kPEP: 9999
Title: Self Type
Author: Pradeep Kumar Srinivasan <gohanpra@gmail.com>,
        James Hilton-Balfe <gobot1234yt@gmail.com>
Sponsor: Jelle Zijlstra <jelle.zijlstra@gmail.com>
Status: Draft
Type: Standards Track
Content-Type: text/x-rst
Created: xxx
Python-Version: 3.11
Post-History:

Abstract

This PEP introduces a simple and intuitive way to annotate methods that return an instance of their class. This behaves the same as the ``TypeVar``-based approach specified in PEP 484 [#pep-484-original]_ but is more concise and easier to follow.

Motivation

A common use case is to write a method that returns an instance of the same class, usually by returning ``self``.

::

```
    class Shape:
        def set_scale(self, scale: float):
            self.scale = scale
            return self

    Shape().set_scale(0.5)  # => should be Shape
```

One way to denote the return type is to specify it as the current class, say, ``Shape``. Using the method makes the type checker infer the type ``Shape``, as expected.

::

```
    class Shape:
        def set_scale(self, scale: float) -> Shape:
            self.scale = scale
            return self

    Shape().set_scale(0.5)  # => Shape
```

However, when we call ``set_scale`` on a subclass of ``Shape``, the type checker still infers the return type to be ``Shape``. This is problematic in situations such as the one shown below, where the type checker will return an error because we are trying to use attributes or methods not present on the base class.

::

```
    class Circle(Shape):
        def set_radius(self, r: float) -> Circle:
            self.radius = r
            return self

    Circle().set_scale(0.5)  # *Shape*, not Circle
    Circle().set_scale(0.5).set_radius(2.7)
    # => Error: Shape has no attribute set_radius
```

The present workaround for such instances is to define a ``TypeVar`` with the base class as the bound and use it as the annotation for the ``self`` parameter and the return type:

::

```
    from typing import TypeVar

    TShape = TypeVar("TShape", bound="Shape")

    class Shape:
        def set_scale(self: TShape, scale: float) -> TShape:
            self.scale = scale
            return self


    class Circle(Shape):
        def set_radius(self, radius: float) -> Circle:
            self.radius = radius
            return self

    Circle().set_scale(0.5).set_radius(2.7)  # => Circle
```

Unfortunately, this is verbose and unintuitive. Because ``self`` is usually not explicitly annotated, the above solution doesn't immediately come to mind, and even if it does, it is very easy to go wrong by forgetting either the bound on the ``TypeVar(bound="Shape")`` or the annotation for ``self``.

This difficulty means that users often give up and either use fallback types like ``Any`` or just omit the type annotation completely, both of which make the code less safe.

We propose a more intuitive and succinct way of expressing the above intention. We introduce a special form ``Self`` that stands for a type variable bound to the encapsulating class. For situations such as the one above, the user simply has to annotate the return type as ``Self``:

::

```
    from typing import Self
——
    class Shape:
        def set_scale(self, scale: float) -> Self:
            self.scale = scale
            return self


    class Circle(Shape):
        def set_radius(self, radius: float) -> Self:
            self.radius = radius
            return self
```

By annotating the return type as ``Self``, we no longer have to declare a ``TypeVar`` with an explicit bound on the base class. The return type ``Self`` mirrors the fact that the function returns ``self`` and is easier to understand.

As in the above example, the type checker will correctly infer the type of ``Circle().set_scale(0.5)`` to be ``Circle``, as expected.

Usage statistics

We analyzed popular open-source projects [#self-type-usage-stats]_ and found that patterns like the above were used about **40%** as often as popular types like ``dict`` or ``Callable``. For example, in typeshed alone, such “Self” types are used 523 times, compared to 1286 uses of ``dict`` and 1314 uses of ``Callable`` as of October 2021 [#callable-dict-usage-stats]_. This suggests that a ``Self`` type will be used quite often and users will benefit a lot from the simpler approach above.

Specification

Use in Method Signatures

``Self`` used in the signature of a method is treated as if it were a ``TypeVar`` bound to the class.

::

```
    from typing import Self

    class Shape:
        def set_scale(self, scale: float) -> Self:
            self.scale = scale
            return self
```

is treated equivalently to:

::

```
    from typing import TypeVar

    SelfShape = TypeVar("SelfShape", bound="Shape")

    class Shape:
        def set_scale(self: SelfShape, scale: float) -> SelfShape:
            self.scale = scale
            return self
```

This works the same for a subclass too:

::

```
    class Circle(Shape):
        def set_radius(self, radius: float) -> Self:
            self.radius = radius
            return self
```

which is treated equivalently to:

::

```
    SelfCircle = TypeVar("SelfCircle", bound="Circle")

    class Circle(Shape):
        def set_radius(self: SelfCircle, radius: float) -> SelfCircle:
            self.radius = radius
            return self
```

One implementation strategy is to simply desugar the former to the latter in a preprocessing step. If a method uses ``Self`` in its signature, the type of ``self`` within a method will be ``Self``. In other cases, the type of ``self`` will remain the enclosing class.

Use in Classmethod Signatures

The ``Self`` type annotation is also useful for ``classmethod``s that return an instance of the class that they operate on. For example, ``from_config`` in the following snippet builds a ``Shape`` object from a given ``config``.

::

```
    class Shape:
        def __init__(self, scale: float) -> None: ...

        @classmethod
        def from_config(cls, config: dict[str, float]) -> Shape:
            return cls(config["scale"])
```

However, this means that ``Circle.from_config(...)`` is inferred to return a value of type ``Shape``, when in fact it should be ``Circle``:

::

```
    class Circle(Shape): ...

    shape = Shape.from_config({"scale": 7.0})     # => type: Shape
    circle = Circle.from_config({"scale": 7.0})   # => type: *Shape*, not Circle

    circle.circumference()
    # Error: `Shape` has no attribute `circumference`
```

The current workaround for this is unintuitive and error-prone:

::

```
    Self = TypeVar("Self", bound="Shape")

    class Shape:
        @classmethod
        def from_config(cls: type[Self], config: dict[str, float]) -> Self:
            return cls(config["scale"])
```

We propose using ``Self`` directly:

::

```
    from typing import Self

    class Shape:
        @classmethod
        def from_config(cls, config: dict[str, float]) -> Self:
            return cls(config["scale"])
```

This avoids the complicated ``cls: type[Self]`` annotation and the ``TypeVar`` declaration with a ``bound``. Once again, the latter code behaves equivalently to the former code.

Use in Parameter Types

Another use for ``Self`` is to annotate parameters that expect instances of the current class:

::

```
    Self = TypeVar("Self", bound="Shape")

    class Shape:
        def difference(self: Self, other: Self) -> float: ...

        def apply(self: Self, f: Callable[[Self], None]) -> None: ...
```

We propose using ``Self`` directly to achieve the same behavior:

::

```
    from typing import Self

    class Shape:
        def difference(self, other: Self) -> float: ...

        def apply(self, f: Callable[[Self], None]) -> None: …
```

Note that specifying ``self: Self`` is harmless, so some users may find it more readable to write the above as:

::

```
    class Shape:
        def difference(self: Self, other: Self) -> float: ...
```

Use in Attribute Annotations

Another use for ``Self`` is to annotate attributes. One example is where we have a ``LinkedList`` whose elements must be subclasses of the current class.

::

```
   from dataclasses import dataclass
   from typing import Generic, TypeVar

   T = TypeVar("T")

   @dataclass
   class LinkedList(Generic[T]):
       value: T
       next: LinkedList[T] | None = None

   # OK
   LinkedList[int](value=1, next=LinkedList[int](value=2))
   # Not OK
   LinkedList[int](value=1, next=LinkedList[str](value=”hello”))
```

However, annotating the ``next`` attribute as ``LinkedList[T]`` allows invalid constructions with subclasses:

::

```
   @dataclass
   class OrdinalLinkedList(LinkedList[int]):
       def ordinal_value(self) -> str:
           return as_ordinal(self.value)

   # Should not be OK because LinkedList[int] is not a subclass of OrdinalLinkedList,
   # but the type checker allows it.
   xs = OrdinalLinkedList(value=1, next=LinkedList[int](value=2))
   if xs.next:
       print(xs.next.ordinal_value())  # Runtime Error.
```

We propose expressing this constraint using ``next: Self | None``:

::

```
   from typing import Self

   @dataclass
   class LinkedList(Generic[T]):
       next: Self | None = None
       value: T


   @dataclass
   class OrdinalLinkedList(LinkedList[int]):
       def ordinal_value(self) -> str:
           return as_ordinal(self.value)

   xs = OrdinalLinkedList(value=1, next=LinkedList[int](value=2))
   # Type error: Expected OrdinalLinkedList, got LinkedList[int].

   if xs.next is not None:
       xs.next = OrdinalLinkedList(value=3, next=None)  # OK
       xs.next = LinkedList[int](value=3, next=None)  # Not OK
```

The code above is semantically equivalent to treating each attribute containing a ``Self`` type as a ``property`` that returns that type:

::

```
    from dataclasses import dataclass
    from typing import Any, Generic, TypeVar

    T = TypeVar("T")
    Self = TypeVar("Self", bound="LinkedList")


    class LinkedList(Generic[T]):
        value: T

        @property
        def next(self: Self) -> Self | None:
            return self._next

        @next.setter
        def next(self: Self, next: Self | None) → None:
            self._next = next

    class OrdinalLinkedList(LinkedList[int]):
        def ordinal_value(self) → str:
            return str(self.value)
```

Use in Generic Classes
^^^^^^^^^^^^^^^^^^^^^^

``Self`` can also be used in generic class methods:

::

```
    class Container(Generic[T]):
        value: T
        def set_value(self, value: T) → Self: ...
```

This is equivalent to writing:

::

```
    Self = TypeVar("Self", bound="Container[Any]")

    class Container(Generic[T]):
        value: T
        def set_value(self: Self, value: T) → Self: ...
```

The behavior is to preserve the type argument of the object on which the method was called. When called on an object with concrete type ``Container[int]``, ``Self`` is bound to ``Container[int]``. When called with an object of generic type ``Container[T]``, ``Self`` is bound to ``Container[T]``:

::

```
    def object_with_concrete_type() → None:
        int_container: Container[int]
        str_container: Container[str]
        reveal_type(int_container.set_value(42))  # ⇒ type: Container[int]
        reveal_type(str_container.set_value("hello"))  # ⇒ type: Container[str]

    def object_with_generic_type(container: Container[T], value: T) → Container[T]:
        return container.set_value(value)  # type: Container[T]
```

The PEP doesn't specify the exact type of ``self.value`` within the method ``set_value``. Some type checkers may choose to implement ``Self`` types using class-local type variables with ``Self = TypeVar("Self", bound=Container[T])``, which will infer a precise type ``T``. However, given that class-local type variables are not a standardized type system feature, it is also acceptable to infer ``Any`` for ``self.value``. We leave this up to the type checker.

Note that we reject using ``Self`` with type arguments, such as ``Self[int]``. This is because it creates ambiguity about the type of the ``self`` parameter and introduces unnecessary complexity:

::

```
    class Container(Generic[T]):
        def foo(self, other: Self[int], other2: Self) -> Self[str]:  ...  # Rejected
```

In such cases, we recommend using an explicit type for ``self``:

::

```
    class Container(Generic[T]):
        def foo(self: Container[T], other: Container[int], other2: Container[T]) -> Container[str]:  ...
```

Use in Protocols

``Self`` is valid within Protocols, similar to its use in classes:

::

```
    from typing import Protocol, Self

    class Shape(Protocol):
        scale: float

        def set_scale(self, scale: float) -> Self:
            self.scale = scale
            return self
```

is treated equivalently to:

::

```
    from typing import TypeVar

    SelfShape = TypeVar("SelfShape", bound="ShapeProtocol")

    class Shape(Protocol):
        scale: float

        def set_scale(self: SelfShape, scale: float) -> SelfShape:
            self.scale = scale
            return self
```

See [#protocol-self-type]_ for details on the behavior of ``TypeVar``s bound to protocols.

Checking a class for compatibility with a protocol: If a protocol uses ``Self`` in methods or attribute annotations, then a class ``Foo`` is considered compatible with the protocol if its corresponding methods and attribute annotations use either ``Self`` or ``Foo`` or any of ``Foo``'s subclasses. See the examples below:

::

```
    from typing import Protocol

    class ShapeProtocol(Protocol):
        def set_scale(self, scale: float) -> Self:  ...

    class ReturnSelf:
        scale: float = 1.0

        def set_scale(self, scale: float) -> Self:
            self.scale = scale
            return self

    class ReturnConcreteShape:
        scale: float = 1.0

        def set_scale(self, scale: float) -> ReturnConcreteShape:
            self.scale = scale
            return self

    class BadReturnType:
        scale: float = 1.0

        def set_scale(self, scale: float) -> int:
            self.scale = scale
            return 42

    class ReturnDifferentClass:
        scale: float = 1.0

        def set_scale(self, scale: float) -> ReturnConcreteShape:
            return ReturnConcreteShape(...)

    def accepts_shape(shape: ShapeProtocol) -> None:
        y = shape.set_scale(0.5)
        reveal_type(y)

    def main() -> None:
        return_self_shape: ReturnSelf
        return_concrete_shape: ReturnConcreteShape
        bad_return_type: BadReturnType
        return_different_class: ReturnDifferentClass

        accepts_shape(return_self_shape)  # OK
        accepts_shape(return_concrete_shape)  # OK
        accepts_shape(bad_return_type)  # Not OK
        accepts_shape(return_different_class)  # Not OK because it returns a non-subclass.
```

Valid Locations for ``Self``

A ``Self`` annotation is only valid in class contexts, and will always refer to the encapsulating class. In contexts involving nested classes, ``Self`` will always refer to the innermost class.

The following uses of ``Self`` are accepted:

::

```
    class ReturnsSelf:
        def foo(self) -> Self: ... # Accepted

        @classmethod
        def bar(cls) -> Self:  # Accepted
            return cls()

        def __new__(cls, value: int) -> Self: ...  # Accepted

        def explicitly_use_self(self: Self) -> Self: ...  # Accepted

        def returns_list(self) -> list[Self]: ...  # Accepted (Self can be nested within other types)

        @classmethod
        def return_cls(cls) -> type[Self]:  # Accepted (Self can be nested within other types)
            return cls

    class Child(ReturnsSelf):
        def foo(self) -> Self: ...  # Accepted (we can override a method that uses Self annotations)

    class TakesSelf:
        def foo(self, other: Self) -> bool: ...  # Accepted

    class Recursive:
        next: Self | None  # Accepted (treated as an @property returning ``Self | None``)

    class CallableAttribute:
        def foo(self) -> int: ...

        bar: Callable[[Self], int] = foo  # Accepted (treated as an @property returning the Callable type)

    TupleSelf = Tuple[Self, Self]
    class Alias:
        def return_tuple(self) -> TupleSelf:
            return (self, self)
    class HasNestedFunction:
        x: int = 42

        def foo(self) -> None:

            def nested(z: int, inner_self: Self) -> Self:  # Accepted (Self is bound to HasNestedFunction)
                print(z)
                print(inner_self.x)
                return inner_self

            nested(42, self)  # OK


    class Outer:
        class Inner:
            def foo(self) -> Self: ...  # Accepted (Self is bound to Inner)
```

The following uses of ``Self`` are rejected.

::

```
    def foo(bar: Self) -> Self: ...  # Rejected (not within a class)

    bar: Self  # Rejected (not within a class)

    class Foo:
        def has_existing_self_annotation(self: T) -> Self: ...  # Rejected (Self is treated as unknown)

    class Foo:
        def return_concrete_type(self) -> Self:
            return Foo()  # Rejected (see FooChild below for rationale)

    class FooChild(Foo):
        child_value: int = 42

        def child_method(self) -> None:
            y = self.return_concrete_type()  # At runtime, this would be Foo, not FooChild.
            y.child_value  # Runtime error: Foo has no attribute child_value

    class Bar(Generic[T]):
        def bar(self) -> T: ...

    class Baz(Foo[Self]): ...  # Rejected
```

Note that we reject ``Self`` in ``staticmethod``s. ``Self`` does not add much value since there is no ``self`` or ``cls`` to return. The only possible use cases would be to return a parameter itself or some element from a container passed in as a parameter. These don’t seem worth the additional complexity.

::

```
    class Base:
        @staticmethod
        def make() -> Self:  # Rejected
            ...  # No possible return value will be valid since a concrete ``Base`` is not compatible with ``Self``.

        @staticmethod
        def return_parameter(foo: Self) -> Self:  # Rejected
            ...  # The only possible return value is ``foo``, which is not very useful.
                 # So, we reject ``Self`` within staticmethods.
```

Likewise, we reject ``Self`` in metaclasses. ``Self`` in this PEP consistently refers to the same type (that of ``self``). But in metaclasses, it would have to refer to different types in different method signatures. For example, in ``__mul__``, ``Self`` in the return type would refer to the implementing class `Foo`, not the enclosing class ``MyMetaclass``. But, in ``__new__``, ``Self`` in the return type would refer to the enclosing class ``MyMetaclass``. To avoid confusion, we reject this edge case.
::

```
    class MyMetaclass(type):
        def __new__(cls, *args: Any) -> Self:  # Rejected
            return super().__new__(cls, *args)

        def __mul__(cls, count: int) -> list[Self]:  # Rejected
            return [cls()] * count

    class Foo(metaclass=MyMetaclass): ...
```

Runtime behavior

Because ``Self`` is not subscriptable, we propose an implementation similar to ``typing.NoReturn``.

::

```
    @_SpecialForm
    def Self(self, params):
        """Used to spell the type of "self" in classes.

        Example::

          from typing import Self

          class ReturnsSelf:
              def parse(self, data: bytes) -> Self:
                  ...
                  return self

        """
        raise TypeError(f"{self} is not subscriptable")
```

Rejected Alternatives
=====================

Allow the Type Checker to Infer the Return Type

One proposal is to leave the ``Self`` type implicit and let the type checker infer from the body of the method that the return type must be the same as the type of the ``self`` parameter:

```
::

    class Shape:
        def set_scale(self, scale: float):
            self.scale = scale
            return self  # Type checker infers that we are returning self
```

We reject this because Explicit Is Better Than Implicit. Beyond that, the above approach will fail for type stubs, which don't have method bodies to analyze.

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

Pyre: TODO
Mypy: https://github.com/Gobot1234/mypy

Resources
=========

Similar discussions on a ``Self`` type in Python started in Mypy around 2016: [#mypy1212]_. However, the approach ultimately taken there was the bounded ``TypeVar`` approach shown in our "before" examples. Other issues that discuss this include [#mypy2354]_.

Pradeep made a concrete proposal at the PyCon Typing Summit 2021: [#type-variables-for-all]_ ([#type-variables-for-all-slides]_).

James brought up the proposal independently on typing-sig: [#james-typing-sig]_.

Other languages have similar ways to express the type of the enclosing class:
  - TypeScript has the ``this`` type [#typescript-this-type]_
  - Rust has the ``Self`` type [#rust-self-type]_

Thanks to the following people for their feedback on the PEP:

Jia Chen, Rebecca Chen, Sergei Lebedev, Kaylynn Morgan, Tuomas Suutari, Alex Waygood, Shannon Zhu, and Никита Соболев

References
==========

.. [#mypy1212] SelfType or another way to spell "type of self"

    https://github.com/python/mypy/issues/1212

.. [#mypy2354] Self types in generic classes

    https://github.com/python/mypy/issues/2354

.. [#type-variables-for-all] Type Variables for All talk

    https://youtu.be/ld9rwCvGdhc?t=3260

.. [#type-variables-for-all-slides] Slides

    https://drive.google.com/file/d/1x-qoDVY_OvLpIV1EwT7m3vm4HrgubHPG/view

.. [#james-typing-sig] Thread

    https://mail.python.org/archives/list/typing-sig@python.org/thread/SJAANGA2CWZ6D6TJ7KOPG7PZQC56K73S/#B2CBLQDHXQ5HMFUMS4VNY2D4YDCFT64Q

.. [#property-workaround] Property workaround

    https://mypy-play.net/?mypy=latest&python=3.8&gist=ae886111451833e38737721a4153fd96

.. [#self-type-usage-stats] Self type usage stats

    https://github.com/pradeep90/annotation_collector/#self-type-stats

.. [#callable-dict-usage-stats] ``Callable`` and ``dict`` usage stats

    https://github.com/pradeep90/annotation_collector/#overall-stats-in-typeshed

.. [#protocol-self-type] Protocol-bound ``TypeVar``

    https://www.python.org/dev/peps/pep-0544/#self-types-in-protocols

.. [#rust-self-type] Rust ``Self`` type

    https://doc.rust-lang.org/std/keyword.SelfTy.html

.. [#typescript-this-type] TypeScript ``this`` type

    https://typescriptlang.org/docs/handbook/2/classes.html#this-types

Copyright

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