# 함수(function) II

- 함수와 스코프
- 재귀 함수

## 함수와 스코프(scope)

함수는 코드 내부에 공간(scope)를 생성합니다. 함수로 생성된 공간은 `지역 스코프(local scope)` 라고 불리며, 그 외의 공간인 `전역 스코프(global scope)` 와 구분됩니다.

- **전역 스코프( `global scope` )**: 코드 어디에서든 참조할 수 있는 공간
- **지역 스코프( `local scope` )**: 함수가 만든 스코프로 함수 내부에서만 참조할 수 있는 공간

- **전역 변수( `global variable` )**: 전역 스코프에 정의된 변수
- **지역 변수( `local variable` )**: 로컬 스코프에 정의된 변수

In [ ]:
```
# 전역스코프와 지역스코프를 알아봅시다.
a = 10  # 전역 스코프에 정의된 전역 변수

def func(b):
    c = 20  # 지역 스코프에 정의된 지역 변수
    print('지역 스코프 입니다.')
    print(a) # 10
    print(b) #5
    print(c) # 10

func(5)

print('지역 스코프 입니다.')
print(a) # 10
print(b) # 에러
print(c) # 에러
```

In [ ]:
```
# 전역스코프와 지역스코프에 같은 이름의 변수를 만들면 어떻게 될까요?
a = 10

def func(b):
    # 지역 스코프
    a = 100 # L
    c = 20
    print(a) # L에서 정의된 게 있으니까 여기서 쓰임
    print(c)

func(20)
```

In [ ]:
```
# 전역스코프와 지역스코프에 같은 이름의 변수를 만들면 어떻게 될까요?
a = 10

def func(b):
    # 지역 스코프
    c = 20
    print(a) # L에서 정의된 게 없으으니까 전역스코프에 있는거 여기서 쓰임
```

```
    print(c)

func(20)
```

# 이름 검색(resolution) 규칙

파이썬에서 사용되는 이름(식별자)들은 이름공간(namespace)에 저장되어 있습니다.

이것을, `LEGB Rule` 이라고 부르며, 아래와 같은 순서로 이름을 찾아나갑니다.

- `L` ocal scope: 정의된 함수
- `E` nclosed scope: 상위 함수
- `G` lobal scope: 함수 밖의 변수 혹은 import된 모듈
- `B` uilt-in scope: 파이썬안에 내장되어 있는 함수 또는 속성

In [ ]:
```
# 이것을 통해 첫시간에 내장함수의 식별자를 사용할 수 없었던 예제에서 오류가 생기는 이유를 확
# Built-in scope와 Global scope를 알아봅시다.
```

In [ ]:
```
print('hi')  # 출력함수로 빌트인 되어있음
print = 6    # 글로벌 스코프에 프린트 이름의 변수를 정의
```

In [ ]:
```
print('hi')   # LEGB 순서인데 G에 6으로 되어있던 것 따라서 빌트인으로 작용 안돼!
```

In [ ]:
```
# 글로벌 스코프에 정의된 이름 지워버리기
del print
```

1. `print()` 코드가 실행되면

1. 함수에서 실행된 코드가 아니기 때문에 `L` , `E` 를 건너 뛰고,

1. `print` 라는 식별자를 Global scope에서 찾아서 `print = ssafy` 를 가져오고,

1. 이는 함수가 아니라 변수이기 때문에 `not callable` 하다라는 오류를 내뱉게 됩니다.

1. 우리가 원하는 `print()` 은 Built-in scope에 있기 때문입니다.

In [ ]:
```
# Global scope와 Local scope를 알아봅시다.
```

In [ ]:
```
# LEGB Rule을 자세히 알아봅시다.
a = 10  # G
b = 20  # G
def enclosed(): # 함수안에 함수를 정의할 수 있구나로만 생각!
    a = 30  # local 함수 입장에서는 Enclosed , Enclosed 함수 입장에서는 local
    def local():
        c = 40  # local
        print(a, b, c)
    local()
    # print아래에 있어서 안나왕 얘는!
```

```
    a = 50  # local 함수 입장에서는 Enclosed , Enclosed 함수 입장에서는 local
enclosed()
```

In [ ]:
```
# 전역 변수를 바꿀 수 있을까요?
```

In [ ]:
```
global_num = 3
def local_scope():
    global_num = 5


local_scope()
print(global_num)
```

In [ ]:
```
global_num = 3
def local_scope():
    # 아 지금부터 얘기하는거 5반 지민이 아니고 방탄 지민이..
    global global_num
    global_num = 5

local_scope()
print(global_num)
```

In [ ]:
```
# 굳이 전역에 있는 변수를 바꾸고 싶다면, 아래와 같이 선언할 수 있습니다.
```

In [ ]:
```
global_num = 3
def local_scope():
    # Local scope!
    # 아 지금부터 이야기하는거 서울 5반 지민이 말고, 방탄 지민이..
    global global_num
    global_num = 5

local_scope()
print(global_num)
```

## 변수의 수명주기(lifecycle)

변수의 이름은 각자의 수명주기(`lifecycle`) 가 있습니다.

- **빌트인 스코프 (`built-in scope`)** : 파이썬이 실행된 이후부터 영원히 유지
- **전역 스코프 (`global scope`)** : 모듈이 호출된 시점 이후 혹은 이름 선언된 이후부터 인터프리터가 끝날때 까지 유지
- **지역(함수) 스코프 (`local scope`)** : 함수가 호출될 때 생성되고, 함수가 가 종료될 때까지 유지 (함수 내에서 처리되지 않는 예외를 일으킬 때 삭제됨) return을 하는 순간 그 것들은 사라짐

# 재귀 함수(recursive function)

재귀 함수는 함수 내부에서 자기 자신을 호출 하는 함수를 뜻합니다.

알고리즘을 설계 및 구현에서 유용하게 활용됩니다.

# 팩토리얼 계산

> 팩토리얼은 1부터 n 까지 양의 정수를 차례대로 곱한 값이며 ! 기호로 표기합니다. 예를 들어 3!은 3 *2* 1이며 결과는 6 입니다.
>
> 팩토리얼(factorial) 을 계산하는 함수 fact(n) 를 작성하세요.
>
> n은 1보다 큰 정수라고 가정하고, 팩토리얼을 계산한 값을 반환합니다.

$$n! = \prod_{k=1}^{n} k$$

$$n! = 1 * 2 * 3 * \ldots * (n-1) * n$$

---

```
예시 출력)
120
```

## 반복문을 이용한 팩토리얼 계산

In [ ]:
```
# 아래에 코드를 작성해주세요.
```

In [ ]:
```
def fact(n):
    # while 종료 조건!
    result = 1
    while n > 1:
        result *= n
        n -= 1
    return result
```

In [ ]:
```
# 해당 코드를 통해 올바른 결과가 나오는지 확인하세요.
fact(5)
```

## 재귀를 이용한 팩토리얼 계산

```
1! = 1
2! = 1 * 2 = 1! * 2
3! = 1 * 2 * 3 = 2! * 3
```

In [ ]:
```
# 아래에 factorial() 를 작성하세요.
```

In [ ]:
```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

In [ ]:
```
# 해당 코드를 통해 올바른 결과가 나오는지 확인하세요.
factorial(3)
```

# 반복문과 재귀함수

```
factorial(3)
3 * factorail(2)
3 * 2 * factorial(1)
3 * 2 * 1
3 * 2
6
```

- 두 코드 모두 원리는 같다!

1. 반복문 코드
   - n이 1보다 큰 경우 반복문을 돌며, n은 1씩 감소한다.
   - 마지막에 n이 1이면 더 이상 반복문을 돌지 않는다.

1. 재귀 함수 코드
   - 재귀 함수를 호출하며, n은 1씩 감소한다.
   - 마지막에 n이 1이면 더 이상 추가 함수를 호출하지 않는다.

- 재귀함수는 기본적으로 같은 문제이지만 점점 범위가 줄어드는 문제를 풀게 된다.
- 재귀함수를 작성시에는 반드시, `base case` 가 존재 하여야 한다.
- `base case` 는 점점 범위가 줄어들어 반복되지 않는 최종적으로 도달하는 곳이다.
- 재귀를 이용한 팩토리얼 계산에서의 base case는 **n이 1일때, 함수가 아닌 정수 반환하는 것**이다.

- 자기 자신을 호출하는 재귀함수는 알고리즘 구현시 많이 사용된다.
- 코드가 더 직관적이고 이해하기 쉬운 경우가 있다.
- 팩토리얼 재귀함수를 Python Tutor에서 확인해보면, 함수가 호출될 때마다 메모리 공간에 쌓이는 것을 볼 수 있다.
- 이 경우, 메모리 스택이 넘치거나(Stack overflow) 프로그램 실행 속도가 늘어지는 단점이 생긴다.
- 파이썬에서는 이를 방지하기 위해 1,000번이 넘어가게 되면 더이상 함수를 호출하지 않고, 종료된다. (최대 재귀 깊이)

## 최대 재귀 깊이

```
def ssafy():
    print('Hello, ssafy!')
    ssafy()

ssafy()
```

---

`ssafy()` 를 호출하면 아래와 같이 문자열이 계속 출력되다가 RecursionError가 발생합니다.

파이썬에서는 최대 재귀 깊이(maximum recursion depth)가 1,000으로 정해져 있기 때문입니다.

---

```
Hello, world!
Hello, world!
...
```

```
Hello, world!
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)

...

<ipython-input-11-2bbb40950c86> in hello()
      1 def hello():
      2     print('Hello, world!')
----> 3     hello()
      4
      5 hello()

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

In [ ]:
```
# 직접 오류를 확인하세요.
```

In [ ]:
```
def ssafy():
    print('Hello, ssafy!', end=" ")
    ssafy()

ssafy()
```

# 피보나치 수열

첫째 및 둘째 항이 1이며 그 뒤의 모든 항은 바로 앞 두 항의 합인 수열입니다.

(0), 1, 1, 2, 3, 5, 8

> 피보나치 수열은 다음과 같은 점화식이 있습니다.
>
> 피보나치 값을 리턴하는 두가지 방식의 코드를 모두 작성해주세요.
>
> 베이스케이스
>
> $$F_0 = F_1 = 1$$
>
> 재귀
>
> $$F_n = F_{n-1} + F_{n-2} \qquad (n \in \{2, 3, 4, \dots\})$$

1) `fib(n)` : 재귀함수

2) `fib_loop(n)` : 반복문 활용한 함수

---

예시 입력)
`fib(10)`

예시 호출)
89

In [ ]:
```
# 재귀를 이용한 코드 fib() 를 작성하세요.
```

In [ ]:
```
#
# n 이 0이나 1일 때는 값도 0, 1이기 때문에 그대로 반환하면 되고,
# 2 이상일 때만 재귀 함수 두개로 분기해 값을 반환합니다.
def fib(n):
    # base case!
    if n < 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

In [ ]:
```
# 해당 코드를 통해 올바른 결과가 나오는지 확인하세요.
fib(10)
```

fib(5)는 몇 번의 함수가 실행이 될까? 총 15번! 그려보고 해보기

In [ ]:
```
# 반복문을 이용한 코드 fib_loop() 를 작성하세요.
```

In [ ]:
```
def fib_loop(n):
    a, b = 1, 1
    for i in range(n-1):
        a, b = b, a+b
    return b
```

In [ ]:
```
# 해당 코드를 통해 올바른 결과가 나오는지 확인하세요.
fib_loop(10)
```

In [ ]:
```
def fib_while(n):
    a, b = 0,1
    while a < n:
        print(a, end = ' ')
        a, b = b, a+b
    return b
fib_while(22)
```

In [ ]:
```

```

# 반복문과 재귀 함수의 차이

- 알고리즘 자체가 재귀적인 표현이 자연스러운 경우 재귀함수를 사용한다.
- 재귀 호출은 변수 사용 을 줄여줄 수 있다.

In [ ]:
```
# 큰 숫자를 재귀로 짜여진 fib() 함수의 인자로 넘겨보세요.
```

In [ ]:
```
import time

t0 = time.time()
fib(31)
t1 = time.time()

total = t1 - t0
print(total)
```

In [ ]:
```
# 100배 되는 숫자를 반복문으로 짜여진 fib_loop() 인자로 넘겨보세요.
```

In [ ]:
```
import time

t0 = time.time()
fib_loop(30000)
t1 = time.time()

total = t1 - t0
print(total)
```