PEP 484 and Python's (future) type annotation system(s)

Who am I?

A Director of the Python Software Foundation; Chair of PSF's Outreach & Education Committee; Chair of PSF's Trademarks Committee.

I wrote the IBM developerWorks column *Charming Python*, the Addison-Wesley book *Text Processing in Python*, and various related articles.

For the last 7 years I consulted for a research lab, D. E. Shaw Research, who have built the world's fastest supercomputer for doing molecular dynamics.

A plug for the Python Software Foundation

This talk is technical, not my administrative one that I give wearing my PSF hat. However, I would like all of you to be aware of our new bylaws, and expanded membership model. Please join the Foundation now, we'd love to have you:

```
https://www.python.org/psf/membership/
```

Why this talk?

part 1

PEP 484 will add a standard language for type annotations to Python 3.5+.

The first tool to utilize these annotations will probably be the existing static typing tool `mypy` on which the PEP is largely based.

Other code analysis tools, IDEs, documentation tools, linters, and other software will be able to rely on consistent use of annotations in future Python versions.

Why this talk

part 2

Annotations were introduced into Python 3.0 with PEP 3107. Way back in 2006, when this happened, type annotations were the first- listed suggested use for annotations.

However, only with PEP 484, will type hints be blessed as the official use for annotations. GvR has decided that no other use has gained sufficient use to warrant such endorsement.

A toy introduction

part 1

A number of tools have used annotations for type checking or hinting. I myself provided a rather simple recipe for doing so in 2013, at ActiveState's Python Cookbook site:

> *Type checking using Python 3.x annotations*
>
> `http://bit.ly/1zdZu50`

My goal there was to do the most obvious thing I could think of. We will see in a few moments.

A toy introduction

part 2

The code for my recipe is quite short, but let's take a look at its use first:

```
from math import sqrt
from simple_typing import typecheck

@typecheck
def radius(c: complex) -> float:
    return sqrt(c.real**2 + c.imag**2)

for c in (3-4j, 5, 0j-5, "five"):
  try:
    print("%s has radius %s" % (c, radius(c)))
  except AssertionError:
    print("Incorrect args in radius(%r)" % c)
```

A toy introduction

part 3

Running this script catches the argument type errors as we would hope:

```
% python3 radius.py
(3-4j) has radius 5.0
Incorrect args in radius(5)
(-5+0j) has radius 5.0
Incorrect args in radius('five')
```

A different function could demonstrate a type error in the return value, but `radius()` always returns a `float` if it runs at all.

A toy introduction

part 4

Notice that the `@typecheck` decorator is not merely catching arguments that would raise exceptions:

```
>>> from math import sqrt
>>> def radius(c: complex) -> float:
...     return sqrt(c.real**2 + c.imag**2)
>>> radius(5)  # An int has .real and .imag
5.0
>>> radius("five")  # Not AssertionError
AttributeError: 'str' object has no attribute 'real'
```

A toy introduction

part 5

```
def typecheck(f):
    @functools.wraps(f)
    def decorated(*args, **kws):
        for i, nm in enumerate(f.__code__.co_varnames):
            argtype = f.__annotations__.get(nm)
            # Check if annotation exists & is a type
            if isinstance(argtyp, type):
                # First len(args) positional, then kws
                if i < len(args):
                    assert isinstance(args[i], argtype)
                elif nm in kws:
                    assert isinstance(kws[nm], argtype)
        result = f(*args, **kws)
        returntype = f.__annotations__.get('return')
        if isinstance(returntype, type):
            assert isinstance(result, returntype)
        return result
    return decorated
```

Mypy passingly

part 1

The type checking my simple decorator performs could easily be extended to cover parameterized types in various ways.

However, what it cannot do is perform static type analysis without actually running the code being checked. The fact Mypy does this is GvR's main inspiration for PEP 484.

Mypy passingly

part 2

The syntax that Jukka Lehtosalo's Mypy uses is almost a superset of my toy decorator, and both are almost the same as what is specified in PEP 484 currently.

The PEP is not final yet, and some details might yet change. Presumably, where names and details differ between Mypy and PEP 484, Mypy will be updated once the PEP is final.

Mypy passingly

part 3

Mypy and PEP 484 introduce a concept of "is-consistent-with" which is similar to "is-subclass-of" but differs in a few cases that this talk will get to.

In particular, `complex` is not a type recognized by Mypy (a deficiency to my mind), and `int` *is-consistent-with* `float` despite these facts:

```
issubclass(int, float) == False
isinstance(1, float) == False
```

Mypy passingly

part 4

In other words:

```
% cat int_float.py
import typing
def asInt(a: float) -> int:
    return int(a)

print(asInt(1.0))
print(asInt(1))
print(asInt("1"))

% mypy int_float.py
int_float.py, line 7: Argument 1 to "asInt" has
incompatible type "str"; expected "float"
```

Mypy passingly

part 5

In contrast, using the `@typecheck` decorator would complain about the `int` argument, and the program will run without error in `python3` itself.

```
def asInt(a: float) -> int:
    return int(a)

print(asInt(1.0))
print(asInt(1))
print(asInt("1"))
```

Mypy passingly

part 6

PEP 484 is primarily about static analysis of types. E.g., I could use the `@typecheck` decorator on this code, but it would take a long time to fail:

```
% cat many_loops.py
import typing
def int_if_small(a: int) -> int:
    return a if a < 1e9 else float(a)
for i in range(int(1e10)): int_if_small(i)

% mypy many_loops.py
many_loops.py: In function "int_if_small":
many_loops.py, line 4: Incompatible return value
type: expected builtins.int, got builtins.float
```

A new toy

part 1

Just for fun, let's adjust the prior example for Mypy. Remember that `complex` is not a type Mypy recognizes, so we need to use a custom subclass:

```
from math import sqrt
import typing

class Complex(complex):
    def __init__(self, real: float,
                       imag: float=0) -> None:
        complex.__init__(self, real, imag)

def radius(c: Complex) -> float:
    return sqrt(c.real**2 + c.imag**2)
```

A new toy

part 2

We now get static checking, for example:

```
c1 = Complex(5,0)          # happy
c2 = Complex('five')       # static violation
radius(Complex(5,0))       # happy
radius(Complex(5))         # happy
radius(5,0)                # static violation
radius(5+0j)               # static violation
radius("five")             # static violation
```

A new toy

part 3

We now get static checking, for example:

```
% mypy radius2.py
radius2.py, line 12: Argument 1 to "Complex" has
 incompatible type "str"; expected "float"
radius2.py, line 15: Too many arguments for
 "radius"
radius2.py, line 15: Argument 1 to "radius" has
 incompatible type "int"; expected "Complex"
radius2.py, line 16: Argument 1 to "radius" has
 incompatible type "complex"; expected "Complex"
radius2.py, line 17: Argument 1 to "radius" has
 incompatible type "str"; expected "Complex"
```

Gradual typing

part 1

An important concept in PEP 484 is Gradual Typing. Jeremy Siek writes about this concept on his blog at:

`http://wphomes.soic.indiana.edu/jsiek/`

The concept is discussed in the informational PEP 483. *Is-consistent-with*, unlike *is-subclass-of*, is not transitive when the new type **`Any`** is involved.

Assigning x to y works if and only if `type(x)` *is-consistent-with* `type(y)`.

Gradual typing

part 2

The relationship *is-consistent-with* is defined by exactly three rules:

- A type `t1` *is-consistent-with* `t2` if we have the relationship `issubclass(t1,t2)`; but not the reverse relationship.
- `Any` *is-consistent-with* every type.
- Every type is a subclass of `Any` (and therefore consistent with it).

Gradual typing

part 3

Mypy and other tools might use type inference along with explicit type annotations. In other words, assignment can constitute a declaration (not required by PEP 484, but not precluded in tools):

```
% cat type_inference.py
import typing
i = 1          # Infer type(i) == int
i = "one"      # Infer type(i) == str

% mypy type_inference.py
type_inference.py, line 3: Incompatible types in
assignment (expression has type "str", variable
has type "int")
```

Gradual typing

part 4

However, PEP 484 also allows explicit declarations:

```
% cat type_declaration_1.py
from typing import Any
a1 = 1        # type: str
a1 = "one"    # consistent with declared type

a2 = 1        # type: Any
a2 = "one"    # consistent with declared type

% mypy type_declaration_1.py
type_declaration_1.py, line 2: Incompatible
types in assignment (expression has type "int",
variable has type "str")
```

Gradual typing

part 5

I find comment type declarations ugly. But also we need a way to type but not assign value to a variable:

```
% cat type_declaration_2.py
from typing import Undefined, Any
a3 = Undefined(Any)  # no instance value given
a3 = 1               # consistent w/ declared
a3 = "one"           # consistent w/ declared
a4 = Undefined(str)  # no instance value given
a4 = 1               # violates declared type

% mypy type_declaration_2.py
type_declaration_2.py, line 6: Incompatible
types in assignment (expression has type "int",
variable has type "str")
```

Parameterized types

part 1

The examples so far are mostly of built-in types. What gets more interesting is the parameterized types in the new `typing.py` module. For example:

```
from typing import Union, Sequence
class Employee(object): pass
class Manager(Employee): pass

def fun(e: Union[Employee, Sequence[Employee]]):
    if isinstance(e, Employee):
        e = [e]
    for employee in e:
        # Do stuff
```

Parameterized types

part 2

`typing.py` provides some interesting classes, making heavy use of clever metaclasses. The basic idea is to overload the `.__getitem__` syntax to return customized class objects (below subject to change).

```
>>> from pep484_typing import *
>>> Union
pep484_typing.Union
>>> Union[Employee, Sequence[Employee]]
pep484_typing.Union
>>> E = Union[Employee, Sequence[Employee]]
>>> E.__union_params__
(__main__.Employee, typing.Sequence)
```

Parameterized types

part 3

Let's not dwell on the details, which GvR may change before release. Just remember that a parameterized `Union` (and other classes in `typing.py`) is itself still a `Union` class with some extra details attached to it:

```
>>> Union
pep484_typing.Union
>>> Union[Employee, Sequence[Employee]]
pep484_typing.Union
>>> E = Union[Employee, Sequence[Employee]]
>>> E.__union_params__
(__main__.Employee, typing.Sequence)
```

Parameterized types

part 4

The classes `Union` and `Sequence` do pretty much what you'd expect from set theory and duck typing:

```
class Employee(object): pass
class Manager(Employee): pass
class CEO(Manager): pass
class Consultant(object): pass
def fun(e: Union[Employee, Sequence[Employee]]):
    pass
dick, jane, carol = Employee(), Manager(), CEO()
arty = Consultant()
fun(carol)                    # happy
fun(arty)                     # incompatible type
fun([dick, jane, carol]) # happy
fun([dick, jane, arty])  # incompatible type
```

Parameterized types

part 5

This is not final for Python 3.5, but it's interesting what Mypy currently reports about the incompatibilities:

```
% mypy parameterized_types.py
parameterized_types.py, line 14: Argument 1 to
"fun" has incompatible type "Consultant";
expected "Union[Employee, Sequence[Employee]]"
parameterized_types.py, line 16: Argument 1 to
"fun" has incompatible type List[object];
expected "Union[Employee, Sequence[Employee]]"
```

Parameterized types

part 6

That is:

```
% head -14 parameterized_types.py | tail -1
fun(arty)                       # incompatible type

parameterized_types.py, line 14: Argument 1 to
"fun" has incompatible type "Consultant";
expected "Union[Employee, Sequence[Employee]]"
```

Clearly a `Consultant` is not an `Employee`, nor a sequence of any kind. So that incompatibility is straightforward.

Parameterized types

part 7

Here is where the incompatibility is more subtle:

```
% head -16 parameterized_types.py | tail -1
fun([dick, jane, arty])  # incompatible type

parameterized_types.py, line 16: Argument 1 to
"fun" has incompatible type List[object];
expected "Union[Employee, Sequence[Employee]]"
```

Since the list we pass to `fun()` contains objects of types `Employee`, `Manager`, and `Consultant`, the best Mypy can do in inferring the type of elements is to pick their closest common superclass—in this case `object` itself. And object is not *consistent-with* `Employee` (albeit, `Any` would be).

Classes in typing.py

part 1

`Union` has been addressed in fair detail and `Sequence` was touched on. Let's take a look at other custom classes that will be added in the new `typing.py` standard library module in Python 3.5.

All of the abstract base classes in `collections.abc` are importable from `typing.py`, with `Set` renamed as `AbstractSet` (other names the same).

Classes in typing.py

part 2

An interesting class included in `typing.py` is `Callable`. This is useful especially for giving signatures to callback functions. Some examples:

```
from typing import Any, AnyArgs, Callable
def feeder(get_next: Callable[[], Item]):
    ...
def async_query(on_success: Callable[[int],None],
    on_error: Callable[[int,Exception],None]):
    ...
def partial(func: Callable[AnyArgs,Any], *args):
    ...
```

Classes in typing.py

part 3

Collections in `collections.abc` have been extended to be parameterizable with types of the items within them. For example.

```
from typing import Mapping, Set
def notify_by_email(employees: Set[Employee],
                    overrides: Mapping[str, str]):
    ...
```

Classes in typing.py

part 4

There will be interesting support for generics using the class `TypeVar`:

```
from typing import Sequence, TypeVar
T = TypeVar('T')      # Could also restrict type
                      # by using a second argument
def first(l: Sequence[T]) -> T:
    return l[0]       # The type returned must be
                      # consistent with the Sequence
                      # parameter
```

PEP 484 Miscellany

part 1

To ease the transition toward normative use of annotations for type information, a couple transitional measures are adopted in PEP 484.

The decorator `@no_type_checks` will instruct tools to skip type checking on annotated functions which currently use annotations for other purposes. This decorator will probably have no runtime behavior since it is for the benefit of static type checking tools.

PEP 484 Miscellany

part 2

An optional documentation "type hint" of, e.g.:

```
a = 1 # type: ignore
```

Will be available to alert type checkers to ignore checking of variables defined on annotated lines.

I am unclear where this could ever differ in behavior from specifying type `Any` to a variable (other than, perhaps, running a static check a few milliseconds more quickly because a variable is simply unexamined rather than being compatible with everything).

PEP 484 Miscellany

part 3

To me, using a comment to specify type information feels unpythonic and I am little fond of that approach (albeit I understand that doing so manages not to change any runtime semantics or syntax of Python 3.4).

The PEP itself comments "*a syntax for typing variables may be provided in a future Python version.*"

PEP 484 Miscellany

part 4

The class `Undefined` is useful to pre-declare a type, in either of two ways (or a third "non-declaration"):

```
>>> from pep484_typing import Union
>>> x = Undefined       # type: List[Employee]
>>> y = Undefined(int) # Specialize the class
>>> z = Undefined       # merely an intent to use
>>> z.__type__
<member '__type__' of 'Undefined' objects>
>>> y.__type__
int
```

PEP 484 Miscellany

part 5

However, a variable being `Undefined` isn't the same as it being undefined. A standard helper function to cover both cases might be useful, e.g.:

```
def is_defined(name):
    try:
        val = eval(name)
        return (type(val) != type or
                not issubclass(val, Undefined))
    except NameError:
        return False
```

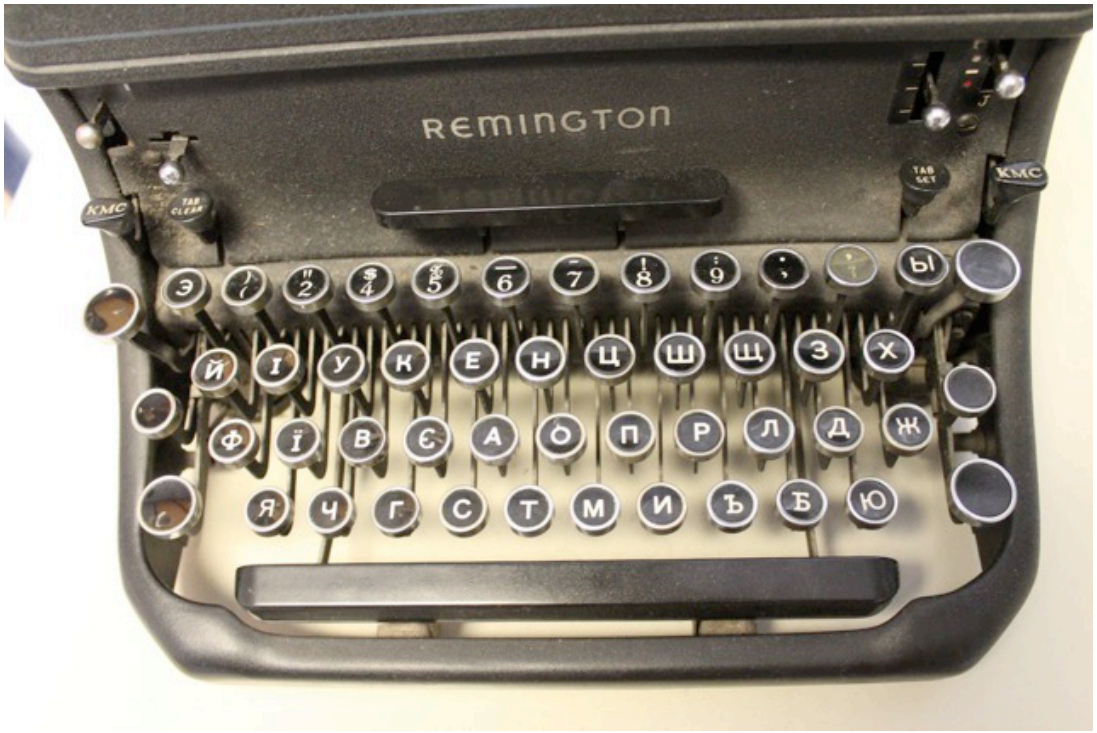
PEP 484 Miscellany

part 6

I suspect GvR will not agree with me, but I'd like a means to enforce runtime type checking that matched the semantics for static checking (but exceptions would be raised only when a violation actually occurred, not every time it is statically possible).

This might be either an enhanced `@typecheck` decorator for specific functions, or a switch to `python` to do so globally, e.g. `--enforce-types`.

Wrap-up / Questions?



If we have time, I'd love feedback during the question period (or catch me in the hallway).