CHAPTER

EIGHTEEN

CONTEXTVARS — CONTEXT VARIABLES

This module provides APIs to manage, store, and access context-local state. The *ContextVar* class is used to declare and work with *Context Variables*. The *copy_context()* function and the *Context* class should be used to manage the current context in asynchronous frameworks.

Context managers that have state should use Context Variables instead of *threading.local()* to prevent their state from bleeding to other code unexpectedly, when used in concurrent code.

See also **PEP 567** for additional details.

New in version 3.7.

18.1 Context Variables

class contextvars.**ContextVar**(*name*[, *, *default*])

This class is used to declare a new Context Variable, e.g.:

```
var: ContextVar[int] = ContextVar('var', default=42)
```

The required *name* parameter is used for introspection and debug purposes.

The optional keyword-only *default* parameter is returned by *ContextVar.get()* when no value for the variable is found in the current context.

Important: Context Variables should be created at the top module level and never in closures. *Context* objects hold strong references to context variables which prevents context variables from being properly garbage collected.

name

The name of the variable. This is a read-only property.

New in version 3.7.1.

get([*default*])

Return a value for the context variable for the current context.

If there is no value for the variable in the current context, the method will:

- return the value of the *default* argument of the method, if provided; or
- return the default value for the context variable, if it was created with one; or
- raise a *LookupError*.

set(*value*)
Call to set a new value for the context variable in the current context.

The required *value* argument is the new value for the context variable.

Returns a *Token* object that can be used to restore the variable to its previous value via the *ContextVar.reset()* method.

reset(*token*)
Reset the context variable to the value it had before the *ContextVar.set()* that created the *token* was used.

For example:

```
var = ContextVar('var')

token = var.set('new value')
# code that uses 'var'; var.get() returns 'new value'.
var.reset(token)

# After the reset call the var has no value again, so
# var.get() would raise a LookupError.
```

class contextvars.Token
Token objects are returned by the *ContextVar.set()* method. They can be passed to the *ContextVar.reset()* method to revert the value of the variable to what it was before the corresponding *set*.

Token.var
A read-only property. Points to the *ContextVar* object that created the token.

Token.old_value
A read-only property. Set to the value the variable had before the *ContextVar.set()* method call that created the token. It points to **Token.MISSING** is the variable was not set before the call.

Token.MISSING
A marker object used by **Token.old_value**.

18.2 Manual Context Management

contextvars.copy_context()
Returns a copy of the current *Context* object.

The following snippet gets a copy of the current context and prints all variables and their values that are set in it:

```
ctx: Context = copy_context()
print(list(ctx.items()))
```

The function has an O(1) complexity, i.e. works equally fast for contexts with a few context variables and for contexts that have a lot of them.

class contextvars.Context
A mapping of *ContextVars* to their values.

Context() creates an empty context with no values in it. To get a copy of the current context use the *copy_context()* function.

Context implements the *collections.abc.Mapping* interface.

run(*callable*, **args*, ***kwargs*)
Execute `callable(*args, **kwargs)` code in the context object the *run* method is called on. Return the result of the execution or propagate an exception if one occurred.

Any changes to any context variables that *callable* makes will be contained in the context object:

```
var = ContextVar('var')
var.set('spam')

def main():
    # 'var' was set to 'spam' before
    # calling 'copy_context()' and 'ctx.run(main)', so:
    # var.get() == ctx[var] == 'spam'

    var.set('ham')

    # Now, after setting 'var' to 'ham':
    # var.get() == ctx[var] == 'ham'

ctx = copy_context()

# Any changes that the 'main' function makes to 'var'
# will be contained in 'ctx'.
ctx.run(main)

# The 'main()' function was run in the 'ctx' context,
# so changes to 'var' are contained in it:
# ctx[var] == 'ham'

# However, outside of 'ctx', 'var' is still set to 'spam':
# var.get() == 'spam'
```

The method raises a *RuntimeError* when called on the same context object from more than one OS thread, or when called recursively.

copy()
Return a shallow copy of the context object.

var in context
Return `True` if the *context* has a value for *var* set; return `False` otherwise.

context[var]
Return the value of the *var* *ContextVar* variable. If the variable is not set in the context object, a *KeyError* is raised.

get(*var*[, *default*])
Return the value for *var* if *var* has the value in the context object. Return *default* otherwise. If *default* is not given, return `None`.

iter(context)
Return an iterator over the variables stored in the context object.

len(proxy)
Return the number of variables set in the context object.

keys()
Return a list of all variables in the context object.

`values()`
Return a list of all variables' values in the context object.

`items()`
Return a list of 2-tuples containing all variables and their values in the context object.

18.3 asyncio support

Context variables are natively supported in *asyncio* and are ready to be used without any extra configuration. For example, here is a simple echo server, that uses a context variable to make the address of a remote client available in the Task that handles that client:

```
import asyncio
import contextvars

client_addr_var = contextvars.ContextVar('client_addr')

def render_goodbye():
    # The address of the currently handled client can be accessed
    # without passing it explicitly to this function.

    client_addr = client_addr_var.get()
    return f'Good bye, client @ {client_addr}\n'.encode()

async def handle_request(reader, writer):
    addr = writer.transport.get_extra_info('socket').getpeername()
    client_addr_var.set(addr)

    # In any code that we call is now possible to get
    # client's address by calling 'client_addr_var.get()'.

    while True:
        line = await reader.readline()
        print(line)
        if not line.strip():
            break
        writer.write(line)

    writer.write(render_goodbye())
    writer.close()

async def main():
    srv = await asyncio.start_server(
        handle_request, '127.0.0.1', 8081)

    async with srv:
        await srv.serve_forever()

asyncio.run(main())

# To test it you can use telnet:
#     telnet 127.0.0.1 8081
```