

함수의 활용

목차

- 시작하기 전에
- 재귀 함수
- 재귀 함수의 문제
- 조기 리턴
- 키워드로 정리하는 핵심 포인트
- 확인문제

[핵심 키워드] : 재귀 함수, 메모화, 조기 리턴

[핵심 포인트]

함수를 활용하는 주요 패턴에 대해 살펴본다.

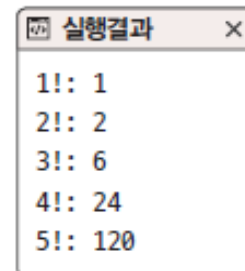
- 팩토리얼 (factorial)

$$n! = n * (n - 1) * (n - 2) * \dots * 1$$

- 반복문으로 팩토리얼 구하기
- 재귀 함수로 팩토리얼 구하기

- 반복문으로 팩토리얼 구하기

```
01  # 함수를 선언합니다.
02  def factorial(n):
03      # 변수를 선언합니다.
04      output = 1
05      # 반복문을 돌려 숫자를 더합니다.
06      for i in range(1, n + 1):
07          output *= i
08      # 리턴합니다.
09      return output
10
11  # 함수를 호출합니다.
12  print("1!:", factorial(1))
13  print("2!:", factorial(2))
14  print("3!:", factorial(3))
15  print("4!:", factorial(4))
16  print("5!:", factorial(5))
```



1!:	1
2!:	2
3!:	6
4!:	24
5!:	120

- 재귀함수로 팩토리얼 구하기

$$n! = n * (n - 1) * (n - 2) * \dots * 1$$

`factorial(n) = n * factorial(n - 1) (n >= 1 일 때)`
`factorial(0) = 1`

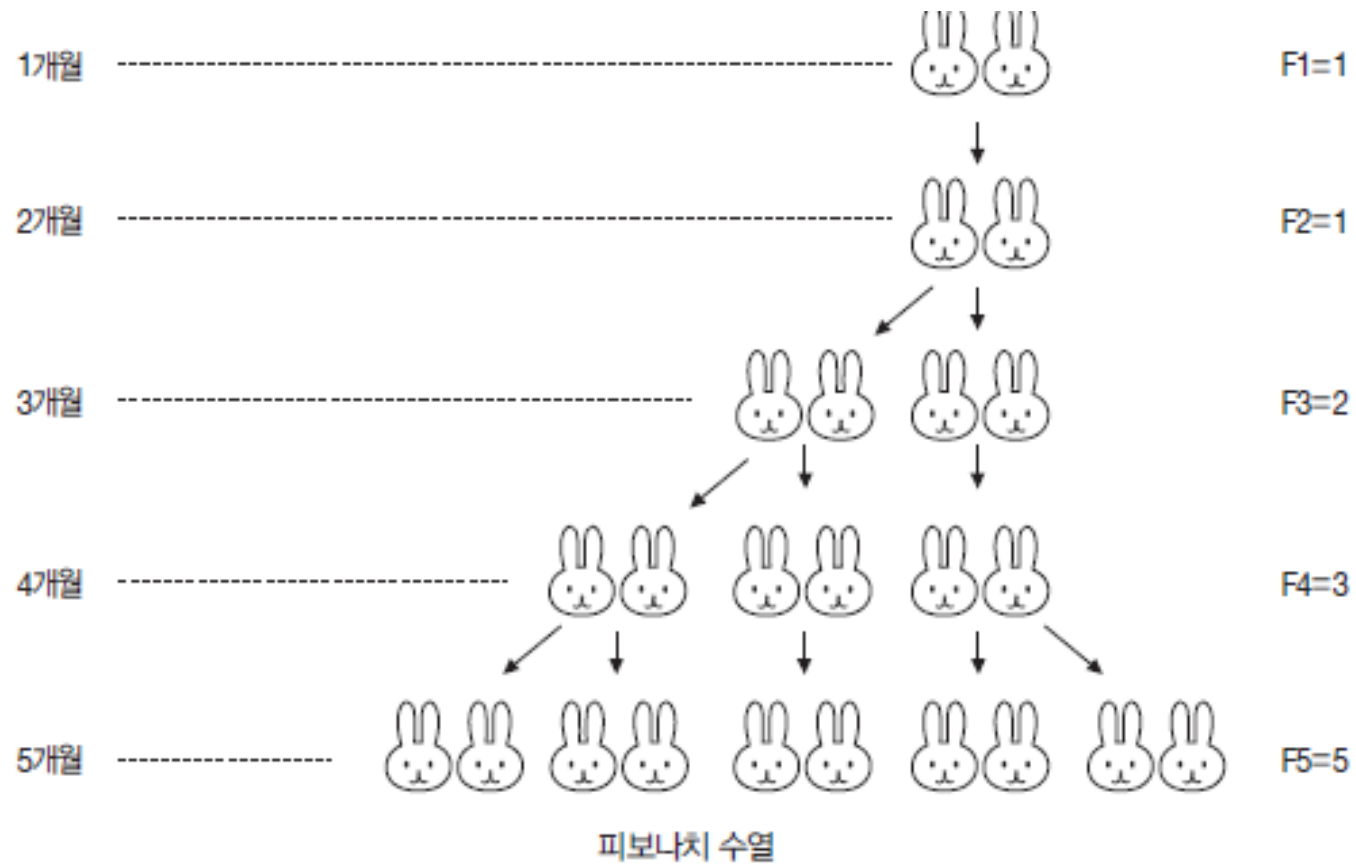
$$\begin{aligned} f(4) &= 4 * f(3) \\ &= 4 * 3 * f(2) \\ &= 4 * 3 * 2 * f(1) * \underline{f(0)} \rightarrow f(0) \text{는 } 1 \text{이므로 곧바로 } 1 \text{로 변경합니다.} \\ &= 4 * 3 * 2 * 1 * 1 \end{aligned}$$

```
01  # 함수를 선언합니다.
02  def factorial(n):
03      # n이 0이라면 1을 리턴
04      if n == 0:
05          return 1
06      # n이 0이 아니면 n * (n-1)!을 리턴
07      else:
08          return n * factorial(n - 1)
09
10  # 함수를 호출합니다.
11  print("1!:", factorial(1))
12  print("2!:", factorial(2))
13  print("3!:", factorial(3))
14  print("4!:", factorial(4))
15  print("5!:", factorial(5))
```

실행결과

1!:	1
2!:	2
3!:	6
4!:	24
5!:	120

- 피보나치 수열




```
01  # 함수를 선언합니다.
02  def fibonacci(n):
03      if n == 1:
04          return 1
05      if n == 2:
06          return 1
07      else:
08          return fibonacci(n - 1) + fibonacci(n - 2)
09
10  # 함수를 호출합니다.
11  print("fibonacci(1):", fibonacci(1))
12  print("fibonacci(2):", fibonacci(2))
13  print("fibonacci(3):", fibonacci(3))
14  print("fibonacci(4):", fibonacci(4))
15  print("fibonacci(5):", fibonacci(5))
```

실행결과

```
fibonacci(1): 1
fibonacci(2): 1
fibonacci(3): 2
fibonacci(4): 3
fibonacci(5): 5
```

- 위와 같이 코드 작성할 경우 처리에 시간이 오래 걸리는 문제 발생

```
01  # 변수를 선언합니다.
02  counter = 0
03
04  # 함수를 선언합니다.
05  def fibonacci(n):
06      # 어떤 피보나치 수를 구하는지 출력합니다.
07      print("fibonacci({})를 구합니다.".format(n))
08      global counter
09      counter += 1
10      # 피보나치 수를 구합니다.
11      if n == 1:
12          return 1
13      if n == 2:
14          return 1
15      else:
16          return fibonacci(n - 1) + fibonacci(n - 2)
17
```

```
18  # 함수를 호출합니다.  
19  fibonacci(10)  
20  print("---")  
21  print("fibonacci(10) 계산에 활용된 덧셈 횟수는 {}번입니다.".format(counter))
```

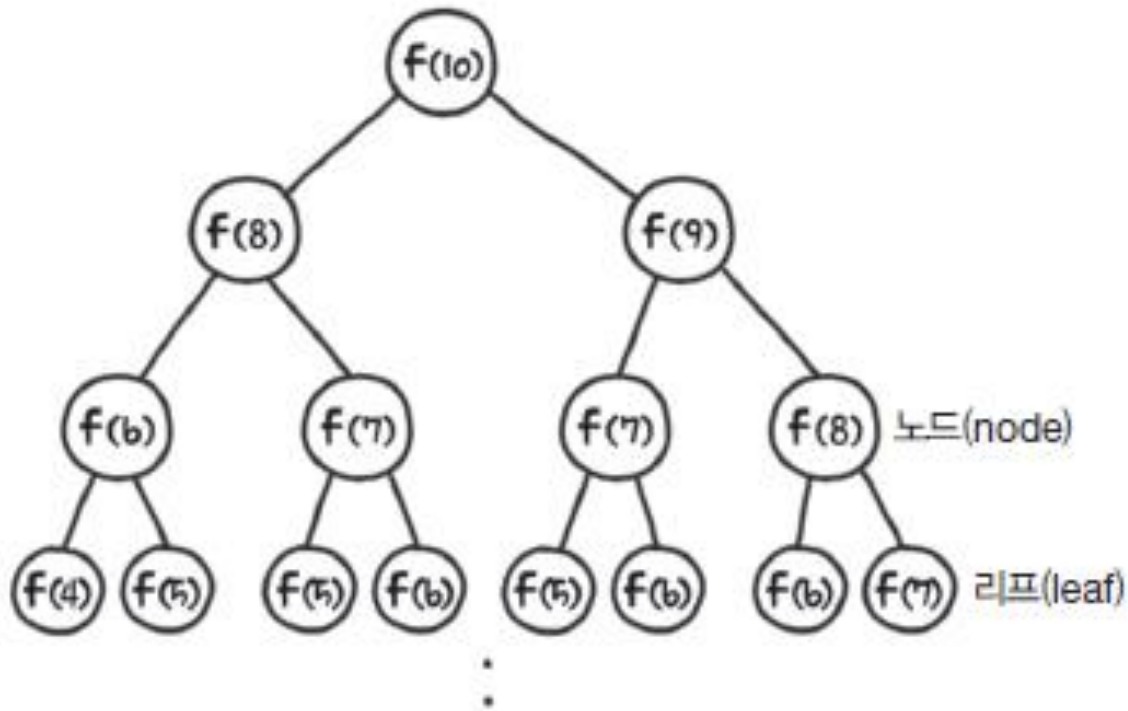
실행결과

```
fibonacci(10)를 구합니다.  
fibonacci(9)를 구합니다.  
...생략...  
fibonacci(1)를 구합니다.  
fibonacci(2)를 구합니다.  
---  
fibonacci(10) 계산에 활용된 덧셈 횟수는 109번입니다.
```

fibonacci(35) 계산에 활용된 덧셈 횟수는 18454929번입니다.

트리 (tree)

- 각 지점 : **노드** (node)
- 노드 중 가장 마지막 단계 지점 : **리프** (leaf)



- `UnboundLocalError`에 대한 처리
 - 슬라이드 #10, 11의 코드에서 global counter 라고 된 부분
 - 해당 부분 지우고 실행하는 경우 `UnboundLocalError` 발생

```
01  # 변수를 선언합니다.  
02  counter = 0  
03  
04  # 함수를 선언합니다.  
05  def fibonacci(n):  
06      counter += 1  
07      # 피보나치 수를 구합니다.  
08      if n == 1:  
09          return 1  
10      if n == 2:  
11          return 1  
12      else:  
13          return fibonacci(n - 1) + fibonacci(n - 2)  
14  
15  # 함수를 호출합니다.  
16  print(fibonacci(10))
```

오류

Traceback (most recent call last):

File "fibonacci_recursion03.py", line 16, in <module>

print(fibonacci(10))

File "fibonacci_recursion03.py", line 6, in fibonacci

counter += 1

UnboundLocalError: local variable 'counter' referenced before assignment

- 파이썬은 함수 내부에서 함수 외부에 있는 변수를 참조할 수 없음
- 아래 global 키워드 구문 사용

global 변수 이름

- 메모화

- 재귀 함수를 사용하면서 코드가 빠르게 실행되려면?
- 같은 값을 한 번만 계산하도록 코드를 수정

```
01  # 메모 변수를 만듭니다.  
02  dictionary = {  
03      1: 1,  
04      2: 2  
05  }  
06  
07  # 함수를 선언합니다.  
08  def fibonacci(n):  
09      if n in dictionary:
```

```
10         # 메모가 되어 있으면 메모된 값을 리턴
11         return dictionary[n]
12     else:
13         # 메모가 되어 있지 않으면 값을 구함
14         output = fibonacci(n - 1) + fibonacci(n - 2)
15         dictionary[n] = output
16         return output
17
18 # 함수를 호출합니다.
19 print("fibonacci(10):", fibonacci(10))
20 print("fibonacci(20):", fibonacci(20))
21 print("fibonacci(30):", fibonacci(30))
22 print("fibonacci(40):", fibonacci(40))
23 print("fibonacci(50):", fibonacci(50))
```

실행결과

```
fibonacci(10): 89
fibonacci(20): 10946
fibonacci(30): 1346269
fibonacci(40): 165580141
fibonacci(50): 20365011074
```


- 메모 (memo)
 - 딕셔너리를 사용해서 한 번 계산한 값을 저장
 - 처리 수행하지 않고 메모된 값 돌려주면서 코드 속도 향상
- 재귀함수와 자주 함께 사용하는 기술

- 조기 리턴 (early return)
 - 흐름 중간에 return 키워드를 사용
 - if else 조건문 만들고 각각의 마지막 부분에서 리턴하게 할 경우

```
# 함수를 선언합니다.  
def fibonacci(n):  
    if n in dictionary:  
        # 메모되어 있으면 메모된 값을 리턴  
        return dictionary[n]  
    else:  
        # 메모되어 있지 않으면 값을 구함  
        output = fibonacci(n - 1) + fibonacci(n - 2)  
        dictionary[n] = output  
    return output
```

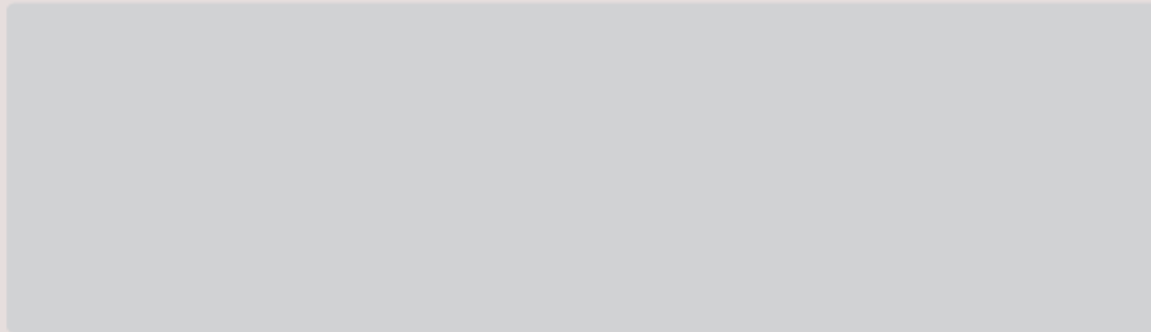
— 조기 리턴 사용

```
# 함수를 선언합니다.  
def fibonacci(n):  
    if n in dictionary:  
        # 메모되어 있으면 메모된 값 리턴  
        return dictionary[n]  
    # 메모되어 있지 않으면 값을 구함  
    output = fibonacci(n - 1) + fibonacci(n - 2)  
    dictionary[n] = output  
    return output
```

- **재귀 함수** : 내부에서 자기 자신을 호출하는 함수
- **메모화** : 한 번 계산한 값을 저장한 후, 계산하는 과정 대신 나중에 이를 다시 활용하는テクニック
- **조기 리턴** : 함수의 흐름 중간에 return 키워드 사용해서 코드 들여쓰기 줄이는 등의 효과 가져오는テクニック

- 다음 빈칸을 재귀함수로 만들어 리스트를 평탄화하는 함수를 만들어보세요.
 - 중첩된 리스트가 있을 때 중첩 모두 제거하고 풀어서 1차원 리스트로 만드는 것을 리스트 평탄화라 합니다.

```
def flatten(data):
```



```
example = [[1, 2, 3], [4, [5, 6]], 7, [8, 9]]  
print("원본:", example)  
print("변환:", flatten(example))
```

실행결과

원본: [[1, 2, 3], [4, [5, 6]], 7, [8, 9]]
변환: [1, 2, 3, 4, 5, 6, 7, 8, 9]

- 이 문제를 풀 때는 리스트의 데이터가 리스트인지 아닌지 구분할 수 있어야 합니다. `type()` 함수를 사용해서 자료형 판별할 때는 다음 코드를 사용합니다.

```
>>> type(10) == int
True
>>> type("10") == str
True
>>> type([]) == list
True
```