파이썬으로 배우는 데이터 구조

한동대학교
전산전자공학부
김영섭 교수

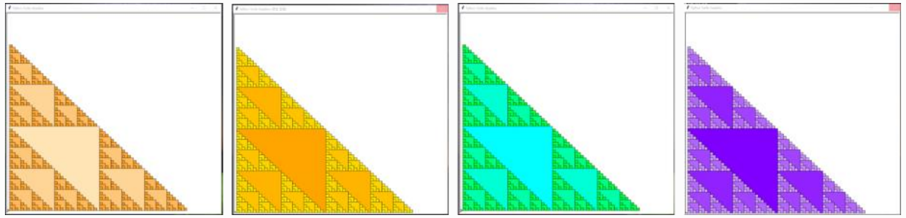
학습 목표

재귀(Recursion)의 개념을 이해하고

간단한 예제들을 통해 깊이 학습한다

Data Structures in Python
Chapter 4

- **Recursion Concepts**
- Recursion Stack and Memoization
- Recursive Algorithms
- Recursive Graphics
- Exercise - Stacking boxes

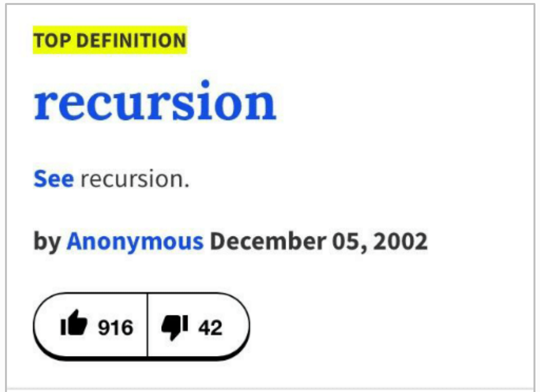


Agenda

- Recursion Definition
  - Definitions and Programming
  - Why recursion?
  - Concept Example
  - More Examples

Recursion Definition

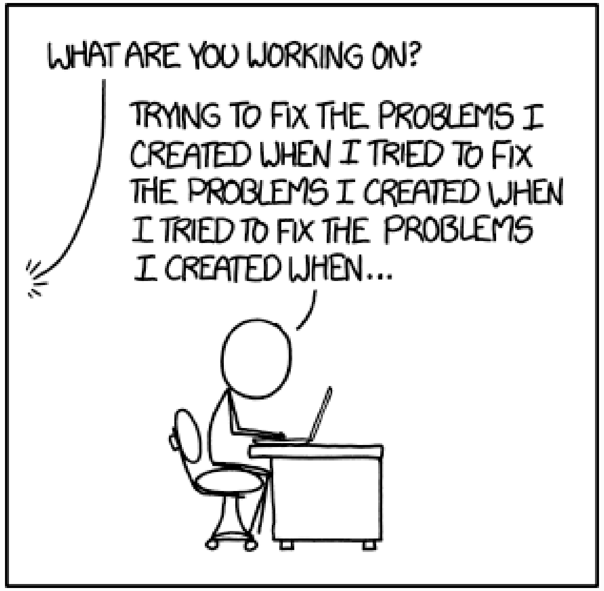
- See Recursion



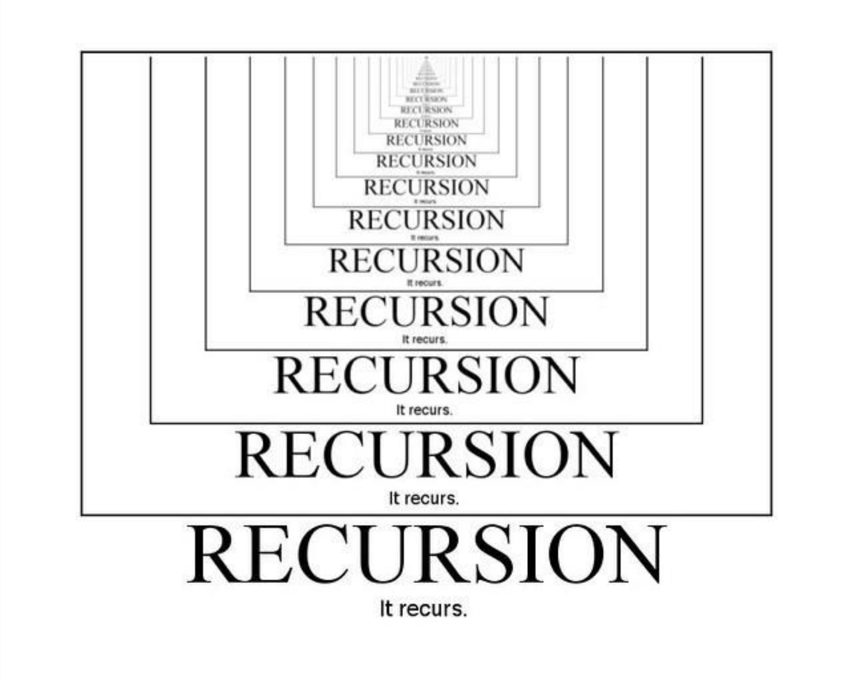
Very descriptive definition

Recursion Definition

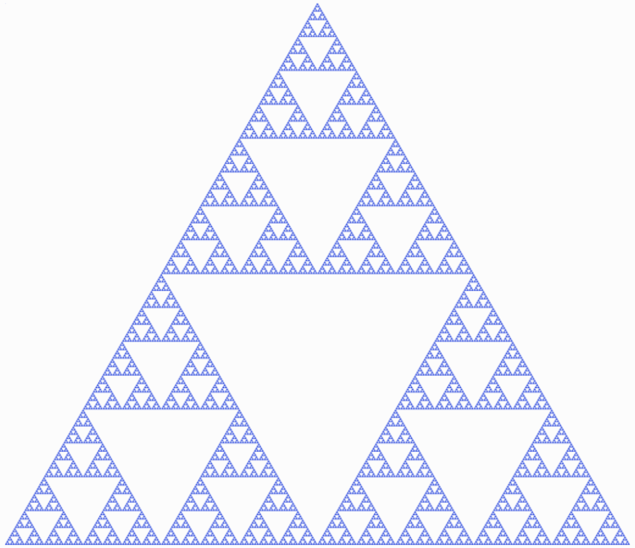
- See Recursion
- Recursion is when a function calls itself
- Recursion simplifies program structure at a cost of function calls



https://www.xkcd.com/1739/



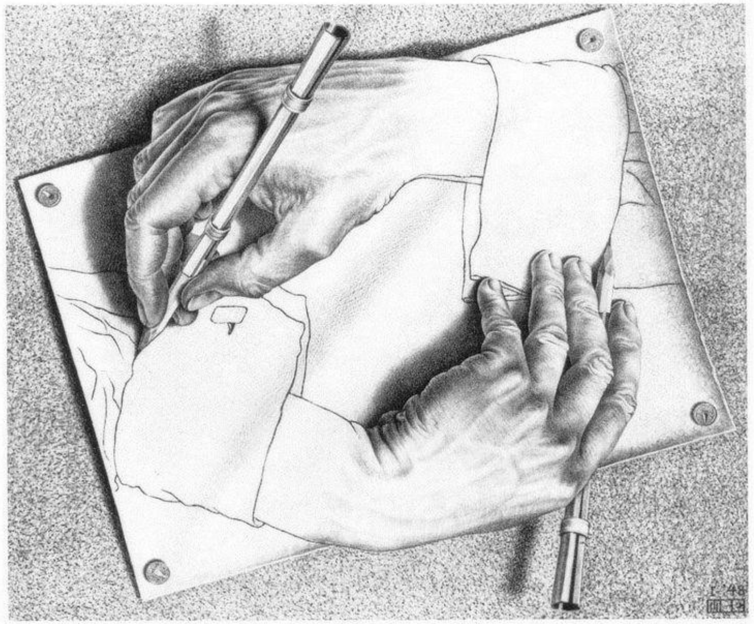
Recursion — Image from AlgoDaily



The [Sierpinski triangle](#)
a confined recursion of triangles that form a fractal

Recursion Definition

- See Recursion
- Recursion is when a function calls itself
- Recursion simplifies program structure at a cost of function calls
- Recursion vs. Leap of faith



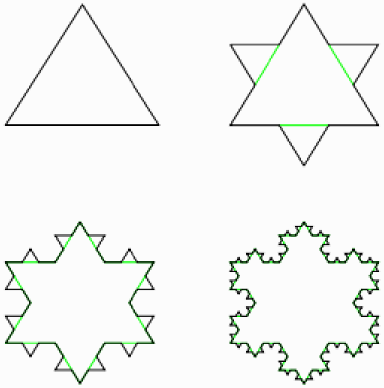
recursion **is**
when a function *calls itself*

Why recursion?

- A new "cultural experience"
  - A different way of thinking of problems or creative thinking
- It can solve some kinds of problems better than iteration.
- It leads to elegant, simplistic, and short code (when used well).
- Believe it or not, there are some programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)
- This skill is a key component of the rest of our course.

Recursion

- Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration).
- Recursive algorithm is expressed in terms of
  1. **base case(s)** for which the solution can be stated non-recursively,
  2. **recursive case(s)** for which the solution can be expressed in terms of **a smaller version of itself.**



Four stages in the construction of a **Koch snowflake**. The stages are obtained via a recursive definition.

Concept Example

- Pick one of students in the row and ask:
  How many students total are next you in your "row"?
  - You have poor vision, so you can see only the people next to you.
    So, you can't just look the sides and count.
  - But you are allowed to ask questions of two persons next to you.
  - How can we solve this problem, recursively?

Concept Example: pass the buck

- Number of people on the both sides of me:
  - If there is someone to the left side of me,
    ask him/her how many people are to the left size of him/her.
  - Do the same to the right side of me.
  - When they respond with a value L from the left and R from the right,
    then I will answer L + R + 1.
- If there is nobody both side of me, I will answer 1.

Recursion and cases

- Every recursive algorithm involves at least 2 cases:
  - **base case:** A simple occurrence that can be answered directly.
  - **recursive case:** A more complex occurrence of the problem that cannot be directly answered but can instead be described in terms of smaller occurrences of the same problem.
- Some recursive algorithms have more than one base or recursive case, but all have at least one of each.
- A crucial part of recursive programming is identifying these cases.

Example 1: Factorial

- Recurrence relation: A mathematical formula that generates the terms in a sequence from previous terms.
  - factorial(n) = n * [(n-1) * (n-2) * ⋯ * 1]
  - factorial(n) = n * factorial(n-1)

- Recursive definition of factorial(n):
  - $factorial(n) = \begin{cases} 1, & if\ n = 0 \\ n * factorial(n-1), & if\ n > 0 \end{cases}$

- Examples:
  - 4! = 4 * 3 * 2 * 1 = 24
  - 7! = 7 * 6 * 5 * 4 * 3 * 2 * 1 = 5040

Example 1: Factorial

- Recursive definition of factorial(n)
  - $factorial(n) = \begin{cases} 1, & if\ n = 0 \\ n * factorial(n-1), & if\ n > 0 \end{cases}$

factorial(n)

```
function factorial
input: integer n such that n >= 0
output: [n × (n-1) × (n-2) × … × 1]
    1. if n is 0, return 1
    2. otherwise, return [ n × factorial(n-1) ]
end factorial
```

factorial(n = 4)

$$
\begin{aligned}
f_4 &= 4 * f_3 \\
&= 4 * (3 * f_2) \\
&= 4 * (3 * (2 * f_1)) \\
&= 4 * (3 * (2 * (1 * f_0))) \\
&= 4 * (3 * (2 * (1 * 1))) \\
&= 4 * (3 * (2 * 1)) \\
&= 4 * (3 * 2) \\
&= 4 * 6 \\
&= 24
\end{aligned}
$$

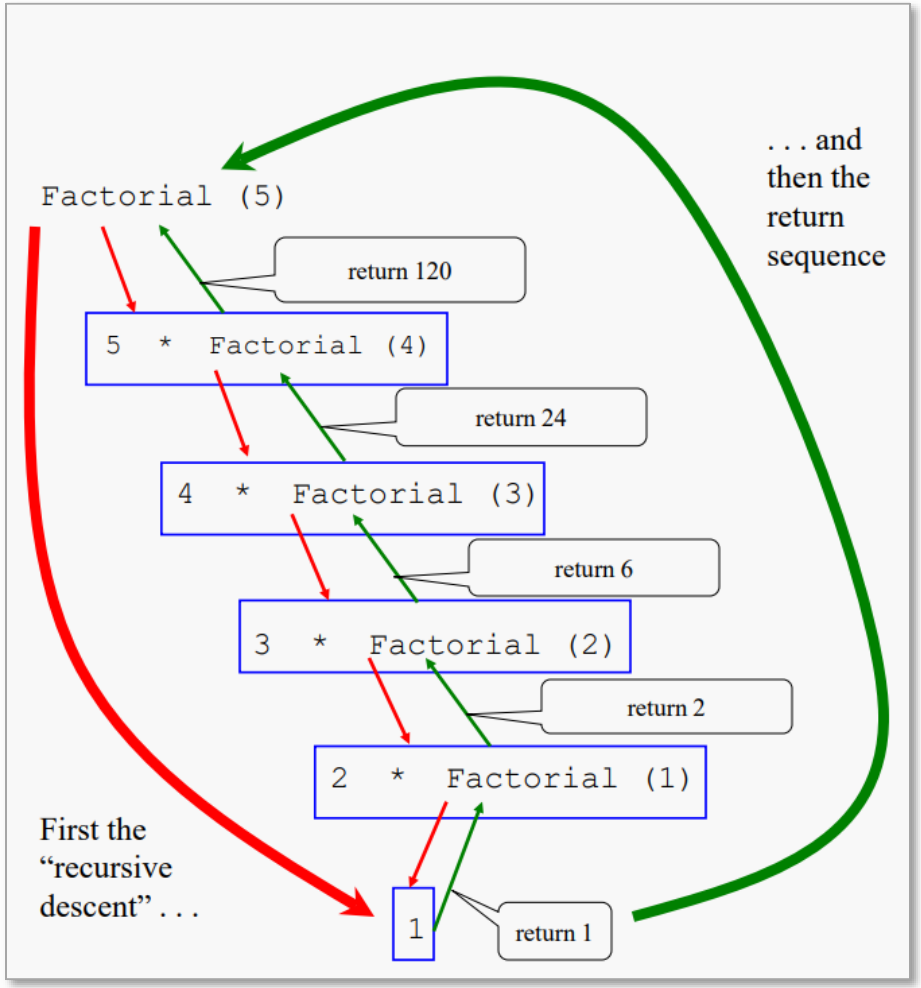
Exercise: With four students, compute 4! using recursion.

Example 1: Factorial

- Recursive definition of factorial(n)
  - $factorial(n) = \begin{cases} 1, & if\ n = 0 \\ n * factorial(n-1), & if\ n > 0 \end{cases}$

factorial(n)

function factorial
input: integer *n* such that *n* >= 0
output: [*n* × (*n*-1) × (*n*-2) × … × 1]
    1. if *n* is 0, **return** 1
    2. otherwise, **return** [*n* × factorial(*n*-1)]
end factorial



Exercise: With four students, compute 4! using recursion.

Example 2: print_stars()

- Consider the following function to print a line of * characters:

```
def print_stars(n):
    """prints a line containing the given number of stars.
       precondition: n >= 0 """
    for i in range(n):
        print('*', end='')
    print()
```

- Write a recursive version of this method (that calls itself).
    - Solve the problem **without using any loops**.
    - Hint: Your solution should print just one star at a time.

Example 2: print_stars()

- What are the cases to consider?
  - What is a very easy number of stars to print without a loop?

```
def print_stars(n):
    if n == 1:
        print('*')
    else:
        ...
    print()
```

Example 2: print_stars()

- Handling additional cases, with no loops (**in a wrong way**):

```
def print_stars(n):
    if n == 1:
        print('*')
    elif n == 2:
        print('**')
    elif n == 3:
        print('***')
    ...
    else:
        ...
    print()
```

Example 2: print_stars()

- Taking advantage of the repeated pattern (**somewhat better**):

```
def print_stars(n):
    if n == 1:
        print('*')
    elif n == 2:
        print_stars(2)
    elif n == 3:
        print_stars(3)
    ...
    else:
        ...
    print()
```

Example 2: Using recursion properly

- Condensing the recursive cases into a single case:

```
def print_stars(n):
    if n == 1:                  # base case:
        print('*')              # print one star
    else:                       # recursive case:
        print('*', end='')      # print one and more stars
        print_stars(n - 1)
```

Example 2: Using recursion properly

- The real, even simpler base case is **an n of 0**, **not 1**:

```
def print_stars(n):
    if n == 0:                    # base case:
        print()                   # end the output
    else:                         # recursive case:
        print('*', end='')        # print one and more stars
        print_stars(n - 1)
```

Bad Recursion Example 1

- Problem:
  - Compute the sum of all integers from 1 to n

```
def bad_sum(n):
    return n + bad_sum(n-1)
```

No base case!!!

Bad Recursion Example 2

- Problem:
  - If n is odd, compute the sum of all odd integers from 1 to n; and if it is even compute sum of all even integers.

```
def bad_sum(n):
    if n == 0:
        return 0
    return n + bad_sum(n-2)
```

Base case cannot be reached!!!

Example 3

- Predict the output of the following code.

```
def foo(n):
    if n < 1: return
    print(n, end = ' ')
    foo(n - 1)
    print(n, end = ' ')
    return

if __name__ == '__main__':
    n = 3
    foo(n)
```

- When main() calls foo(3), main() and **n=3** are pushed to the system stack. It will finish when foo(3) returned.

Example 3

- Predict the output of the following code.

```
def foo(n):
    if n < 1: return
    print(n, end = ' ')
    foo(n - 1)
    print(n, end = ' ')
    return

if __name__ == '__main__':
    n = 3
    foo(n)
```

```
def foo(n=3)
  if n < 1: return
  print(n, end=' ')
  foo(2)
  print(n, end=' ')
  return
```

foo(3) calls foo(2)

- When main() calls foo(3), main() and **n=3** are pushed to the system stack. It will finish when retured from foo(3).
- **foo(3) prints '3'** and calls foo(2). Then, **foo(2)** and **n=2** are pushed to the system stack.

Example 3

- Predict the output of the following code.

```
def foo(n):
    if n < 1: return
    print(n, end = ' ')
    foo(n - 1)
    print(n, end = ' ')
    return

if __name__ == '__main__':
    n = 3
    foo(n)
```

```
def foo(n=3)
  if n < 1: return
  print(n, end=' ')
  foo(2)
  print(n, end=' ')
  return
```

foo(3) calls foo(2)

```
def foo(n=2)
  if n < 1: return
  print(n, end=' ')
  foo(1)
  print(n, end=' ')
  return
```

foo(2) calls foo(1)

```
def foo(n=1)
  if n < 1: return
  print(n, end=' ')
  foo(0)
  print(n, end=' ')
  return
```

foo(1) calls foo(0)

```
def foo(n=0)
  if n < 1: return
```

- When main() calls foo(3), main() and **n=3** are pushed to the system stack. It will finish when retured from foo(3).
- **foo(3) prints '3'** and calls foo(2). Then, **foo(2)** and **n=2** are pushed to the system stack.
- Similarly, **foo(2) prints '2'** and calls foo(1). Then **foo(1) prints '1'** and calls foo(0).

Example 3

- Predict the output of the following code.

```
def foo(n):
    if n < 1: return
    print(n, end = ' ')
    foo(n - 1)
    print(n, end = ' ')
    return

if __name__ == '__main__':
    n = 3
    foo(n)
```

```
def foo(n=3)
  if n < 1: return
  print(n, end=' ')
  foo(2)
  print(n, end=' ')
  return
```

foo(3) calls foo(2)

```
def foo(n=2)
  if n < 1: return
  print(n, end=' ')
  foo(1)
  print(n, end=' ')
  return
```

foo(2) calls foo(1)

```
def foo(n=1)
  if n < 1: return
  print(n, end=' ')
  foo(0)
  print(n, end=' ')
  return
```

foo(1) calls foo(0)

```
def foo(n=0)
  if n < 1: return
```

returns to foo(n=1)

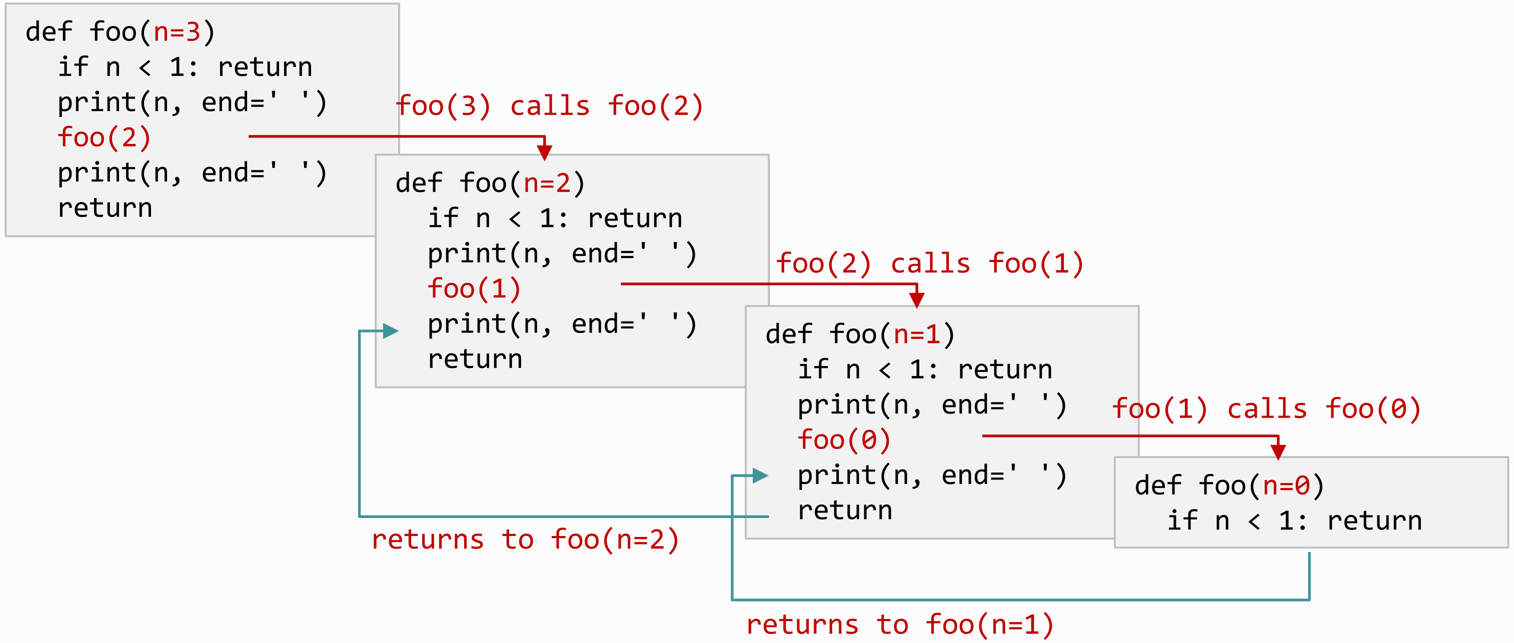
- When main() calls foo(3), main() and **n=3** are pushed to the system stack. It will finish when retured from foo(3).
- **foo(3) prints '3'** and calls foo(2). Then, **foo(2)** and **n=2** are pushed to the system stack.
- Similarly, **foo(2) prints '2'** and calls foo(1). Then **foo(1) prints '1'** and calls foo(0).
- foo(0) goes to if and returns to foo(1). This is the first return ever. Now **foo(1)** popped from the stack **prints '1'**.

Example 3

- Predict the output of the following code.

```
def foo(n):
    if n < 1: return
    print(n, end = ' ')
    foo(n - 1)
    print(n, end = ' ')
    return

if __name__ == '__main__':
    n = 3
    foo(n)
```



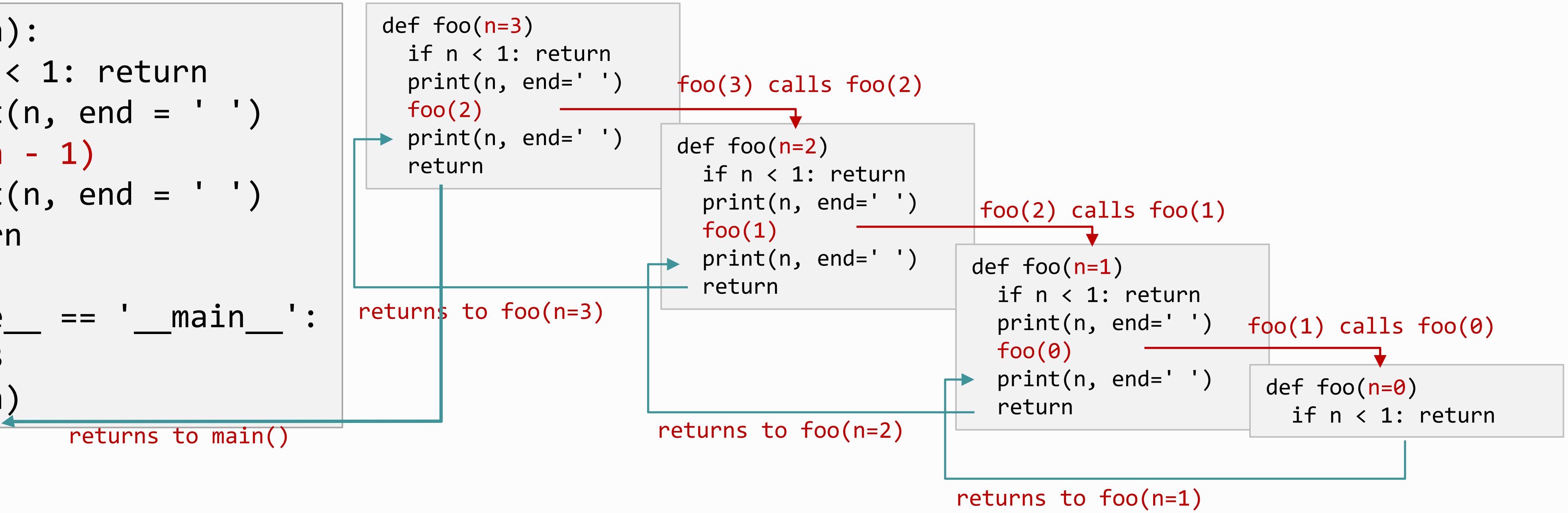
- When main() calls foo(3), main() and **n=3** are pushed to the system stack. It will finish when retured from foo(3).
- **foo(3) prints '3'** and calls foo(2). Then, **foo(2)** and **n=2** are pushed to the system stack.
- Similarly, **foo(2) prints '2'** and calls foo(1). Then **foo(1) prints '1'** and calls foo(0).
- foo(0) goes to if and returns to foo(1). This is the first return ever. Now **foo(1)** popped from the stack **prints '1'.**
- foo(1) returns or finishes. Then, foo(2) popped from the stack prints '2'. It returns to foo(3) popped from the stack.

Example 3

- Predict the output of the following code.

```
def foo(n):
    if n < 1: return
    print(n, end = ' ')
    foo(n - 1)
    print(n, end = ' ')
    return

if __name__ == '__main__':
    n = 3
    foo(n)
```



- When main() calls foo(3), main() and **n=3** are pushed to the system stack. It will finish when retured from foo(3).
- **foo(3) prints '3'** and calls foo(2). Then, **foo(2)** and **n=2** are pushed to the system stack.
- Similarly, **foo(2) prints '2'** and calls foo(1). Then **foo(1) prints '1'** and calls foo(0).
- foo(0) goes to if and returns to foo(1). This is the first return ever. Now **foo(1)** popped from the stack **prints '1'.**
- foo(1) returns or finishes. Then, foo(2) popped from the stack prints '2'. It returns to foo(3) popped from the stack.
- foo(3) prints '3' and returns to the main(). Then, the main() finishes the program.

Reminder

- Recursion is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration).

Recursion Exercise 1

- What is the result of the following call mystery(648)?
  Do it by hands, not running the code, and draw a diagram for the function calls.
- How many kinds of the results will you get if you give many different n?

```
def mystery(n):
    if n < 10:
        return n
    else:
        a = n // 10
        b = n % 10
        return mystery(a + b)
```

Recursion Exercise 1

- What is the result of the following call mystery(648)?
  Do it by hands, not running the code.
- How many kinds of the results will you get if you give many different n?

```
def mystery(n):
    if n < 10:
        return n
    else:
        a = n // 10
        b = n % 10
        return mystery(a + b)
```

```
mystery(648):
    a = 648 // 10         # 64
    b = n % 10            # 8
    return mystery(72)    # mystery(72)
```

Recursion Exercise 1

- What is the result of the following call mystery(648)?
  Do it by hands, not running the code.
- How many kinds of the results will you get if you give many different n?

```
def mystery(n):
    if n < 10:
        return n
    else:
        a = n // 10
        b = n % 10
        return mystery(a + b)
```

```
mystery(648):
    a = 648 // 10         # 64
    b = n % 10            # 8
    return mystery(72)    # mystery(72)
```

```
mystery(72):
    a = 72 // 10       # 7
    b = n % 10         # 2
    return mystery(9) # mystery(9)
```

Recursion Exercise 1

- What is the result of the following call mystery(648)?
  Do it by hands, not running the code.
- How many kinds of the results will you get if you give many different n?

```
def mystery(n):
    if n < 10:
        return n
    else:
        a = n // 10
        b = n % 10
        return mystery(a + b)
```

```
mystery(648):
    a = 648 // 10       # 64
    b = n % 10          # 8
    return mystery(72)  # mystery(72)
```

```
mystery(72):
    a = 72 // 10       # 7
    b = n % 10         # 2
    return mystery(9) # mystery(9)
```

```
mystery(9):
    return 9
```

Recursion Exercise 2

- What is result of the following call, mystery(234) and mystery(5067), respectively? Do it by hands and draw the function call diagrams like the previous example.

```
def mystery(n):
    if n < 10:
        return 10 * n + n
    else:
        a = mystery(n // 10)
        b = mystery(n % 10)
        return a * 100 + b
```

Recursion Exercise 2

- What is result of the following call, mystery(234) and mystery(5067), respectively? Do it by hands and draw the function call diagrams like the previous example.

```
def mystery(n):
    if n < 10:
        return 10 * n + n
    else:
        a = mystery(n // 10)
        b = mystery(n % 10)
        return a * 100 + b
```

```
mystery(234):
    a = ...







    b = ...



    return ...
```

Recursion Exercise 3

- Predict the output of the following code.

```
def mystery(n):
    if n < 10:
        return 'b-return: ' + str(n)
    a = n // 10
    b = n % 10
    c = mystery(a + b)
    print(a, b, c)
    return 'r-return: ' + str(n)
if __name__ == '__main__':
    print(mystery(6480))
```

1:

```
1 0 b-return: n=1
7 3 r-return: n=10
64 9 r-return: n=73
648 1 r-return: n=649
r-return: n=6481
```

2:

```
r-return: n=6481
648 1 r-return: n=649
64 9 r-return: n=73
7 3 r-return: n=10
1 0 b-return: n=1
```

Recursion Exercise 4

- We have bunnies and funnies standing in a line, numbered 1, 2, ... The odd bunnies (1, 3, ..) have the normal 2 ears. The even funnies (2, 4, ..) we'll say have 3 ears, because they each have a raised foot. Recursively return the number of "ears" in the bunny and funny line 1, 2, ... n (without loops or multiplication).

- Expected Ouput

funnyEars(0) → 0
funnyEars(1) → 2
funnyEars(2) → 5
funnyEars(3) → 7
funnyEars(4) → 10
funnyEars(10) → 25
funnyEars(11) → 27

Recursion Exercise 4

- We have bunnies and funnies standing in a line, numbered 1, 2, ... The odd bunnies (1, 3, ..) have the normal 2 ears. The even funnies (2, 4, ..) we'll say have 3 ears, because they each have a raised foot. Recursively return the number of "ears" in the bunny and funny line 1, 2, ... n (without loops or multiplication).

- Expected Ouput

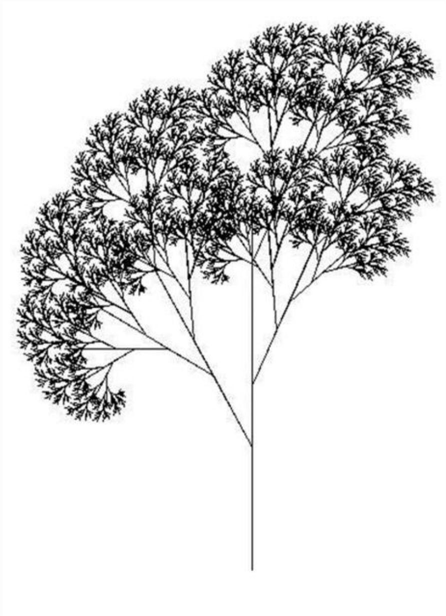
funnyEars(0) → 0
funnyEars(1) → 2
funnyEars(2) → 5
funnyEars(3) → 7
funnyEars(4) → 10
funnyEars(10) → 25
funnyEars(11) → 27

```
def funnyEars(n):
    if n <= 0 return 0
    if n % 2 == 0:
        return ____________________________
    return _____________________

if __name__ == '__main__':
    funnies = [0, 1, 2, 3, 4, 10, 11]
    for n in funnies:
        print('n:', funnyEars(n))
```

Summary

- **Recursion:** *see Recursion*
- Recursion is when a function calls itself
  - It can be used to simplify complex solutions to difficult problems.
- A recursive algorithm **passes the buck** repeatedly to the same function.



학습 정리

1) Recursion: See Recursion

2) 재귀함수(Recursion)를 이용하여 간결한 프로그램을 구현할 수 있다

3) 재귀함수는 base case와 recursive case로 구분된다


파이썬으로 배우는
데이터 구조
수고했습니다
곧 다음 시간에
다시 뵙겠습니다