Dropbox

Optional static type checking for Python using mypy

Guido van Rossum
Principal Engineer, Dropbox

Static type checking

- Type checking done by the compiler, before running the code

talk — -bash — 63×8

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

public static void main(String args[]) {
    System.out.println("The answer is " + add(2, 40));
}
```

Dynamic type checking

- Type checking is based on actual values at run time

talk — -bash — 20×5

```
def add(a, b):
    return a + b
```

talk — python3.8 — 65×13

```
>>>
>>> from util import add
>>> add(2, 40)
42
>>> add("hello ", "world")
'hello world'
>>> add(42, "hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/guido/talk/util.py", line 2, in add
    return a + b
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

Perfect, right?

- Why would you want it any other way?

Perfect, right?

- Why would you want it any other way?

talk — -bash — 22×9

```
def add(a, b):
    return a + b

def total(ns):
    t = 0
    for n in ns:
        t = add(t, n)
    return t
```

talk — python3.8 — 65×13

```
>>> from util import total
>>> total(range(10))
45
>>> total(["a", "b", "c"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/guido/talk/util.py", line 7, in total
    t = add(t, n)
  File "/Users/guido/talk/util.py", line 2, in add
    return a + b
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
>>>
```

Long story short...

- Python is hard to beat for young and/or small code bases
- But for old, large code bases, is there a better way?
  - Other than rewriting everything in Java :-)
- Based on the response to mypy, yes!
- *Optional static type checking*, a.k.a. *gradual typing* helps
  - Doesn't compromise runtime type safety
  - Type annotations are an added safety feature
  - Like a linter on steroids

Why type annotations

- Find bugs faster/cheaper
- Find different classes of bugs than tests
- Help with refactoring
    - Special case: Python 2 to 3 migration
- Docs that are automatically verified
- While debugging
    - What's expected compared to actual value

Non-goals

- Run-time type checking
  - Too slow, especially for generics

- Machine code generation
  - Python's precise semantics are hard to capture
  - *(However...)*

Gradual typing

- You can't annotate millions of lines in one fell swoop
- Gradual typing lets you add annotations one function at a time
  - Annotated and unannotated code can be mixed freely
- At the boundaries, type checks are suppressed
  - This is a compromise for usability

talk — -bash — 24×6

```
def foo(x):
    return x + 1

def bar(x: str) -> str:
    return foo(x)
```

Basics

- Function annotations

- Type constructors
  - Union[int, str]
  - Optional[float]
  - Tuple[int, int, str]
  - Tuple[int, ...]

- Variable declarations

talk — -bash — 34×6

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

talk — -bash — 43×8

```
class Point:
    x: float
    y: float

    def length(self) -> float:
        z: float = self.x**2 + self.y**2
        return math.sqrt(z)
```

Generics

- List[int], Dict[str, float], Set[str], ...
- Iterable[str], Sequence[Tuple[int, int]], Mapping[str, List[str]], ...

talk — -bash — 45×12

```
from typing import Generic, TypeVar, Iterable

T = TypeVar("T")

class Array(Generic[T]):

    def __init__(self, xs: Iterable[T]):
        self.xs = list(xs)

    def __getitem__(self, i: int) -> T:
        return self.xs[i]
```

talk — -bash — 41×7

```
IntArray = Array[int]

counts = IntArray([42, 123, 1_000_000])

i = counts[0]
x = counts[3.14]  # ERROR
```

Protocols

- For duck typing
  - Similar to Go interfaces

talk — -bash — 42×10

```
from typing import Protocol, TypeVar

T = TypeVar("T")

class Stack(Protocol[T]):

    def push(self, x: T) -> None: ...

    def pop(self) -> T: ...
```

talk — -bash — 36×19

```
class IntStack:

    xs: List[int]

    def __init__(self):
        self.xs = []

    def push(self, x: int) -> None:
        self.xs.append(x)

    def pop(self) -> int:
        return self.xs.pop()

def f(x: Stack[int]) -> int:
    x.push(42)
    return x.pop()

f(IntStack())
```

Pragmatics

- Must import things from typing
- Stub files, e.g. builtins.pyi
  - Contain signatures for classes/functions
  - Collection of stub files on GitHub: typeshed
- Type comments
  - For Python 2
  - Also allowed in Python 3
    - For straddling code

talk — -bash — 34×6

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

Nasty bits

- Unions and Optional are verbose
- Forward references are ugly
  - But... from __future__ import annotations
- Callable[[int, int], int] is verbose
- Sometimes need cast(type, value) or # type: ignore
- Unannotated code is not checked by default
  - This is the essence of gradual typing, but still a surprise

History lesson

- In the early 1990s, dynamic typing was the underdog
  - Think Perl vs. C++
- As the web grew, LAMP popularized dynamic languages
  - Perl/PHP/Python/Ruby (also JavaScript)
  - Python was the secret weapon of web startups
- As early as 1998, a types-sig was formed
  - Burned up quickly, disbanded in 2000
  - I found a talk from 1/19/2000 that was pretty prescient!
- Here are some sample slides

Optional Static Typing

Guido van Rossum
(with Paul Prescod, Greg Stein,
and the types-SIG)

Why Add Static Typing?

- Two separate goals:
  - faster code (OPT)
  - better compile-time errors (ERR)
- Mostly interested in (ERR)
  - (OPT) will follow suit
- Of course it will be optional
  - and (mostly) backwards compatible

Declaration Syntax

- Two forms: *inline* and *explicit*
  - explicit form is easy to remove
- Inline:
  - def gcd(a**: int**, b**: int**) **->** **int**: ...
- Explicit (two variants):
  - **decl gcd: def(int, int) -> int**
    def gcd(a, b): ...
  - def gcd(a, b):
        decl a: int, b: int
        decl return: int
        ...

Constructing Types

- Syntax for type composition:
  - list with items of type T: **[T]**
  - tuple of T1, T2, T3: **(T1, T2, T3)**
    - (this explains why we have both tuples and lists!)
  - dict with key/value types T1/T2: **{T1: T2}**
  - union of types T1 and T2: **T1 | T2**
  - function (e.g.): **def(T1, T2)->T3**
- Example:
  - **{str: (int, int) | (int, int, str) | None}**

Parameterized Types

- Needed e.g. for container classes:

```
class Stack<T>:
    decl st: T
    def __init__(self):    self.st = []
    def push(self, x: T): self.st.append(x)
    def pop(self) -> T: x = self.st[-1]; del self.st[-1]; return x

decl IntStack = Stack<int> # template instantiation
decl s: IntStack
s = IntStack() # or s = Stack() ???
s.push(1)
decl x: int
x = s.pop()
s.push("spam") # ERROR
```

History repeats itself

- In 2004-2005 I wrote two large blog posts about the topic
- Many of the same ideas, but improved
- Basic function annotation still the same
- Uses sequence(T) instead of Sequence[T]
- Different syntax to create a generic class
  - Not constrained by existing syntax

All Things Pythonic

Adding Optional Static Typing to Python

by Guido van van Rossum
December 23, 2004

Summary
Optional static typing has long been requested as a Python feature. It's been studied in depth before (e.g. on the type-sig) but has proven too hard for even a PEP to appear. In this post I'm putting together my latest thoughts on some issues, without necessarily hoping to solve all problems.

An email exchange with Neal Norwitz that started out as an inquiry about the opening of a stock account for the PSF (talk about bizarre conversation twists) ended up jogging my thoughts about optional static typing for Python.

Let's look at a simple function:

```
def gcd(a, b):
  while a:
    a, b = b%a, a
  return b
```

This pretty much only makes sense with integer arguments, but the compiler won't stop you if you call it with string or floating point arguments. Purely based on the type system, those types are fine: the % operator on two strings does string formatting (e.g. "(%s)" % "foobar" gives "(foobar)"), and Python happens to define % on floats as well (3.7 % 0.5 gives 0.2). But with string arguments the function is likely to raise a TypeError (gcd("", "%s") notwithstanding) and float arguments often cause bogus results due to the rounding errors.

So let's consider a simple type annotation for this function:

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

This time it happened

- Much debate followed
- There was no agreement on generics
- But we agreed on function annotations
- In 2006, PEP 3107 was accepted
- Syntax for arguments and return types
- Semantics left to 3rd party code
  - Annotations end up in __annotations__ dict on function

PEP 3107 -- Function Annotations

PEP:	3107
Title:	Function Annotations
Author:	Collin Winter <collinwinter at google.com>, Tony Lownds <tony at lownds.com>
Status:	Final
Type:	Standards Track
Created:	2-Dec-2006
Python-Version:	3.0
Post-History:	

Fundamentals of Function Annotations

Before launching into a discussion of the precise ins and outs of Python 3.0's function annotations, let's first talk broadly about what annotations are and are not:

1. Function annotations, both for parameters and return values, are completely optional.

2. Function annotations are nothing more than a way of associating arbitrary Python expressions with various parts of a function at compile-time.

   By itself, Python does not attach any particular meaning or significance to annotations. Left to its own, Python simply makes these expressions available as described in Accessing Function Annotations below.

   The only way that annotations take on meaning is when they are interpreted by third-party libraries. These annotation consumers can do anything they want with a function's annotations. For example, one library might use string-based annotations to provide improved help messages, like so:

```
def compile(source: "something compilable",
            filename: "where the compilable thing comes from",
            mode: "is this a single statement or a suite?"):
```

Then, nothing

- At least not until 2014, when Python 3.5 was being hatched
- In 2015, after much fireworks, PEP 484 was accepted
- Here's how that happened
- Thanks to a soft-spoken Finn, Jukka Lehtosalo
- Read about it on the Dropbox blog
  - "Our journey to type checking 4 million lines of Python"
  - Posted September 5, 2019

Type checking at Dropbox

- 2012: Jukka designs Alore; gradually typed, translates to Python
- 2013: Guido suggests to target PEP 3107; Dropbox hires Jukka
- 2014: Dropbox Hack Week experiments, PEP 484 started
- 2015: Python 3.5 ships with PEP 484, mypy matures
- 2016: Introduction of mypy in Dropbox CI; mypy team formed
- 2017-2019: mypy conquers Dropbox, and the world

Performance

- When you have a popular tool, performance becomes an issue
- We went through several stages
    1. Incremental mode: cache unchanged modules on disk
    2. Download pre-computed cache
    3. Daemon: cache unchanged functions in memory
    4. Shave seconds off integration scripts startup time
    5. Write a compiler (mypyc) — 4x speedup
- Watch Michael Sullivan's talk at PyCon US 2019

Future

- Several typing PEPs in the works
  - 585: replace List[int] with list[int] (etc.) — finally!
  - 593: Annotated[t, extra, ...] — to add 3rd party metadata
  - 604: replace Union[int, str] with int|str — finally!
  - TBD: more explicit syntax for type aliases
- Type system features to support numpy, pandas etc.
  - Shape types: matmul(Array[N, M], Array[M, K]) -> Array[N, K]
  - Variadic type variables: sum(Array[Ts], Array[Ts]) -> Array[Ts]
- Productionize mypyc
  - Looking for early adopters

Open source

- Fork us on GitHub!
  - github.com/python/mypy
  - github,com/python/typeshed

Dropbox