# Python Typing Constructs Without \_\_type\_params\_\_

Below is a comprehensive list of standard typing module constructs that take type parameters but do **not** have a \_\_type\_params\_\_ attribute (accessing it would raise an AttributeError). For each, we note how type arguments are stored (usually in an internal tuple \_\_args\_\_, sometimes alongside other attributes):

## Special Typing Forms and Aliases

* **Union** – Used as Union[T1, T2, …] (now often written as T1 | T2). It accepts multiple type arguments and flattens them into a tuple in \_\_args\_\_[[1]](https://stackoverflow.com/questions/48572831/how-to-access-the-type-arguments-of-typing-generic#:~:text=Note%20that%20the%20same%20methods,str%2C%20NoneType). For example, Union[int,str] yields \_\_args\_\_ == (<class 'int'>, <class 'str'>). No \_\_type\_params\_\_ attribute is present.
* **Optional** – Shorthand for a union with None (Optional[X] == Union[X, None][[2]](https://docs.python.org/3/library/typing.html#:~:text=typing)). For instance, Optional[int] results in \_\_args\_\_ == (<class 'int'>, <class 'NoneType'>) (treated as a union). No \_\_type\_params\_\_ is defined (it’s just a specialized Union form).
* **Literal** – Used as Literal[v1, v2, …] to indicate specific constant values. The literal values themselves are stored in the \_\_args\_\_ tuple[[3]](https://stackoverflow.com/questions/59163553/getting-the-literal-out-of-a-python-literal-type-at-runtime#:~:text=,get_args%28l%29%20%28%27add%27%2C%20%27mul). For example, Literal[42, "hi"].\_\_args\_\_ == (42, 'hi'). No \_\_type\_params\_\_ attribute exists.
* **Annotated** – Used as Annotated[T, metadata...] to attach metadata to a type. The underlying type is stored in \_\_args\_\_ (e.g. Annotated[int, ...].\_\_args\_\_ == (<class 'int'>,)) and the metadata is stored in a separate \_\_metadata\_\_ attribute[[4]](https://docs.python.org/3/library/typing.html#:~:text=At%20runtime%2C%20the%20metadata%20associated,attribute). For example, X = Annotated[int, "note"] will have X.\_\_origin\_\_ == int and X.\_\_metadata\_\_ == ("note",)[[5]](https://docs.python.org/3/library/typing.html#:~:text=,very%27%2C%20%27important%27%2C%20%27metadata%27%29). No \_\_type\_params\_\_ attribute is present.
* **ClassVar** – Denotes class-level variables (ClassVar[T]). The wrapped type is stored in \_\_args\_\_ (e.g. ClassVar[int].\_\_args\_\_ == (<class 'int'>,)), and its origin is a special marker typing.ClassVar. No \_\_type\_params\_\_ is defined.
* **Final** – Indicates a final (non-reassignable) variable (Final[T]). Internally works like ClassVar: the type argument is in \_\_args\_\_ (e.g. Final[str].\_\_args\_\_ == (<class 'str'>,)) and origin is typing.Final. No \_\_type\_params\_\_ attribute.
* **Type** – Parameterized as Type[C] to mean “type of a subclass of C”. It stores the target class in \_\_args\_\_ (e.g. Type[int].\_\_args\_\_ == (<class 'int'>,)) and typically has \_\_origin\_\_ = <class 'type'>. No \_\_type\_params\_\_ attribute exists.
* **TypeGuard** – Used as TypeGuard[T] in user-defined type guard functions. The guarded type is in \_\_args\_\_ (e.g. TypeGuard[SomeType].\_\_args\_\_ == (<class 'SomeType'>,)) and origin is the special form typing.TypeGuard. No \_\_type\_params\_\_ attribute.
* **Required[T] / NotRequired[T]** – Marks a key in a TypedDict as required or not required (PEP 655). Each is parameterized by the value type. The given type is stored in \_\_args\_\_ (e.g. Required[int].\_\_args\_\_ == (<class 'int'>,)), with origin typing.Required or typing.NotRequired. They do not define \_\_type\_params\_\_.
* **Concatenate** – A special form for higher-order callables (PEP 612). Used as Concatenate[T1, T2, ..., P] (last arg must be a ParamSpec). All arguments are stored in \_\_args\_\_ (including the ParamSpec). For example, Concatenate[int, str, P].\_\_args\_\_ == (<class 'int'>, <class 'str'>, ~P). No \_\_type\_params\_\_ attribute.
* **Unpack** – A special form to unpack a TypeVarTuple. Used as Unpack[Ts]. It stores the tuple type variable in \_\_args\_\_ (e.g. Unpack[Ts].\_\_args\_\_ == (Ts,)). No \_\_type\_params\_\_ attribute is present (origin is typing.Unpack).
* **Callable** – Parameterized as Callable[[T1, T2, ...], R] for a function with argument types and return type. Internally, it flattens the argument types and return type into a single \_\_args\_\_ tuple[[6]](http://www.brython.info/src/Lib/typing.py#:~:text=_ConcatenateGenericAlias%29%29%20def%20_should_unflatten_callable_args%28typ%2C%20args%29%3A%20,args%29%20%3D%3D%202%20and). For example, Callable[[int,str], bool] has \_\_args\_\_ == (<class 'int'>, <class 'str'>, <class 'bool'>) (the last item is the return type)[[6]](http://www.brython.info/src/Lib/typing.py#:~:text=_ConcatenateGenericAlias%29%29%20def%20_should_unflatten_callable_args%28typ%2C%20args%29%3A%20,args%29%20%3D%3D%202%20and). typing.get\_args will regroup these into ([int, str], bool) for convenience[[3]](https://stackoverflow.com/questions/59163553/getting-the-literal-out-of-a-python-literal-type-at-runtime#:~:text=,get_args%28l%29%20%28%27add%27%2C%20%27mul). No \_\_type\_params\_\_ attribute exists.

## Concrete Generic Collection Types

* **List** – Generic alias for the built-in list (List[T]). For example, List[int] stores its type argument in \_\_args\_\_ (yielding (<class 'int'>,))[[7]](https://stackoverflow.com/questions/48572831/how-to-access-the-type-arguments-of-typing-generic#:~:text=%3E%3E%3E%20List%5Bint%5D.__args__%20%28) and has \_\_origin\_\_ pointing to the built-in list type. No \_\_type\_params\_\_ attribute.
* **Tuple** – Used as Tuple[T1, T2, …] for fixed-length tuples or Tuple[T, ...] for variable-length homogeneous tuples. All component types are stored in \_\_args\_\_[[8]](https://stackoverflow.com/questions/48572831/how-to-access-the-type-arguments-of-typing-generic#:~:text=match%20at%20L524%20,class%20%27str). For instance, Tuple[int,str].\_\_args\_\_ == (<class 'int'>, <class 'str'>)[[8]](https://stackoverflow.com/questions/48572831/how-to-access-the-type-arguments-of-typing-generic#:~:text=match%20at%20L524%20,class%20%27str), and Tuple[int, ...].\_\_args\_\_ == (<class 'int'>, Ellipsis). No \_\_type\_params\_\_ attribute.
* **Dict** – Generic alias for the built-in dict (Dict[K, V]). For example, Dict[str,int] has \_\_args\_\_ == (<class 'str'>, <class 'int'>)[[9]](https://stackoverflow.com/questions/48572831/how-to-access-the-type-arguments-of-typing-generic#:~:text=typing%20we%20can%20see%3A) and origin dict. No \_\_type\_params\_\_ defined.
* **DefaultDict** – Alias for collections.defaultdict (DefaultDict[K, V]). Stores key and value types in \_\_args\_\_ just like Dict (e.g. DefaultDict[K,V].\_\_args\_\_ == (K, V)) and has origin collections.defaultdict. No \_\_type\_params\_\_.
* **OrderedDict** – Alias for collections.OrderedDict (OrderedDict[K, V]). Behaves like Dict in terms of parameters: \_\_args\_\_ holds the key and value types. No \_\_type\_params\_\_ attribute.
* **Deque** – Alias for collections.deque (Deque[T]). Stores its item type in \_\_args\_\_ (e.g. one type in the tuple). No \_\_type\_params\_\_.
* **Set** – Alias for built-in set (Set[T]). Type argument is in \_\_args\_\_ (e.g. (T,)). No \_\_type\_params\_\_.
* **FrozenSet** – Alias for built-in frozenset (FrozenSet[T]). Type argument stored in \_\_args\_\_. No \_\_type\_params\_\_.
* **ChainMap** – Alias for collections.ChainMap (ChainMap[K, V]). Key and value types in \_\_args\_\_ tuple. No \_\_type\_params\_\_.
* **Counter** – Alias for collections.Counter (Counter[T]). Element type in \_\_args\_\_. No \_\_type\_params\_\_.
* **Pattern** – Alias for re.Pattern (Pattern[T], where T is typically str or bytes). The text type is stored in \_\_args\_\_ (e.g. (str,)) and origin is re.Pattern. No \_\_type\_params\_\_.
* **Match** – Alias for re.Match (Match[T]). Similarly stores the text type in \_\_args\_\_ (e.g. (bytes,) for Match[bytes]). No \_\_type\_params\_\_.
* **IO** – Generic class for file I/O streams (IO[AnyStr]). The type parameter (usually str or bytes) is stored in \_\_args\_\_. For example, IO[str] has \_\_args\_\_ == (<class 'str'>,) and origin typing.IO (a class). No \_\_type\_params\_\_ attribute is present on these stream types.

## Generic ABCs and Protocols

* **Iterable** – (Iterable[T]) from collections.abc. Stores the item type in \_\_args\_\_ (e.g. (T,)). No \_\_type\_params\_\_.
* **Iterator** – (Iterator[T]). Item type in \_\_args\_\_. No \_\_type\_params\_\_.
* **Sequence** – (Sequence[T]). Type in \_\_args\_\_. No \_\_type\_params\_\_.
* **MutableSequence** – (MutableSequence[T]). Type in \_\_args\_\_. No \_\_type\_params\_\_.
* **Mapping** – (Mapping[K, V]). Key and value types in \_\_args\_\_. No \_\_type\_params\_\_.
* **MutableMapping** – (MutableMapping[K, V]). Key/value in \_\_args\_\_. No \_\_type\_params\_\_.
* **Mapping Views** – **KeysView[K]**, **ValuesView[V]**, **ItemsView[K, V]**: generic view types for dictionary keys, values, and items. They store their type parameters in \_\_args\_\_ (e.g. KeysView[int].\_\_args\_\_ == (<class 'int'>,), ItemsView[str,int].\_\_args\_\_ == (<class 'str'>, <class 'int'>)). No \_\_type\_params\_\_ attribute.
* **Collection** – (Collection[T]). Stores element type in \_\_args\_\_. No \_\_type\_params\_\_.
* **Container** – (Container[T]). Element type in \_\_args\_\_. No \_\_type\_params\_\_.
* **Reversible** – (Reversible[T]). Element type in \_\_args\_\_. No \_\_type\_params\_\_.
* **Awaitable** – (Awaitable[T]). Result type in \_\_args\_\_. No \_\_type\_params\_\_.
* **AsyncIterable** – (AsyncIterable[T]). Element type in \_\_args\_\_. No \_\_type\_params\_\_.
* **AsyncIterator** – (AsyncIterator[T]). Element type in \_\_args\_\_. No \_\_type\_params\_\_.
* **Coroutine** – (Coroutine[Y, S, R]) for async coroutine result. It stores **Y**, **S**, **R** (yield type, send type, return type) in \_\_args\_\_. For example, Coroutine[int, str, float].\_\_args\_\_ == (<class 'int'>, <class 'str'>, <class 'float'>). No \_\_type\_params\_\_.
* **AsyncGenerator** – (AsyncGenerator[Y, S]) for asynchronous generators. Stores yield type and send type in \_\_args\_\_ (e.g. AsyncGenerator[int, str].\_\_args\_\_ == (<class 'int'>, <class 'str'>)). No \_\_type\_params\_\_.
* **Generator** – (Generator[Y, S, R]) for generator objects. Stores yield type, send type, and return type in \_\_args\_\_. For example Generator[int, None, float].\_\_args\_\_ == (<class 'int'>, <class 'NoneType'>, <class 'float'>). No \_\_type\_params\_\_ attribute.
* **ContextManager** – (ContextManager[T]) from contextlib. Stores the context result type in \_\_args\_\_ (e.g. (T,)). No \_\_type\_params\_\_.
* **AsyncContextManager** – (AsyncContextManager[T]). Result type in \_\_args\_\_. No \_\_type\_params\_\_.
* **Numeric Protocols** – **SupportsAbs[T]**, **SupportsFloat[T]**, **SupportsInt[T]**, **SupportsBytes[T]**, **SupportsComplex[T]**, **SupportsRound[T]**, **SupportsIndex[T]** are protocol classes that allow specifying the return type of the respective special method (e.g. \_\_abs\_\_, \_\_float\_\_, etc.). Each accepts one type parameter (covariant) and stores it in \_\_args\_\_. For example, SupportsAbs[int].\_\_args\_\_ == (<class 'int'>,). None of these define a \_\_type\_params\_\_ attribute.

**Sources:** The internal \_\_args\_\_ storage for type parameters is documented in code and examples[[9]](https://stackoverflow.com/questions/48572831/how-to-access-the-type-arguments-of-typing-generic#:~:text=typing%20we%20can%20see%3A)[[7]](https://stackoverflow.com/questions/48572831/how-to-access-the-type-arguments-of-typing-generic#:~:text=%3E%3E%3E%20List%5Bint%5D.__args__%20%28). All the above constructs rely on \_\_args\_\_ (and similar dunder attributes like \_\_metadata\_\_ for Annotated) to hold their parameter types at runtime, and **none expose \_\_type\_params\_\_** – that attribute is introduced by newer generics (PEP 695) and is not present on these legacy typing types[[10]](https://peps.python.org/pep-0695/#:~:text=Accessing%20Type%20Parameters%20at%20Runtime,classes%2C%20functions%2C%20and%20type%20aliases)[[11]](https://docs.python.org/3/library/typing.html#:~:text=__type_params__%C2%B6).

[[1]](https://stackoverflow.com/questions/48572831/how-to-access-the-type-arguments-of-typing-generic#:~:text=Note%20that%20the%20same%20methods,str%2C%20NoneType) [[7]](https://stackoverflow.com/questions/48572831/how-to-access-the-type-arguments-of-typing-generic#:~:text=%3E%3E%3E%20List%5Bint%5D.__args__%20%28) [[8]](https://stackoverflow.com/questions/48572831/how-to-access-the-type-arguments-of-typing-generic#:~:text=match%20at%20L524%20,class%20%27str) [[9]](https://stackoverflow.com/questions/48572831/how-to-access-the-type-arguments-of-typing-generic#:~:text=typing%20we%20can%20see%3A) python - How to access the type arguments of typing.Generic? - Stack Overflow

<https://stackoverflow.com/questions/48572831/how-to-access-the-type-arguments-of-typing-generic>

[[2]](https://docs.python.org/3/library/typing.html#:~:text=typing) [[4]](https://docs.python.org/3/library/typing.html#:~:text=At%20runtime%2C%20the%20metadata%20associated,attribute) [[5]](https://docs.python.org/3/library/typing.html#:~:text=,very%27%2C%20%27important%27%2C%20%27metadata%27%29) [[11]](https://docs.python.org/3/library/typing.html#:~:text=__type_params__%C2%B6) typing — Support for type hints — Python 3.13.7 documentation

<https://docs.python.org/3/library/typing.html>

[[3]](https://stackoverflow.com/questions/59163553/getting-the-literal-out-of-a-python-literal-type-at-runtime#:~:text=,get_args%28l%29%20%28%27add%27%2C%20%27mul) Getting the literal out of a python Literal type, at runtime? - Stack Overflow

<https://stackoverflow.com/questions/59163553/getting-the-literal-out-of-a-python-literal-type-at-runtime>

[[6]](http://www.brython.info/src/Lib/typing.py#:~:text=_ConcatenateGenericAlias%29%29%20def%20_should_unflatten_callable_args%28typ%2C%20args%29%3A%20,args%29%20%3D%3D%202%20and) www.brython.info

<http://www.brython.info/src/Lib/typing.py>

[[10]](https://peps.python.org/pep-0695/#:~:text=Accessing%20Type%20Parameters%20at%20Runtime,classes%2C%20functions%2C%20and%20type%20aliases) PEP 695 – Type Parameter Syntax | peps.python.org

<https://peps.python.org/pep-0695/>