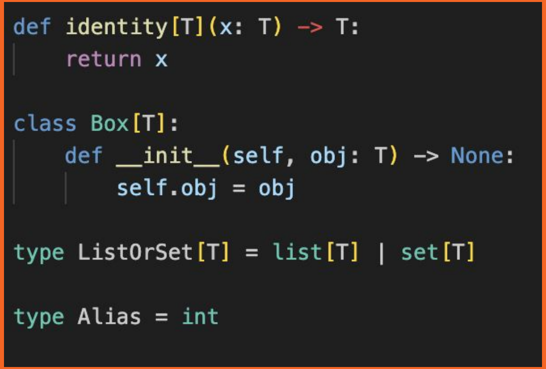
PEP 695

or,

How typing syntax led to a scoping rabbit hole



```python
def identity[T](x: T) -> T:
    return x

class Box[T]:
    def __init__(self, obj: T) -> None:
        self.obj = obj

type ListOrSet[T] = list[T] | set[T]

type Alias = int
```

Jelle Zijlstra
Quora

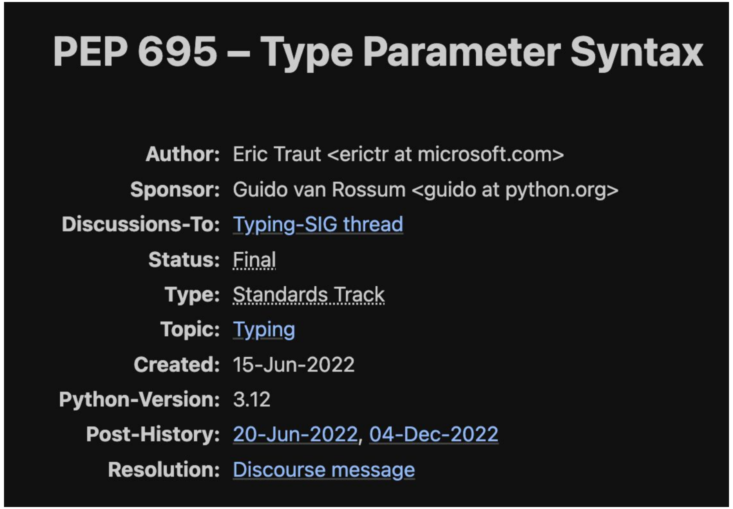
Who am I?

- Jelle Zijlstra
- Software engineer at Quora
- CPython core developer
- Typing Council member
- Most importantly…

Who am I?

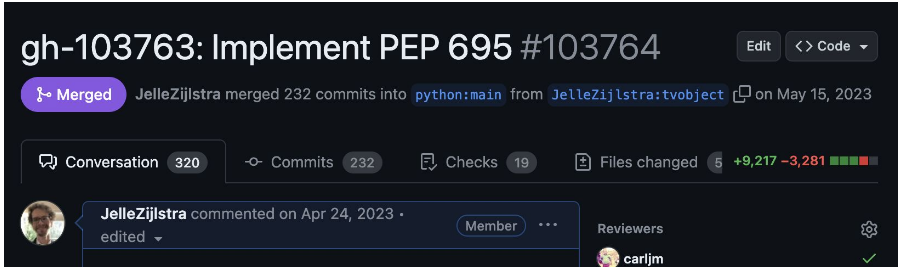
- Jelle Zijlstra
- Software engineer at Quora
- CPython core developer
- Typing Council member
- Most importantly…
- Wrote the runtime implementation of PEP 695

PEP 695



PEP 695 – Type Parameter Syntax

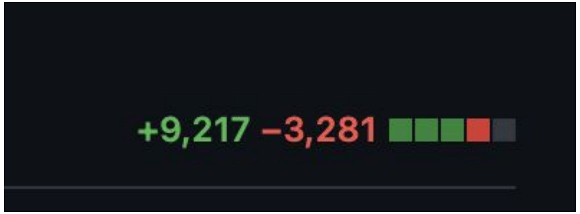
Author:	Eric Traut <erictr at microsoft.com>
Sponsor:	Guido van Rossum <guido at python.org>
Discussions-To:	Typing-SIG thread
Status:	Final
Type:	Standards Track
Topic:	Typing
Created:	15-Jun-2022
Python-Version:	3.12
Post-History:	20-Jun-2022, 04-Dec-2022
Resolution:	Discourse message

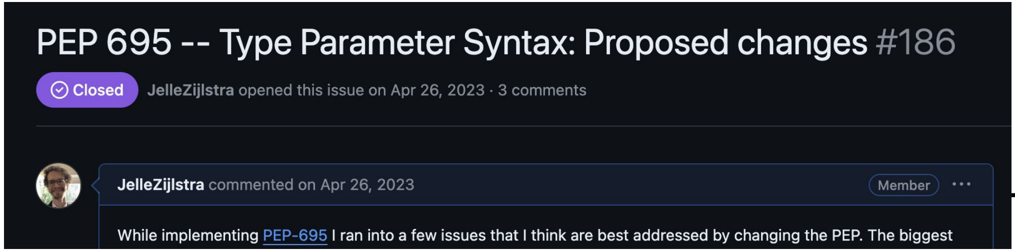


gh-103763: Implement PEP 695 #103764

Merged JelleZijlstra merged 232 commits into `python:main` from `JelleZijlstra:tvobject` on May 15, 2023

JelleZijlstra commented on Apr 24, 2023 • edited





PEP 695 -- Type Parameter Syntax: Proposed changes #186

Closed JelleZijlstra opened this issue on Apr 26, 2023 · 3 comments

JelleZijlstra commented on Apr 26, 2023

While implementing PEP-695 I ran into a few issues that I think are best addressed by changing the PEP. The biggest

Generics

Let’s talk about (a simplified version of) the filter builtin:

```
def filter(pred, it):
    return (elt for elt in it if pred(elt))
```

How would we add type annotations?

Generics

```
def filter(
   pred: Callable[[?], bool],
   it: Iterable[?],
) -> Iterable[?]:
   return (elt for elt in it if pred(elt))
```

Generics

```python
T = TypeVar("T")
def filter(
    pred: Callable[[T], bool],
    it: Iterable[T],
) -> Iterable[T]:
    return (elt for elt in it if pred(elt))
```

Generic classes

```
T = TypeVar("T")
class list(Generic[T]):
    def append(self, elt: T, /) -> None: ...
    def __getitem__(self, i: int, /) -> T: ...
```

Generic type aliases

```
T = TypeVar("T")
PairList = list[tuple[T, T]]


def f(pairs: PairList[int]):
   for x, y in pairs:
       distance = sqrt(x*x + y*y)
```

Bounds

```
T = TypeVar("T", bound=Sized)
def longest(iter: Iterable[T]) -> T:
    return max(iter, key=len)
```

PEP 695: Syntax

```
def filter[T](
    pred: Callable[[T], bool], it: Iterable[T],
) -> Iterable[T]:
    return (elt for elt in it if pred(elt))
class list[T]:
    def append(self, elt: T, /) -> None: …
type PairList[T] = list[tuple[T, T]]
def longest[T: Sized](iter: Iterable[T]) -> T:
    return max(iter, key=len)
```

PEP 695: Function syntax

Before:

```python
T = TypeVar("T")
def identity(arg: T) -> T:
    return arg
```

After:

```python
def identity[T](arg: T) -> T:
    return arg
```

PEP 695: Class syntax

Before:

```
T = TypeVar("T")
class list(Generic[T]):
    def append(self, obj: T):
        ...
```

After:

```
class list[T]:
    def append(self, obj: T):
        ...
```

PEP 695: Type alias syntax

Before:

```
T = TypeVar("T")
ListOrSet: TypeAlias = (
    list[T] | set[T]
)
```

After:

```
type ListOrSet[T] = (
    list[T] | set[T]
)
```

Why?

It’s verbose

We have a trend:

- `typing.List[int]` -> `list[int]` (Python 3.9)
- `typing.Optional[str]` -> `str | None` (Python 3.10)
- `typing.Callable[[int], str]` -> `(int) -> str` (Python 3.11)

It’s verbose

We have a trend:

- `typing.List[int]` -> `list[int]` (Python 3.9)
- `typing.Optional[str]` -> `str | None` (Python 3.10)
- ~~`typing.Callable[[int], str]` -> `(int) -> str` (Python 3.11)~~
  - Oh no, that one got rejected

It’s verbose

We have a trend:

- `typing.List[int]` -> `list[int]` (Python 3.9)
- `typing.Optional[str]` -> `str | None` (Python 3.10)
- ~~`typing.Callable[[int], str]` -> `(int) -> str` (Python 3.11)~~
  - Oh no, that one got rejected
- `T = TypeVar("T"); def f(x: T): ...` -> `def f[T](x: T): ...` (Python 3.12)

Unclear scoping

```
T = TypeVar("T")
U = TypeVar("U")
def filter(
    pred: Callable[[T], bool], it: Iterable[T],
) -> Iterable[T]: …
def map(
    func: Callable[[T], U], it: Iterable[T],
) -> Iterable[U]: …
```

Unclear scoping: Classes

```python
T = TypeVar("T")
class list:
    def append(self, elt: T, /) -> None: …
    def __getitem__(self, i: int, /) -> T: …
```

Variance declarations

```
T_co = TypeVar("T_co", covariant=True)
class tuple(Generic[T_co]):
   def __getitem__(self, i: int, /) -> T: …
```

Forward declarations

```
NodeT = TypeVar("NodeT", bound="Node")

class Node:

    def copy(self: NodeT) -> NodeT: …
```

Forward declarations

```
JSON: TypeAlias = (
    list["JSON"] | dict[str, "JSON"] |
    str | int | float | bool | None
)
```

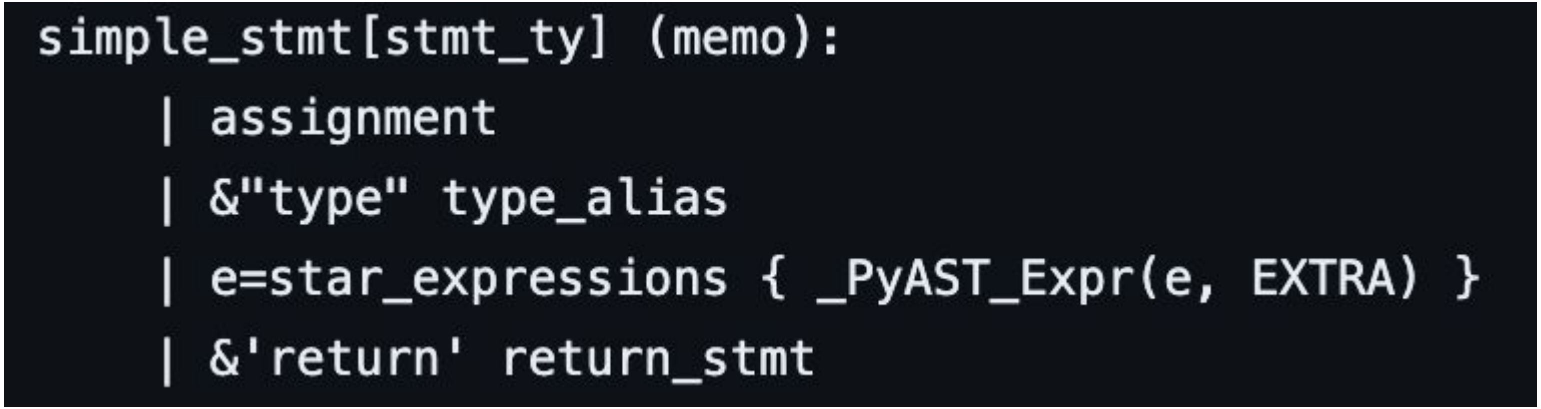
Implementation

The parser

```
type PairList[T] = list[tuple[T, T]]
```

The parser ❤️ soft keywords

```
type PairList[T] = list[tuple[T, T]]
```



```
simple_stmt[stmt_ty] (memo):
    | assignment
    | &"type" type_alias
    | e=star_expressions { _PyAST_Expr(e, EXTRA) }
    | &'return' return_stmt
```

Everything can be a type now

```
>>> type type[type: type] = type
>>> type.__type_params__
(type,)
>>> type.__type_params__[0].__bound__
type
>>> type.__value__
type
```

The symbol table: Requirements

```
T = 1

def f[T](x: T):  # Can use T in annotation
    local_variable: T  # Allowed

print(T)  # 1
f()  # OK, f is in scope
```

The symbol table: Solutions

- Overlays?
  - ❌
- Name mangling?
  - ❌
- Lambda lifting?
  - ✅

Lambda lifting

```
def func[T](arg: T): …
```

= (*)

```
def __generic_parameters_of_func():
    T = TypeVar("T")
    def func(arg: T): …
    func.__type_params__ = (T,)
    return func
func = __generic_parameters_of_func()
```

Bytecode

```
>>> dis.dis("type X[T] = int")
…
          LOAD_CONST               1 ('T')
          CALL_INTRINSIC_1         7 (INTRINSIC_TYPEVAR)
…
          CALL_INTRINSIC_1        11 (INTRINSIC_TYPEALIAS)
```

Moving to C

- TypeVar, Generic, etc. are now implemented in C
    - But it’s hard to tell the difference
- Some operations call into Python code

```
>>> class X[T]: pass
...
>>> X[int]
__main__.X[int]
>>> typing._generic_class_getitem = print
>>> X[int]
<class '__main__.X'> <class 'int'>
```

Lazy evaluation

```
type BinOp = Literal["+", "-"]

type LeftParen = Literal["("]

type RightParen = Literal[")"]

type SimpleExpr = int | Parenthesized

type Parenthesized = tuple[LeftParen, Expr, RightParen]

type Expr = SimpleExpr | tuple[SimpleExpr, BinOp, Expr]
```

Class scopes are weird

What does this do?

```
x = "global"
def f():
    x = "function"
    class Nested:
        print(x)
f()
```

How about this one?

```
x = "global"
def f():
    x = "function"
    class Nested:
        x = "class"
        print(x)
f()
```

OK, how about this?

```
x = "global"
def f():
    x = "function"
    class Nested:
        print(x)
        x = "class"
f()
```

And did you know you could do this?

```
x = "global"
def f():
    x = "function"
    class Nested:
        global x
        print(x)
f()
```

What makes class scopes different?

```
x = "global"
class Cls:
    x = "class"
    def method(self):
        print(x)
Cls().method()
```

We want this to work

```
class Outer:
    class Nested:
        pass
    type Alias = Nested
    def meth1[T: Nested](self): pass
    def meth2[T](self, arg: Nested): pass
```

How to implement it

- Symbol table: Mark scope as special
    - `ste_can_see_class_scope`
- Runtime: Give the scope access to the class dict
    - Always look in class first, then in global or enclosing scope
    - You never know what’s actually in the class dict

But what about this?

```
class Cls:
   T = "before"
   type Alias = T
Cls.T = "after"
print(Cls.Alias.__value__)
```

Implementation: __classdict__

```
class X:
    type A = __classdict__
    A_val = A.__value__
    type B = __classdict__
B_val = X.B.__value__
assert X.A_val != B_val
```

More bugs!

- What if you put a generator expression inside the base class of a generic class that is nested in a generic class?
  - 💥
  - But now you get a SyntaxError (in 3.12)
  - Fixed in 3.13
- `yield` or `await` in an annotation scope?
  - SyntaxError

What’s next?

Python 3.13: More annotation scopes

- TypeVar defaults (PEP 696)
- Lazy evaluation of annotations (PEP 649)

Python 3.13: More annotation scopes

- TypeVar defaults (PEP 696)
- ~~Lazy evaluation of annotations (PEP 649)~~
    - Slipped to 3.14

Someday?: Better implementation

- Less calling into Python code
- Reduce overhead of calling dummy functions
    - Like we do with list comprehensions (PEP 709)

What I skipped

- ParamSpec and TypeVarTuple
- Bounds vs. constraints
- Special case for nonlocal

See also:

- https://jellezijlstra.github.io/pep695
- https://github.com/python/cpython/pull/103764

Thank you