Python Programming

by Narendra Allam

Chapter 5

Functions

Topics Covering

- Purpose of a function
- Defining a function
- Calling a function
- Function parameter passing
- Formal arguments
- Actual arguments
- Positional arguments
- Keyword arguments
- Variable arguments
- Variable keyword arguments
- Use-Case *args,* *kwargs
- Function call stack
  - locals()
  - globals()
  - Stackframe
- Call-by-object-reference
- Shallow copy - copy.copy()
- Deep copy - copy.deepcopy()
- recursion
- Passing functions to functions
- Defining functions within functions
- Returning function from a function
- Closure and Capturing
- Decorators
- Generators
- Closures
- Interview Questions
- Exercises

Purpose:

- Maximizing code reuse and minimizing redundancy
- Procedural decomposition which makes code maintainable

Lets start with factorial program

In [1]:

```
n = 5
f = 1
for x in range(1, n+1):
    f = f * x
print(f)
```

```
120
```

What if, we want to write nCr program?

```
nCr = n!/(n-r)!*r!
```

Three times we need to compute factorial, for 'n', 'n-r' and 'r'. Do we have to copy the above logic at all the three places? Let's assume we copied the logic.

In [2]:

```
# NCR Program

n = 5
r = 2

# n!
nfact = 1
for x in range(1, n+1):
    nfact = nfact * x

# n-r!
nrfact = 1
for x in range(1, n-r+1):
    nrfact = nrfact * x

# r!
rfact = 1
for x in range(1, r+1):
    rfact = rfact * x


print ('nCr: ', nfact/(nrfact*rfact))
```

```
nCr:  10.0
```

There are 3 problems here

- This increases code size, new variables are required.
- If there is any bug hidden, bug will be replicated, and we will have to fix at all the places
- If we want to enhance the logic, again we have to do it at all the places

What if, we are able to name the block of reusable code and, can execute this block, by simply calling its name whereever it is necessary?

Functions

- Function is a block of reusable code, which performs a task.
- Functions can take data and return data, but it is optional.

- Functions are first class objects in python.

Syntax:

```
def func_name(Param1, Param2, ...):
    statements
    return value(s)
```

Example:

```
def add(x, y):
    z = x + y
    return z
```

In [3]:

```
def add(x, y): # function definition
    z = x + y
    return z

result = add(3, 4) # function call
print (result)
```

```
7
```

In [4]:

```
# computing nCr by reusing the factorial program
def fact(a):
    f = 1
    for x in range(1, a+1):
        f = f * x
    return f

def nCr(n, r):
    res = fact(n)/(fact(n-r)*fact(r))
    return res

print('nCr = ', nCr(5, 2))
```

```
nCr =  10.0
```

Receiving parameters and returning values is just optional

In [5]:

```
# function which doesn't take anything and doesn't return anything
def greet():
    print('Hello, How are you today?')
res = greet()
print(res)
```

```
Hello, How are you today?
None
```

Note: In python it is allowed to expect a value when a function doesn't return anything. We get **None**.

Returning multiple values

In [6]:

```
def sum_avg(x, y, z):
    s = x + y + z
    a = s/3.0
    return s, a
```

In [7]:

```
total, avg = sum_avg(3, 4 ,5)
print('Total: {}, Average: {}'.format(total, avg))
```

```
Total: 12, Average: 4.0
```

Function Parameter passing

In [8]:

```
# definition of function add
def add(x, y, z): # formal parameters
    s = x + y + z
    return s

a = 2
b = 4
# function call
final_sum = add(a, 3, b)# actual arguments
print(final_sum)
```

```
9
```

In the above code

- a, 3, b are actual arguments
- x, y, z are formal parameters

In [9]:

```
add(4,5,6)
```

Out[9]:

```
15
```

Positional arguments

In [10]:

```
# Positional arguments are mandatory arguments,
# Arguments will be recieved by the formal arguments
# in the same order of actual arguments.
# we cannot skip passing them.
def add(x, y, z):
    s = x + y + z
    return s

add(3, 4, 5) # positional arguments
```

Out[10]:

```
12
```

Keyword Arguments

In [11]:

```
def add(x, y, z):
    s = x + y + z
    return s

add(2, z=4, y=5) # z and y are keyword arguments
```

Out[11]:

```
11
```

In [12]:

```
add(2, z=5, 10) # is not possible
```

```
  File "<ipython-input-12-5a481a665e75>", line 1
    add(2, z=5, 10) # is not possible
                ^
SyntaxError: positional argument follows keyword argument
```

In [13]:

```
add(2, 5, y=10) # Is this valid?
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
 last)
<ipython-input-13-3aafe06a3aa0> in <module>
----> 1 add(2, 5, y=10) # Is this valid?

TypeError: add() got multiple values for argument 'y'
```

Default Arguments

In [ ]:

```
def add(x, y, z=5): # default arguments
    s = x + y + z
    return s

print(add(3, 4)) # z takes default value 5
print(add(4, 3, 10)) # z's default value replaced by the actual argument, 10
```

In [ ]:

```
def add(x, y=0, z=0): # default arguments
    s = x + y + z
    return s

print(add(4))  # y, z takes default value 0
print(add(4, 5)) # y's default value replaced by the actual argument value 5 and
print(add(4, z=5)) # z takes default value 0 and z's default value replaced by
```

***** non-default arguments must not follow default arguments**

In [ ]:

```
def add(x=0, y, z=0):
    r = x + y + z
    return r

print(add(4, 5))
```

Variable arguments

usecase : sum(), avg()

In [ ]:

```
def add(*args):
    print(type(args))
    s = 0
    for x in args:
        s = s + x
    return s
```

Note: Varaible arguments are sent to the function as a tuple

In [ ]:

```
add(2, 3, 4.0, 5, 6, 7.5, 8, 9)
```

In [ ]:

```
add()
```

Paramater unpacking

In [ ]:

```
t = 2, 3, 4

def fun(x, y, z):
    print(x + y + z)

fun(t[0], t[1], t[2])
```

In [ ]:

```
fun(*t)
```

Ternary Operator

```
Syntax:- result = val_1 if (condition) else val_2
```

In [14]:

```
y = 7
x = 20 if y > 0 else 30
print(x)
```

```
20
```

Variable Keyword arguments

- Variable keyword arguments are sent to a function as a dict
- Easy to maintain API, we can easily incorporate new changes in the implementation of a function, without changing its signature, thus without breaking applications.

In [15]:

```
def add(**kwargs):
    print (type(kwargs))
    print (kwargs)

add(a=20, b=30, c=40) # ==> add({'a':20, 'b':30, 'c':40})
```

```
<class 'dict'>
{'a': 20, 'b': 30, 'c': 40}
```

Note: Varaible keyword arguments are sent to the function as a dict.

Iterating variable keyword arguments:

In [16]:

```
def add(**kwargs):
    for var, val in kwargs.items():
        print (var,'->', val)

add(a=20, b=30, c=40)
```

```
a -> 20
b -> 30
c -> 40
```

A function with complete signature can be seen below:

In [17]:

```
def fun(a, b, c=10, d=20, *args, **kwargs):

    print ('--------------------')
    print ('Positional arguments')
    print ('--------------------')
    print ('a = ', a, ' b = ', b)

    print ('--------------------')
    print ('default arguments')
    print ('--------------------')
    print ('c = ', c, ' d = ', d)

    print ('--------------------')
    print ('Variable arguments')
    print ('--------------------')
    print (args)

    print ('--------------------')
    print ('Variable Keyword arguments')
    print ('--------------------')
    print (kwargs)

```

- a, b - positional arguments
- c, d - default arguments
- args - variable arguments
- kwargs - variable keyword arguments

In [18]:

```
res = fun(2, 3, 4, 5, 6, 7, 8, p=10, q=20)
```

```
--------------------
Positional arguments
--------------------
a =  2  b =  3
--------------------
default arguments
--------------------
c =  4  d =  5
--------------------
Variable arguments
--------------------
(6, 7, 8)
--------------------
Variable Keyword arguments
--------------------
{'p': 10, 'q': 20}
```

In [19]:

```
res = fun(2, 3, 5, 6)
```

```
--------------------
Positional arguments
--------------------
a =  2  b =  3
--------------------
default arguments
--------------------
c =  5  d =  6
--------------------
Variable arguments
--------------------
()
--------------------
Variable Keyword arguments
--------------------
{}
```

In [20]:

```
res = fun(2,3, k=10)
```

```
-------------------
Positional arguments
-------------------
a =  2  b =  3
-------------------
default arguments
-------------------
c =  10  d =  20
-------------------
Variable arguments
-------------------
()
-------------------
Variable Keyword arguments
-------------------
{'k': 10}
```

In [21]:

```
res = fun(4,5)
```

```
-------------------
Positional arguments
-------------------
a =  4  b =  5
-------------------
default arguments
-------------------
c =  10  d =  20
-------------------
Variable arguments
-------------------
()
-------------------
Variable Keyword arguments
-------------------
{}
```

In [22]:

```
def median(*args):
    l = sorted(args)
    mid = len(args)//2
    return l[mid]
```

In [23]:

```
median(45, 8, 2, 1, 6, 7)
```

Out[23]:

```
7
```

Parameter Unpacking

In [24]:

```
d = {'x':20, 'y':30, 'z':40}
def fun(x, y, z):
    print(x + y + z)

fun(d['x'], d['y'], d['z'])
# fun(**d)
```

```
90
```

Use-Case: Variable Arguments and Variable Keyword arguments

*args, **kwargs are generally placed at the end of each function signature, in the initial phase of library development as place holders for future enhancements and for backward compatibility with newer applications.

Scope

In the below program n and x in main() function are completely different from x and n are in fun(). x and n in main() function are only accessible for main function. When we pass n to fun(), the value of n will have a new name in new locality which. This called locality of reference. x, n inside fun() are brand new x and n. As long as execution control is in fun() it has its own set of local names.

Globals and Locals

In [25]:

```
g = 9.8
def fun(n):
    x = 50
    y = 90
    n = n*10
    print ('---- in side fun()-------')
    print ('----- locals() ----------')
    print (locals())
    print ('----- globals() ----------')
    print (globals())

def start():
    x = 20
    n = 30
    fun(n)
    print ('---- in side fun()-------')
    print ('----- locals() ----------')
    print (locals())
    print ('----- globals() ----------')
    print (globals())

start()
```

```
---- in side fun()-------
----- locals() ----------
{'y': 90, 'x': 50, 'n': 300}
----- globals() ----------
{'__name__': '__main__', '__doc__': 'Automatically created module for
IPython interactive environment', '__package__': None, '__loader__': N
one, '__spec__': None, '__builtin__': <module 'builtins' (built-in)>,
'__builtins__': <module 'builtins' (built-in)>, '_ih': ['', 'n = 5\nf
= 1\nfor x in range(1, n+1):\n    f = f * x\nprint(f)', "# NCR Program
\n\nn = 5\nr = 2\n\n# n!\nnfact = 1\nfor x in range(1, n+1):\n    nfac
t = nfact * x\n\n# n-r!\nnrfact = 1\nfor x in range(1, n-r+1):\n    nr
fact = nrfact * x\n    \n# r!    \nrfact = 1\nfor x in range(1, r+
1):\n    rfact = rfact * x\n    \n    \nprint ('nCr: ', nfact/(nrfact*
rfact))", 'def add(x, y): # function definition\n    z = x + y\n    re
turn z\n\nresult = add(3, 4) # function call\nprint (result)', "# comp
uting nCr by reusing the factorial program\ndef fact(a): \n    f = 1
\n    for x in range(1, a+1): \n        f = f * x\n    return f\n\ndef
nCr(n, r):\n    res = fact(n)/(fact(n-r)*fact(r))\n    return res\n\np
rint('nCr = ', nCr(5, 2))", "# function which doesn't take anything an
```

Below is the output of above program:

```
---- in side fun()-------
----- locals() ----------
{'n': 300, 'x': 50, 'y': 90}
----- globals() ----------
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<_frozen_importlib_external.SourceFileLoader object at 0x10b255128>, '__spec
__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-
in)>, '__file__': '/Users/nikky/PycharmProjects/Batch27/local_global.py', '_
_cached__': None, 'g': 9.8, 'fun': <function fun at 0x10b1ec268>, 'start': <
function start at 0x10b75fd90>}
---- in side fun()-------
----- locals() ----------
{'x': 20, 'n': 30}
----- globals() ----------
{'__name__': '__main__', '__doc__': None, '__package__': None, '__loader__':
<_frozen_importlib_external.SourceFileLoader object at 0x10b255128>, '__spec
__': None, '__annotations__': {}, '__builtins__': <module 'builtins' (built-
in)>, '__file__': '/Users/nikky/PycharmProjects/Batch27/local_global.py', '_
_cached__': None, 'g': 9.8, 'fun': <function fun at 0x10b1ec268>, 'start': <
function start at 0x10b75fd90>}
```

locals(): This function returns all the local identifiers(varibles and any functions) of that function.

globals(): This function returns all the global identifiers(varibles and any functions) whicha are available outside.

g is a global variable in the above program which is available in all the functions. And each function is availble to all other functions including to itslef.

global keyword:

In [26]:

```
g = 9.8

def fun():
    g = 10


def start():
    print ('Before: ', g)
    fun()
    print ('After: ', g)

start()
```

```
Before:  9.8
After:  9.8
```

In the above example, after function call fun(), g value must be changed. But it didn't happen. When we are ssigning a value into a variable, python creates a local version of the variable, unless we explicitly declare it as global. Look at the below example.

In [27]:

```
g = 9.8

def fun():
    global g
    g = 10

def start():
    print ('Before: ', g)
    fun()
    print ('After: ', g)

start()
```

```
Before:  9.8
After:  10
```

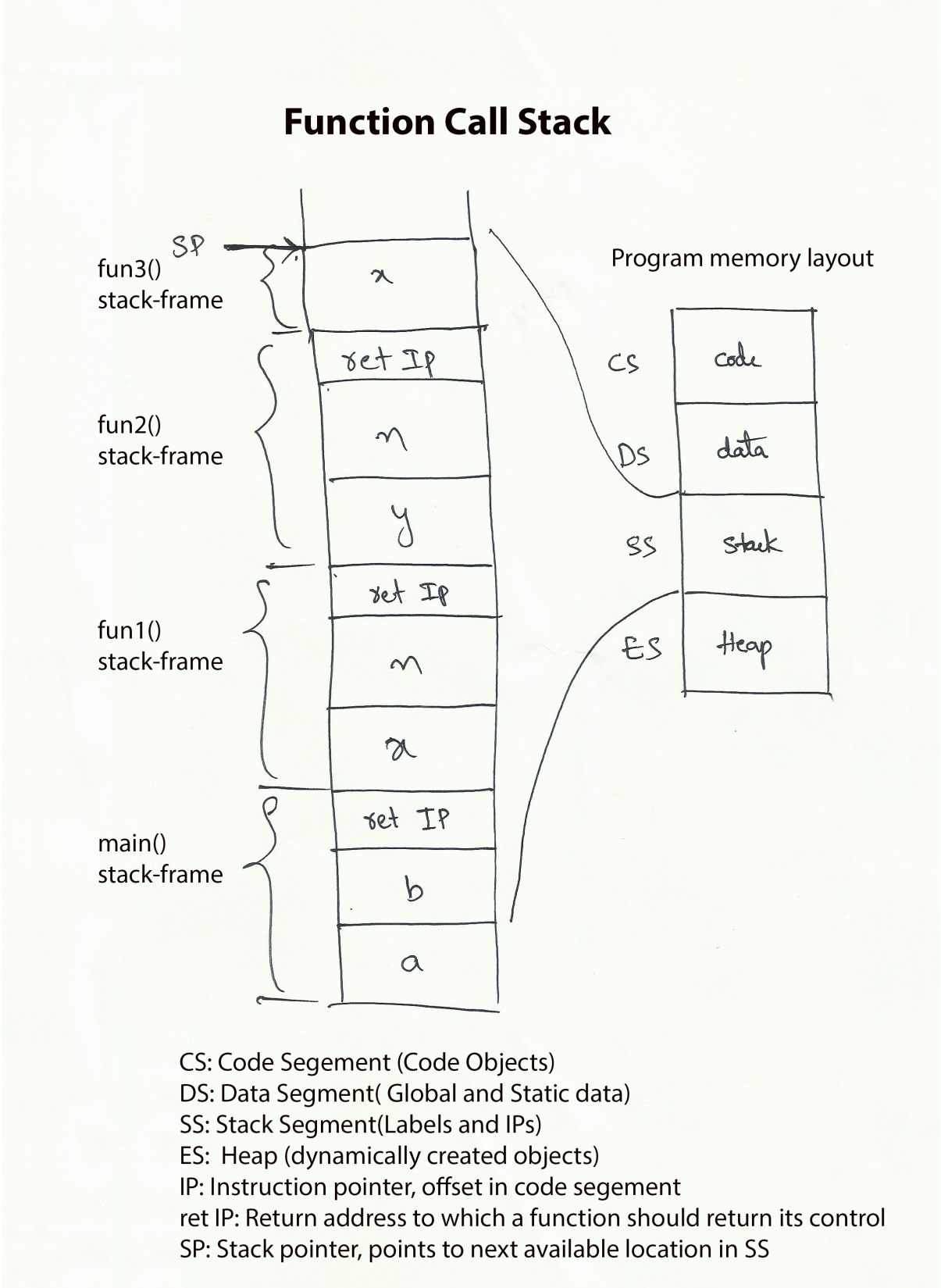
Understanding function Call stack

In the below example each function has its own set of local variables.

In [28]:

```
g = 9.8

def fun3(x):
    print ('fun3 start')
    # print 'stack frame of fun3: ', locals()
    print ('fun3 end')

def fun2(n):
    print ('fun2 start')
    y = 30
    n = n + 1
    # print 'stack frame of fun2: ', locals()
    fun3(n)
    print ('fun2 end')

def fun1(n):
    print ('fun1 start')
    n = n + 1
    x = 20
    # print 'stack frame of fun1: ', locals()
    fun2(n)
    print ('fun1 end')

def main():
    print ('main starts here')
    a = 100
    b = 200
    # print 'stack frame of fun: ', locals()
    fun1(a)
    print ('main ends here')

if __name__ == '__main__':
    main()
```

```
main starts here
fun1 start
fun2 start
fun3 start
fun3 end
fun2 end
fun1 end
main ends here
```



The place where these locals are stored is called function **stack-frame** of that function. Stack-frame also contains an instruction pointer(code address) to which control should be returned. Stack frame of a function is destroyed before control leaving the function. A function is alive as long as its stack frame resides in memory. The memory layout where all the stack frames are created is called function **call-stack**.

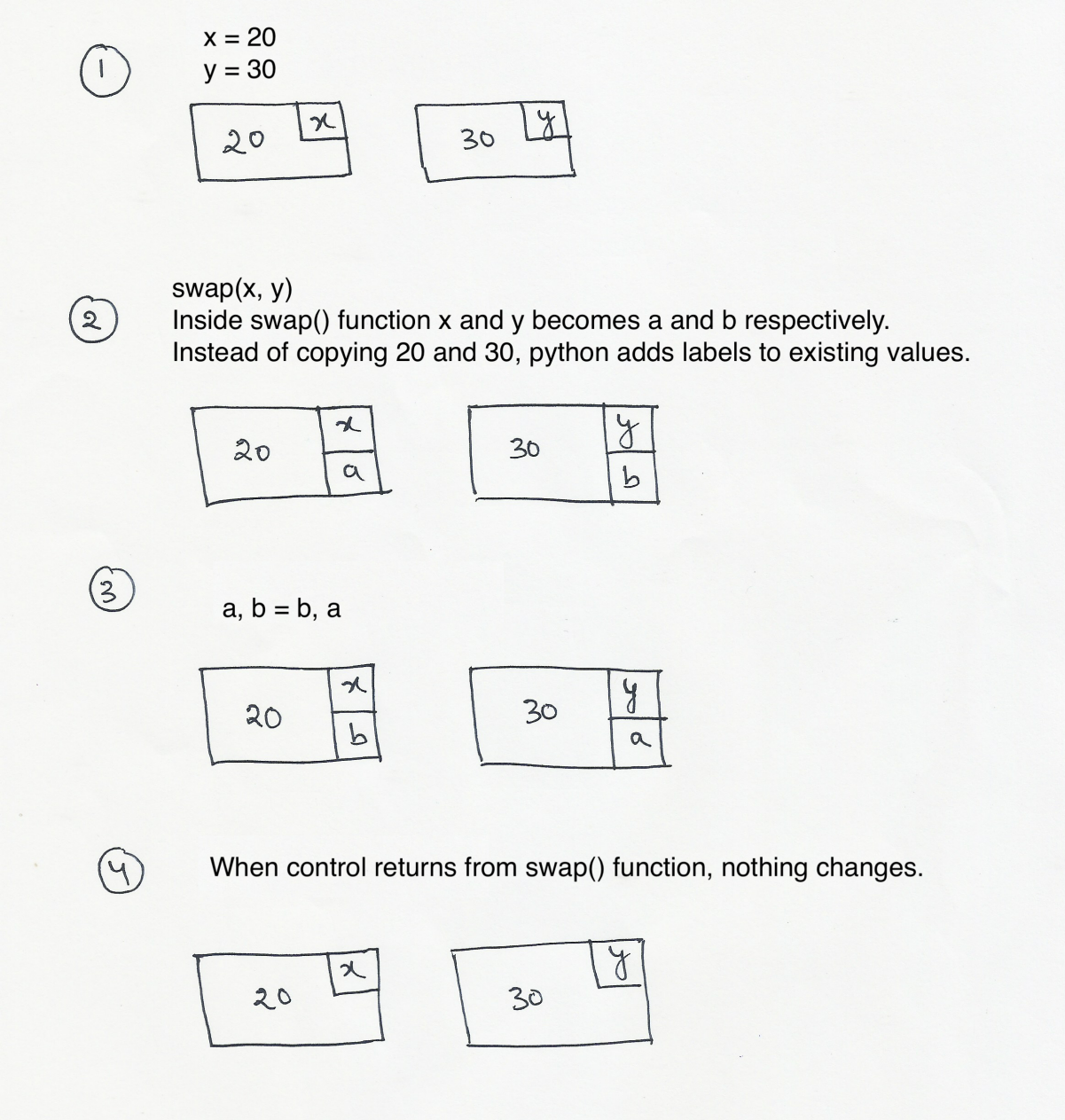
call-by-object-reference

In [29]:

```
def swap(a, b):
    a, b = b, a

x = 20
y = 30
swap(x, y)

print ('x = ', x, ' y = ', y)
```

```
x =  20  y =  30
```

In the above example, 'x' and 'y' values are not changed as 'a' an 'b' will be new lables for 20 and 30, infact we are only exchanging lables for them. As all the primitive types are immutable, there is no affect on 'x' and 'y'.

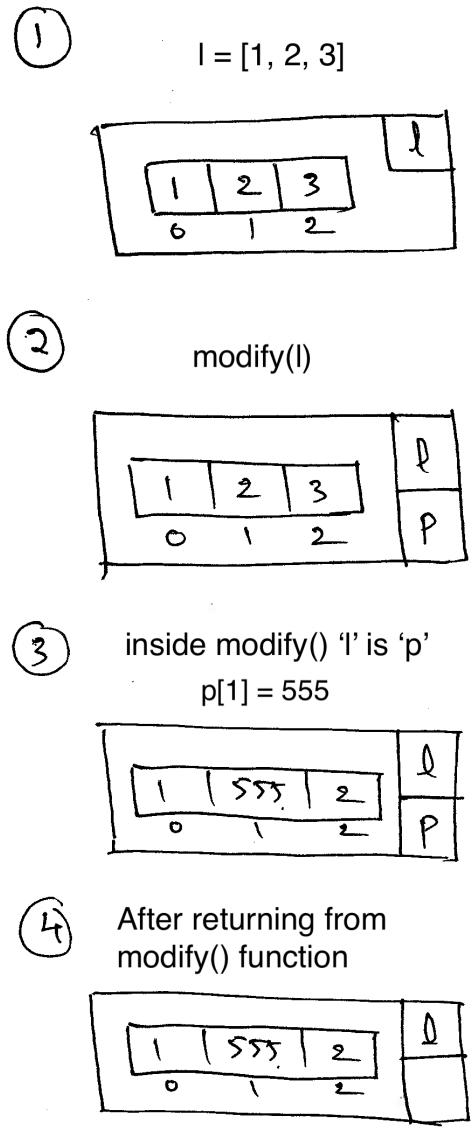


In [30]:

```
# Arguments are sent to a function by call-by-object-reference
def modify(p):
    p[1] = 555

def main():
    l = [1, 2, 3]
    modify(l)
    print(l)

if __name__ == '__main__':
    main()
```

```
[1, 555, 3]
```

In the above example p is another label for same list(i.e, l) and we are accessing individual element of list l through p. So once we come back from modify() function there will be effect on l as l and p are referring same list.



In [31]:

```
# replacing with another list

def modify(p):
    p = [4, 5, 6]

def main():
    l = [1, 2, 3]
    modify(l)
    print(l)

if __name__ == '__main__':
    main()
```

```
[1, 2, 3]
```

if we completely replace p with some other value(it can be a list or single value), there is no effect on l, afterall p is just an another label for same list. Label p moves from list [1, 2, 3] to [4, 5, 6].

How do we prevent modifying 'l' in function modify ?

Just create a copy and pass it to modify() function

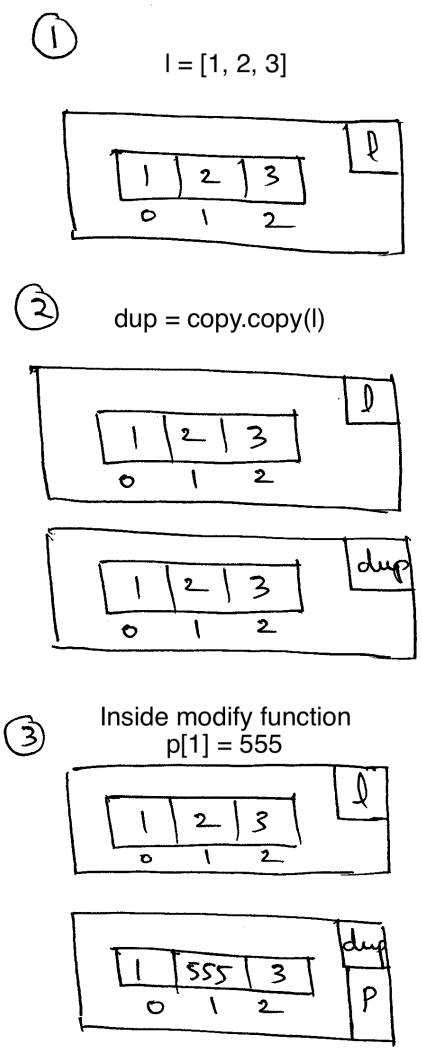
copy - shallow copy

When we want to create duplicate copy of the elements in a container(list, set, dictionary etc) we use copy function from copy module.

In [32]:

```
import copy

def modify(p):
    p[1] = 555

def main():
    l = [1, 2, 3]
    dup = copy.copy(l) # Alternative l.copy()
    modify(dup)
    print(l, dup)

if __name__ == '__main__':
    main()
```

```
[1, 2, 3] [1, 555, 3]
```



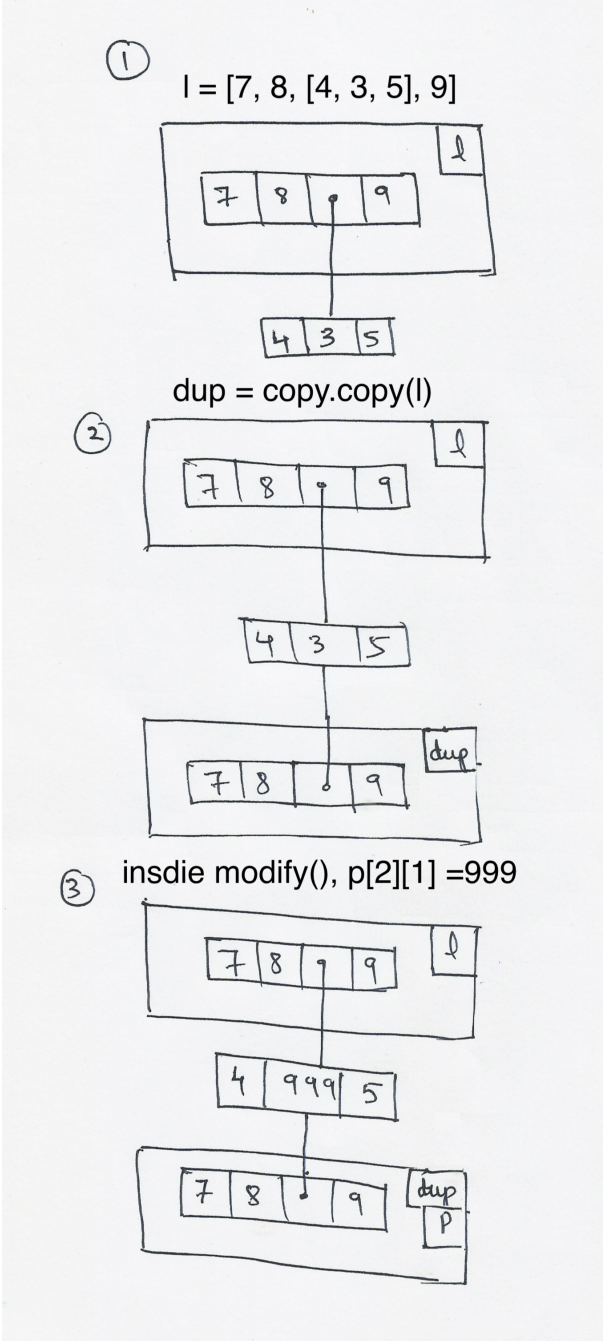
In [33]:

```
1  import copy
2
3  def modify(p):
4      p[2][1] = 999
5
6  def main():
7      l = [7, 8, [4, 3, 5], 9]
8      dup = copy.copy(l)
9      modify(dup)
10     print(l)
11
12 if __name__ == '__main__':
13     main()
```

```
[7, 8, [4, 999, 5], 9]
```

Why values in inner list are being modified?

Because copy() copies elements in the first level only. We have a solution for it.



deepcopy

copy() function only copies elements in the first level, but deepcopy copies all the objects even they are in multiple levels. e.g, list of lists, list of dictionaries etc.

In [34]:

```
import copy

def modify(p):
    p[2][1] = 999

def main():
    l = [7, 8, [4, 3, 5], 9]
    dup = copy.deepcopy(l)
    modify(dup)
    print(l)

if __name__ == '__main__':
    main()
```

```
[7, 8, [4, 3, 5], 9]
```

Recursion

"A function calling itself is called recursion."

Recursion is generally used when a problem has recursive sub problems.

In [35]:

```
def fun():
    print('Apple')
    fun()
fun()
```

```
Apple
Apple
Apple
Apple
Apple
Apple
Apple
Apple
Apple
Apple
Apple
Apple
Apple
Apple
Apple
Apple
Apple
Apple
Apple
Apple
```

Above function repeatedly prints 'Apple', and never stops. If we use the recursion with control, we can solve complex problems easily. A condition which controls recurssion is called base-case.

In [36]:

```
def fun(n):
    if n > 0: # base-case
        print(n)
        fun(n-1)
fun(5)
```

```
5
4
3
2
1
```

In [37]:

```
def fact(n):
    if n == 0 or n == 1:
        return 1
    return n*fact(n-1)

fact(4)
```

Out[37]:

```
24
```

N stairs Problem

In [38]:

```
def ways(n):

    if n == 1:
        return 1

    if n == 2:
        return 2

    return ways(n-1) + ways(n-2)


print(ways(35))
```

```
14930352
```

In [39]:

```
import time

start = time.process_time()
count = 0

def ways(n):
    global count
    count += 1

    if n == 1:
        return 1

    if n == 2:
        return 2

    return ways(n-1) + ways(n-2)


print(ways(35), '->', count)
end = time.process_time()
print(end - start)
```

```
14930352 -> 18454929
2.4263235950000004
```

In [40]:

```
import time

start = time.process_time()

cache = {2:2, 1:1}
count = 0

def ways(n):
    global count
    count += 1

    if n not in cache:
        cache[n] = ways(n-2) + ways(n-1)

    return cache[n]

print (ways(35), '->', count)

print(time.process_time() - start)
```

```
14930352 -> 67
0.00041604499999969846
```

In [41]:

```
6.50347/0.00061
```

Out[41]:

```
10661.426229508197
```

Decorator

Passing a function to another function

Functions are first class objects in python. We can pass a function to another function like any other type.

In [42]:

```
def fun():
    """
    Basic help of function
    """
    print ('Hello')

print(fun)
```

```
<function fun at 0x7fc280c5bf28>
```

In the below example greet method is passed to 'fun' function.

In [43]:

```
def greet():
    print ("Welcome Python!")

def fun(f):
    print ('*'*20)
    f()
    print ('*'*20)

fun(greet)
```

```
********************
Welcome Python!
********************
```

Defining a function within a function

In [44]:

```
def fun():
    pi = 22/7.0

    def area(r):
        a = pi* r*r
        print ("Area = ", a)

    print ("-------------------------")
    area(5)
    print ("-------------------------")

fun()
```

```
-------------------------
Area =  78.57142857142857
-------------------------
```

Note: area() is a function which is defined inside fun() function and can only be accessible to fun().

Returning a function from a function

In [45]:

```
def fun():
    pi = 22/7.0

    def area(r):
        a = pi* r*r
        print ("Area = ", a)

    return area

x = fun()
x(5)
```

```
Area =  78.57142857142857
```

In [46]:

```
fun()(5)
```

```
Area =  78.57142857142857
```

Passing a function to another function along with its arguments

Sequence unpacking:

In [47]:

```
def fun(x, y, z):
    print(x + y + z)

t = 20, 30, 40

fun(t[0], t[1], t[2])
```

```
90
```

In [48]:

```
def fun(x, y, z):
    print(x + y + z)

t = 20, 30, 40
fun(*t)
```

```
90
```

In [49]:

```
def fun(x, y, z):
    print(x + y + z)

d = {'x':20, 'y':30, 'z':40}

fun(d['x'], d['y'], d['z'])

```

```
90
```

In [50]:

```
def fun(x, y, z):
    print(x + y + z)

d = {'y':30, 'z':40, 'x':20}

fun(**d)
```

```
90
```

Passing a function to another function along with its arguments

In [51]:

```
def add(x, y):
    print(x + y)

def volume(h, l, b):
    print(h*l*b)

def compute(task, *args, **kwargs):
    task(*args, **kwargs)

compute(add, 2,3)
compute(volume, 4, 5, 6)
```

```
5
120
```

Decorator

Decorator is a design pattern, which is used to additional functionality to a function dynamically. In the below example stars function is adding addtional stars to the ouput of every function that is being passed.

Wrapper Functions:

In [52]:

```
def stars(f, *args, **kwargs):
    print("***********")
    ret = f(*args, **kwargs)
    print("***********")
    return ret

def greet():
    print("Hello World!")

stars(greet)
```

```
***********
Hello World!
***********
```

In [53]:

```
def stars(f, *args, **kwargs):
    print("***********")
    ret = f(*args, **kwargs)
    print("***********")
    return ret

def add_avg(x, y):
    print(x + y)
    return (x + y)/2

z = add_avg(20, 30)
print(z)
print()
z = stars(add_avg, 20, 30)
print(z)
```

```
50
25.0

***********
50
***********
25.0
```

Python provides smart and cleaner syntax to achieve this.

In [54]:

```
def stars(f):
    def wrapper(*args, **kwargs):
        print("***********")
        ret = f(*args, **kwargs)
        print("***********")
        return ret
    return wrapper

@stars
def add(x, y):
    s = x + y
    print(s)


def mul(x, y, z):
    p = x * y * z
    print(p)

# ------------------

r1 = add(3, 4)
r2 = mul(3, 4, 5)
```

```
***********
7
***********
60
```

In [55]:

```
def stars(f):
    def wrapper(*args, **kwargs):
        print("***********")
        ret = f(*args, **kwargs)
        print("***********")
        return ret
    return wrapper

def dash(f):
    def wrapper(*args, **kwargs):
        print("-----------")
        ret = f(*args, **kwargs)
        print("-----------")
        return ret
    return wrapper

@stars
@dash
def add(x, y):
    s = x + y
    print(s)


#@stars
def mul(x, y, z):
    p = x * y * z
    print(p)


# ------------------

r1 = add(3, 4)
r2 = mul(3, 4, 5)
```

```
***********
-----------
7
-----------
***********
60
```

Whenever we want to call the function with additional functionality, We don't need to pass the function to stars(), instead, just add @stars on the top of function definition, that adds additional functionality to function definition iteself.

Practical use-case

There are times when we want to profile the function timings. Below is the @timer decroator which prints the time taken by process function. We can also store these timings in a database table or logger for further analysis. Python decorators prevent adding unnecessary code in code base. There is only one place in the code where we have to make changes; adding decroator to function definition.

In [56]:

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        ret = func(*args, **kwargs)
        end = time.perf_counter()
        print("Time taken: ", end - start)
        return ret
    return wrapper

@timer
def process(n):
    for i in range(n):
        i = i * i

process(10000000)
```

```
Time taken:  0.382862894970458
```

Closure

Closure is the context captured by an inner function, which will be used out-side of the outer function scope, even parent is not alive.

In [57]:

```
def n_circles(n):
    pi = 22/7.0

    def area(r):
        a = pi*r*r*n
        #print ('Local vars of area:', locals())
        return a

    return area

num = 5
x = n_circles(num)
print ('Area of {} circles is {}'.format(num, x(3)))

#print ('Closure data:')
# print (x.func_closure[0].cell_contents)
# print (x.func_closure[1].cell_contents)
print(x.__closure__)
```

```
Area of 5 circles is 141.42857142857142
(<cell at 0x7fc280df6d38: int object at 0xa75020>, <cell at 0x7fc280df
6588: float object at 0x7fc2816f1810>)
```

In the above example , we are returning area() function from n_circles() functions and assigned to x. Here 'x' is 'area'. We can call x() now. If we look at the output, we can see 'n' and 'pi' as local variables of 'x'. Infact 'x' is function area(). 'n' and 'pi' are local variables of n_circles() not area(). But how area() is able to use them even

after n_circles() exit. This is called 'closure'. Function area() captured all varibales required for its execution. In python, function is an object. This function object has a property called 'func_closure', a tuple of captured values. In the above out put we can see how to access the content of closure.

Generator

"A function with 'yield' statement instead 'return' is called as generator in python." Instead of calling entire function each time. Python creates a context for function with yield statement and returns a generator object.

- Generator returns a value by executing next iteration of the generator expression.
- We can pass a generator to 'next()' function, to get the subsequent value of the generator.

Note:

- Iterators are used to traverse exisiting data
- Generators produces values on demand and also can be used as iterators

Creating Custom generators

In [58]:

```
range(10)
```

Out[58]:

```
range(0, 10)
```

In [59]:

```
def fun(n):
    i = 0
    while i < n:
        i += 1
        return i


print(fun(5))
print(fun(5))
print(fun(5))
```

```
1
1
1
```

In [60]:

```
i = 0
def fun(n):
    global i
    while i < n:
        i += 1
        return i
print(fun(5))
print(fun(5))
print(fun(5))
```

```
1
2
3
```

In [61]:

```
def fun(n):
    i = 1
    while i <= n:
        yield i
        i += 1

gen = fun(5)
```

In [62]:

```
print(gen)
```

```
<generator object fun at 0x7fc280d8faf0>
```

In [63]:

```
next(gen)
```

Out[63]:

```
1
```

In [64]:

```
gen = fun(5)
```

In [65]:

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

```
1
2
3
4
5
```

In [66]:

```
print(next(gen, None))
```

```
None
```

In [67]:

```
def fun(l=[]):
    for x in l:
        if x%5 == 0:
            yield x

l = [3, 4, 15, 12, 20, 31, 65]

gen = fun(l)
```

In [68]:

```
print(next(gen, None))
```

```
15
```

In [69]:

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

```
20
65
```

Above function call fun(5), always returns 1, as loop ends in the fist iteration iteself.

Replace 'return' wth 'yield'

In [70]:

```
def fun(n):
    i = 1
    while i <= n:
        yield i
        i += 1

gen = fun(5)
print (type(gen))
```

```
<class 'generator'>
```

now, fun(5) returns a generator. We can get next value from a generator using built-in next() function

In [71]:

```
next(gen)
```

Out[71]:

```
1
```

next value,

In [72]:

```
next(gen, None)
```

Out[72]:

```
2
```

and so on ...

We can also is this generator in for loop.

In [73]:

```
gen = fun(5)
for i in gen:
    print (i)
```

```
1
2
3
4
5
```

After values exhausted, next returns StopIteration error.

To prevent this we can have a default values after values exhausted.

In [74]:

```
print (next(gen, None))
```

```
None
```

Program: Implement xrange() function

In [75]:

```
def my_xrange(*args):
    if len(args) <=3 and len(args) >= 1:
        start = 0
        step =1

        if len(args) == 1:
            end = args[0]

        elif len(args) == 2:
            start = args[0]
            end = args[1]

        elif len(args) == 3:
            start = args[0]
            end = args[1]
            step = args[2]

        i = start
        while i < end:
            yield i
            i += step
    else:
        print ('Invalid parameter count')

```

In [76]:

```
for x in my_xrange(1, 11, 3):
    print (x)
```

```
1
4
7
10
```

Explicit iterators

We cannot resume the exection once we break the iteration of any sequence. Because we do not have the control on implicit iterator object maintained by for loop.

In [77]:

```
l = [20, 30, 10, 20, 25, 35, 25, 30, 45, 15]

for x in l:
    print (x)
```

```
20
30
10
20
25
35
25
30
45
15
```

We can use iter() function to create on iterator object, which can be controlled by the developer. If we use this iterator object to iterate a sequence, we can resume the iteration, even after breaking the iteration abruptly.

using iter()

In [78]:

```
l = [20, 30, 10, 20, 25, 35, 25, 30, 45, 15]

s = 0

for val in l:
    if s + val > 150:
        print (val)
        break
    print (val, end= ' ')
    s += val

print ('Remaining values:', end=' ')
for val in l:
    print (val, end=' ')

```

```
20 30 10 20 25 35 25
Remaining values: 20 30 10 20 25 35 25 30 45 15
```

In [79]:

```
l = [20, 30, 10, 20, 25, 35, 25, 30, 45, 15]
it = iter(l)

s = 0

for val in it:
    if s + val > 150:
        print(val)
        break
    print (val, end=' ')
    s += val

print ('Remaining values:', end=' ')

for val in it:
    print(val, end=' ')

```

```
20 30 10 20 25 35 25
Remaining values: 30 45 15
```

Exercise problems

1. Write functions to compute incometax, PF, Professional Tax and net salary per month. Assume that PF is 12% gross salary and PT is 10% of PF.
2. Write two decorators and apply on a function

Interview questions

3. How do you write custom generators?
4. What does the yield statement do?
5. What is Recursion? Give Example?
6. What is decorator, usage?
7. Write a function decorator in Python?
8. How are arguments passed by value or by reference in python?
9. Mention what are the rules for local and global variables in Python?
10. How to use *args and* *kwargs in python?
11. What are positional arguments?
12. What are default arguments?
13. What are keyword arguments?
14. What are variable arguments?
15. What are variable keyword arguments?
16. What is a closer in Python?
17. When do you use variable keyword arguments?
18. How can you get all global variables and local variables in a scope?
19. How “call by value” and “call by reference” works in Python?
20. Difference between “cmp()” function, “==” and “is”?
21. Mention the use of the split() function in Python?
22. What is the purpose of zip(), enumerate()? How to unzip list of tuples to multiple lists?
23. How can you implement functional programming and why would you?
24. What is lambda and how it works in Python?

25. How method overloading works in Python?
26. Difference between “deepcopy” and “shallow copy”
27. What is docstring in Python?
28. What is pure function?
29. What is the use of next function?
30. **Program:** Flatten the below list using recursion

```
l= [34, 5, [ 4, 5, 3, [ 3, 19, 9, 1, 2 ] , 5], 13, 4, [ 6, 14] ]
```

In [80]:

```
def calculate_income_tax(annual_salary):
    tax = 0.0
    salary = annual_salary

    if salary > 1000000:
        slab = salary - 1000000
        tax = slab * 0.3
        salary -= slab

    if salary > 500000:
        slab = salary - 500000
        tax += slab * 0.2
        salary -= slab

    if salary > 300000:
        slab = salary - 300000
        tax += slab * 0.05
        salary -= slab

    return tax

def calculate_PF(annual_salary):
    return annual_salary*0.12

def calculate_PT(annual_salary):
    return calculate_PF(annual_salary)*0.1

def net_per_month(annual_salary):
    amount_deduct = calculate_income_tax(annual_salary) \
                    + calculate_PF(annual_salary) \
                    + calculate_PT(annual_salary)
    net_month = (annual_salary - amount_deduct)/12
    return net_month

annual_salary = 1370000
print("Net salary for {} is {} per month" \
      .format(annual_salary, net_per_month(annual_salary)))

```

```
Net salary for 1370000 is 80680.0 per month
```