Static Typing for Python

Jukka Lehtosalo, Guido van Rossum,
David Fisher, Greg Price
with thanks to Reid Barton

PyCon, 5/29/2016, Portland, OR


GOOD NEWS, EVERYONE!

Good news

Python has optional static types (in beta).

You can use them in your code today (2.7 included!)

This workshop: why and how.

Talk preview

- Quick hello-world (Greg)
- Why, and what (Greg)
- History (Guido)
- How (Jukka)
- … with demo (David)
- Future (David)
- Q&A

type: hello world

Hello world

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

  print(gcd(10, 15))
  ```
  ```

Hello world: writing types

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

  print(gcd(10, 15))

Hello world: writing types, in Python 2 or 3

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

  print(gcd(10, 15))

Hello world: writing types

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

  print(gcd(10, 15))

Hello world: run it

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

    print(gcd(10, 15))

- $ python3 program.py
  5

Hello world: check it

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

  print(gcd(10, 15))
  ```

- $ python3 program.py
  5

- $ mypy program.py

Hello wrlod (with a bug)

- def gcd(a: int, b: int) -> int:
        while b:
            a, b = b, a % b
        return a
    print(gcd('**x**', 15))

Hello wrlod: run it

- def gcd(a: int, b: int) -> int:
        while b:
            a, b = b, a % b
        return a
  print(gcd('**x**', 15))

- $ python3 program.py
  Traceback (most recent call last):
    File "program.py", line 5, in <module>
      print(gcd('x', 15))
    File "program.py", line 3, in gcd
      a, b = b, a % b
  TypeError: not all arguments converted during string formatting

Hello wrlod: check it

- def gcd(a: int, b: int) -> int:
        while b:
            a, b = b, a % b
        return a
  print(gcd('**x**', 15))

- $ mypy program.py
  program.py:5: error: Argument 1 to "gcd" has incompatible type "str"; expected "int"

Why static types

Why static types

Lots of things static types **can** do

… even lots of things “static types” can mean

For us it’s all about one thing

Why static types: understanding code

Humans: big part of the craft
  e.g., 40% of eng time at Dropbox (2015 survey)

Computers: cross-reference, refactor, find bugs

Understanding code

What does this code do?

```
for entry in entries:
    entry.data.validate()
```

Understanding code

What does this code do?

```
for entry in entries:
    entry.data.validate()
```

Need to find “validate”. Grep for its definition...

Understanding code

What does this code do?

```
for entry in entries:
    entry.data.validate()
```

Need to find “validate”. Grep for its definition...

... there are 154 of them. (True story!)

Understanding code: types

What does this code do?

```
for entry in entries:
    entry.data.validate()
```

Grep for “validate” definition… 154 of them. Which?

Have to know the type.

Understanding code: types

What does this code do?

```
for entry in entries:
    entry.data.validate()
```

So we need to know what an "entry.data" is...

Understanding code: types

What does this code do?

```
for entry in entries:
    entry.data.validate()
```

So we need to know what an "entry.data" is...

... which we could figure out if we knew what "entry" is...

Understanding code: types

What does this code do?

```
for entry in entries:
    entry.data.validate()
```

So we need to know what an "entry.data" is...

... which we could figure out if we knew what "entry" is...

... which we would know if we knew what "entries" is.

Understanding code: types

```
for entry in entries:
    entry.data.validate()
```

What's the type of "entries"?

Understanding code: types

```
for entry in entries:
    entry.data.validate()
```

What's the type of "entries"?

If a parameter, go find call sites...

Understanding code: types

```
for entry in entries:
    entry.data.validate()
```

What's the type of "entries"?

If a parameter, go find call sites...
… but name may have same issue as “validate”

Understanding code: types

```
for entry in entries:
    entry.data.validate()
```

What's the type of "entries"?

If a parameter, go find call sites...

… but name may have same issue as “validate”

… and at each call site, same problem recursively!

Understanding code: types

```
for entry in entries:
    entry.data.validate()
```

What's the type of "entries"?

If a parameter, go find call sites;
if a return value, find that method...

Understanding code: types

```
for entry in entries:
    entry.data.validate()
```

What's the type of "entries"?

If a parameter, go find call sites;
if a return value, find that method;
if an attribute, go study that class...

Understanding code: types

```
for entry in entries:
    entry.data.validate()
```

What's the type of "entries"?

If a parameter, go find call sites;
if a return value, find that method;
if an attribute, go study that class; etc.

Each case, the same problem recursively.

Understanding code: types

What's the type of (some expression)?

Without type annotations, can find out, but a lot of work

Understanding code: types

What's the type of (some expression)?

Without type annotations, can find out, but a lot of work

... and maybe the type varies!

e.g.: subclasses, duck typing, generic containers

Understanding code: types

What's the type of (some expression)?

Without type annotations, can find out, but a lot of work

... and maybe the type varies!

e.g.: subclasses, duck typing, generic containers

… and undecidable -- can never do for all programs!

Understanding code: types!

What's the type of (some expression)?

Lots of excuses, but:

The **author** had some kind of answer.

> “int”; “iterable of string”; “could actually be anything”

That would be enough.

Understanding code: static types

What's the type of (some expression)?
The author had some kind of answer.

static type, *n.*: the expectation the author had of the (runtime) type of an expression's value *(usage note: many competing definitions in academia)*

Static types: writing them down

static type, *n.*: the expectation the author had of the (runtime) type of an expression's value

Explicit is better than implicit.

Static types: writing them down

Explicit is better than implicit.

```
def validate_entries(self, entries):
    '''entries: a list of LogEntry'''
    for entry in entries:
        entry.data.validate()
```

Static types: writing them down

Explicit is better than implicit.

Checked is better than unchecked.
(for statements about the code)

Static types: writing them down

static type, *n.*: the expectation the author had of the (runtime) type of an expression's value

New strategy: make these explicit and checked.

Static types: writing them down

static type, *n.*: the expectation the author had of the (runtime) type of an expression's value

New strategy: make these explicit and checked.

1. Write your types at “def” and where not obvious

Static types: writing them down

static type, *n.*: the expectation the author had of the (runtime) type of an expression's value

New strategy: make these explicit and checked.

1. Write your types at “def” and where not obvious
2. Run mypy to check the types, routinely like your tests

Static types: writing them down

static type, *n.*: the expectation the author had of the (runtime) type of an expression's value

New strategy: make these explicit and checked.

1. Write your types at “def” and where not obvious
2. Run mypy to check the types, routinely like your tests
3. Read code, learn types for free

Static types: writing them down

1. Write your types at “def” and where not obvious
2. Run mypy to check the types, routinely like your tests
3. Read code, learn types for free

We’re doing this at Dropbox!

Started with a handful of users, they spread it rapidly to coworkers.

What Dropbox engineers say

- "It was really easy for us to adopt, it has improved our code and helped our velocity."
- "This makes it sooo much easier for me to read code and figure out what parameters are supposed to be!"
- "Refactors are already so much easier with the typing that's been added. Game changing [...]!"
- “It has addressed my personal biggest pain point with python. It’s really increased the scope of things I’m excited to consider python as great for.”

Some History

This is Not a Sudden Change!

- I've been thinking about this for nearly two decades

- Let me walk you through some significant events
  - naming names of significant influencers

First Sighting of Function Annotations

- 1998-2002: types-sig
  - started out with discussion about interfaces, initiated by Jim Fulton
    - (this work eventually led to zope.interface)
  - a talk I gave in 2000 has the exact gcd example!

```
def gcd(a: int, b: int) -> int:
    # etc.
```

    - joint work with Greg Stein, Paul Prescod

First Sighting of Generics

- 2004-2005: Artima blog posts
  - shows use of list(int), iterable(int) or Iterable(int)
  - also generic functions:

```
def min(a: T, b: T) -> T:
    # etc.
```

  - also overloading:

```
@overloaded
def min(a: T, b: T) -> T:
    # etc.
@overloaded
def min(a: iterable(T)) -> T:
    # etc.
```

PEP 3107 Compromise

- 2006: PEP 3107, function annotations
  - we couldn't agree on the syntax for complex types
  - but we could agree on where they would go
  - same form as proposed in 2000

```
def foo(arg1: expr1, arg2: expr2) -> expr3:
    # etc.
```

  - values of expr1, expr2, expr3 available for runtime introspection

```
foo.__annotations__
{'arg1': expr1, 'arg2': expr2, 'return': expr3}
```

  - expr1, expr2, expr3 evaluated once at function definition time
    - same constraints as argument default values

History of mypy

- 2012: Jukka also started thinking about this
  - started mypy as an "experimental Python variant"

- 2013: I convinced Jukka to stay compatible with Python 3
  - PEP 3107 annotation syntax
  - List[int] etc. for generic classes

- 2014: Bob Ippolito's EuroPython talk "What can python learn from Haskell?"
  - bob.ippoli.to/python-haskell-ep2014 or watch on YouTube
  - two unrealistic suggestions (drop mutability, add ADTs)
  - one actionable idea: "adopt mypy"

Standardization

- 2014: discussion about PEP 484 started
  - influenced by mypy, also influencing mypy
  - megathreads on python-ideas, python-dev
  - discovered Jeremy Siek's "Gradual Typing" work
- 2015: BDFL presentations about PEP 484 (BayPiggies, PyCon, EuroPython)
- 2015 (May 22): PEP 484 accepted by BDFL-delegate Mark Shannon
  - Jukka started to make mypy compliant with the PEP
- PyCharm and Google's pytype also start adopting this standard

Dropbox Adoption

- Started for real in January 2016
- Planning to annotate most of our code
  - millions of lines of Python 2.7
- Several people working full-time on mypy
  - all our work goes into the open source mypy project
  - (there's a healthy set of outside contributors too!)

All This is Optional and Gradual

- We're not taking Python's dynamic behavior away
- Python ignores annotations (except setting __annotations__)
- You can add annotations to all, some, or none of your code
- mypy is silent when unannotated and annotated code meet
  - annotated code calls unannotated function — treats as 'Any'
  - unannotated code calls annotated function — not checked
  - this minimizes annoying noise from false positives

Try It!

- Mypy is quite usable today
- pip install **mypy-lang**
- Come find us at the sprints for mypy clinics
  - Thu: Greg, David, Reid, Jukka, Guido
  - Fri-Sat: Greg, David, Red
- Open space (time TBD)

Annotations

Built-in Types

```
def box(message: str, min_width: int = 0) -> str:
    width = max(len(message) + 4, min_width)
    return (width * '#' + '\n' +
            '# {} #\n'.format(message.center(width - 4)) +
            width * '#')

>>> print(box('hello', 12))
############
#   hello  #
############
```

User-defined Types and Library Types; None

```
import datetime

class Person:
    def __init__(self, name: str, date_of_birth: datetime.date) -> None:
        self.name = name
        self.date_of_birth = date_of_birth

def dump(p: Person) -> None:
    print('Name: {}'.format(p.name))
    print('Born: {}'.format(p.date_of_birth))
```

List[t]

```
from typing import List

def print_box(lines: List[str]) -> None:
    width = max(len(line) for line in lines) + 4
    print(width * '#')
    for line in lines:
        print('# {} #'.format(line.center(width - 4))
    print(width * '#')
```

The typing Module

- typing defines building blocks for types and other helpers
  - List[str]
  - Dict[int, str]
  - Iterable[bool], Sequence[int]
  - … and many others
- In Python 3.5+ standard library (provisional)
- Back-ported to Python 2.7 and 3.2+ (pip install typing)

Abstract Base Classes (ABCs)

An ABC describes an *interface* but doesn't provide a full *implementation*.

Many concrete classes may conform to an ABC.

Examples:

- Iterable[t] is something with an __iter__ method
- Iterator[t] is something with __iter__ and __next__

(ABCs were introduced back in Python 2.6.)

Sequence[t]

from typing import Sequence

```
def print_box(lines: Sequence[str]) -> None:
        width = max(len(line) for line in lines) + 4
        print(width * '#')
        for line in lines:
                print('# {} #'.format(line.center(width - 4))
        print(width * '#')

print_box(['hello', 'there'])                # ok (list)
print_box((‘one’, ‘two’, ‘three’))           # ok (tuple)
print_box(3)                                 # error
```

Dict[...] and Tuple[...]

from typing import Dict, Iterator, Tuple

```
def reverse_dict(d: Dict[str, int]) -> Iterator[Tuple[int, str]]:
    return ((value, key) for key, value in d.items())
```

We could do even better:

- Use typing.Mapping, not Dict
- Get the function to work for all key and value types (generic function)

Dict[...] and Tuple[...]

```
from typing import Dict, Iterator, Tuple, TypeVar

KT = TypeVar('KT')
VT = TypeVar('VT')

def reverse_dict(d: Dict[KT, VT]) -> Iterator[Tuple[VT, KT]]:
    return ((value, key) for key, value in d.items())
```

Type Inference

```
squares = []

for i in range(5):
    squares.append(i * i)  # mypy infers type List[int] for squares

print(', '.join(squares))  # error: incompatible List[int]; expected Iterable[str]
```

Variable Annotations

```
class Order:
    def __init__(self) -> None:
        self._items = []  # "Need type annotation for variable"

    def add_item(self, item: str) -> None:
        self._items.append(item)
```

Variable Annotations

```
from typing import List

class Order:
    def __init__(self) -> None:
        self._items = []  # type: List[str]

    def add_item(self, item: str) -> None:
        self._items.append(item)
```

Type Checking and Library Modules

```
os.fsdecode('/dir/file.ext')  # Argument 1 to "fsdecode" has incompatible
                              # type "str"; expected "bytes"
os.fsdecode(‘/path’) + 1  # Unsupported operand types for + ("str" and "int")
```

- ‘os.fsdecode’ is a standard library function
- How does mypy know the type signature of ‘fsdecode’?

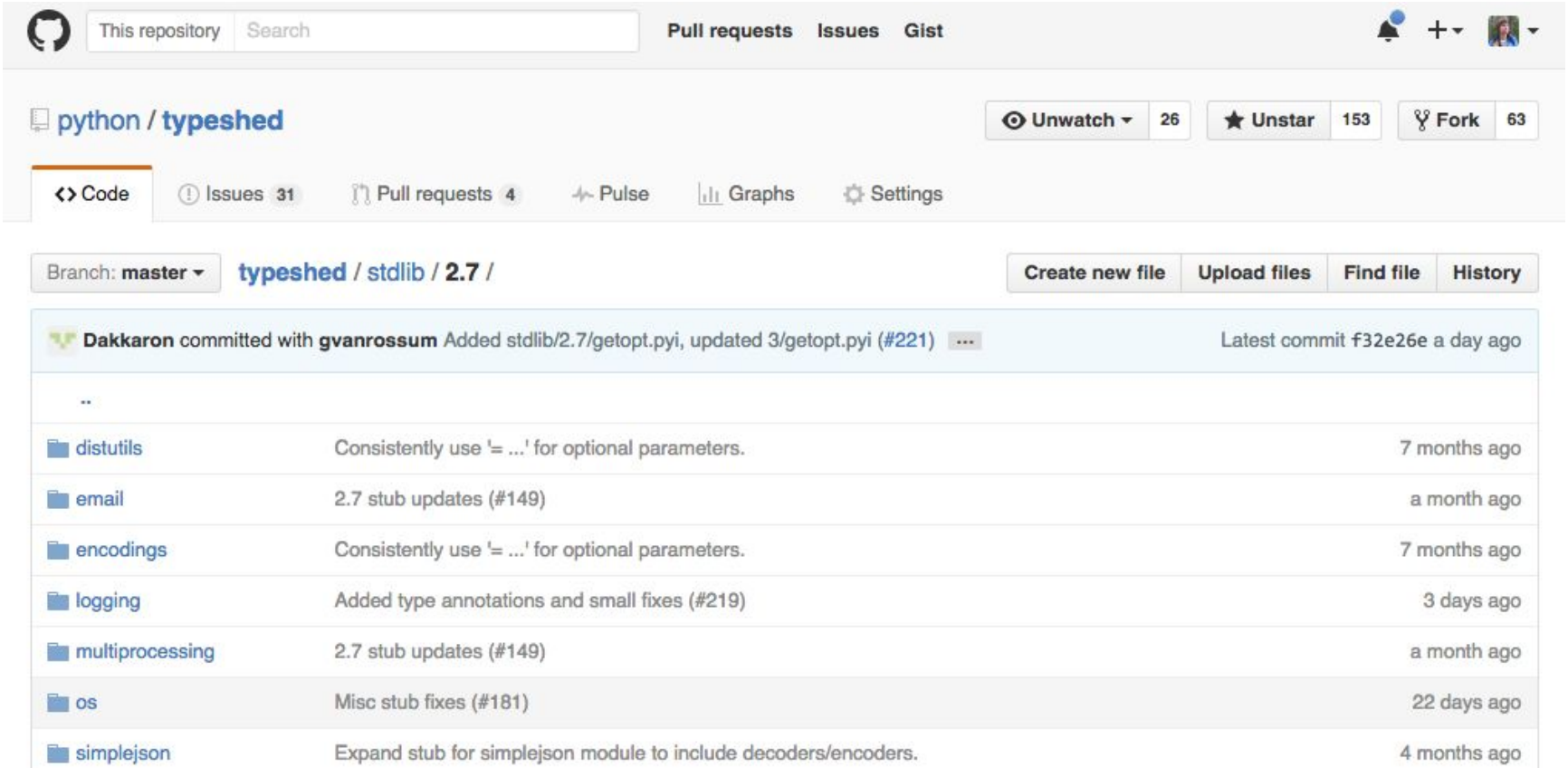
Stub Files

- A stub file has the skeleton of module structure + types
- Empty function bodies (“…”)
- Extract from os/__init__.pyi:

```
…
def fsencode(filename: str) -> bytes: ...
def fsdecode(filename: bytes) -> str: …      # Literal ‘...’
    ... and so on
```

- .pyi extension, but Python syntax
- For stdlib and 3rd party modules and C extensions

Typeshed: Stub Repository (stdlib and 3rd party)



The Any Type

```
def call_sorted(f: Any, *args: Any) -> Any:
    return f(*sorted(args))
```

Callable Types

```
def call_twice(cb: Callable[[int], None]) -> None:
    cb(1)
    cb(2)
```

Optional Types

```
def path_if_exists(path: str) -> Optional[str]:
    if os.path.exists(path):
        return path
    else:
        return None
```

Union Types

```
def as_bool(x: Union[str, int]) -> bool:
    if isinstance(x, str):
        return x.lower() in (‘true’, ‘yes’)        # Here x is str
    else:
        return x != 0                              # Here x is int
```

Live Demo

Conclusion

The Future of Mypy (short term)

None Checking

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "banks.py", line 31, in print_history
    print("Transferred ${} to {}.", account.name, amount)
AttributeError: 'NoneType' object has no attribute 'name'
```

Performance

The Future of Mypy (long term)

Editor Integration

API for:

- type-aware autocomplete
- inline errors
- type information
- advanced navigation (jump to definition, find occurrences, etc)
- automated refactoring (rename, add arguments, etc)

Plugin System

- add additional type system features
- SQLAlchemy, protobufs, Django, etc

Automated Annotations

- type inference
- runtime type inspection

Try It Out!

Try It!

- Install Python 3 (any version >= 3.3)
- pip3 install **mypy-lang**
  - or python3 -m pip install mypy-lang
- mypy program.py
  - mypy --py2 program.py # If you have Python 2 code
- Remember to put in annotations!
- See you at the sprints for mypy clinics

Q & A

Handy References

- mypy-lang.org (home page)
- mypy.readthedocs.io (docs, tutorial)
- github.com/python/mypy (source code)

- PEP 484 (specification)
- PEP 483 (theory behind it)
- github.com/python/typing (typing.py module)

- github.com/python/typeshed (stubs repository)