Python *namespaces, closures and other fantastic beasts*

Overview

- Closures
- Scopes and namespaces
- Anonymous functions
- Decorators

Closures

Closures

Python's functions are *first-class* objects. You can assign them to variables, store them in data structures, pass them as arguments to other functions, *define inside* other functions, and even *return them as values* from other functions.

Closures

```
>>> def fabric_function(number):
...     """ Enclosing function """
...     def print_function():
...         """ Nested function """
...         print(number)
...
...     return print_function
```

Closures

```
>>> def fabric_function(number):
...     """ Enclosing function """
...     def print_function():
...         """ Nested function """
...         print(number)
...
...     return print_function


>>> print_1 = fabric_function(1)
... print_2 = fabric_function(2)
... print_3 = fabric_function(3)
```

Closures

```
>>> print_1 = fabric_function(1)
... print_2 = fabric_function(2)
... print_3 = fabric_function(3)

>>> print_1()
... print_2()
... print_3()
1
2
3
```

Closures

```
>>> print(fabric_function.__closure__)
None

>>> print(print_1.__closure__)
(<cell at 0x7fcac5aeeaf8: int object at 0x108f285a0>,)

>>> print(print_1.__closure__[0].cell_contents)
1
```

Closures

```
>>> def log_prefix(prefix):
...     def log(text):
...         print(f'{prefix}: {text}')
...     return log


>>> log_info = log_prefix("INFO")
>>> log_info("Call status Ok")

INFO: Call status Ok
```

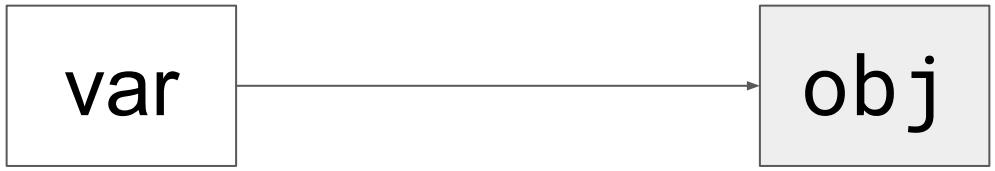
Overview

- The closure is a function, or more strictly speaking, **an inner function**, which is defined within the scope of the other function (termed outer function).

- The inner function **binds variables defined outside of its own scope**.

Namespaces and scopes

Symbolic names



Namespace

A ***namespace*** is a collection of currently defined symbolic names along with information about the object that each name references

Scope

A ***scope*** defines which namespaces will be looked in and in what order.

The ***scope*** of any reference always starts in the local namespace, and moves outwards until it reaches the module's global namespace

Namespace and scope

```
>>> name = 1
... def func():
...     name = func.__name__
...     print(locals())
...     print(globals()
...
... func()

{'name': 'func'}

{'__name__': '__main__', '__doc__': None, '__package__':
None, ..., '__builtins__': <module 'builtins' (built-in)>,
'name': 1, 'func': <function func at 0x7fbe556c79e0>}
```

LEGB Rule

- **L**ocal - contains the names that you define inside the function.

- **E**nclosing (nonlocal) scope is a special scope that only exists for nested functions.

- **G**lobal (or module) scope is the top-most scope in a Python program, script, or module.

- **B**uilt-in scope is a special Python scope that’s created or loaded whenever you run a script or open an interactive session.

LEGB Rule

Local - contains the names that you define inside the function.

These names will only be visible from the code of the function.

It's created at function call, *not* at function definition, so you'll have as many different local scopes as function calls.

This is true even if you call the same function multiple times, or recursively.

Each call will result in a new local scope being created.

LEGB Rule

Enclosing (nonlocal) scope is a special scope that only exists for nested functions.

This scope contains the names that you define in the enclosing function.

The names in the enclosing scope are visible from the code of the inner and enclosing functions.

LEGB Rule

Global scope is the top-most scope in a Python program, script, or module.

This Python scope contains all of the names that you define at the top level of a program or a module.

Names in this Python scope are visible from everywhere in your module's code.

LEGB Rule

Built-in scope is a special Python scope that's created or loaded whenever you run a script or open an interactive session.

This scope contains names such as keywords, functions, exceptions, and other attributes that are built into Python.

Names in this Python scope are available from everywhere in your code.

It's automatically loaded by Python when you run a program or script.

Scopes

```
>>> name = "Global"                          # global
>>> def outer_function():
...     name = "Outer function scope"       # enclosing
...     def inner_fuction():
...         name = "Inner function scope" # local
...         print(name)
...     return inner_fuction

>>> func = outer_function()

>>> func()
Inner function scope
```

Closures

```
>>> name = "Global"
>>> def outer_function():
...     name = "Outer function scope"
...     def inner_fuction():
...         print(name)
...     return inner_fuction

>>> func = outer_function()

>>> func()
Outer function scope
```

Closures

```
>>> name = "Global"
>>> def outer_function():
...     def inner_fuction():
...         print(name)
...     return inner_fuction

>>> func = outer_function()

>>> func()
Global
```

Closures

```
>>> name = "Global"
>>> def outer_function():
...     def inner_fuction():
...         print(name)
...     return inner_fuction

>>> func = outer_function()

>>> func()
Global
```

global

```
>>> num = 1
...
...
... def func():
...     num = num + 1
...     return num
...
...
... print(func())
Traceback (most recent call last):
  File "<stdin>", line 9, in <module>
  File "<stdin>", line 5, in func
UnboundLocalError: local variable 'num' referenced before
assignment

local variable 'num' referenced before assignment
```

global

global allows to modify variable value from global scope

global

```
>>> num = 1
...
...
... def func():
...     global num
...     num = num + 1
...     return num
...
...
... print(func())
2
```

nonlocal

```
>>> def find_number(seq, num):
...     found = False
...
...     def helper():
...         indexes = []
...         for index, item in enumerate(seq):
...             if item == num:
...                 indexes.append(index)
...                 found = True
...         return indexes
...
...     indexes = helper()
...     return indexes, found

>>> find_number([1, 2, 3, 4, 4, 4, 4], 4)
([3, 4, 5, 6], False)
```

nonlocal

nonlocal allows to modify variable value from enclosing scope

nonlocal

```
>>> def find_number(seq, num):
...     found = False
...
...     def helper():
...         nonlocal found
...         indexes = []
...         for index, item in enumerate(seq):
...             if item == num:
...                 indexes.append(index)
...                 found = True
...         return indexes
...
...     indexes = helper()
...     return indexes, found

>>> find_number([1, 2, 3, 4, 4, 4, 4], 4)
([3, 4, 5, 6], True)
```

Namespaces and scopes

- There are four scopes in Python: Built-In, Global, Enclosing, Local
- LEGB Rule: name search is processed from Local to Built-In
- Assignment operations create name in current scope unless name defined as **global** or **nonlocal**

Anonymous functions (lambdas)

Anonymous functions (lambdas)

In Python, an anonymous function is a function that is defined without a name

Anonymous functions (lambdas)

```
lambda arguments: expression
```

Anonymous functions (lambdas)

```
lambda *args, **kwargs: print(args, kwargs)

# Smth wrong
>>> a = lambda *args, **kwargs: print(args, kwargs)
>>> a()
() {}

>>> a(1, 2, 3, k=1)
(1, 2, 3) {'k': 1}
```

Anonymous functions (lambdas)

```
>>> lst = [{'key': 20}, {'key': 1}, {'key': 11}]
>>> sorted(lst)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'dict' and
'dict'

'<' not supported between instances of 'dict' and 'dict'
```

Anonymous functions (lambdas)

```
>>> lst = [{'key': 20}, {'key': 1}, {'key': 11}]
>>> sorted(lst, key=lambda x: x['key'])
[{'key': 1}, {'key': 11}, {'key': 20}]
```

filter

```
>>> lst = [1, 2, 3, 4, 5]

>>> even_filter = filter(lambda x: not (x % 2), lst)

>>> even_filter
<filter object at 0x7fcac81bd7f0>


>>> list(even_filter)
[2, 4]
```

filter

The function is called with **all** the items in the list and a new list is returned which contains items for which the function evaluates to **True**

map

```
>>> lst = ['1', '2', '3', '4', '5']

>>> int_lst = map(lambda x: int(x), lst)

>>> int_lst
<map object at 0x7fcac8b68e80>


>>> list(int_lst)
[1, 2, 3, 4, 5]
```

map

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item

Decorators

Decorators

```
decorators                ::=  decorator+
decorator                 ::=
"@" dotted_name ["(" [argument_list [","]] ")"]
NEWLINE
```

Decorators

```
decorators          ::=  decorator+
decorator           ::=
"@" dotted_name ["(" [argument_list [","]] ")"] NEWLINE
```

```
@f1(arg)
@f2
def func(): pass

~

def func(): pass
func = f1(arg)(f2(func))
```

Dummy decorator

```
>>> def dummy_decorator(func):
...     print('Hello')
...
...
... @dummy_decorator
... def func():
...     print('Inner execution of f')

Hello

 ~


>>> func = dummy_decorator(func)
Hello
>>> type(func)
<class 'NoneType'>
```

Decorators

```
import time

def time_decorator(func):

    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f'Execution time {end - start}')

        return result

    return wrapper

@time_decorator
def join(seq, delimiter):
    return delimiter.join(seq)

print(join(['h', 'e', 'l', 'l', 'o'], delimiter=''))
Execution time 1.9073486328125e-06
hello
```

Decorators

```
def upper(func):

    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs).upper()
        return res

    return wrapper

@upper
def hello(string: str) -> str:
    return string


>>> hello("hello")
'HELLO'
```

Decorators

```
def upper(func):

    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs).upper()
        return res

    return wrapper

@upper
def hello(string: str) -> str:
    return string


>>> hello.__name__
'wrapper'
```

Decorators

```
import functools

def upper(func):

    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        res = func(*args, **kwargs).upper()
        return res

    return wrapper

@upper
def hello(string: str) -> str:
    return string

>>> hello("hello")
'HELLO'

>>> hello.__name__
'hello'
```

Parameterized decorators

```
def decorator_factory(log=True):

    def decorator(func):

        @functools.wraps(func)
         def wrapper(*args, **kwargs):
             if log:
                 print(f"Called with {args!r} {kwargs!r}")
             return func(*args, **kwargs)

         return wrapper

    return decorator

@decorator_factory(log=True)
def hello(string: str) -> str:
    return string

>>> hello("hello")
Called with ('hello',) {}
'hello'
```

Parameterized decorators

```
def decorator_factory(log=True):

    def decorator(func):

        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            if log:
                print(f"Called with {args!r} {kwargs!r}")
            return func(*args, **kwargs)

        return wrapper

    return decorator

@decorator_factory(log=False)
def hello(string: str) -> str:
    return string

>>> hello("hello")
'hello'
```

Parameterized decorators

```
class LogDecorator:
    def __init__(self, log=False):
        self.log = log

    def __call__(self, func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            if self.log:
                print(f"Called with {args!r} {kwargs!r}")
            return func(*args, **kwargs)
        return wrapper

@LogDecorator(log=True)
def hello(string: str) -> str:
    return string

>>> hello("Hello")
Called with ('Hello',) {}
'Hello'
```

Decorators

```
def un_private_class(cls: type = None):
    """ Return the same class as was wrapped in with changed
    __getattr__ method to get access for private attributes
    Usage:
            @un_private_class
            class A:
                ...
        or
            A = un_private_class(A)
    """
    def getattribute_w(self: object, name: str):
        if name.startswith("__") and not name.endswith("__"):
            name = "_%s%s" % (cls.__name__, name)
        return self.__getattribute__(name)

    cls.__getattr__ = getattribute_w
    return cls
```

Decorators

```
>>> class A:
...     def __init__(self):
...         self.__private = "private"

>>> a = A()

>>> a.__private
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute '__private'
```

Decorators

```
>>> @un_private_class
... class A:
...     def __init__(self):
...         self.__private = "private"

>>> a = A()

>>> a.__private
'private'
```

Decorators

```
>>> from functools import lru_cache

>>> @lru_cache(maxsize=128)
... def fib(n):
...     if n < 2:
...         return n
...     return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(16)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
610]

>>> fib.cache_info()
CacheInfo(hits=28, misses=16, maxsize=128, currsize=16)
```

Decorators

```
>>> @lru_cache(maxsize=None)
... def fib(n):
...     if n < 2:
...         return n
...     return fib(n-1) + fib(n-2)

>>> [fib(n) for n in range(128)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377,
610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657,
46368, 75025, 121393, 196418, 317811, 514229, 832040,
1346269, 2178309, 3524578, ... ]

>>> fib.cache_info()
CacheInfo(hits=252, misses=128, maxsize=None,
currsize=128)
```

Decorators

```
>>> from functools import total_ordering
>>> @total_ordering

... class Student:
...     def __init__(self, lastname, firstname):
...         self.firstname = firstname
...         self.lastname = lastname
...     def __eq__(self, other):
...         return ((self.lastname.lower(), self.firstname.lower()) ==
...                 (other.lastname.lower(), other.firstname.lower()))
...     def __lt__(self, other):
...         return ((self.lastname.lower(), self.firstname.lower()) <
...                 (other.lastname.lower(), other.firstname.lower()))
...
...
... student1 = Student("Bond","James")
... student2 = Student("Haddock","Captain")
...
... print(student2 > student1)  # True
... print(student2 >= student2) # True
... print(student2 != student1) # True
True
True
True
```

Decorators

```
>>> from functools import total_ordering
>>> @total_ordering

... class Student:
...     def __init__(self, lastname, firstname):
...         self.firstname = firstname
...         self.lastname = lastname
...     def __eq__(self, other):
...         return ((self.lastname.lower(), self.firstname.lower()) ==
...                 (other.lastname.lower(), other.firstname.lower()))
...     def __lt__(self, other):
...         return ((self.lastname.lower(), self.firstname.lower()) <
...                 (other.lastname.lower(), other.firstname.lower()))
...
...
... student1 = Student("Bond","James")
... student2 = Student("Haddock","Captain")
...
... print(student2 > student1)  # True
... print(student2 >= student2) # True
... print(student2 != student1) # True
True
True
True
```

Decorators

```
>>> from functools import singledispatch

>>> @singledispatch
... def fun(arg, verbose=False):
...     if verbose:
...         print("Let me just say,", end=" ")
...     print(arg)

>>> @fun.register
... def _(arg: int, verbose=False):
...     if verbose:
...         print("Strength in numbers, eh?", end=" ")
...     print(arg)

>>> @fun.register
... def _(arg: list, verbose=False):
...     if verbose:
...         print("Enumerate this:")
...     for i, elem in enumerate(arg):
...         print(i, elem)
```

Decorators

```
>>> fun([1, 2, 3], verbose=True)
Enumerate this:
0 1
1 2
2 3

>>> fun(1, verbose=True)
Strength in numbers, eh? 1
```

Decorators

- A decorator is a design pattern in Python that allows a user to add new functionality to an existing object without modifying its structure
- Decorators are usually called before the definition of a function you want to decorate