# Lecture #10 | 재귀호출 (recursion)

SE213 프로그래밍 (2019)

Written by Mingyu Cho
Edited by Donghoon Shin

# 공지

- 3rd assignment
  - 5/8 ~ 5/20 (midnight)
  - 3 problems at dgist.elice.io

# 지난 시간에 다룬 내용

- 객체지향 프로그래밍 (Object-Oriented Programming)
  - 객체지향 프로그래밍 소개
  - 파이썬과 객치지향 프로그래밍
  - 클래스 정의
  - 클래스 변수/함수의 사용
  - 인스턴스 변수/함수의 사용

# 오늘 다룰 내용

- 재귀호출
  - 정의와 구성
  - 예시
    - Factorial
    - 리스트 원소들의 합
    - 피보나치 수열
  - 동적계획법(Dynamic programming) or Memoization
  - 재귀호출과 반복문 비교
  - 분할정복법(divide and conquer)

# 재귀호출 (recursion)

- 정의
  - 스스로를 호출하는 함수
  - 이러한 함수를 이용한 프로그래밍 방법론
- 사용하는 목적
  - 큰 문제를 작은 문제로 나누어 해결할 때 주로 사용
    - 분할정복법(divide-and-conquer)을 이용할 때 주로 이용
  - 복잡한 문제를 비교적 간결하게 표현할 수 있음
- 참고
  - 수학에서의 점화식(recurrence formula or recursion formula)
  - 꿈 속의 꿈 속의 꿈 속의 꿈... (인셉션?)

# 재귀함수의 구성

- 재귀 함수의 구성 요소
  - Base case: 문제의 해결이 매우 간단한 경우
  - Recursive case: 작은/간단한 문제의 답을 이용하여, 더 큰/복잡한 문제의 답을 표시하는 경우
  - 참고: 일반적으로, 함수의 앞에 base case, 뒤에 recursive case 작성
- 데이터(정보)의 전달
  - 문제의 크기/데이터를 인자를 사용하여 전달
  - 문제의 답은 반환값/변환가능한 자료구조(`list`, `dict`) 등을 이용하여 전달
- 함수를 호출할 때마다, 매개변수와 지역변수를 위한 공간이 생성됨
  → 각각의 함수호출할 때 만들어지는, 매개변수와 지역변수의 값들은 독립적임

# 예시: 팩토리얼

- 초기항: 0! = 1
- 점화관계식: n! = n * (n-1)! for n = 1, 2, 3, ...

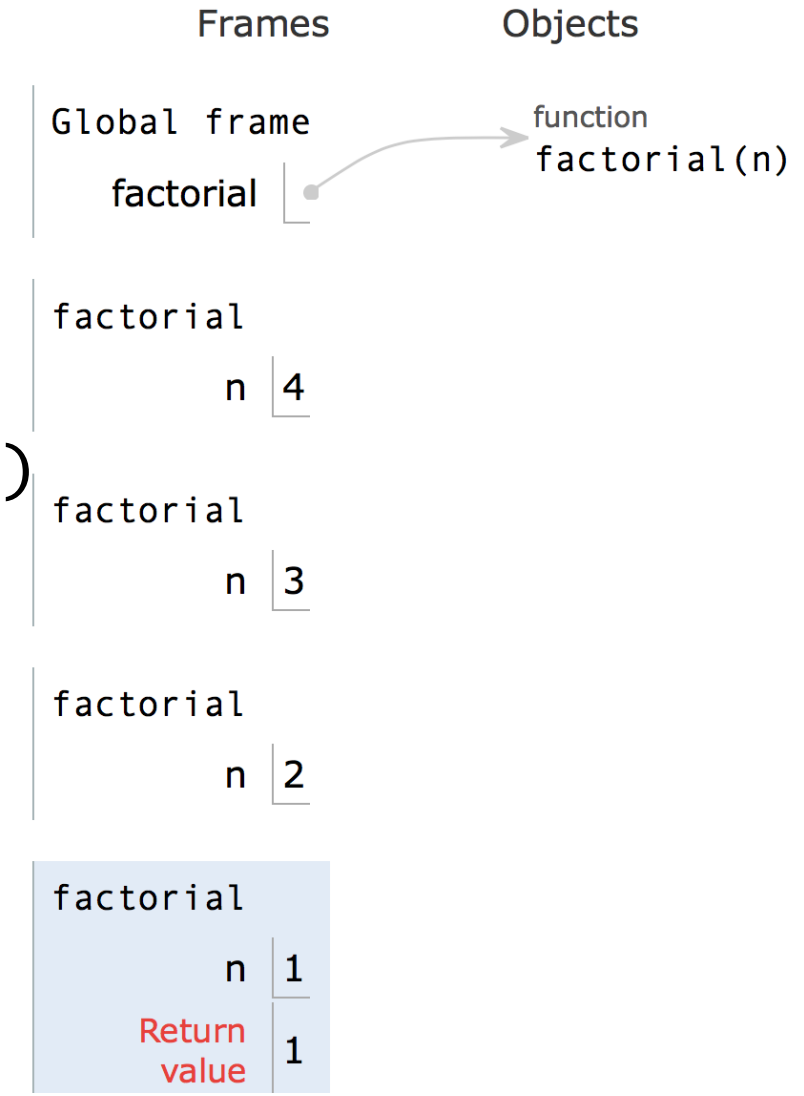
```python
def factorial(n):
    # base case
    if n == 0:
        return 1

    # recursive case
    return n * factorial(n - 1)

print(factorial(4))
```

```
24
```

```
factorial(3)

def factorial(n):  # n = 3
    if n == 0:
        return 1
    return n * factorial(n - 1)

def factorial(n):  # n = 2
    if n == 0:
        return 1
    return n * factorial(n - 1)

def factorial(n):  # n = 1
    if n == 0:
        return 1
    return n * factorial(n - 1)

def factorial(n):  # n = 0
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

# 예시: 팩토리얼 (함수 호출 다이어그램)

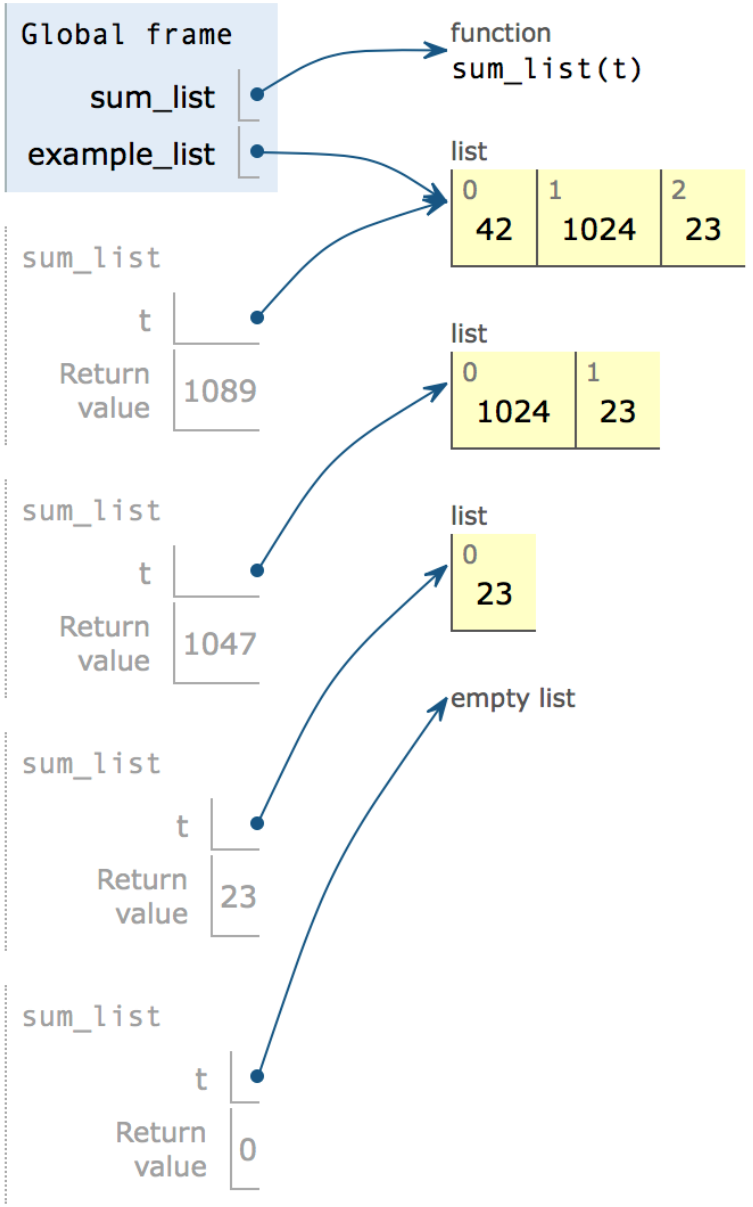
- factorial(4)
  = 4 * factorial(3)
  = 4 * (3 * factorial(2))
  = 4 * (3 * (2 * factorial(1)))
  = 4 * (3 * (2 * (1 * factorial(0))))
  = 4 * (3 * (2 * (1 * 1)))
  = 4 * (3 * (2 * 1))
  = 4 * (3 * 2)
  = 4 * 6
  = 24
- 실행예시



# 예시: 리스트 원소들의 합

- 리스트에 있는 아이템들의 합을 0번방의 아이템과 나머지들의 합으로 표현 (실행예시)

```
def sum_list(t):
    # base_case
    if len(t) == 0:
        return 0

    # recursive case
    return t[0] + sum_list(t[1:])

example_list = [42, 1024, 23]
print(sum_list(example_list))
```



# 예시: 피보나치 수열 (1/2)

- Base case: $F_n = 1$ for $n = 0, 1$
- Recursive case: $F_n = F_{n-1} + F_{n-2}$ for $n = 2, 3, 4, ...$

```
def fibonacci(n):
    # base case
    if n == 0 or n == 1:
        return 1

    # recursive case
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(5))
```

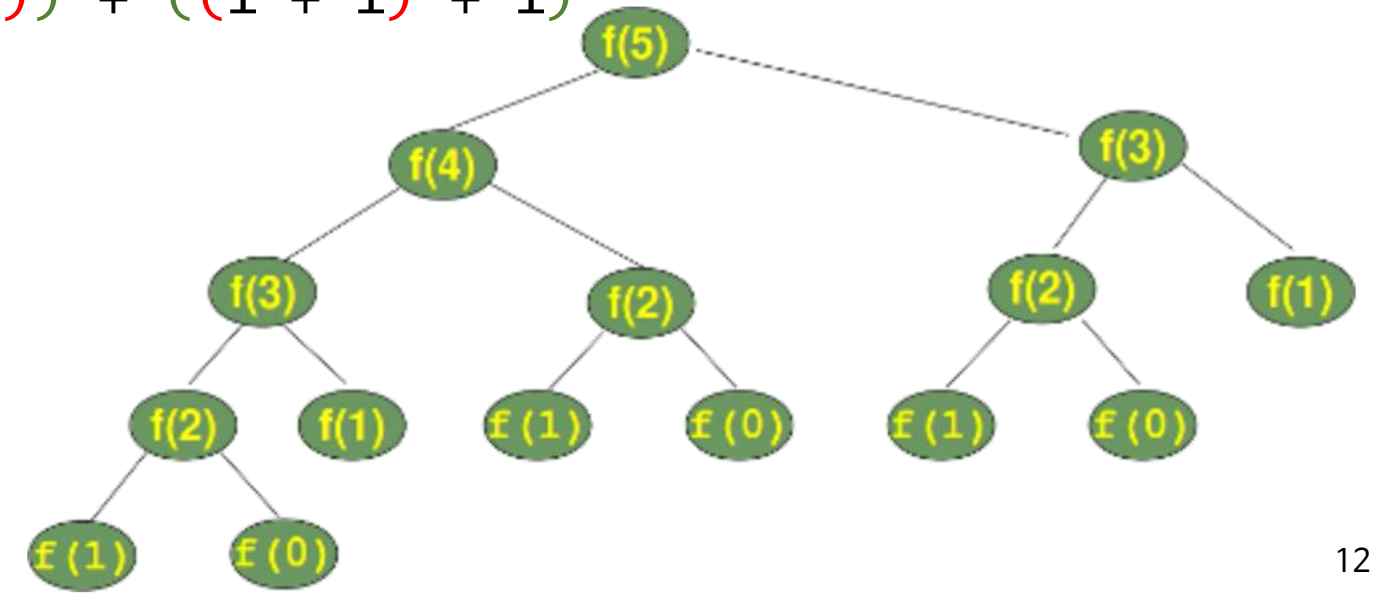
```
8
```

# 예시: 피보나치 수열 (2/2)

- fibonacci()를 f()로 줄여서 나타냈음

```
f(5)
= f(4) + f(3)
= (f(3) + f(2)) + (f(2) + f(1))
= ((f(2) + f(1)) + (f(1) + f(0))) + ((f(1) + f(0)) + f(1))
= (((f(1) + f(0)) + f(1)) + (f(1) + f(0))) + ((f(1) + f(0)) + f(1))
= (((1 + 1) + 1) + (1 + 1)) + ((1 + 1) + 1)
= ((2 + 1) + 2) + (2 + 1)
= (3 + 2) + 3
= 5 + 3
= 8
```

→ 같은 값이 여러 번 계산됨



# 예시: 피보나치 수열 - 동적계획법

- 계산된 값들을 fib라는 사전에 저장하여 중복계산을 피함
  (동적계획법, dynamic programming 혹은 memoization) 실행예시

```
def fibonacci_dyn(n, fib):
    # base case, using dynamic programming
    if n in fib:
        return fib[n]
    # recursive case
    fib[n] = fibonacci_dyn(n - 1, fib) + fibonacci_dyn(n - 2, fib)
    return fib[n]

fib = {0: 1, 1: 1}  # Using
print(fibonacci_dyn(5, fib))
```

```
8
```

# 예시: 피보나치 수열 - Helper 함수 사용

- 중간 결과를 저장하는 것을 신경쓰지 않고 함수를 사용할 수 있도록, 함수를 2개 정의함
  - 중간 결과*를 저장하는 공간을 초기화하고, 재귀호출을 하는 함수를 호출하는 함수
  - 재귀호출을 하는 함수

```
def fibonacci_helper(n, fib):
    # base case
    if n in fib:
        return fib[n]
    # recursive case
    fib[n] = fibonacci_helper(n - 1, fib) + fibonacci_helper(n - 2, fib)
    return fib[n]

def fibonacci(n):
    fib = {0: 1, 1: 1}
    return fibonacci_helper(n, fib)

print(fibonacci(5))
```

```
8
```

* 동적 계획법을 사용하지 않아도, 중간결과를 저장해야 되는 경우도 포함

# 예시: 피보나치 수열 - 디폴트 인자 사용 (1/2)

```
def fibonacci_dyn2(n, fib=None):
    print(f'Calling fibonacci_dyn2() with n={n}, fib={fib}') # track
function call
    if not fib:             # initialize dict fib, when function is invoked
        fib = {0: 1, 1: 1} # for the first time
    # base case
    if n in fib:
        return fib[n]
    # recursive case
    fib[n] = fibonacci_dyn2(n - 1, fib) + fibonacci_dyn2(n - 2, fib)
    return fib[n]

print(fibonacci_dyn2(5))
```

# 예시: 피보나치 수열 - 디폴트 인자 사용 (2/2)

```
Calling fibonacci_dyn2() with n=5, fib=None
Calling fibonacci_dyn2() with n=4, fib={0: 1, 1: 1}
Calling fibonacci_dyn2() with n=3, fib={0: 1, 1: 1}
Calling fibonacci_dyn2() with n=2, fib={0: 1, 1: 1}
Calling fibonacci_dyn2() with n=1, fib={0: 1, 1: 1}
Calling fibonacci_dyn2() with n=0, fib={0: 1, 1: 1}
Calling fibonacci_dyn2() with n=1, fib={0: 1, 1: 1, 2: 2}
Calling fibonacci_dyn2() with n=2, fib={0: 1, 1: 1, 2: 2, 3: 3}
Calling fibonacci_dyn2() with n=3, fib={0: 1, 1: 1, 2: 2, 3: 3, 4: 5}
8
```

# 재귀와 반복 (recursion v.s. iteration)

- 재귀로 구현될 수 있는 모든 것은 반복문으로 표현 가능함
- 일반적인 문제 접근 방법
  - 재귀의 경우는 Top-down (큰 문제부터 나누어서)
  - 반복의 경우는 Bottom-up (작은 문제부터 해결하면서)
- 중간 결과 저장
  - 재귀 호출을 이용할 경우에는 중간결과를 함수가 호출될 때마다 만들어지는 저장공간 혹은 반환값 등으로 저장할 수 있음
  - 반복문을 이용할 경우에는, 별도의 변수에 중간 결과를 저장해야 될 경우가 있음

# 예시: 재귀와 반복문의 비교

**재귀**

- n!을 (n-1)!을 이용하여 계산
- (n-1)!을 (n-2)!을 이용하여 계산
- …

**반복문**

- n!을 1!, 2!, 3!, ... 순으로 계산

```
def factorial(n):
    # base case
    if n == 0:
        return 1

    # recursive case
    return n * factorial(n - 1)
```

```
def factorial_loop(n):
    product = 1
    for i in range(1, n + 1):
        product *= i

    return product
```

# 분할정복법(divide and conquer)을 이용한 프로그래밍

- 문제를 해결하기 위하여, 다음의 각각의 단계로 나누어서 해결
  - 분할(divide): 한 문제를 같은 유형의 다수의 (작은/단순한) 부분 문제로 분할
  - 정복(conquer): 부분 문제를 해결
    - 많은 경우 재귀적으로 해결
    - 부분 문제의 크기가 충분히 작으면 직접적인 방법으로 해결
  - 결합(merge): 부분 문제의 해를 이용하여, 큰 문제의 해를 만들기
- 각각의 문제를 나타낼 수 있는 데이터와 전달방법 설계
  - 문제의 정의: 인자 혹은 자료구조
  - 문제의 해: 반환값 혹은 자료구조

# 읽을 거리

- Think python: 아래 중 '재귀'와 '재귀적 함수의 스택다이어그램' 참조: http://www.flowdas.com/thinkpython/05-conditionals-and-recursion/

# ANY QUESTIONS?