AI 프로그래밍 5

융합학과 권오영
oykwon@koreatech.ac.kr

함수

함수

❖ 함수 정의

def *name of function* (*list of formal parameters*):
    body of function

```
def max(x, y):
    if x>y:
        return x
    else
        return y

z = max(3,4)
    # 3,4 actual parameters (or arguments)
```

❖ 함수의 파라미터들 매칭

- Positional -> 각 해당 위치의 파라미터들로 매칭, 즉 첫 actual parameter는 첫 formal parameter와 매칭
- Keyword arguments -> 위치와 상관없이 formal parameter 이름을 사용해서 actual parameter를 할당

함수파라미터

❖ def printName(firstName, lastName, reverse):

```
    if reverse:
        print (lastName + ', ' + firstName)
    else:
        print (firstName, lastName)
```

❖ 아래 호출이 모두 동일함

```
printName('Olga', 'Puchmajerova', False)
printName('Olga', 'Puchmajerova', reverse = False)
printName('Olga', lastName = 'Puchmajerova', reverse = False)
printName(lastName = 'Puchmajerova', firstName = 'Olga', reverse = False)
```

❖ Keyword argument 뒤에 non-keyword argument가 오는 것은 오류

```
printName('Olga', lastName = 'Puchmajerova', False)
```

default 파라미터

❖ def printName(firstName, lastName, reverse = False):

```
    if reverse:
        print lastName + ‘, ‘ + firstName
    else:
        print firstName, lastName
```

※ python 2.x 은 print 에 ( )가 필요없고, python 3.x에서는 ( )가 필요하다.

❖ 함수파라미터중 reverse의 default 값을 False로 설정

```
printName(‘Olga’, ‘Puchmajerova’)
printName(‘Olga’, ‘Puchmajerova’, True)
printName(‘Olga’, ‘Puchmajerova’, reverse = True)
```

가변 파라미터

❖ 파라미터의 수가 정해지지 않는 경우

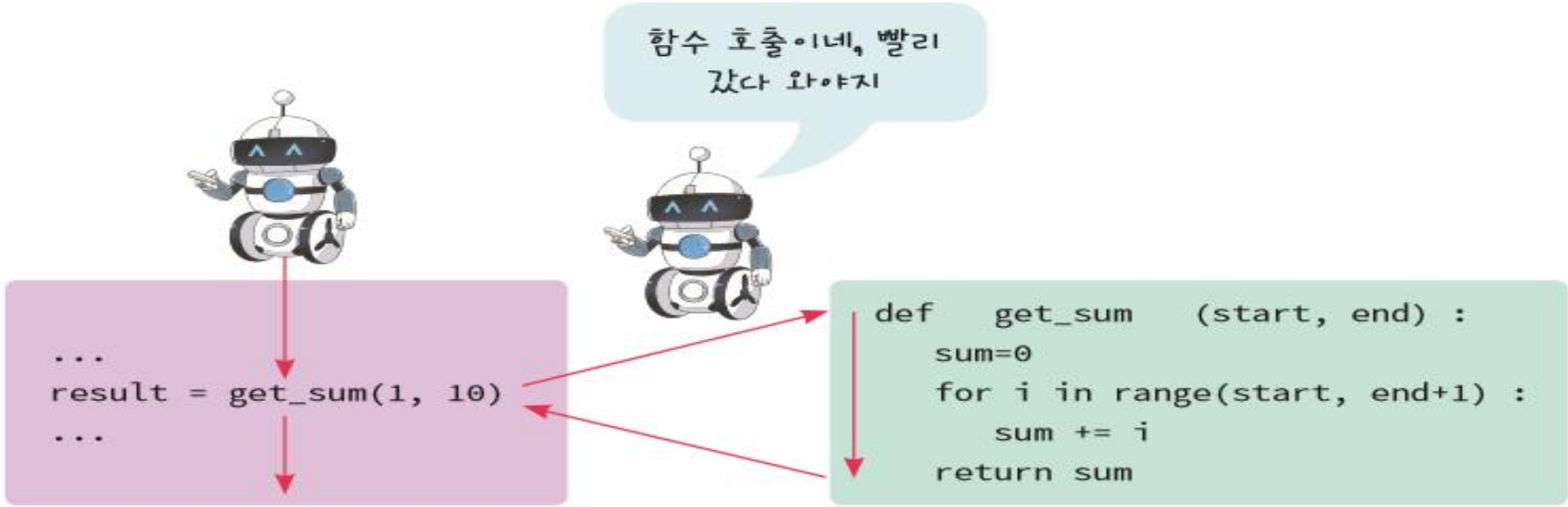
```
def 함수이름 (매개변수, 매개변수, … , *가변매개변수):
    함수몸체
```

```
def print_n_times(n, *values):
        for i in range(n):
            for value in values:
                print(value)
            print()

print_n_times(3, “Hello”, “Fun”, “Python Programming”)
```

함수호출 흐름

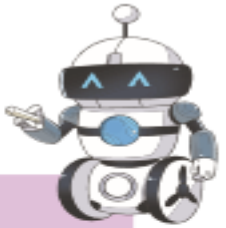
❖ 함수 호출을 통한 재사용
(돌아올 주소를 스택에 저장)



KOREATECH 한국기술교육대학교

함수호출 흐름

❖ 함수 호출을 통한 재사용
(돌아올 주소를 스택에 저장)



```
...
x = get_sum(1, 10)
...
y = get_sum(1, 20)
```

```
def   get_sum   (start, end) :
    sum=0
    for i in range(start, end+1) :
        sum += i
    return sum
```

범위(Scoping)

❖ 함수는 자신의 name space (scope)을 형성

```
def f(x): #name x used as formal parameter
        y = 1
        x = x + y
        print ('x =', x)
        return x

x = 3
y = 2
z = f(x)
print ('z =', z)
print ('x =', x)
print ('y =', y)
```

- 수행결과는

```
x = 4  # 함수 내부 변수
z = 4
x = 3  # top level 변수
y = 2
```

범위(Scoping)

❖ 정적 범위 (lexical scoping) -> 프로그램의 정적인 내포관계에 따라 변수의 영향을 끼치는 범위가 결정되는 방법

```
def f(x):
    def g():
        x = 'abc'
        print ('x =', x)
    def h():
        z = x
        print ('z =', z)
    x = x + 1
    print ('x =', x)
    h()
    g()
    print ('x =', x)
    return g

x = 3
z = f(x)
print ('x =', x)
print ('z =', z)
z()
```

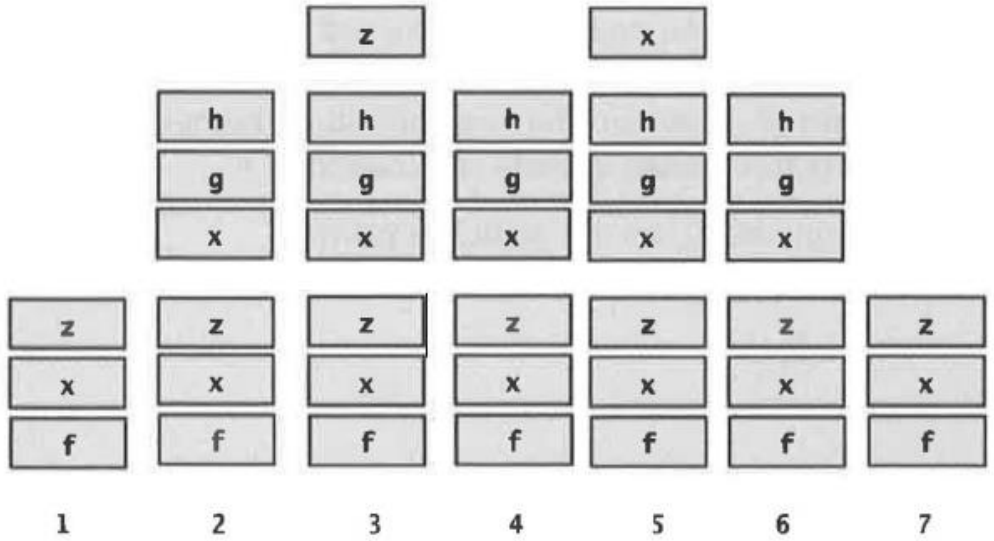


Figure 4.4 Stack frames

실행

x = 4

z = 4

x = abc

x = 4

x = 3

z = <function g at 0x… >

x = abc

전역변수

❖ 전역변수(global variable)

- 가급적 전역변수의 사용은 최소화 하는 것이 바람직하다.
- 파이썬에서는 global 이라는 한정자를 변수 앞에 붙여서 전역변수임을 알려준다.

```
def fib(x):
    """Assumes x an int >= 0
       Returns Fibonacci of x"""
    global numFibCalls
    numFibCalls += 1
    if x == 0 or x == 1:
        return 1
    else:
        return fib(x-1) + fib(x-2)

def testFib(n):
    for i in range(n+1):
        global numFibCalls
        numFibCalls = 0
        print 'fib of', i, '=', fib(i)
        print 'fib called', numFibCalls, 'times.'
```

함수 명세(specification)

❖ 함수 명세 -> 사용자에게 함수에서 가정한 것과 결과 값을 설명해준다.

- Assumptions : 함수를 사용하는 사람에게 제약조건을 명확히 알려주는 역할을 한다.
- Guarantees : 가정에 맞게 함수를 호출하면, 함수가 제공해야하는 조건을 기술한다.

❖ 함수를 선언할 때 위의 두가지를 꼭 기술하자. 최소한 formal parameters들에 대한 설명(Assumptions)과 함수의 반환값(Guarantees)에 대한 설명은 꼭 하는 습관을 갖도록 노력하자.

❖ 함수는 프로그램을 작성하는 elements

- Decomposition : 문제를 작은 모듈들로 분해해서 구조를 만들어 낸다.
- Abstraction : 자세한 처리 과정은 함수의 몸체 안으로 숨기는 역할을 한다.

❖ 추상화(Abstraction)

- 프로그래머는 함수 명세를 보고 코드를 구현
  -> 일을 규모있게 처리할 수 있다.

Help 함수

```
def findRoot(x, power, epsilon):
    """Assumes x and epsilon int or float, power an int,
           epsilon > 0 & power >= 1
       Returns float y such that y**power is within epsilon of x.
           If such a float does not exist, it returns None"""
    if x < 0 and power%2 == 0:
        return None
    low = min(-1.0, x)
    high = max(1.0, x)
    ans = (high + low)/2.0
    while abs(ans**power - x) >= epsilon:
        if ans**power < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2.0
    return ans

def testFindRoot():
    epsilon = 0.0001
    for x in (0.25, -0.25, 2, -2, 8, -8):
        for power in range(1, 4):
            print 'Testing x = ' + str(x) +\
                  ' and power = ' + str(power)
            result = findRoot(x, power, epsilon)
            if result == None:
                print '   No root'
            else:
                print '   ', result**power, '~=', x
```

- """ doc string """ : 다중라인을 포함하고,
  help 함수는 doc string을 보여준다.
  help(findRoot) 하면
  Assumes x and … 라는 doc string을 확인할 수 있다.
  -> 직접 함수를 작성 확인

- 함수 findRoot와 findRoot가 올바로 작동하는지 검증하는 testFindRoot함수 작성
- 검증을 하는 testFindRoot함수를 작성하는 것이 시간을 낭비하는 것 처럼 생각(초보자)되지만 실제로는 큰 이득을 얻는 행동(숙련자)이다.
- 디버깅과정의 단축

Recursion

❖ 재귀함수 (함수 자신을 호출)

- Base case
- Recursive(inductive) case

❖ 수학적 귀납법을 생각

- 초기조건 -> base case
- 가정 (n) 을 충족
- 다음 스텝(n+1)은? -> inductive case

❖ 팩토리얼 계산 (n!)

- 초기조건 1! = 1
- 가정 n! 을 구했다고 가정
- 다음 (n+1)! 은 ?

  (n+1)! = (n+1) * n!

Recursion

❖ 반복(iterative)법과 재귀(recursive)법

```
def factI(n):
    """Assumes that n is an int > 0
       Returns n!"""
    result = 1
    while n > 1:
        result = result * n
        n -= 1
    return result

def factR(n):
    """Assumes that n is an int > 0
       Returns n!"""
    if n == 1:
        return n
    else:
        return n*factR(n - 1)
```