Gradual Typing for Python 3 (AKA PEP 484)

Eli Gur (Using GvR slides)

eli@eligur.com

PyCon IL, May 3rd, 2016

Outline

- Overview, motivation
- Syntax
- Discussion, history
- Conclusion?
- Q&A

Why a type checker

- Find bugs sooner
  - in annotated code
- The larger your project the more you need it
  - annotations help spelunking code
- Large teams are already running static analysis
  - Google, Dropbox building their own
  - also products like Semmle

Why type hints

- Help the type checker
  - in dynamic Python the flow of objects is hard to follow
- Serve as (additional) documentation
  - replace existing docstring conventions
- Help IDEs
  - improve suggestions
  - improve interactive code checks

Why oh why

- Python is dynamically typed and we like it that way!
- Yes, and…
  - large projects are already using static analysis tools
    - but current static checkers are often stumped by dynamic typing
  - it's still optional!
  - in fact, in PY3.5 it's provisional (PEP 411)
    - no code will break

Overview

- Static type checker a separate program
  - like a linter; developer chooses whether to use it
- Function annotations for type hints
  - in your code; only used by the type checker
- Stub files to annotate code you cannot change
  - dummy declarations seen only by the type checker

Why stub files

- C extensions (stdlib or otherwise)
- 3rd party packages you can't update
- Legacy code you don't want to change
- PY2 compatibility
- There's no time to annotate the world
  - new stub files can be released separately

Type hints outline

- Gradual typing basics
- The typing.py module
- Annotations
- Generics
- Pragmatics

Gradual typing basics

- Type hints for some code

```
def greeting(name: str) -> str:
    return "Hello, {}.".format(name)
```

- No type hints for other code

```
def greet(name):
    print(greeting(name))
```

- Something useful happens where they meet

Gradual typing principle

- Annotated code must conform to the type hints
- Un-annotated (dynamic) code always checks OK
- Absence of type hint === type hint of *Any*
  - *# does not complain about use of Any as a str*

```
def greet(name: Any) -> Any:
    print(greeting(name))
```

The magic of Any

- Any is at the top *and* the bottom of the class tree
- Like *object*, *isinstance(x, Any)* is always true
  - and so is *issubclass(C, Any)*
- Unlike *object*, *issubclass(Any, C)* is also true
  - however, in this case *issubclass()* is not transitive!
    - otherwise every class would be a subclass of every other class
- Technically, should say "is consistent with"

Gradual typing according to Siek

- *T1* is consistent with *T2*
  - value of type *T1* can be assigned to variable of type *T2*
  - not symmetric or transitive!
- Mostly follows subclassing: *issubclass(T1, T2)*
- Differs for special type *Any*
  - *Any* is consistent with *T*, *T* is consistent with *Any*; for all *T*
- Jeremy Siek (Indiana U.), "What is Gradual Typing" blog post

The typing.py module

- The only concrete part of the proposal!
  - Python 3.5 will perform no type checking
  - No new syntax
  - No changes to other stdlib modules
  - typing.py is backwards compatible with Python 3.2–3.4
- Import magic objects from typing.py
  - *from typing import Any, Union, Dict, List, …*

Type hints without magic

- Use built-in or your own classes as type hints:
  - *class Chart:*

```
def setlabel(self, x: float, y: float, name: str)
    -> bool: ...
def getnearest (self, x: float, y: float)
    -> str: ...
def make_label(c: Chart, a: str) -> bool:
    return c.setlabel(a)
def get_labels(c: Chart, points: list) -> list:
    return [c.getnearest(x,y) for x,y in points]
```

Sprinkle a bit of magic…

- Signature of *get_labels()* is imprecise: lists of what?

```
def get_labels(c: Chart, points: list) -> list:
    return [c.getnearest(x,y) for x,y in points]
```

- The typing package is here to help

```
from typing import List, Tuple
def get_labels(c: Chart,
               pts: List[Tuple[float, float]])
        -> List[str]:
    return [c.getnearest(x, y) for x, y in pts]
```

Sprinkle a bit of magic…

- Or, using ABCs:

```
from typing import List, Tuple, Iterable
def get_labels(c: Chart,
               pts: Iterable[Tuple[float, float]])
        -> List[str]:
    return [c.getnearest(x, y) for x, y in pts]
```

Wait, what?!

- *Iterable[Tuple[float, float]]; List[str]*
  - *typing.Iterable* is almost *collections.abc.Iterable*
    - an ABC (abstract base class) defining iterable behavior: *__iter__()*
  - *typing.List* resembles *builtins.list*
  - *typing.Tuple* somewhat resembles *builtins.tuple*
    - *Tuple* is a bit special (more a struct than a sequence)
- In a better world we could write *list[str]* etc.

Types vs. classes

- Types are for the checker, classes for run-time
  - types are (almost) only used in function annotations
- A class is a type:

```
def new_chart(name: str) -> Chart: …
```

- Some types aren't classes (e.g. *Any*, *Union*)
- This is super subtle
  - And maybe not the best terminology

Meanwhile, back on planet Earth…

- Things you can use as type hints:
  - classes: *object, int, float, Chart, …*
  - generic types: *List[int], Dict[str, int], Iterable[int], …*
    - both (pseudo-)concrete and abstract generic types exist
  - magic primitives: *Any, Union[…], Tuple[…], Callable[…], …*
  - DIY generic types
    - will show later

Union, Optional

- Value may be one of several types

```
def double(a: Union[int, float, str]) -> ...:
    return a + a
```

- *Optional[int]* means "either *int* or *None*"

```
Union[int, type(None)]
```

  - Also

```
Union[int, None]
```

Tuple

`Tuple[int, int, str]`

- e.g. *(0, 42, 'hello')*

`Tuple[float, ...]` `# literal ellipsis`

- immutable sequence of *float*

Callable

```
Callable[[float, float, str], bool]
```

- function taking *(float, float, str)* returning *bool*

```
Callable[..., float] # literal ellipsis
```

- function with unrestricted arguments returning *float*

DIY generic types

- Make your own type constructors:

```
from typing import TypeVar, Generic
T = TypeVar('T')
class Chart(Generic[T]):
    def setlabel(self, x, y, name: T): …
    def getnearest (self, x: float, y: float)
        -> T: …
```

- Now you can use *Chart[str], Chart[bytes]* etc.

DIY generic functions

- Same idea:

```
T = TypeVar('T')
def get_labels (c: C[T], xys: List[Tuple[float,
float]]) -> List[T]:
    return [c.narf(x, y) for x, y in xys]
```

AnyStr (1)

- Consider a function:

```
def split1(line: str, sep: str = None)
        -> List[str]:
    return line.split(sep, 1)
```

- It works for bytes too, so try to spec it like this:

```
AnyStr = Union[str, bytes]     # type alias
def split1(line: AnyStr, sep: AnyStr = None)
        -> List[AnyStr]:
    return line.split(sep, 1)
```

AnyStr (2)

- That's too liberal:

```
>>> split1(b'x y z', ' ')
TypeError: a bytes-like object is required, not
'str'
```

- A type checker should notice, and mypy does:

```
z.py, line 4: Argument 1 to "split" of "str" has incompatible type
"Union[str, bytes]"; expected "str"
z.py, line 4: Argument 1 to "split" of "bytes" has incompatible type
"Union[str, bytes]"; expected "Union[bytes, bytearray]"
z.py, line 4: Incompatible return value type: expected
builtins.list[Union[builtins.str, builtins.bytes]], got
Union[builtins.list[builtins.str], builtins.list[builtins.bytes]]
```

AnyStr (3)

- Use a constrained type variable with instead:

```
from typing import TypeVar
AnyStr = TypeVar('AnyStr', str, bytes)
def split1(line: AnyStr, sep: AnyStr = None)
        -> List[AnyStr]:
    return line.split(sep, 1)
split1(b'x y z', ' ')
```

```
z.py, line 5: Type argument 1 of "split1" has incompatible
value "object"
```

AnyStr (4)

- AnyStr is so useful it's actually built into typing:

```
from typing import AnyStr
```

- Note the TypeVar arguments:

```
AnyStr = TypeVar('AnyStr', str, bytes)
```

  - This requires the actual type to be one of the given types
  - This is called a constraint
  - The details concern mostly academics :-)

Pragmatics

- Forward references: use string quotes

```
class Node:
    def set_left(self, n: 'Node'): …
```

- Variable annotations: *# type: <type>* comments

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

- Cast function to relax the checker

```
y = cast(int, x)
```

Stub files

- *foo.pyi* is a stub for *foo.py* (or C extension *foo*)
  - Type checker prefers stubs over "real" modules
- Contains dummy classes, methods and functions
  - Method/function bodies are ignored

@overload

- Fake multiple dispatch, only allowed in stub files
  - mostly useful for polymorphic built-ins
  - typically constrained type variables are better
- Example from class stub for bytes:

```
@overload
def __getitem__(self, i: int) -> int: pass
@overload
def __getitem__(self, s: slice) -> bytes: pass
```

Disabling type checks

- If you use PEP 3107 annotations for something else
  - only if you (or your users) want to use a checker
- Disable by class or method: *@no_type_check*
- Disable for the whole file: *# type: ignore*
- Or use a stub file

History

- 1998-2002: Optional Static Typing talk (SIG)
  - Type hint annotations
- 2004–2005: GvR blogs at Artima
  - Generic functions
  - Generic types
- 2006: PEP 3107
  - function annotations
  - syntax with no semantics

Recent history

- 2013: GvR Met Jukka Lehtosalo (mypy) at PyCon
  - convinced him to make mypy compatible with PEP 3107
  - using *List[T]* etc.
- 2014: Bob Ippolito recommends mypy at EuroPython
  - "What Python can learn from Haskell"
- 2015: PEP 484: type hints, gradual typing
  - Jukka Lehtosalo, Łukasz Langa, GvR

Why a PEP

- "Let them use mypy if they want to"
- A standard notation helps everyone
- E.g. PyCharm currently has its own set of stubs; Google has another set…
- Waiting helps nobody
- Type systems are fun :-)

Other uses of annotations

- What if you're using PEP 3107 for something else?
  - *def main(verbose: '-v' = False, output: '-o' = sys.stdout): …*
- Nothing will break in Python 3.5
  - just don't run a type checker
- Or you can shut up the type checker
  - *@no_type_check*
    def main(verbose: '-v' = False, output: '-o' = sys.stdout): …

Those pesky academics might say…

- We need more features
  - e.g. "F-bounded polymorphism" (Java)
- The type system is not "sound"
  - code may pass the type check but still break
- The type system is not "complete"
  - type of some functions cannot be expressed

Links

- types-sig
  - python.org/community/sigs/retired/types-sig/
- Static typing talk 2000
  - python.org/~guido/static-typing/
- Artima blog post
  - artima.com/weblogs/viewpost.jsp?thread=85551
- Jeremy Siek's What is Gradual Typing blog post
  - wphomes.soic.indiana.edu/jsiek/what-is-gradual-typing/
- Guido Van Rossum - Gradual Typing for Python 3
  - www.dropbox.com/s/s82883a1ogk2omx/BayPiggies2015GradualTyping.pptx