1. Recursion, Callback, Generators, Closures, Decorators

08 November 2023 08:09

TYPES OF FUNCTIONS

- Recursion
- Callback
- Generator
- Closures
- Decorators

} Increases programming efficiency

RECURSION

Calling some function within the same function until some base condition / stop condition is met that ends the recursion

→ whichever problem statement / condition you already know the solution for

Example: $n! = \begin{cases} n \times (n-1)! & n \neq 0 \\ 1 & n = 0 \end{cases}$ → base condition

$$gcd(m,n) = \begin{cases} m \text{ (or) } n & m = n \\ gcd(m-n, n) & m > n \\ gcd(m, n-m) & m < n \end{cases}$$ → base condition

CHECK: Activation record, symbol table

Python 3.0 Essentials Book

Tower of Hanoi problem

NOTE: Recursion always moves towards the base condition

CALLBACK

Passing a function as an argument to a function ⟶ callback

Note: You do not call the function that is being passed as an argument; otherwise, only the returned value of that function is passed

Example: f1(f2)

→ callback function

```
def f1():
    print("in f1")

def f2():
    print("in f2")

def f3(f):
    f()

f3(f1)
f3(f2)
```

```
in f1
in f2
>>>
```

- Callback functions ⟶ make code more modular, requires fewer changes to code to add new functionality

Example using key parameter of sort():

```
def func(n):
    if len(str(n))>1:
        return int(str(n)[1])
    else:
        return n


l=[18,783,827,1,23,9]
l.sort(key=func)
print(l)
```

```
[1, 827, 23, 18, 783, 9]
>>>
```

proxy values are used to sort the original values

NOTE: Key parameter

Takes a function as its value and applies that function to every value to get some proxy value.
These proxy values are used to perform the original function.

GENERATORS

PREREQUISITES

NOTE: __iter__

If __iter__ is associated with a particular class, that class is iterable.

__iter__ in dir(list)
>>> True
__iter__ in dir(int)
>>> False

NOTE: Iterator

__iter__() function returns an iterator

l = [1, 5, 6, 200]

lit = l.__iter__()
→ type(lit) = list iterator

Iterator ⟶ lazy object
next(lit) ⟶ returns a value from lit

```
>>> l=[1,5,6,200]
>>> lit=l.__iter__()
>>> next(lit)
1
>>> next(lit)
5
>>> next(lit)
6
>>> next(lit)
200
>>> next(lit)
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    next(lit)
StopIteration
```

⟶ This is the exception handled automatically by for loop

GENERATORS

- At least one "yield" statement
- "return" statement ends generator execution; even if there is some value in the return statement, that value is not considered

- Returns an iterator object that is of class "generator"
  - contains all values that are yielded in the function

```
#generator function

def gen():
    print(1)
    yield "a"
    print(2)
    yield "b"
    print(3)
    yield "c"


print(gen()) #does not print values since generator is a lazy object

for i in gen():
    print(i)
```

```
<generator object gen at 0x000002054B56C5F0>
1
a
2
b
3
c
```

NOTE:
When a value is returned by the next() function from an iterator object, that value is removed from the iterator.

GENERATOR EXPRESSION

g = ([write required condition here]) } generator expression

- g → defined as a generator object

```
>>> g=(i*i for i in range(1,6))
>>> next(g)
1
>>> next(g)
4
>>> next(g)
9
>>> next(g)
16
>>> next(g)
25
>>> next(g)
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    next(g)
StopIteration
```

CLOSURE

PREREQUISITE: Nested functions

```
def f1():          → Nesting/enclosing function
    def f2():      → Nested/enclosed functions
        return
```

- f2 cannot be called outside f1, behaves like a local variable

Some cases of nested functions

```
def f1():
    print("in f1")
    def f2():
        print("in f2")
    return f2
f1()
f2()
```

```
in f1
Traceback (most recent call last):
  File "E:/PES/Sem 1/Python/Recursion/NestingFuncs.py", line 7, in <module>
    f2()
NameError: name 'f2' is not defined
```

```
def f1():
    print("in f1")
    global f2
    def f2():
        print("in f2")

f1()
f2()
```

```
in f1
in f2
```

```
def f1():
    print("in f1")
    def f2():
        print("in f2")
    print("f2:",id(f2),type(f2))
    return f2

f=f1()
print("f:",id(f),type(f))
f()
```

```
in f1
f2: 2399991266896 <class 'function'>
f: 2399991266896 <class 'function'>
in f2
```

```
def f1():
    print("in f1")
    def f2():
        print("in f2")
    print("f2:",id(f2),type(f2))
    return f2

f=f1()
print("f:",id(f),type(f))
del f1
f()
```

```
in f1
f2: 1453482278480 <class 'function'>
f: 1453482278480 <class 'function'>
in f2
```

local and nonlocal variables

Nested function does not have access to global variables or any local variables that are defined in the nesting function.

```
x=10
def f1():
    print("in f1")
    def f2():
        x=x+1
        print("in f2",a)
    return f2

f=f1()
f()
```

```
in f1
Traceback (most recent call last):
  File "E:/PES/Sem 1/Python/Recursion/NestingFuncs.py", line 10, in <module>
    f()
  File "E:/PES/Sem 1/Python/Recursion/NestingFuncs.py", line 5, in f2
    x=x+1
UnboundLocalError: local variable 'x' referenced before assignment
```

```
def f1():
    print("in f1")
    a=10
    print(a)
    def f2():
        a=a+1
        print("in f2",a)
    return f2

f=f1()
f()
```

```
in f1
10
Traceback (most recent call last):
  File "E:/PES/Sem 1/Python/Recursion/NestingFuncs.py", line 11, in <module>
    f()
  File "E:/PES/Sem 1/Python/Recursion/NestingFuncs.py", line 6, in f2
    a=a+1
UnboundLocalError: local variable 'a' referenced before assignment
```

```
def f1():
    print("in f1")
    a=10
    print(a)
    def f2():
        nonlocal a
        a=a+1
        print("in f2",a)
    return f2

f=f1()
f()
```

```
in f1
10
in f2 11
```

CLOSURES

Characteristics

- Nested function
- Should have a free variable (variable that is not bound in local scope), and the enclosed function should have access to it
- Nested function should be returned by the enclosing function

DECORATORS

- For some function, decorator function should be executed first before moving on to the actual functionality of the original function
- Extends functionality of original function
- Decorator is used with "@" operator right before original function definition

```
def deco_f1(f):
    def func():
        print("before call")
        f()
        print("after call")
    return func

@deco_f1  #deco_f1(f1)
def f1():
    print("in f1")


f1()
```

```
before call
in f1
after call
>>>
```

- Multiple decorator functions can be used for the same original function

```
def decx(f):
    def func():
        print("x"*20)
        f()
```

```python
def decx(f):
    def func():
        print("x"*20)
        f()
        print("x"*20)
    return func

def decy(f):
    def func():
        print("y"*20)
        f()
        print("y"*20)
    return func

@decx
@decy    #decx(decy(f1))
def f1():
    print("in f1")

f1()
```

```
xxxxxxxxxxxxxxxxxxxx
yyyyyyyyyyyyyyyyyyyy
in f1
yyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxx
>>>
```