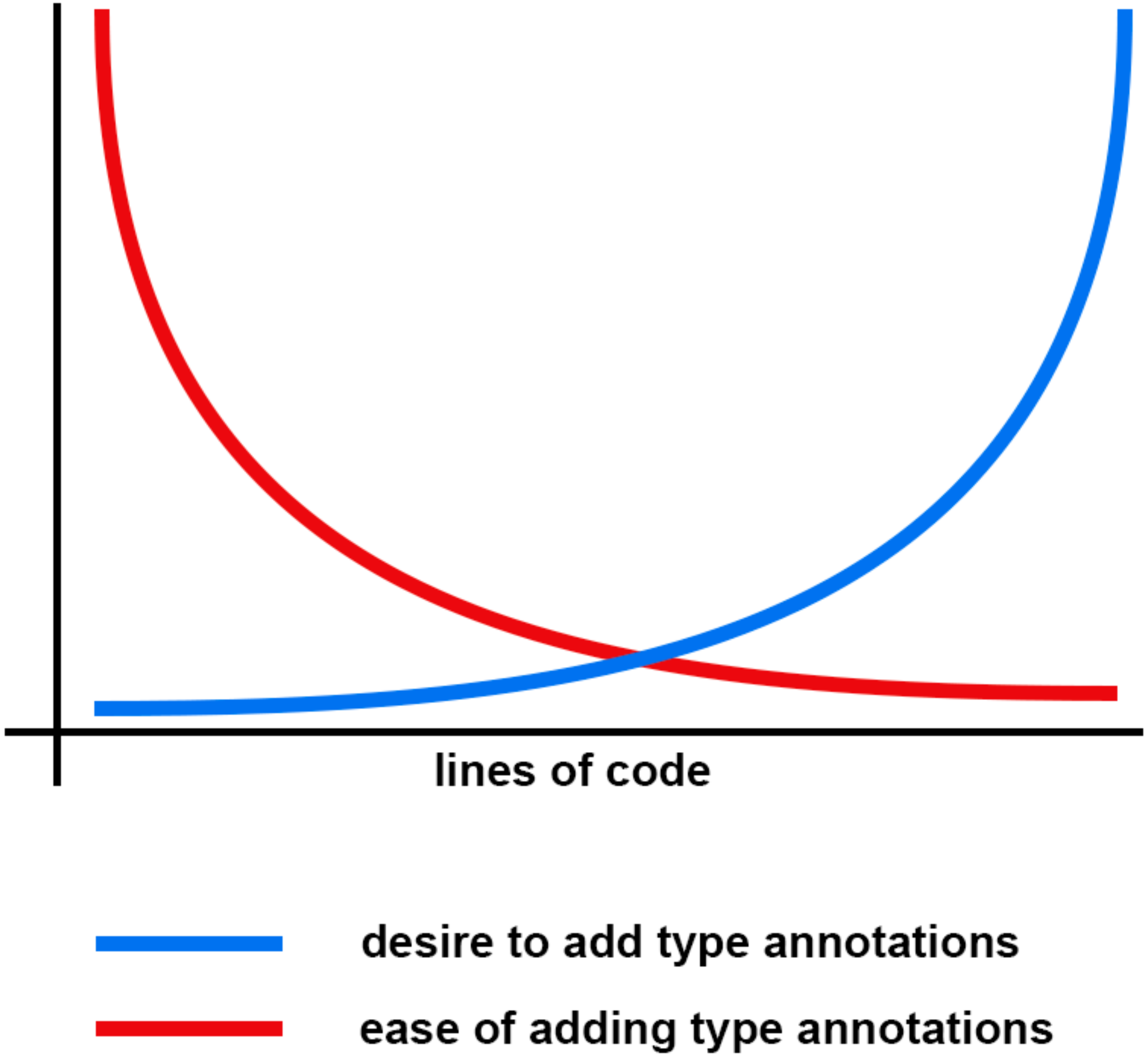
Basically as much as possible

Use static typing:

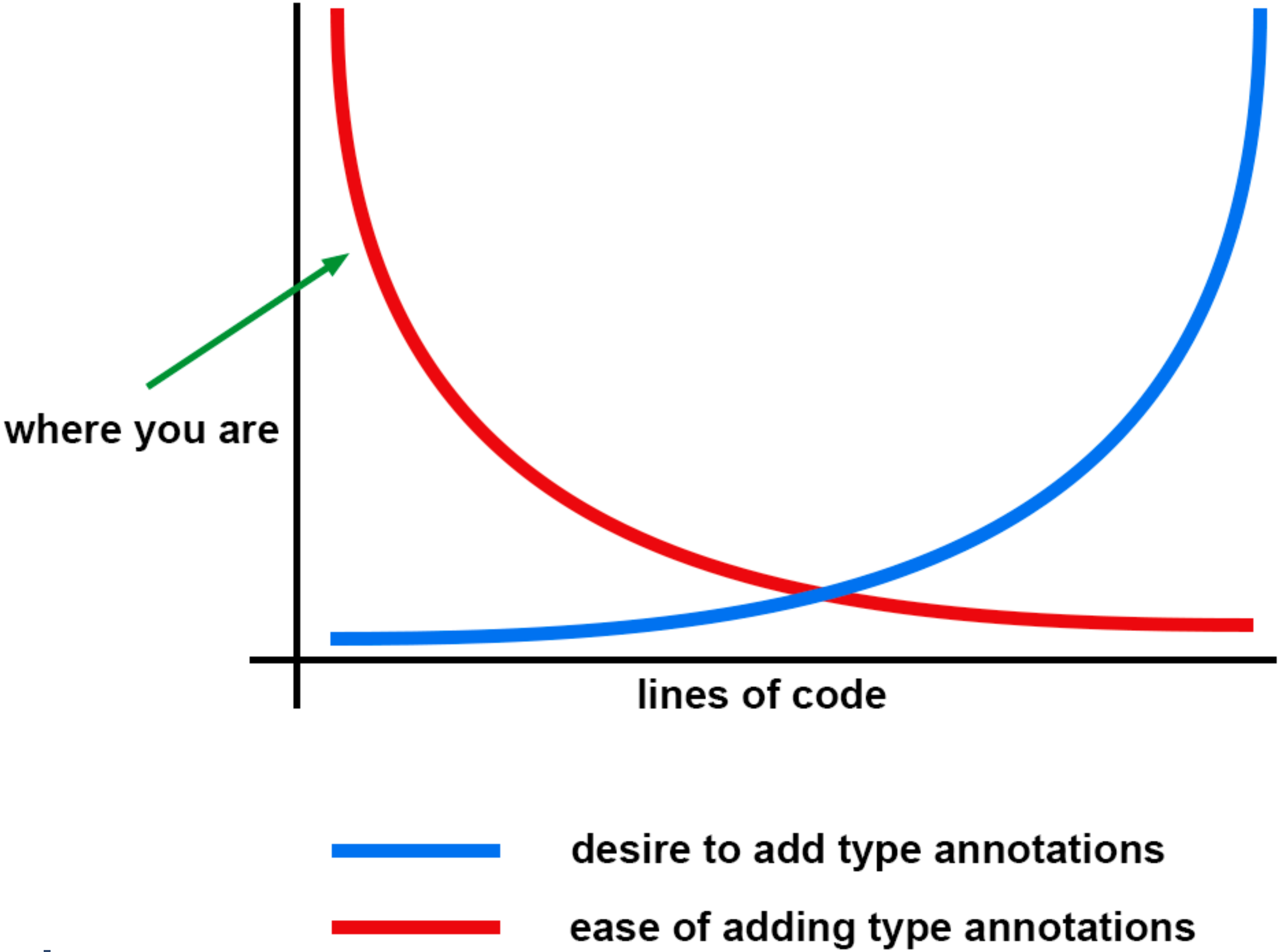
When you're millions-of-lines scale

"At our scale—millions of lines of Python—the dynamic typing in Python made code needlessly hard to understand and started to seriously impact productivity."

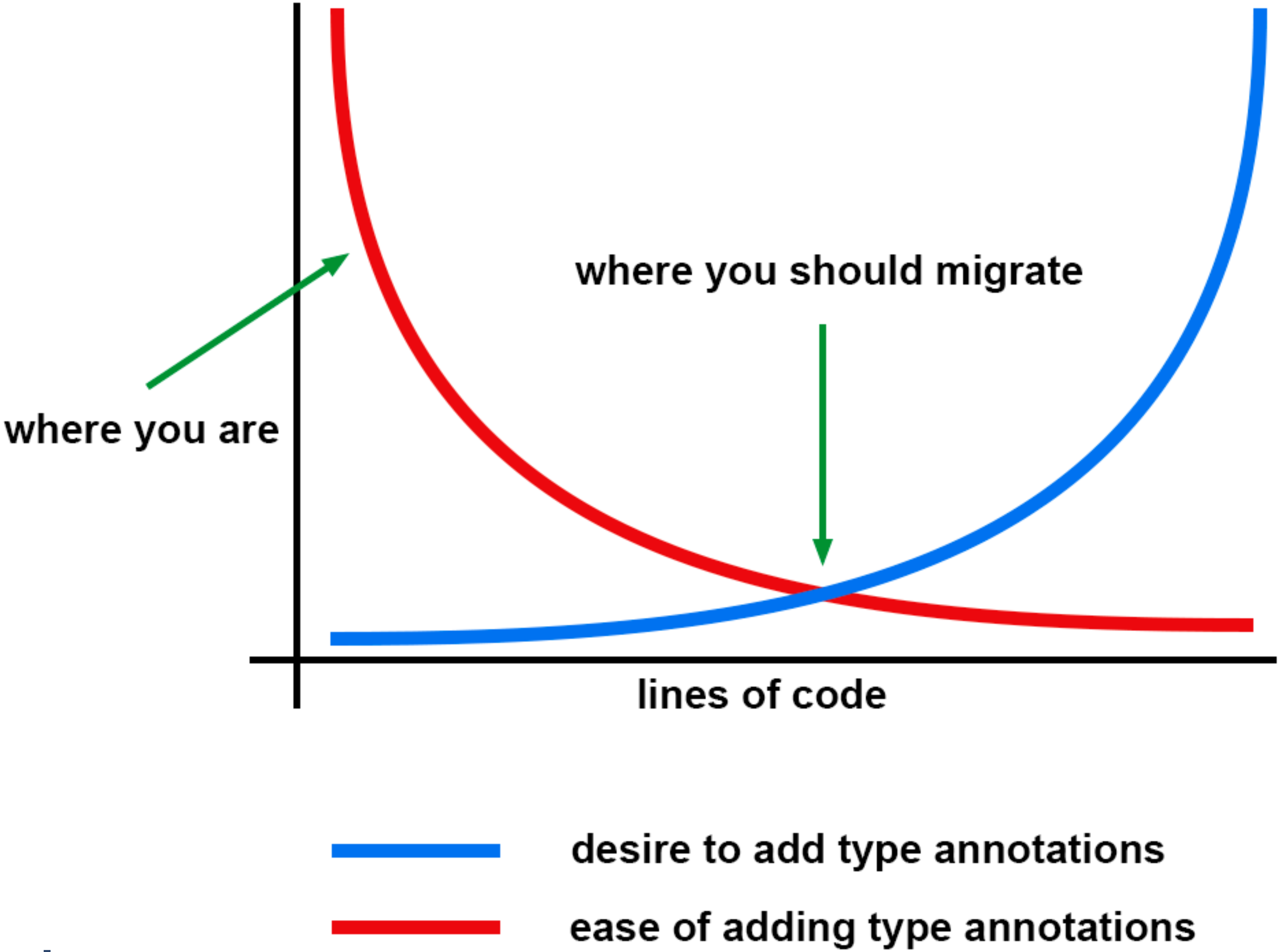
– Jukka Lehtosalo


lines of code
desire to add type annotations
ease of adding type annotations

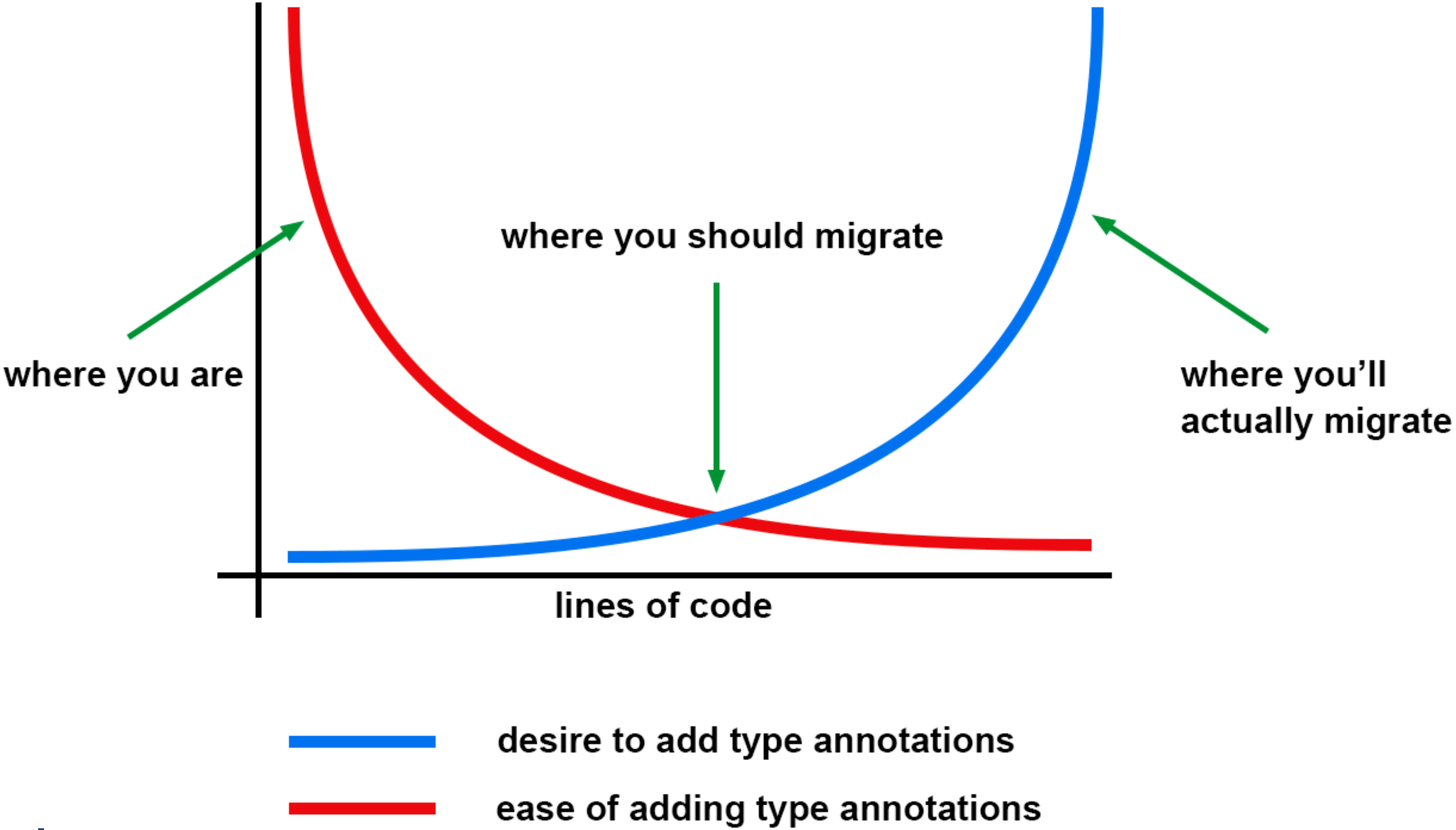
@di_codes


where you are
lines of code
desire to add type annotations
ease of adding type annotations

@di_codes


where you should migrate
where you are
lines of code
desire to add type annotations
ease of adding type annotations

@di_codes


where you should migrate
where you are
where you'll
actually migrate
lines of code
desire to add type annotations
ease of adding type annotations

Use static typing:

When your code is confusing

Use static typing:

When your code is for public consumption

Use static typing:

Before migrating or refactoring

Use static typing:

To experiment with static typing

How to use static typing in Python

In just *five* easy steps!

@di_codes

1. Migrate to Python >= 3.6 (optional)

1. Migrate to Python >= 3.6 (optional)
2. Install a typechecker locally

1. Migrate to Python >= 3.6 (optional)
2. Install a typechecker locally
3. Start optionally typing your codebase

1. Migrate to Python >= 3.6 (optional)
2. Install a typechecker locally
3. Start optionally typing your codebase
4. Run a typechecker with your linting

1. Migrate to Python >= 3.6 (optional)
2. Install a typechecker locally
3. Start optionally typing your codebase
4. Run a typechecker with your linting
5. Convince all your coworkers to join you

Thanks!

@di_codes