Python Decorators and Namespaces

Namespaces

A namespace is a space that holds names(identifiers).Programmatically speaking, namespaces are dictionary of identifiers(keys) and their objects(values)

There are 4 types of namespaces:

- Builtin Namespace
- Global Namespace
- Enclosing Namespace
- Local Namespace

Scope and LEGB Rule

A scope is a textual region of a Python program where a namespace is directly accessible.

The interpreter searches for a name from the inside out, looking in the local, enclosing, global, and finally the built-in scope. If the interpreter doesn’t find the name in any of these locations, then Python raises a NameError exception.

In [9]:

```
# local and global
# global var
a = 2 # it is global

def temp():
  # local var
  b = 3
  print(b)

temp()
print(a)
```

```
3
2
```

if local and global are with same name then it will go with local

In [16]:

```
# local and global -> same name
a = 2

def temp():
  # local var
  a = 3
  print(a)

temp()
print(a)
```

```
3
2
```

In [18]:

```
# local and global -> local does not have but global has
a = 2

def temp():
  # local var
  print(a)

temp()
print(a)
```

```
2
2
```

In [ ]:

```
# local and global -> editing global
a = 2

def temp():
  # local var
  a += 1
  print(a)

temp()
```

```
print(a)
# we can not modify a global variable in local namespace
```

Global keyword :

- use to modify global variable in local if required

In [26]:

```
a = 2

def temp():
  # local var
  global a
  a += 1
  print(a)

temp()
print(a)
```

```
3
3
```

In [28]:

```
# local and global -> global created inside local
def temp():
  # local var
  global a
  a = 1
  print(a)

temp()
print(a)
```

```
1
1
```

In [ ]:

```
# local and global -> function parameter is local
def temp(z):
  # local var
  print(z)

a = 5
```

```
temp(5)
print(a)
print(z) # z is not in global so it will give error
```

In [30]:

```
# built-in scope
import builtins
print(dir(builtins))
```

```
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BaseExceptionGroup', 'BlockingIOError', 'BrokenPipeEr
ror', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError', 'ConnectionRefusedErro
r', 'ConnectionResetError', 'DeprecationWarning', 'EOFError', 'Ellipsis', 'EncodingWarning', 'EnvironmentError', 'Exception',
'ExceptionGroup', 'False', 'FileExistsError', 'FileNotFoundError', 'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOE
rror', 'ImportError', 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None', 'NotADirectoryError', 'NotImplem
ented', 'NotImplementedError', 'OSError', 'OverflowError', 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupErro
r', 'RecursionError', 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration', 'StopIterati
on', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError', 'UnboundLoca
lError', 'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'WindowsError', 'ZeroDivisionError', '__IPYTHON__', '__build_class__', '__debug__', '__doc__', '__impo
rt__', '__loader__', '__name__', '__package__', '__spec__', 'abs', 'aiter', 'all', 'anext', 'any', 'ascii', 'bin', 'bool', 'bre
akpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits', 'delattr', 'dic
t', 'dir', 'display', 'divmod', 'enumerate', 'eval', 'exec', 'execfile', 'filter', 'float', 'format', 'frozenset', 'get_ipytho
n', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass', 'iter', 'len', 'l
icense', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'propert
y', 'range', 'repr', 'reversed', 'round', 'runfile', 'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'supe
r', 'tuple', 'type', 'vars', 'zip']
```

We cannot rename or reuse built-ins

In [ ]:

```
L = [1,2,3]
print(max(L))
def max():
  print('hello')

print(max(L))
# it will give error because it is calling its own max function
```

In [41]:

```
# Enclosing scope
def outer():
  a=10
  def inner():
    print(a)
  inner()
  print('outer function')


outer()
print('main program')
```

```
10
outer function
main program
```

Non Local Keyword :

- it tells that it is not local to where it existing

In [49]:

```
# nonlocal keyword
def outer():
  a = 1
  def inner():
    nonlocal a
    a += 1
    print('inner',a)
  inner()
  print('outer',a)


outer()
print('main program')
```

```
inner 2
outer 2
main program
```

Decorators

A decorator in python is a function that receives another function as input and adds some functionality(decoration) to and it and returns it.

This can happen only because python functions are 1st class citizens.

There are 2 types of decorators available in python

- `Built in decorators` like `@staticmethod`, `@classmethod`, `@abstractmethod` and `@property` etc
- `User defined decorators` that we programmers can create according to our needs

In [58]:

```python
# python are 1st class citizen so we can take a fn as argument and can return it
def modify(func,num):
    return func(num)
def square(num):
    return num**2
modify(square,10)
```

Out[58]:

```
100
```

In [66]:

```python
# example of decorator
def my_decorator(func):
    def wrapper():
        print("---------------")
        func()
        print("+++++++++++++++")
    return wrapper

def func1():
    print("hello")

a=my_decorator(func1)
a()
```

```
---------------
hello
+++++++++++++++
```

In [91]:

```python
def decorate(func):
    def wrapper(lang):
        print("Hello to world of ",end=' ')
        func(lang)
    return wrapper
```

```
@decorate
def func(lang):
    print(lang)

func("Python")
```

```
Hello to world of  Python
```

Points to be take care

- wrapper function use "func" argument instead parent function
- wrapper function finally return a wrapped new function
- closure - wrapper function able to use outer function variable and its scope instead outer function no more exist

In [80]:

```
def outer():
    a=5
    def inner():
        print(a)
    return inner
b=outer()
b()
```

```
5
```

In [83]:

```
# Better syntax# simple example

def my_decorator(func):
  def wrapper():
    print('***********************')
    func()
    print('***********************')
  return wrapper

@my_decorator
def hello():
  print('hello')

hello()
```

```
***********************
hello
***********************
```

In [85]:

```
# meaningful decorator
import time

def timer(func):
  def wrapper(*args):
    start = time.time()
    func(*args)
    print('time taken by',func.__name__,time.time()-start,'secs')
  return wrapper

@timer
def hello():
  print('hello wolrd')
  time.sleep(2)

@timer
def square(num):
  time.sleep(1)
  print(num**2)

@timer
def power(a,b):
  print(a**b)

hello()
square(2)
power(2,3)
```

```
hello wolrd
time taken by hello 2.0008041858673096 secs
4
time taken by square 1.0017287731170654 secs
8
time taken by power 0.0 secs
```

Big problem : checking data type of argument

In [99]:

```
def sanity_check(data_type):
  def outer_wrapper(func):
    def inner_wrapper(*args):
      if type(*args) == data_type:
        func(*args)
      else:
        raise TypeError('Ye datatype nai chalega')
    return inner_wrapper
  return outer_wrapper

@sanity_check(int)\

def square(num):
  print(num**2)

@sanity_check(str)
def greet(name):
  print('hello',name)

square(2)
```

```
4
```

Step-by-Step Flow of the Code

1. **Defining the `sanity_check` Decorator Factory**:
   - `sanity_check(data_type)` is a decorator factory function.
   - It takes `data_type` as an argument (like `int` or `str`), which defines the data type you want to check for in the decorated function’s arguments.
2. **Calling `sanity_check(int)` for the `square` Function**:
   - `@sanity_check(int)` is applied to the `square` function, so `sanity_check(int)` is called with `data_type` set to `int`.
   - This returns the `outer_wrapper` function, with `data_type` now set to `int`.
3. **Applying `outer_wrapper` to `square`**:
   - `outer_wrapper` takes the `square` function as its `func` argument.

  - Inside `outer_wrapper`, it defines `inner_wrapper`, which wraps `func` (the `square` function) with additional functionality.
  - `outer_wrapper` returns `inner_wrapper`, so `square` is now replaced by `inner_wrapper`. From this point onward, any call to `square` will actually call `inner_wrapper`.

4. **Executing `square(2)`**:

   - When `square(2)` is called, it actually invokes `inner_wrapper(2)` due to the decorator.
   - Now, `inner_wrapper`:
     - Receives `2` as part of `*args`.
     - Checks if all arguments in `*args` match the `data_type` (`int` in this case) using `all(isinstance(arg, data_type) for arg in args)`.
     - Since `2` is an integer, the check passes.
     - `inner_wrapper` then calls `func(*args)`, which is the original `square` function with `num=2`.
   - `square(2)` executes and prints `4` (i.e., `2**2`).

5. **Calling `greet` with `@sanity_check(str)`**:

   - Similarly, `@sanity_check(str)` decorates `greet`, so `sanity_check(str)` is called with `data_type` set to `str`.
   - `outer_wrapper` wraps `greet` with `inner_wrapper`, which now checks if all arguments are of type `str` before calling the original `greet` function.

6. **Error Handling**:

   - If a different data type is passed (e.g., calling `square("2")`), `inner_wrapper` detects a mismatch in types and raises a `TypeError` with the message `"Ye datatype nai chalega"`.

Summary

The flow includes:

1. **Decorator factory (`sanity_check`) setup**.
2. **Decorator application** via `outer_wrapper` and `inner_wrapper`.
3. **Type-checking logic** in `inner_wrapper`.
4. **Execution of the original function** only if the type check passes.
5. **Error handling** if the type check fails.

This flow ensures the decorated functions only run if the arguments match the specified type, providing type validation as a reusable decorator.

END