5. 함수

contents

- 함수
- lambda
- 사용자 입출력
- 파일 입출력
- 프로그램의 입출력

1. 함수

- **함수란 무엇인가?**

  - 우리는 믹서에 과일을 넣고, 믹서를 사용해서 과일을 갈아 과일 주스를 만듦

  - 믹서에 넣는 과일 = 입력
  - 과일주스 = 출력(결괏값)
  - 믹서 = ?

믹서는 과일을 입력받아 주스를 출력하는 함수와 같다.

1. 함수

- **함수를 사용하는 이유는 무엇일까?**

  - 반복되는 부분이 있을 경우 '반복적으로 사용되는 가치 있는 부분'을 한 뭉치로 묶어
    '어떤 입력값을 주었을 때 어떤 결괏값을 리턴해 준다'라는 식의 함수로 작성하는 것이 현명함

  - 프로그램의 흐름을 파악하기 좋고 오류 발생 지점도 찾기 쉬움

1. 함수

- **파이썬 함수의 구조**
  - def : 함수를 만들 때 사용하는 예약어
  - 함수 이름은 임의로 생성 가능
  - 매개변수는 함수에 입력으로 전달되는 값을 받는 변수
  - return : 함수의 결괏값(리턴값)을 돌려주는 명령어

```
def 함수_이름(매개변수):
    수행할_문장1
    수행할_문장2
    ...
```

```
def add(a, b):
    return a + b
```

  - 함수의 풀이

이 함수의 이름은 add이고 입력으로 2개의 값을 받으며 리턴값(출력값)은 2개의 입력값을 더한 값이다.

1. 함수

파이썬 함수의 구조

예) add 함수

add 함수 만들기

```
>>> def add(a, b):
...     return a + b
...
>>>
```

add 함수 사용하기

```
>>> a = 3
>>> b = 4
>>> c = add(a, b)  <-- add(3, 4)의 리턴값을 c에 대입
>>> print(c)
7
```

1. 함수

매개변수와 인수

- 매개변수와 인수는 혼용해서 사용되는 헷갈리는 용어로 잘 구분하는 것이 중요!

- 매개변수(parameter)
  - 함수에 입력으로 전달된 값을 받는 변수

- 인수(arguments)
  - 함수를 호출할 때 전달받는 입력값

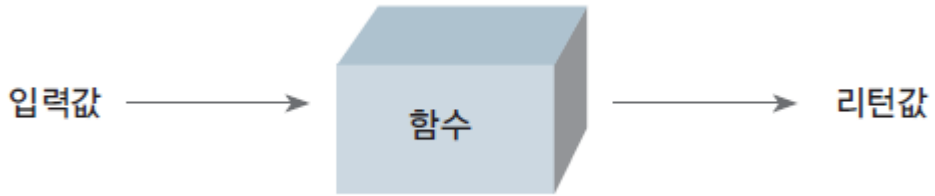
```
def add(a, b):  <- a, b는 매개변수
    return a + b

print(add(3, 4))  <- 3, 4는 인수
```

1. 함수

입력값과 리턴값에 따른 함수의 형태

- 함수는 들어온 입력값을 받은 후 어떤 처리를 하여 적절한 값을 리턴해 줌



- 함수의 형태는 입력값과 리턴값의 존재 유무에 따라 4가지 유형으로 나뉨

1. 함수

입력값과 리턴값에 따른 함수의 형태

```
def 함수_이름(매개변수):
    수행할_문장
    ...
    return 리턴값
```

일반적인 함수

- 입력값이 있고 리턴값이 있는 함수

- 일반 함수의 전형적인 예
  - add 함수는 2개의 입력값을 받아 서로 더한 결괏값을 리턴함

```
>>> def add(a, b):
...     result = a + b
...     return result  <- a + b의 결괏값 리턴
```

```
>>> a = add(3, 4)
>>> print(a)
7
```

```
리턴값을_받을_변수 = 함수_이름(입력_인수1, 입력_인수2, ...)
```

1. 함수

입력값과 리턴값에 따른 함수의 형태

입력값이 없는 함수

- 입력값이 없는 함수도 존재함

- say 함수는 매개변수 부분을 나타내는 함수 이름 뒤의 괄호 안이 비어있음
- 함수 사용 시 say( )처럼 괄호 안에 아무 값도 넣지 않아야 함

```
>>> def say():
...     return 'Hi'
```

```
>>> a = say()
>>> print(a)
Hi
```

```
리턴값을_받을_변수 = 함수_이름()
```

1. 함수

입력값과 리턴값에 따른 함수의 형태

리턴값이 없는 함수

- 리턴값이 없는 함수는 호출해도 리턴되는 값이 없음

```
>>> def add(a, b):
...     print("%d, %d의 합은 %d입니다." % (a, b, a + b))
```

```
>>> add(3, 4)
3, 4의 합은 7입니다.
```

- 사용 방법

```
함수_이름(입력_인수1, 입력_인수2, ...)
```

1. 함수

입력값과 리턴값에 따른 함수의 형태

리턴값이 없는 함수

- 리턴값이 진짜 없을까?
  - 리턴받을 값을 a 변수에 대입하여 출력하여 확인
  - None이란 거짓을 나타내는 자료형으로, 리턴값이 없을 때 쓰임

```
>>> a = add(3, 4) <— add 함수의 리턴값을 a에 대입
3, 4의 합은 7입니다.
>>> print(a) <— a 값 출력
None
```

1. 함수

입력값과 리턴값에 따른 함수의 형태

입력값도, 리턴값도 없는 함수

- 입력 인수를 받는 매개변수도 없고 return 문도 없는, 즉 입력값도 리턴값도 없는 함수

```
>>> def say():
...     print('Hi')
```

```
>>> say()
Hi
```

```
함수_이름()
```

1. 함수

매개변수를 지정하여 호출하기

- 함수 호출 시 매개변수 지정 가능
  - 예) sub 함수

```
>>> def sub(a, b):
...     return a - b
```

  - 매개변수를 지정하여 사용

```
>>> result = sub(a=7, b=3) <— a에 7, b에 3을 전달
>>> print(result)
4
```

- 매개변수를 지정하면 매개변수 순서에 상관없이 사용할 수 있는 장점

```
>>> result = sub(b=5, a=3) <— b에 5, a에 3을 전달
>>> print(result)
-2
```

1. 함수

- **입력값이 몇 개가 될지 모를 때는 어떻게 해야 할까?**
  - 파이썬에서의 해결 방법
    - 일반 함수 형태에서 괄호 안의 매개변수 부분이 *매개변수로 바뀜

```
def 함수_이름(*매개변수):
    수행할_문장
    ...
```

1. 함수

- **입력값이 몇 개가 될지 모를 때는 어떻게 해야 할까?**
  - 여러 개의 입력값을 받는 함수 만들기
    - 매개변수 이름 앞에 *을 붙이면 입력값을 전부 모아 튜플로 만들어 줌

```
>>> def add_many(*args):
...     result = 0
...     for i in args:
...         result = result + i  <- *args에 입력받은 모든 값을 더한다.
...     return result
```

```
>>> result = add_many(1, 2, 3) <- add_many 함수의 리턴값을 result 변수에 대입
>>> print(result)
6
>>> result = add_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
>>> print(result)
55
```

1. 함수

- **입력값이 몇 개가 될지 모를 때는 어떻게 해야 할까?**
  - 여러 개의 입력값을 받는 함수 만들기
    - *args 매개변수 앞에 choice 매개변수를 추가할 수도 있음

```
>>> def add_mul(choice, *args):
...     if choice == "add":  <- 매개변수 choice에 "add"를 입력받았을 때
...         result = 0
...         for i in args:
...             result = result + i  <- args에 입력받은 모든 값을 더한다.
...     elif choice == "mul":  <- 매개변수 choice에 "mul"을 입력받았을 때
...         result = 1
...         for i in args:
...             result = result * i  <- *args에 입력받은 모든 값을 곱한다.
...     return result
```

```
>>> result = add_mul('add', 1, 2, 3, 4, 5)
>>> print(result)
15
>>> result = add_mul('mul', 1, 2, 3, 4, 5)
>>> print(result)
120
```

1. 함수

- **키워드 매개변수, kwargs**
  - 매개변수 앞에 별 2개(**)를 붙임

```
>>> def print_kwargs(**kwargs):
...     print(kwargs)
```

```
>>> print_kwargs(a=1)
{'a': 1}
>>> print_kwargs(name='foo', age=3)
{'age': 3, 'name': 'foo'}
```

- **함수의 리턴값은 언제나 하나이다**
  - 2개의 입력 인수를 받아 리턴하는 함수

```
>>> def add_and_mul(a, b):
...     return a+b, a*b
```

```
>>> result = add_and_mul(3, 4)
```

```
result = (7, 12)
```

- 하나의 튜플 값을 2개의 값으로 분리하여 리턴

```
>>> result1, result2 = add_and_mul(3, 4)
```

- return 문을 2번 사용하면 리턴값은 하나뿐

```
>>> def add_and_mul(a, b):
...     return a + b
...     return a * b
```

```
>>> result = add_and_mul(2, 3)
>>> print(result)
5
```

1. 함수

- **매개변수에 초깃값 미리 설정하기**
  - 매개변수에 초깃값을 미리 설정

```
def say_myself(name, age, man=True):
    print("나의 이름은 %s입니다." % name)
    print("나이는 %d살입니다." % age)
    if man:
        print("남자입니다.")
    else:
        print("여자입니다.")
```

  - man=True처럼 매개변수에 미리 값을 넣어 함수의 매개변수 초깃값을 설정

1. 함수

- **매개변수에 초깃값 미리 설정하기**
  - 매개변수에 들어갈 값이 항상 변하는 것이 아니면, 초깃값을 미리 설정하는 것이 유용함
  - say_myself 함수 사용법

```
say_myself("박응용", 27)
```

```
say_myself("박응용", 27, True)
```

실행 결과

```
나의 이름은 박응용입니다.
나이는 27살입니다.
남자입니다.
```

```
say_myself("박응선", 27, False)
```

실행 결과

```
나의 이름은 박응선입니다.
나이는 27살입니다.
여자입니다.
```

  - 입력값으로 ("박응용", 27)처럼 2개를 주면 name에는 "박응용"이 old에는 27이 대입됨
  - man이라는 변수에는 입력값을 주지 않았지만 초깃값 True를 갖게 됨

1. 함수

매개변수에 초깃값 미리 설정하기

주의할 점

- 초기화하고 싶은 매개변수는 항상 뒤쪽에 놓아야 함

```
def say_myself(name, man=True, age):
    print("나의 이름은 %s입니다." % name)
    print("나이는 %d살입니다." % age)
    if man:
        print("남자입니다.")
    else:
        print("여자입니다.")
```

- 파이썬 인터프리터는 27을 man 매개변수와 age 매개변수 중 어느 곳에 대입해야 할지 판단이 어려워 오류 발생

```
say_myself("박응용", 27)
```

실행 결과

```
SyntaxError: non-default argument follows default argument
```

1. 함수

- **함수 안에서 선언한 변수의 효력 범위**
  - 함수 안에서 사용할 변수의 이름을 함수 밖에서도 동일하게 사용한다면?

```
a = 1               # 함수 밖의 변수 a
def vartest(a):     # vartest 함수 선언
    a = a + 1

vartest(a)          # vartest 함수의 입력값으로 a를 대입
print(a)            # a 값 출력
```

  - 따라서 vertest 함수는 매개변수 이름을 바꾸어도 이전 함수와 동일하게 동작함

- 함수를 실행해 보면, 결괏값은 1이 나옴
- 함수 안에서 사용하는 매개변수는 함수 안에서만 사용하는 '함수만의 변수'이기 때문
- 즉, 매개변수 a는 함수 안에서만 사용하는 변수일 뿐, 함수 밖의 변수 a와는 전혀 상관없음

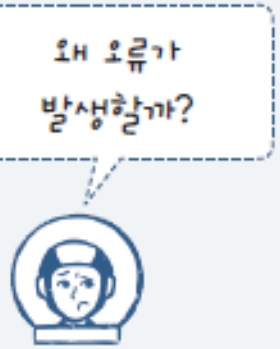
```
def vartest(hello):
    hello = hello + 1
```

1. 함수

- **함수 안에서 선언한 변수의 효력 범위**
  - 함수 안에서 선언한 매개변수는 함수 안에서만 사용될 뿐, 함수 밖에서는 사용되지 않음

```
def vartest(a):
    a = a + 1


vartest(3)
print(a)
```



  - print(a)에서 사용한 a 변수는 어디에도 선언되지 않았기 때문

1. 함수

- **함수 안에서 함수 밖의 변수를 변경하는 방법**

1. return 사용하기

```
a = 1
def vartest(a):
    a = a + 1
    return a

a = vartest(a)    # vartest(a)의 리턴값을 함수 밖의 변수 a에 대입
print(a)
```

2. global 명령어 사용하기

```
a = 1
def vartest():
    global a
    a = a + 1

vartest()
print(a)
```

  - 단, 함수는 독립적으로 존재하는 것이 좋기 때문에 이 방법은 피하는 것이 좋음

2. lambda

- **lambda 예약어**
  - 함수를 생성할 때 사용하는 예약어
  - 함수를 한 줄로 간결하게 만들 때 사용
  - def와 동일한 역할
  - 우리말로 '람다'
  - def를 사용해야 할 정도로 복잡하지 않거나 def를 사용할 수 없는 곳에 주로 쓰임

```
함수_이름 = lambda 매개변수1, 매개변수2, ... : 매개변수를_이용한_표현식
```

2. lambda

- **lambda 예약어**
  - 사용 예시

```
>>> add = lambda a, b: a + b
>>> result = add(3, 4)
>>> print(result)
7
```

    - add는 2개의 인수를 받아
      서로 더한 값을 리턴하는 lambda 함수

  - def를 사용한 경우와 하는 일이 완전히 동일함

```
>>> def add(a, b):
...     return a + b
...
>>> result = add(3, 4)
>>> print(result)
7
```

2. lambda

- 람다 함수의 다양한 형태

```
>>> f = lambda x, y: x + y
>>> f(1, 4)
5
>>>
>>> f = lambda x: x ** 2
>>> f(3)
9
>>>
>>> f = lambda x: x / 2
>>> f(3)
1.5
>>> f(3, 5)
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: <lambda>() takes 1 positional argument but 2 were given
```

3. 사용자 입출력

사용자 입력 활용하기

input 사용하기

- 사용자가 키보드로 입력한 모든 것을 문자열로 저장

```
>>> a = input()
Life is too short, you need python  <-- 사용자가 문장을 입력
>>> a
'Life is too short, you need python'
```

3. 사용자 입출력

사용자 입력 활용하기

프롬프트를 띄워 사용자 입력받기

- 사용자에게 입력받을 때 안내 문구 또는 질문을 보여 주고 싶을 때

```
input("안내_문구")
```

```
>>> number = input("숫자를 입력하세요: ")
숫자를 입력하세요:
```

```
>>> number = input("숫자를 입력하세요: ")
숫자를 입력하세요: 3 <-- 3 입력
>>> print(number)
3
```

3. 사용자 입출력

- **print 자세히 알기**
  - 데이터를 출력하는 데 사용

```
>>> a = 123
>>> print(a) <-- 숫자 출력하기
123
>>> a = "Python"
>>> print(a) <-- 문자열 출력하기
Python
>>> a = [1, 2, 3]
>>> print(a) <-- 리스트 출력하기
[1, 2, 3]
```

3. 사용자 입출력

- **print 자세히 알기**
  - 큰따옴표로 둘러싸인 문자열은 + 연산과 동일하다

```
>>> print("life" "is" "too short") ←①
lifeistoo short
>>> print("life"+"is"+"too short") ←②
lifeistoo short
```

  - 문자열 띄어쓰기는 쉼표로 한다

```
>>> print("life", "is", "too short")
life is too short
```

- 한 줄에 결괏값 출력하기

```
>>> for i in range(10):
...     print(i, end = ' ')
...
0 1 2 3 4 5 6 7 8 9 >>>
```

4. 파일 입출력

- **파일 생성하기**
  - 사용자가 직접 '입력'하고 모니터 화면에 결괏값을 '출력'하는 방법만 있는 것은 아님
  - 파일을 통한 입출력도 가능

```
f = open("새파일.txt", 'w')
f.close()
```

  - 소스코드를 실행하면 프로그램을 실행한 디렉터리에 새로운 파일이 하나 생성됨
  - 파일을 생성하기 위해 파이썬 내장 함수 open을 사용한 것

```
파일_객체 = open(파일_이름, 파일_열기_모드)
```

  - f.close( )는 열려 있는 파일 객체를 닫아 주는 역할(생략 가능하지만 사용하는 것을 추천)

4. 파일 입출력

파일 생성하기

파일 열기 모드

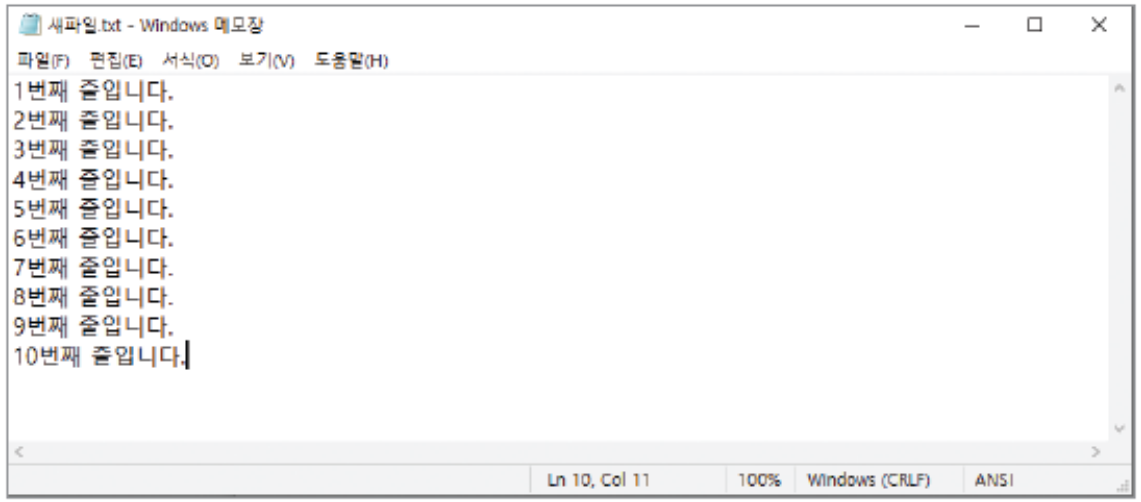
파일 열기 모드	설명
r	읽기 모드: 파일을 읽기만 할 때 사용한다.
w	쓰기 모드: 파일에 내용을 쓸 때 사용한다.
a	추가 모드: 파일의 마지막에 새로운 내용을 추가할 때 사용한다.

- 파일을 쓰기 모드(w)로 열면 해당 파일이 이미 존재할 경우 원래 있던 내용이 모두 사라지고,
  해당 파일이 존재하지 않으면 새로운 파일이 생성됨

4. 파일 입출력

- **파일을 쓰기 모드로 열어 내용 쓰기**
  - 문자열 데이터를 파일에 직접 써서 출력

```
f = open("C:/doit/새파일.txt", 'w')
for i in range(1, 11):
    data = "%d번째 줄입니다.\n" % i
    f.write(data)    # data를 파일 객체 f에 써라.
f.close()
```



4. 파일 입출력

파일을 읽는 여러 가지 방법

readline 함수 사용하기

- f.open("새파일.txt",'r')로 파일을 읽기 모드로 연 후 readline( )을 사용해서 파일의 첫 번째 줄을 읽어 출력하는 코드

```
f = open("C:/doit/새파일.txt", 'r')
line = f.readline()
print(line)
f.close()
```

```
1번째 줄입니다.
```

- 모든 줄을 읽어 화면에 출력하는 코드

```
f = open("C:/doit/새파일.txt", 'r')
while True:
    line = f.readline()
    if not line: break
    print(line)
f.close()
```

  - 무한루프 안에서 f.readline( )을 사용해 파일을 계속 한 줄씩 읽어 들임
  - 더 이상 읽을 줄이 없으면 break 수행
  - readline( )은 더 이상 읽을 줄이 없을 경우 빈 문자열("")을 리턴

4. 파일 입출력

- **파일을 읽는 여러 가지 방법**
  - readlines 함수 사용하기
    - 파일의 모든 줄을 읽어서 각각의 줄을 요소로 가지는 리스트를 리턴
    - ["1번째 줄입니다.\n","2번째 줄입니다.\n",...,"10번째 줄입니다.\n"]를 리턴

```
f = open("C:/doit/새파일.txt", 'r')
lines = f.readlines()
for line in lines:
    print(line)
f.close()
```

4. 파일 입출력

파일을 읽는 여러 가지 방법

read 함수 사용하기

- f.read( )는 파일의 내용 전체를 문자열로 리턴
- data는 파일의 전체 내용

```
f = open("C:/doit/새파일.txt", 'r')
data = f.read()
print(data)
f.close()
```

파일 객체를 for 문과 함께 사용하기

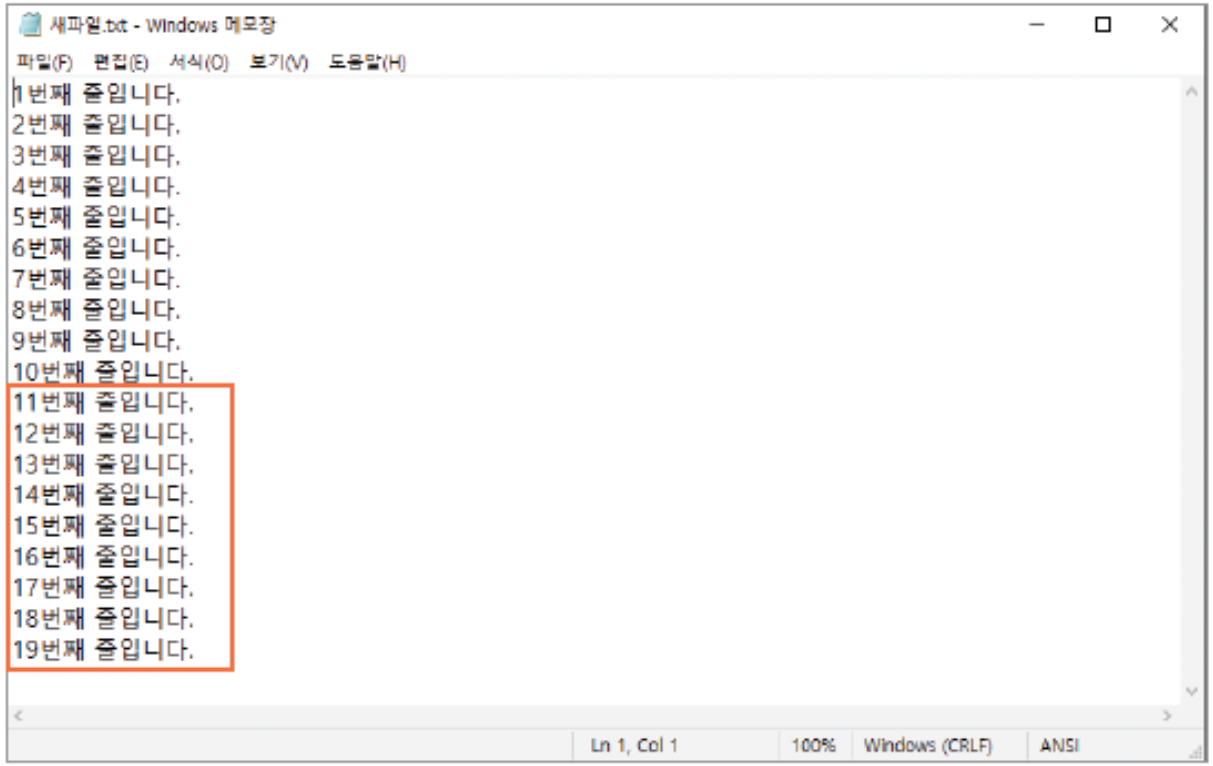
- 파일 객체(f)는 for 문과 함께 사용하여 파일을 줄 단위로 읽을 수 있음

```
f = open("C:/doit/새파일.txt", 'r')
for line in f:
    print(line)
f.close()
```

4. 파일 입출력

- **파일에 새로운 내용 추가하기**
  - 원래 있던 값을 유지하면서 단지 새로운 값만 추가해야 할 경우
  - 파일을 추가 모드('a')로 열기

```
f = open("C:/doit/새파일.txt", 'a')
for i in range(11, 20):
    data = "%d번째 줄입니다.\n" % i
    f.write(data)
f.close()
```



4. 파일 입출력

- **with 문과 함께 사용하기**
  - 지금까지 파일을 열고 닫은 방법

```
f = open("foo.txt", 'w') ← 파일 열기
f.write("Life is too short, you need python")
f.close() ← 파일 닫기
```

  - f.close( )는 열려 있는 파일 객체를 닫아 주는 역할
  - 쓰기 모드로 열었던 파일을 닫지 않고 다시 사용하면 오류가 발생하기 때문에, close( )를 사용해서 열려 있는 파일을 직접 닫아 주는 것이 좋음

4. 파일 입출력

- **with 문과 함께 사용하기**
  - with 문은 파일을 열고 닫는 것을 자동으로 처리해주는 문법
  - 앞선 예제를 with 문을 사용하여 수정한 코드

```
with open("foo.txt", "w") as f:
    f.write("Life is too short, you need python")
```

  - with 문을 사용하면 with 블록을 벗어나는 순간 열린 파일 객체 f가 자동으로 닫힘

5. 프로그램의 입출력

- **sys 모듈 사용하기**
  - 파이썬에서는 sys 모듈을 사용하여 프로그램에 인수 전달 가능
  - import 명령어 사용

```
import sys

args = sys.argv[1:]
for i in args:
    print(i)
```

  - argv는 프로그램 실행 시 전달된 인수

- argv[0]은 파일 이름 sys1.py, argv[1]부터는 뒤에 따라오는 인수가 차례대로 argv의 요소

5. 프로그램의 입출력

- **sys 모듈 사용하기**
  - 전달된 인수를 모두 대문자로 바꾸는 간단한 프로그램 만들기

```
import sys
args = sys.argv[1:]
for i in args:
    print(i.upper(), end=' ')
```

  - 명령 프롬프트

```
C:\doit>python sys2.py life is too short, you need python
```

  - 실행 결과

```
LIFE IS TOO SHORT, YOU NEED PYTHON
```