Context Manager and Decorator in Python

Min Khant

Context Manager

- Used mostly where opening and closing of a file or a connection is required
- Context manager patterns
  - open / close
  - enter / exit
  - Lock / release
  - Change / reset
  - Setup / teardown
  - connect / disconnect

```
with open(file_name.ext) as f:

with sqlite3.connect(url) as con:
```

Creating Custom Context Manager

```
@contextlib.contextmanager
def database(url):
     connection = None
     try:
          connection = sqlite3.connect(url)
          yield db
     except:
          print("Error in connection")
     finally:
          if connection:
          connection.close()

url = "data.db"
with database(url) as mydb:
     Element = mydb.execute(Query)
```

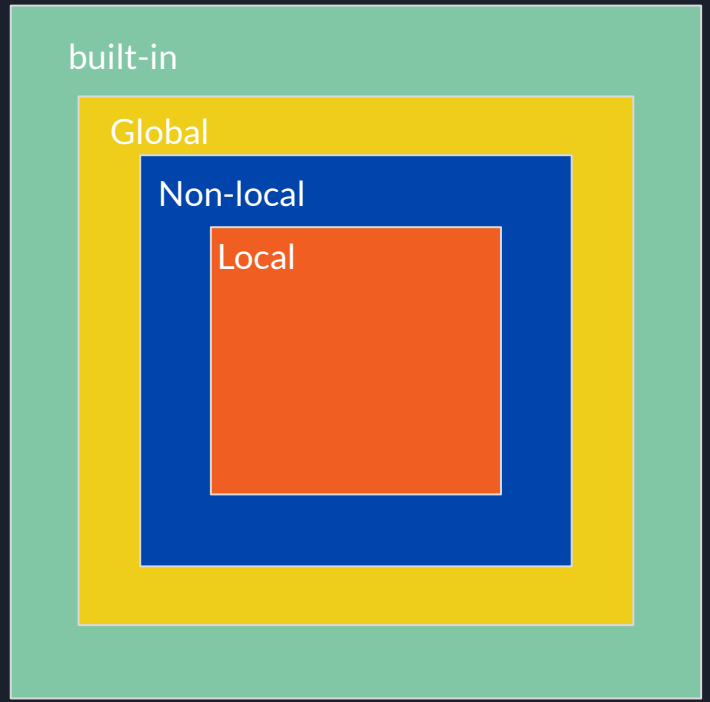
How Python find variables

Before we move to the decorator, we need to know about how Python works with variables.

Python prioritise them as follows:

1. Local
2. Non-local
3. Global
4. Built-in

Python check that the variable is defined in the local scope or not when it is called. If it is not defined in the local scope, the execution finds in other scopes as in the above order.



Closure

The closure is a tuple of variables that are no longer in scope, but that is required for a function to run.

```
def foo():
    a = 5
    def bar():
        print(a)
    return bar

func = foo()
func()  # return 5
a = 10
func()  # return 5
```

```
a = 5
def foo():
    def bar():
        print(a)
    return bar

func = foo()
func()  # return 5
a = 10
func()  # return 10
```

```
a = 5
def foo(value):
    def bar():
        print(value)
    return bar

func = foo(a)
func()  # return 5
a = 10
func()  # return 5
```

1. The variable `a` is non-local. Assigning `a` in the global scope won't change non-local variable `a`.
2. Since `a` is global variable, re-defining it will also change the result.
3. Although `a` is global variable, it becomes the **closure** of the function since it is passed to the function.

Continued...

The closure of a function can be assessed as belows.

```
>>func.__closure__      # return a tuple of variable values
>>func.__closure__[0].cell_contents  # return the first value in the closure
```

Decorator

What would you do if you need to process the inputs in a same way, every time you call a function? A good program must eliminate the problem of repeating yourself to do a single thing. That is where the decorator comes in.

Decorator is one of powerful Python abilities to make things easier. It takes a function as an input and consists of a nested wrapper function to modify the inputs of the specified function.

Creating custom decorator

```
from functools import wraps

def decorator_function(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        # your modification to the inputs (args and kwargs)
        return func(*args,  **kwargs)
    return wrapper
```

To be able to print metadata of func

Using decorator

There are two ways to use a defined decorator. Both ways return the same result.

```
@decorator_function
def function(*args):
    """This is a customized function"""
    # your function
```

```
def function(*args):
    """This is a customized function"""
    # your function

new_function = decorator_function(function)
```

Passing parameters to the decorators

What if we want to pass an argument to the decorator? We already know that it only accepts a function as the argument. Then, how can we give it? It is easy. Create a parent function and make the whole decorator code block as a child function.

```
def deco_with_parameters(n):                    Parent function
    def deco_func(func):                        Child function
        def wrapper(*args, **kwargs):
            # your modification to the input
        return wrapper`
    return deco_func

@deco_with_parameters(n)
def new_function():
    # your codeblock
```

Custom Timeout function

```
import signal
def raise_timeout(*args, **kwargs):
    raise TimeoutError()

signal.signal(signal.SIGALRM, handler = raise_timeout)

def timeout(n_seconds):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            signal.alarm(n_seconds)     # set alarm
            try:
                return func(*args, **kwargs)
            finally:
                signal.alarm(0)         # cancel alarm
        return wrapper
    return decorator
```

Timeout decorator Usage

```
@timeout(5)
def sleep_print():
     time.sleep(10)
     print(“Hello”)

sleep_print()

# This will raise error since
time out in 5s.
```

```
@timeout(20)
def sleep_print():
     time.sleep(10)
     print(“Hello”)

sleep_print()

# This will print “Hello” without
error.
```