Python:

From Beginner to **Intermediate**

Sunglok Choi, Assistant Professor, Ph.D.

Computer Science and Engineering Department, SEOULTECH

sunglok@seoultech.ac.kr | https://mint-lab.github.io/

Python: Basic

- **Why Python? What is Python? How to Use Python**

- **Data Types**
  - Dynamically-typed → no type specifier
  - Built-in compound data types: **str**, **tuple***, **list***, **set***, **dict*** (*support heterogenous data types)
- **Operators**
  - More natural (e.g. **&&** → **and**, condition **?** A **:** B → A **if** condition **else** B)
- **Flow Control**
  - Easier access to elements in compound data with **for** statement
- **Function Definition**
  - Multiple return values (as a **tuple**)
- **Object-oriented Programming**
  - Dynamically-typed → no definition for member variables

My Comments for Better Python Programming

1. Take advantages of Python itself (a.k.a. *Pythonic*)

- e.g. Swap using unpacking

```
temp = a
a = b          VS.      (a, b) = (b, a)
b = temp
```

- References

  - Code Style, The Hitchhiker's Guide to Python

  - Write More Pythonic Code, Real Python

  - PEP 8 – Style Guide for Python Code, Python

2. Utilize the exiting libraries (a.k.a. *Don't reinvent the wheel*) **and master them if they are useful**

- Problem #1) Too many libraries

  - Search your keywords in **Google**/Github (with *python*), PyPI, and …

- Problem #2) A few documents and examples

  - Select a popular one (if possible)

  - Search your issues in **Google** (or read and analyze the source codes)

Data Types

- **Numbers**

  - **int**: Integers with <u>an unlimited range</u>

    ```
    a = 329
    ```

  - **float**: Double-precision (64-bit) floating-point numbers

    ```
    a = 3. # Same with 'a = 3.0'
    b = 3.29
    ```

  - **bool**: Boolean values (False or True)

    ```
    a = False
    b = (3 == 3.) # Check this result
    ```

  Note) Dynamically-typed → no type specifier

Data Types

- **Numbers**

  - **int**: Integers with <u>an unlimited range</u>

    ```
    a = 329         # Decimal number
    b = 0b101001001 # Binary number
    c = 0o511       # Octal number
    d = 0x149       # Hexadecimal number
    ```

  - **float**: Double-precision (64-bit) floating-point numbers

    Note) Python <u>does not support</u> single-precision (32-bit) floating-point numbers <u>by itself</u>.

  - **complex**: Complex numbers with imaginary and real float numbers

    ```
    a = 3.29 + 8.2j
    ```

  - **Built-in constants**: False, True, None, …

    - None: Not available (<u>N/A</u>), not determined, … (e.g. when your function does not `return` a value)
    - Note) **None == False** # False

        ```
        type(None)    # NoneType
        ```

Data Types

- **String**
  - **str**: A text encoded in <u>Unicode</u>

    name **=** 'Choi'   or   name **=** "Choi"

    For multi-line text

    name = 'Sunglok\nChoi'   or   name **=** """Sunglok
                                  Choi""" # Also used for a block comment (~ /* ... */ in C++)

    Note) String is a built-in type, not in a library (~ std::string in C++).

Data Types

- **String**
  - **String formatting [Real Python] [Format Specifiers]**

    ```
    prof_dict = {'name': 'Choi', 'room_no': 327, 2021: True}
    ```

    - % operator (~ printf in C): Mandatory type/format specifiers

      ```
      'My name is %s and my room is %d' % ('Choi', 327)

      'My name is %s and my room is %d' % ['Choi', 327]              # Error!

      'My name is %(name)s and my room is %(room_no)04d' % prof_dict # 327 -> 0327
      ```

    - str.format function (~ std::cout in C++): Optional type/format specifiers

      ```
      'My name is {} and my room is {}'.format('Choi', 327)

      'My name is {1} and my room is {0}'.format(327, 'Choi')

      'My name is {name} and my room is {room_no}'.format(name='Choi', room_no=327)

      'My name is {name} and my room is {room_no}'.format('Choi', 327)      # Error!

      'My name is {name} and my room is {room_no:04d}'.format(**prof_dict) # 327 -> 0327
      ```

Data Types

- **String**
  - **String formatting [Real Python] [Format Specifiers]**

    ```
    name = 'Choi'

    room_no = 327 # Need to print the next room as 0328
    ```

    - % operator (~ printf in C): Mandatory type/format specifiers

      ```
      'My name is %s and my next room is %04d' % (name, room_no + 1)
      ```

    - str.format function (~ std::cout in C++): Optional type/format specifiers

      ```
      'My name is {} and my next room is {:04d}'.format(name, room_no + 1)
      ```

    - \+ operator: Unnecessary to match argument position

      ```
      'My name is ' + name + ' and my next room is ' + str(room_no + 1)
      ```

    - f-string (Python 3.6+): Concise (without + operators) and no type conversion (to str)

      ```
      f'My name is {name} and my next room is {room_no + 1}'
      f'My name is {name} and my next room is {room_no + 1:04d}'
      ```

Data Types

- **String**
  - **String trimming**

```python
full_name = '\t Sunglok Choi \n'

full_name.strip()        # 'Sunglok Choi'

full_name.strip('\t\n') # ' Sunglok Choi '

full_name.lstrip()       # 'Sunglok Choi \n'

full_name.rstrip()       # '\t Sunglok Choi'
```

Data Types

- **String**
  - **String splitting**

```python
prof = 'Choi, 327, 1'

prof.split(',')        # ['Choi', ' 327', ' 1']
prof.split(', ')       # ['Choi', '327', '1']
prof.split('|')        # ['Choi, 327, 1']
prof.partition(', ')   # ('Choi', ', ', '327, 1') / (~ std::strtok in C)


profs = 'Choi, 327, 1\nKang, 328, 1'

profs.split(', ')      # ['Choi', '327', '1\nKang', '328', '1']
profs.splitlines()     # ['Choi, 327, 1', 'Kang, 328, 1']
```

Data Types

- **String**

  – **String matching**: Whether both are **same** or not

  – **String searching**: Whether the <u>given text</u> **contains** the <u>query text</u> or not

    - Also known as *substring matching*

```python
profs = [
    'My name is Choi and my E-mail is sunglok@seoultech.ac.kr.',
    'My name is Kim and my e-mail address is jindae.kim@seoultech.ac.kr.'
]
for line in profs:
    print('e-mail' == line)        # False False  Note) Matching
    print('e-mail' in line)        # False True   Note) Searching
    print('e-mail' in line.lower()) # True  True   Note) upper()
    print(line.find('e-mail'))     # -1    22      Note) Starting index
    print(line.endswith('.'))      # True  True    Note) startswith()
```

Data Types

- **Compound data**

    - **tuple**: Comma-separated <u>arbitrary</u> Python objects

        ```
        prof_tuple = ('Choi', 327, True) or prof_tuple = 'Choi', 327, True
        ```

    - **list**: A list of <u>arbitrary</u> Python objects (~ std::array in C++)

        ```
        prof_list = ['Choi', 327, True]
        ```

    - **set**: A unordered set of unique <u>arbitrary</u> objects (~ std::set in C++)

        ```
        prof_set =  {'Choi', 327, True}
        prof_set == {'Choi', True, 327}       # True (unordered)
        prof_set == {'Choi', 327, True, True} # True (unique)
        ```

    - **dict**: A hash table of <u>arbitrary</u> values indexed by <u>arbitrary</u> keys (~ std::map in C++)

        ```
        prof_dict = {'name': 'Choi', 'room_no': 327, 2021: True}
        ```

    Note) The *compound* data can contain heterogenous data types (not same with *arrays* with a homogeneous data type).

Data Types

- **Compound data**

  Given) `prof_str   =  'Choi'`
        `prof_tuple = ('Choi', 327, True)`
        `prof_list  = ['Choi', 327, True]`
        `prof_set   = {'Choi', 327, True}`
        `prof_dict  = {'name': 'Choi', 'room_no': 327, 2021: True}`

  - **Indexing**

    ```
    prof_tuple[0] == 'Choi'
    prof_list[-1] == True # Reverse indexing
    prof_set[0]            # Error!
    prof_dict['name'] == 'Choi'
    prof_dict[2021] == True
    ```

  - **Slicing**

    ```
    prof_str[1:3] == 'ho'
    prof_str[1:] == 'hoi'
    prof_str[1::2] == 'hi'
    prof_list[::-1] == [True, 327, 'Choi']
    ```

13

Data Types

- **Compound data**

  Given) `prof_str  =  'Choi'`

  `prof_tuple = ('Choi', 327, True)`

  `prof_list  = ['Choi', 327, True]`

  `prof_set   = {'Choi', 327, True}`

  `prof_dict  = {'name': 'Choi', 'room_no': 327, 2021: True}`

  – **Concatenation**: Merging two compounds

  `new_str  = prof_str  + ' Sunglok'`

  `new_list = prof_list + ['Mirae Hall', 'SEOULTECH']`

  `prof_set.union({327, 'Mirae Hall', 'SEOULTECH'}) # prof_set  = {'Choi', ..., 'SEOULTECH'}`

  `prof_dict.update({'room_no': 109, 'building': 'Mirae Hall', 'school': 'SEOULTECH'}) # ...`

  – **Appending**: Adding an item to a compound

  `prof_list.append('Mirae Hall')            # prof_list = ['Choi', ..., 'MiraeHall']`

  `prof_set.add('Mirae Hall')                # prof_set  = {'Choi', ..., 'MiraeHall'}`

  `prof_dict['building'] = 'Mirae Hall'      # prof_dict = {'name': ..., 'building': 'MiraeHall'}`

  Check) How about concatenation and appending for a **tuple**?

14

Data Types

Compound data

- **tuple vs. list**

  ```python
  prof_tuple[0] = 'Sunglok' # Error!
  prof_list[0]  = 'Sunglok'
  ```

- **Why tuple?** ~~Collecting and modifying data~~

  - **Packing** and **unpacking**

  - e.g. Swap

    ```python
    (a, b) = (b, a) # a, b = b, a
    ```

  - e.g. Multiple return values

  - Note) Extended unpacking

    ```python
    name, *others = prof_tuple
    # name = 'Choi'
    # others = (327, True)
    ```

Sequential

		Indexing / Slicing
Type	**Ordered**	**Mutable**
str	O	X
tuple	O	X
list	O	O
set	X	O
dict	X	O

Sequential · Editable

```python
def mean_var(data):
    n = len(data)
    if n > 0:
        mean = sum(data) / n
        sum2 = 0
        for datum in data:
            sum2 += datum**2
        var  = sum2 / n - mean**2
        return mean, var
    return None, None


data = [3, 2, 9, 1, 0, 8, 7, 5 ]
pair = mean_var(data)        # pair = (4.375, 9.984)
mean, var = mean_var(data)  # mean = 4.375, var = 9.984
```

15

Data Types

- **Useful built-in functions**
  - **Type check**

    ```
    type(prof_str) == str
    ```

  - **Type casting**

    ```
    int(3.29) == 3
    str(3) == '3'
    int('29') == 29
    ```
    Note) The above two conversions are more easy-to-use than C/C++/Java.

  - **Length of compound data**

    ```
    len(prof_name) == 4 # The number of items
    ```

Operators

- **Operator precedence**

Operator Types	Operators	Description

Compound data (Parentheses)	`(expressions...),`
	`[expressions...],`
	`{key: value...},`
	`{expressions...}`
		Binding (tuple) / parenthesized expression,
		list,
		dictionary,
		set

Subscription	`x[index],`
	`x[index:index],`
	`x(arguments...),`
	`x.attribute`
		Indexing,
		slicing,
		call,
		attribute reference

Arithmetic (Bitwise)	`**`	Exponentiation
	`+x, -x, ~x`	Positive, negative, bitwise NOT
	`*, @,` `/, //, %`	Multiplication, matrix multiplication, division, floor division, remainder
	`+, -`	Addition and subtraction
Bitwise	`<<, >>`	Bitwise shifts
	`&`	Bitwise AND
	`^`	Bitwise XOR
	`|`	Bitwise OR

Membership Identity Comparison	`in, not in,`
	`is, is not,`
	`<, <=, >, >=, !=, ==`
		Membership tests,
		Identity tests,
		Comparisons

Logical	`not`	Boolean NOT
	`and`	Boolean AND
	`or`	Boolean OR
Ternary	`if – else`	Conditional expression
Lambda	`lambda`	Lambda expression
Assignment	`=, +=, -=, *=, /=`	Assignment expression

17

Operators

- **Arithmetic** operators

```python
type(4 / 2) == float  # Always 'float' type (not 'int' type)

(7.5 % 2) == 1.5      # Modulo (remainder)

(7.5 // 2) == 3       # Floor division (integer division; 'int' type)

(-7.5 // 2) == -4

(2 ** 4) == 16        # Exponentiation
```

Note) Please distinguish division (/; `float` type) and floor division (//; `int` type).

- **Logical** operators

```python
 not 3.29 > 3  and  10.18 < 10 or 5.12 > 5  # Note) They are not '!', '&&', and '||'.

(not 3.29 > 3) and (10.18 < 10 or 5.12 > 5) # Check two results
```

- **Ternary** operators

```python
x = 3

is_odd = True if x % 2 == 1 else False # In Python

is_odd = (x % 2) == 1 ? 1 : 0;         // In C/C++
```

Operators

- **Lambda expression**: A short function (as a variable)

```python
is_odd = lambda x: True if x % 2 == 1 else False

is_odd(3) # True
```

- **Note) Sorting**

```python
data = [3, 2, 9]

# list.sort(): An in-place function
data.sort()                       # data = [2, 3, 9]
data.sort(reverse=True)           # data = [9, 3, 2]
data.sort(key=lambda x: abs(x - 4)) # data = [3, 2, 9]

# sorted(): Returning a new list
sorted(data)                      # [2, 3, 9]
sorted(data, reverse=True)        # [9, 3, 2]
sorted(data, key=lambda x: abs(x - 4)) # [3, 2, 9]
```

Example) Sorting points according to their distance from the origin (0, 0)

```python
pts = [(3, 29), (10, 18), (10, 27), (5, 12)]
nearest_pts = sorted(pts, key=lambda pt: pt[0]**2 + pt[1]**2)
```

Operators

- **Lambda expression**: A short function (as a variable)

  ```
  is_odd = lambda x: True if x % 2 == 1 else False
  is_odd(3) # True
  ```

- **Note) List comprehensions**

  Example) Generate a list with $[0^2, 1^2, \ldots, 8^2, 9^2]$

  - **C-style codes**

    ```
    squares = []
    for x in range(10):
        squares.append(x**2)
    ```

  - **map function** (Note: filter, reduce)

    ```
    squares = list(map(lambda x: x**2, range(10)))
    ```

  - **List comprehension**

    ```
    squares = [x**2 for x in range(10)]
    ```

Operators

- **Identity** operator

   Equality (value) vs. **identity** (~ address)

```python
x = 2021
y = 2020 + 1
x == y  # True  Note) print(x, y)
x is y  # False Note) print(id(x), id(y))
```

- **Membership** operator (~ string search)

```python
1 in [0, 1, 2, 3, 4]       # True

5 not in range(5)          # True

1 not in range(0, 5, 2)   # True

'name' in {'name': 'Choi', 'room_no': 327, 2021: True} # True
```

Flow Control

- **Condition**: `if` statement, ~~`switch-case` statement~~ `match-case` statement (Python 3.10+)

- **Loop**: `for` statement, `while` statement
  - Loop control: `break`, `continue`, and `else` statements

- **No action**: `pass` statement (similar to `;` and `{ }` in C/C++)

- Example) Factorial (of a positive integer n)
  - The product of all positive integers less than or equal to n
  - $n! = n \cdot (n-1)! = n \cdot (n-1) \cdot (n-2) \cdots \cdots 3 \cdot 2 \cdot 1$

```python
n = 7       # The given integer
f = 1       # The result of factorial
if n < 0:   # Note) No curly-bracket for a block
    pass
elif n == 0:
    pass
else:
    while n > 0:
        f = f * n
        n = n - 1
```

22

Flow Control

- **Condition**: `if` statement, ~~switch-case statement~~ `match-case` statement (Python 3.10+)

- **Loop**: `for` statement, `while` statement
  - Loop control: `break`, `continue`, and `else` statements

- **No action**: `pass` statement (similar to ; and { } in C/C++)

- Example) Prime number
  - A natural number ($n > 1$) that is not a product of two smaller natural numbers

```python
n = 7 # The given integer
for x in range(2, n):
    if n % x == 0:
        print(n, 'equals', x, '*', n//x)
        break
else:
    print(n, 'is a prime number')
```

Flow Control

- **Loop**: for statements

  - for statements with **sequential data** (string, list, tuple, ...)

    - You can loop with each index or each item (~ std::iterator in C++) or both.

```python
year_list = [1982, 1984, 2014, 2016]
for idx in range(len(year_list)):            # Loop with indices
    print(idx)
for item in year_list:                        # Loop with items
    print(item)
for idx, item in enumerate(year_list):        # Loop with indices and items
    print(idx, item)
```

    - You can merge sequential compound data using zip.

```python
year_list  = [1982, 1984, 2014, 2016]
month_list = ['March', 'October', 'October', 'May']
for idx in range(len(year_list)):                  # Loop with indices
    print(year_list[idx], month_list[idx])
for idx, year in enumerate(year_list):             # Loop with indices and items
    print(year, month_list[idx])
for year, month in zip(year_list, month_list): # Loop with items
    print(year, month)
```

Flow Control

- **Loop**: for statements

  - for statements with **set-type data**

    ```python
    year_set = {1982, 1984, 2014, 2016}
    for item in year_set:
        print(item)
    for num, item in enumerate(year_set):
        print(num, item) # 'num' is not a index.
    ```

  - for statements with **dictionary-type data**

    ```python
    year_dict = {'S': 1982, 'K': 1984, 'J': 2014, 'Y': 2016}
    for key in year_dict:
        print(key)
    for key in year_dict.keys():
        print(key)
    for value in year_dict.values():
        print(value)
    for pair in year_dict.items():
        print(pair)
    for key, value in year_dict.items():
        print(key, value)
    ```

Flow Control

- **Condition**: `if` statements

  - Note) [Falsy and trusy values](#)

```python
values = [False, 0, 0., None, '', (), [], {}, range(0), (None), (None,), [None], {None}]

for val in values:
    if val:
        print(f'{val} is True.')
    else:
        print(f'{val} is False.')
```

Function Definition

- **Function definition**
    - Example) Factorial (of a positive integer n)
        - The product of all positive integers less than or equal to n
        - $n! = n \cdot (n-1)! = n \cdot (n-1) \cdot (n-2) \cdot \cdots \cdot 3 \cdot 2 \cdot 1$

```python
def factorial_for(n):
    f = 1
    for m in range(1, n + 1):
        f *= m
    return f

def factorial_rec(n=1): # Default argument values
    if n <= 0:
        return 1
    else:
        return n * factorial_rec(n - 1)

factorial_for(10) # 3628800
factorial_rec(10) # 3628800
factorial_for()   # Error!
factorial_rec()   # 1
```

Function Definition

- **Multiple return values (as a tuple)**

  - Example) Mean and variance of data

    - Mean (a.k.a. average): $\mu = \mathrm{E}(X) = \frac{1}{n}(x_1 + x_2 + \cdots + x_n)$

    - Variance: $\mathrm{Var}(X) = \mathrm{E}\big((X - \mu)^2\big) = \mathrm{E}(X^2) - \mu^2$

```python
def mean_var(data):
    n = len(data)
    if n > 0:
        mean = sum(data) / n
        sum2 = 0
        for datum in data:
            sum2 += datum**2
        var = sum2 / n - mean**2
        return mean, var
    return None, None

data = [3, 2, 9, 1, 0, 8, 7, 5 ]
pair = mean_var(data)          # pair = (4.375, 9.984)
mean, var = mean_var(data)     # mean = 4.375, var = 9.984
mean, _ = mean_var(data)       # Get only the first one
mean = mean_var(data)[0]       # Get only the first one
```

```python
sum2 = sum([datum**2 for datum in data])
```

28

Function Definition

- **Function overloading is not supported.** [Stack Overflow]

    - Example) My range function #1

```python
def range1(end):          # Ignored
    return list(range(end))

def range1(start, end): # Ignored
    return list(range(start, end))

def range1(start, end, step):
    return list(range(start, end, step))

range1(10)                # Error!
```

Function Definition

- **Various argument passing**
  - Positional arguments, **keyword arguments**, and more
  - Example) My range function #2

```python
def range2(end, start=0, step=1):
    if start > end:
        (start, end) = (end, start)
    return list(range(start, end, step))

print(range2(10))                       # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(range2(1, 10))                    # [1, 2, 3, 4, 5, 6, 7, 8, 9]
print(range2(1, 10, 2))                 # [1, 3, 5, 7, 9]
print(range2(10, step=2))               # [0, 2, 4, 6, 8]
print(range2(step=2, 10))               # Error!
print(range2(step=2, end=10))           # Check the result
print(range2(10, step=2, start=1))      # [1, 3, 5, 7, 9]
print(range2(10, 2, start=1))           # Check the result
arg_tuple = (1, 10, 2)
print(range2(*arg_tuple))               # [1, 3, 5, 7, 9]
arg_dict = {'end': 10, 'step': 2, 'start': 1}
print(range2(**arg_dict))               # [1, 3, 5, 7, 9]
```

Object-oriented Programming

- **Class definition and object instantiation**
    - Example) Dice and coin

```python
from random import randint

class Dice:
    def throw(self):
        return randint(1, 6)

dice = Dice()
print(dice.throw()) # [1, 6]
```

```python
from random import randint

class Dice:
    def __init__(self, boundary=(1, 6)): # A constructor
        self.start = min(boundary)
        self.end = max(boundary)

    def throw(self):
        return randint(self.start, self.end)

dice = Dice()
print(dice.throw()) # [1, 6]

coin = Dice((0, 1))
print(coin.throw()) # 0 or 1
```

Object-oriented Programming

- **Class definition and object instantiation**
  - Example) Dice and coin

```python
from random import import randint

class Dice:
    def throw(self):
        return randint(1, 6)

dice = Dice()
print(dice.throw()) # [1, 6]
```

```python
class Coin(Dice):
    def throw(self):
        return super().throw() % 2

coin = Coin()
print(coin.throw()) # 0 or 1
```

```python
from random import import randint

class Dice:
    def __init__(self, boundary=(1, 6)): # A constructor
        self.start = min(boundary)
        self.end = max(boundary)

    def throw(self):
        return randint(self.start, self.end)

dice = Dice()
print(dice.throw()) # [1, 6]

coin = Dice((0, 1))
print(coin.throw()) # 0 or 1
```

What is better programming (in the view of OOP or users)?

Object-oriented Programming

- **Private member definition**
  - Example) Dice and coin

```python
from random import randint

class Dice:
    def throw(self):
        return randint(1, 6)

dice = Dice()
print(dice.throw()) # [1, 6]
```

```python
class Coin(Dice):
    def throw(self):
        return super().throw() % 2

coin = Coin()
print(coin.throw()) # 0 or 1
```

```python
from random import randint

class Dice:
    def __init__(self, boundary=(1, 6)): # A constructor
        self.__start = min(boundary)
        self.__end = max(boundary)

    def throw(self):
        return randint(self.start, self.end)

dice = Dice()
print(dice.__start, dice.__end)         # Error!
print(dice._Dice__start, dice._Dice__end) # 1 6
```

What is better programming (in the view of OOP or users)?

File Input and Output

- Example) Read a text file using `read()`

```python
f = open('data/class_score_kr.csv', 'r')
lines = f.read()          # Read all lines together
print(lines)
f.close()
```

```
123, 94
112, 92
97, 98
87, 90
89, 87
...
```

- Example) Read a text file using `readline()`

```python
f = open('data/class_score_kr.csv', 'r')
while True:
    line = f.readline() # Read a line sequentially
    if not line:
        break
    print(line)
f.close()
```

```
123, 94

112, 92

97, 98
...
```

- Example) Read a text file using `readlines()`

```python
with open('data/class_score_kr.csv', 'r') as f:
    for line in f.readlines(): # Read all lines as a list
        print(line.strip())
```

```
123, 94
112, 92
97, 98
87, 90
89, 87
...
```

File Input and Output

- Example) Write a text file using `write()`

```python
with open('data/class_score_kr.csv', 'r') as fi, open('class_score_mean.csv', 'w') as fo:
    for line in fi.readlines():
        values = [int(word) for word in line.split(',')]
        mean = sum(values) / len(values)
        for val in values:
            fo.write(f'{val}, ')
        fo.write(f'{mean}\n')
```

- Check) I provide you two files, `'data/class_score_kr.csv'` and `'data/class_score.csv'`.

    Please try the above example with following files.

    – `'data/class_score.csv'` (which does not exist) → FileNotFoundErrror: [Errno 2] No such file or directory: …

    – `'data/class_score_en.csv'` (which contains the header) → ValueError: invalid literal for int() with based 10: …

```
# midterm (max 125), final (max 100)
113, 86
104, 83
110, 78
...
```

Exception Handling

- **Exception** (~ runtime error)
  - Anomalous or exceptional conditions requiring special processing (during the execution of a program)
  - Built-in exceptions in Python

- Example) Calculate averaged scores from a text file with the header

```python
with open('data/class_score_en.csv', 'r') as fi, \
     open('class_score_mean.csv', 'w') as fo:
  for line in fi.readlines():
      try:
          values = [int(word) for word in line.split(',')]
          mean = sum(values) / len(values)
          for val in values:
              fo.write(f'{val}, ')
          fo.write(f'{mean}\n')

      except ValueError as ex:
          print(f'A line is ignored. (message: {ex})')
```

Exception Handling

- **Exception** (~ runtime error)
  - Anomalous or exceptional conditions requiring special processing (during the execution of a program)
  - Built-in exceptions in Python

- Example) Calculate averaged scores from a not-exist text file

```python
try:
    with open('data/class_score.csv', 'r') as fi, \
         open('class_score_mean.csv', 'w') as fo:
        for line in fi.readlines():
            try:
                values = [int(word) for word in line.split(',')]
                mean = sum(values) / len(values)
                for val in values:
                    fo.write(f'{val}, ')
                fo.write(f'{mean}\n')

            except ValueError as ex: # Try 'data/class_score_en.csv' without this try/except
                print(f'A line is ignored. (message: {ex})')
except Exception as ex:
    print(f'Cannot run the program. (message: {ex})')
```

Package Import

- Example) Calculate mean and variance of scores using the previous example

mean_var.py

```python
def mean_var(data):
    n = len(data)
    if n > 0:
        mean = sum(data) / n
        sum2 = sum([datum**2 for datum in data])
        var = sum2 / n - mean**2
        return mean, var
    return None, None

data = [3, 2, 9, 1, 0, 8, 7, 5 ]
mean, var = mean_var(data)
print(f'mean = {mean:.3f}, var = {var:.3f}')
```

Result)

```
mean = 4.375, var = 9.984
```

class_score_mean.py

```python
from mean_var import mean_var

try:
    with open('data/class_score_kr.csv', 'r') as fi, \
            open('class_score_mean.csv', 'w') as fo:
        for line in fi.readlines():
            try:
                values = [int(word) for word in line.split(',')]
                mean, var = mean_var(values)
                for val in values:
                    fo.write(f'{val}, ')
                fo.write(f'{mean}, {var}\n')

            except ValueError as ex:
                print(f'A line is ignored. (message: {ex})')
        print('The program was terminated successfully.')
except Exception as ex:
    print(f'Cannot run the program. (message: {ex})')
```

Result)

```
mean = 4.375, var = 9.984
The program was terminated successfully.
```

Package Import

- Example) Calculate mean and variance of scores using the previous example

mean_var.py

```python
def mean_var(data):
    n = len(data)
    if n > 0:
        mean = sum(data) / n
        sum2 = sum([datum**2 for datum in data])
        var = sum2 / n - mean**2
        return mean, var
    return None, None

if __name__ == '__main__':
    data = [3, 2, 9, 1, 0, 8, 7, 5 ]
    mean, var = mean_var(data)
    print(f'mean = {mean:.3f}, var = {var:.3f}')
```

Result)

```
mean = 4.375, var = 9.984
```

mean_var.py: __name__ == '__main__' (entry point)

class_score_mean.py

```python
from mean_var import mean_var

try:
    with open('data/class_score_kr.csv', 'r') as fi, \
         open('class_score_mean.csv', 'w') as fo:
        for line in fi.readlines():
            try:
                values = [int(word) for word in line.split(',')]
                mean, var = mean_var(values)
                for val in values:
                    fo.write(f'{val}, ')
                fo.write(f'{mean}, {var}\n')

            except ValueError as ex:
                print(f'A line is ignored. (message: {ex})')
        print('The program was terminated successfully.')
except Exception as ex:
    print(f'Cannot run the program. (message: {ex})')
```

Result)

```
The program was terminated successfully.
```

class_score_mean.py: __name__ == '__main__' (entry point)
mean_var.py   : __name__ == 'mean_var'

39

Summary

- **My Comments for Better Python Programming**

  1) Take advantages of Python itself / 2) Utilize the exiting libraries and master them if they are useful

- **Data Types**
  - **String**: Formatting, trimming, splitting (tokenization), matching, and searching
  - Compound data: In-place methods, **tuple (immutable**; packing/unpacking**) vs. list (mutable)**

- **Operators**
  - Lambda operator: A way to define a function shortly
    - `key` in **sorting**
    - `map`(`/filter/reduce`) function → **list comprehensions**
  - Identity (`id` ~ address) and membership (`in`) operators

- **Flow Control**
  - Easier access to elements in compound data with `for` statement
  - Falsy and trusy values

- **Function Definition: Keyword argument association**, passing compound data as a series of arguments

- **Object-oriented Programming:** Inheritance, private member definition

- **File Input and Output:** `open` function

- **Exception Handling:** `try` - `except` statement

- **Package Import:** Why `__name__ == '__main__'` is necessary?