



# Samsung Innovation Campus

| Coding and Programming

Together for Tomorrow!  
**Enabling People**

Education for Future Generations

Chapter 3.

# **Effective Python Programming**

## **- Function, Closure, and Class**

| Coding and Programming

# Chapter Description

---

## Chapter objectives

- ✓ Learners will be able to use functions and modular programming techniques. Learners will be able to use lambda expressions, which are expressions used to express simple functions. Learners will be able to understand the closure that receives the return value as a function after defining nested functions. In addition, learners will be able to define classes, which are important concepts of object-oriented languages.

## Chapter contents

- ✓ Unit 17. Function
- ✓ Unit 18. Recursion Function Call
- ✓ Unit 19. Lambda
- ✓ Unit 20. Closure
- ✓ Unit 21. Class

Unit 17.

# Function

## Learning objectives

- ✓ Understand functions and be able to create user-defined functions.
- ✓ Be able to name a function using the def statement and call the function using the function name.
- ✓ Be able to use parameters to receive values from inside functions.
- ✓ Understand and be able to use variable parameters, which are techniques for using convenient functions.
- ✓ Be able to use default parameters when declaring functions. When calling a function using this default parameter, the default value is used when the parameter is not passed.
- ✓ Be able to use parameters in any order by specifying keyword arguments when calling functions.
- ✓ Be able to use the return statement to return the result of a function.

## Lesson overview

- ✓ Learn how to use built-in functions.
- ✓ Understand the need for user-defined functions and learn how to define them.
- ✓ Learn how to define and call a user-defined function using the def statement.
- ✓ Learn how to use parameters to pass values to the called function.
- ✓ Learn the types of type errors that occur when using parameters and how to deal with errors when they occur.
- ✓ Learn how to efficiently pass values using variable parameters, default parameters, and keyword parameters.
- ✓ Learn function return statements using return.

### Concepts You Will Need to Know From Previous Units

- ✓ How to use input and output functions.
- ✓ How to use the main string methods such as join, split, and format methods, and use Python built-in functions such as sum and len.
- ✓ How to use loop statements and use a break and continue to break or control loops.

# Keywords

Function

Parameter

Type Error

Variable, Default,  
Keyword Parameter

Return

## 1. Basic Structure of Function

### 1.1. Function is the basic building blocks of a program

- | In this unit, we are going to learn functions that play a very important role in programming.
- | A function is a set of codes that performs a specific operation within a program. It has the characteristic of being executed when it is called.
- | A function is the basic building block of a program.
- | The software we use is made by combining various functions and modules.
- | Think about assembling Lego blocks to create the shape you want. It is the same for making programs. To make a large program with more than tens of thousands of code lines, the necessary functions must be well combined and arranged in order.



Functions, Modules, and Classes



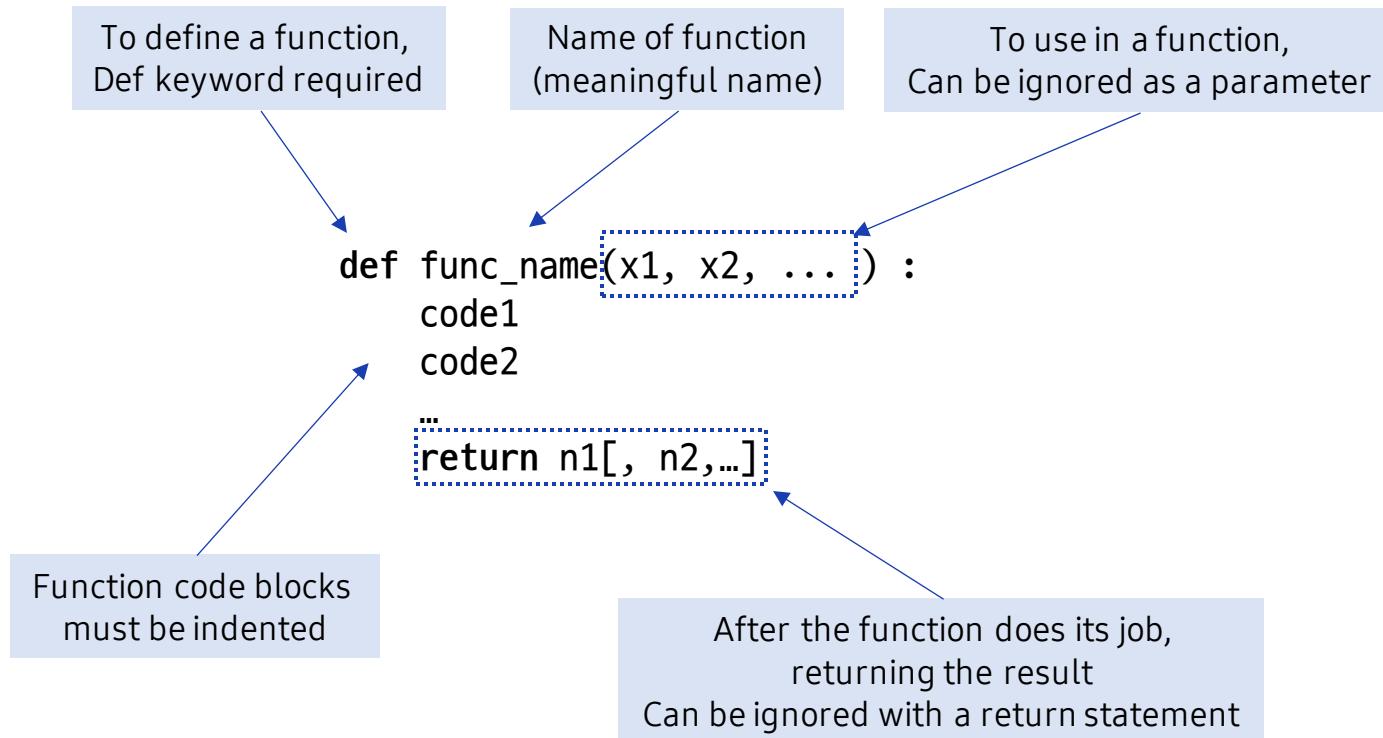
Software

Just like making a car shape with small pieces of blocks, we can make programs by combining functions!

## 1. Basic Structure of Function

### 1.2. Basic structure of function

| Learn the basic structure of functions that are essential in a program.



## 1. Basic Structure of Function

### 1.3. How to define and call functions essential in the program

I Learn how to define a simple function and call it.

```
1 def print_star(): # Define a function to output an asterisk
2     print('******')
3
4 print_star()      # Call a function to output an asterisk
```

\*\*\*\*\*

 Line 1, 2, 4

- In def print\_star():, a function is defined using the def keyword. After writing the name of the function, add parentheses and colon :. A colon ":" means that a block of code follows. This is the same as for if, for, and while we learned in the previous unit.
- The statements that make up the function are entered. The statements constituting the function must be indented. This chunk is called a block, and a function executes this block.
- The 4th line is a statement that calls a function in the form of print\_star() and executes it.

## 1. Basic Structure of Function

### 1.4. Call a function

The print\_star function is a set or block of commands defined to do something. You can check that this set is executed every time it is called from the outside.

```
1 def print_star():
2     print('*****')
3
4 print_star() # Asterisk output function call 1
5 print_star() # Asterisk output function call 2
6 print_star() # Asterisk output function call 3
7 print_star() # Asterisk output function call 4
```

```
*****
*****
*****
*****
```

#### Line 1,2

- Define a print\_star function that prints asterisk.
- Lines 4 to 7 call the print\_star function.

## 1. Basic Structure of Function

### 1.5. Advantages of functions

- I Advantages of functions based on what we have learned so far.
  1. Structured programming is possible because one large program can be divided into several parts.
  2. It is possible to reuse functions in other programs.
  3. Code readability is increased.
  4. Even when modifying a program, maintenance is easy because only some functions need to be modified.
  5. If you use an already developed function, you can save time and money in program development.

## 2. Function Parameter

### 2.1. Key terms related to functions

I Learn the key terms related to functions.

- ▶ **Parameter** : It is a **variable** defined in the function or method header. It receives the actual value when the function is called.

**Ex** m and n of def foo(m, n):

- ▶ **Argument**: It refers to the actual **value** passed when a function or method is called. It is simply called an **argument**.

**Ex** 3 and 4 of foo(3, 4)

## 2. Function Parameter

### 2.1. Key terms related to functions



When calling a function, it is the parameter that receives the actual value, and the actual value that is passed the value is called the **argument**.

Variables m, n with values 3 and 4 to be passed: parameters

```
def foo(m, n):  
    code  
    ...  
    return n1[, n2,...]
```

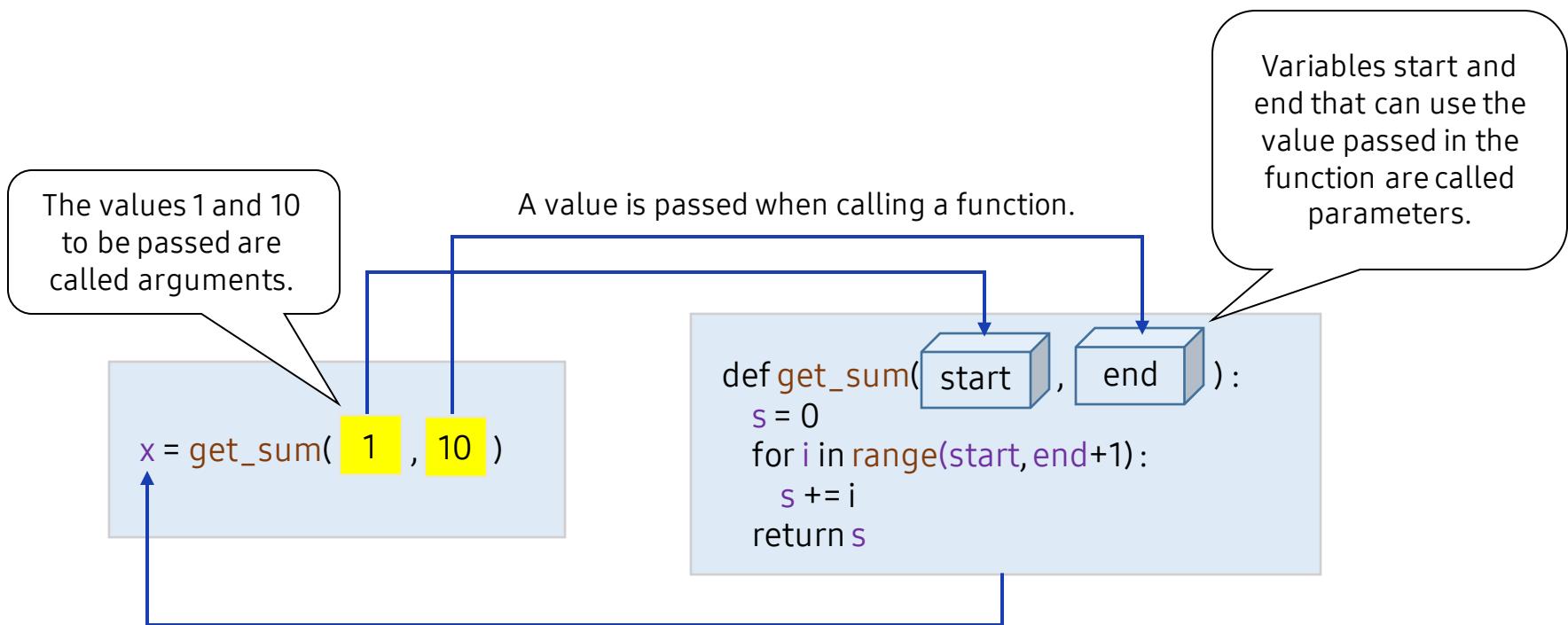
```
foo(3, 4)
```

Values to be passed to the function called foo 3, 4: Arguments

## 2. Function Parameter

### 2.1. Key terms related to functions

- | Example of a function that returns the sum of integers from 1 to 10
  - ▶ The following function calculates the sum of integers from 1 to 10 and returns it.



## 2. Function Parameter

### 2.2. Asterisk output function with parameters

- | Create an asterisk output function with parameters and call it with arguments.
- | With an argument of 4, the asterisk output can be repeated 4 times.

```
1 # Program that repeats the asterisk output for parameter n times
2 def print_star(n):
3     for _ in range(n):
4         print('*****')
5
6 print_star(4) # Give the argument value of 4 for asterisk output
```

```
*****
*****
*****
*****
```

#### Line 2, 3, 4, 6

- `print_star` takes the parameter `n` as input.
- It loops `n` times in the `for` loop and prints a star.
- Give the argument value 4 to `print_star`.

## 3. Parameter and Type Error

### 3.1. Case A using parameters

Control the number of times to output 'Hello' through the argument as follows. n inside the function is the parameter that receives the argument.

```
1 def print_hello(n):          # Repeat output using parameters
2     print('Hello ' * n)
3
4 print('It prints Hello twice.')
5 print_hello(2)
6 print('It prints Hello 3 times.')
7 print_hello(3)
8 print('It prints Hello 4 times.')
9 print_hello(4)
```

It prints Hello twice.  
Hello Hello  
It prints Hello 3 times.  
Hello Hello Hello  
It prints Hello 4 times.  
Hello Hello Hello Hello

## 3. Parameter and Type Error

### 3.2. Case B using parameters

- | Find the sum of the argument values using parameters and print them.
- | The sum of arguments 10, 20, and 100, 200 can be calculated inside the function.

```
1 def print_sum(a, b):          # Function with two parameters
2     result = a + b
3     print('The sum of' a, 'and', b, 'is', result)
4
5 print_sum(10, 20)
6 print_sum(100, 200)
```

The sum of 10 and 20 is 30.

The sum of 100 and 200 is 300.

#### Line 1,2,3,5

- print\_sum takes a and b as arguments.
- Assigns the value of  $a + b$  to result and prints it.
- It is called by putting 10 and 20 in print\_sum.

## 3. Parameter and Type Error

### 3.3. Parameter type error

- If only one argument is given when calling a function, the `TypeError` message appears, and the function does not execute because there is no value of the parameter `b` required by the function.
- Matching the number of arguments is very important.

 `TypeError`

```
1 def print_sum(a, b):
2     result = a + b
3     print('The sum of' a, 'and', b, 'is', result)
4
5 print_sum(10) # error
```

In this case, the number of arguments does not match.

```
TypeError Traceback (most recent call last)
<ipython-input-13-ec8bc3c39141> in <module>
      3     print('The sum of' a, 'and', b, 'is', result)
      4
----> 5 print_sum(10) # error

TypeError: print_sum() missing 1 required positional argument: 'b'
```

## 4. Arbitrary Parameter

### 4.1. Arbitrary argument passing

I Learn arbitrary argument passing.

- ▶ When using a function, there are cases where the number of arguments is not determined. This is called an arbitrary argument.
- ▶ When receiving an arbitrary argument from a function, put an asterisk (\*) in front of the parameter. A variable inside a function that receives an arbitrary argument is an **arbitrary parameter**.
- ▶ Arbitrary parameters can be used in for - in statements, like tuples and lists.

## 4. Arbitrary Parameter

### 4.1. Arbitrary argument passing

| Create a function that works well even with 3 or 2 arguments through arbitrary argument passing.

```
1 def greet(*names):
2     for name in names:
3         print('Hello', name, '!')
4
5 greet('A', 'B', 'C')      # 3 arguments
6 greet('James', 'Thomas')  # 2 arguments
```

```
Hello A !
Hello B !
Hello C !
Hello James !
Hello Thomas !
```

#### Line 1, 2, 3

- It takes an arbitrary argument called names.
- Each element in names is printed using the for statement.

## 4. Arbitrary Parameter

### 4.1. Arbitrary argument passing

- | Count the number of arbitrarily passed arguments using the len function
  - ▶ It is also possible to print the number of arguments passed arbitrary as follows by using the len function.

```
1 def foo(*args):  
2     print('The number of arguments:', len(args))  
3     print('Arguments :', args)  
4  
5 foo(10, 20, 30)
```

The number of arguments: 3  
Arguments : (10, 20, 30)

#### Line 1, 2, 3

- It takes an arbitrary argument called args.
- Print the number of args using the len function.
- Print the elements of args.
- Looking at the result, the arguments 10, 20, and 30 of the function are output in the form of a tuple.

## 4. Arbitrary Parameter

### 4.2. Finding the sum of numbers entered as arguments

- | Create a program that finds the sum of the numbers entered as arguments.
- | If the number of arguments to be passed to the sum\_nums function is not known in advance, all arguments can be received in a tuple format by using an arbitrary parameter called \*numbers that takes arbitrary arguments.

```
1 def sum_nums(*numbers):
2     result = 0
3     for n in numbers:
4         result += n
5     return result
6
7 print(sum_nums(10, 20, 30))      # Print the sum of arguments of 10, 20, 30
8 print(sum_nums(10, 20, 30, 40, 50)) # Print the sum of arguments of 10, 20, 30, 40, 50
```

60  
150

#### Line 1, 2, 3, 4, 5

- The sum\_nums function takes an arbitrary parameter called numbers.
- Through a for loop, the elements of numbers are stored in result.
- Return result.

## 5. Default Parameter

### 5.1. Default parameters and robust programs

| Create a robust program with default parameters. Why does the code below cause an error?



```
1 def print_star(n):      # One argument required
2     for _ in range(n):
3         print('*****')
4
5 print_star() # An error occurs because there is no argument
```

```
TypeError                                     Traceback (most recent call last)
<ipython-input-18-6d1876493ea4> in <module>
      3     print('*****')
      4
----> 5 print_star() # An error occurs because there is no argument

TypeError: print_star() missing 1 required positional argument: 'n'
```



```
1 def print_star(n = 1): # The parameter n has a default value of 1
2     for _ in range(n):
3         print('*****')
4
5 print_star() # Execute without error even if there is no argument
```

\*\*\*\*\*

If the default parameter value is set to 1, no error occurs.

## 5. Default Parameter

### 5.2. Calling a function without arguments

I Learn the case of calling a function without arguments.

- ▶ In order to assign a specific task to a function, it is necessary to enter the correct argument. Sometimes, however, it is convenient to use the default value to prevent the error and work more flexible.
- ▶ In this case, the default parameter is used.
- ▶ As shown in the code below, you can assign a default value such as = 1 to a parameter.

```
1 def print_star(n = 1): # The parameter n has a default value of 1
2     for _ in range(n):
3         print('*' * n)
4
5 print_star() # Execute without error even if there is no argument
```

```
*****
```

- ▶ Even if it is called without an argument, the default value of 1 is passed to the parameter n. Therefore, a single asterisk line is normally output.

## 5. Default Parameter

### 5.2. Calling a function without arguments

 Focus If there is a value passed as an argument, the default value is not taken (if ignored).

- ▶ If argument 2 is input when calling the print\_star function, it does not take the default parameter 1, but takes the argument value 2 as the n value.
- ▶ In this case, it outputs two asterisk lines. Default parameter 1 is ignored.

```
1 def print_star(n = 1): # The parameter n has a default value of 1
2     for _ in range(n):
3         print('*****')
4
5 print_star(2) # Since the argument value is 2, the default parameter n=1 is not performed
```

\*\*\*\*\*  
\*\*\*\*\*

## 5. Default Parameter

### 5.3. Practice using default parameter

- | Create a function called div that takes two parameters.
- | Assign a default value of 2 to the second parameter.

```
1 def div(a, b = 2):  
2     return a / b  
3  
4 print('div(4) =', div(4))  
5 print('div(6, 3) =', div(6, 3))
```

```
div(4) = 2.0  
div(6, 3) = 2.0
```

#### Line 1, 2

- a must receive an argument value, but b has already been assigned a default value of 2 even if it does not receive an argument value.
- Returns a / b.
- Since there is only 4 in the argument value, b is assigned as 2 (default value).

## 5. Default Parameter

### 5.3. Practice using default parameter

 SyntaxError: non-default argument follows default argument

- ▶ What if we assign a default value of 2 to the first parameter and skip the second parameter as follows?

```
1 def div(a = 2, b):  
2     return a / b
```

```
File "<ipython-input-22-48c2fea5df1e>", line 1  
def div(a = 2, b):  
^
```

SyntaxError: non-default argument follows default argument

- ▶ Default parameters should be assigned to all variables or from the last variable in the order of appearance of the parameters.

## 5. Default Parameter

### 5.3. Practice using default parameter

- | Learn the function call statement considering default parameters and the argument values
- | In the table below, div() is the same as div(1, 2). Blue 1 and 2 are default arguments. When calling a function, if empty, they are automatically given.
- | If div(4) is inserted, div(4, 2) is passed as an argument as shown in the table.

```
1 def div(a = 1, b = 2):  
2     return a / b  
3  
4 print('div() =',div())  
5 print('div(4) =',div(4))  
6 print('div(6, 3) =',div(6, 3))
```

div() = 0.5  
div(4) = 2.0  
div(6, 3) = 2.0

Function Call Statement	Arguments passed during actual execution (the blue argument is the default value)	Return Value
div()	div(1, 2)	0.5
div(4)	div(4, 2)	2.0
div(6, 3)	div(6, 3)	2.0

## 6. Keyword Parameter

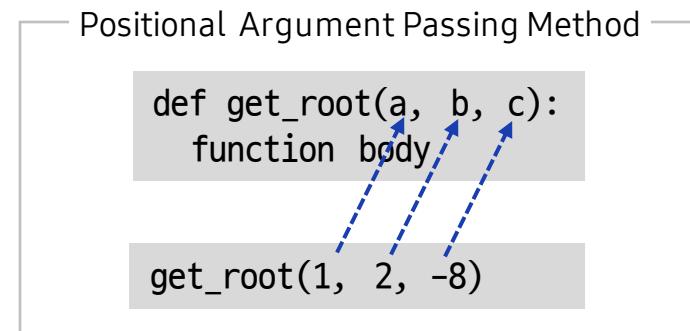
### 6.1. Keyword parameters and argument passing method

#### I Keyword parameter passing method

- The default argument passing method in Python is called the positional argument method.

```
1 def get_root(a, b, c):
2     r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
3     r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
4     return r1, r2
5
6 # When calling a function, use a constant value as an argument
7 # Return the result value using result 1 and result 2
8 result1, result2 = get_root(1, 2, -8)
9 print('The result value is', result1, 'or', result2)
```

The result value is 2.0 or -4.0



## 6. Keyword Parameter

### 6.2. Necessity of keyword parameter

I Learn why we need keyword parameter.

- ▶ When calling a function, instead of passing only the value of the argument, the method of passing the argument by specifying the name is called a keyword parameter.
- ▶ In the code below, if -8, 2, 1 is given as an argument, the result is different from when 1, 2, -8 is given as an argument.

```
1 result1, result2 = get_root(-8, 2, 1)
2 print('The result value is', result1, 'or', result2)
```

The result value is -0.25 or 0.5



When passing a value to a parameter, it is passed in the order a, b, c. In the positional argument passing method, the order in which the arguments are listed is important.

## 6. Keyword Parameter

### 6.3. Advantages of keyword parameters

I Keyword parameters can be positioned anywhere.

- ▶ The keyword parameter is a method where the argument value is determined by the keyword regardless of the position.

```
1 # All three codes below are a=1, b=2, c=-8, so the result is the same
2 result1, result2 = get_root(a = 1, b = 2, c = -8)
3 print('The result value is', result1, 'or', result2)
```

The result value is 2.0 or -4.0

```
1 result1, result2 = get_root(a = 1, c = -8, b = 2)
2 print('The result value is', result1, 'or', result2)
```

The result value is 2.0 or -4.0

```
1 result1, result2 = get_root(c = -8, b = 2, a = 1)
2 print('The result value is', result1, 'or', result2)
```

The result value is 2.0 or -4.0

- ▶ The result of codes above is all the same.
- ▶ When keyword arguments are used, the position of the arguments is not important.

Positional Argument Passing Method

```
def get_root(a, b, c):
    function body
get_root(a=1, c=-8, b=2)
```

## 6. Keyword Parameter

### 6.4. Notes on using keyword parameters

 Note the order of positional and keyword parameters.

- | When passing an argument to a Python function, there are two methods: a method of passing by a positional argument and a method of passing by a keyword argument. Also, there is a method to mix those two methods.
- | When mixing the two methods, the keyword argument must come after the positional argument.

#### SyntaxError: positional argument follows keyword argument

- ▶ When a keyword argument and a positional argument are used together, the positional argument must come first. (If you do the following, an error will occur.)

```
1 # An error occurs when:  
2 result1, result2 = get_root(c = -8, b = 2, 1)  
3 print('The result value is', result1, 'or', result2)
```

```
File "<ipython-input-29-e61e9d57ef12>", line 2  
    result1, result2 = get_root(c = -8, b = 2, 1)  
          ^
```

SyntaxError: positional argument follows keyword argument

## 6. Keyword Parameter

### 6.4. Notes on using keyword parameters

#### Mix keyword arguments and positional arguments

- When using a mixture of keyword arguments and positional arguments, be sure to write the positional arguments first.
- In the following case, the keyword argument value becomes -8 with c = -8. The remaining a and b values are passed in a positional argument method.

```
1 result1, result2 = get_root(1, 2, c = -8)
2 print('The result value is', result1, 'or', result2)
```

The result value is 2.0 or -4.0

```
def get_root(a, b, c):
    function body
```

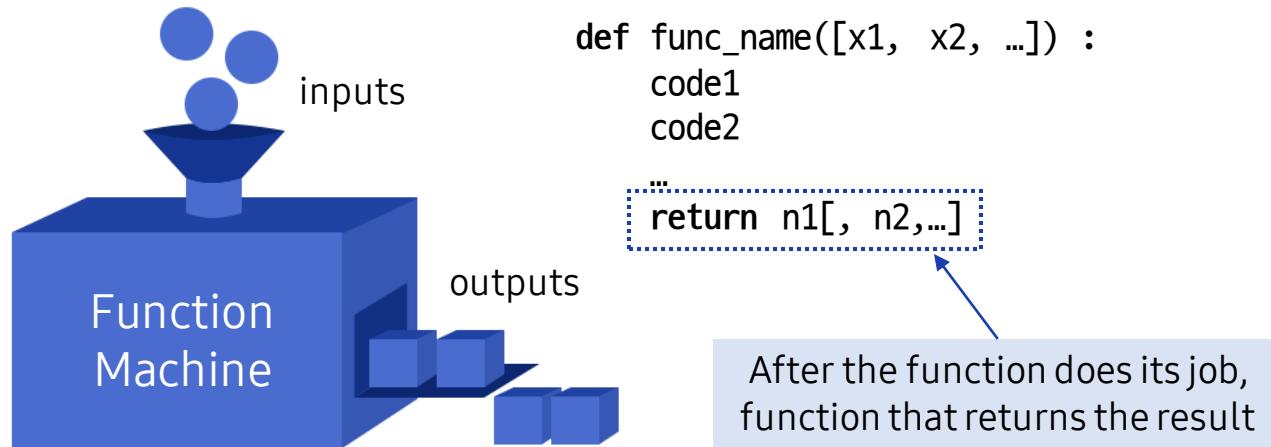
```
get_root(1, 2, c = -8)
```

## 7. Return

### 7.1. A return statement that returns a result

I Learn the return statement.

- ▶ In general, it is assumed that the inside of a function is a black box.
- ▶ The inside of a function has specific code. It can perform a given task and return a result.
- ▶ The code below is an example of a Python function that returns one or more values using the return keyword.



## 7. Return

### 7.1. A return statement that returns a result

- I Return result to get\_sum function that returns the sum of two values
  - ▶ Let's put the sum of two numbers a and b into the result variable and return this result through the return statement.
  - ▶ n1 returns 30, the sum of 10 and 20, and n2 returns 300, the sum of 100 and 200.

```
1 def get_sum(a, b):      # A function that returns the sum of two numbers
2     result = a + b
3     return result        # Return the result using the return statement
4
5 n1 = get_sum(10, 20)
6 print('The sum of 10 and 20 =', n1)
7
8 n2 = get_sum(100, 200)
9 print('The sum of 100 and 200 =', n2)
```

The sum of 10 and 20 = 30

The sum of 100 and 200 = 300

## 7. Return

### 7.1. A return statement that returns a result

I Modify the get\_sum function that returns the sum of two values

- When returning a value using return, it is possible to return the result of the expression by entering an expression after the return keyword as follows.
- Let's see how to use the get\_sum function and return statement that returns the sum of two values.

```
1 def get_sum(a, b):  
2     return a + b  
3  
4 result = get_sum(100, 200)  
5 print('The sum of two numbers', result)
```

The sum of two numbers 300

- The previous page and this page have the same result.

## 7. Return

### 7.2. A return statement that returns a tuple

#### I Return two values as a tuple

- ▶ Another way is to return two values as a tuple. In this case, the return value can be received in the form of a tuple from outside.

```
1 def get_root(a, b, c):
2     r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
3     r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
4     return r1, r2
5
6 # When calling a function, use a constant value as an argument
7 # Return the result value using result 1 and result 2
8 result1, result2 = get_root(1, 2, -8)
9 print('The result value is', result1, 'or', result2)
```

Returns the two roots r1 and r2 of the quadratic equation.

Result1 and result2 receive the values of r1 and r2.

The result value is 2.0 or -4.0

#### Line 2,3

- The quadratic formula is expressed in Python modifiers.
- The quadratic formula is  $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ .

# | Paper coding

**Try to fully understand the basic concept before moving on to the next step.**

**Lack of understanding basic concepts will increase your burden in learning this course, which may make you fail the course.**

**It may be difficult now, but for successful completion of this course we suggest you to fully understand the concept and move on to the next step.**

**Q1.** Define a function named my\_greet that prints "Welcome." and call this function twice to print this greeting twice.

Condition for Execution	Welcome. Welcome.
Time	5 minutes

 Write the entire code and the expected output results in the note.

## Q2.

Implement the max2(m, n) function, which takes two parameters named m and n, and returns the larger of these two values, and the min2(m, n) which also takes two parameters named m and n and returns the smaller of these two values. Assign 100 and 200 as arguments and call two functions to check the results.

Condition for Execution

The greater of 100 or 200 is : 200  
The smaller of 100 or 200 is : 100

time

10 minutes



Write the entire code and the expected output results in the note.

**Q3.**

We want to change the value of the mile, the unit mainly used in the United States, to the value of the kilometer, the international standard unit. Implement the mile2km(mi) function that takes a mile value as a parameter and returns it in kilometers and calls this function to output 1 to 5 miles as kilometers. In this case, use for - in range to make it repeatable. (Define 1 mile as 1.61 km.)

Condition for Execution	1 mile = 1.61 kilometers 2 mile = 3.22 kilometers 3 mile = 4.83 kilometers 4 mile = 6.44 kilometers 5 mile = 8.05 kilometers
Time	5 minutes



Write the entire code and the expected output results in the note.

**Q4.**

Implement the cel2fah(cel) function that takes a temperature in Celsius (Celsius) as a parameter and returns it in Fahrenheit. Then, call this function to change from 10 to 50 degrees Celsius in units of 10 degrees, and output it in Fahrenheit temperature as the following result.

Condition for Execution

10 degrees Celsius = 50.0 degrees Fahrenheit
20 degrees Celsius = 68.0 degrees Fahrenheit
30 degrees Celsius = 86.0 degrees Fahrenheit
40 degrees Celsius = 104.0 degrees Fahrenheit
50 degrees Celsius = 122.0 degrees Fahrenheit

Time

5 minutes

conversion formula : Fahrenheit = Celsius  $\times$  9/5 + 32



Write the entire code and the expected output results in the note.

| Let's code

# 1. String Processing

## 1.1. Various string processing functions and methods

- | Let's dive deeper into the string methods we learned in unit 14.
- | Learn the count method applicable to strings.
- | Also, let's get the length of a string using the len function.
  - ▶ The string count method returns the number of occurrences of a substring in a string.

```
1 birth = '1998.05.13'      # Date of birth example  
2 birth.count('.')          # Tell you how many times '.' occurs
```

2

- ▶ In addition to the methods mentioned above, built-in functions can also be applied to strings. The len function returns the length of a string.

```
1 birth = '1998.05.13'  
2 print('length of birth :', len(birth))    # Return the length of the string birth
```

length of birth : 10

## 1. String Processing

### 1.2. String processing using built-in functions

- | You can use the max and min functions to get the largest and smallest characters. The criterion of greater or lesser is based on the Unicode code value.
- | Unicode is an industry standard code system to defines characters that can be used in computers.
- | The ord function can get the Unicode value for a character, and the chr value returns the character corresponding to the input Unicode value.

```
1 birth = '1998.05.13'  
2 max(birth)
```

'9'

```
1 ord(max(birth)) # Return a Unicode value of 9
```

57

```
1 ord(min(birth)) # Return the Unicode value of the smallest Unicode value within the string birth
```

46

```
1 chr(57), chr(46) # Return the character corresponding to Unicode values 57 and 46
```

('9', '.')

# 1. String Processing

## 1.3. Various string methods

| Learn various string methods and how to use them.

```
1 a = 'I love python'  
2 a.count('o')      # Number of occurrences of the 'o' character
```

2

```
1 a.count('k')      # Number of occurrences of the 'k' character
```

0

```
1 a.find('o')      # Return the index when the 'o' character appears
```

3

```
1 a.find('on')     # Return the index when the 'on' character appears
```

11

```
1 a.startswith('I')  # Does it start with the character 'I'?
```

True

```
1 a.endswith('python') # Does it end with the character 'python'?
```

True

# 1. String Processing

## 1.3. Various string methods

| The string methods for uppercase/lowercase letters

```
1 a = 'I Love Python'  
2 g = 'i love python'  
3 h = 'I LOVE PYTHON'
```

```
1 a.upper()      # Convert all uppercase
```

```
'I LOVE PYTHON'
```

```
1 h.lower()      # Convert all lowercase
```

```
'i love python'
```

```
1 a.swapcase()   # Convert uppercase to lowercase and lowercase to uppercase
```

```
'i LOVE pYTHON'
```

```
1 a.istitle()    # Is it a title text?
```

```
True
```

# 1. String Processing

## 1.4. Using the string module

### Print uppercase/lowercase letters using string module

- ▶ Python has a module called string that helps with string processing. In this module, ascii\_uppercase contains uppercase letters and ascii\_lowercase contains lowercase letters.
- ▶ Print the uppercase letters of the English alphabet using the string module.

```
1 import string  
2 src_str = string.ascii_uppercase  
3 print('src_str =', src_str)
```

src\_str = ABCDEFGHIJKLMNOPQRSTUVWXYZ

- ▶ The following is how to print lowercase letters of the alphabet.

```
1 src_str = string.ascii_lowercase  
2 print('src_str =', src_str)
```

src\_str = abcdefghijklmnopqrstuvwxyz

# 1. String Processing

## 1.4. Using the string module

### I Try slicing the alphabet

- ▶ If the string up to 'ABCDE..YZ' is moved one character at a time, what method should be used to create a string like 'BCDEF...ZA'?
- ▶ In this case, we can solve this problem with the following simple slicing and addition operations.

```
1 src_str = string.ascii_uppercase  
2 dst_str = src_str[1:] + src_str[:1]  
3 print('dst_str =', dst_str)
```

```
dst_str = BCDEFGHIJKLMNOPQRSTUVWXYZA
```

#### Line 1, 2

- Assign uppercase letters A to Z to src\_str.
- Add [:1], the letter A, to [1:], the string from B to Z. In this case, src\_str[1:] + src\_str[:1] is used to create concatenated characters.

# 1. String Processing

## 1.5. Using the index method

- | The position of the letter A in src\_str can be asked with the following index method.
- | Conversely, if n obtained by using this index is searched for dst\_str as an index, it can be confirmed that the letter of dst\_str corresponding to position 'A' of src\_str is 'B'.

```
1 n = src_str.index('A')
2 print('The index of src_str =', n)
3 print('The character in dst_str at position A in src_str =', dst_str[n])
```

The index of src\_str = 0

The character in dst\_str at position A in src\_str = B

## 2. Finding the Solution of a Quadratic Equation Using Parameters

### 2.1. Solutions of quadratic equations

| If the coefficients of the quadratic equation are a, b, and c, respectively, the quadratic formula is as follows. Assume that a is non-zero here.

$$ax^2 + bx + c = 0$$

| Enter the values corresponding to a( $a \neq 0$ ), b, and c according to the equation.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

| Let's find the solution when the coefficients are a, b, and c. Save this solution in variables r1 and r2 and print it out.

```
1 def print_root(a, b, c):
2     r1 = (-b + (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
3     r2 = (-b - (b ** 2 - 4 * a * c) ** 0.5) / (2 * a)
4     print('The solution is', r1, 'or', r2)
5
6 # Solve quadratic equations with different coefficient values
7 print_root(1, 2, -8)
8 print_root(2, -6, -8)
```

The solution is 2.0 or -4.0

The solution is 4.0 or -1.0

- It receives three externally passed arguments as a, b, and c parameters, and performs quadratic formula operation in the function body. As a result, it is a function code that outputs r1 and r2.
- Using functions makes the code much more concise and easier to use.

## 3. Multiple Return Statement

### 3.1. A function that returns the area and perimeter of a circle

- | Use the radius of the circle to find the area and perimeter of the circle.
- | The `circle_area_circum(radius)` function is implemented to receive a radius 10 as an input and return two values, an area and a circumference.

```
def circle_area_circum(radius):
    area = 3.14 * radius ** 2
    circum = 2 * 3.14 * radius
    return area, circum
```

## 3. Multiple Return Statement

### 3.1. A function that returns the area and perimeter of a circle

- | Return statements that return two or more values are called multiple return statements.
- | In multiple return statements, two values separated by a comma are returned as a tuple type.

```
1 def circle_area_circum(radius):
2     area = 3.14 * radius ** 2
3     circum = 2 * 3.14 * radius
4     return area, circum # Returns a tuple - the return value (area, circum)
5
6 radius = 10
7 area, circum = circle_area_circum(radius) # Return the area and perimeter of a circle
8 print("The area of a circle of radius {} is {:.1f}, and the circumference of the circle is {:.1f})".format(radius, area, circum))
```

The area of a circle of radius 10 is 314.0, and the circumference of the circle is 62.8

#### Line 1,7-8

- In the code above, the `circle_area_circum` function receives a radius as a parameter and returns two values corresponding to the area and circumference of the circle.
- Outside the function of Line 7-8, the function `circle_area_circum` is called and the calculation result is stored in the variables `area` and `circum` and print them out.

## 4. Global Variable

### 4.1. The concept of global variables

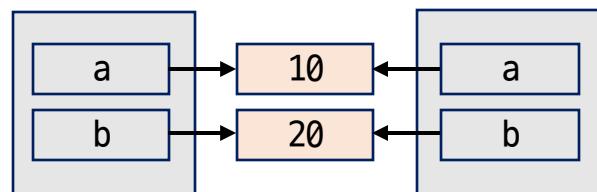
I Learn the term global variable.

- ▶ A global variable is a variable declared outside a function and it is available to the entire code area.
- ▶ If you look at the code and diagram below, you can see that the variables a and b are called and used in both the script file with the Python code and the print\_sum function.

```
1 def print_sum():
2     result = a + b
3     print('print_sum() inside : the sum of', a, 'and', b, 'is', result)
4
5 a = 10
6 b = 20
7 print_sum()
8 result = a + b
9 print('print_sum() outside : the sum of', a, 'and', b, 'is', result)
```

print\_sum() inside : the sum of 10 and 20 is 30  
print\_sum() outside : the sum of 10 and 20 is 30

Global variables a and b can be used in the entire code.



Python  
script file

print\_sum function  
using global variables

## 4. Global Variable

### 4.1. The concept of global variables

#### I Global Variables and Example Code

- ▶ What if a global variable defined outside a function is changed inside the function?
- ▶ Take a look at the code below. Inside the print\_sum function, a and b were changed to 100 and 200, respectively.

```
1 def print_sum():
2     a = 100
3     b = 200
4     result = a + b
5     print('print_sum() inside : the sum of', a, 'and', b, 'is', result)
6
7 a = 10
8 b = 20
9 print_sum()
```

```
print_sum() inside : the sum of 100 and 200 is 300
```

## 4. Global Variable

### 4.1. The concept of global variables

#### I Change of global variable and code to check

- ▶ Change the global variable values a and b inside the function and execute the code to check the values externally.

```
1 def print_sum():
2     a = 100 ← Create new a and b variables referencing 100, 200
3     b = 200
4     result = a + b
5     print('print_sum() inside : the sum of', a, 'and', b, 'is', result)
6
7 a = 10 ← Create variables a and b referencing 10 and 20
8 b = 20
9 print_sum()
10 result = a + b
11 print('print_sum() outside : the sum of', a, 'and', b, 'is', result)
```

```
print_sum() inside : the sum of 100 and 200 is 300
print_sum() outside : the sum of 10 and 20 is 30
```

- ▶ After executing print\_sum, add a and b once again outside the function, assign it to result, and print the result.
  - Were a and b changed to 100, 200 inside the function changed outside the function?
  - If you check the result, you can see that the a and b values inside print\_sum are different from the a and b values outside the function.

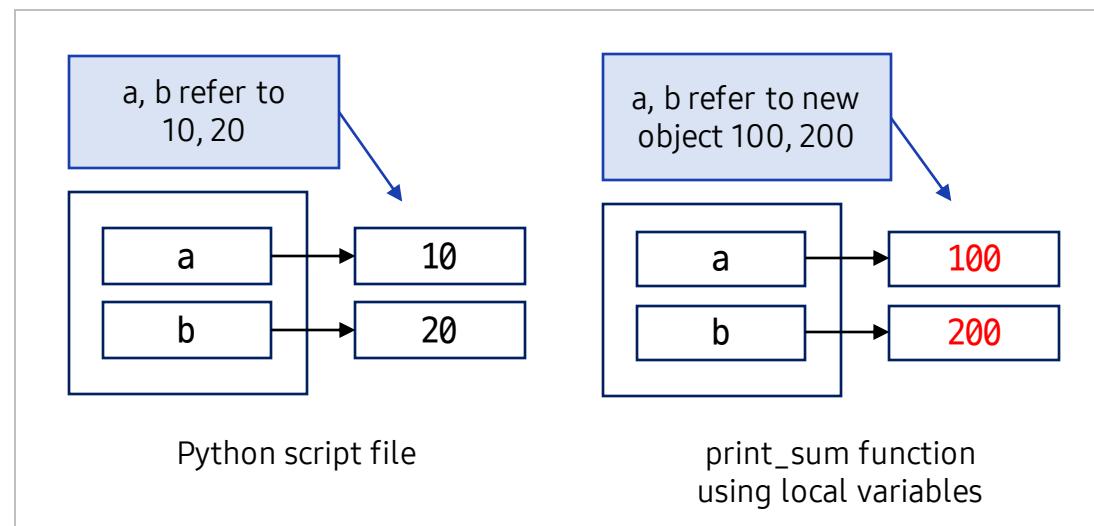
## 4. Global Variable

### 4.2. The concept of local variables



Results and interpretation of the previous code

- | The Python assignment statement is : a = 100, b = 200
- | Inside a function, this assignment statement creates a local variable.
- | Global variables a and b refer to global values 10 and 20.
- | Inside the print\_sum function, a and b refer to the new objects 100 and 200.



- ▶ As shown in the figure on the left, global variables a and b exist in the Python script file.
- ▶ Variables a and b inside the `print_sum` function refer to a separate new object, not the object referenced by the global variable.

## 4. Global Variable

### 4.3. global keyword

| Learn how to refer to global variables using the global keyword.

```
1 def print_sum():
2     global a, b      # a and b use a and b declared outside the function
3     a = 100
4     b = 200
5     result = a + b
6     print('print_sum() inside : the sum of', a, 'and', b, 'is', result)
7
8 a = 10
9 b = 20
10 print_sum()
11 result = a + b
12 print('print_sum() outside : the sum of', a, 'and', b, 'is', result)
```

Global variables a and b refer to 10 and 20

Global variables a and b refer to 100 and 200

```
print_sum() inside : the sum of 100 and 200 is 300
print_sum() outside : the sum of 100 and 200 is 300
```

## 4. Global Variable

### 4.3. global keyword



#### Beware of SyntaxError

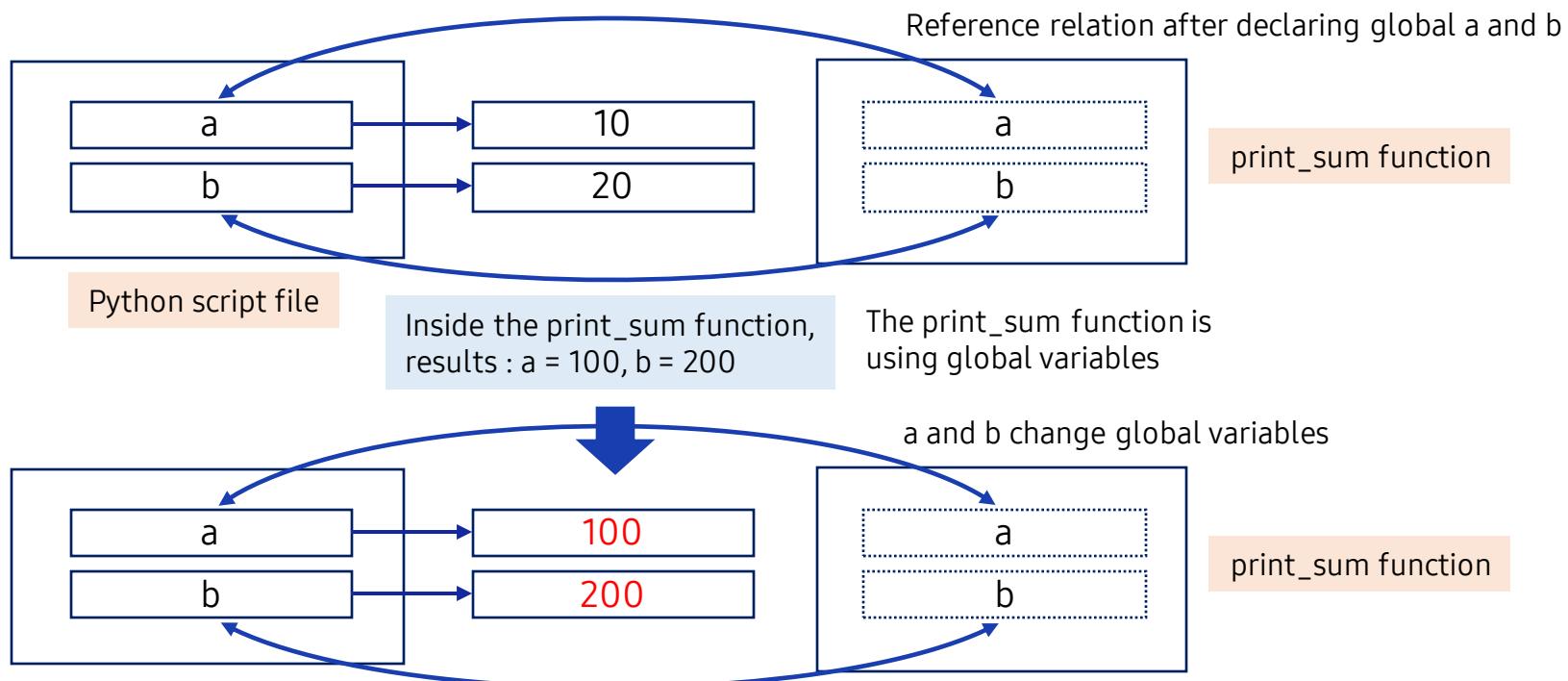
```
1 def print_sum():
2     global a = 0      # a and b use a and b declared outside the function
File "<ipython-input-55-15b3891832ee>", line 2
    global a = 0      # a and b use a and b declared outside the function
SyntaxError: invalid syntax
```

- ▶ Note: `global a = 100` causes a syntax error. The `global` declaration and assignment operator cannot be used at the same time.

## 4. Global Variable

### 4.4. Global variable reference process

The process of using global variables a and b in a function through an explicit global declaration is as follows.



- If you change the value of a variable declared as global a or b, the value is also changed outside the function.

## 4. Global Variable

### 4.5. Notes on using global constants in global variables

#### Global Variables vs. Global Constants

- ▶ Using global variables is a very bad practice in all programming languages, not just Python. When the length of the code becomes long, global variables become the main cause of errors.
- ▶ On the other hand, global constants are not necessarily bad.
- ▶ A global constant is declared with the keyword `global` as follows. It can be declared outside of a function and referenced throughout the module. Global constant values are usually in uppercase.
- ▶ A constant is a number that does not change.
- ▶ Looking at the code below, after assigning a value of 1024 to a variable named `GLOBAL_VALUE`, it was declared as `global GLOBAL_VALUE` to call and use the value of this variable in the `foo` function.

```
1 GLOBAL_VALUE = 1024 # Capitalizing global constants makes them easier to distinguish
2
3 def foo():
4     global GLOBAL_VALUE
5     a = GLOBAL_VALUE * 100
6     print(a)
7
8 foo()
```

102400

## 5. Functions and Methods

### 5.1. Difference between function and method

- | Functions and methods are equivalent in that they perform a defined action to do a specific task. However, there is a difference between the two.
- | Function – A module in programming that receives parameters (optionally ignored) when called, performs a specific operation, and returns the result if necessary.
- | Method - A function attached to an object dealt with in object-oriented programming.
  - ▶ A method can call objects with different values, called instances.
  - ▶ This will be covered in class later.

## 5. Functions and Methods

### 5.2. format method

- | Learn how to do formatted output when outputting a string. Formatted output can be produced by calling the string method `format`.
- | Placeholder: Braces {} in a string used to output an argument are called a placeholder.
- | If you put an argument in the `.format()` method as follows, this argument value appears in the placeholder.

```
1 '{} Python!'.format('Hello')
```

```
'Hello Python!'
```

What goes into the placeholder

```
1 '{} Python!'.format('Hi')
```

```
'Hi Python!'
```

Placeholder : {}

```
1 '{0} Python!'.format('Hello')
```

```
'Hello Python!'
```

This is called a base string or template string

## 5. Functions and Methods

### 5.2. format method

- | You can control the output order by assigning the required integer value (index) within the placeholder. (By default, 0, 1, .. are assigned.)

```
1 'I like {} and {}'.format('Python', 'Java')
```

'I like Python and Java'

```
1 'I like {0} and {1}'.format('Python', 'Java')
```

'I like Python and Java'

```
1 'I like {1} and {0}'.format('Python', 'Java')
```

'I like Java and Python'

You can put index in placeholders with 0, 1, ... etc. Depending on the index, Python or Java is entered, respectively.

'I like {} and {}'.format('Python', 'Java')



'I like {0} and {1}'.format('Python', 'Java')



'I like {1} and {0}'.format('Python', 'Java')



Role of placeholders and indexes

## 5. Functions and Methods

### 5.2. format method

- | You can make various outputs by using the number of the placeholder, such as {0}, {1}, {2}.
- | A placeholder can be not only a string, but also an arbitrary data type such as an integer type like 100 or 200 or a real number type.
- | Let's take a look at the different output methods.

```
1 '{0}, {0}, {0}! Python'.format('Hello')
```

```
'Hello, Hello, Hello! Python'
```

```
1 '{0}, {0}, {0}! Python'.format('Hello', 'Hi')
```

```
'Hello, Hello, Hello! Python'
```

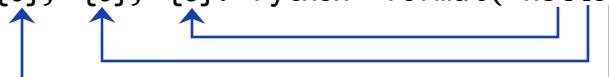
```
1 '{0} {1}, {0} {1}, {0} {1}!'.format('Hello', 'Python')
```

```
'Hello Python, Hello Python, Hello Python!'
```

```
1 '{0} {1}, {0} {1}, {0} {1}!'.format(100, 200)
```

```
'100 200, 100 200, 100 200!'
```

```
' {0}, {0}, {0}! Python'.format('Hello', 'hi')
```



You can select an output value by putting an index in the placeholder.

## 5. Functions and Methods

### 5.2. format method

In format, you can put not only string literals (fixed values such as integers, strings, and Boolean data types), but also variables or objects as follows.

```
1 greet = 'Hello'  
2 '{} World!'.format(greet)
```

'Hello World!'

```
1 name = 'David'  
2 print('My Name is {}!'.format(name))
```

My Name is David!

#### Line 1, 2

- Enter the string 'David' in name.
- If you enter a name using the format method, the name value is entered in {} (placeholder).

## 5. Functions and Methods

### 5.2. format method

**Ex** This is an example code that takes a user's name, age, and job as inputs, stores them in variables named name, age, and job, and outputs them using the format method.

```
1 name = input('Enter your name : ')
2 age = input('Enter your age : ')
3 job = input('Enter your job : ')
4
5 print('Your name is {}, age is {}, job is {}.'.format(name, age, job))
```

```
Enter your name : David
Enter your age : 23
Enter your job : Programmer
Your name is David, age is 23, job is Programmer.
```

- ▶ The meaning of {} inside the print function means to print the argument value of the format function where {} is present.
- ▶ Using the format method of a string, the name, age, and job are printed out in the {} fields, respectively.
- ▶ If the format method is not used as above, the print statement appears in a complex form. In this case, there are too many commas and quotation marks, so it makes the code hard to read and causes errors.

## 6. Product Set

### 6.1. The concept of a product set

- I Learn a product set, the advanced topic of sets discussed in the previous section.
  - ▶ Let's see how to easily create a product set using a function.
  - ▶ The product set A<sup>3</sup>, a concept used in mathematics, can be defined as A × A × A.
  - ▶ A × A × A has the following elements.

$$\begin{aligned}A \times A \times A &= (A \times A) \times A = \{(1, 1), (1, 3), (3, 1), (3, 3)\} \times \{(1, 3)\} \\&= \{((1, 1), 1), ((1, 1), 3), ((1, 3), 1), ((1, 3), 3), ((3, 1), 1), ((3, 1), 3), ((3, 3), 1), \\&\quad ((3, 3), 3)\}\end{aligned}$$

- ▶ The above operation can be implemented by repeating the product\_set function.

## 6. Product Set

### 6.1. The concept of a product set

- | Find the set of products using a function.
- | Review the product set discussed in the key concept of the previous unit once more.

```
1 def product_set(set1, set2) :  
2     res = set()  
3     for i in set1:  
4         for j in set2:  
5             res = res | {(i, j)}    # Product set using double for loop  
6     return res  
7  
8 def exp(input_set, exponent) : # A function that performs powers on input_set  
9     res = input_set      # res initialization  
10    for _ in range(exponent-1) : # Repeated for (exponent-1) to become a power  
11        res = product_set(res, input_set)  
12    return res  
13  
14 A = {1, 3}  
15 A3 = exp(A, 3) # Perform 3 powers on set A  
16 print(A3)
```

```
{((3, 1), 3), ((3, 3), 1), ((3, 1), 1), ((1, 3), 3), ((1, 1), 1), ((3, 3), 3), ((1, 1), 3), ((1, 3), 1)}
```

#### Line 15

- When an argument is passed in the exp function, exp calls product\_set to perform a multiplication set. That is, the third power is performed.

## 6. Product Set

### 6.2. The number of all possible outcomes when a die is rolled twice

| Use the product\_set function to count the number of cases in which a dice is rolled twice.

```
1 def product_set(set1, set2) :
2     res = set()
3     for i in set1:
4         for j in set2:
5             res = res | {(i, j)}      # Product set using double for loop
6     return res
7
8 cases = { 1, 2, 3, 4, 5, 6 }          # Integers from 1 to 6 on the die
9 cases_2times = product_set(cases, cases) # Get product_set for case
10 print(cases_2times)
```

```
{(3, 4), (4, 3), (3, 1), (5, 4), (4, 6), (5, 1), (2, 2), (1, 6), (2, 5), (1, 3), (6, 2), (6, 5), (4, 2), (4, 5), (3, 3),
(5, 6), (3, 6), (5, 3), (2, 4), (1, 2), (2, 1), (1, 5), (6, 1), (6, 4), (3, 2), (4, 1), (3, 5), (5, 2), (4, 4), (5, 5),
(1, 1), (1, 4), (2, 3), (2, 6), (6, 6), (6, 3)}
```

 Line 8,9

- The cases variable is assigned as a set type from 1 to 6.
- Multiply the two cases. Then, the combination of two numbers is assigned as a set type. All cases are returned without duplicates.

# | Pair programming



# Pair Programming Practice

## Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns “driving the mouse.” Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

## Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a “divide-and-conquer” strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

## Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.



# Pair Programming Practice



## | Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

## | Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide and conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

**Q1.**

Let the user input three integers a, b, and c. And print the average, maximum, and minimum values of these three numbers as follows. In this case, mean3(a, b, c), max3(a, b, c), min3(a, b, c) that takes three numbers as parameters and returns the average, maximum, and minimum values of these three numbers. Define and call each function.

## Output Example

```
Enter three numbers : 9 2 6
The average value of 9, 2, 6 is 5.666666666666667
The maximum value of 9, 2, 6 is 9
The minimum value of 9, 2, 6 is 2
```

Unit 18.

# Recursion Function Call

## Learning objectives

- ✓ Understand and be able to use recursive functions.
- ✓ Understand the concept of factorial and be able to implement factorial with iterative statements.
- ✓ Be able to implement factorial and Fibonacci functions through recursive calls rather than loops.
- ✓ Understand the shortcomings of recursive functions and be able to implement faster Fibonacci sequences through memoization.

## Lesson overview

- ✓ Learn how to solve the problem by calling a function itself.
- ✓ Learn how to implement factorial in a loop statement to check the efficiency of recursive functions.
- ✓ Learn how to create a recursive function and call this function to find the factorial and Fibonacci numbers.
- ✓ Learn the disadvantages and limitations of recursive calls, and how to implement the Fibonacci sequence in a better way through memoization.

## Concepts You Will Need to Know From Previous Units

- ✓ How to define and call user-defined functions using the def statement.
- ✓ How to pass values to functions efficiently using variable parameters, default parameters, and keyword parameters.
- ✓ How to return multiple values using the return statement of a function.

# Keywords

Recursive Call

Factorial

Fibonacci Sequence

memoization

## 1. Finding the Factorial with a Loop Statement

### 1.1. Definition of Factorial

- | Factorial is a term introduced from mathematics. It is expressed as  $n!$  for a natural number  $n$ .
  - ▶ The factorial is the product of all natural numbers from 1 to  $n$  when  $n$  is any natural number.

$$n! = \prod_{k=1}^n k = n \cdot (n - 1) \cdot (n - 2) \cdots \cdots 3 \cdot 2 \cdot 1$$

## 1. Finding the Factorial with a Loop Statement

### 1.2. Factorial and the for loop

| Define and use a factorial with simple code without a return statement. The for loop can be used as follows.

```
1 n = int(input('Enter a number : '))
2 fact = 1
3 for i in range(1, n+1):
4     fact = fact * i
5
6 print('{}! = {}'.format(n, fact))
```

```
Enter a number : 5
5! = 120
```

 Line 3, 4

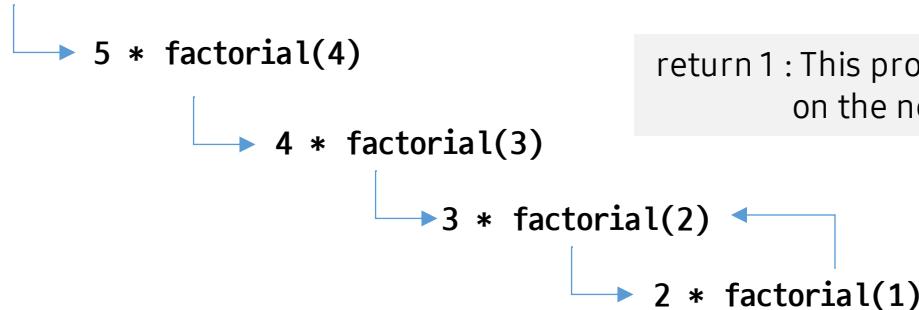
- From 1 to n, the fact variable is multiplied by accumulating by i.

## 2. Finding the Factorial with a Recursive Function

### 2.1. What is a recursive function?

- A recursion is a function that calls itself inside a function. Recursion is a very useful troubleshooting technique. It can intuitively and simply solve problems that are difficult to solve with procedural techniques.
- Factorial expression and definition :  $n! = n * (n - 1)!$ , if  $n \leq 1$  : return 1

```
def factorial(5):
    if n <= 1 :
        return 1
    else :
        return n * factorial(n-1)
```

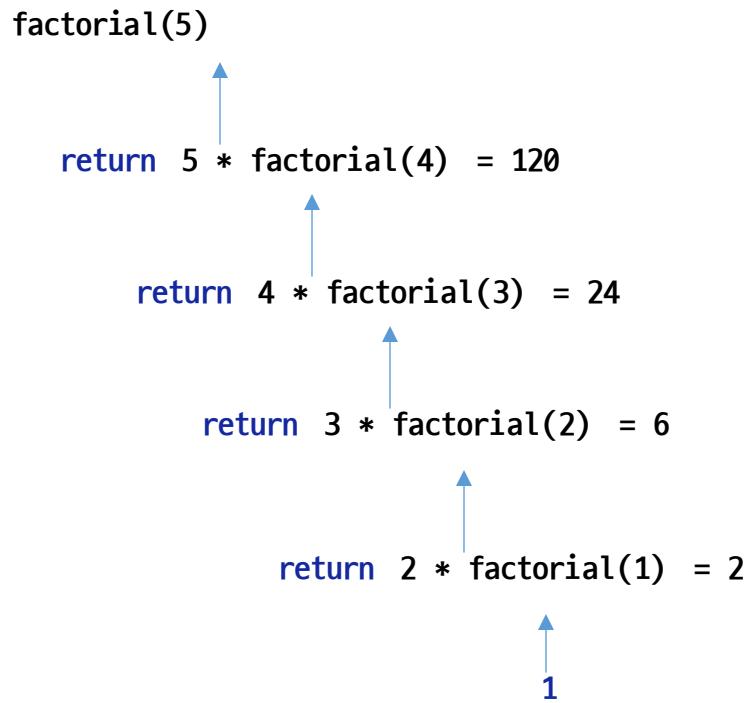


return 1 : This process is explained  
on the next page

## 2. Finding the Factorial with a Recursive Function

### 2.2. The flow of recursive function

The flow of the factorial recursive function is as follows. Factorial(1) returns 1. By factorial(2),  $2 * \text{factorial}(1) = 2 * 1$  is returned. It has a flow, and the result of the iteration return is 120.



## 2. Finding the Factorial with a Recursive Function

### 2.2. The flow of recursive function

 Focus In a recursive function, the termination condition is absolutely necessary.

- | By definition, the factorial is  $1! = 1$ ,  $n! = n \cdot (n-1)!$ . If this is implemented as a recursive function code, it can be expressed as follows.

```
1 def factorial(n):      # Implement recursive function of n!
2     if n <= 1 :          # Termination condition is required
3         return 1
4     else :
5         return n * factorial(n-1)    # n * (n-1)! implementation by definition
6
7 n = 5
8 print('{}! = {}'.format(n, factorial(n)))
```

$5! = 120$

- | The key point in a recursive function is when the program ends.
- | By definition,  $1! = 1$ , the value is defined, but there is no need to define a factorial for a value less than 1. If  $n$  is less than or equal to 1 in the if statement, 1 is returned.



# One More Step

## 1. Recursion and Divide-and-Conquer Algorithm

- | In recursive calling, when a function calls itself and the size of the problem is  $n$ , it is common to call it by breaking it into smaller problems than  $n$ .
- | This solution method is closely related to the divide-and-conquer algorithm.
- | The divide-and-conquer algorithm can be also called multi-branch recursion. The factorial example shown earlier is not a divide-and-conquer algorithm because it is a one-branch recursive call where the recursive call is performed only once.
- | When the size of the problem is  $n$ , divide the problem into two by size  $n/2$  and pass each as an argument to the recursive call, which will be a typical divide-and-conquer algorithm. In addition to splitting in half, the methods of applying recursive calls by splitting them into more than two small problems are all divide-and-conquer algorithms.
  - ▶ In a divide and conquer problem, there is a constraint that the sum of the divided problems must not be larger than the original problem.
- | If you learn data structures or algorithms later, you will learn a sorting algorithm called quick sort.
  - ▶ The sort algorithm can be implemented in a divide-and-conquer method using recursive calls.

## 3. Disadvantages of Recursive Functions and Memoization

### 3.1. Advantages and disadvantages of recursive functions and loop statements

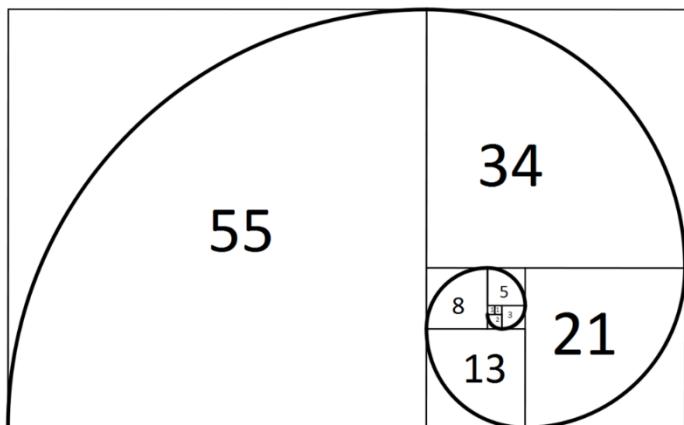
| Compare the pros and cons of recursive functions through the table below.

Recursive Functions		Loop Statements
Pros	Simple code	Fast
Cons	Use a lot of memory Slow	Complex code

### 3. Disadvantages of Recursive Functions and Memoization

#### 3.2. Definition of the Fibonacci sequence

A Fibonacci sequence is a sequence in which the first and second terms are 1, and all subsequent terms are the sum of the two immediately preceding terms.



Geometric pattern created by the Fibonacci sequence

<https://shoark7.github.io/assets/img/algoritm/fibonacci-logo.png>

- ▶ The first term  $F_0$  of the Fibonacci sequence is 0,  $F_1$  is 1, and  $F_n$  is defined as follows.

$$F_n = F_{n-1} + F_{n-2}$$

- ▶ According to this definition, the Fibonacci sequence is:
  - 0, 1, 1, 2, 3, 5, 8, 13, 21,...
- ▶ The Fibonacci sequence can create a geometric pattern as shown in the figure on the left.

### 3. Disadvantages of Recursive Functions and Memoization

#### 3.2. Definition of the Fibonacci sequence

- | Define the Fibonacci sequence using a recursive function.
  - ▶ It is a structure that calls the Fibonacci function twice inside.

```
1 def fibonacci(n):          # A recursive implementation of the Fibonacci function
2     if n <= 1:              # Termination condition of the Fibonacci function
3         return n
4     else:
5         return(fibonacci(n-1) + fibonacci(n-2)) #  $F_n = F_{n-1} + F_{n-2}$ 
6
7 nterms = int(input("How many Fibonacci numbers do you want?"))
8
9 # Cannot find Fibonacci number if it is negative
10 if nterms <= 0:
11     print("Error : Enter a positive number.")
12 else:
13     print("Fibonacci sequence: ", end=' ')
14     for i in range(nterms):
15         print(fibonacci(i), end=' ')
```

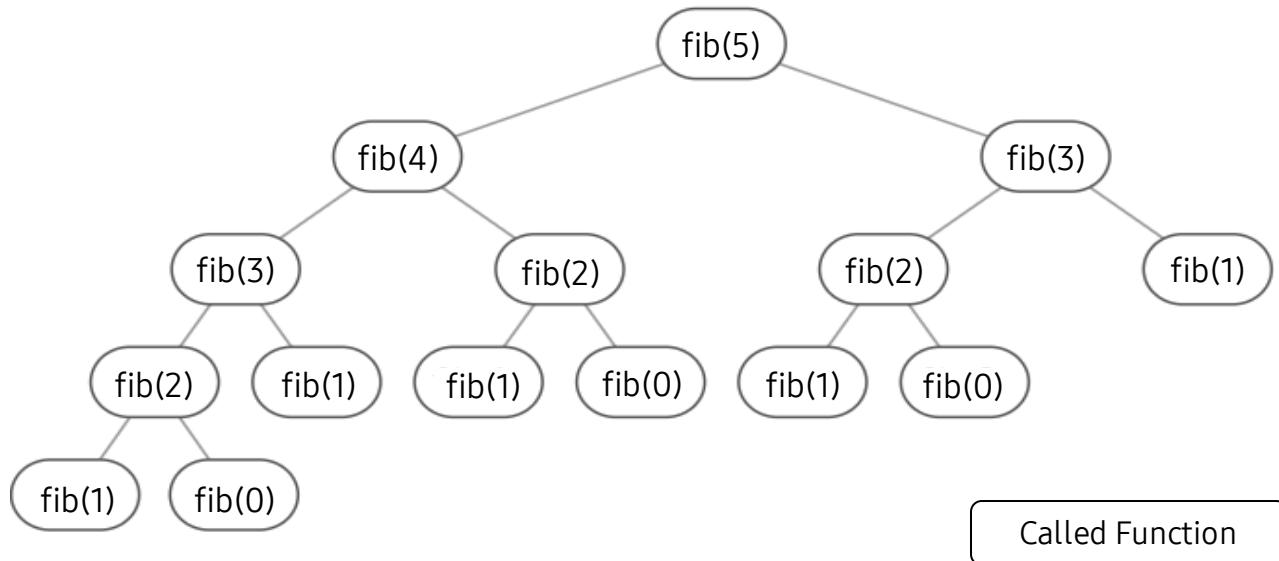
How many Fibonacci numbers do you want? 5

Fibonacci sequence: 0 1 1 2 3

### 3. Disadvantages of Recursive Functions and Memoization

#### 3.3. Fibonacci function execution process

- | Learn the execution process of Fibonacci function fib(5).
- | fib(5) calls fib(4) and fib(3), and fib(4) recursively calls fib(3) and fib(2). It results in a very large execution tree structure.



### 3. Disadvantages of Recursive Functions and Memoization

#### 3.4. Implementation of the Fibonacci function

| The implementation of the recursive function of the Fibonacci sequence is as follows.

```
1 def fibonacci(n):
2     if n == 0:
3         return 0
4     elif n == 1:
5         return 1
6     else:
7         return fibonacci(n-1) + fibonacci(n-2)
8
9
10 print(fibonacci(10))
```

55



Line 7

- The value is added as it is called recursively until the termination condition is met.

### 3. Disadvantages of Recursive Functions and Memoization

#### 3.5. Efficiency of implementing the Fibonacci function as a recursive function

- The biggest problem with the recursive function is that it is not efficient, because it repeats the calculation without meaning even if it is a value that has already been calculated. Let's look at the process of finding the 30th Fibonacci sequence as an example.
- We will use the module for time measurement. Timer() provided by the time measurement module measures the execution speed of the statement entered as the first argument. For this, refer to the following timeit module manual. (<https://docs.python.org/3/library/timeit.html>)

```
1 from timeit import *
2
3 def fibonacci_1(n):
4     if n == 0:
5         return 0
6     elif n == 1:
7         return 1
8     else:
9         return fibonacci_1(n-1) + fibonacci_1(n-2)
10
11 t3 = Timer("fibonacci_1(30)", "from __main__ import fibonacci_1")
12 print("fibonacci_1(30) * 20 times : ", t3.timeit(number = 20), "seconds")
```

fibonacci\_1(30) \* 20 times : 9.716442599999937 seconds

- It repeats the task of finding the 30th value 20 times and takes 9.7 seconds. (t3.timeit(number = 20) repeats 20 times)
- If you write a higher value than this, it will probably take a longer time.

### 3. Disadvantages of Recursive Functions and Memoization

#### 3.6. Fibonacci function and memoization



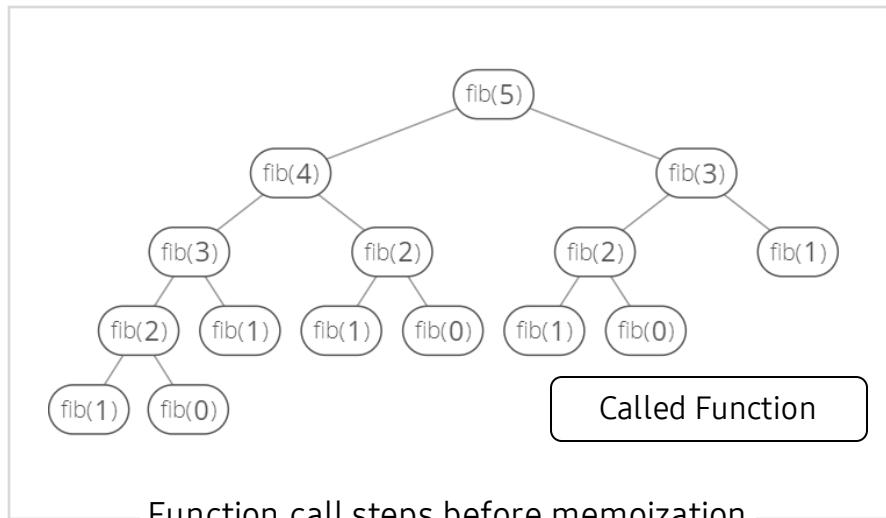
Memoization is a method of storing the results of small problems that are repeated over and over but the results do not change.

- | To solve the above problems, **memoization** in **Dynamic Programming (DP)** should be used.
- | Dynamic programming is one of the algorithm troubleshooting techniques to be dealt with during the algorithm course.
- | Key points :
  - ▶ In dynamic programming, you have to solve small problems to solve the big problems.
  - ▶ The results of those small problems is often constant.
  - ▶ Small problems that do not change the results occur repeatedly.
- | When a specific problem in which results are already recorded through **memoization** is repeated, unnecessary calculations can be skipped, and only recorded values can be retrieved very quickly.
- | A recursive function also has to solve small problems until a termination condition is met to solve a large problem, and there may be small problems that are repeated in the process.

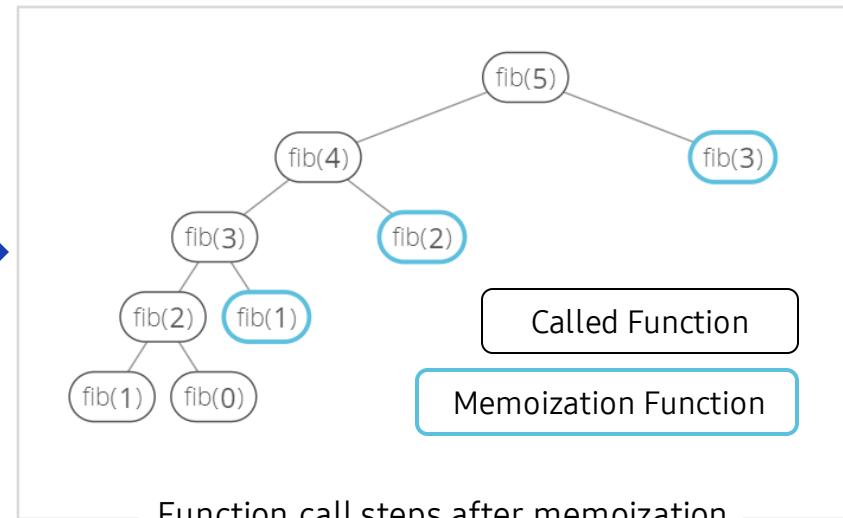
### 3. Disadvantages of Recursive Functions and Memoization

#### 3.6. Fibonacci function and memoization

- | This is an example of introducing memoization to the function fib(5) to find the Fibonacci number 5.
- | On the left is the case where 15 function calls were made without memoization. On the right, the number of function calls was reduced to 9 through memoization. Memoization can be implemented through a Python dictionary.



Function call steps before memoization



Function call steps after memoization

### 3. Disadvantages of Recursive Functions and Memoization

#### 3.7. The Fibonacci function and the principle of memoization

 In order to prevent double-counting, if the values of the terms are stored as elements in dic, the values calculated once do not need to be re-calculated.

The code that introduces memoization to the function to find the nth Fibonacci number is as follows.

```
1 from timeit import * # Import all classes and methods of timeit
2
3 dic = {0:0, 1:1} ←
4 def fibonacci_2(n):
5     if n in dic:
6         return dic[n] ←
7
8     dic[n] = fibonacci_2(n-1) + fibonacci_2(n-2) ←
9     return dic[n]
10
11 t2 = Timer("fibonacci_2(30)", "from __main__ import fibonacci_2")
12 print("fibonacci_2(30) * 20 times ", t2.timeit(number = 20), "seconds")
```

Declare a dictionary for memoization as a global variable, dic = {0:0, 1:1} Calling fibonacci\_2(1) returns 1, which is dic[1]

If the dictionary already has a computed result, it returns the result and does not make a recursive call.

If the dictionary has no computed result, the function makes a recursive call. The call result is stored in a dictionary.

```
fibonacci_2(30) * 20 times 3.32999981360743e-05 seconds
```

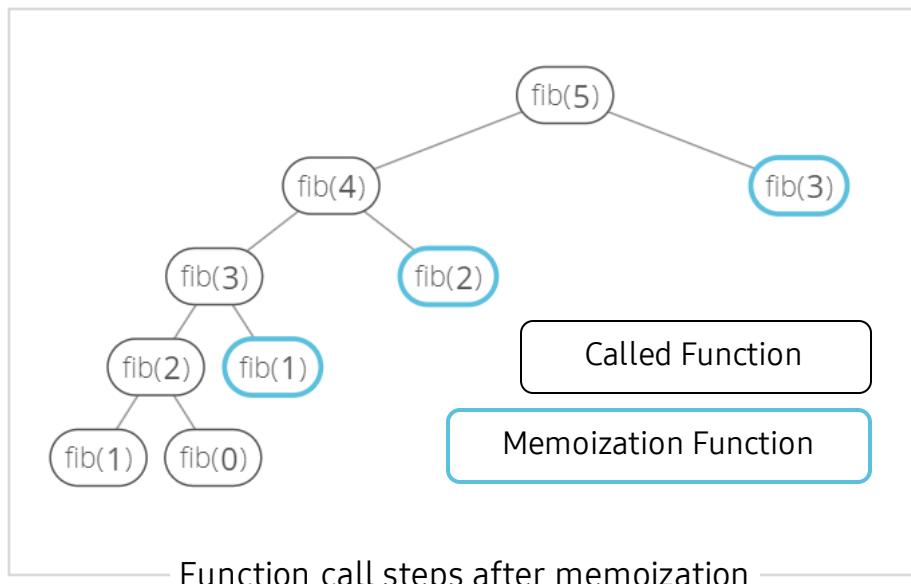
The code without memoization took 9.7 seconds, but the code with memoization took 0.000033 seconds.

If the efficiency of the recursive function is greatly reduced, it would be good to increase the efficiency by storing the result each time the recursive function is called. (memoization)

### 3. Disadvantages of Recursive Functions and Memoization

#### 3.7. The Fibonacci function and the principle of memoization

 Focus Learn how dictionaries are updated.



The initial dictionary state : dic = {0:0, 1:1}

Call fib(2) : return fib(1) + fib(0)

Call of fib(2) : return dic[0] + dic[1], which becomes 1  
dic = {0:0, 1:1, 2:1} state updated

Call fib(3) : return fib(2) + fib(1)

Call fib(3): return dic[1] + dic[2], which becomes 2  
dic = {0:0, 1:1, 2:1, 3:2} state updated

...

# | Paper coding

**Try to fully understand the basic concept before moving on to the next step.**

**Lack of understanding basic concepts will increase your burden in learning this course, which may make you fail the course.**

**It may be difficult now, but for successful completion of this course we suggest you to fully understand the concept and move on to the next step.**

## Q1.

Let's take a number n as input and find the sum from 1 to n. Write this function using a recursive function call.

Condition for Execution	Enter a number : 10 55
Time	5 minutes



Write the entire code and the expected output results in the note.

## Q2.

Python has the `**` operator, which indicates a square. However, let's take `x` and `n` as inputs without using an operator and use a recursive function to output `x` to the `n`th power. Let's try to output  $2^{10}$  by inputting `2` as the `x` value and `10` as the `n` value as follows.

Condition for Execution

```
Enter x : 2  
Enter n : 10  
1024
```

Time

5 minutes



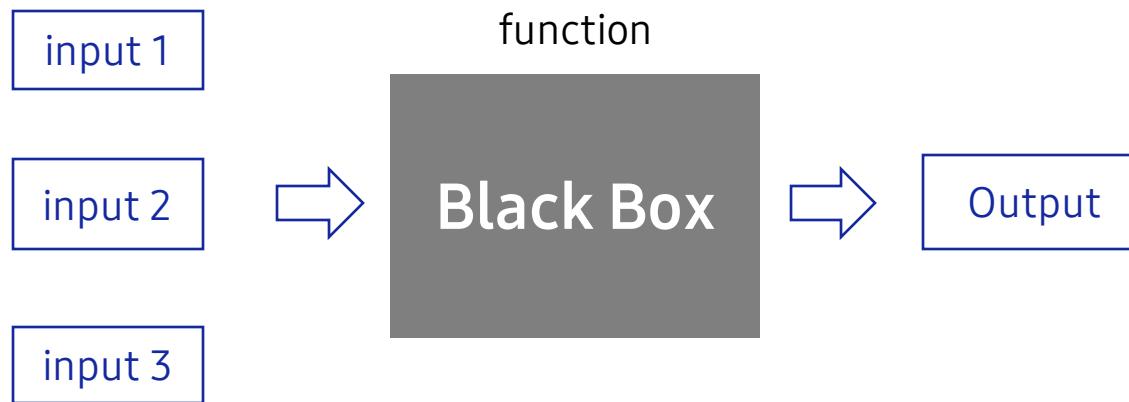
Write the entire code and the expected output results in the note.

| Let's code

## 1. Built-in Functions

### 1.1. Built-in functions expressed as black boxes

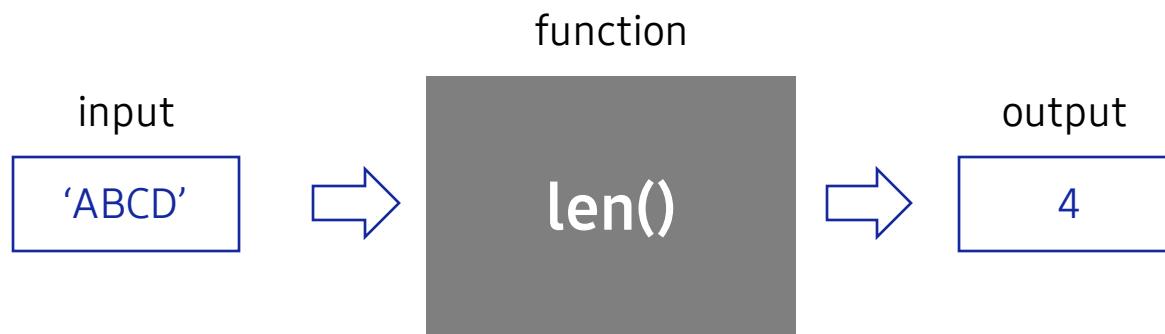
- | We have been using functions such as print and input so far. We will learn more about this.
- | In programming, a function is often referred to as a black box.
- | This is because the user who calls the function only needs to provide the correct input value and use the resulting output (result value).



## 1. Built-in Functions

### 1.1. Built-in functions expressed as black boxes

- | For example, since we know what the len function does for the string 'ABCD', we can use it without knowing the detailed operation process.
- | The print function and input function also belong to these functions.
- | Functions that are implemented by default in Python and provided are called built-in functions of Python.
- | Python provides over 100 built-in functions.



## 1. Built-in Functions

### 1.2. Python and built-in functions

Built-in functions in python				
abs	dict	help	min	setattr
all	dir	hex	next	slice
any	divmod	id	object	sorted
ascii	enumerate	input	oct	staticmethod
bin	eval	int	open	str
bool	exec	isinstance	ord	sum
bytearray	filter	issubclass	pow	super
bytes	float	iter	print	tuple
callable	format	len	property	type
chr	frozenset	list	range	vars
classmethod	getattr	locals	repr	zip
compile	globals	map	reversed	_import_
complex	hasattr	max	round	
delattr	hash	memoryview	set	

## 1. Built-in Functions

### 1.3. How to use built-in functions

| Code that uses built-in functions does not require a separate import.

```
1 abs(-100)           # A function that returns an absolute value
```

```
100
```

```
1 min(200, 100, 300, 400) # A function that returns the minimum value among multiple elements
```

```
100
```

```
1 max(200, 100, 300, 400) # A function that returns the maximum value among multiple elements
```

```
400
```

- ▶ The abs function takes an integer of -100 and returns an absolute value of 100.
- ▶ The min function returns the smallest value among integers with values of 200, 100, 300, and 400.
- ▶ The max function returns the largest value among integers with values of 200, 100, 300, and 400.

## 1. Built-in Functions

### 1.4. The return value of a built-in function

```
1 str1 = "FOO"          # Creates a string object in the format "FOO" or 'FOO'  
2 len(str1)            # Return the length of a string
```

3

```
1 type(str1)           # Return the type (type) of an object
```

str

```
1 id(str1)             # Return the ID value of an object
```

140168168956624

```
1 eval("100+200+300")    # Convert strings to numeric values and operators and evaluate them
```

600

- ▶ The id function returns the identity of the variable str1 where the string “FOO” is stored.
- ▶ The type function returns the data type of the corresponding variable.
- ▶ The len function returns the length of the string stored in the variable str1.
- ▶ The eval function receives a string, formulates the contents of the string, evaluates it, and returns the evaluated value.
- ▶ Evaluates the string “100+200+300” as a Python statement, interprets it as 100+200+300, and returns 600.

## 1. Built-in Functions

### 1.5. Built-in functions related to sorting

| The code sorts a list of strings and integers in ascending and descending order.

```
1 sorted("EABFD")      # Sort string
```

```
['A', 'B', 'D', 'E', 'F']
```

```
1 n_list = [200, 100, 300, 400, 50]
2 sorted(n_list)      # Sort the list in ascending order
```

```
[50, 100, 200, 300, 400]
```

```
1 sorted(n_list, reverse = True) # Sort the list in descending order
```

```
[400, 300, 200, 100, 50]
```

#### Line 1,2

- The sorted function receives a string and returns the characters that make up the string sorted in alphabetical order.
- The sorted function also has a function to sort and return a list.
- reverse = true to sort in descending order.

## 1. Built-in Functions

### 1.6. The id function and object-oriented programming

#### I The id function and object-oriented programming

- ▶ Python is an object-oriented programming language.
- ▶ The paradigm in which objects with various properties and functions compose a program is the core of object-oriented language.
- ▶ Python objects have a unique identity that distinguishes them from other objects. The id function returns the object's identity as an integer. Objects will be dealt with in detail in Unit 21.

```
1 a_str = "Hello Python!"  
2 id(a_str)
```

2549478283440

```
1 n = 300  
2 id(n)
```

2549477426192

## 1. Built-in Functions

### 1.7. type function

The type function returns the data type of an object. Learn the various data types in Python.

```
1 type(123)
```

int

```
1 type('Hello String!')
```

str

```
1 type(120.3)
```

float

```
1 type([100, 300, 600])
```

list

## 1. Built-in Functions

### 1.8. eval function

- | This is an eval function where the Python interpreter executes the string value entered as an argument and outputs the result as it is.
- | The eval function is very powerful, such as executing the script itself.
- | It should be used with caution because it can directly call system commands to obtain information on all files in the system where Python is running, or even delete files.

```
1 eval('10 + 20')      # The Python translator executes the sentence of 10 + 20
```

30

```
1 eval('(5 * 20) / 2') # The Python translator executes the sentence of (5 * 20) / 2
```

50.0

```
1 chr(65)            # The Unicode value 65 is the alphabet 'A', which the chr() function returns
```

'A'

```
1 ord('A')           # Return the Unicode value 65 of the alphabet 'A'
```

65

Unit 19.

# Lambda

## Learning objectives

- ✓ Utilize functionalized operations without def statements by using the lambda function.
- ✓ Filter elements of certain conditions from lists by using the filter function.
- ✓ Modify the value of elements from lists by using the map function.
- ✓ Compute the accumulated total of multiple data by using the reduce function.

## Learning overview

- ✓ Learn the utilities of the lambda function, and how to define and call the lambda function.
- ✓ Learn how to extract elements that satisfy a certain condition from a list by using the filter function.
- ✓ Learn how to modify values of units in a list by using the map function.
- ✓ Learn how to compute accumulated total from multiple data by using the reduce function.

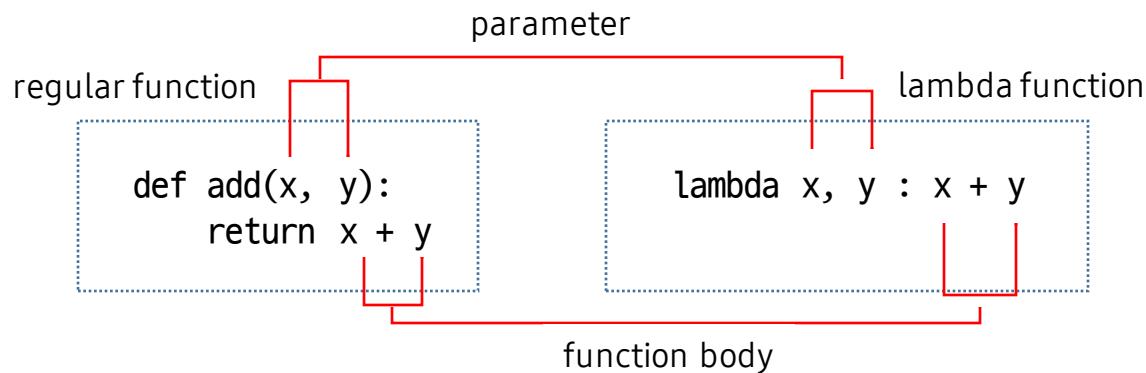
## Concepts You Will Need to Know From Previous Units

- ✓ Understanding about functions and being able to explain built-in functions and user-defined functions
- ✓ Using parameters to pass values to the called function
- ✓ Generate lists and use list-related methods

# 1. Lambda Expression

## 1.1. Lambda expression or lambda function syntax

- | Define a def statement to return the sum of the parameters x, y by writing a function called add as shown below.
- | Creating a function with def add:, writing the body, and assigning the return value with return statement may be cumbersome.
- | In Python, you can create a function for one-time use that encompasses all these steps. This is called a lambda function.
- | The lambda function is a function without a name and is also called a lambda expression. Due to such characteristic, it is also called an anonymous function.
- | Comparison of a regular function and a lambda function : A regular function consists of def keyword, name of the function, parameters, colon, and the body while a lambda function only has the functionalities of parameter and function.



## 1. Lambda Expression

### 1.1. Lambda expression or lambda function syntax

- Understand how to define a lambda function and pass an argument. Define `lambda` as a keyword and insert arguments in the bracket.

```
lambda [parameter 1, parameter 2, ...] : [expression]
```



```
(lambda [parameter 1, parameter 2, ...] : [expression])(argument 1, argument 2, ...)
```

- The syntax of the lambda function is simple as follows. The expression consists of [parameters] and colons(:).

```
lambda [parameters] : {expression}
```

## 1. Lambda Expression

### 1.2. Syntax of regular function

**Ex** Example of a code that returns a sum of integers using the add function.

```
1 # addition using add() function
2 def add(x, y):
3     return x + y
4
5 print('sum of 100 and 200 :', add(100, 200))
```

sum of 100 and 200 : 300

Although the function executes a simple operation that adds two numbers, the code must follow all syntax including def keyword, function name, function body, and return.

## 1. Lambda Expression

### 1.3. Syntax of regular function and lambda function

- | Use a lambda function to execute what the add function does without defining the function.

```
1 # addition using the lambda function
2 add = lambda x, y: x + y
3 print('sum of 100 and 200 :', add(100, 200))
```

sum of 100 and 200 : 300

- | Define lambda function with the keyword lambda. The expression receives x, y as parameters and returns x + y.
- | The function adds two arguments without a head like def add(x, y):
- | Now substitute the add variable with the anonymous function lambda. This executes an addition operator with 100, 200 as parameter inputs.
- | This verifies that the lambda function successfully performs tasks of functions carried out by regular functions.

## 1. Lambda Expression

### 1.3. Syntax of a regular function and a lambda function

| Lambda function operates well even with such simple statement. Even the add variable is not necessary.

```
1 print('sum of 100 and 200 :', (lambda x, y: x + y)(100, 200))
```

```
sum of 100 and 200 : 300
```

#### Line1

- Lambda function executes  $x + y$  and returns the values.
- Passes arguments 100, 200 to the parameters x and y.

## 1. Lambda Expression

### 1.3. Syntax of a regular function and a lambda function

- | The lambda function enables simple coding.

```
1 a = [1, 2, 3, 4, 5, 6, 7]
2 square_a = list(map(lambda x: x**2, a))
3 print(square_a)
```

```
[1, 4, 9, 16, 25, 36, 49]
```

- | The map function, which you will learn in the next chapter, is used like the lambda expression. It shows great performance with simple coding.
- | The code above is a lambda expression and a map function that return x square.

## 2. filter, map, reduce

### 2.1. Syntax of filter function

- | Python offers functions of numerous forms. filter function is one of them.
- | filter function receives iterable elements and returns only those that are True in a bundle.
- | The syntax of the filter function is as follows.

```
filter((To be applied function, {iterable object}))
```

## 2. filter, map, reduce

### 2.2. Application example of a filter function

**Ex** There is a list that stores ages of many people. Build a code that extracts ages of adults above 19.

```
1 # return True for values over 19, and False for those that are not.
2 def adult_func(n):
3     if n >= 19:
4         return True
5     else:
6         return False
7
8 ages = [34, 39, 20, 18, 13, 54]
9 print('adults list :')
10 for a in filter(adult_func, ages): # filter ages by using filter () function
11     print(a, end = ' ')
```

Adults list :  
34 39 20 54



#### Line 1-6

- adult\_func function returns True if the value of the variable n is above 19 and False if the value is below 19. The variable n is a parameter that signifies age.

## 2. filter, map, reduce

### 2.2. Application example of a filter function

Ex There is a list that stores ages of many people. Build a code that extracts ages of adults above 19.

```
1 # return True for values over 19, and False for those that are not.
2 def adult_func(n):
3     if n >= 19:
4         return True
5     else:
6         return False
7
8 ages = [34, 39, 20, 18, 13, 54]
9 print('adults list :')
10 for a in filter(adult_func, ages): # filter ages by using filter () function
11     print(a, end = ' ')
```

Adults list :  
34 39 20 54

#### Line 8 - 11

- ages list contains random values of age. The code put adult\_func function and iterable list variable ages as arguments.
- The filter function passes units of ages to adult\_func function one by one and returns a list that only contains units that returned True.
- You can see that 18, 13 from the ages variable are not printed.

## 2. filter, map, reduce

### 2.3. Characteristics of filter function adult\_func

| The adult\_func function has following characteristics.

1. The algorithm is relatively simple.
2. It does not need an extra inner variable.
3. You do not need to reuse it once it filters adults list.

| In such case where the functionality is simple and reuse is not necessary, lambda function is appropriate.

## 2. filter, map, reduce

### 2.3. Transform adult\_func filter function to lambda function

- | You can simplify the adult\_func function by transforming it to an anonymous function by using the lambda keyword.
- | Using the lambda function as follows, the code became much simpler and shorter.

```
1 ages = [34, 39, 20, 18, 13, 54]
2 print('adults list :')
3 for a in filter(lambda x: x >= 19, ages): # filter ages using filter () function
4     print(a, end = ' ')
```

adults list :  
34 39 20 54

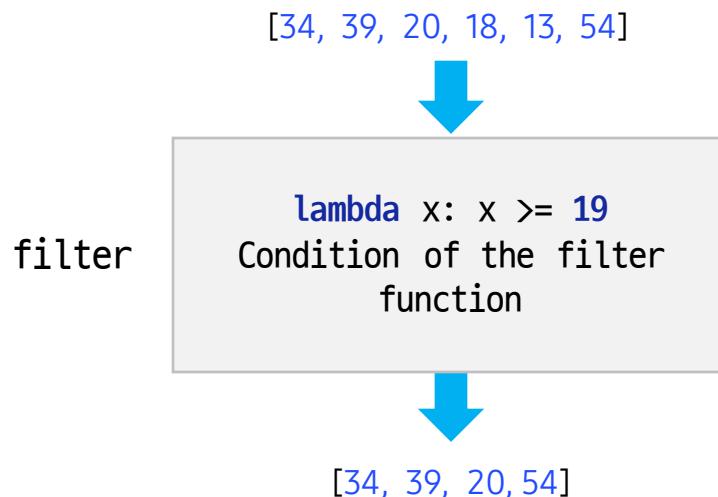


- Line 3
- It returns True for x equal to or over 19, and only prints values of ages that are equal to or over 19.
  - The filter condition is lambda x: x>= 19.

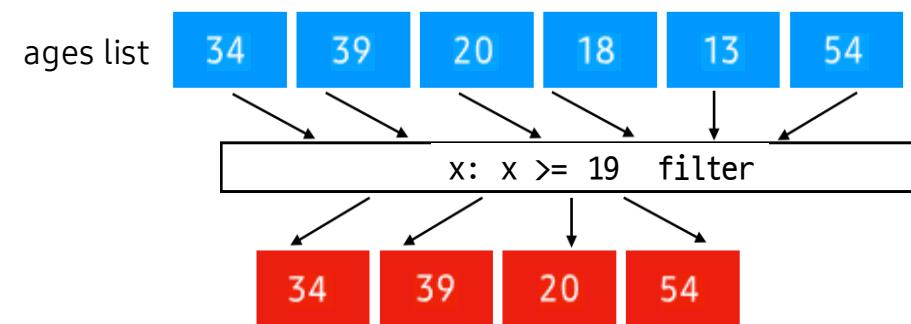
## 2. filter, map, reduce

### 2.4. Operating procedure of lambda function used in filter function

- | The below diagram describes the operating procedure.
- | The code is much simpler and shorter because it used a one-line lambda function instead of adult\_func function.



The behavior of the filter function and the role of the lambda function



Iterable objects and returned values of the filter function

## 2. filter, map, reduce

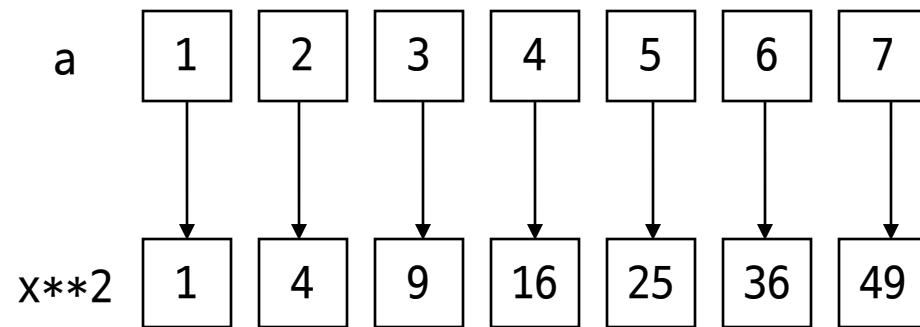
### 2.5. Application of the lambda function

- | Observe the behavior of the lambda function which acts as a filter.
- | A filter function with a specific condition filters units from the input list [34, 39, 20, 18, 13, 54]. It passes units equal to or above 19, and filters units below 19.
- | This lambda function returns True or False according to the result discerned by the condition expression  $x \geq 19$  on  $x$  of lambda  $x$ :
- | The filter function receives filter condition function and iterable object as parameters.

## 2. filter, map, reduce

### 2.6. map function

- | Python offers a built-in function called map. This function returns an iterable datatype by applying mapping function on each unit of the iterable datatype.
- | Consider a case where you return a list of [1, 4, 9, 16, 25, 36, 49] by applying square operation on each value of a list called a, which is [1, 2, 3, 4, 5, 6, 7].



## 2. filter, map, reduce

### 2.7. Computing square value of all units in the list

**Ex** If the elements of list a are as follows, add the square value of units of a to the list.

- | Solve the problem with methods we already know.
- | Execute square operation on each unit x of list a by using the for statement. Then, add this value to a list called square\_a by using append method.

```
1 a = [1, 2, 3, 4, 5, 6, 7]
2 square_a = []
3 for n in a:
4     square_a.append(n**2) # add n square to list square_a
5 print(square_a)
```

[1, 4, 9, 16, 25, 36, 49]

 Line 3, 4

- Square each units of a and add to list square\_a

## 2. filter, map, reduce

### 2.8. Behavior of map function

- | You can easily modify the previous code by using map function and square function. The syntax of map function is as follows. The iterable objects are objects such as list or tuple, which will be discussed later.

```
map(function_to_be_applied, iterable_object, ...)
```

- | The map function takes two or more arguments. The first is a mapping function and the second is an iterable object such as a list that will be inserted to the mapping function.
- | By using the map function, each value in the iterable object like a list is applied on the mapping function and returns list [1, 4, 9, 16, 25, 36, 49] which has square values of [1, 2, 3, 4, 5, 6, 7]

```
1 def square(x):
2     return x ** 2
3
4 a = [1, 2, 3, 4, 5, 6, 7]
5 # apply the return value of square function to each term of a
6 square_a = list(map(square, a))
7 print(square_a)
```

```
[1, 4, 9, 16, 25, 36, 49]
```

## 2. filter, map, reduce

### 2.9. Combination of map function and lambda function

- I Expressing a simple code using the map function and the lambda function
- I You can simplify the code by using the map function and lambda function. The lambda function operates as one-time function. It is efficient than externally writing a def statement to define a function.

```
1 a = [1, 2, 3, 4, 5, 6, 7]
2 square_a = list(map(lambda x: x**2, a))
3 print(square_a)
```

```
[1, 4, 9, 16, 25, 36, 49]
```

#### Line2

- In the map function there are the lambda function which returns x square and the list a.
- Each square value of elements of a is stored in square\_a in list type.

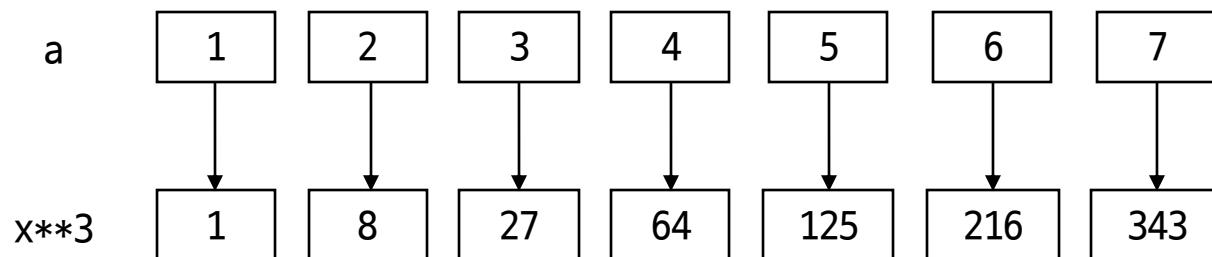
## 2. filter, map, reduce

### 2.9. Combination of map function and lambda function

| You can transform units of the list into a simple expression by using the map function and lambda function.

```
1 a = [1, 2, 3, 4, 5, 6, 7]
2 cubic_a = list(map(lambda x: x**3, a))
3 print(cubic_a)
```

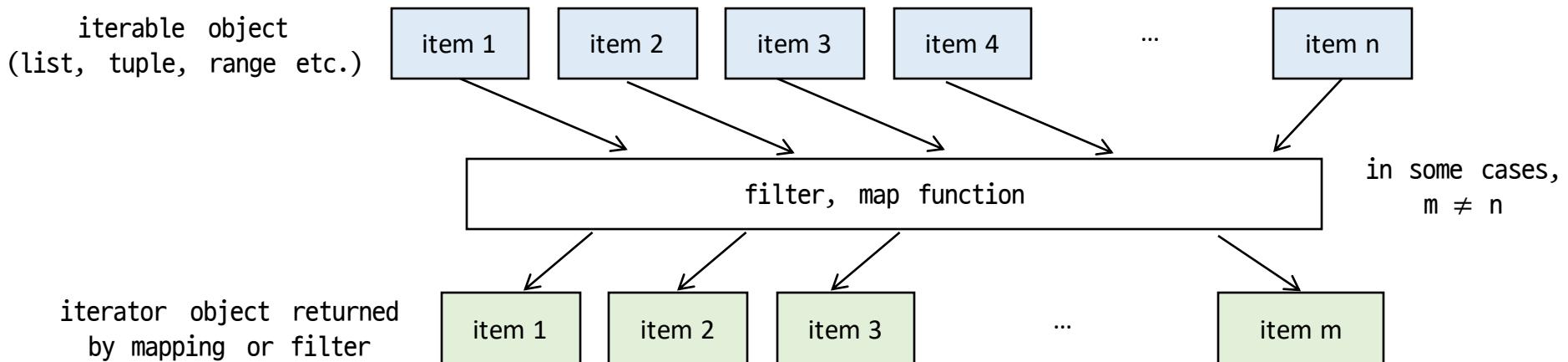
[1, 8, 27, 64, 125, 216, 343]



## 2. filter, map, reduce

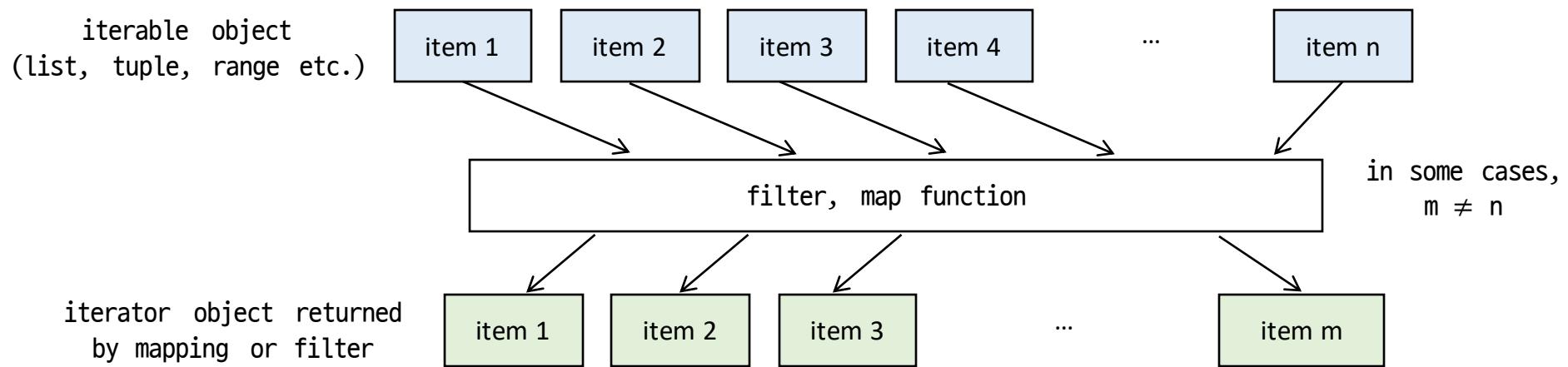
### 2.10. iterable data types and filter, map function

- | Python offers built-in functions called filter and map. Apply mapping function on each element of an iterable data type.
- | These functions return iterable objects. This object can be modified into a list type by using the list function.



## 2. filter, map, reduce

### 2.10. Iterable data types and filter, map function



- Each item of iterable data types such as list, tuple, range etc. are modified by mapping functions like filter or map and returned. Each value is an iterator object, and these objects can be modified back into an iterable data type by using data type modifier function such as list.

## 2. filter, map, reduce

### 2.10. Iterable data types and filter, map function

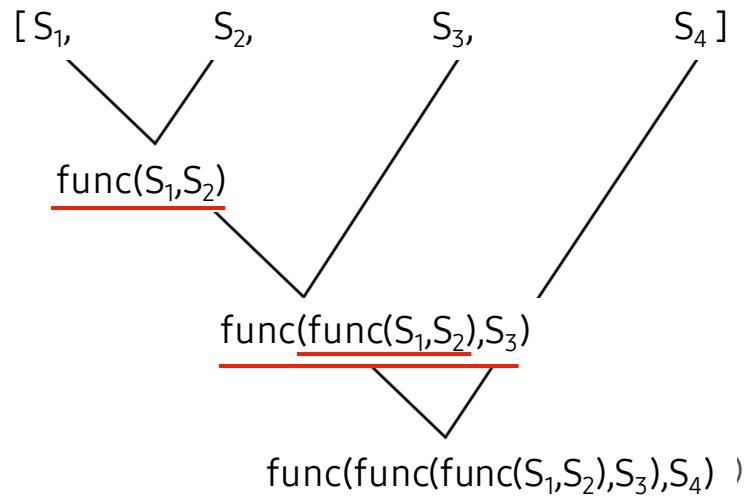
 Focus Using a lambda function on the filter and map accelerates execution by reducing function call overhead.

- | Why do you filter and map each item of iterable data types such as list, tuple, range etc. by using functions such as filter and map?
- | Such task can be done by function call, but too much function call produces has a shortcoming. During function call, it saves the state before the call, and after the call, it restores the previous state, and frequent execution of this process is a problem.
- | This slows down the program operating speed.

## 2. filter, map, reduce

### 2.11. reduce function

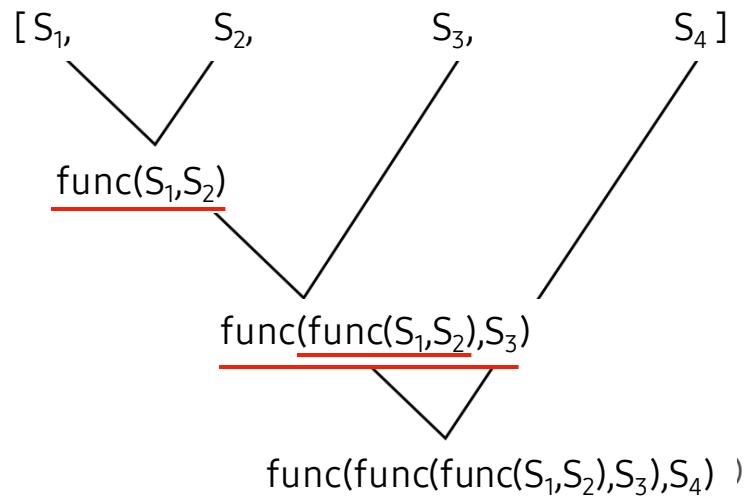
- | Reduce function is included in a module called `functools`. This returns a single value by executing an operation with a given function on items of an iterable object.
- | Examine the following exercise. With a list called `seq = []`, the `reduce(func, seq)` function executes the following procedure.



## 2. filter, map, reduce

### 2.11. reduce function

- | The first and second item of list seq, s1 and s2, are given as inputs of func function, which acts as a parameter of reduce function. Then a certain value is returned to func( , ), which is then given as input for the func function with the next item of seq, s3.
- | In expression, it is  $\text{func}(\text{func}( , ), )$ . This process is recursively iterated until the last item is given as an input.
- | In the end it returns a single value.



## 2. filter, map, reduce

### 2.12. Example of reduce function

**Ex** Example using the reduce function

```
1 from functools import reduce  
2  
3 a = [1, 2, 3, 4]  
4 n = reduce(lambda x, y: x + y, a)  
5 print(n)
```

10

- | 1, 2, 3, 4 are given as items of list variable a.
- | First, 3 is returned from lambda function lambda x, y: x+y when the first two values 1, 2 are given as inputs.
- | Then, the return value 3, with the third item 3, is given an input value for the lambda function. It then returns 6, the sum of the two numbers, then this value 6 and the last item 4 are given as input for the lambda function, thus returning the final value 10.

$((1 + 2) + 3) + 4 \Rightarrow \text{returns } 10$

# | Paper coding

**Try to fully understand the basic concept before moving on to the next step.**

**Lack of understanding basic concepts will increase your burden in learning this course, which may make you fail the course.**

**It may be difficult now, but for successful completion of this course we suggest you to fully understand the concept and move on to the next step.**

**Q1.**

There is a list with integer element values called n\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Return even\_list which only contains items of even number values from n\_list by using the filter function and the lambda function.

The returned even\_list = [2, 4, 6, 8, 10]

Conditions for Execution	even_list = [2, 4, 6, 8, 10]
Time	5 Minutes

Create an empty list named even\_list and add even value items by the append method. Use the for statement and the filter function. Use lambda function inside the filter function.

**Q2.**

There is a list with integer unit values called n\_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]. Return even\_list which only contains items of even number values from n\_list by using a lambda function. This time, do not use for statement, and instead use list function.

Returned even\_list = [2, 4, 6, 8, 10]

Conditions for Execution	even_list = [2, 4, 6, 8, 10]
Time	5 Minutes

Modify objects returned by the filter function to list objects by the list function, and assign them to even\_list.



Write the entire code and the expected output results in the note.

## Q3.

Write a map function that converts a\_list which contains lowercase alphabets like ['a', 'b', 'c', 'd'] to a upper\_a\_list which contains upper case alphabets like ['A', 'B', 'C', 'D'].

Also, define a function named to\_upper that receives lowercase letters as parameters and returns uppercase letters, and convert those lowercase letters.

Conditions  
for Execution

```
upper_a_list = ['A', 'B', 'C', 'D']
```

Time

5 Minutes



Write the entire code and expected output result in the note.

## Q4.

Compute the sum of integers from 1 to 100 by using reduce function and lambda expression inside it. Use range (1, 101) as an input.

Condition for Execution

Sum of 1 to 100 : 5050

Time

5 Minutes

| Let's code

# 1. Exception Handling

## 1.1. Importance of exception handling

- | A programmer must code in preparation of exceptions. A code below is in the danger of termination due to an incorrect input. Enclose the code with try ~ except statement.

```
1 try:  
2     a, b = input('Enter two numbers : ').split()  
3     result = int(a) / int(b)  
4     print('{} / {} = {}'.format(a, b, result))  
5 except :  
6     print('Check if the numbers are correct.')
```

Enter two numbers : 10 2  
10/2 = 5.0

It behaves as intended for correct inputs.

```
1 try:  
2     a, b = input('Enter two numbers : ').split()  
3     result = int(a) / int(b)  
4     print('{} / {} = {}'.format(a, b, result))  
5 except :  
6     print('Check if the numbers are correct.')
```

Enter two numbers : 100 two  
Check if the numbers are correct.

except statement processes the code even though the user entered a string 'two'.

Enter two numbers : 100 two  
Check if the numbers are correct.

If the user enter 100 and 0, the code tries 100/0. The except statement handles the exception.

# 1. Exception Handling

## 1.2. Various Exceptions

- | What kinds of exceptions are there?
- | The table shows many cases of exceptions.
- | These exceptions will be studied later.

BaseException	ProcessLookupError	UnboundLocalError
Exception	TimeoutErrorReferenceError	OSError
ArithmaticError	RuntimeError	BlockingIOError
FloatingPointError	NotImplementedError	ChildProcessError
OverflowError	RecursionError	ConnectionError
ZeroDivisionError	StopIteration	BrokenPipeError
AssertionError	StopAsyncIteration	ConnectionAbortedError
AttributeError	SyntaxError	ConnectionRefusedError
BufferError	IndentationError	ConnectionResetError
EOFError	TabError	FileExistsError
ImportError	SystemError	FileNotFoundError
ModuleNotFoundError	TypeError	InterruptedError
LookupError	ValueError	IsADirectoryError
IndexError	UnicodeError	NotADirectoryError
KeyError	UnicodeDecodeError	PermissionError
MemoryError	UnicodeEncodeError	FutureWarning
NameError	UnicodeTranslateError	ImportWarning

## 1. Exception Handling

### 1.2. Various Exceptions

| You can examine the type of exception with the following code. print('error :', e) statement prints the exception.

```
1 try:  
2     b = 2 / 0  
3     a = 1 + 'hundred'  
4 except Exception as e:  
5     print('error :', e) ← object 2 contains the information of the exception.  
  
error : division by zero
```

| The above code tells that the division by zero exception occurred.

 One More Step

## 1. Exception Handling

- | A sophisticated exception handling as the following is possible by using try - except – else – finally.
- | This code contains a specific output statement for the specific exception ZeroDivisionError. It has a statement that operates by the finally, regardless of the exception.

```
1 def divide(x, y):
2     try:
3         result = x / y
4     except ZeroDivisionError:
5         print('error by zero division')
6     else:
7         print('result :', result)
8     finally:
9         print('execution complete')
10
11 print('divide(100, 2) function call :')
12 divide(100, 2)
13 print('divide(100, 0) function call :')
14 divide(100, 0)
```

```
divide (100, 2) function call :
result : 50.0
execution complete
divide (100, 0) function call :
error by zero division
execution complete
```

 One More Step

## 1. Exception Handling

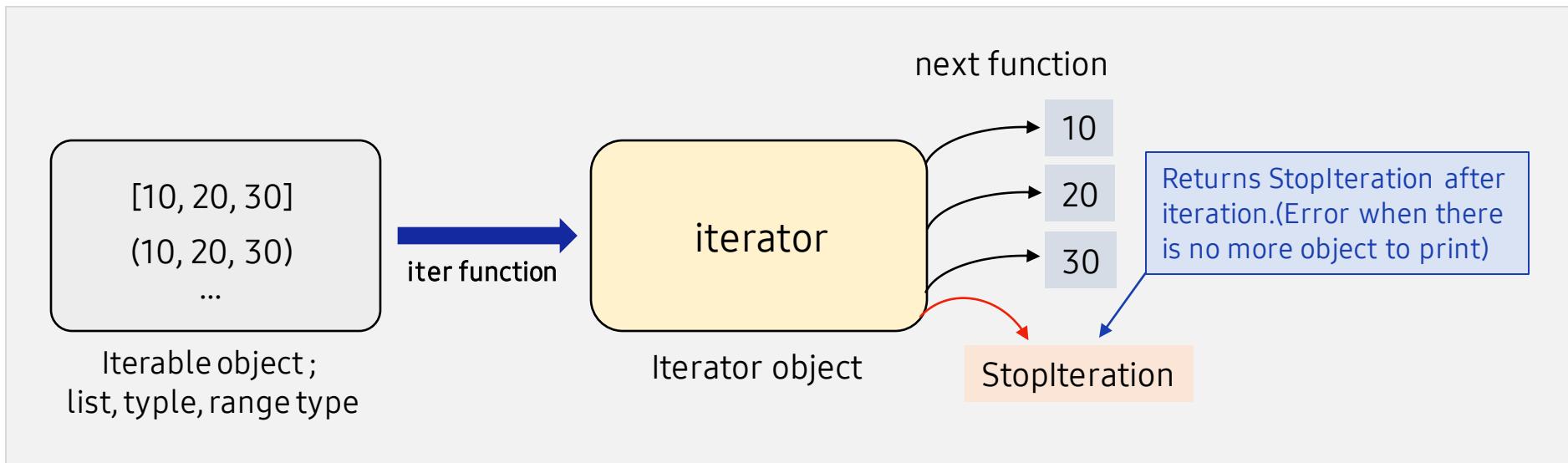
| Examine specific functions of try – except –else –finally statements.

name	description
try:	Encloses the statement of exception with try clause. If the exception doesn't occur because the try clause statement was processed, it jumps past except. If exception does occur, it checks the error and passes onto the matching part of except.
except:	The block that operates in the case of exception. It contains descriptions of how to handle the exception.
else:	This block operates when there is no exception.
finally:	This block always operates regardless of the exception.

## 2. Iterator Object

### 2.1. Iterator is an object that behaves behind an iterating statement of Python

- | Study iterator, one of Python's high-level functions.
- | Iterator object is an object that uses data consecutively from a data structure with one or more items.





# One More Step

## 2.1. Iterator Object

- | Python's [10, 20, 30] list is called repeatedly in iterations such as for i in [10, 20 ,30]: .
- | How do you know if the for statement reached the end of list [10, 20, 30]?
- | This is possible because StopIteration object is hidden and returned at the end of [10, 20, 30] like [10, 20, 30, StopIteration]
- | StopIteration object notifies the user that there is no more item to return in the iteration loop.
- | This object is created by the built-in function next and the special method \_\_next\_\_ .
- | Refer to the Python development document for more info.

## 2. Iterator Object

### 2.2. Built-in function for iterable objects

- | Many built-in functions can be applied to the iterable objects in Python.
- | Functions such as min or max receive iterable objects as input and returns the minimum and maximum values. Other high-level built-in functions such as all, any, ascii, bool, filter, iter are also offered.

## 2. Iterator Object

### 2.2. Iterator and iter, next function

- | list data type with items of [10, 20, 30] has been converted to a data type called list\_iterator by the iter function.
- | The iter function receives iterator data as input and returns iterators.
- | Generally you use iter function with the next function.
- | The next function receives an iterator as input and returns the item the iterator must print next.

```
1 iter_a = iter([10, 20, 30])
2 next(iter_a)
```

10

```
1 next(iter_a)
```

20

## 💡 One More Step

### 2.2. Iterator and iter, next function

- | For list a=[10, 20, 30], if you call next(a) to view the items of the list, following error occurs. Therefore, a list object cannot return items consecutively.
- | Python's iter function makes this list a into an iter\_a object.
- | This iter\_a repeatedly returns items by next(iter\_a)
- | next function consecutively calls the items of the iterable object.

```
1 a = [10, 20, 30]
2 next(a)
```

```
-----  
TypeError                                         Traceback (most recent call last)
<ipython-input-4-e4a568985220> in <module>
      1 a = [10, 20, 30]
----> 2 next(a)

TypeError: 'list' object is not an iterator
```

```
1 iter_a = iter([10, 20, 30])
2 next(iter_a)
```

10

```
1 next(iter_a) # Returns the next time from the iterator
```

20

## 2. Iterator Object

### 2.3. Non-iterable data type and usage of iter function



#### TypeError

```
1 n = 100 # Non-iterable data type int
2 n_iter = iter(n)
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-18-59ae9a7950d8> in <module>  
      1 n = 100 # Non-iterable data type int  
----> 2 n_iter = iter(n)
```

**TypeError:** 'int' object is not iterable

- ▶ If you pass int type variable with the value 100 as argument for the iter function, TypeError occurs.
- ▶ Integer data type int cannot be converted to an iterator object because it is not an iterable data type.

## 2. Iterator Object

### 2.4. next function and extracting units from iterator object

```
1 lst = [10, 20, 30]  
2 l_iter = iter(lst) # Converts list type object to an iterator  
3 type(l_iter)
```

list\_iterator

```
1 next(l_iter)
```

10

```
1 next(l_iter)
```

20

```
1 next(l_iter)
```

30

- | First create the list object lst.
- | Apply this object to the iter function and the variable l\_iter. You can see that the object is of list\_iterator data type.
- | Iterator type can extract items one by one by using the next function.
- | As a result, you can consecutively obtain items of the lst object 10, 20, 30.

## 2. Iterator Object

### 2.5. StopIteration Exception



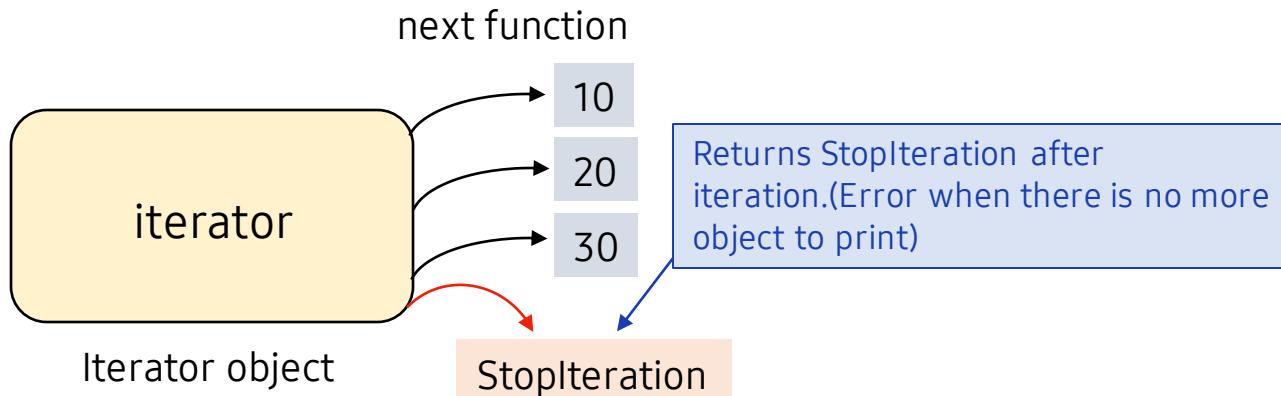
#### Return StopIteration exception

```
1 next(l_iter)
```

```
StopIteration                                     Traceback (most recent call last)
<ipython-input-25-43aece5a3964> in <module>
      ----> 1 next(l_iter)
```

StopIteration:

- ▶ If you call a next function for an iterator object that returned all items, it generates a StopIteration exception.



## 2. Iterator Object

### 2.6. Using special method `__next__`

- | A method in Python class that carries out specific tasks is called a special method or magic method.
- | A special method has double under bar in front and back of the name. (e.g.) `__next__()`
- | It is used to re-define an operator or a function, which has been defined in Python.
- | As before, it generates the variable named `l_iter` and converts the list `lst` into an iterator object to act as an input.
- | '`object.__next__`' behaves like the `next` function. It returns only when there is an object for return.

```
1 lst = [10, 20, 30]
```

```
1 l_iter = iter(lst) # converts list type object into an iterator
```

```
1 l_iter.__next__()
```

Call special method `__next__`

10

```
1 l_iter.__next__()
```

20

```
1 l_iter.__next__()
```

30

## 💡 One More Step

### 2.6. Using special method `__next__`

- | Operating `10 + 20` gives 30.
- | This addition operator is the result of an operation executed by the special method `__add__` on the integer object 10.
- | Thus, the code `10 + 20` is equal to `(10).__add__(20)`.
- | Python has many methods that have names written as `__XXX__`. These are special methods.
- | More about class and method in Unit 21.

1	<code>10 + 20</code>
30	

1	<code>(10).__add__(20)</code>
30	

## 2. Iterator Object

### 2.7. Code that verifies if the object is iterable

```
1 try:  
2     l = [10, 20, 30]  
3     iterator = iter(l)  
4 except TypeError:  
5     print('list is not an iterable object.')  
6 else:  
7     print('list is an iterable object.')
```

list is an iterable object.

#### Line 1 - 3

- Using iter (l), check if the list object l can be converted into an iterator in try : clause.
- Exception does not occur.
- The message 'is an iterable object' pops up.

## 2. Iterator Object

### 2.7. Code that verifies if the object is iterable

```
1 # verifies if the tuple is an iterable object
2 try:
3     t = ('Hong gil dong', 22, 79.7)
4     iterator = iter(t)
5 except TypeError:
6     print('tuple is not an iterable object.')
7 else:
8     print('tuple is an iterable object.')
```

tuple is an iterable object.

#### Line 1 - 3

- Using iter (l), check if the tuple object l can be converted into an iterator in try : clause.
- Exception does not occur.
- The message 'is an iterable object' pops up.

## 2. Iterator Object

### 2.7. Code that verifies if the object is iterable

```
1 # verifies if the tuple is an iterable object
2 try:
3     n = 100
4     iterator = iter(n)
5 except TypeError:
6     print('n is not an iterable object.')
7 else:
8     print('n is an iterable object.'
```

n is not an iterable object.

#### Line 2 - 4

- Through `iter(n)` in the `try : clause`, check if the integer object `n` can be converted into an iterator
- Exception occurs. `except:` clause processes the exception.
- Message 'is not an iterable object' pops up because `TypeError` exception occurred.

## 2. Iterator Object

### 2.8. Code that accesses items of range\_iterator type object

| You can access various items of range\_iterator type object by using the next function.

```
1 type(range(3))
```

```
range
```

```
1 r_iter = iter(range(3)) # converts range type object into an iterator
2 type(r_iter)
```

```
range_iterator
```

```
1 next(r_iter) # you can use next() function because it is an iterator
```

```
0
```

```
1 next(r_iter)
```

```
1
```

```
1 next(r_iter)
```

```
2
```

## 2. Iterator Object

### 2.9. Built-in function for iterable objects : all

- | all function returns True only when all returnable items are True.
- | If there is a single 0, "", or None value, it returns False.
- | In the code below, only all(l1) returns True.

```
1 l1 = [1, 2, 3, 4] # all elements are not 0
2 l2 = [0, 2, 4, 8] # one element is 0
3 l3 = [0, 0, 0, 0] # all elements are 0
```

```
1 all(l1) # returns True only when all elements are true
```

True

```
1 all(l2) # returns True only when all elements are true
```

False

```
1 all(l3) # returns True only when all elements are true
```

False

## 2. Iterator Object

### 2.10. Built-in function for iterable objects : any

- | Any returns True if there is at least one iterable object that is true.
- | In the case below, only list l3 returns False for any(l3) function.

```
1 l1 = [1, 2, 3, 4] # all elements are not 0
2 l2 = [0, 2, 4, 8] # one element is 0
3 l3 = [0, 0, 0, 0] # all elements are 0
```

```
1 any(l1) # returns True if there is at least one element that is True
```

True

```
1 any(l2) # returns True if there is at least one element that is True
```

True

```
1 any(l3) # returns True if there is at least one element that is True
```

False

### 3. Comparison of List Comprehension and Lambda Expression

#### 3.1. Syntax of list comprehension

| The syntax of the list comprehension is as follows.

```
[ {expression} for {variable} in {iterator/sequence} if {conditional expression} ]
```

| The if conditional expression can be omitted in the list comprehension.

```
[ {expression} for {variable} in {iterator/sequence} ]
```

### 3. Comparison of List Comprehension and Lambda Expression

#### 3.1. Syntax of list comprehension

| Computing square value of a list by using the map and the lambda function

```
1 a = [1, 2, 3, 4, 5, 6, 7]      # List of consecutive values
2 a = list(map(lambda x: x**2, a)) # apply lambda function for each element of the list
3 print(a)
```

[1, 4, 9, 16, 25, 36, 49]

| Computing square value of a list by using the list comprehension

```
1 a = [1, 2, 3, 4, 5, 6, 7]      # List of consecutive values
2 a = [x**2 for x in a]          # apply x**2 for each element of the list
3 print(a)
```

[1, 4, 9, 16, 25, 36, 49]

| Computing square value of a list by using the list comprehension and range

```
1 a = [x**2 for x in range(1, 8)]
2 print(a)
```

[1, 4, 9, 16, 25, 36, 49]

| The results of the above three codes are the same. Python offers various methods as such.

### 3. Comparison of List Comprehension and Lambda Expression

#### 3.1. Syntax of list comprehension

- Below code shows filtering by the list function, the filter function and the lambda function.

```
1 ages = [34, 39, 20, 18, 13, 54]
2 adult_ages = list(filter(lambda x: x >= 19, ages))
3 print('adults list:', adult_ages)
```

adult list : [34, 39, 20, 54]

- List comprehension of the above code.

```
1 ages = [34, 39, 20, 18, 13, 54]
2 print('adults list:', [x for x in ages if x >= 19])
```

adult list : [34, 39, 20, 54]

- Two codes return same results.

Two codes use similar conditional filters and simple expressions

### 3. Comparison of List Comprehension and Lambda Expression

#### 3.2. List comprehension and lambda expression to simplify code

| List comprehension and lambda expression are similar in that they both simplify code.

```
1 [x for x in range(10)]      # list of numbers from 0 to 9  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
1 [x * x for x in range(10)] # square value of numbers from 0 to 9  
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

```
1 [x for x in range(10) if x % 2 == 0] # value of even numbers from 0 to 9  
[0, 2, 4, 6, 8]
```

```
1 [x for x in range(10) if x % 2 == 1] # value of odd numbers from 0 to 9  
[1, 3, 5, 7, 9]
```

```
1 [x * x for x in range(10) if x % 2 == 0] # square value of even numbers from 0 to 9  
[0, 4, 16, 36, 64]
```

```
1 [x * x for x in range(10) if x % 2 == 1] # square value of odd numbers from 0 to 9  
[1, 9, 25, 49, 81]
```

### 3. Comparison of List Comprehension and Lambda Expression

#### 3.2. List comprehension and lambda expression to simplify code

 Focus A list comprehension enables one-line coding as shown below.

```
1 s = input('enter multiple integers :').split()
```

```
enter multiple integers : 10 20 30 40 50
```

```
1 lst = [int(x) for x in s]
2 lst
```

```
[10, 20, 30, 40, 50]
```

```
1 [int(x) for x in input('enter multiple integers : ').split()]
```

```
enter multiple integers : 1 2 3
```

```
[1, 2, 3]
```

| You can easily convert integer strings into a list of integer type values by using the int function.

### 3. Comparison of List Comprehension and Lambda Expression

#### 3.2. List comprehension and lambda expression to simplify code

| This code generates a new list by multiplying two lists. It used the double for loop.

```
1 product_xy = []
2 for x in [1, 2, 3]: # computes multiplication of elements of two lists by using double for loop
3     for y in [2, 4, 6]:
4         product_xy.append(x * y)
5 print(product_xy)
```

[2, 4, 6, 4, 8, 12, 6, 12, 18]

| Above code can be condensed into a one-line code by using a list comprehension.

```
1 product_xy = [x * y for x in [1, 2, 3] for y in [2, 4, 6]]
2 print(product_xy)
```

[2, 4, 6, 4, 8, 12, 6, 12, 18]

### 3. Comparison of List Comprehension and Lambda Expression

#### 3.3. Application of list comprehension and lambda expression

**Ex** Computing multiple of 2 and 3 by using list comprehension and lambda expression

```
1 [n for n in range(1, 31) if n % 2 == 0 if n % 3 == 0]
```

```
[6, 12, 18, 24, 30]
```

```
1 list(filter(lambda x : (x % 2 == 0 and x % 3 == 0), range(1, 31)))
```

```
[6, 12, 18, 24, 30]
```

**Ex** Computing multiple of 2, 3, and 5 by using a list comprehension and a lambda expression

```
1 [n for n in range(1, 31) if n % 2 == 0 if n % 3 == 0 if n % 5 == 0]
```

```
[30]
```

```
1 list(filter(lambda x : (x % 2 == 0 and x % 3 == 0 and x % 5 == 0), range(1, 31)))
```

```
[30]
```

You can change the condition by adding more ifs. (computing multiple of 2, 3, 5 among numbers from 1 to 30)

# | Pair programming



# Pair Programming Practice

## Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns “driving the mouse.” Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

## Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a “divide-and-conquer” strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

## Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.



# Pair Programming Practice



## | Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

## | Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide and conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

**Q1.**

If you treat students' scores of English, math, and science exams as list of three elements, it can be expressed as a list such as [100, 90, 95]. If there are two students, their scores can be expressed as [100, 90, 95, 90, 85, 93]. If a student did not apply to the exam of a certain subject, denote that score as 0. Print how many students' scores are contained in the given scores list, the number of students with valid scores for all subjects (that is students with no 0 for all subject), and the scores of students with only valid scores.

### Example of Input

```
scores = [100, 90, 95, 90, 80, 70, 0, 80, 90, 90, 0, 90, 100, 75, 20, 30, 50, 90]
```

### Example of Output

```
scores = [100, 90, 95, 90, 80, 70, 0, 80, 90, 90, 0, 90, 100, 75, 20, 30, 50, 90]
```

The number of total students is 6.

The number of students with valid scores is 4.

```
[[100, 90, 95], [90, 80, 70], [100, 75, 20], [30, 50, 90]]
```



Write the entire code and the expected output results in the note.

Unit 20.

# Closure

## Learning objectives

- ✓ Understand and utilize usage scope of variables by understanding the scoping rule of variables.
- ✓ Call and operate a function inside a function through outer functions and nested-functions.
- ✓ Understand the concept of a local variable inside a function and a global variable defined outside a function. Also, detect the closest variable by using the nonlocal keyword inside a function.
- ✓ Return the reference of an inner function contained within a function by a closure.
- ✓ Build a closure by using a lambda function.
- ✓ Convert a local variable of a closure by using the nonlocal.

## Learning overview

- ✓ Learn the valid scope of Python variables.
- ✓ Create a nested-function and create and call a function inside a function.
- ✓ Learn how to use nonlocal keyword which finds the nearest local variable.
- ✓ Understand why closure is necessary and how to define it.
- ✓ Learn how to define closure through lambda.
- ✓ Learn how to convert local variable of a closure.

## Concepts You Will Need to Know From Previous Units

- ✓ Utilizing lambda functions.
- ✓ Utilizing the filter, map, reduce and Python built-in functions in which lambda functions can be easily applied.
- ✓ Being able to efficiently pass values by using variable parameters, basic parameters, and keyword parameters.

# Keywords

scoping rule

nested function

local variable and  
global variable

closure

nonlocal

## 1. Python's scoping rule

### 1.1. Scope rule

- | This unit introduces many high-level functions and abstract concepts.
- | This unit focuses on examining the valid scope of variables or function names.
- | Programming languages including Python sets a rule on scope of access and valid context for a variable, and this is called the Scope Rule.
- | Python distinguishes the scope of variables as follows.
  - ▶ L(Local) : a local variable defined within a function
  - ▶ E(Enclosing Function Local): a scope of a function that contains another function
    - Python, unlike other programming languages, can define a function inside a function. (inner function)
  - ▶ G(Global): a variable of a module-level namespace that is not included in the scope of the function
  - ▶ B(Built-in): a variable of the built-in namespace.

## 1. Python's scoping rule

### 1.2. Python's scope rule

- | If a variable is defined inside a function, it becomes a local variable. If defined outside a function, it becomes a module-level global variable.
- | Examine L, G, E variables through the exercise below.
- | B is a variable from the built-in domain and does not appear below.
- | You will not be able to understand the result below if you do not observe the features of each variable.

```
1 x = 10
2 y = 11          # corresponds to G
3 def foo():
4     x = 20      # foo function corresponds to L
5     def bar():
6         a = 30  # bar function corresponds to E
7         print(a, x, y) # each variable corresponds to L, E and G
8     bar()
9     x = 40
10    bar()
11
12 foo()
```

30 20 11  
30 40 11

# 1. Python's scoping rule

## 1.2. Python's scope rule

💡 Take note of TypeError that occurs in built-in functions

```
1 abs      # built-in function abs()  
<function abs(x, /)>  
  
1 abs = 10 # global function abs()  
  
1 abs(-5) # global function abs() blocks built-in function abs()
```

```
TypeError                                     Traceback (most recent call last)  
<ipython-input-13-cb24a21394fc> in <module>  
----> 1 abs(-5) # global function abs() blocks built-in function abs()
```

TypeError: 'int' object is not callable

```
1 del abs      # deletes global function  
2 print(abs(-10)) # built-in function abs appears  
10
```

- You need to be cautious with built-in functions. Like `abs=10`, the name of a built-in function is often used as a variable.
- The built-in function `abs` was blocked because the name `abs` was used by the global variable.
- The built-in function executes its task once the reserved word `del` deletes global function's `abs`.

## 1. Python's scoping rule

### 1.2. Python's scope rule

- | Now generate a global variable counter, and use an assignment such as counter = 200 inside the function. 200 will be printed inside the function. Will both the function's inside and outside print 200?

```
1 def print_counter():
2     counter = 200
3     print('counter =', counter) # counter value inside the function
4
5 counter = 100
6 print_counter()
7 print('counter =', counter) # counter value outside the function
```

```
counter = 200
counter = 100
```

- | When you assign counter = 200, this variable is newly declared and defined inside the function.
- | Likewise, a variable generated inside a function is called a local variable. The value 200 referred by the local variable counter has a separate memory space from that of 100 referred by the global variable. The local variable disappears once the function is completed.
- | Therefore 100 instead of 200 is printed when you print the counter variable outside the function.

# 1. Python's scoping rule

## 1.3. Local variable and global variable

| Compare a local variable, which is a variable inside a function, with a global variable, which is used in the entire code.

Local variable (L)C which is generated when the function is called and disappears from the memory once the calling is complete.

```
1 def print_counter():
2     counter = 200
3     print('counter =', counter) # counter value inside the function
4
5 counter = 100
6 print_counter()
7 print('counter =', counter) # counter value outside the function
```

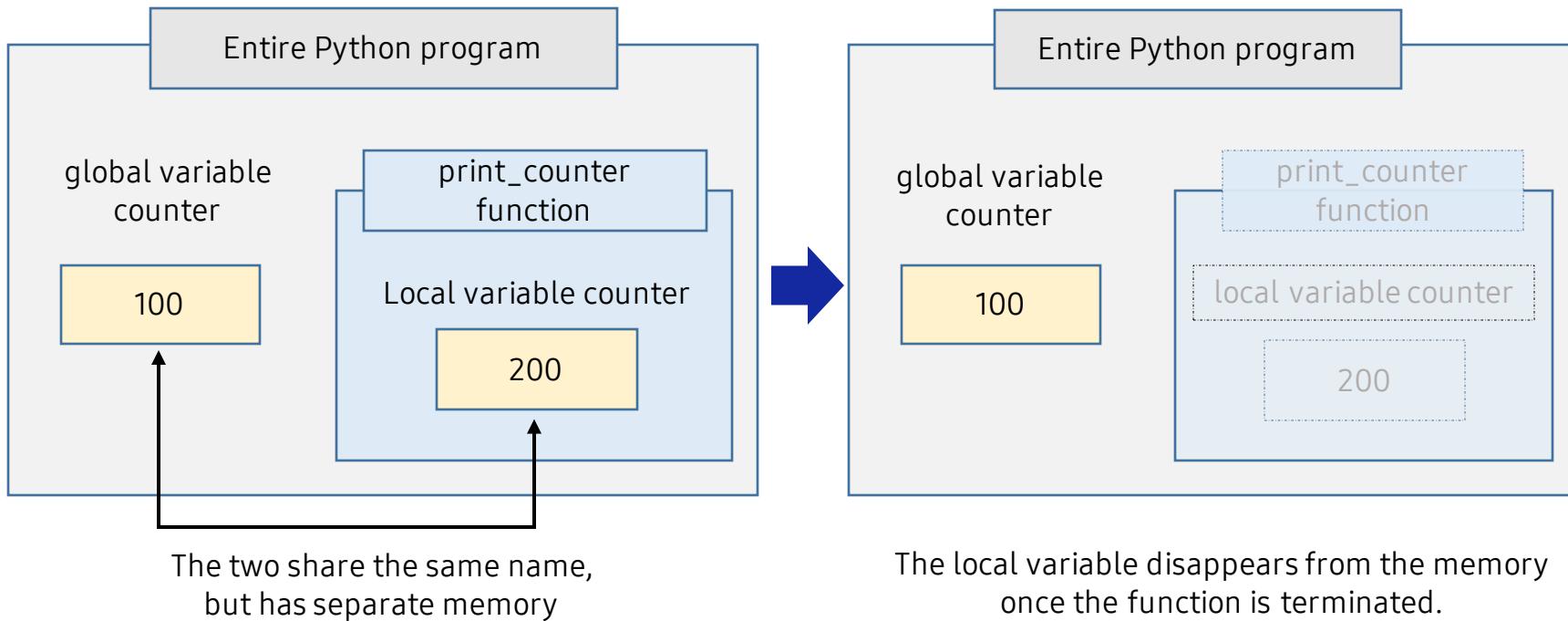
counter = 200  
counter = 100

Global variable(G) which is stored in the memory regardless of the function call of print\_counter

## 1. Python's scoping rule

### 1.3. Local variable and global variable

 Focus A local variable disappears from the memory once the function terminates.



# 1. Python's scoping rule

## 1.3. Local variable and global variable

 The global keyword means that the function will use a global variable outside the function.

- | What should you do to call the global variable counter without creating a new variable inside a function? Use the global keyword as shown below. global counter declares that the function will use the global variable counter outside the function.

```
1 def print_counter():
2     global counter      # declaration to use the global variable counter outside the function
3     counter = 200
4     print('counter =', counter) # counter value inside the function
5
6 counter = 100
7 print_counter()
8 print('counter =', counter)      # counter value outside the function
```

```
counter = 200
counter = 200
```

- | With such declaration, the value of the global variable changes to 200 when you assign count = 200 because print\_counter uses the outer global variable counter.

# 1. Python's scoping rule

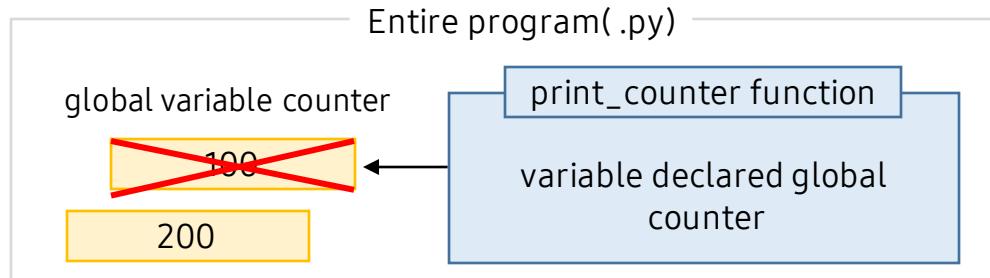
## 1.3. Local variable and global variable

| The global keyword changes the value outside the function.

With global declaration, the function uses the global variable, not the local variable counter. If you change the value inside the function, the value outside the function changes as well.

```
1 def print_counter():
2     global counter      # declaration to use the global variable counter outside the function
3     counter = 200
4     print('counter =', counter) # counter value inside the function
5
6     counter = 100
7     print_counter()
8     print('counter =', counter)      # counter value outside the function
```

```
counter = 200
counter = 200
```



The variable declared global references the global variable

# 1. Python's scoping rule

## 1.4. First-class function

I Study the first-class function

- ▶ A first-class function can be treated like any other object, like we generate numbers, strings, classes and store them in variables.
- ▶ You can pass a function to a parameter, insert it to another variable or use it as a return value. You can even store a function in data types like a list or a dictionary. The code below executes callfunc function by receiving a function as a parameter.

```
1 def callfunc(func):  
2     func()  
3  
4 def greet():  
5     print('Hello')  
6  
7 print('callfunc(greet) function call')  
8 callfunc(greet)
```

callfunc(greet) function call  
Hello

Insert the greet function as an argument for calling the callfunc function.

# 1. Python's scoping rule

## 1.4. First-class function

| A Python's function can be stored in a list and called like a variable.

```
1 def plus(a, b):
2     return a + b
3 def minus(a, b):
4     return a - b
5 l_list = [plus, minus]
6 a = l_list[0](100, 200)
7 b = l_list[1](100, 200)
8 print('a =', a)
9 print('b =', b)
```

```
a = 300
b = -100
```

You can store names of the functions in a list and call them using indexes.

## 1. Python's scoping rule

### 1.5. High-level functions implemented by first-class functions

| You can execute complex tasks by using the features of first-class functions

- ▶ Pass a function as an argument for another function. (Store a function in a variable)
- ▶ Pass a function as a return value of another function. (Use a function as a return value)
- ▶ Store a function in a variable or a data structure.

```
1 def add(a, b):  
2     return a + b  
3  
4 def f(g, a, b):  
5     return g(a,b)  
6 f(add, 3, 4)
```

7

| You can use a function as an argument or a return value as shown above, because all functions in Python are first-class functions.

## 2. Outer Function and Nested Function

### 2.1. Example of defining nested function in Python

- In Python, you can define italic, bold function in the decorate function as shown in the code below and can pass a function as a return value of a function.
- ▶ This exercise used the feature of Python which is that you can define a function inside a function, and that you can pass a function as a return value of a function.
  - ▶ You can structurally write a code that should have been written with a control statement in other languages.
  - ▶ You can simplify a code by using less control statements.

```
1 def decorate(style = 'italic'):  
2     def italic(s):  
3         return '<i>' + s + '</i>'  
4     def bold(s):  
5         return '<b>' + s + '</b>'  
6     if style == 'italic':  
7         return italic  
8     else:  
9         return bold  
10    dec = decorate()  
11    print(dec('hello'))  
12    dec2 = decorate('bold')  
13    print(dec2('hello'))
```

We call an inner function that was declared inside a function as a nested-function.

```
<i>hello</i>  
<b>hello</b>
```

## 2. Outer Function and Nested Function

### 2.2. Reasons for using nested function

- | Nested function is a function inside a function. Unlike the functions located on the outside, it can freely read variables of the parent function.
- | You can improve legibility by using a nested function.
- | However, just for legibility purposes, you can declare a function outside a function. Examine the code below.
- | The code below runs properly.

```
1 def another_func():
2     print("hello")
3
4 def outer_func():
5     return another_func()
6
7 outer_func()
```

hello

## 2. Outer Function and Nested Function

### 2.2. Reasons for using nested function

- | A more important reason for using nested function is a notion called closure.
- | One of the necessary conditions of implementing closure is executing the nested function.
- | The nested function is an important concept for understanding closure.

### 3. Process of Finding Nonlocal Variable

#### 3.1. Keyword 'global'

- If you need to edit n1 inside a function after declaring it as a global variable, declare global n1. This means that "I will use the global variable n1, not the n1 inside the current function."

```
1 n1 = 1 ← Global variable finds its variable from the
2 def func1():
3     def func2():
4         global n1 name domain of a global variable
5         n1 += 1
6         print(n1) # print 2
7     func2()
8
9 func1()
```

2

### 3. Process of Finding Nonlocal Variable

#### 3.2. Necessity of nonlocal keyword

##### ! NameError

```
1 def func1():
2     n2 = 1
3     def func2():
4         global n2
5         n2 += 1
6         print(n2) # error
7     func2()
8
9 func1()
```

```
NameError                                 Traceback (most recent call last)
<ipython-input-8-4a8342131d9b> in <module>
      7     func2()
      8
----> 9 func1()

NameError: name 'n2' is not defined
```

- If you try to refer to global keyword n2 when there is no global variable n2, NameError: name'n2' is not defined occurs. This is because there is no n2 declared as a global variable.
- For such cases, use the nonlocal keyword instead of the global keyword.

### 3. Process of Finding Nonlocal Variable

#### 3.2. Necessity of nonlocal keyword

```
1 def func1():
2     n3 = 1
3     def func2():
4         nonlocal n3
5         n3 += 1
6         print(n3)    # It's not error
7     func2()
8
9 func1()
```

nonlocal n3 variable is not a local variable.  
It is not a global variable either, so the nearest n3 variable is connected.

2

- | Use the nonlocal keyword when you use n3 variable which is not a local variable nor a global variable in the current scope.
- | If you set nonlocal n3 with the intention to use the nearest n3 variable that is not a local variable, there is no problem.
- | Connecting with the nearest variable as such is called binding.

### 3. Process of Finding Nonlocal Variable

#### 3.3. nonlocal keyword and binding

 Focus You can bind with a variable that is located one level outside the function that used the nonlocal.

| nonlocal : nonlocal keyword here declares that x is not a local variable.

```
1 x = 20 # global variable
2 def f():
3     x = 40 ←
4     def g():
5         nonlocal x
6         x = 80
7     g()
8     print(x)
9
10 f()
11 print(x)
```

nonlocal x variable is not a local variable.  
It is not a global variable either, so the variable of f function is bound.

80

20

 Line 7, 8, 11

- Apply the nonlocal by executing function g.
- Print x value from function f. (the variable of function g has changed to 80 due to the nonlocal.)
- After the execution of all functions, print the variable x. (output value is the first value 20.)
- **Binding is the variable x being tied with an actual value.**

### 3. Process of Finding Nonlocal Variable

#### 3.4. Relation between nonlocal and global variable

##### ! nonlocal and global variable

```
1 x = 70 # global variable
2 def f():
3     nonlocal x
4     x= 140
5
6 f()
7 print(x)
```

```
File "<ipython-input-33-5aa47a6c6d0e>", line 3
    nonlocal x
    ^

```

SyntaxError: no binding for nonlocal 'x' found

##### Line 3

- Define only one function and declare the nonlocal inside this function in order to keep the global variable intact.
- In this case, syntax error occurs.

### 3. Process of Finding Nonlocal Variable

#### 3.5. Process of finding nonlocal variable

- | Suppose a case where a g function is defined in an f function, and an h function is defined in a g function.
- | The value of a changed because after the h function inside the g was executed, a referenced a of the g by the nonlocal.
- | a didn't change after the g inside the f was executed. This is because it did not reference a value of the f due to the nonlocal.
- | That is, the nonlocal finds a variable from the nearest namespace.

```
1 def f():
2     a = 777
3     def g():
4         a = 100 ←
5         def h():
6             nonlocal a ←
7             a = 333
8             h()
9             print("[Level 2] a = {}".format(a))
10            g()
11            print("[Level 1] a = {}".format(a))
12
13 f()
```

Nonlocal variable find a variable from  
the nearest namespace.

```
[Level 2] a = 333
[Level 1] a = 777
```

## 4. Concept of Closure

### 4.1. About Closure

- | The basic form of a closure function is as follows. It creates an nested function inside a function and returns the nested function as a return value.
- | Here you can say that the nested function 'was closed.' If the function is closed, the variable inside the body of the original function (outer function) does not disappear from the memory. Then, in the next call it can utilize the nested function.
- | Study the concept of the closure with the following code.
- | What are free variables? A free variable is only used in a certain code block, but it is not a global variable, and it is not defined within the block. It is a relative concept.
- | That is, in the point of view of mult function, a is a free variable. However, for closure-calc, a is only a local variable.

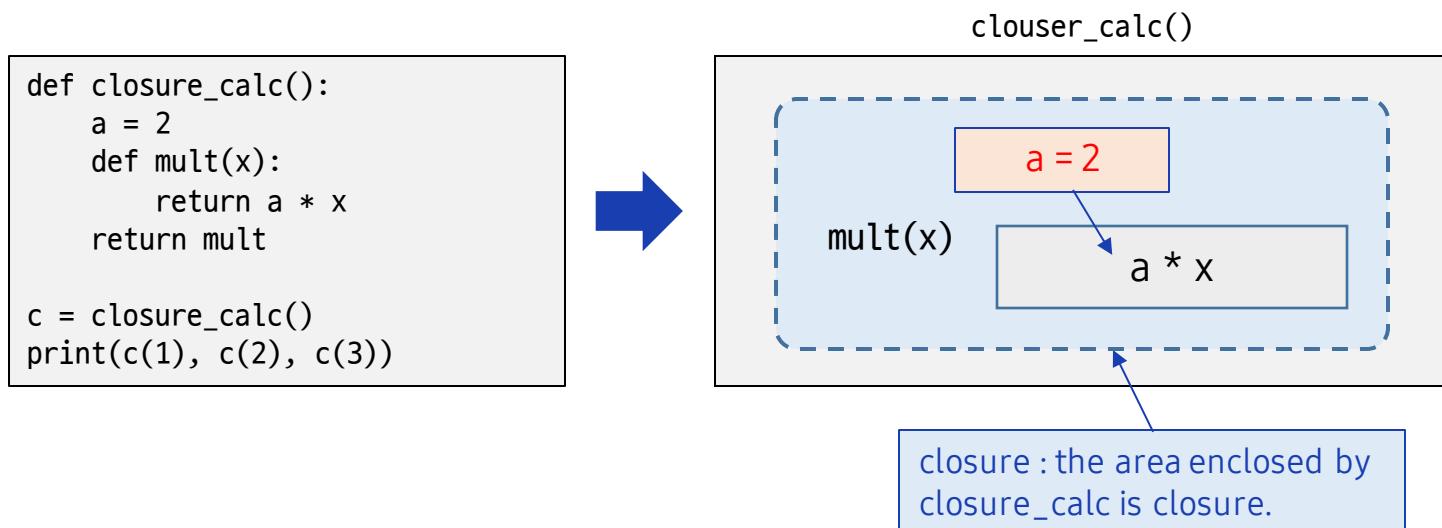
```
1 def closure_calc():
2     a = 2
3     def mult(x):
4         return a * x
5     return mult
6
7 c = closure_calc()
8 print(c(1), c(2), c(3))
```

2 4 6

## 4. Concept of Closure

### 4.2. Analyzing closure function

- In the code below, the return value of the closure\_calc function is the mult function. It returns the reference of its inner function.
- Because the return value of closure\_calc is mult, and python's functions are all first-class functions, it can call mult through c.
- However, mult was processed even though the variable a, which was supposed to be used by c, was a local variable of closure\_calc. This means that the variable a of mult did not disappear after the execution of mult was completed.



## 4. Concept of Closure

### 4.3. Purpose of closure

- | In traditional programming you had to declare a variable as a global variable in order to use it for multiple functions regardless of a certain function's termination.
- | However, indiscriminate generation of global variables causes side effects.
- | Programming itself becomes easy if you use global variables. However, too much will lead to side effects, creating difficulties for debugging.
- | To create function closure, define the variable for closure and function body and enclose it with another function.
- | If there is an outer function that encloses function closure, the return type of this outer function becomes closure.
- | This can reduce frequency of general variable and hide the function's inner workings from the outside.

## 4. Concept of Closure

### 4.4. Main usage of closure

| Function closure is mainly used for following two purposes.

1. Limiting usage of global variables
2. Data hiding

- ▶ In terms of memory operating efficiency, closure is inefficient. However it could be more effective because it reduces global variables.
- ▶ If you want to hide data, declare a closure as the enclosing function's local variable.

| A function closure provides an independent namespace for each function. Then you can define a function that behaves independently like an instance object.

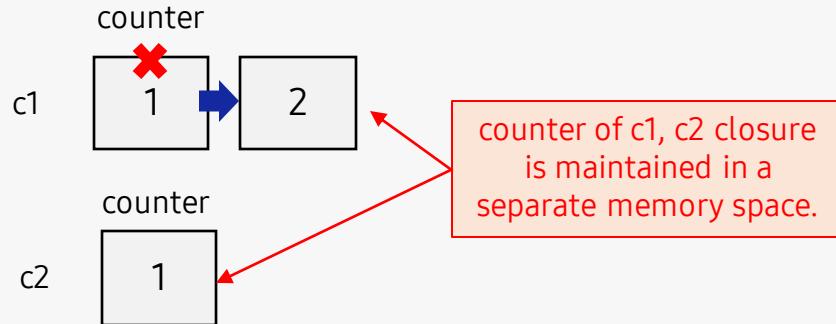
- ▶ More about instances in Class chapter.

## 4. Concept of Closure

### 4.4. Main usage of closure

| A closure has a separate memory for each variable.

```
1 def makecounter():
2     count = 0
3     def counter():
4         nonlocal count
5         count += 1
6         return count
7     return counter
8 c1 = makecounter()
9 c2 = makecounter()
10 print('c1', c1())
11 print('c1', c1())
12 print('c2', c2())
```



```
c1 1
c1 2
c2 1
```

- ▶ From the above exercise, you can see that the closure function has a separate memory space for each variable.
- ▶ Each of `count` for `c1` and `count` for `c2` has its own memory space.

## 4. Concept of Closure

### 4.5. How to generate a closure

| You generate a closure by writing a nested-function and returning it.

```
1 def calc():
2     a = 3
3     b = 5
4     def mul_add(x):
5         return a * x + b
6     return mul_add
7
8 c = calc()
9 print(c(1), c(2), c(3), c(4), c(5))
```

8 11 14 17 20

#### Line 5, 6

- Computes by using the local variable a, b located outside the function.
- Returns mul\_add function.

## 5. Making Closure with Lambda

### 5.1. Code example of making closure with lambda

- | Create a lambda expression like return lambda x: a\* x + b and return the lambda expression itself.
- | As such, you can create a closure in a simpler way by using a lambda.
- | Normally a closure is jointly used with a lambda expression, and it is easy to confuse the two.
- | A lambda means an anonymous function, and a closure means a function that is reused after maintaining its enclosing environments.

```
1 def clouser_calc():
2     a = 2
3     b = 3
4     return lambda x : a * x + b    # making closure with lambda
5
6 c = clouser_calc()
7 print(c(1), c(2), c(3), c(4), c(5)) # prints 5, 7, 9, 11, 13
```

5 7 9 11 13

## 5. Making Closure with Lambda

### 5.1. Code example of making closure with lambda

| Compare the code before and after using lambda

```
1 def calc():
2     a = 3
3     b = 5
4     def mul_add(x):
5         return a * x + b
6     return mul_add
7
8 c = calc()
9 print(c(1), c(2), c(3), c(4), c(5))
```

8 11 14 17 20

Before using lambda



```
1 def clouser_calc():
2     a = 3
3     b = 5
4     return lambda x : a * x + b    # making closure with lambda
5
6 c = clouser_calc()
7 print(c(1), c(2), c(3), c(4), c(5)) # prints 5, 7, 9, 11, 13
```

8 11 14 17 20

After using lambda: Simpler expression

## 6. Change Local Variable of Closure

### 6.1. Code example of changing a local variable of a closure

- | Use the nonlocal to change the local variable of a closure
- | The following accumulates the results of  $a * x + b$  in the local variable in the total of function calc.

```
1 def calc():
2     a = 2
3     b = 3
4     total = 0 ←
5     def mult_add(x):
6         nonlocal total
7         total = total + a * x + b
8         return total
9     return mult_add
10
11 c = calc()
12 print(c(1), c(2), c(3))
```

A nonlocal finds a variable from the nearest namespace.  
Here it is used as a local variable of a closure.

5 12 21

# | Paper coding

**Try to fully understand the basic concept before moving on to the next step.**

**Lack of understanding basic concepts will increase your burden in learning this course, which may make you fail the course.**

**It may be difficult now, but for successful completion of this course we suggest you to fully understand the concept and move on to the next step.**

**Q1.**

Create a nested-function by defining a function named greetings and another function named say\_hi inside that function. Call say\_hi function within greetings. Then call greetings and print 'hello'. say\_hi function is shown below.

```
def say_hi():
    print('hello')
```

Condition for Execution	hello
Time	5 Minutes



Write the entire code and the expected output results in the note.

**Q2.** Write the following function calc and assign calc to variable num.  
Then, execute num(3). Make the execution result 14 as follows.

```
def calc():
    a = 3
    b = 5
    def mul_add(x):
        return a * x + b
    return mul_add
```

Condition for Execution

14

Time

5 Minutes



Write the entire code and the expected output results in the note.

**Q3.** Build mul\_add, the inner function of the nested function calc from the previous problem, by using lambda expressions, and print the following result.

Condition for Execution	14
Time	5 Minutes



Write the entire code and the expected output results in the note.

| Let's code

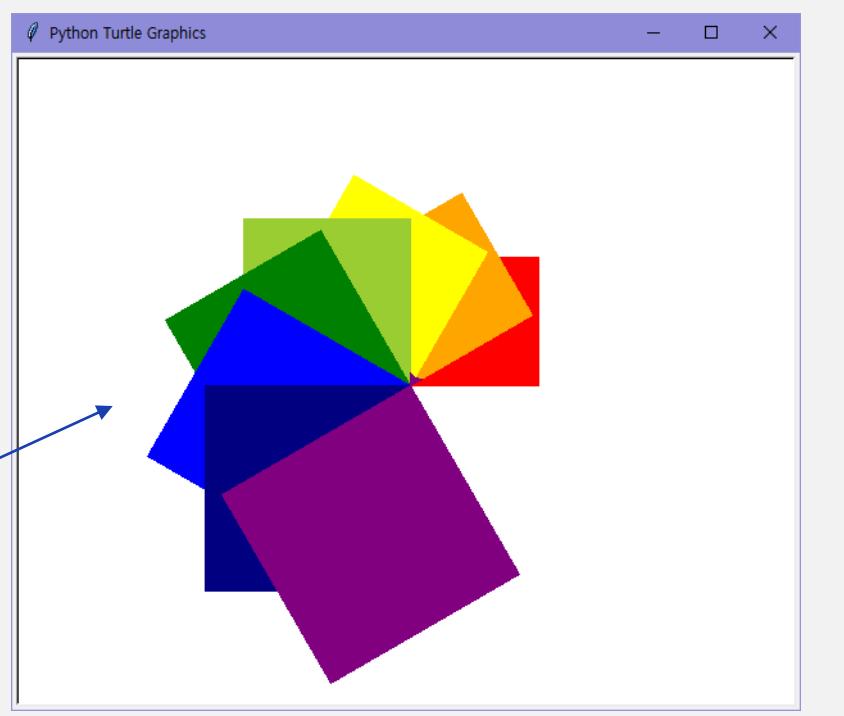
## 1. Turtle Graphics

### 1.1. Turtle graphics : a drawing tool in Python

| This unit's Let's code discusses turtle graphics, a drawing tool in Python.

- ▶ Supports drawing graphics in Python.
- ▶ Built-in as a Python's basic module.
- ▶ Contains various methods.
- ▶ Has abundant features.

You can see beautiful  
graphics in the computer screen

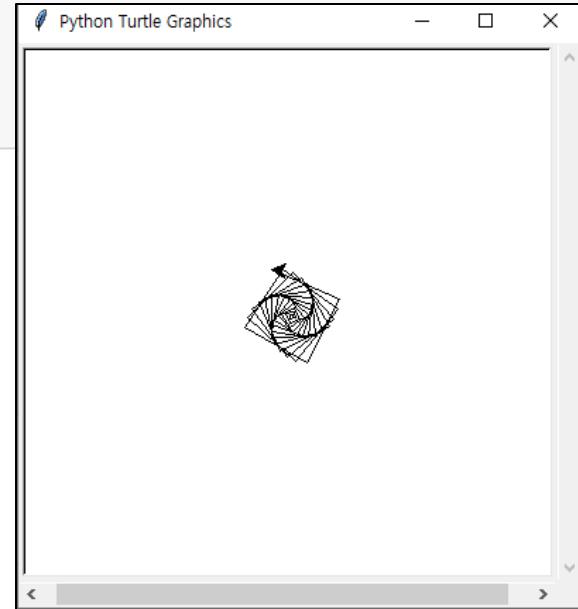


## 1. Turtle graphics

### 1.2. Drawing a picture with turtle

- | If you run the code below, a separate window pops up and a cursor draws a picture.
- | In this code, t is a nickname of the turtle module.

```
1 import turtle as t
2
3 t.setup(width = 400, height = 400)
4 for i in range(200):
5     t.forward(i)
6     t.left(93)
7 t.done()
```



## 1. Turtle graphics

### 1.3. Turtle initialization and modifying shapes

- | A new window pops up if you enter the following code in the Python chat window.
- | t.setup function assigns the size, title and other features of the chat window.

```
1 import turtle as t
2
3 t.setup(width = 400, height = 400)
4 for i in range(200):
5     t.forward(i)
6     t.left(93)
7 t.done()
```

- ▶ t.forward(i) moves the cursor to the left for i pixels while drawing a black line.
- ▶ t.left(93) rotates the cursor to 98 degrees left.

# 1. Turtle graphics

## 1.4. Turtle commands

So far, we have learned simple turtle graphics commands such as `forward()`, `left()`, `right ()` etc. Turtle graphics offers many more features. The table below shows the commands of the turtle and their tasks.

command	task
<code>begin_fill() ... end_fill()</code>	Colors the area of code between <code>begin_fill()</code> and <code>end_fill ()</code> . You can note the turtle's coordinate or shape in the ... section.
<code>color(c)</code>	Changes the color of the turtle. You can choose many colors such as 'red', 'green', 'blue', 'black', 'gray', 'pink' with the value c.
<code>shape(s)</code>	Changes the shape of the turtle. Values of s can be 'arrow', 'turtle', 'circle', 'square', 'triangle', 'classic' etc.
<code>shapesize(s), shapesize(w, h)</code>	Modifies the size of the turtle.
<code>pos(), position()</code>	Returns the current position of the turtle.
<code>xcor()</code>	Returns the current x coordinate of the turtle.
<code>ycor()</code>	Returns the current y coordinate of the turtle.
<code>heading()</code>	Returns the current heading.
<code>distance(x, y)</code>	Computes the distance between the current position of the turtle and a designated position.
<code>penup(), pu(), up()</code>	Picks up the pen (disables drawing).

# 1. Turtle graphics

## 1.4. Turtle commands

### I Initializing turtle and modifying shapes

pendown() , pd(), down()	Puts the pen down (enables drawing).
pensize(w), width(w)	Changes the thickness of the Pen
circle(r)	Draws a circle with the size of radius r on the current position.
goto(x, y), setpos(x,y), setposition(x,y)	Sends the cursor to a specific coordinate. It draws a line if penup(), does not if pendown(),
stamp()	Shows the size, color and shape of the turtle on the current position of the turtle.
home()	Initializes the position and direction of the turtle.
textinput()	Displays the chat window for text input, and this window receives strings. (Caution : must use with <code>turtle.textinput()</code> )

- ▶ There are other various commands that enable the user to draw any kind of drawing.

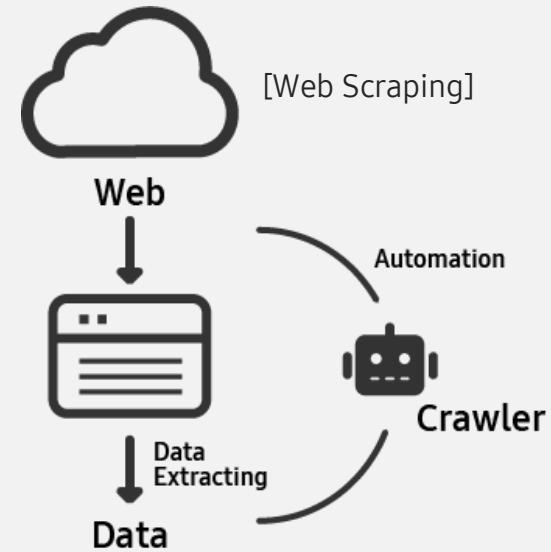
## 2. Regular Expressions and Meta Characters

### 2.1. Regular expressions

- ▶ Regular expressions are a language that contains the patterns used by strings with certain rules.
- ▶ Regular expressions are important for web page parsing and crawling.
- ▶ Learn about regular expressions with some simple exercises.

Extracting web pages is called crawling.

<https://stackhoarder.com/2019/08/18/python%EB%B6%80%ED%84%BO-web-scraping-%EA%B9%8C%EC%A7%80-%EC%B5%9C%EB%8B%A8-%EC%8B%9C%EA%B0%84%EC%97%90-%EC%9D%B5%ED%98%80%EB%B3%B4%EC%9E%90/>



## 2. Regular Expressions and Meta Characters

### 2.2. Searching with regular expressions

- | The syntax of regular expressions are complex. This chapter will treat only basic search functions.
- | You need to include re module to use regular expressions in Python.
- | You can check if a string contains a specific string by using the search function.
- | Check if the two strings txt1 and txt2 contain the Life string.

```
1 import re
2
3 txt1 = "Life is too short, you need python."
4 txt2 = "The best moments of my life."
5 print(re.search('Life', txt1))    # checks if Life is in the sentence
```

```
<re.Match object; span=(0, 4), match='Life'>
```

```
1 print(re.search('Life', txt2))    # checks if Life is in the sentence
```

```
None
```

- ▶ The result of the print statement for txt1 says 'match' because txt1 contains Life with capital L.
  - If there is a matching string, messages such as <re.Match object ..> are printed.
- ▶ The result of the print statement for txt2 says None because txt2 does not contain Life with capital L.

## 2. Regular Expressions and Meta Characters

### 2.2. Searching with regular expressions

- | Call the group method of the match variable. Since Life shows only once, the group prints Life.
- | The group method finds the matching part of the entire regular expression. Here Life is printed.

```
1 match = re.search('Life', txt1)
2 match.group()
```

'Life'

- ▶ To interpret this output, assign the result value of re.search to the match variable, and call start, end, span method to the variable.

```
1 match.start()
```

0

```
1 match.end()
```

4

```
1 match.span()
```

(0, 4)

## 2. Regular Expressions and Meta Characters

### 2.2. Searching with regular expressions

- | The group method returns the string matched by regular expression. It shows start and end index of the matched string by using start, end method.
- | Also, span method returns a tuple containing the (start, end) positions of the match

```
1 match.start()
```

```
0
```

```
1 match.end()
```

```
4
```

```
1 match.span()
```

```
(0, 4)
```

```
1 txt[0:4]
```

```
'Life'
```

## 2. Regular Expressions and Meta Characters

### 2.2. Searching with regular expressions

- | In case of a text string as follows, the compile function of the re module searches regular expression patterns and compiles them to regular expression objects.
- | Then, a regular expression object called regex is created, and this object searches text with a given pattern.
- | \d{3} finds patterns of 3 integers. \d{4} finds patterns of 4 integers. The bracket places these patterns into group 1 and group 2.
  - ▶ That is, \d{3} finds three-digit integer patterns like 010.

```
1 import re
2
3 text = 'please call 010-2345-1234'
4 regex = re.compile('(\d{3})-(\d{4}-\d{4})')
5 match_obj = regex.search(text)
6 print(match_obj.group())
```

010-2345-1234

group(1)

group(1)

```
1 print(match_obj.group(1))
```

010

```
1 print(match_obj.group(2))
```

2345-1234

## 2. Regular Expressions and Meta Characters

### 2.2. Searching with regular expressions

- | Examine the result of search function of the regular expression.
- | It searches the intended string, but it cannot find life with lowercase l in txt2. Try the following method.

```
1 txt2 = "The best moments of my life."  
2 print(re.search('Life', txt2))    # checks if Life is in the sentence
```

None

```
1 print(re.search('Life|life', txt2))    # checks if Life is in the sentence  
<re.Match object; span=(23, 27), match='life'>
```

- ▶ In this case, since 'life' is located in txt2, it displays the index of 'life' through the result span (23, 27).
- ▶ The symbol | does not mean find 'Life|life'. Instead, it means find Life or life string.

## 2. Regular Expressions and Meta Characters

### 2.3. Meta characters

- Since Life or life share the same characters except for the first l, the following expression returns the same result.

```
1 print(re.search('[Ll]ife', txt2))    # checks if Life or life is in the sentence  
<re.Match object; span=(23, 27), match='life'>
```

- Like the previous exercise, [Ll]ife means Life or life. Square bracket [] expresses the range of character selection. For example, [0-9] means all number characters from 0 to 9.
- As such, you can search and replace certain characters by using characters with special meanings such as [], -, |. Such characters for special use are called meta characters.

## 2. Regular Expressions and Meta Characters

### 2.3. Meta characters

| Study ^ which means the start of a string in regular expressions.

```
1 txt1 = "Life is too short, you need python."  
2 txt2 = "The best moments of my life."  
3 txt3 = "My Life My Choice."  
4 print(re.search('^Life', txt1))    # checks if Life is the first word
```

→ <re.Match object; span=(0, 4), match='Life'>

```
1 print(re.search('^Life', txt2))    # checks if Life is the first word
```

→ None

```
1 print(re.search('^Life', txt3))    # checks if Life is the first word
```

→ None

- ▶ The above result shows that if you search txt3 using ^Life, the output is None. This is because the meta character ^ matches sentences with Life as the first word.

## 2. Regular Expressions and Meta Characters

### 2.3. Meta characters

| Apply regular expressions to the following strings.

```
1 txt1 = 'Who are you to judge the life I live'  
2 txt2 = 'The best moments of my life'  
3 print(re.search('life$', txt1)) # checks if Life or life is contained
```

None

```
1 print(re.search('life$', txt2)) # checks if Life or life is contained
```

```
<re.Match object; span=(23, 27), match='life'>
```

- ▶ You can find strings with life as the last word by using the \$ character in the second row of the table.

## 2. Regular Expressions and Meta Characters

### 2.4. Essential meta characters

| Let's discuss essential regular expressions functions. Essential meta characters for regular expressions are as follows.

Regular expression	Function	Description
^	start	Matches the start of the string
\$	end	Matches the end of the string
.	character	Matches one character.
\d	number	Matches one number.
\w	character or number	Matches one character or one number.
\s	whitespace characters	Matches a space, a tab, a carriage return and a line feed
\S	except whitespace characters	All characters except whitespace characters
*	repetition	0 or more repetitions
+	repetition	1 or more repetitions
[abc]	character range	[abc] displays a or b or c
[^abc]	character range	[^abc] is any character that is not a or b or c.

\* The most important among meta characters are the dot (.) and the asterisk (\*). The dot means that any string is possible, and the asterisk means that any amount of repetition is possible.

## 2. Regular Expressions and Meta Characters

### 2.4. Essential meta characters

- | The dot meta character.
- | This is a meta character that means any random character.
- | For example, among strings ABA, ABBA, ABBBA, the one that matches the condition A..A is ABBA, which is also the name of the Swedish band ABBA.

```
1 re.search('A..A', 'ABA')      # does not fit the condition
```

```
1 re.search('A..A', 'ABBA')      # fits the condition
```

```
<re.Match object; span=(0, 4), match='ABBA'>
```

```
1 re.search('A..A', 'ABBBA')    # does not fit the condition
```

## 2. Regular Expressions and Meta Characters

### 2.4. Essential meta characters

- The asterisk (\*) is a very powerful meta character.
- It matches with strings that repeat 0 times or more any random pattern of the string before the asterisk.

```
1 re.search('AB*', 'A')    # fits the condition
```

```
<re.Match object; span=(0, 1), match='A'>
```

```
1 re.search('AB*', 'AA')    # fits the condition
```

```
<re.Match object; span=(0, 1), match='A'>
```

```
1 re.search('AB*', 'J-HOP')  # does not fit the condition
```

```
1 re.search('AB*', 'X-MAN')  # fits the condition
```

```
<re.Match object; span=(3, 4), match='A'>
```

```
1 re.search('AB*', 'CABBA')    # fits the condition
```

```
<re.Match object; span=(1, 4), match='ABB'>
```

```
1 re.search('AB*', 'CABBBBBA')    # fits the condition
```

```
<re.Match object; span=(1, 7), match='BBBBBB'>
```

## 2. Regular Expressions and Meta Characters

### 2.4. Essential meta characters

| Now the next widely used character ?. This character matches with patterns that doesn't repeat or repeat once a random character before the character.

```
1 re.search('AB?', 'A')    # fits the condition  
<re.Match object; span=(0, 1), match='A'>
```

```
1 re.search('AB?', 'AA')    # fits the condition  
<re.Match object; span=(0, 1), match='A'>
```

```
1 re.search('AB?', 'J-HOP')  # does not fit the condition
```

```
1 re.search('AB?', 'X-MAN')  # fits the condition  
<re.Match object; span=(3, 4), match='A'>
```

```
1 re.search('AB?', 'CABBA')    # fits the condition  
<re.Match object; span=(1, 3), match='AB'>
```

```
1 re.search('AB?', 'CABBBBBA')  # fits the condition  
<re.Match object; span=(1, 3), match='AB'>
```

| The result seems to resemble that of \*, but there is a big difference in that the matching string is AB for CABBBBBA and CABBA.

## 2. Regular Expressions and Meta Characters

### 2.4. Essential meta characters

- | The meta character + matches patterns that repeat once or more a random pattern in front of +.
- | Thus AB+ does not match with A, but matches with strings of patterns like AB, ABB, ABBB, CABBA.

```
1 re.search('AB+', 'A')      # does not fit the condition
1 re.search('AB+', 'AA')     # does not fit the condition
1 re.search('AB+', 'J-HOP')   # does not fit the condition
1 re.search('AB+', 'X-MAN')   # does not fit the condition
1 re.search('AB+', 'CABBA')    # 'ABB' string fits the condition
<re.Match object; span=(1, 4), match='ABB'>
1 re.search('AB+', 'CABBBBB')  # 'ABBBBB' fits the condition
<re.Match object; span=(1, 7), match='ABBBBB'>
```

## 2. Regular Expressions and Meta Characters

### 2.4. Essential meta characters

- | About.findall which is a search command.
- | Findall of regular expressions extracts all strings that fit the regular expression.

```
1 txt3 = 'My life my life my life in the sunshine'  
2 re.findall('[Mm]y', txt3)
```

```
['My', 'my', 'my']
```

# | Pair programming



# Pair Programming Practice

## Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns “driving the mouse.” Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

## Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a “divide-and-conquer” strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

## Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.



# Pair Programming Practice



## | Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

## | Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide and conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

**Q1.**

Extract list result from list lst which has values of 1 ~ 100. list result has elements of list lst that are divisible by 5 or 7. Declare func1(a) function and nest func2 and func3 function. Then, call the two functions in func1 function and print numbers divisible by 5 or 7. Here, align the values by using the sorted () function.

**Print example**

```
def func2():
    result1 = []
    for i in a:
        if i % 5 == 0:
            result1.append(i)
    return result1

def func3():
    result2 = []
    for i in a:
        if i % 7 == 0:
            result2.append(i)
    return result2
```

```
lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 6
2, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92,
93, 94, 95, 96, 97, 98, 99, 100]
result = [5, 7, 10, 14, 15, 20, 21, 25, 28, 30, 35, 35, 40, 42, 45, 49, 50, 55, 56, 60, 63, 65, 70, 70, 75, 77, 80, 84, 8
5, 90, 91, 95, 98, 100]
```

Unit 21.

# Class

## Learning objectives

- ✓ Understand the basic concept of object-oriented programming and its necessity.
- ✓ Understand the basic concept of encapsulation and construct encapsulated codes for stability of program execution.
- ✓ Understand the concept of class instances and objects and denote the differences.
- ✓ Construct a class, understand how to construct an object and construct an object.
- ✓ Understand the basic concept of a constructor and use the self parameter of a class as a constructor.
- ✓ Understand the structure of method and define methods.
- ✓ Understand the basic concept of reference and explain the inner mechanism of assigning a class instance to a variable.
- ✓ Define a new class that inherits from a pre-existing class.

## Learning overview

- ✓ Study object-oriented programming and understand its necessity.
- ✓ Understand the basic concept of encapsulation and apply this concept to coding for program stability.
- ✓ Learn the basic concept of an object and a class instance.
- ✓ Construct a class, understand how to construct an object and construct an object.
- ✓ Assign a class object to a variable by an assigning operator. Learn the concept of reference which will be used here.
- ✓ Learn how to define a new class by inheriting a certain class.

## Concepts You Will Need to Know From Previous Units

- ✓ Creating a nested function and calling a function inside a function.
- ✓ Utilizing nonlocal keyword, which finds the nearest local keyword, and global keyword.
- ✓ Applying indentation and block statement.

# Keywords

Object

Class

Instance

Constructor

self Parameter

Inheritance

## 1. Definition and Concept of Object

### 1.1. Programming and object

| We will study objects in depth.

```
1 animals = ['lion', 'tiger', 'cat', 'dog']
2 animals.sort()
3 animals
['cat', 'dog', 'lion', 'tiger']
```

```
1 animals.append('rabbit')
2 animals
['cat', 'dog', 'lion', 'tiger', 'rabbit']
```

```
1 animals.reverse()
2 animals
['rabbit', 'tiger', 'lion', 'dog', 'cat']
```

- ▶ The animals list is an object that has 'lion', 'tiger', 'cat', 'dog' as elements (attributes).
- ▶ It also has sort, append, remove, reverse, pop as member functions (methods) (calls methods by a . (dot) notation )
  - The member functions owned by the animals object are called methods.
- ▶ Object
  - As such, elements in the computer system that have attributes and methods are called objects.



## One More Step

| Below is a comparison of object-oriented programming and procedural programming.

| Object-oriented programming (OOP)

- ▶ A technique that models a program in resemblance to the real world
- ▶ Expresses a task executed by the computer as **interactions among objects**.
- ▶ A concept that intends to develop software by sets of classes or objects.
- ▶ Adopted by many widely-used programming languages such as Java, Python, C++, C#, Swift etc.

| Procedural programming language

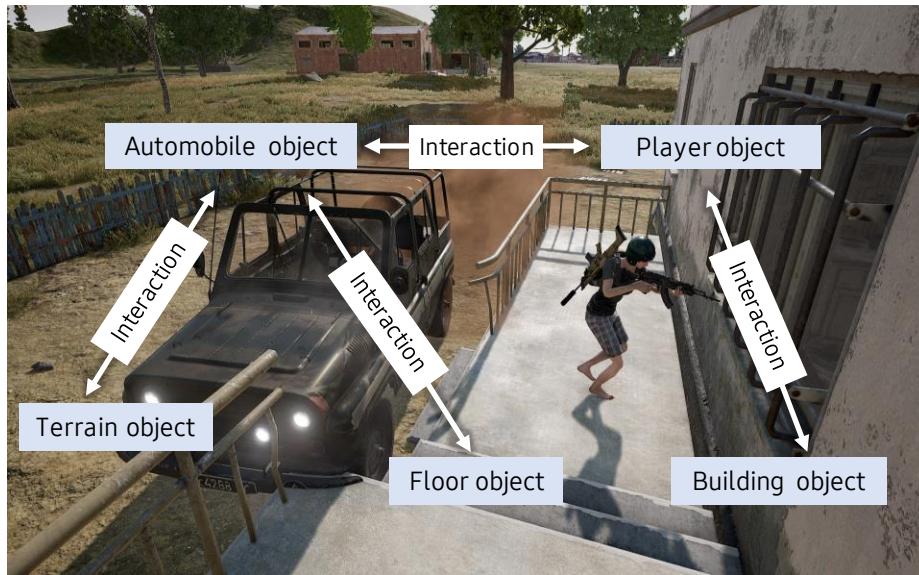
- ▶ A method that operates by calling functions and models in the order of problem solving.
- ▶ Adopted by traditional programming languages such as C, Fortran, Basic etc.
- ▶ Incapable of dealing with issues with systems of various graphic components such as a graphic user interface system.

## 1. Definition and Concept of Object

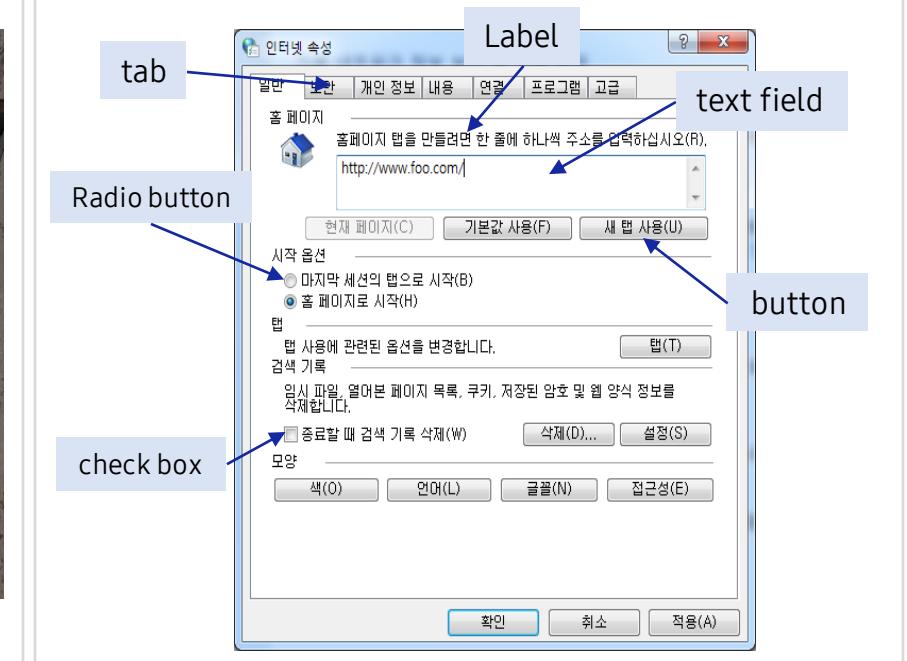
### 1.1. Programming and object

The concept of objects : Many materials that interact in a game are called objects. The elements of a graphic user interface are also called objects.

Interactions among objects in a game



Objects in a graphic user interface



## 1. Definition and Concept of Object

### 1.2. Comparison of procedural programming and object-oriented programming

#### | Procedural programming

- ▶ Demands a large numbers of arrows and functions calls if data and functions expand and increase.
- ▶ Difficult to handle large scale projects.

#### | Objected-oriented programming (OOP)

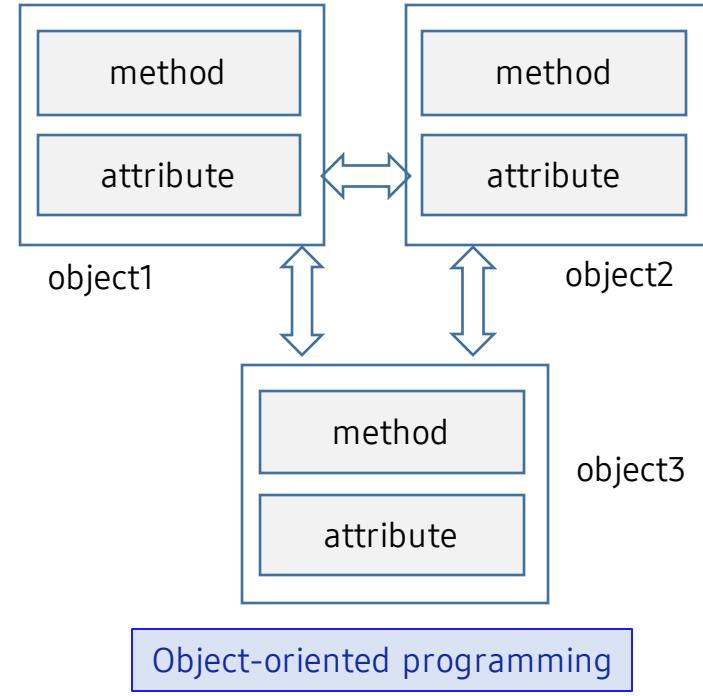
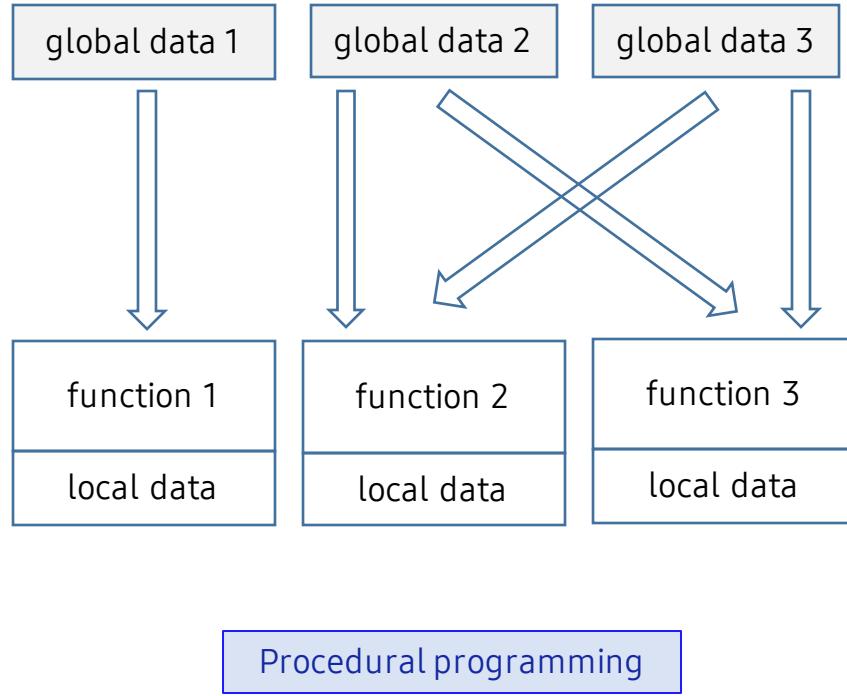
- ▶ Method that generates objects by effectively constructed classes.
- ▶ Classes are constructed to have actions and attributes. OOP applies these classes to programs by creating objects that interact.

#### | OOP requires little amount of maintenance cost in case of development or software updates. Programming nowadays mostly prefer OOP.

## 1. Definition and Concept of Object

### 1.2. Comparison of procedural programming and object-oriented programming

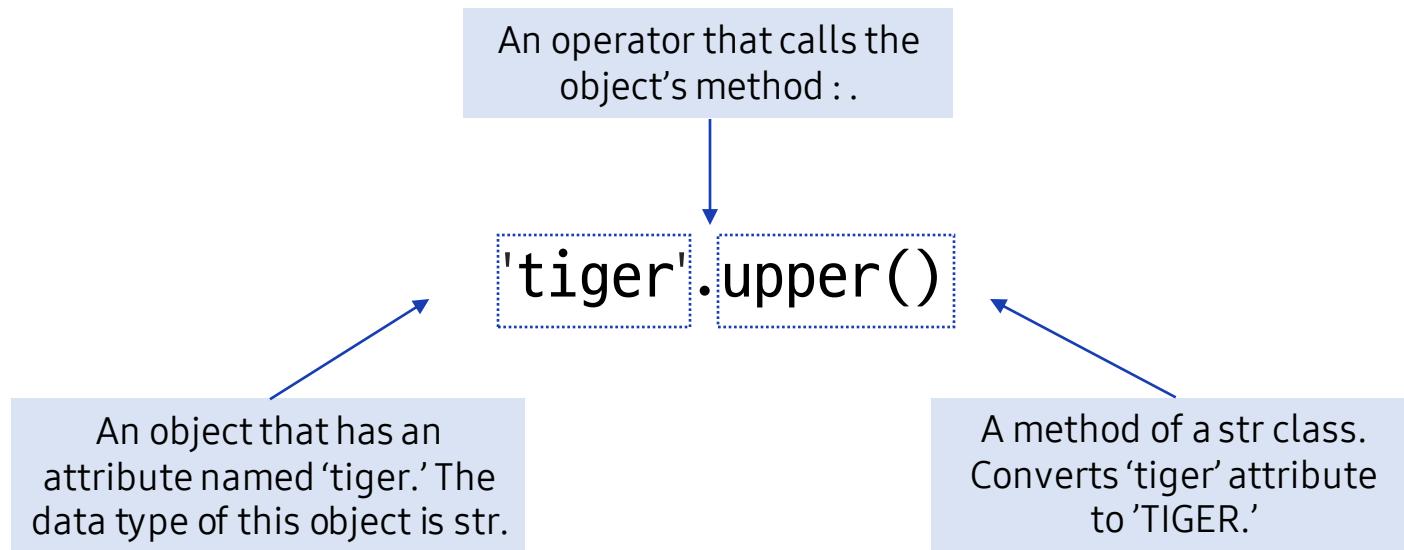
| Below are data flow and function (method) diagram of procedural programming and object-oriented programming.



## 1. Definition and Concept of Object

### 1.3. Syntax for calling Python's objects and methods

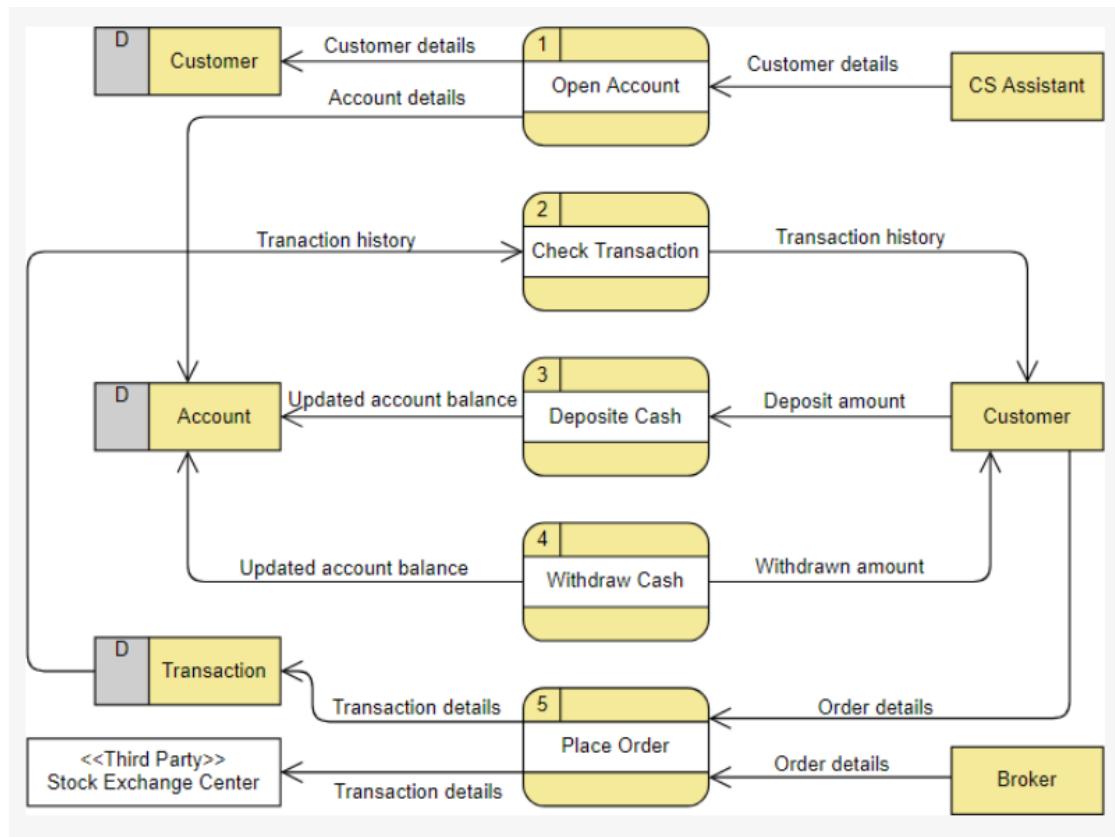
| Use .(dot) operator to call methods for an object.



```
1 'tiger'.upper()  
'TIGER'
```

# 1. Definition and Concept of Object

## 1.4. The limitations of the flowchart



- If a software becomes too complex, expressing its flow with a flowchart becomes inconvenient.
- To solve such a problem, you could use data flow diagram, as shown on the left, to describe the flow of data.

<https://developpaper.com/how-to-create-a-data-flow-diagram-dfd-online/>



## One More Step

- | Everything in Python is an object.
- | Study the code below.
  - ▶ `'tiger'.upper()`
  - ▶ `animals.append('rabbit')`
- | `animals.append('rabbit')`
- | The string 'tiger' and the list animals execute their tasks by using methods such as upper and append. Such components of programming are called objects.
- | Python is an object-oriented programming language and all data used in Python are objects.
- | Then, are n, 100 or 200, which are used in numerical equations or operations, also objects?
  - ▶ `n = 100 + 200`
- | Yes. The above code is equivalent to the following code.
  - ▶ `n = (100).__add__(200)`
- | That is, in the above code, an integer object 100 adds the object 200 by using the `__add__` method.
- | And this object is accessed by the variable n. From this viewpoint, the method calling code and execution procedure of an addition operator are in essence identical to those of `animals.append('rabbit')`.

## 2. Examples of Class Usage in Previous Units

### 2.1. Examples of using classes and objects

- | The strength of OOP is that you can define powerful methods and easily utilize those methods.
- | In the following code, it extracts string s through animals.pop.
- | Various methods such as upper, find are defined in advance for str data types.
- | Likewise, you can utilize a wide variety of methods in OOP.

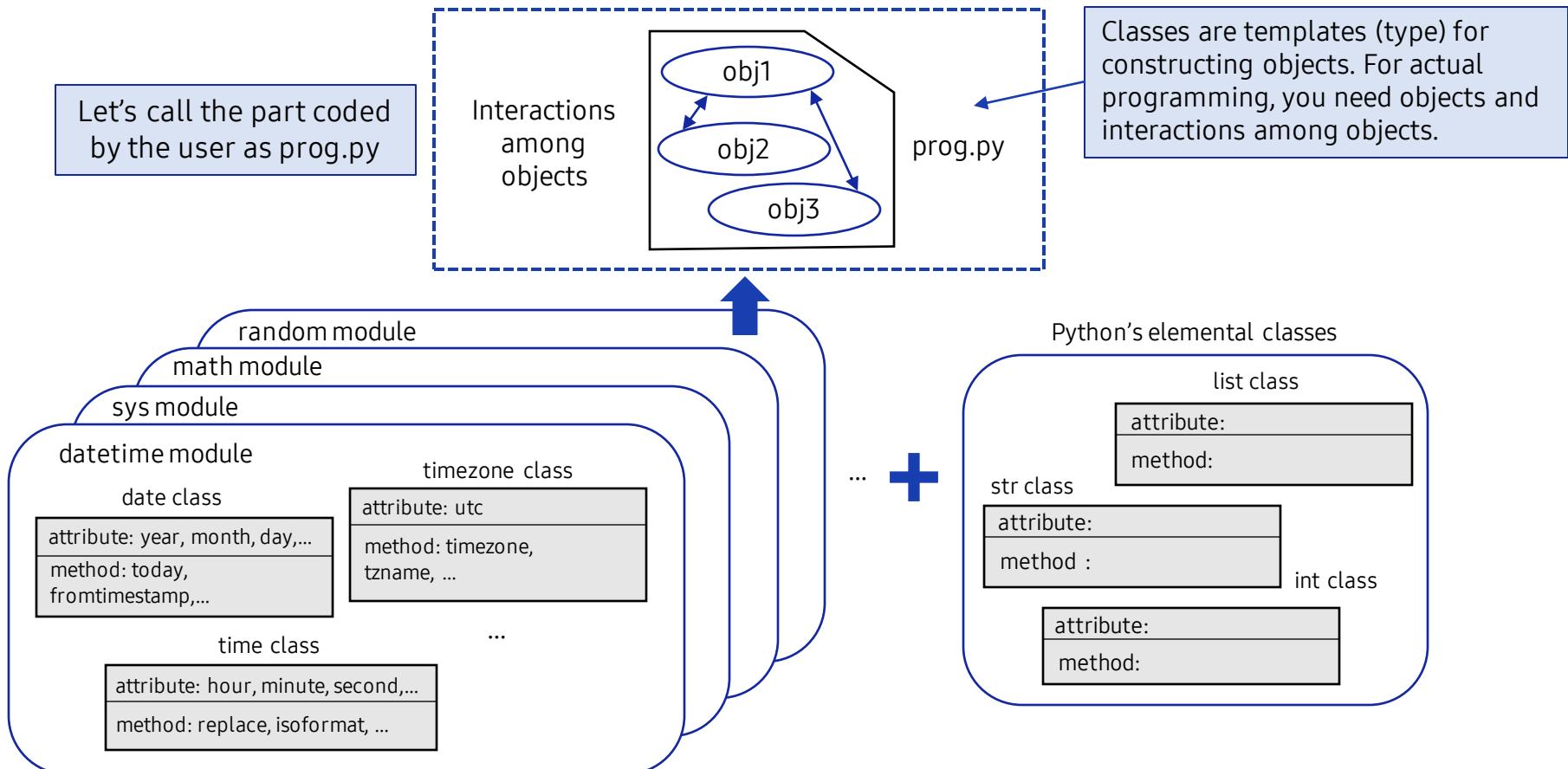
```
1 s = animals.pop()  
2 s  
'cat'
```

```
1 s.upper()  
'CAT'
```

```
1 s.find('a')  
1
```

## 2. Examples of Class Usage in Previous Units

### 2.2. Various functions and classes included in modules



## 2. Examples of Class Usage in Previous Units

### 2.3. Specific functionalities of type and id function

- | type function returns the data type of an object.
- | Each object, when created, has its own id value. You can obtain this id by the id function.

```
1 animals = ['lion', 'tiger', 'cat', 'dog']
2 type(animals)
```

list

```
1 id(animals)
```

2911697529408

```
1 s = 'tiger'
2 type(s)
```

str

```
1 id(s)
```

2911697427632

## 2. Examples of Class Usage in Previous Units

### 2.3. Specific functionalities of type and id function

- | By running the id function, you can verify if variables are referencing the same memory space.
  - ▶ Each object has its own id value.
  - ▶ If the objects are identical, so are the ids.
  - ▶ At times you can reference the same object with variables of different names. Let's check this with the id function.

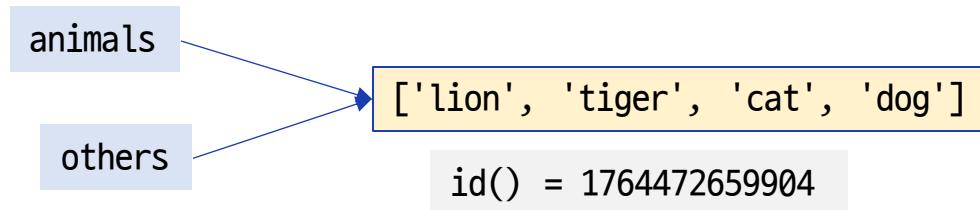
```
1 animals = ['lion', 'tiger', 'cat', 'dog']
2 others = animals
```

```
1 id(animals)
```

1764472659904

```
1 id(others)
```

1764472659904



## 2. Examples of Class Usage in Previous Units

### 2.4. Examples of str class and its methods

| Example : str class has various methods as shown below.

- upper
- lower
- capitalize
- startswith
- strip
- find
- split
- join
- casefold
- center
- count
- endswith
- format
- index
- isalnum
- isalpha
- isdecimal
- islower
- ....

str class alone offers such large number of methods. Python offers, by default, int, list, tuple, dic, date, time and other classes, and their methods (functionalities) are inexhaustible.



## 2. Examples of Class Usage in Previous Units

### 2.5. Examples of int class and its methods

- `__add__`
- `__sub__`
- `__mult__`
- `__truediv__`
- `__mod__`
- `__pow__`
- `__lshift__`
- `__rshift__`, ...

A code executes arithmetic operations (+, -, \*, @, /, //, %, \*\*, <<, >>, &, ^, |) by using these various methods. The various operators we have studied in Unit 3 behave by internally calling these methods.



#### I Understand the difference between `__div__` and `__truediv__` method.

- ▶ The operator `/`, which executes division in Python, was interpreted as `__div__` method in Python 2, but in Python 3, it is interpreted as `__truediv__` method. Thus, `200/100` is interpreted as `(200).__truediv__(100)` in Python 3 and returns a float value.
- ▶ Consequentially, `(200). __div__(100)` produces an error in Python 3.

## 3. Differences Between Instances and Objects

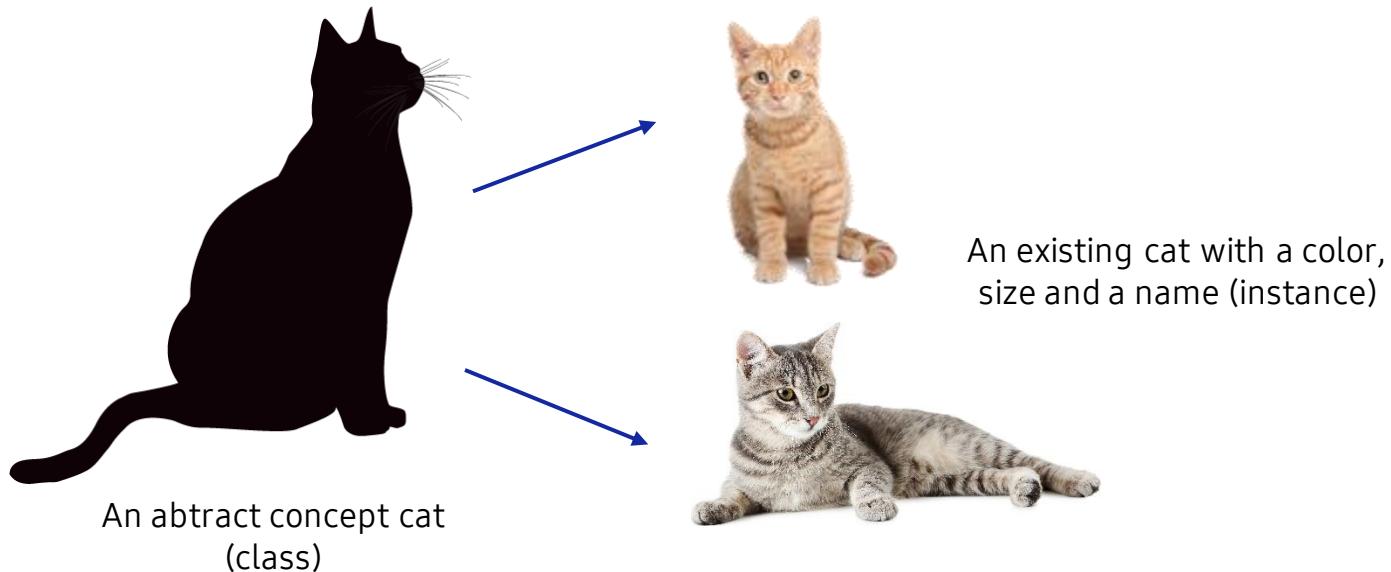
### 3.1. Defining instances and objects

#### | Class

- ▶ An abstract concept that denotes a set of attributes and actions used in programs.

#### | Instance

- ▶ An individual object created from a class. It has specific attribute values.



## 3. Differences Between Instances and Objects

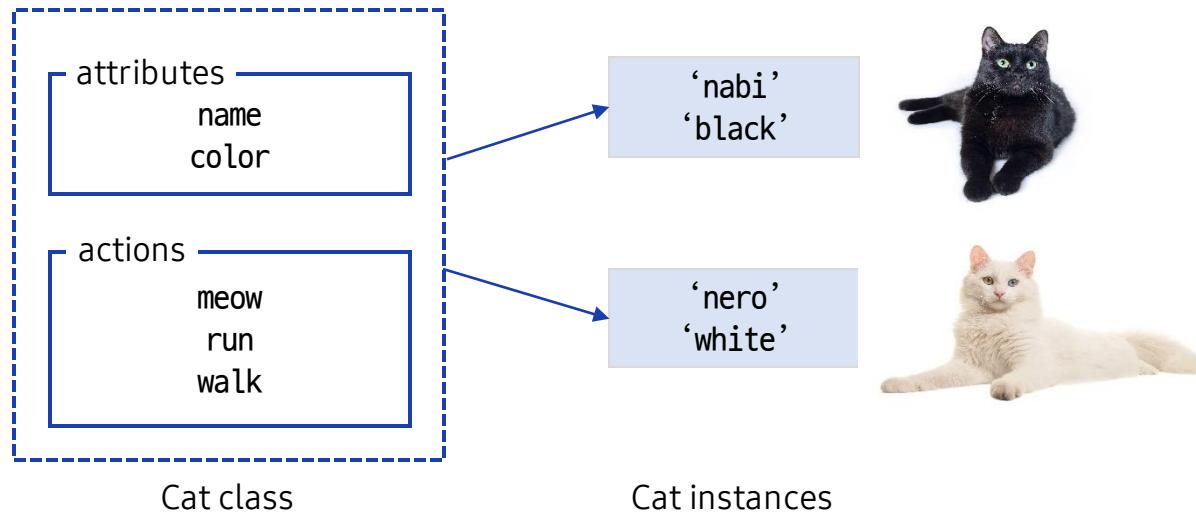
### 3.1. Defining instances and objects

#### | Class

- ▶ A set of attributes and actions used in programs.
- ▶ A plan, template or blueprint of objects.

#### | Instance

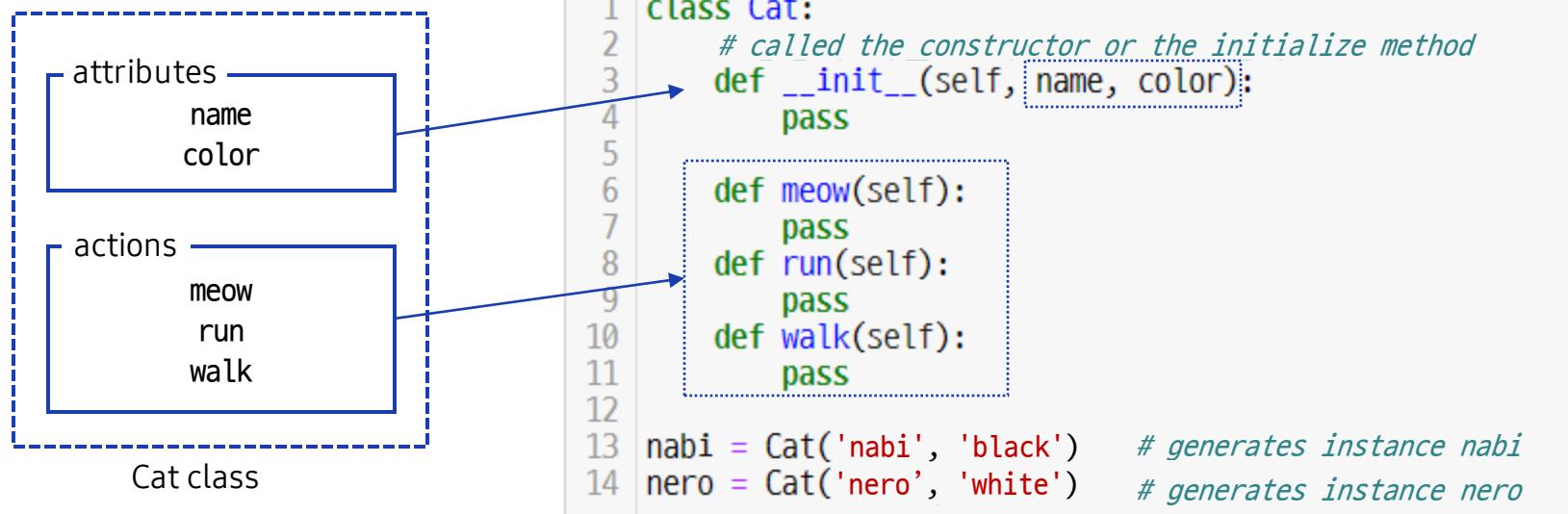
- ▶ An individual object created from a class.
- ▶ Different instances can have different attribute values.



### 3. Differences Between Instances and Objects

#### 3.2. Syntax for class definition

To define a class, you assign a class keyword, write a def statement and indent.



## 3. Differences Between Instances and Objects

### 3.3. Precise distinction between object and instance

- | Many documents mix object and instances as terms of similar concepts.
- | The two terms are similar, but to be more precise, the object can be defined as a material or a thing, while the instance is defined as a material or thing created by a class.
- | For example, from the previous diagram, Cat is a class acting as a framework. The material nabi created by the class is an "object" and at the same time an "instance of the class Cat."
- | To summarize, the following descriptions are correct.
  - 1) All objects of Python has a data type.
  - 2) Python's instances are created from classes.
  - 3) Python's objects are created from classes.
  - 4) Python's class is an object.
  - 5) nabi is an object.
  - 6) The instance of the Cat class is nabi.
  - 7) 100 is an integer type object.

## 4. Declaring Class, and Constructor and self Parameter

### 4.1. Simple syntax for class definition

- | Define a simple class and run a test.
- | Write a keyword named class and write a name of the class.
  - ▶ Then write necessary attributes and methods in accordance with Python syntax.
- | The pass statement is a Python statement that does nothing.
- | Since the class's internal functions are not actualized, use the pass statement.

```
1 class Cat:           # Define the Cat class
2     pass
3
4 nabi = Cat()         # Generate an instance of the Cat
5 print(nabi)
```

```
<__main__.Cat object at 0x000002A5EEAE71F0>
```

#### Line 4,5

- Parentheses are needed to generate a Cat instance.
- When running print(nabi), the id value of the object is printed in hexadecimals.

## 4. Declaring Class, and Constructor and self Parameter

### 4.1. Simple syntax for class definition



TIP

The naming convention for class names in Python are as follows.

- ▶ Use a lowercase letter for the first letter of a function, an object or a variable.
- ▶ Use uppercase letters for the first letter of a class.,
- ▶ If the name has equal or more than two words, uppercase the first letter of second word.
- ▶ When you are defining a protected attribute in an object or a class, begin the first word with an underscore (\_). (A protected attribute means that you should not directly access the attribute from an outer class or object)
- ▶ To use names of reserved words as a name of a variable, add an underscore after the reserved word.
- ▶ A private attribute has a structure in which the names are internally modified so that external access is denied. If you add \_\_ (double underscore), \_class name automatically follows.
- ▶ For special attributes or methods inside Python, add \_\_ in the front and end of a name.

## 4. Declaring Class, and Constructor and self Parameter

### 4.2. Using methods and self

- | Study methods and self through the following example of the Cat class.
- | A method or a member function is a function defined inside a class that is used by the class or class instances.
- | The self parameter of meow method is a variable that references itself. It is mandatory as the first parameter of a method.

```
1 class Cat:  
2     def meow(self):  
3         print('meow~')  
4  
5 nabi = Cat()  
6 nabi.meow()
```

The method which is a function inside Cat class

Creates an object of Cat class through Cat  
Now you can call the method through nabi.meow

meow~

## 4. Declaring Class, and Constructor and self Parameter

### 4.3. Syntax for calling methods

- | Use . (dot) operator to call methods for an instance.
  - ▶ Then write necessary attributes and methods in accordance with Python syntax.
- | The instance nabi below can use the meow method in the Cat class.

```
1 class Cat:  
2     def meow(self):  
3         print('meow meow~~~')  
4  
5 nabi = Cat()  
6 nabi.meow() ← nabi object executes meow  
7 nero = Cat()  
8 nero.meow() ← nero object executes meow  
9 mimi = Cat()  
10 mimi.meow() ← mimi object executes meow  
  
meow meow ~~  
meow meow ~~  
meow meow ~~
```

## 4. Declaring Class, and Constructor and self Parameter

### 4.4. Constructor method `__init__` and `__str__`

#### | Constructor `__init__`

- ▶ A method that assigns a default value to a variable inside an instance when creating an object.
- ▶ Its name is `__int__`.
- ▶ Automatically executed when an object is generated.

#### | `__str__` method

- ▶ A method that displays the information of an object in string format.
- ▶ An example within the Cat class : `__str__` method that displays the name and color of a cat.

## 4. Declaring Class, and Constructor and self Parameter

### 4.5. Class definition and `__init__`, `__str__` special method

- | Add `__str__` method to the Cat class as shown below. This method is automatically called by the print function.
- | `__str__` method returns nicely printable string representation of an object.
  - ▶ If you run `print(nabi)` to print the color and name of the cat, `print(nabi.__str__())` is automatically called.

```
1 class Cat:  
2     def __init__(self, name, color):  
3         self.name = name  
4         self.color = color  
5  
6     # A string expression format of the Cat class  
7     def __str__(self):  
8         return 'Cat(name='+self.name+', color='+self.color+')'  
9  
10  
11 nabi = Cat('nabi', 'black') # generate instance nabi  
12 nero = Cat('nero', 'white') # generate instance nero  
13  
14 print(nabi)  
15 print(nero)
```

Cat(name=nabi, color=black)  
Cat(name=nero, color=white)

Initializes the values by being called when Cat instances are generated.

A special method that defines string expression format of the Cat class

## 4. Declaring Class, and Constructor and self Parameter

### 4.6. Constructor `__init__` and self parameter

- | This method's first parameter self refers to the instances of the current class.
- | The second parameter name and the third parameter color are variables that assign corresponding names and colors, which are instance attributes.

```
class Cat:  
    def __init__(self, name, color):  
        ...  
  
nabi =Cat('nabi', 'black')  
nero =Cat('nero', 'white')  
mimi =Cat('mimi', 'brown')
```

## 4. Declaring Class, and Constructor and self Parameter

### 4.7. When is the `__str__` special method called?

| Generally, `__str__` special methods are automatically called.

- ▶ In the code below, in which the information is printed by the format method with a {} placeholder, `__str__` method is automatically called and executed.

```
1 print('information of nabi : {}' .format(nabi))
```

```
information of nabi : Cat (name=nabi, color=black)
```

## 5. Structure of Method

### 5.1. Definition of method and exemplary code of calling method

```
1 class Cat:  
2     # Called as a constructor or the initialize method  
3     def __init__(self, name, color):  
4         self.name = name    # Generates an instance variable named name  
5         self.color = color  # Generates an instance variable named color  
6  
7     # The method that prints the information of a cat  
8     def meow(self):  
9         print('My name is {}, color is {}, meow meow~~' .format(self.name, self.color))  
10  
11 nabi = Cat('nabi', 'black')      # Generates the instance Nabi  
12 nero = Cat('nero', 'white')      # Generates the instance Nero  
13 mimi = Cat('mimi', 'brown')      # Generates the instance Mimi  
14  
15 nabi.meow()  
16 nero.meow()  
17 mimi.meow()
```

Defines meow of the Cat class via the def keyword.

meow(), a method of the Cat class, can be called by instances of the Cat, Nabi, Nero and Mimi.

My name is Nabi, color is black, meow meow~~

My name is Nero, color is white, meow meow~~

My name is Mimi, color is brown, meow meow~~

## 5. Structure of Method

### 5.2. Instance variables, member variables, fields

- | Study the terms instance variable, member variable and field.
- | Each of these terms denote variables that store distinct attributes of each instances.

self.name = name  
self.color = color  
self is interpreted as such for each instance.

nabi  
instance

nabi.name = 'Nabi'  
nabi.color = 'black'

nero  
instance

nero.name = 'Nero'  
nero.color = 'white'

mimi  
instance

mimi.name = 'Mimi'  
mimi.color = 'brown'

## 5. Structure of Method

### 5.3. Exemplary code for calling methods

```
1 class Cat:  
2     # Called a constructor or the initialize method  
3     def __init__(self, name, color):  
4         self.name = name      # Generates an instance variable named name  
5         self.color = color    # Generates an instance variable named color  
6  
7     # The method that prints the information of a cat  
8     def meow(self):  
9         print('My name is {}, color is {}, meow meow~~'.format(self.name, self.color))  
10  
11 nabi = Cat('nabi', 'black')  # Generates the instance Nabi  
12 nero = Cat('nero', 'white')  # Generates the instance Nero  
13 mimi = Cat('mimi', 'brown')  # Generates the instance Mimi  
14  
15 nabi.meow()  
16 nero.meow()  
17 mimi.meow()
```

Mv name is Nabi. color is black. meow meow~~

My name is Nero, color is white, meow meow~~

My name is Mimi, color is brown, meow meow~~



Line 4,5

- Nabi, Nero, Mimi are instances of the Cat class. Therefore, you can call meow, which is a method of the Cat class with the . (dot) operator.

## 6. Class Variables and Methods

### 6.1. What are instance variables?

- | Study about the instance variable, which is used while defining a class.
- | Since class attributes differ according to individual instances, they can have distinct values.
- | These are called instance variables. Examine the Circle class written below.

```
1 class Circle:  
2     def __init__(self, name, radius, PI):  
3         self.__name = name      # instance variable  
4         self.__radius = radius # instance variable  
5         self.__PI = PI        # instance variable  
6  
7     # Compute the area by multiplying radius**2 to current instance's PI  
8     def area(self):  
9         return self.__PI * self.__radius ** 2  
10  
11  
12 c1 = Circle("C1", 4, 3.14)  
13 print("Area of c1:", c1.area())  
14 c2 = Circle("C2", 6, 3.141)  
15 print("Area of c2:", c2.area())  
16 c3 = Circle("C3", 5, 3.1415)  
17 print("Area of c3:", c3.area())
```

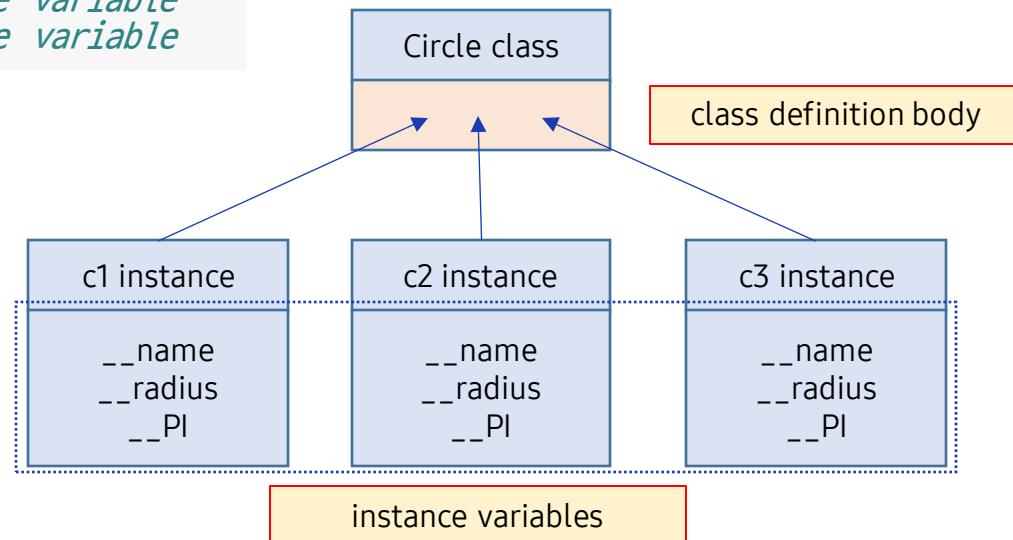
```
Area of c1: 50.24  
Area of c2: 113.076  
Area of c3: 78.53750000000001
```

## 6. Class Variables and Methods

### 6.1. What are instance variables?

- | \_\_name, \_\_radius, \_\_PI in the code below can have different values for each instance.
- | Such variables are called instance variables.
- | The reason for \_\_ is to hide these instance from external access.

```
class Circle:  
    def __init__(self, name, radius, PI):  
        self.__name = name      # instance variable  
        self.__radius = radius  # instance variable  
        self.__PI = PI          # instance variable
```



## 6. Class Variables and Methods

### 6.2. What are class variables?

- | Instance variables may have common attributes that must be shared by the class.
- | In the code example, PI is an attribute of such sort. If this attribute is shared by the individual instances, data overlap will be reduced, and error detection will be easier.
- | If the value of self.\_\_PI of the Circle class were to be constant, c1, c2 and c3 must share the PI value.
- | Here, you use what is called a class variable.

## 6. Class Variables and Methods

### 6.2. What are class variables?

- The class variable of Circle, PI is a variable shared by the instances of this class. The class attributes `__name` and `__radius` are instance variables that are owned by each instances.

```
class Circle:  
    PI = 3.1415      # class variable  
    def __init__(self, name, radius):  
        self.__name  = name  # instance variable  
        self.__radius = radius # instance variable
```

Circle class variable :  
Instances share this variable

Attributes of Circle class :  
Variables owned by each instance

## 6. Class Variables and Methods

### 6.3. Defining class variables

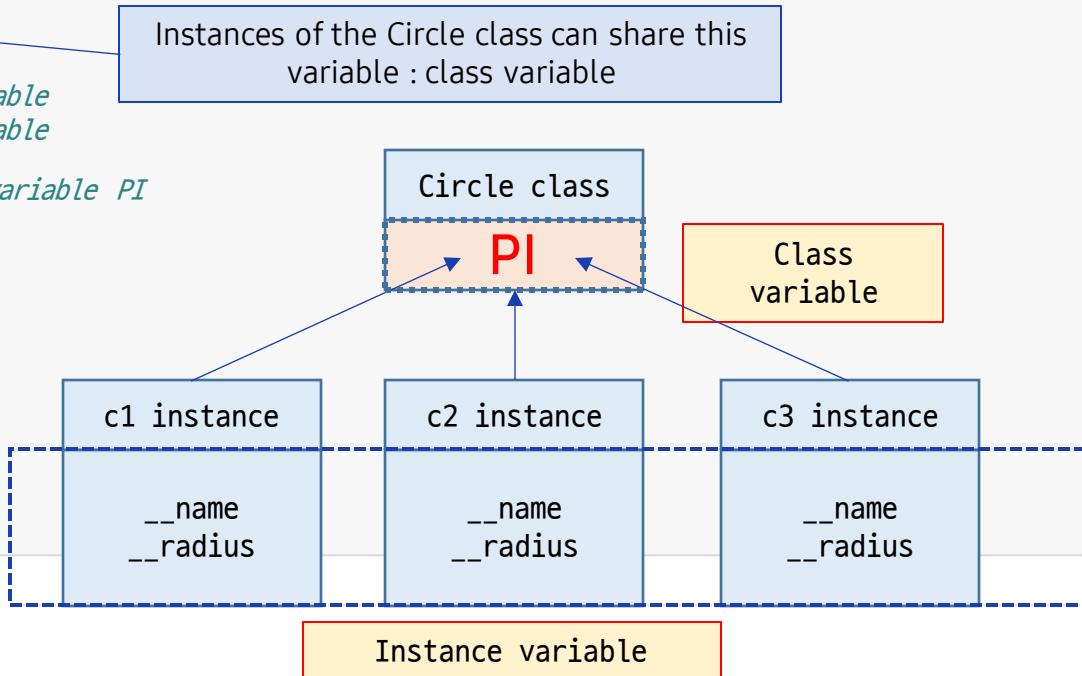
- Class variables are defined outside the methods as shown below. This class variable PI is shared by instances c1, c2 and c3.
- If you need to assign a more accurate value to the PI value, then you only need to modify the variable like PI=3.141592.

```

1 class Circle:
2     PI = 3.1415 # class variable
3     def __init__(self, name, radius):
4         self.__name = name # instance variable
5         self.__radius = radius # instance variable
6
7     #Compute the area by using Circle's class variable PI
8     def area(self):
9         return Circle.PI * self.__radius ** 2
10
11
12 c1 = Circle("C1", 4)
13 print("Area of c1:", c1.area())
14 c2 = Circle("C2", 6)
15 print("Area of c2:", c2.area())
16 c3 = Circle("C3", 5)
17 print("Area of c3:", c3.area())

```

Area of c1: 50.264  
 Area of c2: 113.09400000000001  
 Area of c3: 78.53750000000001



## 6. Class Variables and Methods

### 6.4. Code using class variable PI

- The area of the Circle class in the code below is a method that can be called by instances of the class Circle. This method uses the class variable PI via Circle.PI.

```
1 class Circle:  
2     PI = 3.1415    # class variable  
3     def __init__(self, name, radius):  
4         self.__name = name      # instance variable  
5         self.__radius = radius # instance variable  
6  
7     # Computes the area by using Circle's class variable PI  
8     def area(self):  
9         return Circle.PI * self.__radius ** 2  
10  
11  
12 c1 = Circle("C1", 4)  
13 print("Area of c1:", c1.area())  
14 c2 = Circle("C2", 6)  
15 print("Area of c2:", c2.area())  
16 c3 = Circle("C3", 5)  
17 print("Area of c3:", c3.area())
```

A method of the class Circle.  
Instances refer the class variable through  
Circle.PI in order to use this method.

```
Area of c1: 50.264  
Area of c2: 113.09400000000001  
Area of c3: 78.53750000000001
```

## 6. Class Variables and Methods

### 6.5. \_\_dict\_\_ which displays attribute information of class objects

- | \_\_dict\_\_ attribute is useful for building programs.
- | You can convert the variable of the class object to a dictionary type.
- | You can easily call attribute values by converting them to dictionary type.

```
1 class Circle:  
2     PI = 3.14  
3     def __init__(self, name, radius):  
4         self.name = name  
5         self.radius = radius  
6  
7     c1 = Circle("C1",4)  
8     print("Attributes of c1:", c1.__dict__)  
9     # You can obtain the value with dic [key] format  
10    print("Value of c1's name variable:", c1.__dict__['name'])  
11    print("Value of c1's radius variable:", c1.__dict__['radius'])
```

- Converts the object's type to dictionary type. Then print the value C1 for name.
- Prints 4, value for variable radius.

```
Attributes of c1 :{'name': 'C1', 'radius':4}  
Value of c1's name variable: C1  
Value of c1's radius variable: 4
```

<https://minimilab.tistory.com/58> [MINIMI LAB]

## 6. Class Variables and Methods

### 6.5. \_\_dict\_\_ which displays attribute information of class objects

- | If there is an instance variable with a \_ (underscore), use the following method.
  - ▶ The Circle class has variable \_\_radius and \_\_ name.
- | You can retrieve instance variable's name with . \_\_dict\_\_ attribute of c1. Use this variable as the dictionary's key.
  - ▶ The variable \_\_name can use \_Circle\_\_name as a key.
  - ▶ The variable \_\_ radius can use \_Circle\_\_radius as a key.

```
1 class Circle:  
2     PI = 3.14  
3     def __init__(self, name, radius):  
4         self.__name = name  
5         self.__radius = radius  
6  
7     c1 = Circle("C1",4)  
8     print("Attributes of c1:", c1.__dict__)  
9  
10    # You can obtain the value with dic_[key] format  
11    print("Value of c1's name variable:", c1.__dict__['Circle__name'])  
12    print("Value of c1's radius variable:", c1.__dict__['Circle__radius'])
```

Attributes of C1 :{'\_Circle\_\_name': 'C1', '\_Circle\_\_radius':4}

Value of C1's\_\_name variable: C1

Value of C1's\_\_radius variable: 4

## 7. Class Inheritance

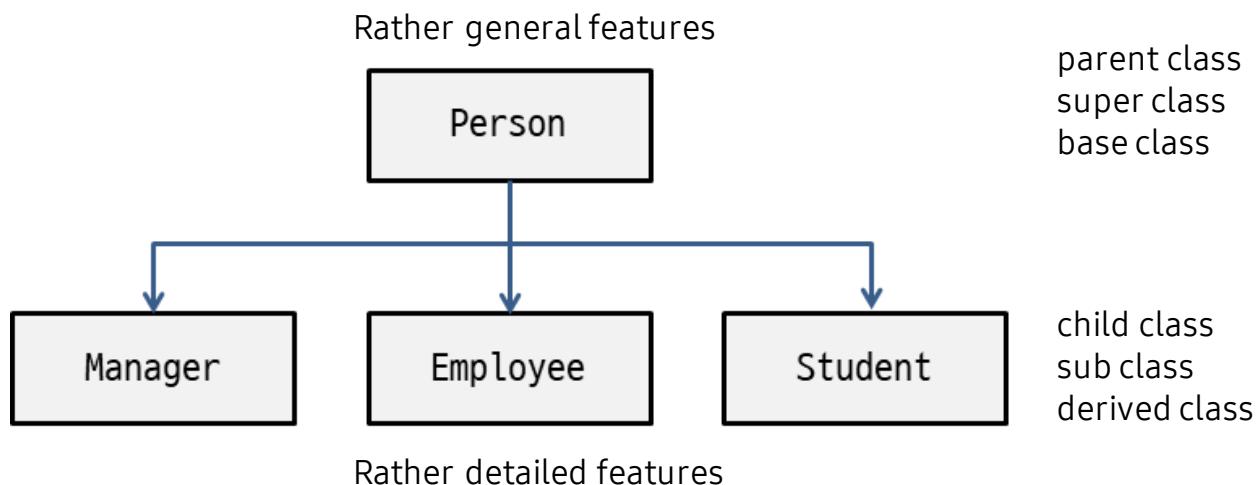
### 7.1. Concept of class and inheritance

- | Inheritance is a technique that allows a user to write a child class that inherits the methods and attributes of a previously constructed class.
- | The reason for inheritance from the parent class is efficient development process.
- | Inheritance is one of the main features of object-oriented programming.
- | Below is a summary of terms before getting more into inheritance.
  - ▶ The upper-level class which hands down methods and attributes is called a parent class, super class or a base class.
  - ▶ The inheriting class is called a child class, sub class or a derived class.

## 7. Class Inheritance

### 7.1. Concept of class and inheritance

- | Such concept of inheritance is shown in the figure below. Person, which is the parent class, has rather general features.
- | The Person will have general features such as age, gender, name etc.
- | However, the child classes Manager, Employee and Student, which inherited attributes from Person, will have both the features of Person as well as more detailed features such as the company name, employee number, pay, student ID number, GPA as attributes.



## 7. Class Inheritance

### 7.2. Syntax of class and inheritance

| How can class inheritance be expressed in Python's syntax? The syntax are written below.

```
1 class A:      # parent class A  
2     statements  
3  
4 class B(A): # child class B with A as the parent  
5     statements
```

- ▶ As shown above, the child class inherits by enclosing the parent class's name with parentheses.
- ▶ Then the child class can inherit the methods and attributes of the parent class A and define new attributes and methods of its own as well.

## 7. Class Inheritance

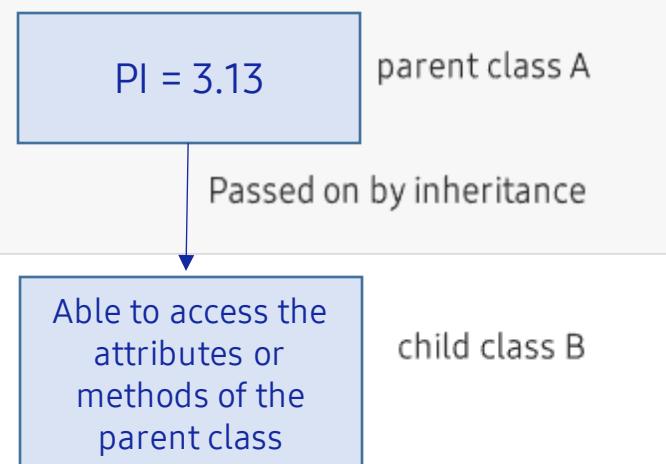
### 7.2. Syntax of class and inheritance

 Focus A child class has access to the parent class's methods or variables.

- In the hierarchy below, A is the parent class, B is the child class.
- Instance a of class A can naturally access PI, a class variable of A.
- However, class B does not have PI as a class variable. Nevertheless, since B is a child class of A, instance b of class B can access the variable of its parent class.

```
1 class A :  
2     PI = 3.14  
3  
4 class B(A):  
5     pass  
6  
7 a = A()    # Generates instance a of class A  
8 b = B()    # Generates instance b of class B  
9 print('a.PI =', a.PI)  
10 print('b.PI =', b.PI)   # b can access the attributes of the parent class
```

```
a.PI = 3.14  
b.PI = 3.14
```



# | Paper coding

**Try to fully understand the basic concept before moving on to the next step.**

**Lack of understanding basic concepts will increase your burden in learning this course, which may make you fail the course.**

**It may be difficult now, but for successful completion of this course we suggest you to fully understand the concept and move on to the next step.**

**Q1.** Construct class Dog and its objects with the functionalities described below.

- a) Has a method named def bark(self): . This method prints a barking sound.
- b) Generates an instance named Dog and refers my\_dog by a command named my\_dog=Dog.
- c) Prints a barking sound with a method named my\_dog.bark()  
“woof woof”

Condition for Execution	woof woof
Time	5 Minutes



Write the entire code and the expected output results in the note.

**Q2.** Define class Dog with the functionalities described below and call instances and methods.

- a) This class Dog has an attribute named name.
- b) Has an initialize method named def \_\_init\_\_(self, name): . This method initializes Dog's name.
- c) Has a method named def bark(self). This method prints a barking sound.
- d) Generates a my\_dog instance that has name 'Bingo' with the command my\_dog=Dog('Bingo')
- e) Prints the following barking sound with the method my\_dog.bark()  
"Bingo : woof woof"

Condition for Execution	Bingo : woof woof
Time	5 Minutes



Write the entire code and the expected output results in the note.

| Let's code

# 1. Class Design and Encapsulation

## 1.1. Importance of encapsulation

- | You assigned an attribute named age for the Cat class.
- | Now construct the object nabi and assign nabi.age=-5. This has no syntax error but has a logical error since a cat's age cannot be negative.
- | How can we reduce such problems?
- | This happened because an attribute was freely modified outside the class.
- | Encapsulation
  - ▶ A way to reduce errors from external access of internal class attributes.
  - ▶ A way that mandates external access to pass designated functions to prevent random manipulation of the attributes.

# 1. Class Design and Encapsulation

## 1.1. Importance of encapsulation

- Limits external modification of methods and variables.
- Protects data.
- Prevents accidental change in values.
- The case of nabi.age = -5 is a logical problem.

```
1 class Cat:  
2     def __init__(self, name, age):  
3         self.name = name  
4         self.age = age  
5  
6     # string expression format of Cat objects  
7     def __str__(self):  
8         return 'Cat(name='+self.name+', age='+str(self.age)+')'  
9  
10 nabi = Cat('nabi', 3) # generates instance nabi  
11 print(nabi)  
12 nabi.age = 4  
13 nabi.age = -5  
14 print(nabi)
```

An abnormal case of age being negative

Cat(name=nabi, age=3)  
Cat(name=nabi, age=-5)

Not a syntax error but a logical problem that occurred because values were not protected.

## 1. Class Design and Encapsulation

### 1.2. Concept of encapsulation

- | A concept map of encapsulation :
- | A functionality that wraps a class's methods and variables to limit external manipulations.



# 1. Class Design and Encapsulation

## 1.3. Code example of encapsulation

This code has been revised to have `set_age`. Due to the conditional statement if `age > 0`: `self.__age = age` does not operate if `age` is negative.

```
1 class Cat:
2     def __init__(self, name, age):
3         self.__name = name
4         self.__age = age
5
6     # string expression format of Cat objects
7     def __str__(self):
8         return 'Cat(name=' + self.__name + ', age=' + str(self.__age) + ')'
9
10    # limits random external access on self.__age and prevents age from being negative
11    def set_age(self, age):
12        if age > 0:
13            self.__age = age
14
15    def get_age(self):
16        return self.__age
17
18 nabi = Cat('nabi', 3) # generates instance nabi
19 print(nabi)
20 nabi.set_age(4)      # approaches age via set_age() method
21 nabi.set_age(-5)    # prevents age from being negative via set_age() method
22 print(nabi)
```

Cat(name=nabi, age=3)  
Cat(name=nabi, age=4)

A way to prevent an illogical situation  
in which the age becomes negative

# 1. Class Design and Encapsulation

## 1.4. Encapsulation and setter

- | Use the if conditional statement so that age is not assigned when age is negative.
- | As shown in the code below, methods that begin with setXXX are called setters.
- | Likewise, those that start with getXXX are called getters and are used to read members' values.
  - ▶ The program will be more stable and safer if the values can only be assigned by setters and attained by getters.
  - ▶ You can safely protect variables inside the members through encapsulation.

Cat class attribute age:  
An attribute that should not be open to external access.

```
class Cat:  
    def __init__(self, name, age):  
        self.__name = name  
        self.__age = age  
    ...  
    def set_age(self, age):  
        if age > 0:  
            self.__age = age
```

setter : accesses the attribute via a method called set\_age

## 2. Object's identity operator : is, is not

### 2.1. Referring identical object via object's id

I Object's identity operator: is, is not

- ▶ The is operator returns True if two instances are identical (that is, if two operands refer the same object) and returns False if the two are not identical. is not does the opposite.

```
1 list_a = [10, 20, 30]                      # list_a which refers a list object
2 list_b = [10, 20, 30]                      # list_b which refers a list object
3 if list_a is list_b:                         # Verifies if the two list objects are identical
4     print('list_a is list_b')
5 else:
6     print('list_a is not list_b')
```

list\_a is not list\_b

## 2. Object's identity operator : is, is not

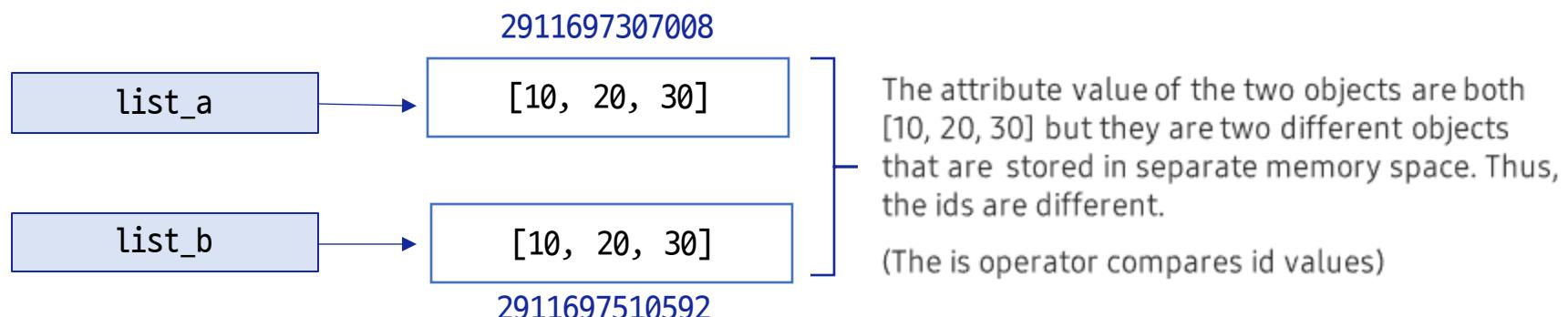
### 2.1. Referring identical object via object's id

| Check the results by printing `id(list_a)` and `id(list_b)`

```
1 print('id(list_a) =',id(list_a),', id(list_b) = ',id(list_b))
```

```
id(list_a) = 2911697307008 , id(list_b) = 2911697510592
```

| The print result shows that `list_a` and `list_b` have different id values.



`list_a is list_b = False`

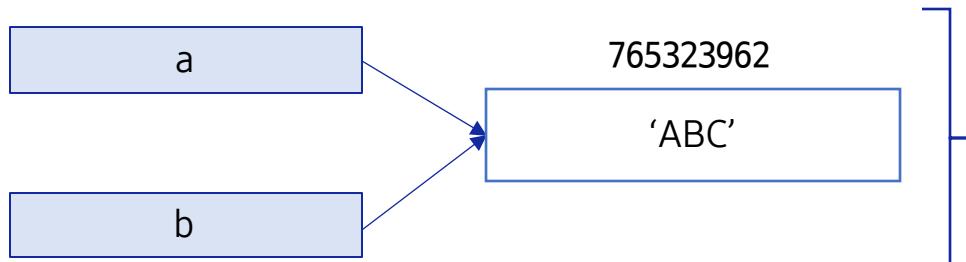
## 2. Object's identity operator : is, is not

### 2.1. Referring identical object via object's id

- | a and b in the code below refers the string 'ABC'. Here a and b refers the same object unlike the lists. Why?
- | Since strings are immutable, this approach doesn't have any issues.

```
1 a = 'ABC'                      # Variable a that refers a string object
2 b = 'ABC'                      # Variable b that refers a string object
3 if a is b:                     # Compares if the string object a and b are identical
4     print('a is b')             # The string object a and b refer the same object
5 else:
6     print('a is not b')
```

a is b



Python's str object stores string objects in a storage table. Then, if it assigns the same string to a and b, it refers a single store location to save memory space.

a and b refer the same object.

## 3. Class and special methods

### 3.1. Necessity of special methods

| Construct a class named Vector 2D that expresses two-dimensional vectors and observe the necessity of special methods in this class. Why does `v1 + v2` produce an error in the code below?

```
1 class Vector2D:  
2     def __init__(self, x, y):  
3         self.x = x  
4         self.y = y  
5  
6 v1 = Vector2D(30, 40)  
7 v2 = Vector2D(10, 20)  
8 v3 = v1 + v2          # + operator has not been defined in Vector2D : prints error  
9 print('v1 + v2 = ',v3)
```

---

```
TypeError                                     Traceback (most recent call last)  
<ipython-input-37-e9d6e99fc219> in <module>  
      6 v1 = Vector2D(30, 40)  
      7 v2 = Vector2D(10, 20)  
----> 8 v3 = v1 + v2          # + operator has not been defined in Vector2D : prints error  
      9 print('v1 + v2 = ',v3)
```

**TypeError**: unsupported operand type(s) for +: 'Vector2D' and 'Vector2D'

## 3. Class and special methods

### 3.1. Necessity of special methods

- The reason for the error is because the code does not state how the class will operate additions within.
- Define a method named add as follows and add the x component and y component of the two vectors. Then, return a Vector2D that has values of these components as the initial value.

```
1 class Vector2D:  
2     def __init__(self, x, y):  
3         self.x = x  
4         self.y = y  
5     def __str__(self):  
6         return "({}, {})".format(self.x, self.y)  
7     def add(self, other):  
8         return Vector2D(self.x + other.x, self.y + other.y)  
9  
10 v1 = Vector2D(30, 40)  
11 v2 = Vector2D(10, 20)  
12 v3 = v1.add(v2)  
13 print('v1.add(v2) = ', v3)
```

v1.add(v2) = (40, 60)



#### Line 12

- Prints the sum of two vectors by using the add method named v1.add(v2).
- Using operators such as +, - will be more convenient than the add method.

## 3. Class and special methods

### 3.2. Defining special methods

- | `__add__` returns the sum of two vectors, and `__sub__` returns the difference between two vectors.
- | `v1 + v2` does what is identical to calling `v1.__add__(v2)`. Since `v1 + v2` is more intuitive, we will use the `+` operator instead of the `__add__` special method.
- | Likewise, Python enables appending new functionalities to the original functionality of `+`.

```
1 class Vector2D:  
2     def __init__(self, x, y):  
3         self.x = x  
4         self.y = y  
5     def __add__(self, other):  
6         return Vector2D(self.x + other.x, self.y + other.y)  
7     def __sub__(self, other):  
8         return Vector2D(self.x - other.x, self.y - other.y)  
9     def __str__(self):  
10        return "({}, {})".format(self.x, self.y)  
11  
12 v1 = Vector2D(30, 40)  
13 v2 = Vector2D(10, 20)  
14 v3 = v1 + v2  
15 print('v1 + v2 =', v3)  
16 v4 = v1 - v2  
17 print('v1 - v2 =', v4)
```

v1 + v2 = (40, 60)

v1 - v2 = (20, 20)

## 3. Class and special methods

### 3.2. Defining special methods

```
1 class Vector2D:  
2     def __init__(self, x, y):  
3         self.x = x  
4         self.y = y  
5     def __add__(self, other):  
6         return Vector2D(self.x + other.x, self.y + other.y)  
7     def __sub__(self, other):  
8         return Vector2D(self.x - other.x, self.y - other.y)  
9     def __str__(self):  
10        return "({}, {})".format(self.x, self.y)  
11  
12 v1 = Vector2D(30, 40)  
13 v2 = Vector2D(10, 20)  
14 v3 = v1 + v2  
15 print('v1 + v2 =', v3)  
16 v4 = v1 - v2  
17 print('v1 - v2 =', v4)
```

v1 + v2 = (40, 60)

v1 - v2 = (20, 20)

#### Line 5-6

- Defines the special method `__add__` to return the sum of two vectors

## 3. Class and special methods

### 3.2. Defining special methods

```
1 class Vector2D:  
2     def __init__(self, x, y):  
3         self.x = x  
4         self.y = y  
5     def __add__(self, other):  
6         return Vector2D(self.x + other.x, self.y + other.y)  
7     def __sub__(self, other):  
8         return Vector2D(self.x - other.x, self.y - other.y)  
9     def __str__(self):  
10        return "({}, {})".format(self.x, self.y)  
11  
12 v1 = Vector2D(30, 40)  
13 v2 = Vector2D(10, 20)  
14 v3 = v1 + v2  
15 print('v1 + v2 =', v3)  
16 v4 = v1 - v2  
17 print('v1 - v2 =', v4)
```

v1 + v2 = (40, 60)

v1 - v2 = (20, 20)

#### Line 14-15

- Now you can use v1 + v2 to call \_\_ add\_\_
- The inner behavior of v1 + v2 is identical that of calling the method v1.\_\_ add(v2)

## 3. Class and special methods

### 3.3. Various kinds of special methods

| Python's main operators and special methods that correspond to these operators

operator	Special method	functionality
$x + y$	<code>__add__(self, other)</code>	Computes the sum of x and y
$x - y$	<code>__sub__(self, other)</code>	Computes the difference between x and y.
$x * y$	<code>__mul__(self, other)</code>	Computes the multiplication of x and y.
$x ** y$	<code>__pow__(self, other)</code>	Computes x power y.
$x / y$	<code>__truediv__(self, other)</code>	Computes x divided by y.
$x // y$	<code>__floordiv__(self, other)</code>	Computes the quotient of x divided by y.
$x \% y$	<code>__mod__(self, other)</code>	Computes the remainder of x divided by y.
$+x$	<code>__pos__(self)</code>	Gives x.
$-x$	<code>__neg__(self)</code>	Gives x negative.

Note : Python 2 uses `__div__` instead of `__truediv__`

## 3. Class and special methods

### 3.3. Various kinds of special methods

| Python's main operators and special methods that correspond to these operators

operator	Special method	functionality
<code>x &lt; y</code>	<code>__lt__(self, other)</code>	Is x smaller than y?
<code>x &lt;= y</code>	<code>__le__(self, other)</code>	Is x smaller than or equal to y?
<code>x &gt;= y</code>	<code>__ge__(self, other)</code>	Is x larger than or equal to y?
<code>x &gt; y</code>	<code>__gt__(self, other)</code>	Is x larger than y?
<code>x == y</code>	<code>__eq__(self, other)</code>	Are x and y equal?
<code>x != y</code>	<code>__ne__(self, other)</code>	Are x and y not equal?

## 3. Class and special methods

### 3.4. Relationship between built-in functions and special methods

- | Python also has numerous built-in functions such as len, float, int, str, abs, hash, iter etc.
- | These built\_in functions are ways to call special methods such as \_\_len\_\_, \_\_float\_\_, \_\_int\_\_, \_\_str\_\_m, \_\_abs\_\_, \_\_hash\_\_, \_\_iter\_\_ etc.
- | That is, len built-in function operates by calling the \_\_len\_\_ special method.

## 4. Meaning of object, reference and assignment operator

### 4.1. Meaning of object, reference and assignment operator

- | Let's study the roles of objects, references and assignment operators.
- | Variable is just a name given to an object in the memory
- | In the code below, it approaches the number object 100 via the variable n by using n = 100 statement.
- | It can be verified by id function, we can see id(100) is same as id(n).

```
1 n = 100  
2 id(100)
```

140719510270864

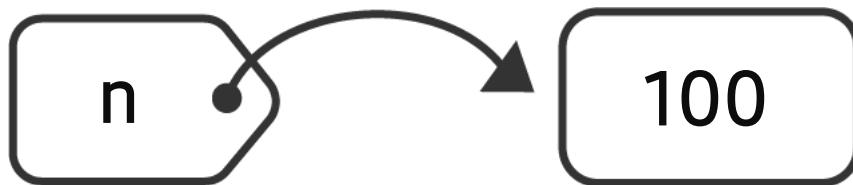
```
1 id(n)
```

140719510270864

## 4. Meaning of object, reference and assignment operator

### 4.1. Meaning of object, reference and assignment operator

- | If there is an object with the value 100, the reference variable n for the object also refers the identical object 100.
- | You can access the object 100 via variable n.
- | Another variable m can have access to the object of variable 100 by using an assignment operator.



id: 140719510270864

## 4. Meaning of object, reference and assignment operator

### 4.1. Meaning of object, reference and assignment operator

- | Assignment operator = refers and re-refers the object.
- | The two variables refer the same object.

```
1 n = 100  
2 m = n  
3 id(n)
```

140719510270864

```
1 id(m)
```

140719510270864

## 4. Meaning of object, reference and assignment operator

### 4.1. Meaning of object, reference and assignment operator

- | How can you refer object with immutable attributes?
- | Integers, real numbers, strings, Booleans and tuples are immutable objects. In the case below, m and n refer to the same object 100.
  - ▶ n = 100
  - ▶ m = 100 # The two are identical.
- | Such mechanism allows efficient usage of memory.

```
1 n = 100
2 m = 100
3 print(id(n))
4 print(id(m))
```

140736051753872  
140736051753872

 Line 3-4

- Since m and n both refer the number 100, the ids of the two objects are identical.

## 4. Meaning of object, reference and assignment operator

### 4.2. Assignment operator means referring other objects

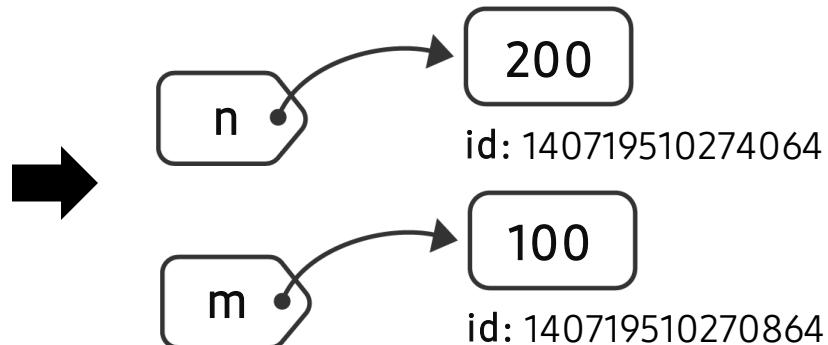
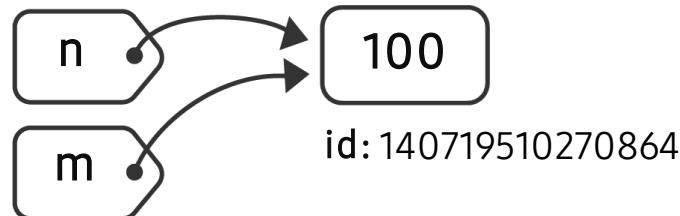
- | Variable n and m which refer an object can also refer new objects.
- | The initial state of n refers to 100, and so does m via  $m = n$ .
- | Then, m and n refer another variable via  $n = 200$ .

```
1 n = 100  
2 m = n  
3 n = 200  
4 id(n)
```

140719510274064

```
1 id(m)
```

140719510270864



## 4. Meaning of object, reference and assignment operator

### 4.2. Assignment operator means referring other objects

- | Object assignment method and operation of  $n = n + 1$  operator.
- | The code below shows that the  $n$  newly assigned via  $n = n + 1$  and the previous  $n$  refer different objects.

```
1 n = 100  
2 id(n)
```

140719510270864

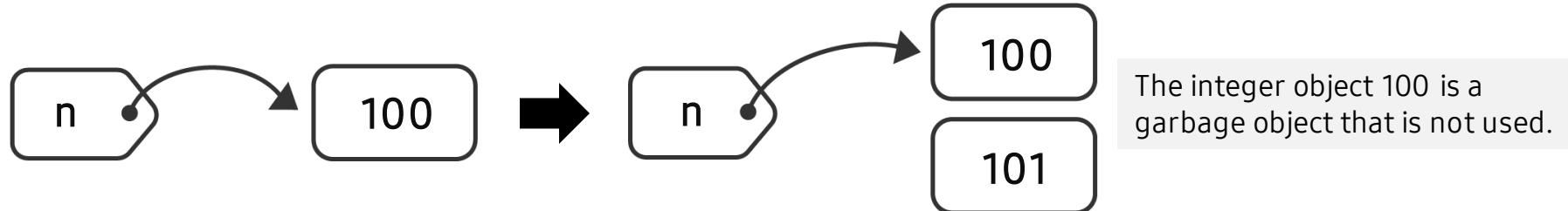
```
1 n = n + 1  
2 id(n)
```

140719510270896

## 4. Meaning of object, reference and assignment operator

### 4.2. Assignment operator means referring other objects

- The integer object referred by n is defined as an immutable variable.

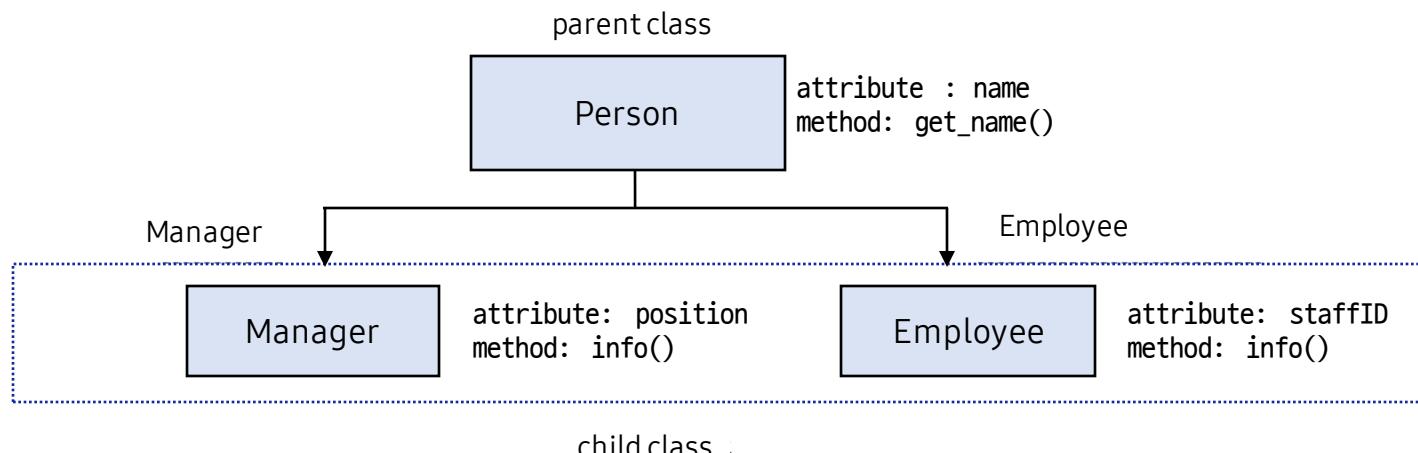


- This causes waste of memory space because there is no longer a variable that refers this object.
- Objects that are no longer viable for reference due to disappearance of reference are called garbage.
- A periodic memory management procedure that removes garbage memories is called garbage collection.

## 5. Class and Inheritance

### 5.1. Class of hierarchical structure

- | Examine the class with hierarchical structure below.
- | You can construct class Manager that means manager and class Employee that means employee by inheriting the class Person, which is the parent class.
- | Since managers and employees are both human-beings, the two child classes should contain the name attribute and get\_name() method from the Person class.
- | If the class Manager and the class Employee inherit the class Person, the two classes will contain the member variables and member functions of the class Person without any extra coding.



## 5. Class and Inheritance

### 5.2. Constructing class of hierarchical structure

- | Define the general structure of the parent class as follows.
- | This class has name attribute and get\_name method.

```
class Person:  
    def __init__(self, name):  
        self.name = name  
    def get_name(self):  
        return self.name
```

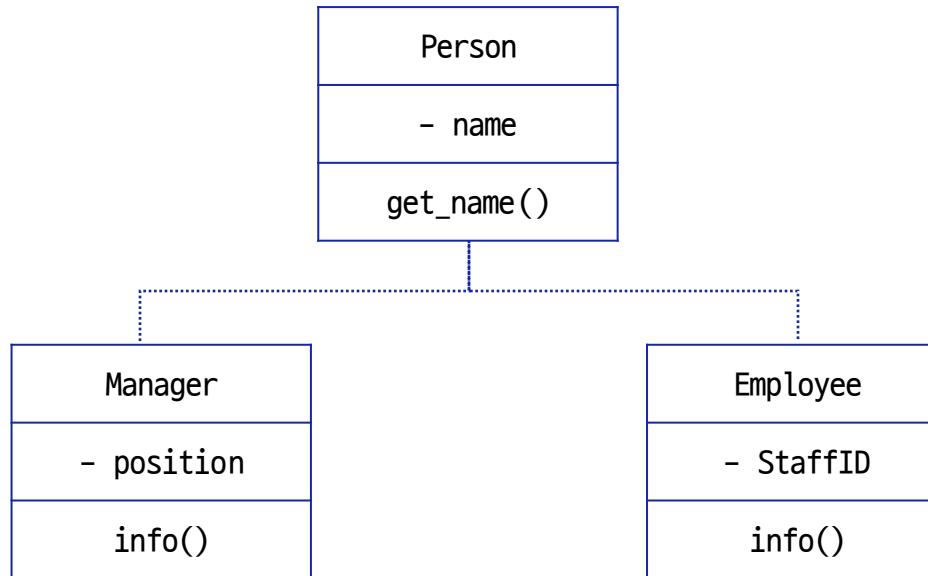
- | The Class Employee and the class Manager which inherited from the class Person can be defined with such syntax.

```
class Manager(Person):  
    ...  
class Employee(Person):  
    ...
```

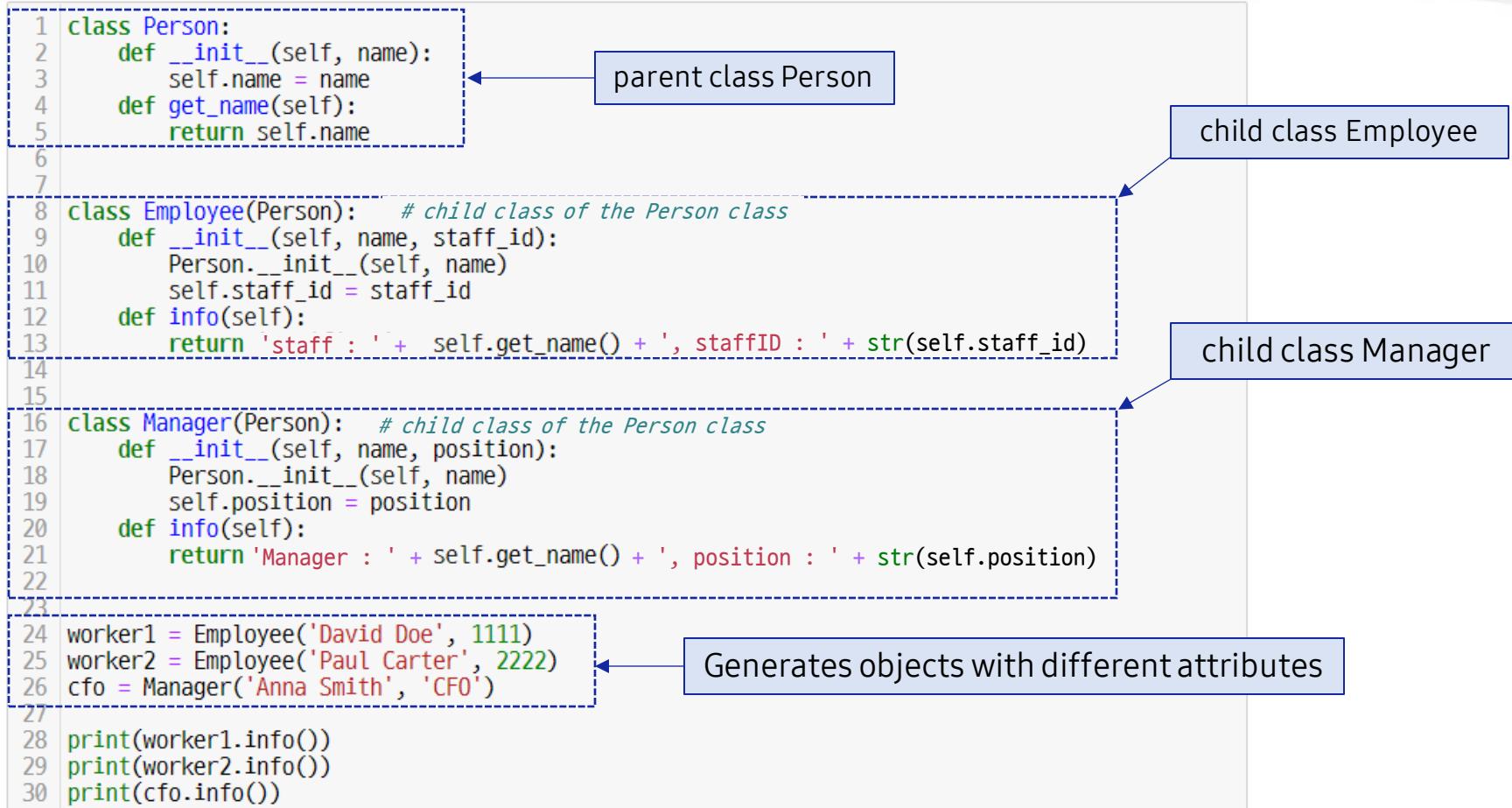
## 5. Class and Inheritance

### 5.2. Constructing class of hierarchical structure

| The next slide shows the code that implemented the hierarchical structure of this class.



You can use the `name` attribute and the `get_name()` method via inheritance.



```
Employee : David Doe, staffID : 1111
Employee : Paul Carter, staffID : 2222
Manager : Anna Smith, position : CFO
```

# | Pair programming



# Pair Programming Practice



## | Guideline, mechanisms & contingency plan

Preparing pair programming involves establishing guidelines and mechanisms to help students pair properly and to keep them paired. For example, students should take turns “driving the mouse.” Effective preparation requires contingency plans in case one partner is absent or decides not to participate for one reason or another. In these cases, it is important to make it clear that the active student will not be punished because the pairing did not work well.

## | Pairing similar, not necessarily equal, abilities as partners

Pair programming can be effective when students of similar, though not necessarily equal, abilities are paired as partners. Pairing mismatched students often can lead to unbalanced participation. Teachers must emphasize that pair programming is not a “divide-and-conquer” strategy, but rather a true collaborative effort in every endeavor for the entire project. Teachers should avoid pairing very weak students with very strong students.

## | Motivate students by offering extra incentives

Offering extra incentives can help motivate students to pair, especially with advanced students. Some teachers have found it helpful to require students to pair for only one or two assignments.



# Pair Programming Practice



## Prevent collaboration cheating

The challenge for the teacher is to find ways to assess individual outcomes, while leveraging the benefits of collaboration. How do you know whether a student learned or cheated? Experts recommend revisiting course design and assessment, as well as explicitly and concretely discussing with the students on behaviors that will be interpreted as cheating. Experts encourage teachers to make assignments meaningful to students and to explain the value of what students will learn by completing them.

## Collaborative learning environment

A collaborative learning environment occurs anytime an instructor requires students to work together on learning activities. Collaborative learning environments can involve both formal and informal activities and may or may not include direct assessment. For example, pairs of students work on programming assignments; small groups of students discuss possible answers to a professor's question during lecture; and students work together outside of class to learn new concepts. Collaborative learning is distinct from projects where students "divide and conquer." When students divide the work, each is responsible for only part of the problem solving and there are very limited opportunities for working through problems with others. In collaborative environments, students are engaged in intellectual talk with each other.

## Q1.

Construct class Student that has following functionalities.

This student took quizzes for English, mathematics and science, and the quiz scores are given as inputs.  
Generate instances by using this class. This class has the following attributes and actions.

### attributes

name : student name stored in string type.  
student\_id : student ID such as s2020001 stored in 8-digit integers.  
eng\_quiz : student's English quiz core stored in list format.  
math\_quiz : student's math quiz score stored in list format. science\_quiz : student's science quiz score stored in list format.

### actions(methods)

`__init__` : initializes with the student's name and studentID.  
`__str__` : returns the student's name, studentID and quiz scores as strings  
  
`set_eng_quiz` : sets the student's English quiz.  
`set_math_quiz` : sets the student's mathematics quiz.  
`set_science_quiz` : sets the student's science quiz.  
  
`get_name` : returns the student's name  
`get_student_id` : returns the studentID  
`get_eng_quiz` : returns the student's English quiz  
`get_math_quiz` : returns the student's math quiz  
`get_science_quiz` : returns the student's science quiz  
`get_total_score` : returns the student's total quiz score  
`get_avg_score` : returns the student's average quiz score

**Q1.**

Construct class Student that has following functionalities.

This student took quizzes for English, mathematics and science, and the quiz scores are given as inputs.

Generate instances by using this class. This class has the following attributes and actions.

## Output example

```
Enter the student's name : David Doe
Enter the student's ID : 20213093
Enter the student's English quiz score : 90
Enter the student's mathematics quiz score : 95
Enter the student's science quiz score : 100
Name : David Doe, ID : 20213093
English quiz score : 90, Mathematics quiz score : 95
Science quiz score : 100,
Total : 285, Average : 95.0
```

A photograph of a person working at a desk. They are wearing an orange long-sleeved shirt and are holding a brown paper coffee cup with a black lid in their left hand. Their right hand is on a black computer keyboard. In front of them is a stack of papers or books. On the far left, a portion of a laptop screen is visible. The background shows a window with vertical blinds and a dark wall.

# End of Document



# Together for Tomorrow! Enabling People

Education for Future Generations

©2022 SAMSUNG. All rights reserved.

Samsung Electronics Corporate Citizenship Office holds the copyright of book.

This book is a literary property protected by copyright law so reprint and reproduction without permission are prohibited.

To use this book other than the curriculum of Samsung Innovation Campus or to use the entire or part of this book, you must receive written consent from copyright holder.