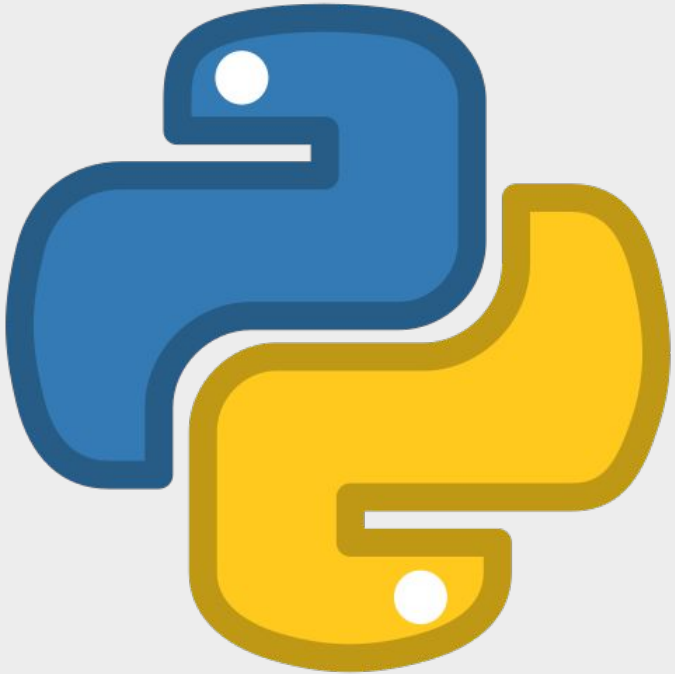


Function, List, Sort, Recursion

2021.01.03
Data eXperience Lab(Instructor: Eunil Park),
Winter Seminar
2018312824 Ryu Chaeeun

OUTLINE

1. LIST

2. FUNCTION

3. RECURSION

4. SORT

LIST

```
>>> my_list = ["a","b","c"]
>>> my_number_list = [1,3,5,7]
```

Lists are used to store multiple items in a single variable.

Lists are one of 4 built-in data types in Python used to store collections of data, along with Tuple, Set, and Dictionary.

LIST : CREATION

Creation

Brackets [] are used to create a lits.

“list()” can be used to make an empty list

Comma(,) is used to separate items in the list.

Lists can be inside lists.

```
>>> my_list = ["a","b","c"]
>>> my_list
['a', 'b', 'c']
>>> my_number_list = [1,3,5,7]
>>> my_number_list
[1, 3, 5, 7]
>>> type(my_list)
<class 'list'>
>>> type(my_number_list)
<class 'list'>
```

```
>>> a = list()
>>> a
[]
>>> type(a)
<class 'list'>
```

```
>>> dimensions = [1,2,3,["a","b","c"]]
>>> dimensions
[1, 2, 3, ['a', 'b', 'c']]
>>> type(dimensions)
<class 'list'>
```

LIST : INDEXING

Indexing

To retrieve an element of the list, we use the *index operator* ([])

- Index of a list starts from 0 to (number of items in a list -1)

```
>>> my_numbers_list = [1,2,3,4,5]
>>> my_numbers_list[0]#get the first item
1
>>> my_numbers_list[2]#get third item
3
>>> my_numbers_list[4]#get fifth item
5
>>> my_numbers_list[-1]#get last item
5
```

- Indexing a list more than one dimension

```
>>> dimensions = [1,2,3,["a","b","c"]]
>>> dimensions[0]#first item in a list
1
>>> dimensions[3]#fourth item
['a', 'b', 'c']
>>> dimensions[3][0]#first item of fourth item in a list
'a'
```

LIST : SLICING

Slicing

Use colon(:) to slice lists.

If you want to slice a list consisting of item with index
of “beginning index” to item with index of “end index”,:

List[beginning index : end index +1]

```
>>> my_numbers_list
[1, 2, 3, 4, 5]
>>> my_numbers_list[0:2]#0th index item ~ (2-1)th index item
[1, 2]
>>> my_numbers_list[:3]#get beginning ~ (3-1)th index item
[1, 2, 3]
>>> my_numbers_list[-2:]#get two last items
[4, 5]
```

FUNCTION

```
def function(param1, param2):
    print("hello")
    return param1 + param2
```

A function is a block of code which only runs when it is called.

FUNCTION : DEFINITION

Definition

A function is a block of code which only runs when it is called.

Types of function:

1. Built-in Functions:
   a. Functions that are built into Python.
   b. print, input, etc.

2. User-defined Functions: Functions defined by the users themselves.

FUNCTION : USER-DEFINED

User-defined function

Why we use:

- our program -> smaller and modular chunks.
- functions make it more organized and manageable.

When to use:

- repeatedly used codes -> functions

Features:

- Function names can be generated the same way as defining variable names
- Formal parameters: names of variables that are called when function is called and used inside the function.
- The result of function will be returned with “return” at the end of function. (Procedure: A function without “return”)

Formal parameters

```
def add_function(param1, param2):
    result = param1+param2
    return result

print(add_function(1,2))
```

```
3
>>>
```

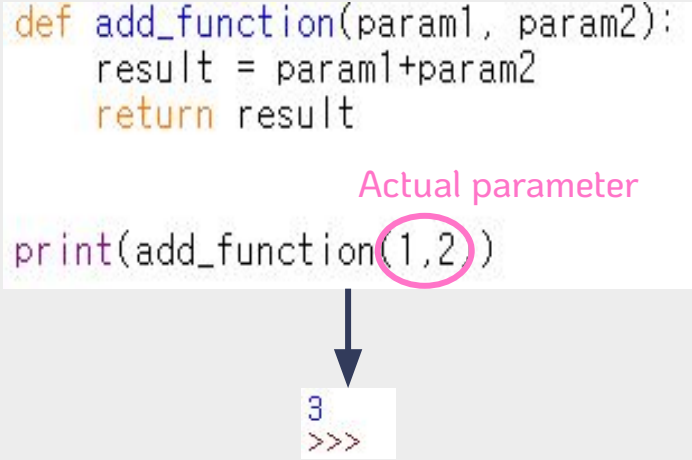
```
def procedure():
    print("Hello, I return nothing")

a = procedure()
print("returned:",a)
```

```
Hello, I return nothing
returned: None
>>>
```

FUNCTION : CALLING FUNCTION

Function call

- Only predefined function can be called.
- Actual Parameter(argument): The actual value that is passed into the method by a caller.
- Number of actual parameters = number of formal parameters



RECURSION

```
def func(): <--
              |
              | (recursive call)
              |
    func() ----
```

Recursion is the process of defining something in terms of itself.

RECURSION : CALLING FUNCTION

How it works:

Top-down method that can be used instead of iteration

Example 1

```
import time
def countdown(n):
    if n>0:
        print(n)
        time.sleep(1)#1초 동안 쉼
        countdown(n-1)
n=0 else:
        print("Go !")

countdown(5)
```

Recursion

output

```
5
4
3
2
1
Go !
>>>
```

sequence

```
countdown(5)
print(5)
time.sleep(1)
countdown(4)
print(4)
time.sleep(1)
countdown(3)
print(3)
time.sleep(1)
countdown(2)
print(2)
time.sleep(1)
countdown(1)
print(1)
time.sleep(1)
countdown(0)
Go !
```

RECURSION : CALLING FUNCTION

Example 2

```
def sigma(n):
    if n>0:
        return n+sigma(n-1)
    else:
        return 0

print(sigma(5))
```

Recursion

```
15
>>>
```

sequence

RECURSION : ANALYSIS

TIME EFFICIENCY

- In proportion to the number of function recursively called
- If function(n) -> recursively called for n times-> in proportion to argument ‘n’

SPACE EFFICIENCY

- In proportion to the number of function recursively called

  (function must remember what to do with the recursively returned value(ex. addition)

- If function(n) -> recursively called for n times-> in proportion to argument ‘n’

RECURSION : TAIL RECURSION

WHAT IS “TAIL RECURSION”

- Nothing to remember when recursion is occurred.
- -> No additional memory space

CREATION OF TAIL RECURSION

- Pre-calculate the necessary calculation and take the result of calculation as the additional argument.

```
#tail recursion
def loop(n, summation):
    if n>0:
        return loop(n-1, n+summation)
    else:
        return summation

def sigma1(n):
    return loop(n,0)


print(sigma1(5))
```

Additional argument

same

Encapsulation →

Local function

```
def sigma1(n):
    def loop(n, summation):
        if n>0:
            return loop(n-1, n+summation)
        else:
            return summation
    return loop(n,0)
```

RECURSION : TAIL RECURSION

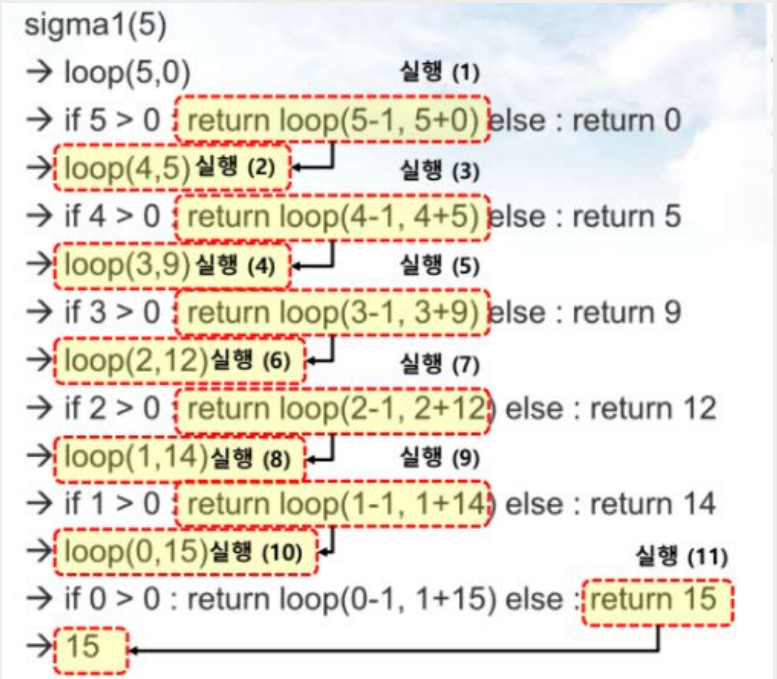
code

```
#tail recursion
                                  Additional argument
def loop(n, summation):
    if n>0:
        return loop(n-1, n+summation)
    else:
        return summation

def sigma1(n):
    return loop(n,0)


print(sigma1(5))
```

output

```
15
>>>
```

sequence



TAIL RECURSION : ANALYSIS

TIME EFFICIENCY

- In proportion to the number of function recursively called
- If function(n) -> recursively called for n+1 times-> in proportion to argument ‘n’

SPACE EFFICIENCY

- Consistent regardless of the number of recursion

```
def sigma(n):
    if n>0:
        return n+sigma(n-1)
    else:
        return 0

print(sigma(5))
```

NORMAL RECURSION

VS

```
#tail recursion
def loop(n, summation):
    if n>0:
        return loop(n-1, n+summation)
    else:
        return summation

def sigma1(n):
    return loop(n,0)

print(sigma1(5))
```

TAIL RECURSION

- Space efficiency enhanced in tail recursion.

1. SELECTION SORT
2. INSERTION SORT
3. MERGE SORT
4. QUICK SORT
5. BUBBLE SORT

SORT

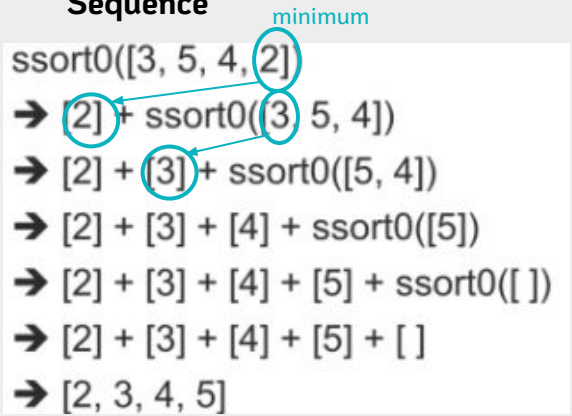
the arrangement of data in a prescribed sequence.

SORT : SELECTION SORT

Definition

Selection Sort sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning.

Sequence



Implementation

```
def ssort0(s):#selection sort (s:list)

    if s != []:#if list(s) is not empty
        smallest = min(s)#find the minimum value in s
        s.remove(smallest)#remove the minimum value
        return [smallest]+ssort0(s)#put at the beginning
    else:#when it is empty
        return []
```

```
print(ssort0([3,5,4,2]))
```

```
[2, 3, 4, 5]
```

SORT : INSERTION SORT

Definition

Insertion sort is a sorting algorithm that places an element at its suitable place in each iteration.

How it works

The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part

Code

Insert(x,ss):
Put ‘x’ into the sorted list ‘ss’ in right order.

insert(**6**, [2, 4, 5, 7, 8]) ➔ [2, 4, 5, **6**, 7, 8]

```
def isort0(s):#insertinon sort

    if s!=[]:#if list to sort is not empty
        return insert(s[0], isort0(s[1:]))

    else:#if there is nothing to sort
        return []
```

sequence

isort0([3, 5, 4, 2]) S: [3,5,4,2]
➔ insert(3, isort([5, 4, 2]))
➔ insert(3, insert(5, isort0([4, 2])))
➔ insert(3, insert(5, insert(4, isort0([2]))))
➔ insert(3, insert(5, insert(4, insert(2, isort0([ ])))))
➔ insert(3, insert(5, insert(4, insert(2, [ ]))))
➔ insert(3, insert(5, insert(4, [2])))
➔ insert(3, insert(5, [2, 4]))
➔ insert(3, [2, 4, 5])
➔ [2, 3, 4, 5]

SORT : INSERTION SORT

Implementation

```
def insert(x,ss):
    if ss != []:
        if x <= ss[0]:#put at the beginning
            ss = [x]+ss
            return ss
        if x>ss[-1]:#put at the end
            ss = ss + [x]
            return ss
        for idx in range(len(ss)-1):#find where to put x
            left = ss[idx]
            right = ss[idx+1]
            if left <= x and x<= right:#x should be put in the middle of left and right
                left_ss = ss[:idx+1]
                right_ss = ss[idx+1:]
                ss = left_ss + [x]+right_ss
                return ss
    else:
        return [x]


def isort0(s):#insertinon sort

    if s!=[]:#if list to sort is not empty
        return insert(s[0], isort0(s[1:]))

    else:#if there is nothing to sort
        return []
```

```
print(isort0([3,5,4,2]))
[2, 3, 4, 5]
>>>
```

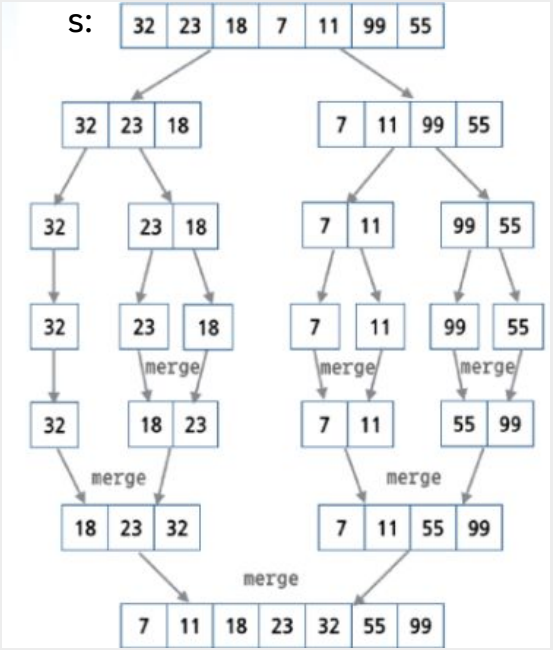
Returned ss from insert function in order

```
initial ss: [3, 5, 4, 2]
[]
[2, 4]
[2, 4, 5]
[2, 3, 4, 5]
```

SORT : MERGE SORT

How it works

In merge sort, the array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.



Algorithm

```
s: unsorted list

if len(s)>1:
  - split s in half and finish sorting through recursion respectively.
  - Merge the splitted pieces by traversing from the front and selecting the minimum value
else:
  - when len(s)<=1
  - no need for sorting, so leave it be.
```

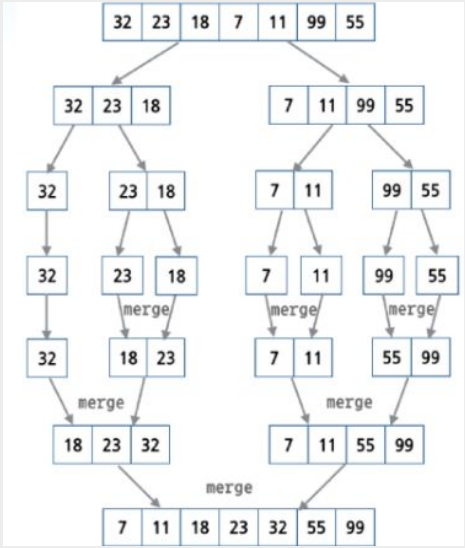
SORT : MERGE SORT

Implementation

```
def merge(left, right):
    if left != [] and right != []:# both not empty
        if left[0]<=right[0]:#compare the first value of both splits
            return [left[0]]+merge(left[1:],right)

        else:#right[0]>left
            return [right[0]]+merge(left,right[1:])
    else:
        return left+right

def msort(s):#merge sort

    if len(s)>1:
        mid = len(s)//2#index for splitting in half
        return merge(msort(s[:mid]),msort(s[mid:]))#return the two halves
    else:
        return s
```

```
print(msort([32,23,18,7,11,99,55]))
```

```
[7, 11, 18, 23, 32, 55, 99]
>>>
```



splits

```
[32, 23, 18, 7, 11, 99, 55]
[32, 23, 18]              [7, 11, 99, 55]
[32]              [23, 18]
[23]              [18]
[7, 11]               [99, 55]
[7]               [11]
[99]              [55]
```

SORT : QUICK SORT

How it works

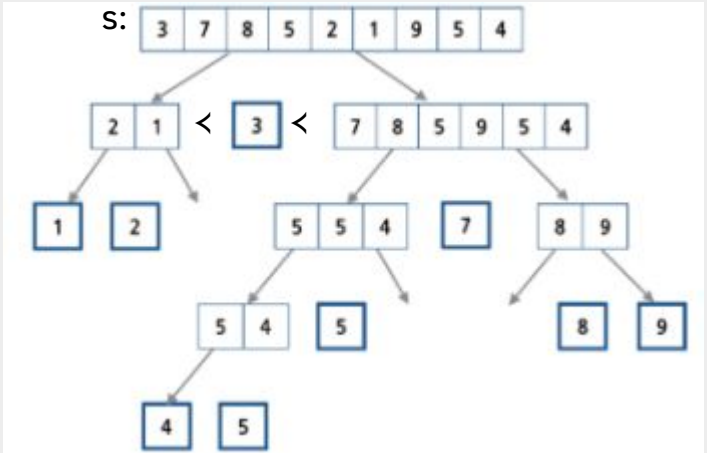
Quicksort works by selecting an element called a pivot and splitting the array around that pivot

Algorithm

```
s: unsorted list

if len(s)>1:
  - Select a value to use as pivot. (For convenience, first value)
  - Move the values in list based on pivot value.
    → pivot<value: move left to pivot
    → pivot>=value: move right to pivot
  - sort left list and right rest by recursion, and place pivot in the middle

else:
  - when len(s)<=1
  - no need for sorting, so leave it be.
```

Sequence

Pivot is in darker blue colored square



SORT : QUICK SORT

Implementation

```
def partition(pivot, s):#partitions list based on pivot value
    left, right = [],[]
    for x in s:#for value in list
        if x<= pivot:# if value <= pivot
            left.append(x)#save the value in left
        else:#if value > pivot
            right.append(x)#save the value in right
    return left, right

def qsort(s):#quick sort
    if len(s)>1:
        pivot = s[0]#set pivot as the first value
        (left, right) = partition(pivot, s[1:])
        return qsort(left)+[pivot]+qsort(right)#put pivot in the middle
    else:
        return s
```

The more the left, the smaller the value, the more the right, the bigger the value

```
s = [3,7,8,5,2,1,9,5,4]
print(qsort(s))
```

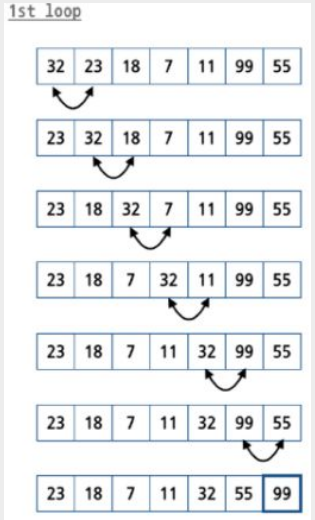
```
[1, 2, 3, 4, 5, 5, 7, 8, 9]
>>>
```

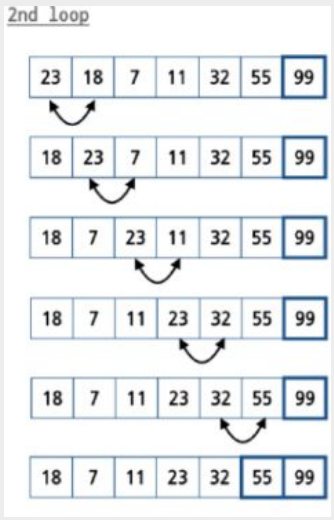
SORT : BUBBLE SORT

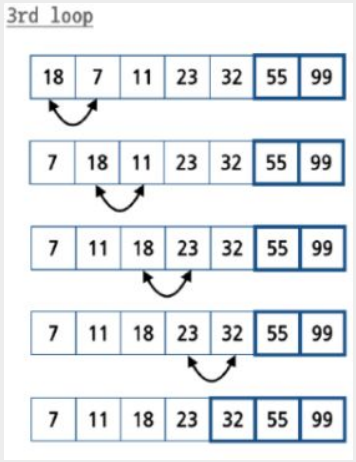
How it works

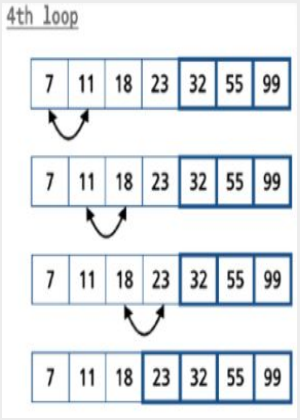
Bubble Sort compares two adjacent values in list, and swap their places, if the order is not right. It is an in-place sort that does not take up additional space, unlike the previous four sorting algorithms.

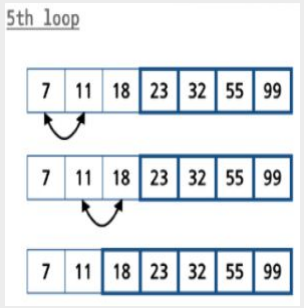
Sequence n-1 loops

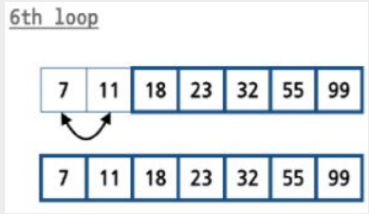












SORT : BUBBLE SORT

Implementation

```
def bubblesort(arr):
    n = len(arr)#number of values in list
    for i in range(n-1): loop
        for j in range(0,n-i-1):
            #swap
            if arr[j]>arr[j+1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr
```

```
arr = [32,23,18,7,11,99,55]
print(bubblesort(arr))
```

```
[7, 11, 18, 23, 32, 55, 99]
>>>
```