일급함수(교재 129)

- 파이썬 일급함수

부성순 bakpak@empas.com

1급 함수(first class function) (1)

- **1급 함수:** 함수가 다른 객체와 동일하게 취급되며 다음 특징이 있음
  - 변수 할당: 함수를 변수에 할당
  - 함수 인자로 전달: 함수를 다른 함수의 인자로 전달
  - 함수에서 반환: 함수가 다른 함수를 반환
  - 데이터 구조에 저장: 함수를 리스트, 딕셔너리 등 데이터 구조에 저장 가능

✓ **함수를 변수에 할당**

```
def great(name):
    return 'Hello, '+name
sayHello=great
sayHello('kim')
```

```
'Hello, kim'
```

1급 함수(first class function) (2)

✓ 함수를 다른 함수의 인자로 전달

```
def square(n):
    return n*n

def cubic(n):
    return n*n*n

def apply_func(L,func):
    L_result=[]
    for i in L:
        L_result.append(func(i))
    return L_result

L=[1,2,3,4,5]
print('L=',L)
print('apply_func(L,square)=',apply_func(L,square))
print('apply_func(L,cubic)=',apply_func(L,cubic))
```

```
L= [1, 2, 3, 4, 5]
apply_func(L,square)= [1, 4, 9, 16, 25]
apply_func(L,cubic)= [1, 8, 27, 64, 125]
```

✓ 함수를 리스트에 저장

```
def square(n):
    return n*n

def cubic(n):
    return n*n*n

func_L=[square, cubic]
L=[1,2,3,4,5]
for f in func_L:
    L_result=[]
    for i in L:
        L_result.append(f(i))
    print(L_result)
```

```
[1, 4, 9, 16, 25]
[1, 8, 27, 64, 125]
```

부성순 bakpak@empas.com

함수장식자(데코레이터)(교재 130~132)

- 함수장식자 (데코레이터)

부성순 bakpak@empas.com

데코레이터 함수(Function Decorator) (1)

➢ **데코레이터 함수(Function Decorator):**

- 기존 함수의 동작을 변경하거나 확장하는데 시용
- 기존함수를 변경하지 않고도 추가적인 기능을 적용할 수 있음
- 함수 수식자는 로깅, 인증 체크, 성능 측정 등 다양한 상황에서 유용하게 사용

기본구조

```
def decorator(func):
    def wrapper():
        # 여기서 어떤 작업을 수행
        return func()
    return wrapper

@decorator
def my_function():
    print("Hello, World!")
```

decorator : 데코레이터 함수, 다른 함수(func)를 인자로 받음

wrappe: 내부함수로, 다른함수(func)의 호출을 감싸는 역할, 이 함수에서 추가적인 작업을 수행할 수 있음

@decorator: @기호와 함께 특정 함수를 수식함

my_function()함수를 호출하면 실제로는 'decorator(my_function)이 호출되어 'wrapper'함수가 실행됨

데코레이터 함수(Function Decorator) (2)

```python
import time

def timer_decorator(func):
    def wrapper():
        start = time.time()
        result = func()
        end = time.time()
        print(f"Function took {end - start} seconds to run.")
        return result
    return wrapper

@timer_decorator
def my_function():
    print("Function is running...")

my_function()
```

```
Function is running...
Function took 0.0 seconds to run.
```

my_function()은 timer_decorator에 의해 수식되어,
함수 실행 시간을 계산하고 출력하는 기능이 추가 됨

부성순 bakpak@empas.com

재귀함수와 제너레이터 (교재132)

- 재귀 함수
- 제너레이터

부성순 bakpak@empas.com

재귀 함수 (1)

➢ 재귀함수

- 자기 자신을 호출하는 함수
- 재귀함수에 종료 조건이 없으면, 함수는 무한히 자기 자신을 호출하여 "최대 재귀 깊이 초과(maximum recursion depth exceeded)" 오류를 발생시킴

```
def hello():
    print('Hello word!')
    hello()

hello()
```

종료 조건이 없어 오류가 남

```
Hello word!
Hello word!
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
Cell In[56], line 5
                    중간 생략
--> 506     return os.getpid() == self._master_pid

RecursionError: maximum recursion depth exceeded while calling a Python object
```

```
def hello(n):
    if n==0:                    ← 종료 조건 만듦
        return
    print('Hello word!')
    n -= 1                      ← n을 1감소
    hello(n)                    ← hello함수 호출

hello(5)
```

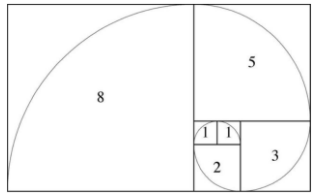
```
Hello word!
Hello word!
Hello word!
Hello word!
Hello word!
```

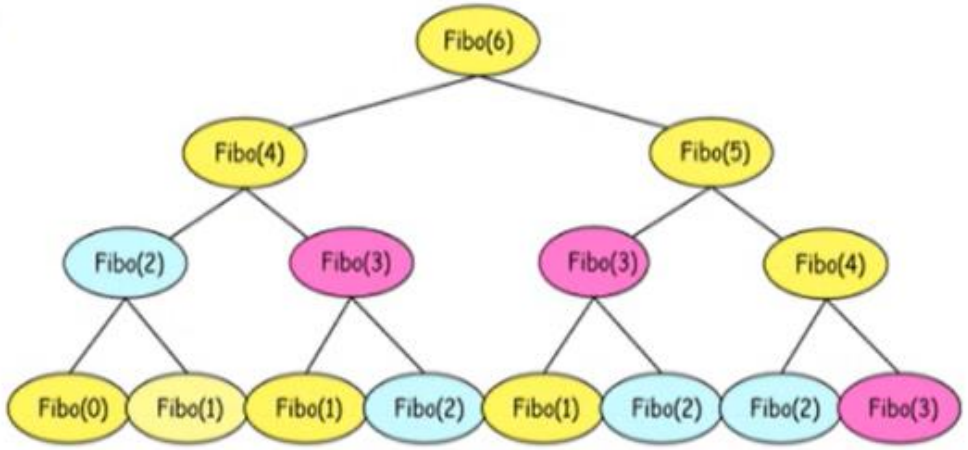
재귀 함수 (2)

➢ 재귀함수 예: 피보나치 수열 계산

- 피보나치 수열: 수열의 시작 두 숫자는 일반적으로 0과 1, 그 다음부터는 앞선 두 숫자의 합으로 구성
  예:0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …
- 재귀함수 피보나치 함수는 n이 커질수록 매우 비효율적이고 같은 계산을 반복적으로 수행
  → 동적 프로그래밍을 사용하면 이미 계산된 값을 저장하여 재사용함으로 속도를 향상 시킬 수 있음

- F(0) = 0, F(1) = 1
- F(n) = F(n-1) + F(n-2)






```
import time
start=time.time()
def f(n):
    if n <=0:
        return 0
    elif n==1:
        return 1
    else:
        return f(n-1) + f(n-2)
print('f(40):', f(40))
print('실행시간:', time.time()-start)
```

피보나치 재귀함수는 n이 40만 되어도
30초이상의 실행시간이 걸림
(일반적으로 재귀함수는 35 넘기 힘듬)

```
f(40): 102334155
실행시간: 30.449120044708252
```

재귀 함수 (3)

➢ 재귀함수와 동적프로그래밍(메모이제이션)사용한 피보나치 수열

동적 프로그래밍(Dynamic Programming, DP):

- 복잡한 문제를 여러 개의 간단한 하위 문제로 나누어 푸는 알고리즘
- 각 하위 문제를 한 번만 계산하고, 저장하여 재사용하는 방식으로 속도 향상

```
import time
start = time.time()

memo={}
def dynFibo(n):
    if n in memo:
        return memo[n]
    elif n<=1:
        memo[n]=n
        return n
    else:
        fibo_n=dynFibo(n-1)+dynFibo(n-2)
        memo[n]=fibo_n
        return fibo_n

print(dynFibo(100))
print('실행시간: ', time.time()-start)
```

```
354224848179261915075
실행시간:  0.0
```

제너레이터 (1)

➢ **제너레이터** (Generator)

- 제너레이터는 데이터를 순차적으로 생성하여 전달
- 제너레이터는 전체 항목을 한 번에 메모리에 적재하지 않고, 반복 가능한 상황에서 그 순간에 필요한 값을 하나씩 생성('yield') 함, 대규모 데이터셋을 다룰 때 메모리를 효율적으로 사용할 수 있음
- 제너레이터 함수는 return 대신 yield를 사용하여 (중간)결과값을 전달

```
def my_range(n):
    i=0
    while i<n:
        yield i
        i+=1
itr=my_range(3)
print(next(itr))
print(next(itr))
print(next(itr))
print(next(itr))
```

yield I → 현재의 i값을 전달하고, 함수를 일시 중지 함. next함를 통해 다음 값이 요청될 때 여기서부터 실행 재개 됨

제터레이터 객체 생성 (0~2까지 숫자 생성 제너레이터)

next(itr) 함수를 호출할 때마다 다음 숫자를 순차적으로 생성 (0,1,2)

next(itr) 함수를 호출할 때 생성할 값이 더 이상 없을 때 StopIteration 예외 발생 함

```
0
1
2
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[84], line 10
```

제너레이터 (2)

- **for와 제너레이터**

  for문은 반복할 때마다 __next__를 호출하므로 yield에서 발생시킨 값을 가져오고
  함수 끝까지 도달하여 StopIteration 예외가 발생하면 반복을 끝냄

```
def my_range(n):
    i=0
    while i<n:
        yield i
        i+=1
for i in my_range(3):
    print(i)
```

```
0
1
2
```

제너레이터 (3)

- **코루틴** (Coroutine)
  - 코루틴은 제너레이터의 일반화된 형태로, 두 개의 함수나 루틴 간에 데이터를 주고받을 수 있는 방법을 제공
  - 실행 중지 및 재개: yield 표현식을 사용하여 실행을 중지하고, 다시 그 지점부터 재개
  - 양방향 통신: 코루틴은 send() 메소드를 통해 외부에서 값을 받고, yield를 통해 값을 반환

- **코루틴의 주요 메소드**
  - send(value): 코루틴에 값을 보냄. 코루틴이 yield 표현식에서 멈춰 있는 경우, send()로 보낸 value가 yield에 의해 반환되고 실행이 재개
  - close(): 코루틴을 종료. 이후 코루틴에 값을 보내려고 하면 StopIteration 예외가 발생
  - throw(type[, value[, traceback]]): 코루틴 내부에 예외를 던짐

  - 제너레이터: next함수를 반복 호출하여 값을 얻어내는 방식
  - 코루틴: next함수를 한 번만 호출한 뒤 send로 값을 주고 받는 방식

부성순 bakpak@empas.com

제너레이터 (4)

➢ **코루틴 사용 예**

```
def simple_coroutine():
    while True:
        x = yield
        print(f'Coroutine received: {x}')

# 코루틴 생성 및 시작
coro = simple_coroutine()
next(coro)  # 코루틴을 시작하기 위해 첫 번째 `yield`까지 실행합니다.

# 코루틴에 데이터 보내기
coro.send(10)
coro.send(20)

# 코루틴 종료
coro.close()
```

```
Coroutine received: 10
Coroutine received: 20
```

코루틴은 시작하기 전에 반드시 next(coro) 또는 coro.send(None)으로 초기화해야 함, 코루틴 yield에서 중지

코루틴에 10을 보냄, 현재 코루틴 x=yield에서 오른쪽 yield에서 중지 됐음으로 왼쪽 x에서부터 재시작. 즉 x에 10 대입되고 다음 명령 계속 수행하다 다시 yield에서 멈춤

제너레이터 (5)

```
def coro_sum():
    try:
        total=0
        while True:
            x=yield
            total +=x
    except RuntimeError as e:
        print(e)
        yield total

co=coro_sum()           #코루틴 객체생성
next(co)                #코루틴을 시작하기 위해 첫 번째 'yield'까지 실행

for i in range(1,11):   #코루틴에 1~10까지 값을 전달
    co.send(i)

print(co.throw(RuntimeError, '예외로 코루틴 끝내기')) #co.Throw(예외이름,에러메시지): 코루틴 안에 예외 발생
```

```
예외로 코루틴 끝내기
55
```