04장 프로그램의 입력과 출력은 어떻게 해야 할까?

지금껏 공부한 내용을 바탕으로 함수, 입력과 출력, 파일 처리 방법 등에 대해서 알아보기로 하자.

입출력은 프로그래밍 설계와 관련이 있다. 프로그래머는 프로그램을 만들기 전에 어떤 식으로 동작하게 할 것인지 설계 부터 하게 되는데 그때 가장 중요한 부분이 바로 입출력의 설계이다. 특정 프로그램만이 사용하는 함수를 만들 것인지 아니면 모든 프로그램이 공통으로 사용하는 함수를 만들 것인지, 더 나아가 오픈 API로 공개하여 외부 프로그램들도 사용할 수 있게 만들 것인지 그 모든 것이 입출력과 관련이 있다.

마지막 편집일시 : 2017년 5월 1일 2:38 오후

이전글 : 03-3 for문

다음글 : 04-1 함수

위키독스

원본 URL: https://wikidocs.net/24

04-1 함수

함수란 무엇인가?

함수를 설명하기 전에 믹서기를 생각해 보자. 우리는 믹서기에 과일을 넣는다. 그리고 믹서기를 이용해서 과일을 갈아 과일 주스를 만든다. 우리가 믹서기에 넣는 과일은 "입력"이 되고 과일 주스는 "출력(결과값)"이 된다.

그렇다면 믹서기는 무엇인가?

(믹서기는 과일을 입력받아 주스를 출력하는 함수와 같다.)

우리가 배우려는 함수가 바로 믹서기와 비슷하다. 입력값을 가지고 어떤 일을 수행한 다음에 그 결과물을 내어놓는 것, 이것이 바로 함수가 하는 일이다. 우리는 어려서부터 함수에 대해 공부했지만 함수에 관해 깊이 생각해 본 적은 별로 없다. 예를 들어 y = 2x + 3도 함수다. 하지만 이를 수학 시간에 배운 직선 그래프로만 알고 있

지 x에 어떤 값을 넣었을 때 어떤 변화에 의해서 y 값이 나오는지 그 과정에 대해서는 별로 관심을 두지 않았을 것이다.

이제 우리는 함수에 대해서 조금 더 생각해 보는 시간을 가져야 한다. 프로그래밍에 있어서 함수는 정말 중요하기 때문이다. 자, 이제 파이썬 함수의 세계로 깊이 들어가 보자.

함수를 사용하는 이유는 무엇일까?

프로그래밍을 하다 보면 똑같은 내용을 반복해서 작성하고 있는 자신을 발견할 때가 종종 있다. 이때가 바로 함수가 필요한 때이다. 즉, 반복되는 부분이 있을 경우 "반복적으로 사용되는 가치 있는 부분"을 한 뭉치로 묶어서 "어떤 입력값을 주었을 때 어떤 결과값을 돌려준다"라는식의 함수로 작성하는 것이 현명하다.

함수를 사용하는 또 다른 이유는 자신이 만든 프로그램을 함수화하면 프로그램의 흐름을 일목요연하게 볼 수 있기 때문이다. 마치 공장에서 원재료가 여러 공정을 거쳐 하나의 상품이되는 것처럼 프로그램에서도 입력한 값이 여러 함수들을 거치면서 원하는 결과값을 내는 것을 볼 수 있다. 이렇게 되면 프로그램의 흐름도 잘 파악할 수 있고 에러가 어디에서 나는지도 금방 알아차릴 수 있다. 함수를 잘 이용하고 함수를 적절하게 만들 줄 아는 사람이 능력 있는 프로그래머이다.

파이썬 함수의 구조

파이썬 함수의 구조는 다음과 같다.

```
def 함수명(매개변수):
    <수행할 문장1>
    <수행할 문장2>
    ...
```

def는 함수를 만들 때 사용하는 예약어이며, 함수명은 함수를 만드는 사람이 임의로 만들 수 있다. 함수명 뒤 괄호 안의 매개변수는 이 함수에 입력으로 전달되는 값을 받는 변수이다. 이렇게 함수를 정의한 다음 if, while, for문 등과 마찬가지로 함수에서 수행할 문장들을 입력한다.

간단하지만 많은 것을 설명해 주는 다음의 예를 보자.

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

위 함수는 다음과 같이 풀이된다.

> "이 함수의 이름(함수명)은 add이고 입력으로 2개의 값을 받으며 결과값은 2개의 입력값을 더한 값이다."

여기서 return은 함수의 결과값을 돌려주는 명령어이다. 먼저 다음과 같이 add 함수를 만들자.

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

이제 직접 add 함수를 사용해 보자.

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

변수 a에 3, b에 4를 대입한 다음 앞서 만든 add 함수에 a와 b를 입력값으로 넣어 준다. 그리고 변수 c에 add 함수의 결과값을 대입하면 print(c)로 c의 값을 확인할 수 있다.

매개변수와 인수

매개변수(parameter)와 인수(arguments)는 혼용해서 사용되는 헷갈리는 용어이므로 잘 기억해 두기로 하자. 매개변수는 함수에 입력으로 전달된 값을 받는 변수를 의미하고 인수는 함수를 호출할 때 전달하는 입력값을 의미한다.

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

print(add(3, 4))  # 3, 4는 인수
```

[같은 의미를 가진 여러 가지 용어들에 주의하자]

프로그래밍을 공부할 때 어려운 부분 중 하나가 용어의 혼용이라고 할 수 있다. 우리는 공부하면서 원서를 보기도 하고 누군가의 번역본을 보기도 하면서 의미는 같지만 표현이 다른 용어들을 자주 만나게 된다. 한 예로 입력값을 다른 말로 함수의 인수, 입력 인수 등으로 말하기도 하고 결과값을 출력값, 리턴값, 돌려주는 값 등으로 말하기도 한다. 이렇듯 많은 용어들이 여러 가지 다른 말로 표현되지만 의미는 동일한 경우가 많다. 이런 것들을 기억해 놓아야 머리가 덜 아플 것이다.

입력값과 결과값에 따른 함수의 형태

함수는 들어온 입력값을 받아 어떤 처리를 하여 적절한 결과값을 돌려준다.

입력값 ---> **함수** ----> 리턴값

함수의 형태는 입력값과 결과값의 존재 유무에 따라 4가지 유형으로 나뉜다. 자세히 알아보자.

일반적인 함수

입력값이 있고 결과값이 있는 함수가 일반적인 함수이다. 앞으로 여러분이 프로그래밍을 할 때 만들 함수는 대부분 다음과 비슷한 형태일 것이다.

```
def 함수이름(매개변수):
    <수행할 문장>
    ...
    return 결과값
```

다음은 일반적인 함수의 전형적인 예이다.

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

add 함수는 2개의 입력값을 받아서 서로 더한 결과값을 돌려준다.

이 함수를 사용하는 방법은 다음과 같다. 입력값으로 3과 4를 주고 결과값을 돌려받아 보자.

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

이처럼 입력값과 결과값이 있는 함수의 사용법을 정리하면 다음과 같다.

```
결과값을 받을 변수 = 함수명(입력 인수 1, 입력 인수 2, ...)
```

입력값이 없는 함수

입력값이 없는 함수가 존재할까? 당연히 존재한다. 다음을 보자.

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

say라는 이름의 함수를 만들었다. 그런데 매개변수 부분을 나타내는 함수명 뒤의 괄호 안이 비어 있다. 이 함수는 어떻게 사용하는 걸까?

다음을 직접 입력해 보자.

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

위의 함수를 쓰기 위해서는 say()처럼 괄호 안에 아무런 값도 넣지 않아야 한다. 이 함수는 입력값은 없지만 결과값으로 Hi라는 문자열을 돌려준다. a = say()처럼 작성하면 a에 Hi라는 문자열이 대입되는 것이다.

이처럼 입력값이 없고 결과값만 있는 함수는 다음과 같이 사용된다.

```
```

```
결과값을 받을 변수 = 함수명()
```

결과값이 없는 함수

결과값이 없는 함수 역시 존재한다. 다음의 예를 보자.

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

결과값이 없는 함수는 호출해도 돌려주는 값이 없기 때문에 다음과 같이 사용한다.

```
>>> add(3, 4)
3, 4의 합은 7입니다.
```

즉, 결과값이 없는 함수는 다음과 같이 사용한다.

```
함수명(입력 인수1, 입력 인수2, ...)
```

결과값이 진짜 없는지 확인하기 위해 다음의 예를 직접 입력해 보자.

```
>>> a = add(3, 4)
3, 4의 합은 7입니다.
```

아마도 여러분은 "3, 4의 합은 7입니다."라는 문장을 출력해 주었는데 왜 결과값이 없다는 것인지 의아하게 생각할 것이다. 이 부분이 초보자들이 혼란스러워하는 부분이기도 한데 print문은 함수의 구성 요소 중 하나인 <수행할 문장>에 해당하는 부분일 뿐이다. 결과값은 당연히 없다. 결과값은 오직 return 명령어로만 돌려받을 수 있다.

이를 확인해 보자. 돌려받을 값을 a라는 변수에 대입하여 출력해 보면 결과값이 있는지 없는지 알 수 있다.

```
>>> a = add(3, 4)
>>> print(a)
None
```

a의 값은 None이다. None이란 거짓을 나타내는 자료형이라고 언급한 적이 있다. add 함수처럼 결과값이 없을 때 a = add(3, 4)처럼 쓰게 되면 함수 add는 리턴값으로 a 변수에 None을 돌려준다. 이것을 가지고 결과값이 있다고 생각하면 곤란하다.

입력값도 결과값도 없는 함수

입력값도 결과값도 없는 함수 역시 존재한다. 다음의 예를 보자.

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

입력 인수를 받는 매개변수도 없고 return문도 없으니 입력값도 결과값도 없는 함수이다. 이 함수를 사용하는 방법은 단 한 가지이다.

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

즉, 입력값도 결과값도 없는 함수는 다음과 같이 사용한다.

```
함수명()
```

매개변수 지정하여 호출하기

함수를 호출 할 때 매개변수를 지정하여 호출할 수도 있다. 다음의 예를 보자.

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

앞서 알아보았던 add함수이다. 이 함수를 우리는 다음과 같이 매개변수를 지정하여 사용할 수 있다.

```
>>> result = add(a=3, b=7)  # a에 3, b에 7을 전달
>>> print(result)
10
```

매개변수를 지정하면 다음과 같이 순서에 상관없이 사용할 수 있다는 장점이 있다.

```
>>> result = add(b=5, a=3)  # b에 5, a에 3을 전달
>>> print(result)
8
```

입력값이 몇 개가 될지 모를 때는 어떻게 해야 할까?

입력값이 여러 개일 때 그 입력값들을 모두 더해 주는 함수를 생각해 보자. 하지만 몇 개가 입력될지 모를 때는 어떻게 해야 할까? 아마도 난감할 것이다. 파이썬은 이런 문제를 해결하기 위해 다음과 같은 방법을 제공한다.

```
def 함수이름(*매개변수):
    <수행할 문장>
    ...
```

일반적으로 볼 수 있는 함수 형태에서 괄호 안의 매개변수 부분이 `*매개변수`로 바뀌었다.

다음의 예를 통해 여러 개의 입력값을 모두 더하는 함수를 직접 만들어 보자. 예를 들어 add_many(1, 2)이면 3을, add_many(1,2,3)이면 6을, add_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)이면 55를 돌려주는 함수를 만들어 보자.

```
>>> def add_many(*args):
...     result = 0
```

```
...     for i in args:
...         result = result + i
...     return result
...
>>>
```

위에서 만든 add_many라는 함수는 입력값이 몇 개이든 상관이 없다. `*args` 처럼 매개변수명 앞에 `*` 을 붙이면 입력값들을 전부 모아서 튜플로 만들어 주기 때문이다. 만약 add_many(1, 2, 3)처럼 이 함수를 쓰면 args는 (1, 2, 3)이 되고, add_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)처럼 쓰면 args는 (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)이 된다. 여기서 `*args` 라는 것은 임의로정한 변수명이다. `*pey` , `*python` 처럼 아무 이름이나 써도 된다.

> ※ args는 입력 인수를 뜻하는 영어 단어인 arguments의 약자이며 관례적으로 자주 사용한다.

실제로 이 함수를 직접 입력해 보자.

```
>>> result = add_many(1,2,3)
>>> print(result)
6
>>> result = add_many(1,2,3,4,5,6,7,8,9,10)
>>> print(result)
55
```

add_many(1,2,3)으로 함수를 호출하면 6을 리턴하고, add_many(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)을 대입하면 55를 리턴한다.

여러개의 입력을 처리할 때 `def add_many(*args)` 처럼 함수의 매개변수로 `*args` 만 단독으로 사용할 수 있는 것은 아니다.

다음의 예를 보자.

```
>>> def add_mul(choice, *args):
...     if choice == "add":
...         result = 0
...         for i in args:
...             result = result + i
...     elif choice == "mul":
...         result = 1
...         for i in args:
...             result = result * i
...     return result
...
>>>
```

add_mul 함수는 여러개의 입력값을 의미하는 `*args` 매개변수 앞에 choice 매개변수가 추가되어 있다.

이 함수는 다음과 같이 사용할 수 있다.

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

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

매개변수 choice에 'add'가 입력된 경우 `*args`에 입력되는 모든 값을 더해서 15를 돌려주고, 'mul'이 입력된 경우 `*args`에 입력되는 모든 값을 곱해서 120을 돌려준다.

키워드 파라미터 kwargs

이번에는 키워드 파라미터인 `**kwargs`에 대해서 알아보자. kwargs는 keyword arguments의 약어이다. `**kwargs`는 `*args`와는 달리 별표시(`*`)가 두 개 사용된다. 역시 이것도 예제로 알아보도록 하자.

먼저 다음과 같은 함수를 작성 해 보자.

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

이 함수는 매개변수 `kwargs`를 출력하는 함수이다.

그리고 이 함수를 다음과 같이 사용해 보자.

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

`a=1` 또는 `name='foo', age=3` 이라는 입력이 모두 딕셔너리로 만들어져서 출력된다는 것을 확인 할 수 있다.

즉, `**kwargs` 처럼 매개변수명 앞에 `**`을 붙이면 매개변수 kwargs는 딕셔너리가 되고 모든 key=value 형태의 입력인수가 그 딕셔너리에 저장된다.

함수의 결과값은 언제나 하나이다

먼저 다음의 함수를 만들어 보자.

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

※ add_and_mul 함수는 2개의 입력 인수를 받아 더한 값과 곱한 값을 돌려주는 함수이다.

이 함수를 다음과 같이 호출하면 어떻게 될까?

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

결과값은 `a+b`와 `a*b` 2개인데 결과값을 받아들이는 변수는 result 하나만 쓰였으니 오류가 발생하지 않을까? 당연한 의문이다. 하지만 오류는 발생하지 않는다. 그 이유는 함수의 결과값은 2개가 아니라 언제나 1개라는 데 있다. add_and_mul 함수의 결과값 `a+b`와 `a*b`는 튜플값 하나인 `(a+b, a*b)`로 돌려준다.

따라서 result 변수는 다음과 같은 값을 갖게 된다.

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

즉, 결과값으로 (7, 12)라는 튜플 값을 갖게 되는 것이다.

만약 이 하나의 튜플값을 2개의 결과값처럼 받고 싶다면 다음과 같이 함수를 호출하면 된다.

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

이렇게 호출하면 result1, result2 = (7, 12)가 되어 result1은 7이 되고 result2는 12가 된다.

또 다음과 같은 의문이 생길 수도 있다.

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

위와 같이 return문을 2번 사용하면 2개의 결과값을 돌려주지 않을까? 하지만 파이썬에서 위와 같은 함수는 참 어리석은 함수이다.

그 이유는 add_and_mul 함수를 호출해 보면 알 수 있다.

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

※ add_and_mul(2, 3)의 결과값은 5 하나뿐이다. 두 번째 return문인 `return a*b`는 실행되지 않았다는 뜻이다.

이 예에서 볼 수 있듯이 두 번째 return문인 `return a*b`는 실행되지 않았다. 따라서 이 함수는 다음과 완전히 동일하다.

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

즉, 함수는 return문을 만나는 순간 결과값을 돌려준 다음 함수를 빠져나가게 된다.

[return의 또 다른 쓰임새]

어떤 특별한 상황이 되면 함수를 빠져나가고자 할 때는 return을 단독으로 써서 함수를 즉시 빠져나갈 수 있다. 다음 예를 보자.

```
>>> def say_nick(nick):
...     if nick == "바보":
...         return
...     print("나의 별명은 %s 입니다." % nick)
...
>>>
```

위의 함수는 "별명"을 입력으로 전달받아 출력하는 함수이다. 이 함수 역시 리턴값은 없다(문자열을 출력한다는 것과 리턴값이 있다는 것은 전혀 다른 말이다. 혼동하지 말자. 함수의 리턴값은 오로지 return문에 의해서만 생성된다).

만약에 입력값으로 '바보'라는 값이 들어오면 문자열을 출력하지 않고 함수를 즉시 빠져나간다.

```
>>> say_nick('야호')
나의 별명은 야호입니다.
>>> say_nick('바보')
>>>
```

이처럼 return으로 함수를 빠져나가는 방법은 실제 프로그래밍에서 자주 사용된다.

매개변수에 초깃값 미리 설정하기

이번에는 조금 다른 형태로 함수의 인수를 전달하는 방법에 대해서 알아보자. 매개변수에 초깃값을 미리 설정해 주는 경우이다.

※ 앞으로 나올 프로그램 소스에서 >>> 표시가없으면 에디터로 작성하기를 권장한다는 뜻이다.

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

※ say_myself 함수는 3개의 입력 인수를 받아서 마지막 인수인 man이 True이면 "남자입니다.", False이면 "여자입니다."를 출력한다.

위의 함수를 보면 매개변수가 name, old, man=True 이렇게 3개다. 그런데 낯선 것이 나왔다. man=True처럼 매개변수에 미리 값을 넣어 준 것이다. 이것이 바로 함수의 매개변수 초깃값을 설정하는 방법이다. 함수의 매개변수에 전달되는 값이 항상 변하는 것이 아닐 경우에는 이렇게 함수의 초깃값을 미리 설정해 두면 유용하다.

say_myself 함수는 다음처럼 사용할 수 있다.

```
say_myself("박응용", 27)
say_myself("박응용", 27, True)
```

입력값으로 "박응용", 27처럼 2개를 주면 name에는 "박응용"이 old에는 27이 전달된다. 그리고 man이라는 변수에는 입력값을 주지 않았지만 초깃값인 True 값을 갖게 된다.

따라서 위의 예에서 함수를 사용한 2가지 방법은 모두 동일한 결과를 출력한다.

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

이제 초깃값이 설정된 부분을 False로 바꿔 보자.

```
say_myself("박응선", 27, False)
```

man 변수에 False 값이 전달되어 다음과 같은 결과가 출력된다.

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

함수의 매개변수에 초깃값을 설정할 때 주의할 것이 하나 있다. 만약 위에서 본 say_myself 함수를 다음과 같이 만들면 어떻게 될까?

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

이전 함수와 바뀐 부분은 초깃값을 설정한 매개변수의 위치이다. 결론을 미리 말하면 이것은 함수를 실행할 때 오류가 발생한다.

얼핏 생각하기에 위의 함수를 호출하려면 다음과 같이 하면 될 것 같다.

```
say_myself("박응용", 27)
```

위와 같이 함수를 호출한다면 name 변수에는 "박응용"이 들어갈 것이다. 하지만 파이썬 인터프리터는 27을 man 변수와 old 변수 중 어느 곳에 전달해야 할지 알 수 없게 된다.

오류 메시지를 보면 다음과 같다.

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

위의 오류 메시지는 초깃값을 설정해 놓은 매개변수 뒤에 초깃값을 설정해 놓지 않은 매개변수는 사용할 수 없다는 말이다. 즉, 매개변수로 (name, old, man=True)는 되지만 (name, man=True, old)는 안 된다는 것이다. 초기화시키고 싶은 매개변수들을 항상 뒤쪽에 위치시키는 것을 잊지 말자.

함수 안에서 선언된 변수의 효력 범위

함수 안에서 사용할 변수의 이름을 함수 밖에서도 동일하게 사용한다면 어떻게 될까? 이런 궁금증이 생겼던 독자라면 이번에 확실하게 답을 찾을 수 있을 것이다.

아래의 예를 보자.

```
# vartest.py
a = 1
def vartest(a):
    a = a +1

vartest(a)
print(a)
```

먼저 a라는 변수를 생성하고 1을 대입한다. 다음 입력으로 들어온 값에 1을 더해 주고 결과값은 돌려주지 않는 vartest 함수를 선언한다. 그리고 vartest 함수에 입력값으로 a를 주었다. 마지막으로 a의 값을 출력하는 print(a)를 입력한다. 과연 결과값은 무엇이 나올까?

당연히 vartest 함수에서 매개변수 a의 값에 1을 더했으니까 2가 출력되어야 할 것 같지만 프로그램 소스를 작성해서 실행시켜 보면 결과값은 1이 나온다. 그 이유는 함수 안에서 새로 만들어진 매개변수는 함수 안에서만 사용되는 "함수만의 변수"이기 때문이다. 즉, def vartest(a)에서 입력 값을 전달받는 매개변수 a는 함수 안에서만 사용되는 변수이지 함수 밖의 변수 a가 아니라는 뜻이다.

따라서 vartest 함수는 다음처럼 변수 이름을 hello로 한 vartest 함수와 완전히 동일하다.

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

즉, 함수 안에서 사용되는 매개변수는 함수 밖의 변수 이름들과는 전혀 상관이 없다는 말이다.

다음의 예를 보면 더욱 분명하게 이해할 수 있을 것이다.

```
# vartest_error.py
def vartest(a):
    a = a + 1

vartest(3)
print(a)
```

위의 프로그램 소스를 에디터로 작성해서 실행시키면 어떻게 될까? 오류가 발생할 것이라고 생각한 독자는 모든 것을 이해한 독자이다. vartest(3)을 수행하면 vartest라는 함수 내에서 a는 4가 되지만 함수를 호출하고 난 뒤에 print(a)라는 문장은 에러가 발생하게 된다. 그 이유는 print(a)에서 사용된 a라는 변수는 어디에서도 찾을 수가 없기 때문이다. 다시 말하지만 함수 안에서 선언된 매개변수는 함수 안에서만 사용될 뿐 함수 밖에서는 사용되지 않는다. 이것을 이해하는 것은 매우 중요하다.

함수 안에서 함수 밖의 변수를 변경하는 방법

그렇다면 위에서 만든 vartest라는 함수를 이용해서 함수 밖의 변수 a를 1만큼 증가시킬 수 있는 방법은 없을까? 이 질문에는 2가지 해결 방법이 있다.

1. return 이용하기

```
# vartest_return.py
a = 1
def vartest(a):
    a = a +1
    return a

a = vartest(a)
print(a)
```

첫 번째 방법은 return을 이용하는 방법이다. vartest 함수는 입력으로 들어온 값에 1을 더한값을 돌려준다. 따라서 a = vartest(a)라고 대입하면 a가 vartest 함수의 결과값으로 바뀐다. 여기서도 물론 vartest 함수 안의 a 매개변수는 함수 밖의 a와는 다른 것이다.

2. global 명령어 이용하기

```
# vartest_global.py
a = 1
def vartest():
    global a
    a = a+1

vartest()
print(a)
```

두 번째 방법은 global이라는 명령어를 이용하는 방법이다. 위의 예에서 볼 수 있듯이 vartest 함수 안의 global a라는 문장은 함수 안에서 함수 밖의 a 변수를 직접 사용하겠다는 뜻이다. 하지만 프로그래밍을 할 때 global 명령어는 사용하지 않는 것이 좋다. 왜냐하면 함수는 독립적으로 존재하는 것이 좋기 때문이다. 외부 변수에 종속적인 함수는 그다지 좋은 함수가 아니다. 그러므로 가급적 global 명령어를 사용하는 이 방법은 피하고 첫 번째 방법을 사용하기를 권한다.

lambda

lambda는 함수를 생성할 때 사용하는 예약어로 def와 동일한 역할을 한다. 보통 함수를 한줄로 간결하게 만들 때 사용한다. 우리말로는 "람다"라고 읽고 def를 사용해야 할 정도로 복잡하지 않거나 def를 사용할 수 없는 곳에 주로 쓰인다.

사용법은 다음과 같다.

```
lambda 매개변수1, 매개변수2, ... : 매개변수를 이용한 표현식
```

한번 직접 만들어 보자.

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

add는 두개의 인수를 받아 서로 더한 값을 리턴하는 lambda 함수이다. 위의 예제는 def를 사용한 아래 함수와 하는 일이 완전히 동일하다.

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

연습문제

(연습문제 풀이 : https://wikidocs.net/17090#04-1)

[문제1] 홀수 짝수 판별

주어진 자연수가 홀수인지 짝수인지 판별해 주는 함수(is_odd)를 작성하시오.

[문제2] 평균값 계산

입력으로 들어오는 모든 수의 평균값을 계산해 주는 함수를 작성해 보자. (단, 입력으로 들어오는 수의 갯수는 정해져 있지 않다.)

마지막 편집일시 : 2018년 12월 30일 9:34 오후

이전글 : 04장 프로그램의 입력과 출력은 어떻게 해야 할까?

다음글 : 04-2 사용자 입력과 출력

04-2 사용자 입력과 출력

우리들이 사용하는 대부분의 완성된 프로그램은 사용자의 입력에 따라 그에 맞는 출력을 내보낸다. 대표적인 예로 게시판에 글을 작성한 후 "확인" 버튼을 눌러야만 (입력) 우리가 작성한 글이 게시판에 올라가는(출력) 것을 들 수 있다.

```
사용자 입력 → 처리(프로그램, 함수 등) → 출력
```

우리는 이미 함수 부분에서 입력과 출력이 어떤 의미를 가지는지 알아보았다. 지금부터는 좀 더 다양하게 사용자의 입력을 받는 방법과 출력하는 방법을 알아보자.

사용자 입력

사용자가 입력한 값을 어떤 변수에 대입하고 싶을 때는 어떻게 해야 할까?

input의 사용

```
>>> a = input()
Life is too short, you need python
>>> a
'Life is too short, you need python'
>>>
```

input은 입력되는 모든 것을 문자열로 취급한다.

프롬프트를 띄워서 사용자 입력 받기

사용자에게 입력을 받을 때 "숫자를 입력하세요"라든지 "이름을 입력하세요"라는 안내 문구 또는 질문이 나오도록 하고 싶을 때가 있다. 그럴 때는 input()의 괄호 안에 질문을 입력하여 프롬프트를 띄워주면 된다.

```
input("질문 내용")
```

다음의 예를 직접 입력해 보자.

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

위와 같은 질문을 볼 수 있을 것이다.

숫자를 입력하라는 프롬프트에 3을 입력하면 변수 number에 3이 대입된다. print(number)로 출력해서 제대로 입력되었는지 확인해 보자.

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

print 자세히 알기

지금껏 print문이 수행해온 일은 우리가 입력한 자료형을 출력하는 것이었다. 그간 알아보았던 print의 사용 예는 다음과 같다.

```
>>> a = 123
>>> print(a)
123
>>> a = "Python"
>>> print(a)
Python
>>> a = [1, 2, 3]
>>> print(a)
[1, 2, 3]
```

이제 print문으로 할 수 있는 일에 대해서 조금 더 자세하게 알아보기로 하자.

큰따옴표(")로 둘러싸인 문자열은 + 연산과 동일하다

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

위의 예에서 ①과 ②는 완전히 동일한 결과값을 출력한다. 즉, 따옴표로 둘러싸인 문자열을연속해서 쓰면 + 연산을 한 것과 같다.

문자열 띄어쓰기는 콤마로 한다

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

콤마(,)를 이용하면 문자열 간에 띄어쓰기를 할 수 있다.

한 줄에 결과값 출력하기

03-3절에서 for문을 배울 때 만들었던 구구단 프로그램에서 보았듯이 한 줄에 결과값을 계속 이어서 출력하려면 입력 인수 end를 이용해 끝 문자를 지정해야 한다.

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

연습문제

(연습문제 풀이 : https://wikidocs.net/17090#04-2)

[문제1] 두 수의 합은?

다음은 두 개의 숫자를 입력받아 더하여 리턴해 주는 프로그램이다.

```
input1 = input("첫번째 숫자를 입력하세요:")
input2 = input("두번째 숫자를 입력하세요:")

total = input1 + input2
print("두 수의 합은 %s 입니다" % total)
```

이 프로그램을 수행 해 보자.

```
첫번째 숫자를 입력하세요:3
두번째 숫자를 입력하세요:6
두 수의 합은 36 입니다
```

3과 6을 입력했을 때 9가 아닌 36이라는 결과가 리턴되었다. 이 프로그램의 오류를 수정하시오.

[문제2] 숫자의 총합

사용자로부터 다음과 같은 숫자들의 입력을 받아 입력받은 숫자들의 총합을 구하는 프로그램을 작성하시오. (단, 숫자들은 콤마로 구분하여 입력한다.)

```
65,45,2,3,45,8
```

[문제3] 문자열 출력

다음 중 출력결과가 다른 것 한개를 고르시오.

1. `print("you" "need" "python")`
2. `print("you"+"need"+"python")`
3. `print("you", "need", "python")`
4. `print("".join(["you", "need", "python"]))`

[문제4] 한줄 구구단

사용자로부터 2~9 까지의 숫자중 하나를 입력받아 해당 숫자의 구구단을 한줄로 출력하는 프로그램을 작성하시오.

실행 예

```
구구단을 출력할 숫자를 입력하세요(2~9): 2
2 4 6 8 10 12 14 16 18
```

마지막 편집일시 : 2018년 12월 29일 10:09 오후

댓글 25 피드백

이전글 : 04-1 함수

다음글 : 04-3 파일 읽고 쓰기

04-3 파일 읽고 쓰기

우리는 이 책에서 이제까지 "입력"을 받을 때는 사용자가 직접 입력하는 방식을 사용했고, "출력"할 때는 모니터 화면에 결과값을 출력하는 방식으로 프로그래밍해 왔다. 하지만 입출력 방법이 꼭 이것만 있는 것은 아니다. 이번에는 파일을 통한 입출력 방법에 대해서 알아보자. 먼저 파일을 새로 만든 다음 프로그램에 의해서 만들어진 결과값을 새 파일에 한번 적어 보고, 또 파일에 적은 내용을 읽어 보는 프로그램을 만드는 것으로 시작해 보자.

파일 생성하기

다음 소스 코드를 에디터로 작성해서 저장한 후 실행해 보자. 프로그램을 실행한 디렉터리에 새로운 파일이 하나 생성된 것을 확인할 수 있을 것이다

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

파일을 생성하기 위해 우리는 open이라는 파이썬 내장 함수를 사용했다. open 함수는 다음과 같이 "파일 이름"과 "파일 열기 모드"를 입력값으로 받고 결과값으로 파일 객체를 돌려준다.

```
파일 객체 = open(파일 이름, 파일 열기 모드)
```

파일 열기 모드에는 다음과 같은 것들이 있다.

파일열기모드	설명
r	읽기모드 - 파일을 읽기만 할 때 사용
w	쓰기모드 - 파일에 내용을 쓸 때 사용
a	추가모드 - 파일의 마지막에 새로운 내용을 추가 시킬 때 사용

파일을 쓰기 모드로 열게 되면 해당 파일이 이미 존재할 경우 원래 있던 내용이 모두 사라지고, 해당 파일이 존재하지 않으면 새로운 파일이 생성된다. 위의 예에서는 디렉터리에 파일이 없는 상태에서 새파일.txt를 쓰기 모드인 'w'로 열었기 때문에 새파일.txt라는 이름의 새로운 파일이 현재 디렉터리에 생성되는 것이다.

만약 새파일.txt라는 파일을 `C:/doit` 이라는 디렉터리에 생성하고 싶다면 다음과 같이 작성해야 한다.

```
f = open("C:/doit/새파일.txt", 'w')
f.close()
```

위의 예에서 f.close()는 열려 있는 파일 객체를 닫아 주는 역할을 한다. 사실 이 문장은 생략해도 된다. 프로그램을 종료할 때 파이썬 프로그램이 열려 있는 파일의 객체를 자동으로 닫아주기 때문이다. 하지만 close()를 사용해서 열려 있는 파일을 직접 닫아 주는 것이 좋다. 쓰기모드로 열었던 파일을 닫지 않고 다시 사용하려고 하면 오류가 발생하기 때문이다.

파일을 쓰기 모드로 열어 출력값 적기

위의 예에서는 파일을 쓰기 모드로 열기만 했지 정작 아무것도 쓰지는 않았다. 이번에는 에디터를 열고 프로그램의 출력값을 파일에 직접 써 보자.

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

위의 프로그램을 다음 프로그램과 비교해 보자.

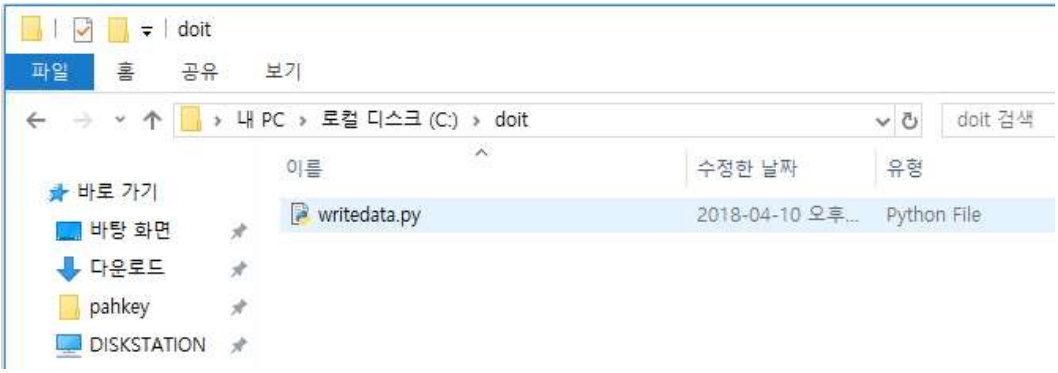
```
for i in range(1, 11):
    data = "%d번째 줄입니다.\n" % i
    print(data)
```

두 프로그램의 다른 점은 data를 출력하는 방법이다. 두 번째 방법은 우리가 계속 사용해 왔던 모니터 화면에 출력하는 방법이고, 첫 번째 방법은 모니터 화면 대신 파일에 결과값을 적는 방법이다. 두 방법의 차이점은 print 대신 파일 객체 f의 write 함수를 이용한 것 말고는 없으니 금방 눈에 들어올 것이다.

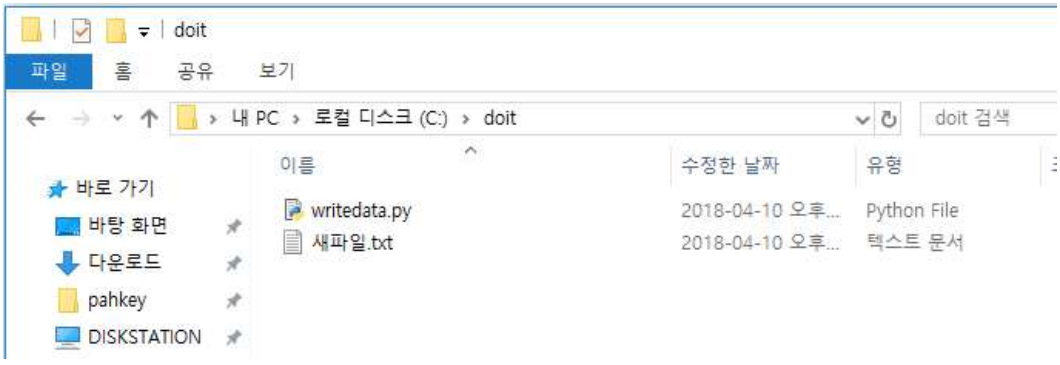
이제 첫 번째 예제를 실행해 보자.

```
C:\Users> cd C:\doit
C:\doit>python writedata.py
C:\doit>
```

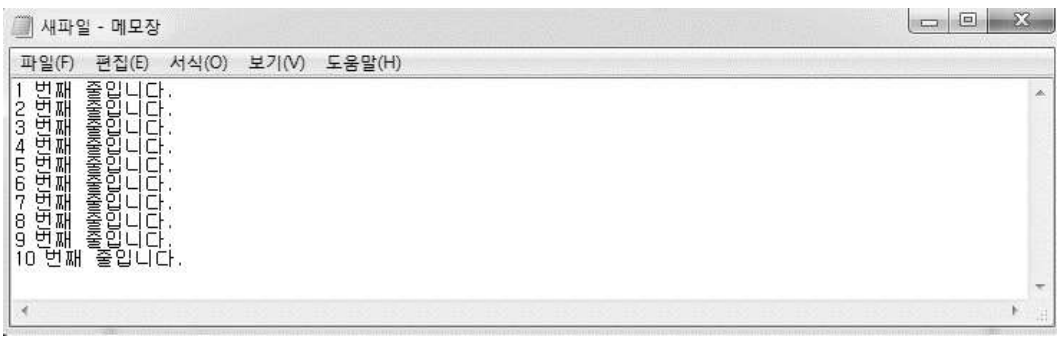
이 프로그램을 실행시킨 `C:/doit` 디렉터리를 살펴보면 새파일.txt라는 파일이 생성되었음을 확인할 수 있다.



위처럼 디렉터리에 파일이 없다가 다음과 같이 생성된다.



새파일.txt 파일에는 어떤 내용이 담겨 있는지 확인해 보자.



모니터 화면에 바로 출력될 내용이 고스란히 파일에 들어가 있는 것을 볼 수 있다.

프로그램의 외부에 저장된 파일을 읽는 여러 가지 방법

파이썬에는 외부 파일을 읽어 들여 프로그램에서 사용할 수 있는 여러 가지 방법이 있다. 이번에는 그 방법들을 자세히 알아보자.

readline() 함수 이용하기

첫 번째 방법은 readline() 함수를 이용하는 방법이다. 다음의 예를 보자.

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

위 예는 f.open("새파일.txt", 'r')로 파일을 읽기 모드로 연 후 readline()을 이용해서 파일의 첫 번째 줄을 읽어 출력하는 경우이다. 이전에 만들었던 새파일.txt를 수정하거나 지우지 않았다면 위 프로그램을 실행시켰을 때 새파일.txt의 가장 첫 번째 줄이 화면에 출력될 것이다.

```
1번째 줄입니다.
```

만약 모든 라인을 읽어서 화면에 출력하고 싶다면 다음과 같이 작성하면 된다.

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

즉, while True:라는 무한 루프 안에서 f.readline()을 이용해 파일을 계속해서 한 줄씩 읽어들이도록 한다. 만약 더 이상 읽을 라인이 없으면 break를 수행한다(readline()은 더 이상 읽을 라인이 없을 경우 None을 출력한다).

앞의 프로그램을 다음 프로그램과 비교해 보자.

```
while 1:
    data = input()
    if not data: break
    print(data)
```

위의 예는 사용자의 입력을 받아서 그 내용을 출력하는 경우이다. 파일을 읽어서 출력하는 예제와 비교해 보자. 입력을 받는 방식만 다르다는 것을 금방 알 수 있을 것이다. 두 번째 예는 키보드를 이용한 입력 방법이고, 첫 번째 예는 파일을 이용한 입력 방법이다.

readlines() 함수 이용하기

두 번째 방법은 readlines() 함수를 이용하는 방법이다. 다음의 예를 보자.

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

readlines() 함수는 파일의 모든 라인을 읽어서 각각의 줄을 요소로 갖는 리스트로 리턴한다. 따라서 위의 예에서 lines는 `["1 번째 줄입니다.\n","2 번째 줄입니다.\n",..., "10 번째 줄입니다.\n"]` 라는 리스트가 된다. f.readlines()에서 f.readline()과는 달리 s가 하나 더 붙어 있음에 유의하자.

read() 함수 이용하기

세 번째 방법은 read() 함수를 이용하는 방법이다. 다음의 예를 보자.

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

f.read()는 파일의 내용 전체를 문자열로 리턴한다. 따라서 위 예의 data는 파일의 전체 내용이다.

파일에 새로운 내용 추가하기

쓰기 모드('w')로 파일을 열 때 이미 존재하는 파일을 열 경우 그 파일의 내용이 모두 사라지게 된다고 했다. 하지만 원래 있던 값을 유지하면서 단지 새로운 값만 추가해야 할 경우도 있다. 이런 경우에는 파일을 추가 모드('a')로 열면 된다. 에디터를 켜고 다음 소스 코드를 작성해 보자.

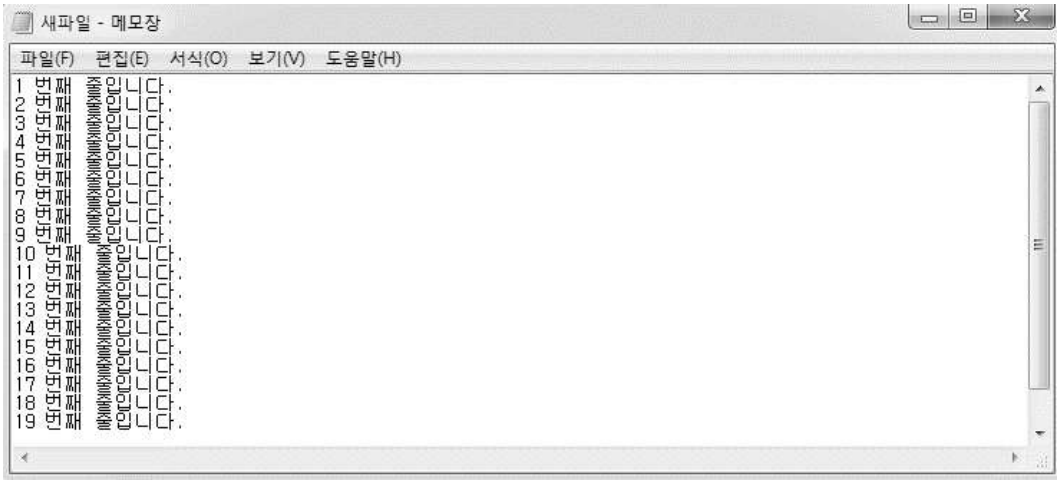
```
# adddata.py
f = open("C:/doit/새파일.txt",'a')
for i in range(11, 20):
    data = "%d번째 줄입니다.\n" % i
    f.write(data)
f.close()
```

위 예는 새파일.txt라는 파일을 추가 모드('a')로 열고 write를 이용해서 결과값을 기존 파일에 추가해 적는 예이다. 여기서 추가 모드로 파일을 열었기 때문에 새파일.txt라는 파일이 원래 가지고 있던 내용 바로 다음부터 결과값을 적기 시작한다.

다음과 같이 작성한 코드를 실행해 보자.

```
C:\Users> cd C:\doit
C:\doit>python adddata.py
C:\doit>
```

그리고 새파일.txt 파일을 확인해 보면 원래 있던 파일 뒷부분에 새로운 내용이 추가되었음을 볼 수 있다.



with문과 함께 사용하기

지금까지 살펴본 예제를 보면 항상 다음과 같은 방식으로 파일을 열고 닫아 왔다.

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

파일을 열면 위와 같이 항상 close해 주는 것이 좋다. 하지만 이렇게 파일을 열고 닫는 것을 자동으로 처리할 수 있다면 편리하지 않을까?

파이썬의 with문이 바로 이런 역할을 해준다. 다음의 예는 with문을 이용해서 위 예제를 다시 작성한 모습이다.

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

위와 같이 with문을 이용하면 with 블록을 벗어나는 순간 열린 파일 객체 f가 자동으로 close되어 편리하다.

※ with구문은 파이썬 2.5부터 지원됨

[sys 모듈로 입력 인수 주기]

도스(DOS)를 사용해 본 독자라면 다음과 같은 명령어를 사용해 봤을 것이다.

```
C:\> type a.txt
```

위의 type 명령어는 바로 뒤에 적힌 파일 이름을 인수로 받아 그 내용을 출력해 주는 도스 명령어이다. 대부분의 도스 명령어들은 다음과 같이 명령행(도스창)에서 입력 인수를 직접 주어 프로그램을 실행시키는 방식을 따른다. 이러한 기능을 파이썬 프로그램에도 적용시킬 수가 있다.

도스 명령어 [인수1 인수2 ...]

파이썬에서는 sys라는 모듈을 이용하여 입력 인수를 직접 줄수 있다. sys 모듈을 이용하려면 아래 예의 import sys처럼 import라는 명령어를 사용해야 한다.

※ 모듈을 이용하고 만드는 방법에 대해서는 뒤에서 자세히 다룰 것이다.

```
#sys1.py
import sys

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

위의 예는 입력받은 인수들을 for문을 이용해 차례대로 하나씩 출력하는 예이다. sys 모듈의 argv는 명령창에서 입력한 인수들을 의미한다. 즉, 아래와 같이 입력했다면 argv[0]는 파일 이름인 sys1.py가 되고 argv[1]부터는 뒤에 따라오는 인수들이 차례로 argv의 요소가 된다.

이 프로그램을 C:/doit 디렉터리에 저장한 후 입력 인수를 함께 주어 실행시키면 다음과 같은 결과값을 얻을 수 있다.

```
C:\doit>python sys1.py aaa bbb ccc
aaa
bbb
ccc
```

위의 예를 이용해서 간단한 스크립트를 하나 만들어 보자.

```
#sys2.py
import sys
args = sys.argv[1:]
for i in args:
    print(i.upper(), end=' ')
```

문자열 관련 함수인 upper()를 이용하여 명령 행에 입력된 소문자를 대문자로 바꾸어 주는 간단한 프로그램이다. 명령 프롬프트 창에서 다음과 같이 입력해 보자.

※ sys2.py 파일이 `C:\doit` 디렉터리 안에있어야만 한다.

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

결과는 다음과 같다.

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

연습문제

(연습문제 풀이 : https://wikidocs.net/17090#04-3)

[문제1] 파일 읽고 출력하기

다음은 "test.txt"라는 파일에 "Life is too short" 라는 문자열을 저장한 후 다시 그 파일을 읽어서 출력하는 프로그램이다.

```
f1 = open("test.txt", 'w')
f1.write("Life is too short")

f2 = open("test.txt", 'r')
print(f2.read())
```

이 프로그램은 우리가 예상한 "Life is too short"라는 문장을 출력하지 않는다. 우리가 예상한 값을 출력할 수 있도록 프로그램을 수정하시오.

[문제2] 파일저장

사용자의 입력을 파일(test.txt)에 저장하는 프로그램을 작성하시오. (단, 프로그램을 다시 실행하더라도 기존 작성한 내용을 유지하고 새로 입력한 내용이 추가되어야 한다.)

다음은 실행 예제이다.

```
저장할 내용을 입력하세요:
```

실행 할 때마다 사용자가 입력한 내용이 test.txt파일에 추가되어야 한다.

[문제3] 역순 저장

다음과 같은 내용의 파일 abc.txt가 있다.

abc.txt

```
AAA
BBB
CCC
DDD
EEE
```

이 파일의 내용을 다음과 같이 역순으로 바꾸어 저장하시오.

```
EEE
DDD
```

```
CCC
BBB
AAA
```

[문제4] 파일 수정

다음과 같은 내용을 지닌 파일 test.txt 가 있다.

test.txt

```
Life is too short
you need java
```

이 파일의 내용중 java라는 문자열을 python으로 바꾸어서 저장하시오.

[문제5] 평균값 구하기

다음과 같이 총 10줄로 이루어진 sample.txt 파일이 있다.

sample.txt

```
70
60
55
75
95
90
80
80
85
100
```

sample.txt 파일의 숫자값을 모두 읽어 총합과 평균값을 구한 후 평균값을 result.txt 라는 파일에 쓰는 프로그램을 작성해 보자.

마지막 편집일시 : 2019년 2월 23일 1:58 오후

이전글 : 04-2 사용자 입력과 출력

다음글 : 05장 파이썬 날개달기