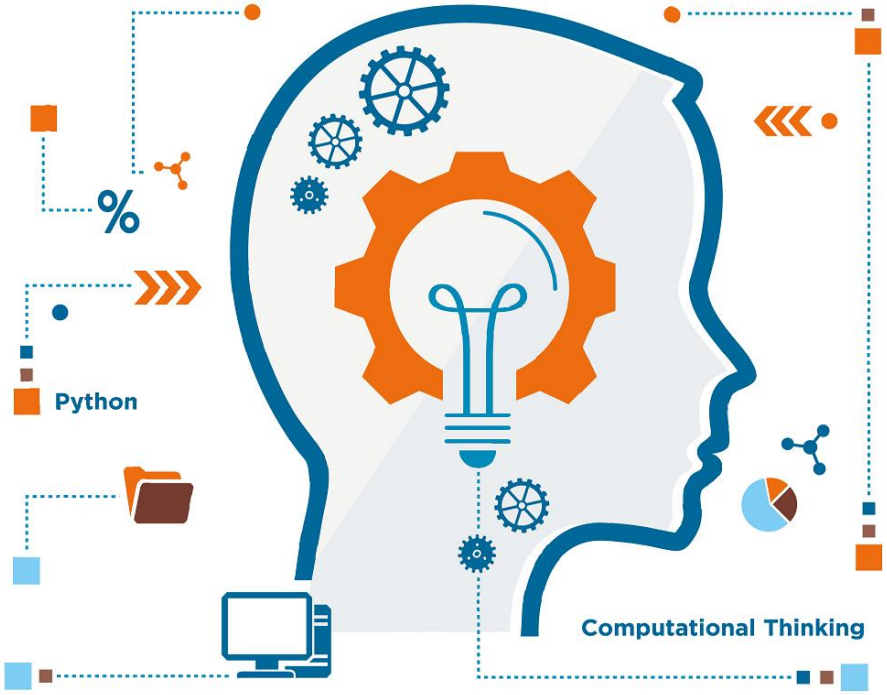



[Python]

Python으로 배우는

# 소프트웨어 원리

## Chapter 10. 제귀함수의 활용과 Class

# 목차

1. 재귀 함수
2. 클래스와 객체
3. 재귀 함수와 클래스의 활용

# 01

# 재귀함수

# 01. 재귀 함수

## I. 재귀 함수

- 재귀 함수(Recursive Function)
  - 함수 내에서 자기 자신을 다시 호출(재귀 호출)하는 형태의 함수
  - 함수가 종료되기 전에 재귀 호출이 일어나므로 언제인가는 끝나고 복귀할 수 있는 조건으로 변경된 (매개변수)조건으로 재귀 호출을 해야한다.
- 재귀함수 조건
  - 재귀함수는 **변경된 조건으로 재귀 호출**되어야 한다.
  - 재귀함수는 **호출한 함수로의 복귀**도 반드시 포함되어야 한다.

```
##재귀함수 예제 : n!
def mul(num):  #재귀 호출 함수
    if num > 1: #재귀 호출 조건
        return num * mul(num-1)   #재귀 호출: 호출 조건 값(num)의 변화
    else:
        return 1  #재귀 호출 종료 후 복귀

n = int(input(">누적 곱할 끝 수? "))
result = mul(n)
print(result)
```

# 02. 함수의 사용

## I. 재귀 함수 호출

- **함수 호출 및 복귀 확인**

```
def mul(num):  #재귀 호출 함수
    print(">Call %d" %num)
    if num > 1: #재귀 호출 조건
        result = num * mul(num-1)  #재귀 호출: 호출 조건 값(num)의 변화 필수
        print(">Return %d" %result)
        return result
    else:
        return 1  #재귀 호출 종료 후 복귀

n = int(input(">누적 곱할 끝 수? "))
result = mul(n)
print(">>", result)
```

# 02

# Class와 객체

# 02. Class와 객체

## I. Class와 객체

- 클래스(Class)와 객체(Object)는 객체지향 프로그래밍의 기본 개념
- **클래스(Class)**
  - 객체를 정의하기 위한 템플릿 또는 설계도 역할
  - 속성(attribute)과 동작(behavior)을 정의하는 변수와 메서드의 집합
  - 객체를 생성하기 위한 기본 틀이며, 여러 객체들을 생성할 수 있음
- **객체(Object)**
  - 클래스의 인스턴스(Instance), 즉 클래스를 토대로 실제로 메모리에 할당된 데이터
  - 클래스로부터 생성된 구체적인 개체로, 클래스에 정의된 속성과 동작을 가짐
  - 실제로 메모리에 할당되어 동작하는 실행 단위

```
class Men:
    def __init__(self, value):
        #self는 Node 자신 객체 의미 (__init__는 생성자 함수 의미, 객체 생성 시 자동 실행)
        self.id = value
        self.name = None
        self.sex = None
        self.age = None
```

# II. Class로 객체 생성

- Class의 생성자 함수가 자동으로 실행되어 객체가 생성된다.
- **생성자 함수(Class)**
  - Class로 객체를 생성할 때 자동으로 실행되어 객체를 생성해주는 Class 내의 특벼롼 함수
  - def __init__(self): 로 정의

```
class Men:
    def __init__(self, value):
        #self는 Node 자신 객체 의미 (__init__는 생성자 함수 의미, 객체 생성 시 자동 실행)
        self.id = value
        self.name = None
        self.sex = None
        self.age = None

men1 = Men('101')    #객체 생성 (생성자 함수가 작동)
print(men1.id)
men1.name = 'Kims'  #객체를 통한 변수 접근
print(men1.id, men1.name)
```

# III. Class의 메서드 정의 및 사용

- **메서드(Method)**
  - **메서드**는 클래스 내에 정의된 함수
  - 클래스명.메서드함수명() 형태로 호출하여 사용
  - 클래스 내의 변수는 클래스명.변수명 으로 접근

```
class Men:
    def __init__(self, value):
        #self는 Node 자신 객체 의미 (__init__는 생성자 함수 의미, 객체 생성 시 자동 실행)
        self.id = value
        self.name = None
        self.sex = None
        self.age = None

    def set_men(self, name, sex=None, age=None):
        self.name = name
        self.sex = sex
        self.age = age

men1 = Men('101')    #객체 생성 (생성자 함수가 작동)
print(men1.id)
men1.set_men(name = 'Kims', age=22, sex='F')
print(men1.id, men1.name, men1.sex, men1.age)
```

# III. Class의 메서드 정의 및 사용

- **[실습] get_men() 메서드 생성**

Ch10-ClassMen.py

◆ 클래스에 get_men() 메서드를 추가하시오..

- ➢ 해당 객체의 (id, name, sex, age) 값을 반환한다.
- ➢ sex나 age 값이 없으면 None으로 반환

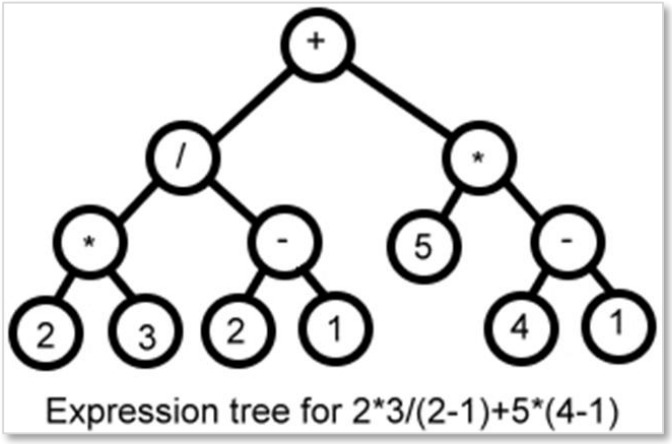
```
class Men:
    def __init__(self, value):  #self는 Node 자신 객체 의미 (__init__는 생성자 함수 의미, 객체 생성 시 자동 실행)
        self.id = value
        self.name = None
        self.sex = None
        self.age = None

    def set_men(self, name, sex=None, age=None):
        self.name = name
        self.sex = sex
        self.age = age

men1 = Men('101')    #객체 생성 (생성자 함수가 작동)
print(men1.id)
men1.set_men(name = 'Kims', age=22, sex='F')
men1.get_men()
id, name, sex, age = men1.get_men()
print(id, name, sex, age)
```

# 02. Class와 객체

## I. Class로 2진트리 구성

### ▪ 이진 트리(Binary Tree)

- 각 노드(Root Node)가 최대 두 개의 자식 노드(Prarent Node)를 가질 수 있는 트리(Tree) 구조
- 계층적으로 구성된다.



Expression tree for 2*3/(2-1)+5*(4-1)

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
```

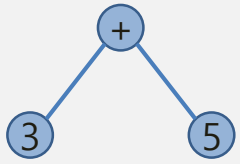
# 02. Class와 객체

## I. Class로 2진트리 구성

### ▪ 이진 트리(Binary Tree)

- 각 노드(Root Node)가 최대 두 개의 자식 노드(Prarent Node)를 가질 수 있는 트리(Tree) 구조
- 계층적으로 구성된다.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

root = Node('+')          #Node Class로 2진 트리 객체 root 생성하여 value 값을 '+'로 초기화
root.left = Node('3')     #root 객체의 left 변수에 새로운 노드를 초기화 하여 대입(연결)
root.right = Node('5')    #root 객체의 right 변수에 새로운 노드를 초기화 하여 대입(연결)
print(root.value)
print(root.left.value)
print(root.right.value)
```



# 02. Class와 객체

## I. Class로 2진트리 구성

- **[실습] 노드의 value, left.value, right.value 검색**

◆ get_node() 메서드를 만들어서 메서드의 value, left.value, right.value 를 반환받는다.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


def get_node(self):
    ##작성할 부분 ##
    return self.value, left, right


root = Node('+')          #Node Class로 2진 트리 객체 root 생성하여 value 값을 '+'로 초기화
root.left = Node('3')     #root 객체의 left 변수에 새로운 노드를 초기화 하여 대입(연결)
root.right = Node('5')    #root 객체의 right 변수에 새로운 노드를 초기화 하여 대입(연결)
print("Root :", root.get_node())
print("Left :", root.left.get_node())
print("Right:", root.right.get_node())
```

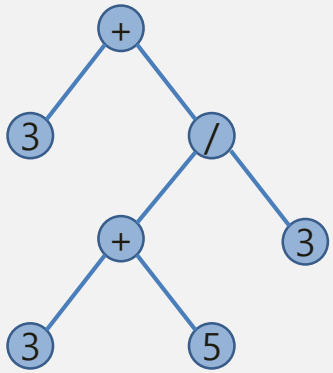
# 02. Class와 객체

## I. Class로 2진트리 구성

- **[실습] 노드의 value, left.value, right.value 검색** Ch10-ClassTree.py

◆ 아래의 2진트리를 구성하고 get_node() 메서드로 각 노드를 확인하여보시오.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def get_node(self):
     if self.left is None:
            left = None
        else:
            left = self.left.value
        if self.right is None:
            right = None
        else:
            right = self.right.value
    return self.value, left, right

root = Node('+')  #Node Class로 2진 트리 객체 root 생성하여 value 값을 '+'로 초기화
root.left = Node('3')  #root 객체의 left 변수에 새로운 노드를 초기화 하여 대입(연결)
root.right = Node('5')  #root 객체의 right 변수에 새로운 노드를 초기화 하여 대입(연결)
print("Root :", root.get_node())
print("Left :", root.left.get_node())
print("Right:", root.right.get_node())
```




```
Root : ('+', '3', '/')
Left : ('3', None, None)
Right: ('/', '*', '3')
Right>Left : ('*', '3', '5')
Right>Right: ('3', None, None)
```

# 03

# 재귀 함수와 Class의 활용

# 03. 재귀함수와 Class의 활용

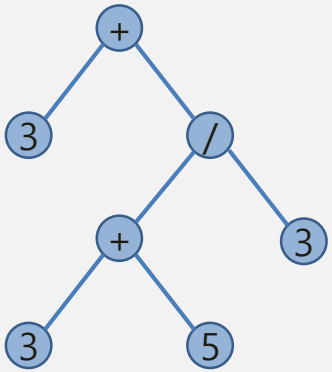
## I. 재귀함수로 2진트리 탐색

### ▪ [실습] 재귀함수로 2진트리 탐색

◆ 아래의 2진트리를 left root right 노드 순으로 탐색하는 in_order_traverse() 함수이다.

- pre_order_traverse()도 작성: root left left 순으로 탐색
- post_order_traverse()도 작성: left left root 순으로 탐색

```
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def in_order_traverse(node):
    if node is not None:
        in_order_traverse(node.left)  # 왼쪽 서브트리 중위 순회
        print(node.value, end=' ')  # 현재 노드 방문
        in_order_traverse(node.right)  # 오른쪽 서브트리 중위 순회

root = Node('+')  #Node Class로 2진 트리 객체 root 생성하여 value 값을 '+'로 초기화
root.left = Node('3')  #root 객체의 left 변수에 새로운 노드를 초기화 하여 대입(연결)
root.right = Node('5')  #root 객체의 right 변수에 새로운 노드를 초기화 하여 대입(연결)
root.right.left = Node('*')
root.right.left.left = Node('3')
root.right.left.right = Node('5')
root.right.right = Node('3')
print("<In-Order>", end=' ')
in_order_traverse(root)  #left > root > right 순회
```



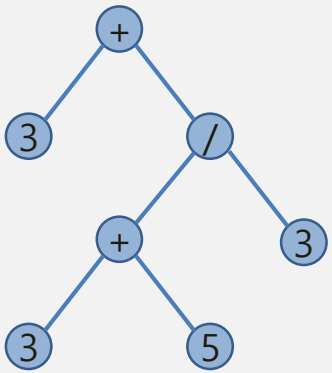
# 03. 재귀함수와 Class의 활용

## I. 재귀함수로 2진트리 구성

### ▪ [실습] 재귀함수로 2진트리 구성 (1)

◆ 다항 연산식을 입력받아 2진트리로 구성하는 construct_binary_tree() 함수이다.
- find_priority_operator()는 ()에 의한 우선순위가 높은 연산자를 식별

```
operators = {'+': 1, '-': 1, '*': 2, '/': 2}  #연산자 우선순위 고려

def find_priority_operator(expression):
    global operators
    #우선순위가 가장 빠른 연산자의 index를 찾음
    operator_index = -1    #선택된 연산자 index
    max_precedence = -1    #가장 빠른 연산 우선순위 index
    parentheses_count = 0  #()의 깊이

    for i in range(len(expression) - 1, -1, -1):
        char = expression[i]
        if char == ')':
            parentheses_count += 1
        elif char == '(':
            parentheses_count -= 1
        elif char in operators and parentheses_count == 0:  #operater이고 ()로 완성되면
            if operators[char] >= max_precedence:
                operator_index = i
                max_precedence = operators[char]
                return i
    return -1
```



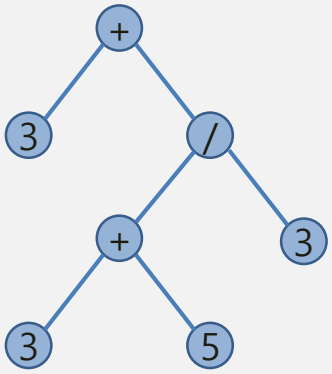
# 03. 재귀함수와 Class의 활용

## I. 재귀함수로 2진트리 구성

### ▪ [실습] 재귀함수로 2진트리 구성 (2)

◆ 다항 연산식을 입력받아 2진트리로 구성하는 construct_binary_tree() 함수이다.
- find_priority_operator()는 ()에 의한 우선순위가 높은 연산자를 식별

```
def construct_binary_tree(expression):
    global operators
    if expression[0] == '(':  #끝의 (나 )를 제거
        expression = expression[1:-1]
    if len(expression) == 0:
        return None

    #우선순위가 가장 빠른 연산자의 index를 찾음
    operator_index = find_priority_operator(expression)
    if operator_index == -1: # 연산자가 없으면 피연산자로 간주
        node = Node(expression)
    else: # 연산자를 기준으로 왼쪽과 오른쪽의 표현식을 나눕니다.
        operator = expression[operator_index]
        left_expression = expression[:operator_index]
        right_expression = expression[operator_index+1:]

        # 노드를 생성하고 재귀적으로 왼쪽과 오른쪽 서브트리를 구성
        node = Node(operator)
        node.left = construct_binary_tree(left_expression)
        node.right = construct_binary_tree(right_expression)

    return node
```



# 03. 재귀함수와 Class의 활용

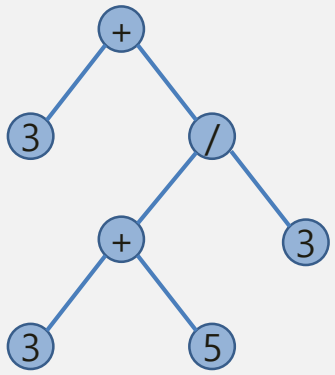
## I. 재귀함수로 2진트리 구성

- **[실습] 재귀함수로 2진트리 구성 (3)**

Ch10-ClassTreeConstruct.py

◆ 다항 연산식을 입력받아 2진트리로 구성하는 construct_binary_tree() 함수이다.
- find_priority_operator()는 ()에 의한 우선순위가 높은 연산자를 식별

```
def in_order_traverse(node):
    if node is not None:
        in_order_traverse(node.left)  # 왼쪽 서브트리 중위 순회
        print(node.value, end=' ')  # 현재 노드 방문
        in_order_traverse(node.right)  # 오른쪽 서브트리 중위 순회

## Main 부분
expression = "3+((3*5)/3)"
expression = expression.replace(' ','')
root = construct_binary_tree(expression)
print("<In-Order>", end=' ')
in_order_traverse(root)   #left > root > right 순회
print("₩n<Pre-Order>", end=' ')
pre_order_traverse(root)  #Root > left > right 순회
print("₩n<Post-Order>", end=' ')
post_order_traverse(root) #left > right > root 순회
```



# 03. 재귀함수와 Class의 활용

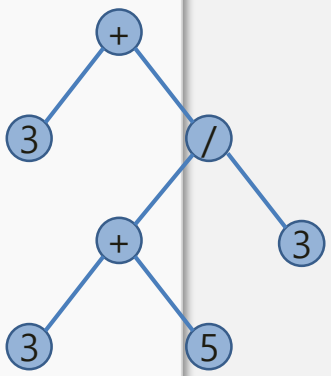
## II. 2진트리 다항식을 스택을 이용한 2항 연산 반복 처리

### ▪ [실습] 다항식 연산을 2항 연산의 반복으로 해결 (1)

◆ 다항 연산식을 2진트리로 구성한 후에 sub-tree에 대한 2항 연산을 반복하여 해결
- sub-tree에 대한 2항 연산은 피연산자 2개를 Push 한 후에 Pop 하여 연산하고 그 결과를 Push하여 다음 연산에 활용 과정을 반복적으로 재귀함수 처리

```
## Main 부분
expression = "3+((3*5)/3)"
expression = expression.replace(' ','')
root = construct_binary_tree(expression)
print("₩n[Post-Order traverse]", end=' ')
post_order_traverse(root) #left > right > root 순회
print('₩n')


result_stack = evaluate_binary_tree(root)
result = result_stack[0]
print("₩n[Last result]", result)
```

```
[Post-Order traverse] 3 3 5 * 3 / +

<<Push: 3
<<Push: 3
<<Push: 5
>>Pop : 3
>>Pop : 5
[Operation]) *
<<Push(result): 15
<<Push: 3
>>Pop : 15
>>Pop : 3
[Operation]) /
<<Push(result): 5.0
>>Pop : 3
>>Pop : 5.0
[Operation]) +
<<Push(result): 8.0

[Last result] 8.0
```



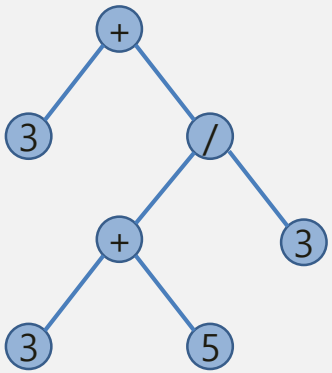
# 03. 재귀함수와 Class의 활용

## II. 2진트리 다항식을 스택을 이용한 2항 연산 반복 처리

- [실습] 다항식 연산을 2항 연산의 반복으로 해결 (2)

◆ 다항 연산식을 2진트리로 구성한 후에 sub-tree에 대한 2항 연산을 반복하여 해결
- sub-tree에 대한 2항 연산은 피연산자 2개를 Push 한 후에 Pop 하여 연산하고 그 결과를 Push하여 다음 연산에 활용 과정을 반복적으로 재귀함수 처리

```
def evaluate_binary_tree(root):
    stack = []
    if root is not None:
        stack.extend(evaluate_binary_tree(root.left))
        stack.extend(evaluate_binary_tree(root.right))
        if root.value.isdigit():
            stack.append(int(root.value))
            #print("<<Push:", root.value)
        else:
            operand2 = stack.pop()
            operand1 = stack.pop()
            #print(">>Pop :", operand1)
            #print(">>Pop :", operand2)
            result = perform_operation(root.value, operand1, operand2)
            print("[Operation])", root.value)
            stack.append(result)
            #print("<<Push(result):", result)
    return stack
```



# 03. 재귀함수와 Class의 활용

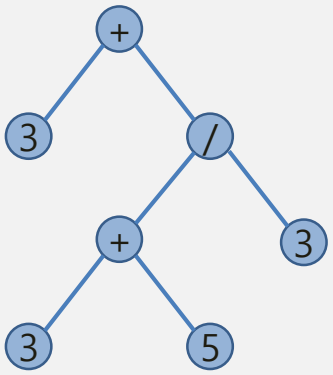
## II. 2진트리 다항식을 스택을 이용한 2항 연산 반복 처리

- **[실습] 다항식 연산을 2항 연산의 반복으로 해결 (3)** Ch10-ClassTreeOperation.py

◆ 다항 연산식을 2진트리로 구성한 후에 sub-tree에 대한 2항 연산을 반복하여 해결

- sub-tree에 대한 2항 연산은 피연산자 2개를 Push 한 후에 Pop 하여 연산하고 그 결과를 Push하여 다음 연산에 활용 과정을 반복적으로 재귀함수 처리

```
def perform_operation(operator, operand1, operand2):
    if operator == '+':
        return operand1 + operand2
    elif operator == '-':
        return operand1 - operand2
    elif operator == '*':
        return operand1 * operand2
    elif operator == '/':
        return operand1 / operand2
```



# Thank You !