

Enhancing Decorators with Type Annotations: Techniques and Best Practices

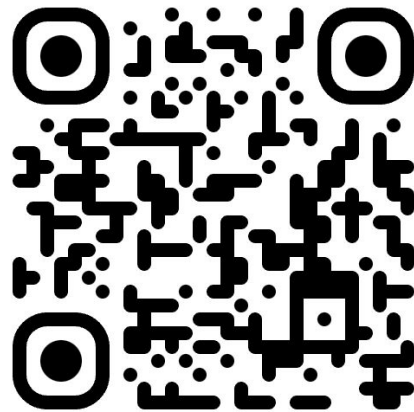
Koudai Aono

Goal

- **Understand Complex Decorators with Type Annotations**
- **Prevent and Detect Bugs Early**
- **Best Practices for Typing Decorators**

About Me

- Occupation: Software Engineer at Tractable, based in Tokyo
- OSS Contributions:
 - Developing PyCharm plugins for Pydantic and Ruff
 - Creating “datamodel-code-generator”, a code generator used by Pydantic and in dataclasses models.
- GitHub: <https://github.com/koxudaxi>

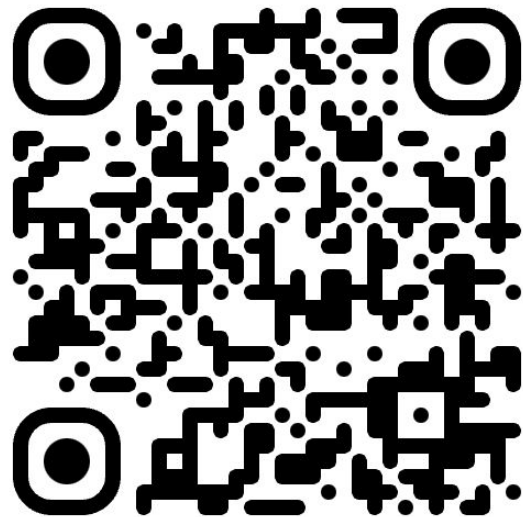


How I Came Up With This Talk

- Motivation: Type hinting is valuable but challenging with decorators.
- Key Questions:
 1. How to define Callable types without ellipsis?
 2. How to manage functions with flexible arguments?
 3. Where to find the best practices?
- Goal: This talk will address these issues and enhance decorators with type annotations.

Structure for Each Section

1. Introduce the Feature
2. Show Sample Code
3. Identify Problems in Current Code
4. Apply the Feature to Fix the Code
5. Recap



Code repo URL of Talk

I'll explain these concepts by creating a logger decorator for HTTP clients.

Agenda

1. Basics of Decorators
2. `typing.Protocol`
3. `typing.ParamSpec`
4. `typing.Concatenate`
5. Type Parameter Syntax in Python 3.12
6. Practical Applications of Decorators
7. `typing.TypeVarTuple`

1. Basics of Decorators

- A decorator is a syntax in Python for higher-order functions.
- Processing can be added before or after a function call.

[PEP 318 – Decorators for Functions and Methods](#)

Decorator

```
print("Before running func")
```

```
func(*args, **kwargs)
```

```
print("After running func")
```

← **Decorated
function**

Decorator

```
print("Before running func")
```

```
func(*args, **kwargs)
```

```
print("After running func")
```

← **Decorated
function**

Decorator

```
print("Before running func")
```

```
func(*args, **kwargs)
```

```
print("After running func")
```

← **Decorated
function**

Decorator

```
print("Before running func")
```

```
func(*args, **kwargs)
```

```
print("After running func")
```

← **Decorated
function**

Let's create a logging decorator for HTTP client

Requirements:

- **log before and after the execution of a function.**
- **The decorator should be designed for HTTP client functions**
- **Use a custom remote logger class for logging**

Decorator

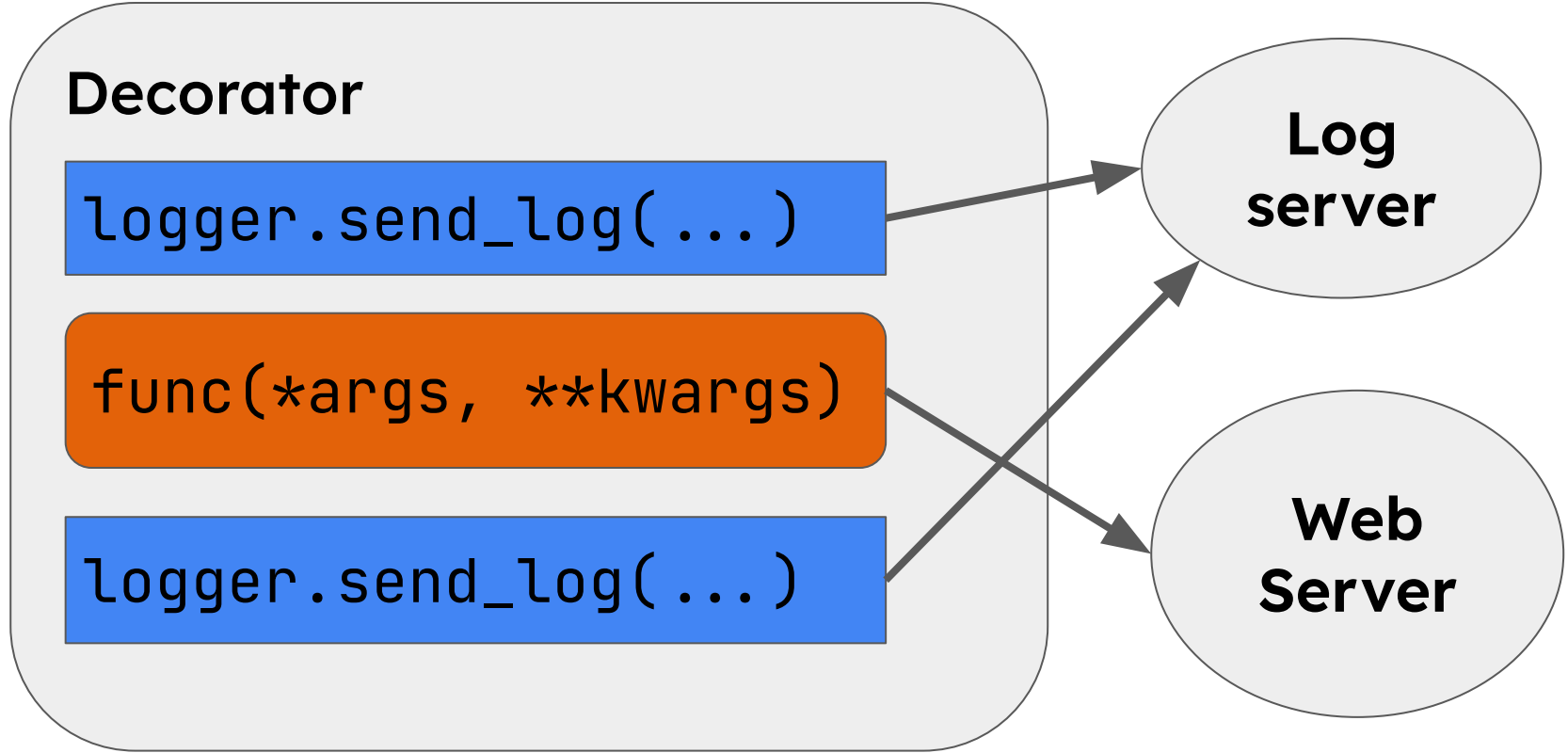
```
logger.send_log(...)
```

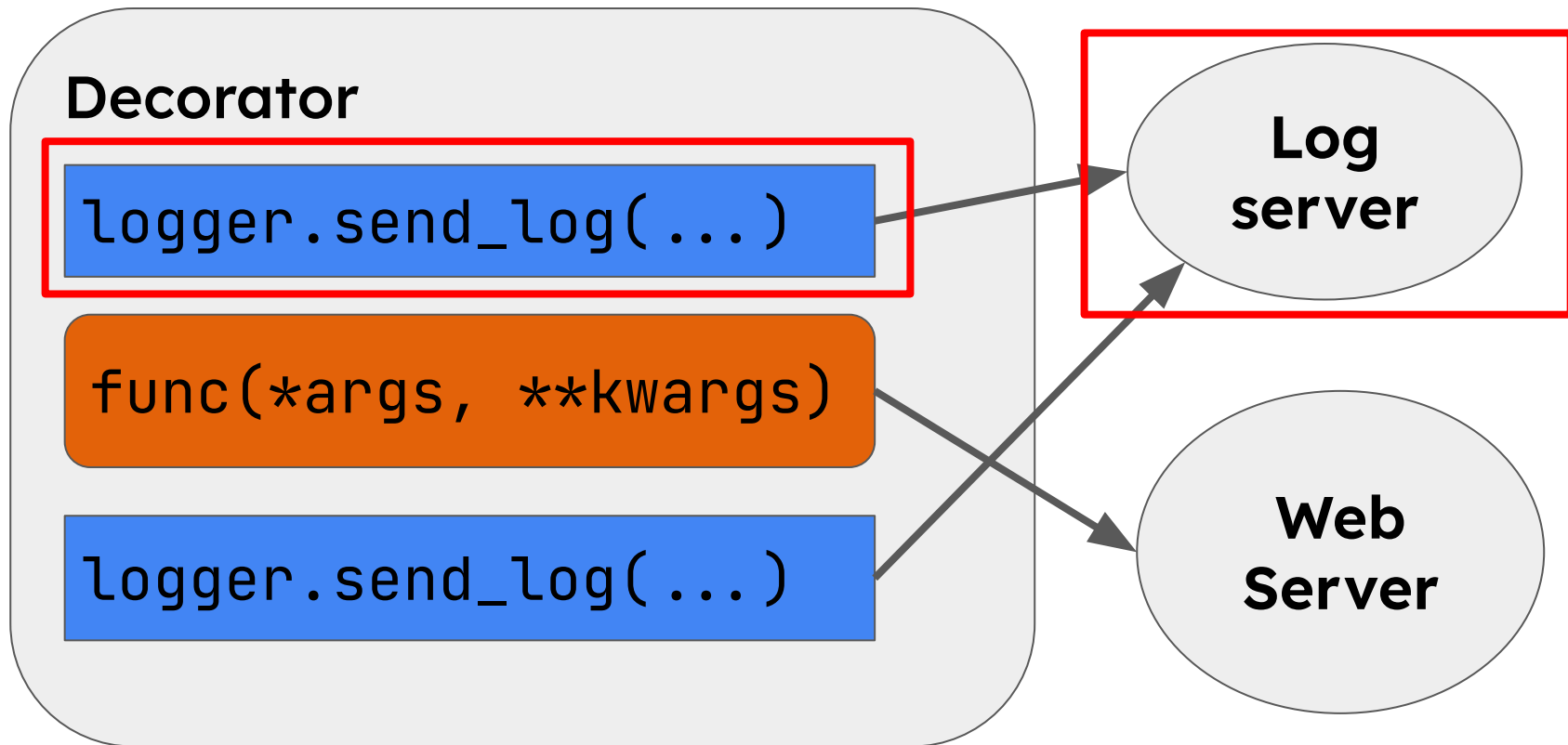
```
func(*args, **kwargs)
```

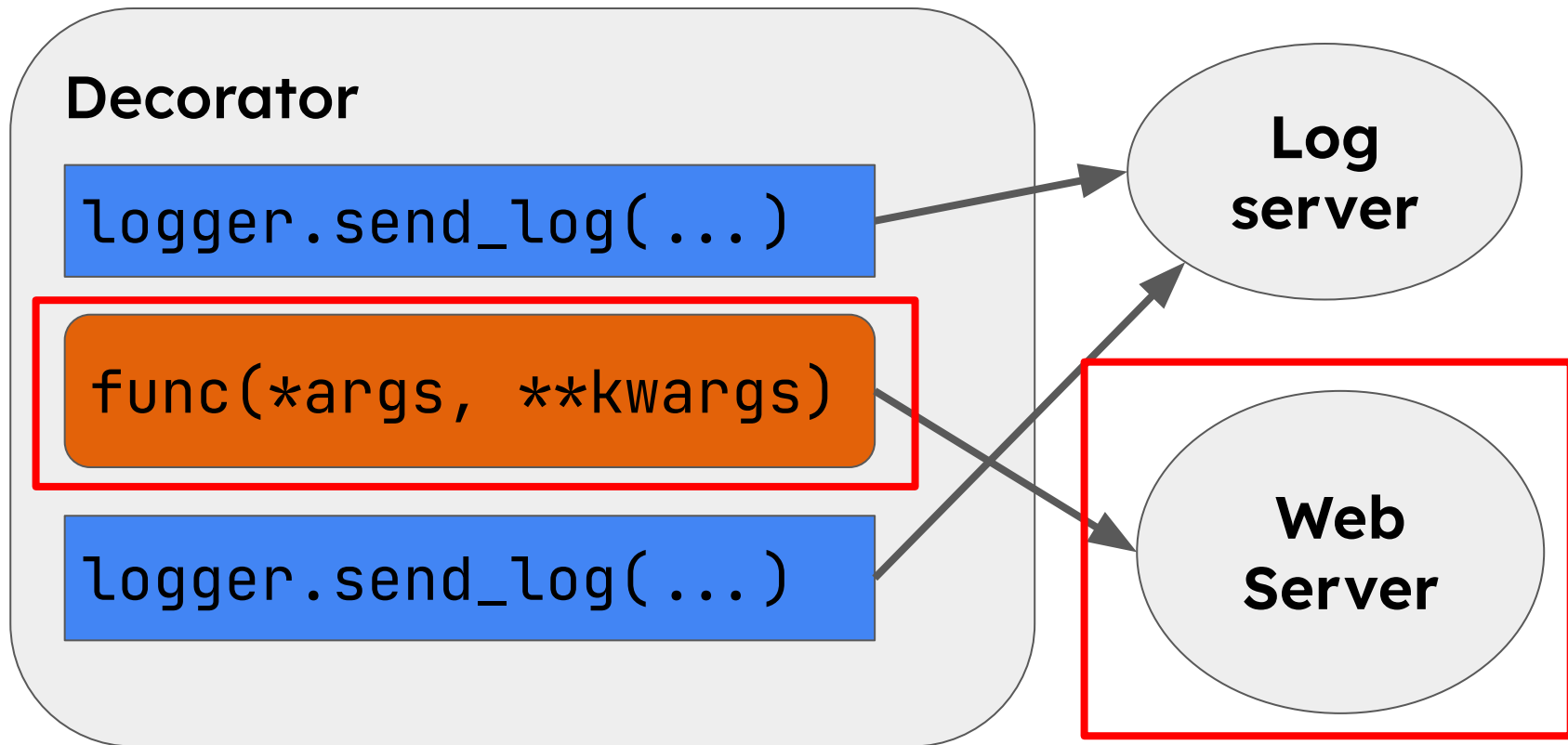
```
logger.send_log(...)
```

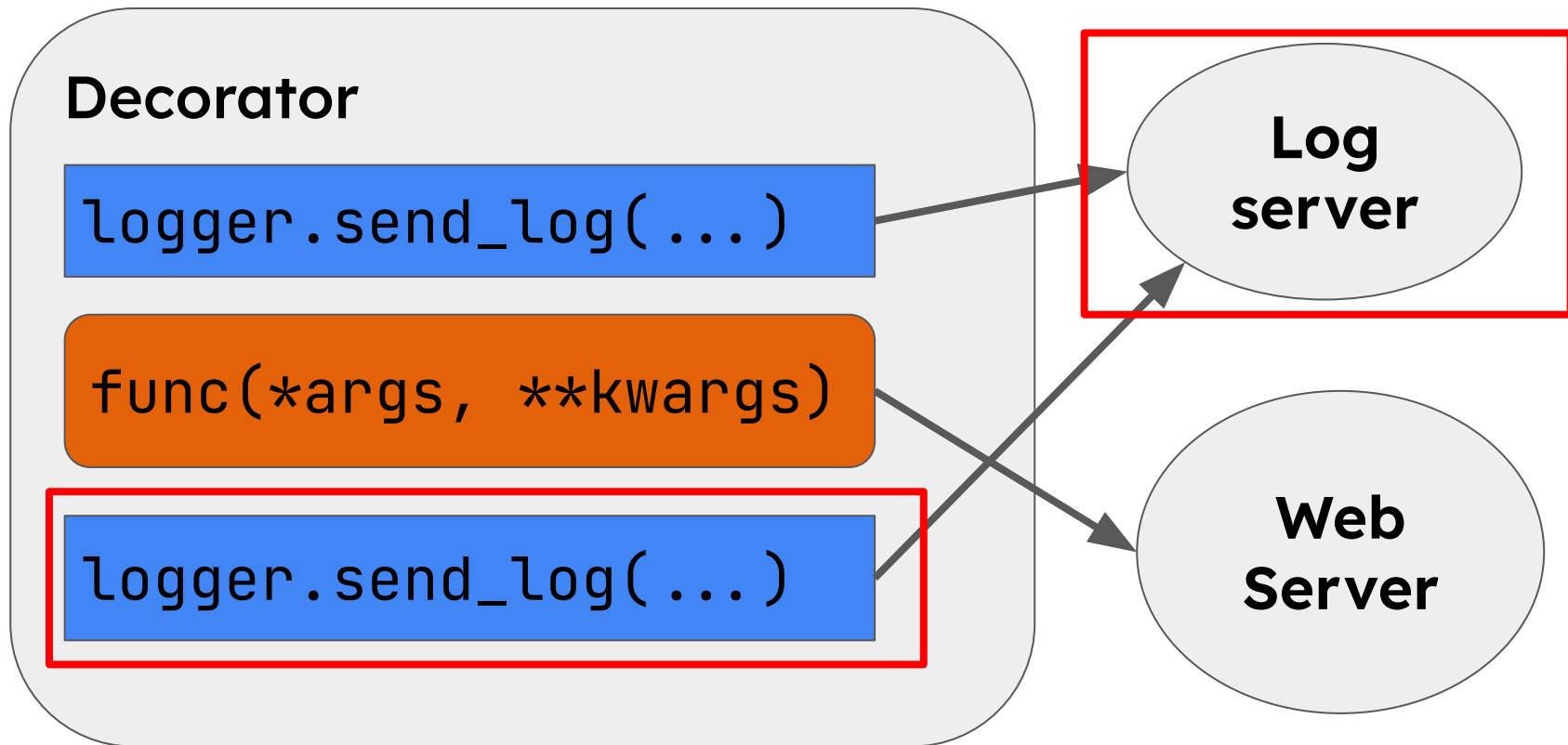
**Log
server**

**Web
Server**









A HTTP client function to apply the decorator

```
import requests

def call_url(url: str) → Any:
    return requests.get(url)

call_url('https://www.example.com/')
```

Here is an example RemoteLogger class to send log

```
class RemoteLogger:
    def __init__(self, name: str, group: str,
level: int):
        ...

    def send_log(self, *args, **kwargs):
        ...
```

Here is an example RemoteLogger class to send log

```
class RemoteLogger:
    def __init__(self, name: str, group: str,
level: int):
        ...

    def send_log(self, *args, **kwargs):
        ...
```

Here is an example RemoteLogger class to send log

```
class RemoteLogger:
    def __init__(self, name: str, group: str,
level: int):
        ...

    def send_log(self, *args, **kwargs):
        ...
```

There is an example RemoteLogger class to send log

```
logger = RemoteLogger(  
    name='test', group='admin', level=0  
)  
logger.send_log(  
    message='hello', user_id=1  
)
```

```
>> {"name": "test", "group": "admin",  
    "level": 0, "message": "hello", "user_id": 1}
```

There is an example RemoteLogger class to send log

```
logger = RemoteLogger(  
    name='test', group='admin', level=0  
)
```

```
logger.send_log(  
    message='hello', user_id=1  
)
```

```
>> {"name": "test", "group": "admin",  
    "level": 0, "message": "hello", "user_id": 1}
```

There is an example RemoteLogger class to send log

```
logger = RemoteLogger(  
    name='test', group='admin', level=0  
)
```

```
logger.send_log(  
    message='hello', user_id=1  
)
```

```
>> {"name": "test", "group": "admin",  
    "level": 0, "message": "hello", "user_id": 1}
```

There is an example RemoteLogger class to send log

```
logger = RemoteLogger(  
    name='test', group='admin', level=0  
)  
logger.send_log(  
    message='hello', user_id=1  
)
```

```
>> {"name": "test", "group": "admin",  
    "level": 0, "message": "hello", "user_id": 1}
```


Logging decorator with RemoteLogger

```
def add_logging(group, level=0):  
    def inner(func):  
        logger = RemoteLogger(  
            func.__name__, group, level)  
        def wrapper(*args, **kwargs):  
            logger.send_log(args=args, kwargs=kwargs)  
            result = func(*args, **kwargs)  
            logger.send_log(result=result)  
            return result  
        return wrapper  
    return inner
```

Logging decorator with RemoteLogger

```
def add_logging(group, level=0):  
    def inner(func):  
        logger = RemoteLogger(  
            func.__name__, group, level)  
        def wrapper(*args, **kwargs):  
            logger.send_log(args=args, kwargs=kwargs)  
            result = func(*args, **kwargs)  
            logger.send_log(result=result)  
            return result  
        return wrapper  
    return inner
```

Logging decorator with RemoteLogger

```
def add_logging(group, level=0):  
    def inner(func):  
        logger = RemoteLogger(  
            func.__name__, group, level)  
        def wrapper(*args, **kwargs):  
            logger.send_log(args=args, kwargs=kwargs)  
            result = func(*args, **kwargs)  
            logger.send_log(result=result)  
            return result  
        return wrapper  
    return inner
```

Apply the decorator to the function

```
@add_logging('http client', 0)
def call_url(url: str) → Any:
    return requests.get(url)
call_url('https://www.example.com/')
```

Apply the decorator to the function

```
@add_logging('http client', 0)
def call_url(url: str) → Any:
    return requests.get(url)
call_url('https://www.example.com/')
```

```
>> {"name": "call_url", "group": "admin",
    "level": 0, "args":
    ["https://www.example.com/"], "kwargs": {}}
>> {"name": "call_url", "group": "admin",
    "level": 0, "result": 200}
```

PEP 3102 – Keyword-Only Arguments

Enforce keyword arguments after the asterisk.

```
def compare(a, b, *, key=None):  
    ...  
compare(1, 2, key=lambda x: x) # OK  
compare(1, 2, lambda x: x) # Throws TypeError
```

PEP 3102 – Keyword-Only Arguments

Enforce keyword arguments after the asterisk.

```
def compare(a, b, *, key=None):
```

```
    ...
```

```
compare(1, 2, key=lambda x: x) # OK
```

```
compare(1, 2, lambda x: x) # Throws TypeError
```

PEP 3102 – Keyword-Only Arguments

Enforce keyword arguments after the asterisk.

```
def compare(a, b, *, key=None):
```

```
    ...
```

```
compare(1, 2, key=lambda x: x) # OK
```

```
compare(1, 2, lambda x: x) # Throws TypeError
```


PEP 3102 – Keyword-Only Arguments

Enforce keyword arguments after the asterisk.

```
def compare(a, b, *, key=None):  
    ...  
compare(1, 2, key=lambda x: x) # OK  
compare(1, 2, lambda x: x) # Throws TypeError
```

What is the problem with this decorator?

```
def add_logging(group: str, level: int = 0):  
    def inner(func: Callable[..., Any] ) →  
Callable[[Callable[..., Any]], Callable[...,  
Any]]:  
        logger = RemoteLogger(  
            func.__name__, group, level)  
  
        def wrapper(*args, **kwargs):  
            logger.send_log(  
                args=args, kwargs=kwargs)
```

The decorator has no argument restrictions.

```
def add_logging(group: str, level: int = 0):  
    def inner(func: Callable[..., Any]) →  
        Callable[[Callable[..., Any]], Callable[...,  
            Any]]:  
        logger = RemoteLogger(  
            func.__name__, group, level)  
  
        def wrapper(*args, **kwargs):  
            logger.send_log(  
                args=args, kwargs=kwargs)
```

Guess argument types and order without the definition?

```
def add_logging(...
```

```
@add_logging(🤔🤔🤔🤔🤔)
```

```
def call_url(url: str) → int:
```

```
    return requests.get(url).status_code
```

If not specified with kwargs, an error will occur

```
def add_logging(group: str, *, level: int =  
0  
) → Callable[[Callable[..., Any]],  
Callable[..., Any]]:  
...
```

```
@add_logging('http client', 0)  
def call_url(url: str) → int:  
    return requests.get(url).status_code
```

If not specified with kwargs, an error will occur

```
def add_logging(group: str, *, level: int =  
0
```

) →

Calla

...

```
@add_logging('http client', 0)
```

```
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

```
TypeError: add_logging() takes 1 positional  
argument but 2 were given
```

```
@add_logging('http client', 0)
```

```
def call_url(url: str) → int:
```

```
    return requests.get(url).status_code
```

If not specified with kwargs, an error will occur

```
def add_logging(group: str, *, level: int =  
0
```

```
) →
```

```
Callable
```

```
...
```

PASSED



```
@add_logging('http client', level=0)
```

```
def call_url(url: str) → Any:
```

```
    return requests.get(url)
```

1. Recap: Basics of Decorators

Enforce Keyword-Only Arguments with *:

- Ensures arguments after * must be keyword arguments, clarifying usage and preventing errors.

```
def compare(a, b, *, key=None):
```

```
    ...
```


2. typing.Protocol

- **Interface Definition:** Provides a way to define expected methods and properties for classes.
- **Type Safety:** Helps catch implementation errors by enforcing method signatures.

[PEP 544 – Protocols: Structural subtyping \(static duck typing\)](#)

Define class interface type

```
from typing import Protocol

class ResponseLike(Protocol):
    status_code: int

    def json(self) → dict:
        ...
```

Define class interface type

```
from typing import Protocol

class ResponseLike(Protocol):
    status_code: int

    def json(self) → dict:
        ...

>> response: ResponseLike =
requests.get('https://example.com')
>> response.json()
>> response.status_code
```

Define function interface type

```
from typing import Protocol

class ClientGetFunction(Protocol):
    def __call__(self, url: str, timeout: float)
    → ResponseLike:
        ...
```

Define function interface type

```
from typing import Protocol

class ClientGetFunction(Protocol):
    def __call__(self, url: str, timeout: float)
→ ResponseLike:
    ...

>>def call_url(url: str, timeout: float) →
ResponseLike:
>>    return requests.get(url, timeout=timeout)
>>func: ClientGetFunction = call_url    # OK
```

Callable vs Protocol

Callable

```
from typing import  
Callable, TypeAlias  
  
GetStatus: TypeAlias =  
Callable[[str, float],  
ResponseLike]
```

Callable vs Protocol

Callable

```
from typing import  
Callable, TypeAlias
```

```
GetStatus: TypeAlias =  
Callable[[str, float],  
ResponseLike]
```

Protocol

```
from typing import  
Protocol
```

```
class GetStatus(Protocol):  
    def __call__(  
        self,  
        url: str,  
        timeout: float = 5  
    ) → ResponseLike:  
        ...
```

requests vs httpx

requests

```
from requests import  
Response  
  
response: Response =  
requests.get(  
'https://www.python.org/')  
assert  
response.status_code ==  
200
```

httpx

```
from httpx import Response  
  
response: Response =  
httpx.get(  
'https://www.python.org/')  
assert  
response.status_code ==  
200
```


Different response types, same attribute

requests

```
from requests import  
Response
```

```
response: Response =  
requests.get(  
'https://www.python.org/')  
assert  
response.status_code ==  
200
```

httpx

```
from httpx import Response
```

```
response: Response =  
httpx.get(  
'https://www.python.org/')  
assert  
response.status_code ==  
200
```

Different response types, same attribute

requests

```
from requests import  
Response  
  
response: Response =  
requests.get(  
    'https://www.python.org/')  
assert  
response.status_code ==  
200
```

httpx

```
from httpx import Response  
  
response: Response =  
httpx.get(  
    'https://www.python.org/')  
assert  
response.status_code ==  
200
```

Different response types, same attribute

requests

```
from typing import Protocol  
import requests
```

```
class ResponseLike(Protocol):  
    status_code: int
```

httpx

```
from typing import Protocol  
import httpx
```

```
class ResponseLike(Protocol):  
    status_code: int
```

Different response types, same attribute

requests

```
from typing import Protocol
import requests

class ResponseLike(Protocol):
    status_code: int

response: ResponseLike =
requests.get('https://www.pyt
hon.org/')
assert response.status_code
== 200
```

httpx

```
from typing import Protocol
import httpx

class ResponseLike(Protocol):
    status_code: int

response: ResponseLike =
httpx.get('https://www.python
.org/')
assert response.status_code
== 200
```

ResponseLike supports httpx and requests' response

```
from typing import Any, Protocol
```

```
class ResponseLike(Protocol):  
    status_code: int
```

```
...
```

```
    def wrapper(  
        *args: Any, **kwargs: Any  
    ) → ResponseLike:
```

```
...
```

2. Recap: typing.Protocol

Benefits:

- **Interoperability:** Works with multiple libraries (requests, httpx).
- **Type Safety:** Ensures response objects have a `status_code`.

```
class ResponseLike(Protocol):  
    status_code: int
```

3. `typing.ParamSpec`

Key Benefits:

- **Flexible Signature Handling:** Capture the signature of functions to write decorators that are more flexible.
- **Type Safety:** Ensure that functions used with decorators maintain their original type annotations.

[PEP 612 – Parameter Specification Variables](#)

typing.ParamSpec example from PEP612

```
from typing import Awaitable, Callable, ParamSpec, TypeVar

P = ParamSpec("P")
R = TypeVar("R")

def add_logging(f: Callable[P, R]) → Callable[P, Awaitable[R]]:
    async def inner(*args: P.args, **kwargs: P.kwargs) → R:
        await log_to_database()
        return f(*args, **kwargs)
    return inner

@add_logging
def takes_int_str(x: int, y: str) → int:
    return x + 7

await takes_int_str(1, "A") # Accepted
await takes_int_str("B", 2) # Correctly rejected by the type checker
```


typing.ParamSpec example from PEP612

```
from typing import Awaitable, Callable, ParamSpec, TypeVar
```

```
P = ParamSpec("P")
```

```
R = TypeVar("R")
```

```
def add_logging(f: Callable[P, R]) → Callable[P, Awaitable[R]]:
```

```
    async def inner(*args: P.args, **kwargs: P.kwargs) → R:
```

```
        await log_to_database()
```

```
        return f(*args, **kwargs)
```

```
    return inner
```

typing.ParamSpec example from PEP612

```
from typing import Awaitable, Callable, ParamSpec, TypeVar

P = ParamSpec("P")
R = TypeVar("R")

def add_logging(f: Callable[P, R]) → Callable[P,
Awaitable[R]]:
    async def inner(*args: P.args, **kwargs: P.kwargs) →
R:
        await log_to_database()
        return f(*args, **kwargs)
    return inner
```

Keyword arguments is defined as Any

```
from typing import Callable, ParamSpec, Protocol

P = ParamSpec("P")

def add_logging(
    group: str, *, level: int = 0
) → Callable[[Callable[... , ResponseLike], Callable[... ,
ResponseLike]]:
    def inner(func: Callable[... , ResponseLike]) →
Callable[... , ResponseLike]:
        """
        def wrapper(*args: Any, **kwargs: Any) →
ResponseLike:
            ...
```

ParamSpec provides P.kwargs for kwargs

```
from typing import Callable, ParamSpec, Protocol

P = ParamSpec("P")

def add_logging(
    group: str, *, level: int = 0
) → Callable[[Callable[P, ResponseLike]], Callable[P,
ResponseLike]]:
    def inner(func: Callable[P, ResponseLike]) →
Callable[P, ResponseLike]:
        ...
        def wrapper(*args: P.args, **kwargs: P.kwargs) →
ResponseLike:
            ...
```

3. Recap: typing.ParamSpec

- Flexible Signature Handling: Capture function signatures for flexible decorators.
- Type Safety: Ensure functions maintain their original type annotations.

```
P = ParamSpec("P")
def inner(func: Callable[P, ResponseLike]) →
    Callable[P, ResponseLike]:
    func(*args: P.args, **kwargs: P.kwargs)
→ ResponseLike:
```

4. typing.Concatenate

This feature is useful when:

- A decorator needs to adjust the function's signature.
- Adding or removing specific arguments while keeping the rest of the signature intact.

[PEP 612 – Parameter Specification Variables](#)

typing.Concatenate example from PEP612

```
from typing import Concatenate

def with_request(f: Callable[Concatenate[Request, P], R]) → Callable[P, R]:
    def inner(*args: P.args, **kwargs: P.kwargs) → R:
        return f(Request(), *args, **kwargs)
    return inner

@with_request
def takes_int_str(request: Request, x: int, y: str) → int:
    # use request
    return x + 7

takes_int_str(1, "A") # Accepted
takes_int_str("B", 2) # Correctly rejected by the type checker
```

typing.Concatenate example from PEP612

```
def with_request(f: Callable[Concatenate[Request, P],  
R]) → Callable[P, R]:  
    def inner(*args: P.args, **kwargs: P.kwargs) → R:  
        return f(Request(), *args, **kwargs)  
  
    ...  
@with_request  
def takes_int_str(request: Request, x: int, y: str) →  
int:  
    # use request  
  
    ...  
takes_int_str(1, "A") # Accepted
```


typing.Concatenate example from PEP612

```
def with_request(f: Callable[Concatenate[Request, P],  
R]) → Callable[P, R]:  
    def inner(*args: P.args, **kwargs: P.kwargs) → R:  
        return f(Request(), *args, **kwargs)  
    ...  
@with_request  
def takes_int_str(request: Request, x: int, y: str) →  
int:  
    # use request  
    ...  
takes_int_str(1, "A") # Accepted
```

typing.Concatenate example from PEP612

```
def with_request(f: Callable[Concatenate[Request, P],  
R]) → Callable[P, R]:  
    def inner(*args: P.args, **kwargs: P.kwargs) → R:  
        return f(Request(), *args, **kwargs)  
  
    ...  
@with_request  
def takes_int_str(request: Request, x: int, y: str) →  
int:  
    # use request  
  
    ...  
takes_int_str(1, "A") # Accepted
```

typing.Concatenate example from PEP612

```
def with_request(f: Callable[Concatenate[Request, P],  
R]) → Callable[P, R]:  
    def inner(*args: P.args, **kwargs: P.kwargs) → R:  
        return f(Request(), *args, **kwargs)  
  
    ...  
@with_request  
def takes_int_str(request: Request, x: int, y: str) →  
int:  
    # use request  
  
    ...  
takes_int_str(1, "A") # Accepted
```

Definition of function

```
@with_request  
def f(request, arg1, arg2):
```

Decorator

```
f(Request(), args1, args2)
```

Function call

```
f("args1", "args2")
```

**Inject an
object as
first
argument**



The diagram consists of two black arrows originating from the text 'Inject an object as first argument'. The first arrow points to the 'Request()' argument in the decorator call 'f(Request(), args1, args2)'. The second arrow points to the 'request' parameter in the function definition 'def f(request, arg1, arg2):'.

Definition of function

```
@with_request  
def f(request, arg1, arg2):
```

Decorator

```
f(Request(), args1, args2)
```

Function call

```
f("args1", "args2")
```

**Inject an
object as
first
argument**



The diagram consists of two arrows originating from the text 'Inject an object as first argument'. One arrow points to the `Request()` argument in the decorator call `f(Request(), args1, args2)`. The other arrow points to the `request` parameter in the function definition `def f(request, arg1, arg2):`.

**Call function
without
request**



The diagram shows a single arrow pointing from the text 'Call function without request' to the function call `f("args1", "args2")` in the 'Function call' section. The function call is highlighted with a red border.

Want to use the logger inside the function

```
@add_logging('http client', level=0)
def call_url(logger: RemoteLogger, url:
str, timeout: int = 5) → ResponseLike:

    logger.send_log(message='in call_url')
    return requests.get(url,
timeout=timeout)

call_url('https://www.example.com/')
```

Want to use the logger inside the function

```
@add_logging('http client', level=0)
def call_url(logger: RemoteLogger, url:
str, timeout: int = 5) → ResponseLike:

    logger.send_log(message='in call_url')
    return requests.get(url,
timeout=timeout)

call_url('https://www.example.com/')
```

Inject logger object to first argument in function

```
@add_logging('http client', level=0)
def call_url(logger: RemoteLogger, url:
str, timeout: int = 5) → ResponseLike:

    logger.send_log(message='in call_url')
    return requests.get(url,
timeout=timeout)

call_url('https://www.example.com/')
```


Apply Concatenate

```
def add_logging(group: str, *, level: int=0) →  
    Callable[[Callable[Concatenate[RemoteLogger, P], ResponseLike]],  
    Callable[[Concatenate[P]], ResponseLike]]:  
    def inner(func: Callable[Concatenate[RemoteLogger, P], ResponseLike]) →  
        Callable[[Concatenate[P]], ResponseLike]:  
        logger = RemoteLogger(func.__name__, group, level)  
  
        def wrapper(*args: P.args, **kwargs: P.kwargs) → ResponseLike:  
            logger.send_log(args=args, kwargs=kwargs)  
            result = func(logger, *args, **kwargs)  
            logger.send_log(result=result)  
            return result  
        return wrapper  
    return inner  
  
@add_logging('http client', level=0)  
def call_url(logger: RemoteLogger, url: str, timeout: int = 5) → ResponseLike:  
    logger.send_log(message='in call_url')  
    return requests.get(url, timeout=timeout)
```

```
def inner(func: Callable[Concatenate[RemoteLogger,  
P], ResponseLike]) → Callable[[Concatenate[P]],  
ResponseLike]:  
    logger = RemoteLogger(func.__name__, group,  
level)  
  
    def wrapper(*args: P.args, **kwargs: P.kwargs)  
→ ResponseLike:  
        logger.send_log(args=args, kwargs=kwargs)  
        result = func(logger, *args, **kwargs)  
  
    ...  
@add_logging('http client', level=0)  
def call_url(logger: RemoteLogger, url: str, timeout:  
int = 5) → ResponseLike:
```

```
def inner(func: Callable[Concatenate[RemoteLogger,  
P], ResponseLike]) → Callable[[Concatenate[P]],  
ResponseLike]:  
    logger = RemoteLogger(func.__name__, group,  
level)  
  
    def wrapper(*args: P.args, **kwargs: P.kwargs)  
→ ResponseLike:  
        logger.send_log(args=args, kwargs=kwargs)  
        result = func(logger, *args, **kwargs)  
    ...  
@add_logging('http client', level=0)  
def call_url(logger: RemoteLogger, url: str, timeout:  
int = 5) → ResponseLike:
```

```
def inner(func: Callable[Concatenate[RemoteLogger,  
P], ResponseLike]) → Callable[[Concatenate[P]],  
ResponseLike]:  
    logger = RemoteLogger(func.__name__, group,  
level)  
  
    def wrapper(*args: P.args, **kwargs: P.kwargs)  
→ ResponseLike:  
        logger.send_log(args=args, kwargs=kwargs)  
        result = func(logger, *args, **kwargs)  
  
    ...  
@add_logging('http client', level=0)  
def call_url(logger: RemoteLogger, url: str, timeout:  
int = 5) → ResponseLike:
```

4. Recap: typing.Concatenate

- **Flexible Signature Manipulation:** Enables adding extra parameters to functions while preserving the original signature.
- **Type Safety:** Ensures the modified signature remains type-safe, including additional parameters.

```
from collections.abc import Callable
from typing import Concatenate
Callable[Concatenate[RemoteLogger, P], R]
```

6. Type Parameter Syntax in Python 3.12

1. Consistent Syntax:

- Introduces a standard way to declare type parameters using class and def.

2. Clearer Code:

- Improves readability and consistency by avoiding the use of complex type hints.

[PEP 695 – Type Parameter Syntax](#)

Have to import TypeAlias when define new Type

Python 3.11

```
from typing import  
TypeAlias
```

```
Url: TypeAlias = str
```

```
def call_url(url: Url  
    ) → Any:  
    ...
```

Python 3.12 provides the new keyword “type”

Python 3.11

```
from typing import  
TypeAlias  
  
Url: TypeAlias = str  
  
def call_url(url: Url  
    ) → Any:  
    ...
```

Python 3.12

```
type Url = str  
  
def call_url(url: Url  
    ) → Any:  
    ...
```


Have to define “T” with `Typing.TypeVar`

Python 3.11

```
from typing import  
TypeVar
```

```
T = TypeVar("T")
```

```
def multiply(x: T, y:  
int  
    ) → T:  
    ...
```

New Syntax reduces steps to use “T”

Python 3.11

```
from typing import  
TypeVar  
  
T = TypeVar("T")  
  
def multiply(x: T, y:  
int  
    ) → T:  
    ...
```

Python 3.12

```
def multiply[T](x: T,  
y: int  
    ) → T:  
    ...
```

Apply Type Parameter Syntax in Python 3.12

```
type LogFunc[** P, R] =  
Callable[Concatenate[RemoteLogger, P], R]
```

```
def add_logging[** P](  
    group: str, *, level: int = 0  
) → Callable[[LogFunc[P, ResponseLike]],  
    Callable[P, ResponseLike]]:  
    ...
```

Apply Type Parameter Syntax in Python 3.12

```
type LogFunc[** P, R] =  
Callable[Concatenate[RemoteLogger, P], R]
```

```
def add_logging[** P](  
    group: str, *, level: int = 0  
) → Callable[[LogFunc[P, ResponseLike]],  
    Callable[P, ResponseLike]]:  
    ...
```

6. Recap: Type Parameter Syntax in Python 3.12

- Consistent Syntax:
 - Provides a standardized way to declare type parameters in classes and functions.
- Clearer Code:
 - Enhances readability and consistency by eliminating complex type hints.

```
type LogFunc[** P, R] =  
Callable[Concatenate[RemoteLogger, P], R]  
def add_logging[** P](...
```

7. Practical Applications of Decorators

Advanced Decorator Example using ParamSpec, Concatenate, and Protocol

Requirements:

- Log the URL, status code, and function name using modern Python features for type safety.

Here is the code with all the steps applied so far

```
from collections.abc import Callable
from typing import Concatenate, Protocol

import httpx
import requests

from .remote_logger import RemoteLogger

class ResponseLike(Protocol):
    status_code: int

type LogFunc[** P, R] = Callable[Concatenate[RemoteLogger, P], R]

def add_logging[** P](
    group: str, *, level: int = 0
) → Callable[[LogFunc[P, ResponseLike]], Callable[P, ResponseLike]]:
    def inner(func: LogFunc[P, ResponseLike]) → Callable[P, ResponseLike]:
        logger = RemoteLogger(func.__name__, group, level)

        def wrapper(*args: P.args, **kwargs: P.kwargs) → ResponseLike:
            logger.send_log(args=args, kwargs=kwargs)
            result = func(logger, *args, **kwargs)
            logger.send_log(status_code=result.status_code)
            return result
        return wrapper
    return inner
```

```
@add_logging('http client', level=0)
def download_with_request(
    logger: RemoteLogger, url: str, timeout: float = 5
) → ResponseLike:
    logger.send_log(message='in download_with_request')
    return requests.get(url, timeout=timeout)

@add_logging('http client', level=0)
def download_with_httpx(
    logger: RemoteLogger, url: str, timeout: float = 5
) → ResponseLike:
    logger.send_log(message='in download_with_httpx')
    return httpx.get(url, timeout=timeout)

download_with_request('https://examples.com/', timeout=10)
download_with_httpx('https://examples.com/', timeout=10)
```

```
from collections.abc import Callable
from typing import Concatenate, Protocol
```

```
...
```

```
class ResponseLike(Protocol):
    status_code: int
```

```
type LogFunc[** P, R] =
    Callable[Concatenate[RemoteLogger, P], R]
```



```
from collections.abc import Callable
from typing import Concatenate, Protocol
```

```
...
```

```
class ResponseLike(Protocol):
    status_code: int
```

```
type LogFunc[** P, R] =
    Callable[Concatenate[RemoteLogger, P], R]
```

```
def add_logging[** P](  
    group: str, *, level: int = 0  
) → Callable[[LogFunc[P, ResponseLike]],  
    Callable[P, ResponseLike]]:  
  
    def inner(  
        func: LogFunc[P, ResponseLike]  
    ) → Callable[P, ResponseLike]:  
        ...
```

```
def add_logging[** P](  
    group: str, *, level: int = 0  
) → Callable[[LogFunc[P, ResponseLike]],  
    Callable[P, ResponseLike]]:  
  
    def inner(  
        func: LogFunc[P, ResponseLike]  
    ) → Callable[P, ResponseLike]:  
        ...
```

```
def wrapper(  
    *args: P.args, **kwargs: P.kwargs  
    ) → ResponseLike:  
    logger.send_log(  
        args=args, kwargs=kwargs  
    )  
    result = func(  
        logger, *args, **kwargs  
    )  
    logger.send_log(  
        status_code=result.status_code  
    )
```

```
def wrapper(  
    *args: P.args, **kwargs: P.kwargs  
    ) → ResponseLike:  
    logger.send_log(  
        args=args, kwargs=kwargs  
    )  
    result = func(  
        logger, *args, **kwargs  
    )  
    logger.send_log(  
        status_code=result.status_code  
    )
```

```
@add_logging('http client', level=0)
def download_with_httpx(
    logger: RemoteLogger,
    url: str,
    timeout: float = 5
) → ResponseLike:
    logger.send_log(message='in the
function)
    return httpx.get(url, timeout=timeout)
download_with_httpx('https://examples.com/'
)
```

```
@add_logging('http client', level=0)
def download_with_httpx(
    logger: RemoteLogger,
    url: str,
    timeout: float = 5
) → ResponseLike:
    logger.send_log(message='in the
function)
    return httpx.get(url, timeout=timeout)
download_with_httpx('https://examples.com/'
)
```

7. `typing.TypeVarTuple`

Benefits:

- Safely handle functions with variable-length arguments.
- Enable advanced generic usage for decorators and classes.

[PEP 646 – Variadic Generics](#)


```
from typing import TypeVar, TypeVarTuple
```

```
T = TypeVar("T")
```

```
Ts = TypeVarTuple("Ts")
```

```
def move_first_element_to_last(
```

```
    tup: tuple[T, *Ts]
```

```
) → tuple[*Ts, T]:
```

```
    return (*tup[1:], tup[0])
```

```
move_first_element_to_last((1, 2, 3))
```

```
#
```

```
↑
```

```
↑
```

```
↑
```

```
#
```

```
T
```

```
*Ts
```

```
from typing import TypeVar, TypeVarTuple
```

```
T = TypeVar("T")
```

```
Ts = TypeVarTuple("Ts")
```

```
def move_first_element_to_last(
```

```
    tup: tuple[T, *Ts]
```

```
) → tuple[*Ts, T]:
```

```
    return (*tup[1:], tup[0])
```

```
move_first_element_to_last((1, 2, 3))
```

```
#
```

```
↑
```

```
↑
```

```
↑
```

```
#
```

```
T
```

```
*Ts
```

```
from typing import TypeVar, TypeVarTuple
```

```
T = TypeVar("T")
```

```
Ts = TypeVarTuple("Ts")
```

```
def move_first_element_to_last(
```

```
    tup: tuple[T, *Ts]
```

```
) → tuple[*Ts, T]:
```

```
    return (*tup[1:], tup[0])
```

```
move_first_element_to_last((1, 2, 3))
```

```
#
```

```
↑
```

```
↑
```

```
↑
```

```
#
```

```
T
```

```
*Ts
```

```
from typing import TypeVar, TypeVarTuple
```

```
T = TypeVar("T")
```

```
Ts = TypeVarTuple("Ts")
```

```
def move_first_element_to_last(
```

```
    tup: tuple[T, *Ts]
```

```
) → tuple[*Ts, T]:
```

```
    return (*tup[1:], tup[0])
```

```
move_first_element_to_last((1, 2, 3))
```

```
#
```

```
↑
```

```
↑
```

```
↑
```

```
#
```

```
T
```

```
*Ts
```

```
from typing import TypeVar, TypeVarTuple
```

```
T = TypeVar("T")
```

```
Ts = TypeVarTuple("Ts")
```

```
def move_first_element_to_last(
```

```
    tup: tuple[T, *Ts]
```

```
) → tuple[*Ts, T]:
```

```
    return (*tup[1:], tup[0])
```

```
move_first_element_to_last((1, 2, 3))
```

```
#                                     ↑  ↑  ↑  
#                                     T  *Ts
```

7. Recap: `typing.TypeVarTuple`

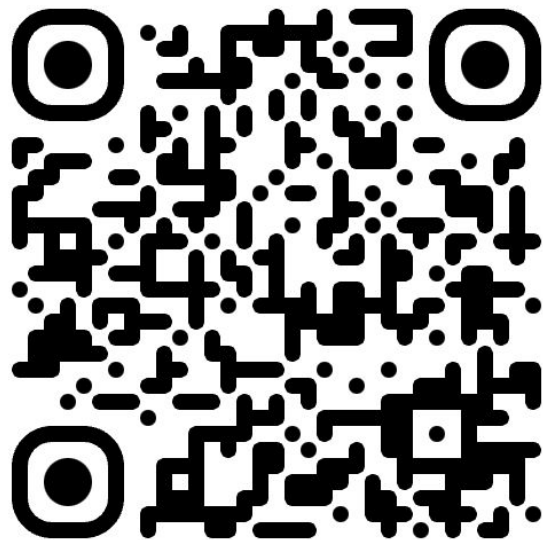
Benefits:

- **Handle Variadic Arguments Safely:** Allows functions to handle variable-length arguments with type safety.

```
Ts = TypeVarTuple("Ts")
def move_first_element_to_last(
    tup: tuple[T, *Ts]
) → tuple[*Ts, T]:
    ...
```

Thank you very much!!

Here is the URL of Repository in the Talk



`https://github.com/koxudaxi/europython_2024`