This is no longer the latest draft. Please see draft 2.

PEP: 9999
  <!-- TODO: Obtain PEP number -->
Title: TypeForm: Type Hint for a Type Expression
Author: David Foster <david at dafoster.net>
Sponsor: Jelle Zijlstra <jelle.zijlstra at gmail.com>
Discussions-To: Discourse thread
Status: Draft
Type: Standards Track
Content-Type: text/x-rst
  <!-- TODO: Convert from Markdown to RST -->
Created: 21-Dec-2020
Python-Version: 3.14
  <!-- NOTE: 3.13 feature freeze is 2024-05-07 -->
Post-History: 24-Jan-2021, 28-Jan-2021, 04-Feb-2021, 14-Feb-2021, 19-Apr-2024

Abstract
========

PEP 484 defines the notation `Type[C]` where `C` is a class, to refer to a class object that is a subtype of `C`. It explicitly does not allow `Type[C]` to refer to typing special forms such as the runtime object `Optional[str]` even if `C` is an unbound `TypeVar`. [^type-c] In cases where that restriction is unwanted, this PEP proposes a new notation `TypeForm[T]` where `T` is a type, to refer to a either a class object or some other typing special form that is a subtype of `T`, allowing any kind of type expression to be referenced.

This PEP makes no Python grammar changes. Correct usage of `TypeForm[]` is intended to be enforced only by static type checkers and need not be enforced by Python itself at runtime.

Motivation
==========

The introduction of `TypeForm` allows new kinds of metaprogramming functions that operate on typing special forms to be type-annotated and understood by typecheckers.

For example, here is a function that checks whether a value is assignable to a variable of a particular type, and if so returns the original value:

```
def trycast[T](form: TypeForm[T], value: object) -> Optional[T]: ...
```

The use of `TypeForm[]` and the type variable `T` enable the return type of this function to be influenced by a `form` value passed at runtime, which is quite powerful.

Here is another function that checks whether a value is assignable to a variable of a particular type, and if so returns True (as a special `TypeIs[]` bool [^TypeIsPep]):

```
def isassignable[T](value: object, form: TypeForm[T]) -> TypeIs[T]: ...
```

The use of `TypeForm[]` and `TypeIs[]` together enables typecheckers to narrow the return type appropriately depending on what form is passed in:

```
request_json: object = ...
if isassignable(request_json, Shape):
    assert_type(request_json, Shape)  # type is narrowed!
```

That `isassignable` function enables a kind of enhanced `isinstance` check which is useful for [checking whether a value decoded from JSON conforms to a particular structure] of nested `TypedDict`s, `List`s, `Optional`s, `Literal`s, and other types. This kind of check was alluded to in PEP 589[^typeddict-no-isinstance] but could not be implemented at the time without a notation similar to `TypeForm[]`.

Why can't `Type` be used?

One might think you could define the example functions above to take a `Type[T]` - which is syntax that already exists - rather than a `TypeForm[T]`. However if you were to do that then certain typing special forms like `Optional[str]` - which are not class objects and therefore not `type`s at runtime - would be rejected:

```
# uses a Type[T] parameter rather than a TypeForm[T]
def trycast_type[T](form: Type[T], value: object) -> Optional[T]: ...

trycast_type(str, 'hi')  # ok; str is a Type
trycast_type(Optional[str], 'hi')  # ERROR; Optional[str] is not a Type
trycast_type(Union[str, int], 'hi')  # ERROR; Union[str, int] is not a Type
trycast_type(MyTypedDict, dict(value='hi'))  # questionable; accepted by mypy 1.9.0
```

To solve that problem `Type` could be widened to include the additional values allowed by `TypeForm`. However doing so would lose `Type`'s current ability to spell a class object which always supports instantiation and `isinstance` checks, unlike arbitrary typing special forms. Therefore `TypeForm` is proposed as new notation instead.

For a longer explanation of why we don't just widen `Type[T]` to accept all typing special forms, see §"Widen Type[T] to support all typing special forms".

Common kinds of functions that would benefit from TypeForm

[A survey of various Python libraries] revealed a few kinds of commonly defined functions which would benefit from `TypeForm[]`:

* Assignability checkers:
    * Returns whether a value is assignable to a type-form.
      If so then also narrows the type of the value to match the type-form.
    * Pattern 1: `def isassignable[T](value: object, form: TypeForm[T]) -> TypeIs[T]`
    * Pattern 2: `def ismatch[T](value: object, form: TypeForm[T]) -> TypeGuard[T]`
    * Examples: beartype.is_bearable, trycast.isassignable, typeguard.check_type,
      xdsl.isa

* Converters:
    * If a value is assignable to (or coercible to) a type-form then
      returns the value narrowed to (or coerced to) that form.
      Otherwise raises an exception.
    * Pattern 1: `def convert[T](value: object, form: TypeForm[T]) -> T`
        * Examples: cattrs.BaseConverter.structure, trycast.checkcast,
          typedload.load
    * Pattern 2:
      ```
      class Converter[T](Generic[T]):
          def __init__(self, form: TypeForm[T]) -> None: ...
          def convert(self, value: object) -> T: ...
      ```
        * Examples: pydantic.TypeAdapter(T).validate_python,
          mashumaro.JSONDecoder(T).decode

* Typed field definitions:
    * Pattern:
      ```
      class Field:
          value_type: TypeForm[T]
      ```
    * Examples: attrs.make_class, dataclasses.make_dataclass, openapify

The survey also identified some introspection functions that take forms (both type-forms and annotation-forms) as input using plain `object`s which would *not* gain functionality by marking those inputs as `TypeForm[]`:

* General introspection operations:

    * Pattern: `def get_form_info(maybe_form: object) -> ...`
    * Examples: typing.{get_origin, get_args}, typing_inspect.{is_*_type, get_origin, get_parameters}

There are also some introspection functions that take type-forms as input and return complex values based on those forms. Such functions would require additional syntax to fully support which is not proposed in this PEP:

* Typed lookup operations:
    * Takes a sequence of type-forms and returns a tuple of instances of those forms.
    * Pattern: `def get_instances(forms: *TypeForm[T]) -> Tuple[*T]`
    * Examples: svcs.svcs_from(...).get(...)
    * Workaround: Use overloads like:
        * ```
          @overload
          def get_instances(t1: TypeForm[T1]) -> Tuple[T1]: ...
          @overload
          def get_instances(t1: TypeForm[T1], t2: TypeForm[T2]) -> Tuple[T1, T2]: ...
          # (... repeat up to tuples of length 7 or so ...)
          ```

Specification

A type-form represents a `type` object or a special typing form such as `Optional[str]`, `Union[int, str]`, or `MyTypedDict`. A type-form type is written as `TypeForm[T]` where `T` is a type or a type variable. It can also be written without brackets as just `TypeForm`, which is treated as shorthand for `TypeForm[Any]`.

Using TypeForms

TypeForm types may be used as function parameter types, return types, and variable types:

```
def is_union_type(form: TypeForm) -> bool: ...  # parameter type
```

```
def union_of[S, T](s: TypeForm[S], t: TypeForm[T]) \
    -> TypeForm[Union[S, T]]: ...  # return type
```

```
STR_TYPE: TypeForm[str] = str  # variable type
```
```

Note however that an *unannotated* variable assigned a TypeForm literal will not be inferred to be of TypeForm type by typecheckers because PEP 484 [^type-alias-syntax] reserves that syntax for defining type aliases:

```
STR_TYPE = str  # OOPS; treated as a type alias!
```

If you want a typechecker to recognize a TypeForm literal in a bare assignment you'll need to explicitly declare the assignment-target as having `TypeForm` type:

```
STR_TYPE: TypeForm[str] = str
```

```
STR_TYPE = str  # type: TypeForm[str]  # the type comment is significant
```

```
STR_TYPE: TypeForm[str]
STR_TYPE = str
```

TypeForm Values

A variable of type `TypeForm[T]` where `T` is a type, may only be assigned a class object or special form which is valid in *all* of the following locations:

* the right-hand-side of a variable declaration,

  ```
  value: *form*
  ```

* the right-hand-side of a parameter declaration,

  ```
  def some_func(value: *form*):
  ```

* the return type of a function:

  ```
  def some_func() -> *form*:
  ```

```

and which is a subtype of `T`.

A runtime object that is valid in only some but not all of the above locations, like `Final[*form*]` (valid only in a variable declaration) or `TypeIs[*form*]` (valid only in a return type), is considered to be an "annotation form" but not a "type form".

Example of type-form values include:

* type objects like `int`, `str`, `object`, and `FooClass`
* generic collections like `List`, `List[int]`, `Dict`, or `Dict[K, V]`
* callables like `Callable`, `Callable[[Arg1Type, Arg2Type], ReturnType]`, `Callable[..., ReturnType]`
* union forms like `int | str`, `Union[int, str]`, `Optional[str]`, or `Never`
* literal forms like `Literal['r', 'rb', 'w', 'wb']`
* type variables like `T` or `AnyStr`
* annotated types like `Annotated[int, ValueRange(-10, 5)]`
* type aliases like `Vector` (where `Vector = list[float]`)
* the `Any` form
* the `Type` and `Type[C]` forms
* the `TypeForm` and `TypeForm[T]` forms
* string literals that spell one of the above values, like `"Optional[str]"`

Incomplete forms like a bare `Optional` or `Union` which do not spell a type are not type-form values.


Forward References
''''''''''''''''''

Type-form values may contain string-based forward references. These forward references are normalized at runtime to be `ForwardRef` instances: [^forward-ref-normalization]

```
>>> IntTree = list[typing.Union[int, 'IntTree']]
>>> IntTree
list[typing.Union[int, ForwardRef('IntTree')]]
```

Therefore `ForwardRef` instances, being equivalent to string-based forward references, are also considered to be type-forms:

```
IntTreeRef: TypeForm = ForwardRef('IntTree') # OK
```
```

[^forward-ref-normalization]: Special forms at runtime normalize string arguments to `ForwardRef` instances using the `typing._type_check()` and `typing._type_convert()` internal helper functions, as of Python 3.12. Runtime typecheckers may wish to implement similar functions when working with string-based forward references.

Stringified TypeForms
'''''''''''''''''''''

A type-form value may itself be a string-based forward reference:

```
IntTreeRef: TypeForm = 'IntTree'  # OK
```

However the string itself must spell a valid type to be considered a type-form:

```
BadUnion1: TypeForm = Union    # ERROR: does not spell a type
BadUnion2: TypeForm = 'Union'  # ERROR: does not spell a type
```

Subtyping

Whether a TypeForm value can be assigned from one variable to another is determined by the following rules for the is-subtype-of and is-consistent-with relationships: [^type-consistency]

TypeForm[] is covariant in its argument type, just like Type[]:

* `TypeForm[T1]` is a subtype of `TypeForm[T2]` iff `T1` is a subtype of `T2`.
* `TypeForm[C1]` is a subtype of `Type[C2]` iff `C1` is a subtype of `C2`.
* `Type[C1]` is a subtype of `TypeForm[C2]` iff `C1` is a subtype of `C2`.

A plain Type can be assigned to a plain TypeForm but not the other way around:

* `Type[Any]` is a subtype of `TypeForm[Any]`. (But not the other way around.)

TypeForm[] is a kind of object, just like Type[]:

* `TypeForm[T]` for any `T` is a subtype of `object`.

Interactions with Type[] and type variables

Both TypeForm[] and Type[] can be used to constrain the same type variable within the same function definition:

```
def as_type[T](form: TypeForm[T]) -> Type[T] | None:
    return form if isinstance(form, type) else None
```

```
def as_instance[T](form: TypeForm[T]) -> T | None:
    return form() if isinstance(form, type) else None
```

Interactions with TypeIs[], TypeGuard[], and type variables

A type variable constrained by TypeForm[] can also be used by a TypeIs[] within the same function definition:

```
def isassignable[T](value: object, form: TypeForm[T]) -> TypeIs[T]: ...

count: int | str = ...
if isassignable(count, int):
    assert_type(count, int)
else:
    assert_type(count, str)
```

or by a TypeGuard[] within the same function definition:

```
def isdefault[T](value: object, form: TypeForm[T]) -> TypeGuard[T]:
    return (value == type()) if isinstance(form, type) else False

value: int | str = ''
if isdefault(value, int):
    assert_type(value, int)
    assert 0 == value
elif isdefault(value, str):
    assert_type(value, str)
    assert '' == value
else:
    assert_type(value, int | str)
```

Interactions with Annotated[] and type variables

Annotated[] forms preserve their metadata at runtime:

```
>>> ValueRange: TypeAlias = slice
>>> PositiveInt: TypeAlias = Annotated[int, ValueRange(1, float('inf'))]
>>> PositiveInt
typing.Annotated[int, slice(1, inf, None)]  # NOT: int
```

However Annotated[] forms need not preserve their metadata at typechecking-time:

```
count: PositiveInt = 1
assert_type(count, int)  # NOT: Annotated[int, ValueRange(1, float('inf'))]
```

In particular when an Annotated[] argument is passed to a TypeForm[] parameter constraining a type variable that is also used by a TypeIs[] or TypeGuard[], the metadata need not be maintained as part of the type inferred by a typechecker:

```
# Similar to isassignable(), but accepts Annotated[] forms describing constraints
def ismatch[T](value: object, form: TypeForm[T]) -> TypeGuard[T]: ...

count: int | str = -1
if ismatch(count, PositiveInt):
    assert_type(count, int)  # NOT: Annotated[int, ValueRange(1, float('inf'))]
else:
    assert_type(count, int | str)
```

Backwards Compatibility
=======================

No backward incompatible changes are made by this PEP.

Reference Implementation
========================

The following will be true when [mypy#9773] is implemented:

    The mypy type checker supports `TypeForm` types. A reference implementation of the runtime component is provided in the `typing_extensions` module.

Rejected Ideas
==============

Widen Type[T] to support all typing special forms

`Type` was [designed] to only be used to describe class objects. A class object can always be instantiated by calling it and can always be used as the second argument of `isinstance()`.

`TypeForm` on the other hand is typically introspected by the user in some way, is not necessarily directly instantiable, and is not necessarily directly usable in a regular `isinstance()` check.

It would be possible to widen `Type` to include the additional values allowed by `TypeForm` but it would reduce clarity about the user's intentions when working with a `Type`. Different concepts and usage patterns; different spellings.

Accept arbitrary annotation-forms

Certain special forms can be used in *some* but not *all* annotation contexts:

For example TypeIs[] and TypeGuard[] can be used as a return type of a function but not as a variable type or a parameter type:

```
def is_positive_int(value: object) -> TypeGuard[int]: ...  # OK

def nonsense(value: TypeGuard[int]): ...  # ERROR: TypeGuard[] not meaningful here

exotic_bool: TypeGuard[int]  # ERROR: TypeGuard[] not meaningful here
```

For example Final[] can be used as a variable type but not as a parameter type or a return type:

```
some_const: Final[str] = ...  # OK

def foo(not_reassignable: Final[object]): ...  # ERROR: Final[] not allowed here

def nonsense() -> Final[object]: ...  # ERROR: Final[] not meaningful here
```

`TypeForm[T]` does not allow matching such annotation-forms which are not type-forms because it is not clear how a type variable in position `T` should be constrained:

```
def ismatch[T](value: object, form: TypeForm[T]) -> TypeGuard[T]: ...

some_value = ...
if ismatch(some_value, TypeGuard[int]):   # ERROR: TypeGuard[int] is not a TypeForm
    reveal_type(some_value)  # ? NOT TypeGuard[int], because invalid for a variable

def foo(some_arg):
    if ismatch(some_arg, Final[int]):  # ERROR: Final[int] is not a TypeForm
        reveal_type(some_arg)  # ? NOT Final[int], because invalid for a parameter
```

Functions that wish to operate on *all* kinds of annotation-forms, including those that are not type-forms, can continue to accept such forms as `object` parameters, as they must do so today:

* typing.py, from the standard library:
    * `def get_origin(maybe_annotation_form: object) -> object: ...`
        * Accepts type-forms like `list[int]`, annotation-forms like `Final[int]`, incomplete forms like `Union`, and non-forms like `1`.
        * Returns type-forms like `list`, annotation-forms like `Final`, incomplete forms like `Union`, and `None`.
    * `def get_args(maybe_annotation_form: object) -> tuple[object, ...]: ...`
        * Accepts type-forms like `list[int]`, annotation-forms like `Final[int]`, incomplete forms like `Union`, and non-forms like `1`.
        * Returns a tuple containing type-forms like `list` and non-forms like ``'annotated_metadata'`` (from inside `Annotated[]` forms).

Copyright


```
..
   Local Variables:
   mode: indented-text
   indent-tabs-mode: nil
   sentence-end-double-space: t
   fill-column: 70
   coding: utf-8
   End:
```