

합수 고급

- 시작하기 전에
- 튜플
- 람다
- 파일 처리
- 제너레이터
- 키워드로 정리하는 핵심 포인트
- 확인문제

[핵심 키워드] : 튜플, 람다, with 구문

[핵심 포인트]

튜플, 람다, 파일 처리는 파이썬만의 특별한 문법이라 따로 공부하지 않으면 다른 프로그래밍 언어를 배웠던 사람도 이해하기 어렵다. 튜플 람다, 파일 처리 등 함수와 관련된 파이썬의 특별한 문법과 기능을 살펴본다.

- 튜플 (tuple)

- 함수와 함께 많이 사용되는 리스트와 비슷한 자료형으로, 한번 결정된 요소를 바꿀 수 없다는 점이 리스트와 다름

- 람다 (lambda)

- 매개변수로 함수를 전달하기 위해 함수 구문을 작성하는 것이 번거롭고 코드 낭비라 생각될 때 함수를 간단하고 쉽게 선언하는 방법

- 튜플 (tuple)
 - 리스트와 유사한 자료형
 - 한번 결정된 요소는 바꿀 수 없음

(데이터, 데이터, 데이터, ...)

```
>>> tuple_test = (10, 20, 30)
```

```
>>> tuple_test[0]
```

```
10
```

```
>>> tuple_test[1]
```

```
20
```

```
>>> tuple_test[2]
```

```
30
```

```
>>> tuple_test[0] = 1
```

```
Traceback (most recent call last):
```

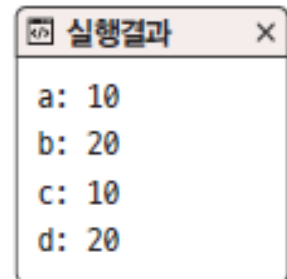
```
  File "<pyshell#1>", line 1, in <module>
```

```
    tuple_test[0] = 1
```

```
TypeError: 'tuple' object does not support item assignment
```

- 괄호 없는 튜플
 - 예시

```
01  # 리스트와 튜플의 특이한 사용
02  [a, b] = [10, 20]
03  (c, d) = (10, 20)
04
05  # 출력합니다.
06  print("a:", a)
07  print("b:", b)
08  print("c:", c)
09  print("d:", d)
```



실행결과

```
a: 10
b: 20
c: 10
d: 20
```

— 괄호를 생략

```
01  # 괄호가 없는 튜플
02  tuple_test = 10, 20, 30, 40
03  print("# 괄호가 없는 튜플의 값과 자료형 출력")
04  print("tuple_test:", tuple_test)
05  print("type(tuple_test:)", type(tuple_test))
06  print()
07
08  # 괄호가 없는 튜플 활용
09  a, b, c = 10, 20, 30
10  print("# 괄호가 없는 튜플을 활용한 할당")
11  print("a:", a)
12  print("b:", b)
13  print("c:", c)
```

→ 튜플을 입력한 것입니다.

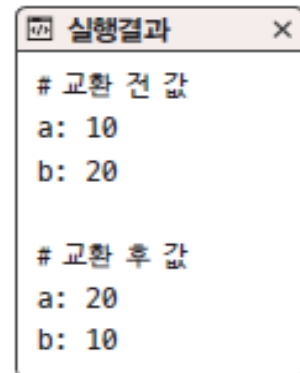
실행결과

```
# 괄호가 없는 튜플의 값과 자료형 출력
tuple_test: (10, 20, 30, 40)
type(tuple_test:) <class 'tuple'>
```

```
# 괄호가 없는 튜플을 활용한 할당
a: 10
b: 20
c: 30
```


— 활용 예시 – 변수의 값을 교환하는 튜플

```
01  a, b = 10, 20
02
03  print("# 교환 전 값")
04  print("a:", a)
05  print("b:", b)
06  print()
07
08  # 값을 교환합니다.
09  a, b = b, a
10
11  print("# 교환 후 값")
12  print("a:", a)
13  print("b:", b)
14  print()
```



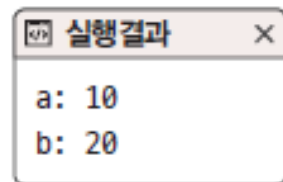
실행결과

```
# 교환 전 값
a: 10
b: 20

# 교환 후 값
a: 20
b: 10
```

- 튜플과 함수
 - 예시 - 여러 개의 값 리턴하기

```
01  # 함수를 선언합니다.
02  def test():
03      return (10, 20)
04
05  # 여러 개의 값을 리턴받습니다.
06  a, b = test()
07
08  # 출력합니다.
09  print("a:", a)
10  print("b:", b)
```



실행결과

a: 10
b: 20

- 람다 (lambda)
 - 기능을 매개변수로 전달하는 코드를 더 효율적으로 작성
- 함수의 매개변수로 함수 전달하기

```
01 # 매개변수로 받은 함수를 10번 호출하는 함수
02 def call_10_times(func):
03     for i in range(10):
04         func()
05
06 # 간단한 출력하는 함수
07 def print_hello():
08     print("안녕하세요")
09
10 # 조합하기
11 call_10_times(print_hello)
```

↓
매개변수로 함수를 전달합니다.

실행결과

안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요
안녕하세요

- filter() 함수와 map() 함수
 - 함수를 매개변수로 전달하는 대표적인 표준함수

```
map(함수, 리스트)
```

```
filter(함수, 리스트)
```

```
01  # 함수를 선언합니다.
02  def power(item):
03      return item * item
04  def under_3(item):
05      return item < 3
06
07  # 변수를 선언합니다.
08  list_input_a = [1, 2, 3, 4, 5]
09
```

```

10 # map() 함수를 사용합니다.
11 output_a = map(power, list_input_a)
12 print("# map() 함수의 실행결과")
13 print("map(power, list_input_a):", output_a)
14 print("map(power, list_input_a):", list(output_a))
15 print()
16
17 # filter() 함수를 사용합니다.
18 output_b = filter(under_3, list_input_a)
19 print("# filter() 함수의 실행결과")
20 print("filter(under_3, output_b):", output_b)
21 print("filter(under_3, output_b):", list(output_b))

```

→ 함수를 매개변수로 넣었습니다.

실행결과

```

# map() 함수의 실행결과
map(power, list_input_a): <map object at 0x03862270>
map(power, list_input_a): [1, 4, 9, 16, 25]

# filter() 함수의 실행결과
filter(under_3, output_b): <filter object at 0x03862290>
filter(under_3, output_b): [1, 2]

```

- 람다란 '간단한 함수를 쉽게 선언하는 방법'

```
lambda 매개변수 : 리턴값
```

```
01  # 함수를 선언합니다.
02  power = lambda x: x * x
03  under_3 = lambda x: x < 3
04
05  # 변수를 선언합니다.
06  list_input_a = [1, 2, 3, 4, 5]
07
08  # map() 함수를 사용합니다.
09  output_a = map(power, list_input_a)
10  print("# map() 함수의 실행결과")
11  print("map(power, list_input_a):", output_a)
12  print("map(power, list_input_a):", list(output_a))
13  print()
```

```

14
15 # filter() 함수를 사용합니다.
16 output_b = filter(under_3, list_input_a)
17 print("# filter() 함수의 실행결과")
18 print("filter(under_3, output_b):", output_b)
19 print("filter(under_3, output_b):", list(output_b))

```

실행결과

```

# map() 함수의 실행결과
map(power, list_input_a): <map object at 0x03862270>
map(power, list_input_a): [1, 4, 9, 16, 25]

# filter() 함수의 실행결과
filter(under_3, output_b): <filter object at 0x03862290>
filter(under_3, output_b): [1, 2]

```

— 예시 – 인라인 람다

- 함수의 매개변수에 람다 곧바로 넣을 수 있음

```

01  # 변수를 선언합니다.
02  list_input_a = [1, 2, 3, 4, 5]
03
04  # map() 함수를 사용합니다.
05  output_a = map(lambda x: x * x, list_input_a)
06  print("# map() 함수의 실행결과")
07  print("map(power, list_input_a):", output_a)
08  print("map(power, list_input_a):", list(output_a))
09  print()
10
11  # filter() 함수를 사용합니다.
12  output_b = filter(lambda x: x < 3, list_input_a)
13  print("# filter() 함수의 실행결과")
14  print("filter(under_3, output_b):", output_b)
15  print("filter(under_3, output_b):", list(output_b))

```

power() 함수를 선언하지도 않고,
매개변수로 바로 넣었습니다.

under_3() 함수를 선언하지도 않고,
매개변수로 바로 넣었습니다.

- 매개변수가 여러 개인 람다

```
lambda x, y: x * y
```

- 텍스트 파일의 처리
- 파일 열기 (open) – 파일 읽기 (read) – 파일 쓰기 (write) – 닫기 (close)
- 파일 열고 닫기
 - open() 함수

파일 객체 = open(문자열: 파일 경로, 문자열: 읽기 모드)

모드	설명
w	write 모드(새로 쓰기 모드)
a	append 모드(뒤에 이어서 쓰기 모드)
r	read 모드(읽기 모드)

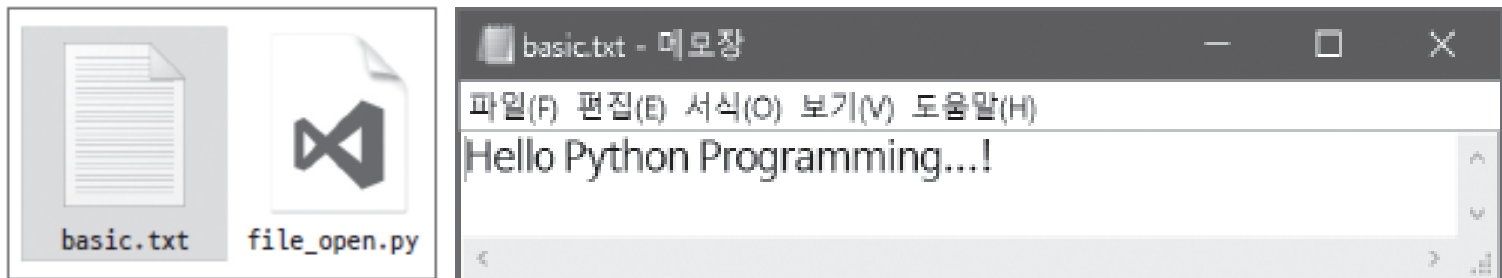
- closed() 함수

파일 객체.close()

- 예시

```
01  # 파일을 엽니다.  
02  file = open("basic.txt", "w")  
03  
04  # 파일에 텍스트를 씁니다.  
05  file.write("Hello Python Programming...!")  
06  
07  # 파일을 닫습니다.  
08  file.close()
```

- 프로그램 실행 시 폴더에 basic.txt 파일 생성
- 실행 시 다음 형태



- open() 함수로 열면 close() 함수로 닫아야 함

- with 키워드

- 조건문과 반복문 들어가다 보면 파일을 열고서 닫지 않는 실수 하는 경우 생길 수 있음
- with 구문 종료 시 파일을 자동으로 닫음

```
with open(문자열: 파일 경로, 문자열: 모드) as 파일 객체:  
    문장
```

```
# 파일을 엽니다.  
with open("basic.txt", "w") as file:  
    # 파일에 텍스트를 씁니다.  
    file.write("Hello Python Programming...!")
```

- 텍스트 읽기
 - `read()` 함수

파일 객체.`read()`

```
01  # 파일을 엽니다.
02  with open("basic.txt", "r") as file:
03      # 파일을 읽고 출력합니다.
04      contents = file.read()
05  print(contents)
```

읽기 모드로 변경했습니다!

실행결과

Hello Python Programming...!

- 텍스트 한 줄씩 읽기

- CSV, XML, JSON 방법 등으로 텍스트를 사용해 데이터를 구조적으로 표현
- CSV 예시

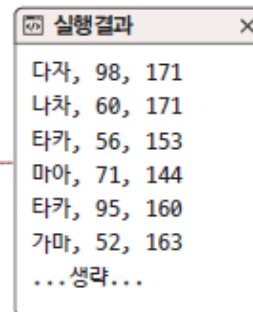
```
이름, 키, 몸무게  
윤인성, 176, 62  
연하진, 169, 50
```

- 한 줄에 하나의 데이터 나타내며 각 줄은 쉼표 사용해 데이터 구분함
- 첫 줄에 헤더 넣어 각 데이터가 나타내는 바 설명
- 한 번에 모든 데이터 올려놓고 사용하는 것이 컴퓨터 성능에
영향 미칠 수도 있음

예시 - 랜덤하게 1000명의 키와 몸무게 만들기

```
01  # 랜덤한 숫자를 만들기 위해 가져옵니다.  
02  import random  
03  # 간단한 한글 리스트를 만듭니다.  
04  hanguls = list("가나다라마바사아자차카타파하")  
05  # 파일을 쓰기 모드로 엽니다.  
06  with open("info.txt", "w") as file:  
07      for i in range(1000):  
08          # 랜덤한 값으로 변수를 생성합니다.  
09          name = random.choice(hanguls) + random.choice(hanguls)  
10          weight = random.randrange(40, 100)  
11          height = random.randrange(140, 200)  
12          # 텍스트를 씁니다.  
13          file.write("{} {}, {}{}\n".format(name, weight, height))
```

info.txt에 생성된 데이터입니다. ←



- 데이터를 한 줄씩 읽어들이는 때는 for 반복문을 다음과 같이 사용

```
for 한 줄을 나타내는 문자열 in 파일 객체:  
    처리
```

- 키와 몸무게로 비만도 계산

```
01  with open("info.txt", "r") as file:  
02      for line in file :  
03          # 변수를 선언합니다.  
04          (name, weight, height) = line.strip().split(", ")  
05  
06          # 데이터가 문제없는지 확인합니다: 문제가 있으면 지나감  
07          if (not name) or (not weight) or (not height):  
08              continue  
09          # 결과를 계산합니다.  
10          bmi = int(weight) / (int(height) * int(height))
```

```
11     result = ""
12     if 25 <= bmi:
13         result = "과체중"
14     elif 18.5 <= bmi:
15         result = "정상 체중"
16     else:
17         result = "저체중"
18
19     # 출력합니다.
20     print('\n'.join([
21         "이름: {}".format(name),
22         "몸무게: {}".format(weight),
23         "키: {}".format(height),
24         "BMI: {}".format(bmi),
25         "결과: {}".format(result)
26     ]).format(name, weight, height, bmi, result))
27     print()
```

실행결과

이름: 타나
몸무게: 63
키: 165
BMI: 0.0023140495867768596
결과: 과체중

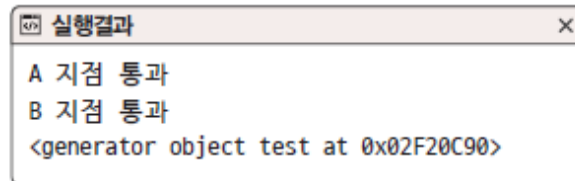
이름: 마나
몸무게: 58
키: 187
BMI: 0.0016586119134090194
결과: 저체중

이름: 바타
몸무게: 53
키: 161
BMI: 0.00204467420238
결과: 저체중

...생략...

- 제너레이터 (generator)
 - 이터레이터를 직접 만들 때 사용하는 코드
 - 함수 내부에 yield 키워드 사용하면 해당 함수는 제너레이터 함수 됨
 - 일반 함수와 달리 호출해도 함수 내부 코드가 실행되지 않음

```
01  # 함수를 선언합니다.  
02  def test():  
03      print("함수가 호출되었습니다.")  
04      yield "test"  
05  
06  # 함수를 호출합니다.  
07  print("A 지점 통과")  
08  test()  
09  
10  print("B 지점 통과")  
11  test()  
12  print(test())
```



실행결과

A 지점 통과
B 지점 통과
<generator object test at 0x02F20C90>

- next() 함수 사용해 내부 코드 실행
 - yield 키워드 부분까지만 실행하며 next() 함수 리턴값으로 yield 키워드 뒤에 입력한 값이 출력됨

```
01 # 함수를 선언합니다.
02 def test():
03     print("A 지점 통과")
04     yield 1
05     print("B 지점 통과")
06     yield 2
07     print("C 지점 통과")
08
09 # 함수를 호출합니다.
10 output = test()
11
12 # next() 함수를 호출합니다.
13 print("D 지점 통과")
14 a = next(output)
15 print(a)
16
17 print("E 지점 통과")
18 b = next(output)
19 print(b)
```

실행결과

```
D 지점 통과
A 지점 통과
1
E 지점 통과
B 지점 통과
2
F 지점 통과
C 지점 통과
Traceback (most recent call last):
  File "generator01.py", line 22, in <module>
    c = next(output)
StopIteration
```

- next() 함수 호출한 이후 yield 키워드 만나지 못하고 함수 끝나면 StopIteration 예외 발생

- **튜플** : 리스트와 비슷하지만, 요소를 수정할 수 없는 파이썬의 특별한 문법.
괄호 생략하여 다양하게 활용할 수 있음
- **람다** : 함수를 짧게 쓸 수 있는 파이썬의 특별한 문법
- **with 구문** : 블록을 벗어날 때 close() 함수를 자동으로 호출하는 구문

- 빈 칸을 채워서 실행결과처럼 출력되게 만들어주세요.

```
numbers = [1, 2, 3, 4, 5, 6]  
  
print(":".join( ))
```

실행결과

1::2::3::4::5::6

- 다음 코드의 빈칸을 채워서 실행결과처럼 결과가 나오게 해주세요.

```
numbers = list(range(1, 10 + 1))

print("# 홀수만 추출하기")
print(list(filter( , numbers)))
print()

print("# 3 이상, 7 미만 추출하기")
print(list(filter( , numbers)))
print()

print("# 제곱해서 50 미만 추출하기")
print(list(filter( , numbers)))
```



```
실행결과
# 홀수만 추출하기
[1, 3, 5, 7, 9]

# 3 이상, 7 미만 추출하기
[3, 4, 5, 6]

# 제곱해서 50 미만 추출하기
[1, 2, 3, 4, 5, 6, 7]
```