05. 파이썬 함수와 모듈

어느 정도 규모의 프로그램을 개발하다 보면 중복으로 사용되는 코드가 생깁니다. 이럴 때 보통 프로그래밍 경험이 많지 않은 초보자 분들은 이전에 작성된 코드를 단순히 복사한 후 붙여넣어서 프로그램을 작성하곤 합니다. 그러나 이런 식으로 프로그램을 작성하게 되면 점점 코드의 유지 보수가 어려워집니다.

예를 들어, 어느 날 이전에 작성한 코드에 문제가 있다는 사실을 알게 되어 해당 문제를 수정한다고 하더라도 이미 해당 코드는 프로그램의 여러 곳에서 복사되어 사용되고 있을 수 있습니다. 이 경우 해당 코드를 하나도 빠짐없이 찾아서 제대로 수정해야 할 것입니다. 어떤가요? 생각만 해도 끔찍하겠죠?

이러한 문제를 막고자 중복 코드를 재사용하고 싶을 때는 단순히 코드를 복사해서 붙여넣는 식으로 프로그램을 작성하는 것이 아니라 함수(function)라고 하는 형태로 코드를 작성하는 것이 좋습니다. 함수라는 것은 한번 잘 작성해두면 두고두고 이를 '호출'해서 사용할 수 있으므로 개발 생산성을 크게 높일 수 있습니다.

파이썬에서 화면에 값을 출력할 때 사용했던 print()라는 함수는 여러분이 직접 구현하지 않았음에도 print() 함수를 호출해서 사용했던 것을 생각해보면 함수가 얼마나 유용한지 알 수 있습니다. 리스트나 튜플에 들어있는 원소의 개수를 확인할 때 사용하는 내장 함수 len()도 기억나실 겁니다. 이번 장에서는 파이썬에서 제공되는 기본 함수들을 살펴보고 유용한 기능을 수행하는 함수도 만들어 보겠습니다.

아울러 파이썬의 모듈에 대해서도 알아보겠습니다. 파이썬의 모듈(module)은 함수보다 더 큰 단위의 코드 묶음을 의미합니다. 보통 함수가 수십 줄 내의 코드로 구성된다면 파이썬의 모듈은 파일 단위의 코드 묶음을 의미합니다. 좀 더 쉽게 설명하자면 우리가 마이크로소프트 워드로 글을 쓴다고 했을 때 한 문단이 '함수'에 해당한다면 워드 파일 하나가 바로 모듈에 해당합니다.

1) 함수

지금까지 파이썬의 변수, 자료구조(리스트, 튜플, 딕셔너리), 분기문(if, else), 반복문(for, while) 등을 배웠습니다. 이것들은 어떤 프로그래밍 언어를 배우더라도 공통으로 존재하는 필수 구성 요소로서 프로그래밍 언어마다 약간의 차이만 있을 뿐입니다. 따라서 2 장 ~ 4 장까지의 내용을 반복적인 실습을 통해 익히는 것이 매우 중요합니다. 사실 프로그래밍이라는 것은 처음에는 어렵지만 반복해서 코드를 입력해서 실행하고, 결괏값을 확인하는 과정을 통해 실력이 향상된답니다.

서론이 좀 길었는데, 이번 장에서는 함수(function)를 배우겠습니다. 함수에 대해서는 아마 초등학교나 중학교 수학 시간에 이미 배웠을 것입니다. 수학 분야에서는 그림 5.1 처럼 입력으로 어떤 값을 넣으면 내부적으로 어떤 처리를 수행하고 최종 결괏값을 출력하는 것을 함수라고 합니다. 여기서 중요한 점은 함수 내부가 어떻게 구현돼 있는지 몰라도 입력을 넣으면 뭔가 출력이 나온다는 점입니다. 그래서 보통 함수 내부는 블랙박스로 생각하고 단순히 입력값과 출력값의 관계만을 생각합니다.

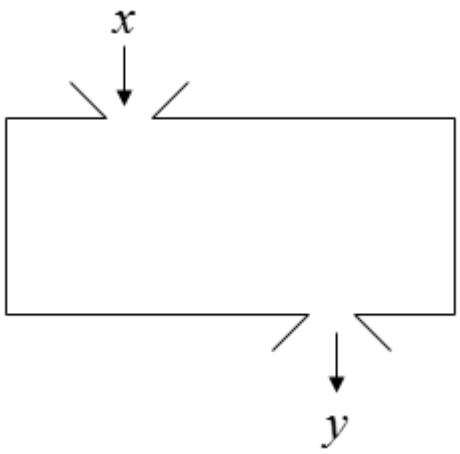

그림 5.1 함수의 입력과 출력

그렇다면 프로그래밍 분야에서 함수란 어떤 것일까요? 결론부터 말씀드리면 프로그래밍 분야의 함수도 수학 분야의 함수와 같습니다. 단지 프로그래머들은 입력값을 넣었을 때 원하는 출력값을 만들어주는 함수를 지금까지 배운 파이썬 문법을 이용해 구현합니다. 프로그래밍 분야에서도 함수 내부의 구현은 블랙박스와 같아서 함수를 사용하는 입장에서는 이를 알 필요가 없습니다. 단순히 함수에 입력값을 넣으면 우리가 원하는 출력값을 얻을 수 있다는 점을 아는 것이 중요합니다.

지금부터는 코드를 작성해보면서 함수를 익혀 보겠습니다. 파이썬 IDLE 를 이용해 화면에 "대신증권"을 출력해 보기 바랍니다.

```
>>> print("대신증권")
대신증권
>>>
```

'대신증권'이라는 문자열을 화면에 출력하신 분들은 이번에는 '대신증권'을 화면에 세 번 출력해보기 바랍니다. 3 번으로 횟수가 정해져 있으므로 반복문 중 for 문을 이용하면 쉬울 것 같습니다.

```
>>> for i in range(3):
        print("대신증권")
```

```
대신증권
대신증권
대신증권
>>>
```

이번에는 함수를 이용해 '대신증권'을 화면에 출력하는 프로그램을 작성하겠습니다. 단순히 '대신증권'을 3 번 출력하는 것이 아니라 함수의 입력으로 출력할 '횟수'를 받은 후 그 '횟수'만큼 '대신증권'을 출력하는 함수를 만들어 보겠습니다.

파이썬에서 함수를 만들 때 중요한 점은 함수의 입력, 출력, 함수의 동작을 파악하는 것입니다. 아래와 같이 대략적인 함수 명세서를 구성해봅시다. 참고로 프로그램을 작성할 때 무조건 손이 가는 대로 프로그램을 작성하기보다는 잠시 컴퓨터에서 떨어져서 구현할 내용에 대해 생각하고 어떻게 구현할지를 설계한 후 프로그래밍하는 습관을 기르는 것이 좋습니다.

```
함수의 입력: 출력 횟수
함수의 출력: '대신증권'이라는 문자열(횟수만큼)
함수의 동작: 출력 횟수만큼 '대신증권' 문자열을 출력
```

파이썬에서는 함수를 만들 때 def 라는 키워드를 사용합니다. 참고로 def 는 definition 이라는 영어 단어의 줄임말입니다. definition 에 '정의'라는 뜻이 있는 것을 생각하면 파이썬에서 함수를 정의할 때 def 키워드를 사용하는 것은 적절한 작명센스 같습니다.

변수명을 적절히 지정해야 하는 것처럼 def 키워드 다음에는 함수의 기능과 연관된 적당한 함수명을 지정해야 합니다. 여기서는 다음과 같이 함수명을 print_ntimes 이라고 작성했습니다.

```
>>> def print_ntimes():
        print("대신증권")
```

```
>>>
```

함수명('print_ntimes') 다음에는 함수의 입력 값을 넣을 괄호가 나오는데, 현재 구현에서는 입력 값이 없으므로 괄호 안에 아무 값도 없습니다. 함수의 몸통 즉, 함수의 구현 부분을 살펴보면 화면에 '대신증권'을 출력하기 위한 print 문이 있습니다. 파이썬에서 함수를 정의할 때도 분기문, 반복문처럼 키워드가 있는 줄의 끝에 콜론(:)이 있고, 실행될 코드 블록은 들여쓰기돼 있어야 합니다.

위에서 구현한 함수는 앞서 작성한 함수 명세에 대한 구현과는 상당한 거리가 있습니다. 일단 함수의 입력으로 출력 횟수를 받아야 하는데 해당 기능이 구현되어 있지 않습니다. 먼저 출력 횟수를 의미하는 n 변수를 함수의 입력으로 받도록 코드를 수정합니다. n 과 같이 함수에 어떤 값을 넘겨줄 때 사용하는 변수를 함수 파라미터 또는 함수 인자라고 합니다.

```
>>> def print_ntimes(n):
        print("대신증권")
```

```
>>>
```

파이썬에서 함수라는 것을 일단 만들기는 했으니, 만들어진 함수를 호출해보겠습니다. 파이썬에서 기본적으로 제공하는 print 함수를 사용할 때 print("hello")처럼 함수명을 입력했던 것처럼 함수를 호출하려면 여러분이 만든 함수의 이름을 파이썬 프롬프트에 입력하면 됩니다. 물론, 여러분이 만든 함수는 인자 값이 하나 있으므로 print_ntimes()가 아니라 print_ntimes(3)처럼 괄호 안에 값을 하나 넣어야 합니다.

```
>>> print_ntimes(3)
대신증권
>>> print_ntimes(2)
대신증권
>>>
```

위 코드를 보면 뭔가 제대로 동작하지 않는 것을 바로 알 수 있습니다. 함수를 호출할 때 함수의 인자 값으로 출력 횟수를 입력했지만(각각 3 과 2) 입력된 출력 횟수와는 상관없이 '대신증권'이라는 문자열은 항상 한 번만 출력됩니다. 이것은 그림 5.2 처럼 함수의 입력 값인 n 값과 상관없이 함수 본체에서는 항상 한 번만 출력하도록 구현했기 때문입니다.

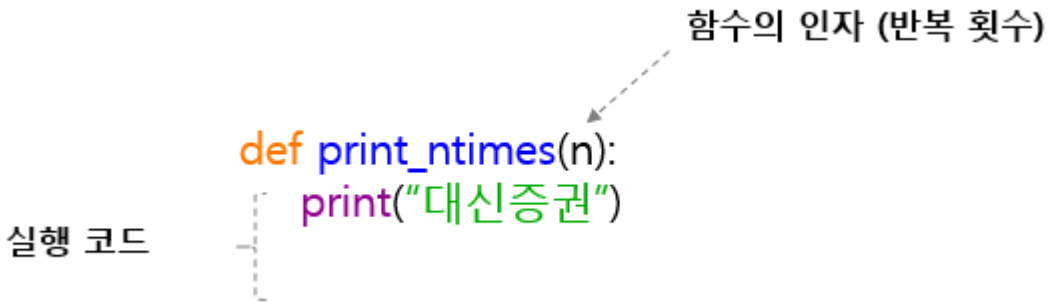

그림 5.2 파이썬 함수 호출 예

함수의 입력 값인 n 만큼 문자열이 출력되게 하려면 함수 내에서 for 문을 사용하면 됩니다. for 와 range()를 사용하여 반복문을 만들고 반복문에서 수행할 문장에 print("대신증권")을 적어주면 됩니다. for 문을 3 번 반복하게 하라면 range(3)을 사용하면 되는데 print_ntimes 함수에서는 함수 인자 n 을 통해 반복 횟수가 입력되므로 함수 인자인 n 만큼 반복 하려면 range(n)을 사용하면 됩니다.

```
>>> def print_ntimes(n):
        for i in range(n):
            print("대신증권")

>>>
```

수정한 함수가 제대로 동작하는지 테스트해 보겠습니다. 잠깐 함수 내부의 구현은 잊고 함수의 입력 값으로 출력 횟수를 넣었을 때 출력 횟수만큼 '대신증권'이 출력되는지 확인해보기 바랍니다. 다음과 같이 함수를 호출해보면 처음으로 만든 함수치고는 상당히 잘 동작하는 것을 확인할 수 있습니다.

```
>>> print_ntimes(1)
대신증권
>>> print_ntimes(2)
대신증권
대신증권
>>> print_ntimes(3)
대신증권
대신증권
```

대신증권

>>>

이처럼 프로그래밍 분야에서 함수라는 것은 처음에 한 번만 잘 만들어 두면 두고두고 호출하는 식으로 원하는 기능을 실행할 수 있습니다. 사실 프로그래밍할 때 꼭 모든 함수를 직접 만들 필요는 없습니다. 여러분 주변이나 인터넷 세상의 누군가가 이미 특정 기능을 하는 함수를 잘 만들어 뒀다면 여러분은 단지 그 함수를 잘 호출하기만 하면 되는 것입니다. 물론 이미 구현된 함수의 인자로 입력 값을 잘 넣어야 올바른 출력 값이 나오므로 함수를 잘 사용하는 것은 여러분의 몫입니다.

2) 반환값이 있는 함수

이번에는 어떤 주식 종목의 전날 종가를 입력받아 상한가(30%)를 계산하고 그 값을 반환(return)하는 함수를 작성해 보겠습니다. 이번 절에서 배울 내용 중 중요한 부분은 앞에서 배운 함수와 달리 함수에 반환값이 있다는 점입니다. 참고로 5.1 절에서 작성한 함수는 함수를 호출했을 때 함수의 결과가 화면에 '출력'됐지 어떤 결괏값이 '반환'된 것은 아니었습니다.

먼저 상한가를 계산하는 함수의 이름을 cal_upper 라고 하고 함수의 입력으로 전날 종가를 받도록 구현해 보겠습니다. 함수 내부는 아직 어떻게 구현하면 좋을지 감이 오지 않으므로 임시방편으로 앞에서 배운 pass 를 사용해 다음과 같이 작성합니다.

```
>>> def cal_upper(price):
        pass

>>>
```

새로 함수를 만들었으니 함수를 호출해 보겠습니다. 10,000 원에 대한 상한가를 계산하기 위해 cal_upper(10000)을 호출했지만 아직은 아무런 동작도 하지 않는 것을 확인할 수 있습니다.

```
>>> cal_upper(10000)
>>>
```

앞서 설명한 것처럼 상한가는 하루에 최대한 오를 수 있는 한도로 2015 년 6 월 15 일부터는 전날 종가에서 30% 오른 금액입니다. 따라서 상한가를 계산하려면 먼저 전날 종가의 30%에 해당하는 금액을 계산해야 합니다. 다음 코드에서 increment 변수는 전날 종가에 해당하는 price 에 0.3 을 곱한 결괏값을 바인딩하고 있습니다.

```
>>> def cal_upper(price):
        increment = price * 0.3

>>>
```

위 코드에서 increment 변수가 전날 종가의 30%에 해당하는 금액을 바인딩하고 있으므로 전날 종가(price)에 increment('전날 종가의 30%')를 더하면 대략적인 상한가를 계산할 수 있습니다. (앞서 설명한 것처럼 정확한 상한가 계산은 호가에 따른 절사 과정을 고려해야 합니다.) 지금까지 설명한 내용을 파이썬 코드로 나타내면 다음과 같습니다.

```
>>> def cal_upper(price):
        increment = price * 0.3
        upper_price = price + increment

>>>
```

위 코드를 보면 먼저 전날 종가의 30%에 해당하는 금액을 increment 가 바인딩하고 있고, 이 값을 전날 종가(price)와 더해서 상한가를 계산합니다. 최종으로 계산된 상한가는 upper_price 가 바인딩하게 됩니다.

수정된 함수가 제대로 동작하는지 확인하기 위해 다시 함수를 호출해 보겠습니다. 함수의 인자로 10,000 원을 입력하면 30% 오른 금액인 13,000 원이 나와야 함수가 제대로 구현된 것입니다. 다음 코드를 보면 함수의 인자로 10000 을 사용해 호출했지만 파이썬 IDLE 에서는 아무런 변화가 없이 명령어 프롬프트(>>>)가 다시 나타나고 있습니다. 함수의 구현 내용을 살펴보면 문제가 없어 보이는데 왜 값이 반환되지 않는 것일까요?

```
>>> def cal_upper(price):
        increment = price * 0.3
        upper_price = price + increment

>>> cal_upper(10000)
>>>
```

2-1) 함수 호출 과정 이해하기

앞서 구현한 코드가 제대로 동작하지 않는 이유를 이해하려면 파이썬이 함수를 호출하는 과정에서 메모리의 상태 변화를 알아야 합니다. 파이썬에서 def 키워드를 통해 함수를 정의하는 것은 말 그대로 함수를 정의하는 것이며, 함수가 정의됐다고 해서 함수가 실행되는 것은 아닙니다. 함수가 실행되려면 함수를 호출해야 합니다. 그렇다면 함수가 호출될 때 메모리에서는 어떤 일이 벌어질까요?

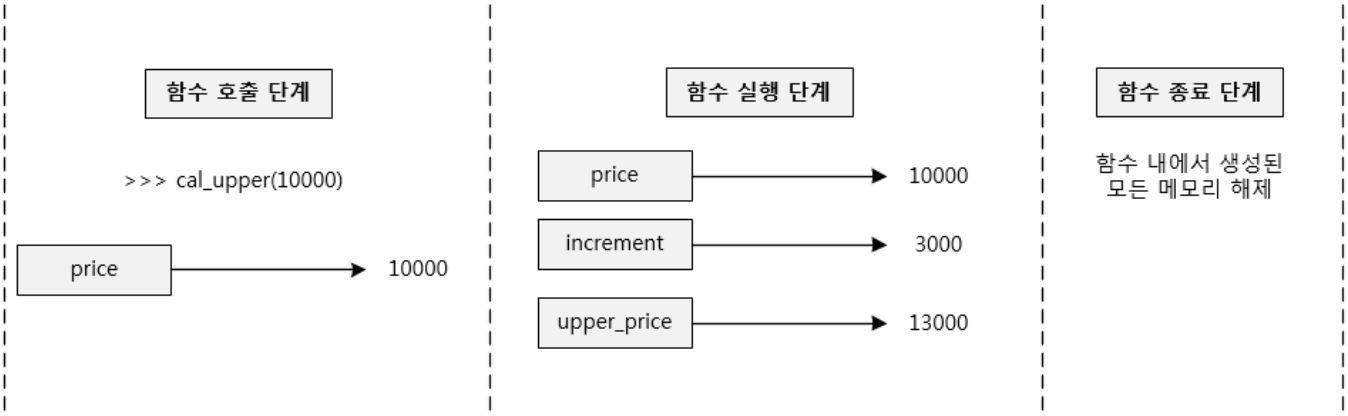

그림 5.3 함수 호출의 3 단계

그림 5.3 은 cal_upper 함수가 호출됐을 때의 함수 호출 3 단계를 나타냅니다. 파이썬에서 함수를 호출하면 함수의 인자에 위치하는 변수(price)가 함수 호출 시 입력된 인자 값(10000)을 바인딩하게 됩니다. 함수 호출 시 입력된 인자 값(10000)은 컴퓨터 메모리의 어딘가에 할당됩니다. 여기까지가 함수 호출 단계의 메모리 상태입니다. 중요한 개념이니 노트에 그림을 그려가면서 이해하시기 바랍니다.

함수 호출의 두 번째 단계는 함수 실행 단계입니다. 이 단계에서는 함수의 수행할 문장이 실행됩니다. cal_upper 함수에서 수행할 문장은 다음과 같습니다.

```
increment = price * 0.3
```

```
upper_price = price + increment
```

먼저 첫 번째 문장이 실행되고 이에 따라 price * 0.3 의 결괏값이 메모리의 어딘가에 할당됩니다. increment 변수는 이 할당된 값을 바인딩합니다. 여기서 중요한 것은 price * 0.3 이라는 연산의 결괏값이 메모리의 어딘가에 실제로 위치한다는 점입니다. 그다음 위의 코드에서 두 번째 문장이 실행되고 이에 따라 price + increment 의 결괏값이 메모리의 어딘가에 할당되고, 그 값을 upper_price 변수가 바인딩하게 됩니다. 이를 그림으로 표현하면 그림 5.3 의 중간 부분에 해당합니다. 코드가 여기까지 수행됐을 때 price, increment, upper_price 변수가 바인딩하는 값이 실제로 메모리에 저장되어 있다는 개념을 이해하는 것이 중요합니다.

함수 호출의 세 번째 단계는 함수 종료 단계입니다. 함수가 호출되면 함수 호출 단계 및 함수 실행 단계에서 중간값들이 메모리에 할당된다는 사실을 배웠습니다. 이처럼 함수가 호출됨에 따라 메모리가 사용되기 때문에 함수를 계속해서 호출하면 언젠가는 시스템의 메모리가 고갈될 것입니다. 가끔 스마트폰이나 컴퓨터에서 메모리가 부족해서 시스템이 중지되거나 느려지는 경험을 한 적이 있을 것입니다. 이것은 모두 프로그램이 실행되면서 메모리를 사용하기 때문에 발생하는 문제입니다.

파이썬에서는 위와 같은 메모리 관리 문제를 해결하기 위해 할당된 메모리를 자동으로 관리하는 메커니즘을 사용합니다. 한 예로 함수 호출 과정에서 할당된 메모리는 함수 호출이 끝나 더는 사용되지 않을 때 자동으로 메모리에서 삭제됩니다. 따라서 그림 5.3 의 함수 종료 단계처럼 앞의 두 단계에서 할당된 메모리는 함수 호출이 끝나면(함수를 호출한 쪽으로 실행 흐름이 되돌아가면) 더는 사용되지 않기 때문에 모두 삭제됩니다. 이처럼 함수가 종료될 때 함수 내부에서 계산된 값이 메모리에서 함께 삭제됐기 때문에 앞서 구현했던 cal_upper 함수가 제대로 동작하지 않았던 것입니다.

그림 5.4 는 함수 호출의 3 단계를 코드의 흐름 측면에서 보여주는 또 다른 그림입니다. 파이썬에서 함수가 호출되면 함수가 정의된 곳으로 가서 메모리를 할당하면서 코드가 실행되고 함수 내부의 수행할 문장에 대한 실행이 완료되면 그림 5.4 의 3 번처럼 다시 함수가 호출된 곳으로 되돌아갑니다. 그림 5.4 의 3 번 과정에서는 이미 함수의 호출이 종료됐기 때문에 함수 내에서 사용된 모든 메모리는 해제됐고 따라서 함수 내부의 변수를 참조할 수도 없습니다.

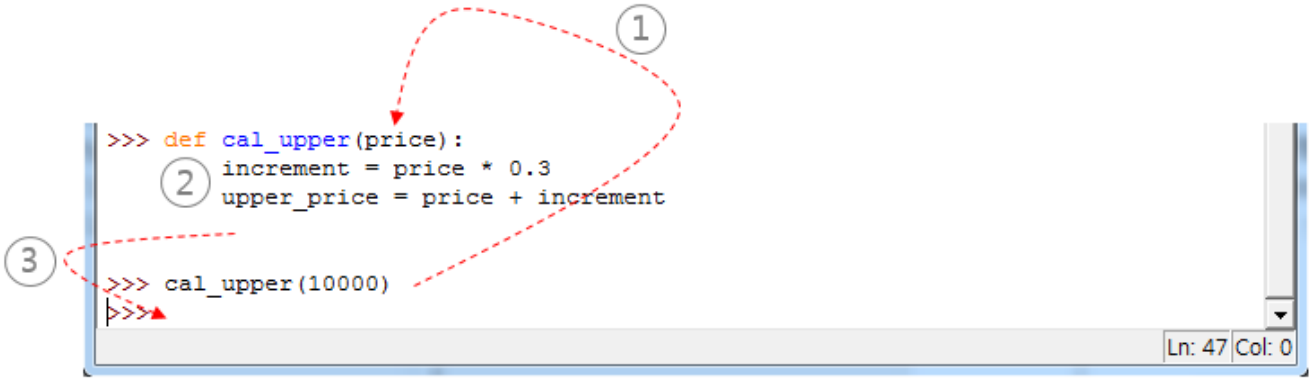

그림 5.4 함수 호출의 3 단계(2)

그림 5.5 는 함수를 호출한 후 함수 내에서 사용했던 변수를 참조해 본 것입니다. 그림 5.5 를 보면 IDLE 에서 함수 호출이 종료된 후 price, increment, upper_price 와 같이 함수 내부에서 사용되던 변수를 참조하면 모두 "정의되지 않았다(not defined)"라는 오류 메시지가 출력되는 것을 확인할 수 있습니다.

```
>>> def cal_upper(price):
        increment = price * 0.3
        upper_price = price + increment


>>> cal_upper(10000)
>>> price
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    price
NameError: name 'price' is not defined
>>> increment
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    increment
NameError: name 'increment' is not defined
>>> upper_price
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    upper_price
NameError: name 'upper_price' is not defined
>>>
```

Ln: 62 Col: 4

그림 5.5 변수 참조 오류의 예

지금까지 설명한 파이썬 함수 호출의 원리를 잘 이해했다면 cal_upper 함수를 호출했을 때 왜 어떤 값도 사용할 수 없었는지 이해하셨을 겁니다.

그렇다면 함수 내부에서 계산된 최종 결괏값을 어떻게 함수의 외부로 전달할 수 있을까요? 파이썬은 함수 내부에서 계산된 값을 함수 외부로(함수를 호출한 곳으로) 보내기 위해 return 이라는 키워드를 사용합니다. return 이라는 영어 단어에 '돌려주다'라는 뜻이 있는 것을 생각하면 함수 내에서 계산된 값을 함수 외부로 보내기 위해 return 이라는 키워드를 사용하는 것이 적절해 보입니다.

다음 코드는 수정된 cal_upper 함수 및 함수의 호출 결과입니다. 수정되기 전의 코드와 비교하면 함수의 끝 부분에 함수 내에서 계산된 값인 upper_price 를 반환하는 코드가 있음을 알 수 있습니다. 이처럼 함수에서 계산된 값은 항상 반환해야(돌려줘야) 함수를 호출한 부분에서 그 값을 사용할 수 있습니다. 수정된 코드에서는 함수가 호출된 다음 파이썬 프롬프트에 13000.0 이 나타나는 것을 확인할 수 있습니다.

```
>>> def cal_upper(price):
        increment = price * 0.3
        upper_price = price + increment
        return upper_price

>>> cal_upper(10000)
13000.0
>>>
```

다음 코드는 수정 된 cal_upper 함수를 이용해 전일 종가가 1,000 원일 때의 상한가와 5,000 원 일 때의 상한가를 계산해 본 것입니다. cal_upper 함수는 함수를 호출한 쪽으로 계산된 상한가를 반환하기 때문에 그 값을 바인딩하기 위해 upper_price 라는 변수를 사용했습니다.

```
>>> upper_price = cal_upper(1000)
>>> print(upper_price)
1300.0
>>> upper_price = cal_upper(5000)
>>> print(upper_price)
```

```
6500.0
>>>
```

함수 호출 과정에서 발생하는 메모리 해제가 잘 이해되지 않는 분들은 다음 이야기들을 읽어보기 바랍니다. "어떤 노인이 평생 모은 돈을 통장(함수의 결괏값에 해당)에 넣어뒀고 통장은 집 어딘가에 숨겨뒀습니다. 노인이 그 돈을 자식에 물려주고자 한다면 자신이 죽기 전에 통장 위치를 자식에게 알려줘야 합니다(값을 반환하는 것에 해당). 만약 노인이 자식에게 통장의 위치를 알려주지 않고 죽어버리면(값을 반환하지 않고 함수를 종료) 자식들은 그 통장에 있는 돈을 영영 찾을 수 없게 됩니다."

어떤가요? 좀 이해되셨나요?

저자는 함수의 호출 과정을 '방'을 이용해 다음과 같이 설명하기도 합니다.

"집에 거실과 안방이 있는데 파이썬 코드는 기본적으로 거실에서 실행됩니다. 이 상태에서 함수가 호출되면 안방 문을 열고 들어갑니다. 이때 함수 인자가 거실에서 안방으로 전달됩니다. 안방 문이 닫히고 안방 내에서 어떤 연산이 수행됩니다.

함수 내의 모든 코드가 실행되고 나면 다시 안방 문이 열리고 결괏값을 거실로 넘겨줍니다(함수 반환). 함수 호출이 완료되면 안방 문이 닫히기 때문에 거실에서는 더는 안방에 어떤 값이 있었는지 확인할 수 없습니다. 오직 함수가 호출될 때, 그리고 함수가 종료될 때만 인자 및 반환값의 전달을 통해서만 값이 전달될 수 있습니다. 함수가 종료된 후 안방 문이 닫히면 안방 내에서 생성된 값들은 모두 정리됩니다(메모리 해제)."

여러분이 매수한 종목이 항상 상한가만 되고 하한가가 되는 일이 없으면 좋겠습니다. 그러나 주식을 하다 보면 가끔 매수한 종목이 하한가로 가는 때가 있습니다. 전날 종가에서 하한가가 됐을 때 얼마가 되는지를 알아보기 위해 이번에는 하한가를 계산하는 cal_lower 라는 함수를 작성해 보겠습니다.

```
>>> def cal_lower(price):
        decrement = price * 0.3
        lower_price = price - decrement
        return lower_price
```

```
>>> cal_lower(1000)
700.0
>>> cal_lower(5000)
3500.0
```

```
>>>
```

위 코드를 보면 먼저 함수의 이름이 cal_lower 로 바뀌었고 함수 내부에서 사용하는 변수가 increment 에서 decrement 로 변경된 것을 알 수 있습니다. 그리고 하한가는 전날 종가에서 30%만큼 하락해야 하므로 price - decrement 로 작성한 것을 확인할 수 있습니다. 작성된 cal_lower 함수를 통해 전날 종가가 1,000 원일 때와 5,000 원일 때의 하한가를 계산해보면 정상적으로 값이 계산되는 것을 확인할 수 있습니다.

2-2) 두 개의 값 반환하기

파이썬을 배우기 전에 C/C++ 같은 프로그래밍 언어를 배운 분들은 함수의 반환값이 오직 하나라고 알고 계실 겁니다. 예를 들어, 상한가와 하한가를 동시에 구하는 함수를 작성하더라도 결괏값을 반환할 때는 두 값을 동시에 반환할 수 없습니다. 그럼 파이썬은 어떨까요? 파이썬도 기본적으로는 함수의 반환값으로 하나의 객체만 반환할 수 있습니다. 다만 파이썬에서는 반환할 여러 개의 데이터를 앞에서 배운 튜플이라는 자료구조에 넣은 후 반환함으로써 동시에 여러 개의 값을 반환할 수 있습니다.

앞에서 상한가와 하한가를 구하는 함수를 각각 작성해봤습니다. 이번에는 하나의 함수에서 전일 종가를 입력받은 후 상한가와 하한가를 한 번에 계산하는 함수인 cal_upper_lower 함수를 작성해보겠습니다.

```
>>> def cal_upper_lower(price):
        offset = price * 0.3
        upper = price + offset
        lower = price - offset
        return (upper, lower)
>>>
```

위 코드를 보면 상한가와 하한가를 구하는 부분은 기존의 함수와 같습니다. 다만 함수에서 결괏값을 반환할 때 '('와 ')' 기호를 사용해 상한가와 하한가를 하나의 튜플로 구성한 후 반환하는 것을 볼 수 있습니다. 어떤가요? 생각보다 간단하죠?

함수를 만들었으니 함수가 정상적으로 동작하는지 테스트해 보겠습니다. 함수를 호출하는 부분에 함수에서 반환되는 두 개의 값을 받기 위해 (upper, lower)라는 튜플을 사용한 것을 볼 수 있습니다. 10,000 원에 대한 상한가와 하한가를 계산해보니 상한가는 13,000 원으로, 하한가는 7,000 으로 잘 계산된 것을 확인할 수 있습니다.

```
>>> (upper, lower) = cal_upper_lower(10000)
>>> upper
13000.0
>>> lower
7000.0
```

```
>>>
```

3) 모듈

이번 절에서는 파이썬 모듈(module)에 대해 배우겠습니다. module 이라는 영어 단어를 사전에서 찾아보면 여러 뜻이 있는데, 그중 컴퓨터 분야에서는 '프로그램이나 하드웨어 기능의 단위'라는 의미로 사용되는 것을 확인할 수 있습니다. 파이썬의 모듈도 '프로그램의 기능 단위'를 의미하는데 다만 파이썬은 파일 단위로 작성된 파이썬 코드를 모듈이라고 부릅니다.

예를 들어, 그림 5.6 의 stock.py 라는 파일에는 두 개의 함수(cal_upper_lower, cal_macd)와 하나의 리스트(mylist)가 있는 것을 볼 수 있습니다. 여기서 함수 및 리스트를 포함하는 파일의 이름이 stock 이므로 파이썬에서는 이 파일을 stock 모듈이라고 부릅니다. 참고로 마이크로소프트의 워드 파일에 .docx 와 같은 확장자가 붙는 것처럼 보통 파이썬 코드는 .py 라는 확장자를 사용합니다.

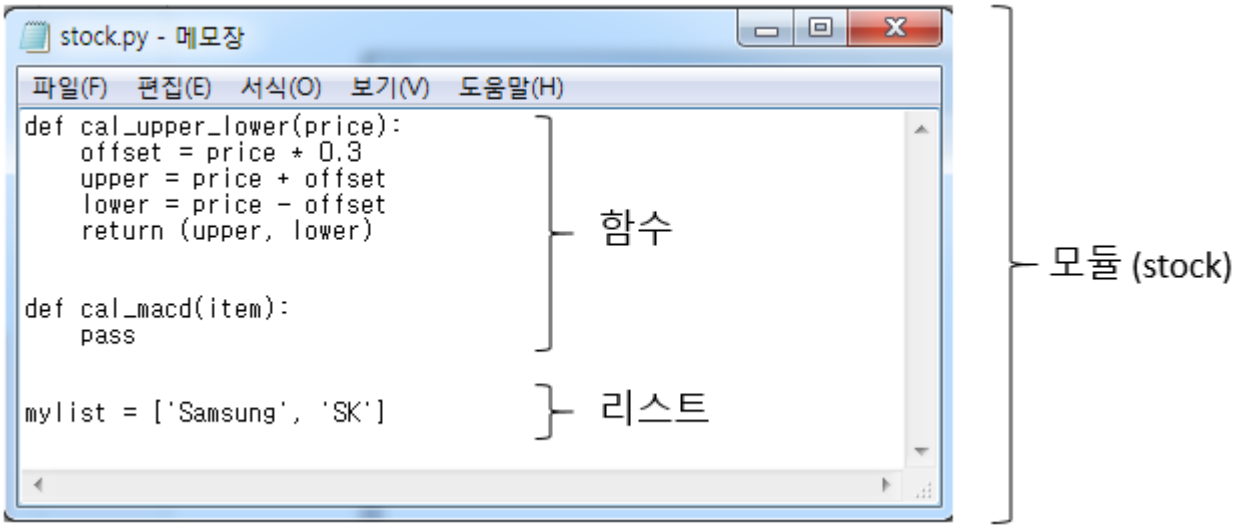

그림 5.6 파이썬 모듈의 구성

그렇다면 파이썬에서는 왜 모듈을 사용할까요? 앞서 함수를 사용하는 이유가 코드의 재사용을 위해서라고 설명해 드렸습니다. 모듈은 함수보다 상위 개념으로 함수와 마찬가지로 코드의 재사용을 위해 사용합니다.

예를 들어, 시스템 트레이딩과 관련된 프로그램을 작성할 경우 다른 사람이 작성한 함수를 재사용하고 싶다면 어떻게 해야 할까요? 먼저 함수의 코드를 복사한 후 여러분의 프로그램에 붙여넣은 다음 함수를 호출해야 합니다. 다른 사람이 작성한 함수를 사용하기 위해 함수 코드 전체를 복사한 후 붙여넣는 작업은 여간 번거로운 작업이 아닙니다. 또한, 시간이 흐른 후에 해당 함수가 업데이트되는 경우를 생각하면 함수가 업데이트될 때마다 코드를 새로 복사한 후 붙여넣기 해줘야 합니다.

파이썬의 모듈을 사용하는 경우에는 함수가 정의돼 있는 파일 자체를 복사한 후 모듈을 임포트(import)하기만 하면 해당 파일(모듈)에 구현된 모든 함수 및 자료구조를 사용할 수 있습니다. 즉, 모듈을 사용하면 함수를 사용하기 위해 함수 코드를 여러분의 코드에 복사 후 붙여넣는 작업을 하지 않아도 됩니다.

3-1) 모듈 만들기

파이썬에는 다양한 기능을 수행하는 모듈이 기본적으로 제공됩니다(파이썬을 설치할 때 함께 설치됩니다). 파이썬으로 프로그래밍하다 보면 많은 부분을 파이썬의 모듈을 통해 구현합니다. 파이썬 모듈이란 자주 사용되는 기능에 대해 이미 코드로 구현해둔 것이기 때문에 파이썬 모듈을 이용하면 직접 특정 기능을 구현하는 것보다 쉽고 빠르게 프로그램을 개발할 수 있습니다.

파이썬의 기본 모듈은 프로그래밍을 잘하는 개발자들이 이미 개발해 둔 것인데, 이처럼 이미 개발된 모듈을 사용할 수도 있지만 직접 모듈을 만들 수도 있습니다. 이번 절에서는 모듈을 직접 만들어보면서 모듈의 개념을 더욱 정확히 이해해 보겠습니다.

앞에서 상한가와 하한가를 구하는 함수를 파이썬 IDLE 에서 구현해 봤습니다. 문제는 파이썬 IDLE 를 닫으면 더는 해당 함수를 사용할 수 없게 된다는 점입니다. 여러분이 구현한 상한가와 하한가를 계산하는 함수를 파일로 작성해 이를 모듈로 만들어 봅시다. 이렇게 모듈로 만들어 두면 두고두고 사용할 수 있습니다. 그림 5.7 과 같이 파이썬 IDLE 에서 File 메뉴를 선택한 후 New File 을 선택합니다.

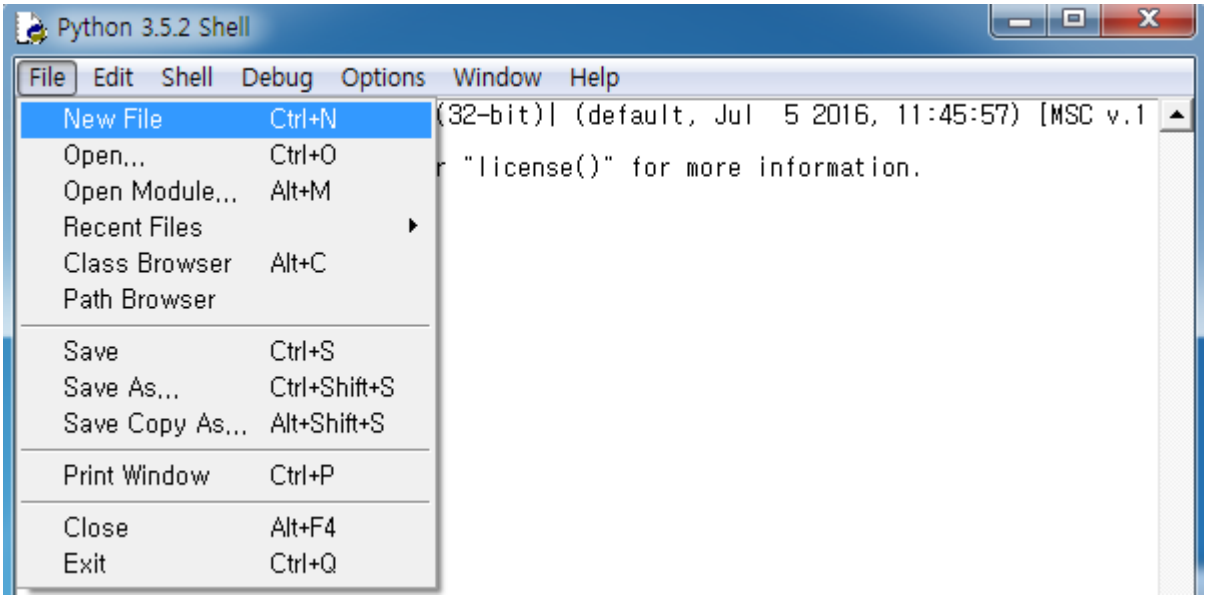

그림 5.7 파이썬 IDLE 에서 새 파일 열기

새 창에 다음과 같이 코드를 작성합니다. 앞에서 구현한 상한가와 하한가를 구하는 함수를 그대로 구현하고 추가로 author 라는 변수가 pystock 이라는 문자열을 바인딩하게 했습니다. 한가지 주의할 점은 다음 코드는 말 그대로 파이썬 코드이므로 >>>와 같은 파이썬 프롬프트가 없다는 점입니다.

```
def cal_upper(price):
    increment = price * 0.3
    upper_price = price + increment
    return upper_price
```

```
def cal_lower(price):
    decrement = price * 0.3
    lower_price = price - decrement
    return lower_price

author = "pystock"
```

코드를 작성하고 나면 그림 5.8 과 같이 File 메뉴에서 Save 메뉴를 클릭해 파일로 저장합니다.

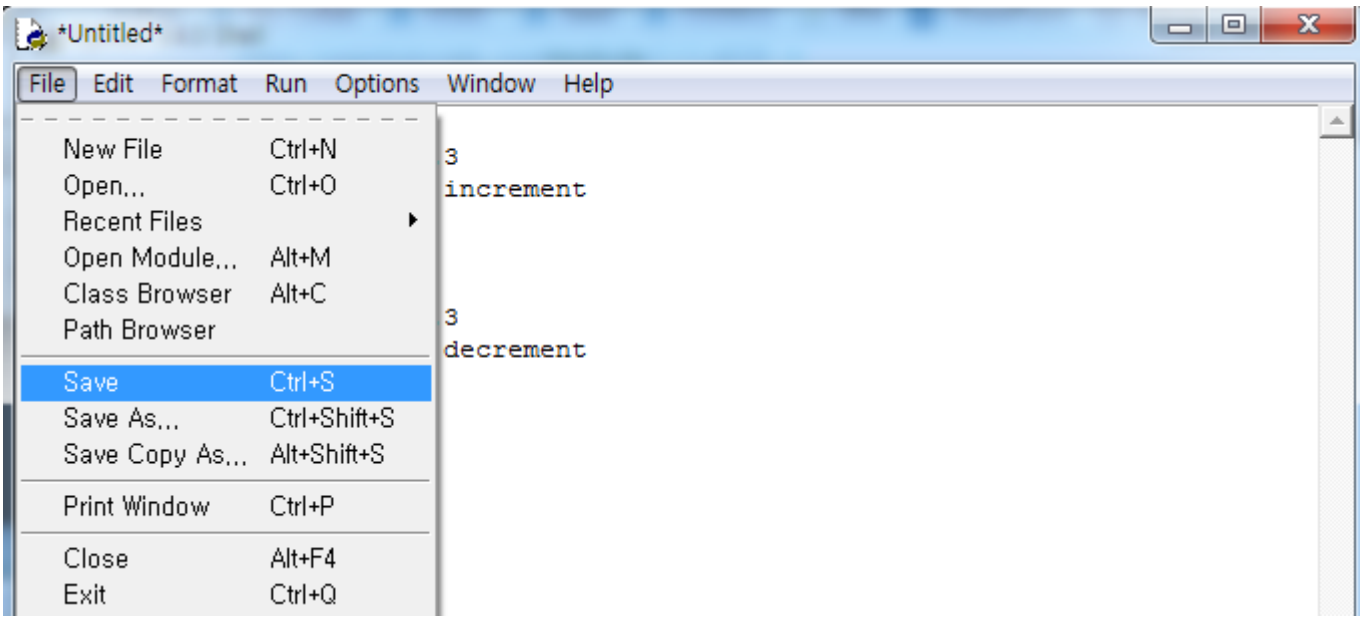

그림 5.8 파이썬 코드 저장

파이썬 코드를 파일로 저장할 때 그림 5.9 와 같이 저장될 파일의 경로를 C:/Anaconda3 디렉터리로 선택한 후 파일명을 stock 으로 변경하고 저장 버튼을 클릭합니다. 따라서 방금 작성한 코드의 저장 경로는 C:/Anaconda3/stock.py 가 됩니다.

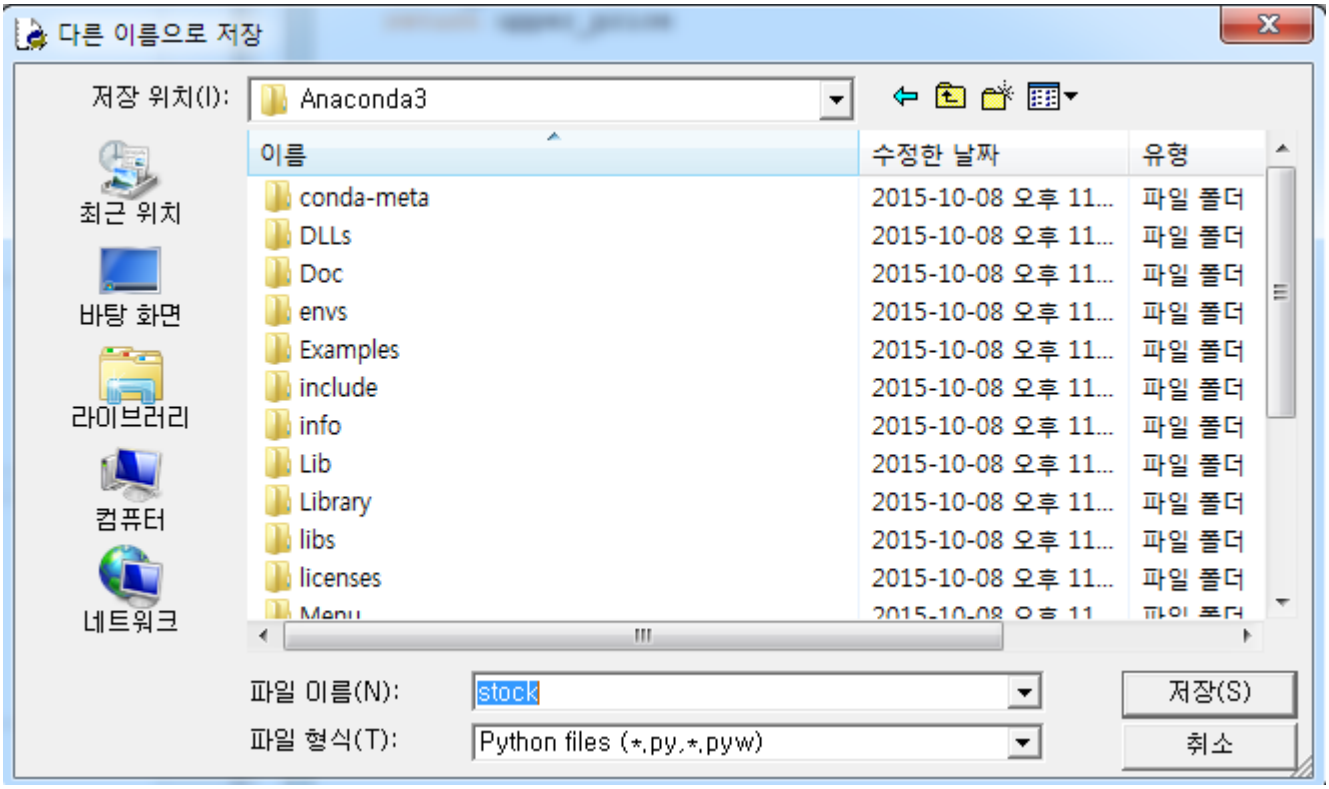

그림 5.9 파이썬 코드 저장(2)

지금까지 작업한 내용을 요약해보면 함수를 구현하고 이를 stock.py 라는 이름으로 하드디스크에 저장했습니다. 이제 앞에서 작성한 stock 이라는 모듈을 사용해 보겠습니다. 앞서 설명한 것처럼 모듈을 사용하려면 모듈을 임포트해야 합니다.

그림 5.10 과 같이 파이썬 IDLE 에서 stock 모듈을 임포트해 봅시다. 여기서 오류가 발생하지 않으면 stock 모듈을 잘 만들었고 해당 모듈이 정상적으로 임포트된 것입니다.

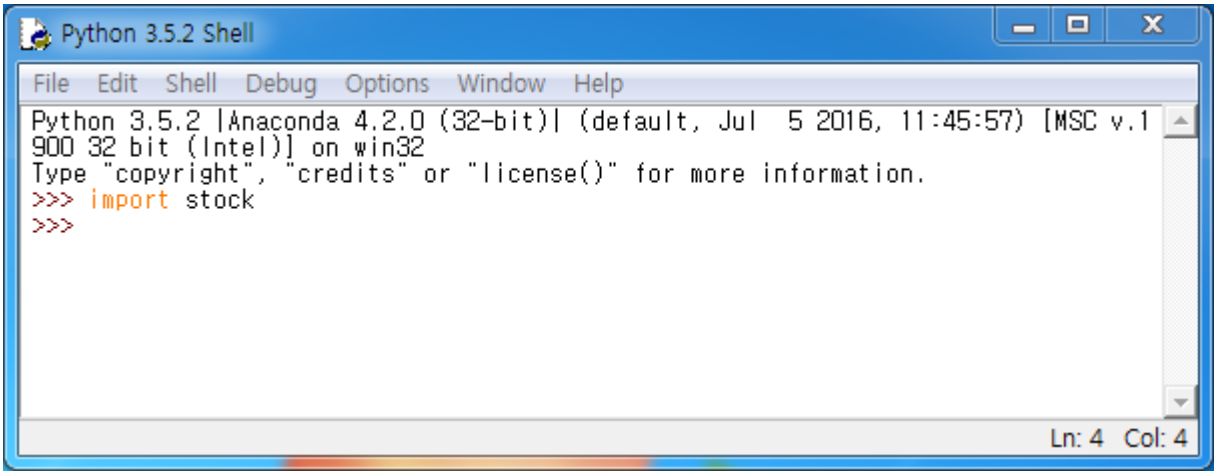

그림 5.10 stock 모듈 임포트

앞에서 구현한 stock 모듈에는 author 라는 변수와 cal_upper 와 cal_lower 라는 함수가 있었습니다. 파이썬에서는 모듈을 임포트하면 모듈 내에 구현된 함수나 변수를 사용할 수 있습니다. 먼저 다음과 같이 stock 모듈 내에 있던 변수 author 를 화면에 출력해봅시다. 이때 author 라는 변수명을 바로 적으면 안 되고 stock.author 라고 적어야 합니다. 모듈 내의 함수나 변수명에 접근할 때는 항상 '모듈명.함수명(변수명)'과 같은 형태로 적어야 합니다.

```
>>> print(stock.author)
pystock
>>>
```

이번에는 stock 모듈 내에 구현된 함수를 호출해 보겠습니다. 함수 역시 먼저 모듈명을 적은 후 '.'를 붙이고 함수명을 적어야 합니다. 이렇게 하는 이유는 해당 함수가 stock 이라는 모듈 내에 정의돼 있다는 것을 명시적으로 알려주기 위해서입니다. stock.py 파일에 구현된 cal_upper 와 cal_lower 함수를 사용할 수 있음을 확인할 수 있습니다.

```
>>> stock.cal_upper(10000)
13000.0
>>> stock.cal_lower(10000)
7000.0
>>>
```

여러분이 앞으로 모듈을 개발하고 이를 배포하고자 한다면 모듈에 구현된 함수가 정상적으로 동작하는지 충분히 테스트한 후 모듈을 배포해야 합니다. 이를 위해서는 모듈 파일 내에서 함수를 직접 호출하는 테스트 코드를 구현하는 것이 좋습니다.

다음과 같이 stock.py 파일에 cal_upper 와 cal_lower 함수를 호출하는 코드를 추가해 줍니다.
그리고 `print(__name__)`도 추가해줍니다.

```
def cal_upper(price):
    increment = price * 0.3
    upper_price = price + increment
    return upper_price

def cal_lower(price):
    decrement = price * 0.3
    lower_price = price - decrement
```

```
    return lower_price

author = "pystock"

print(cal_upper(10000))
print(cal_lower(10000))
print(__name__)
```

수정 된 stock.py 파일을 실행하려면 그림 5.11 처럼 Run -> Run Module 메뉴를 차례로 선택합니다.

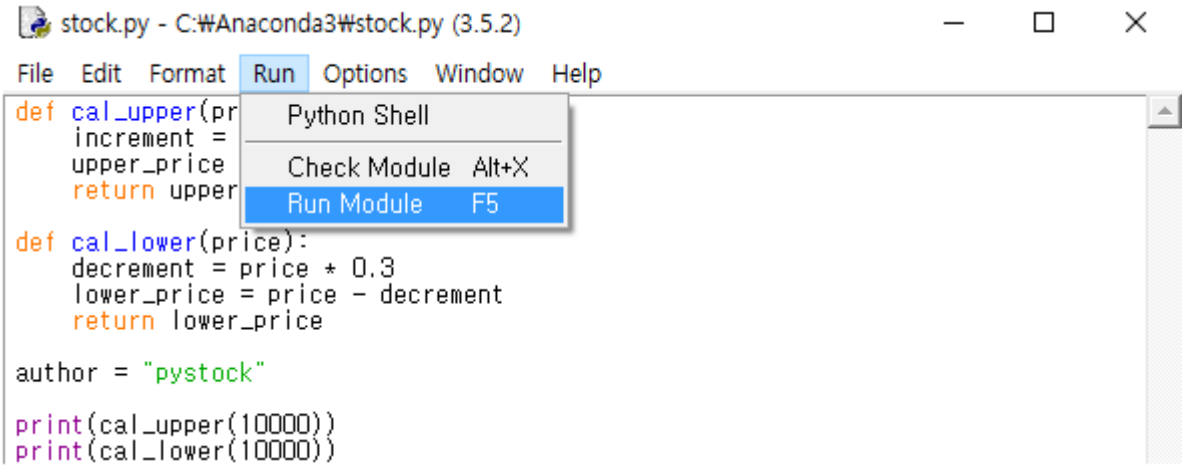

그림 5.11 모듈 코드 실행

파이썬 IDLE 에 다음과 같이 stock.py 파일의 실행 결과가 출력됩니다. 10,000 원에 대한 상한가와 하한가가 모두 정상적으로 출력됨을 확인할 수 있습니다. 그리고 `print(__name__)`의 결과로 `__main__`이라는 문자열이 화면에 출력됨을 확인할 수 있습니다.

```
>>>
========== RESTART: C:₩Anaconda3₩stock.py ==========
13000.0
7000.0
__main__
>>>
```

업데이트된 stock 모듈을 다시 임포트 해보겠습니다. stock 모듈을 임포트하면 앞서 추가했던 print 구문의 결괏값이 출력됨을 확인할 수 있습니다. 모듈을 테스트하는 입장에서는 모듈 내에 테스트 코드를 넣는 방식은 유용하지만, 모듈을 사용하는 입장에서는 임포트할 때 모듈을 테스트하는 코드가 같이 실행되는 것은 불필요한 기능입니다.

```
>>> import stock
13000.0
7000.0
stock
>>>
```

여기서 한가지 눈여겨볼 점은 모듈을 직접 실행했을 때는 `print(__name__)`의 결과로 `__main__`이라는 문자열이 출력되지만, 해당 모듈을 임포트했을 때는 모듈의 이름인 stock 이라는 문자열이 출력된다는 점입니다.
사실 파이썬에서 `__name__`이라는 변수는 파이썬 자체에서 사용하는 변수로 특정 파이썬 파일이 직접 실행된 것인지 또는 다른 파이썬 파일에서 임포트 된 것인지를 확인하는 용도로 사용됩니다. 특정 파이썬 파일이 독립적으로 실행됐다면 `__name__`이라는 변수는 `__main__`이라는 문자열을 바인딩하며 다른 파일에 임포트 된 경우에는 자신의 파일명을 바인딩합니다.
파이썬의 `__name__`이라는 변수를 사용하면 앞서 모듈의 테스트 코드가 모듈을 임포트했을 때 자동으로 호출되는 문제를 해결할 수 있습니다. stock.py 파일에 추가했던 테스트 코드를 다음과 같이 수정합니다.

```
if __name__ == "__main__":
    print(cal_upper(10000))
    print(cal_lower(10000))
    print(__name__)
```

추가된 코드의 의미를 살펴보면 `__name__`이라는 파이썬 변수가 바인딩하고 있는 값이 `__main__`이라는 문자열이라면 들여쓰기 된 세 개의 print 문을 실행합니다. 즉, stock.py 파일이 직접 실행이 된 경우라면 세 개의 print 문이 실행됩니다. 만약 다른 파일에 stock.py 파일(모듈)이 임포트 된다면 `__name__`은 모듈의 이름인 stock 을 바인딩하므로 if 문의 조건식을 만족하지 않고 따라서 세 개의 print 문도 실행되지 않습니다.

stock.py 파일을 수정했다면 stock 모듈을 다시 임포트해보기 바랍니다. stock.py 파일에 여전히 모듈을 테스트하는 코드가 존재하지만 해당 코드가 실행되지 않았음을 확인할 수 있습니다.

```
Python 3.5.2 |Anaconda 4.1.1 (32-bit)| (default, Jul  5 2016, 11:45:57) [MSC v.1900 32 bit
(Intel)] on win32
```

```
Type "copyright", "credits" or "license()" for more information.
>>> import stock
>>>
```

그림 5.11 과 같이 stock.py 파일을 직접 실행시키면 다음과 같이 여전히 테스트가 코드가 출력됨을 확인할 수 있습니다.

```
>>>
========== RESTART: C:₩Anaconda3₩stock.py ==========
13000.0
7000.0
__main__
>>>
```

3-2) 파이썬에서 시간 다루기

알고리즘 트레이딩을 하려면 파이썬에서 시간을 잘 다룰 수 있어야 합니다. 유식한 말로 주식 시장의 데이터를 시계열 데이터라고 하는데 여기서 시계열이란 '일정 시간 간격으로 배치된 데이터의 수열'을 말합니다. 그렇다면 파이썬에서 시간과 날짜를 다루려면 어떻게 해야 할까요?

바로 앞에서 파이썬의 모듈을 공부했다는 점을 생각해보면 파이썬에서 시간을 다루는 가장 쉬운 방법은 모듈을 사용하는 것임을 눈치채셨을 수도 있습니다. 파이썬에는 시간과 날짜를 다루기 위한 time 과 datetime 이라는 기본 모듈이 있습니다.

먼저 time 모듈을 사용해 보겠습니다. 다음과 같이 time 모듈을 사용하려면 먼저 time 모듈을 임포트해야 합니다. 이미 알고 계시겠지만 여기서 모듈의 이름이 time 인 이유는 파일명이 time.py 이기 때문입니다. time 모듈에서 현재 시각을 구하는 함수는 time()입니다. time()은 현재 시각을 반환하는 함수인데 1970 년 1 월 1 일 0 시 0 분 0 초를 기준으로 초 단위로 지난 시간을 알려줍니다.

```
>>> import time
>>> time.time()
1444532446.467043
>>> time.ctime()
'Sun Oct 11 12:00:50 2015'
>>> type(_)
<class 'str'>
>>>
```

위 코드를 보면 time.time()의 결과가 우리가 알기 어려운 실수 형태로 반환됐음을 확인할 수 있습니다. 사람들이 좀 더 쉽게 읽을 수 있는 시간을 구하려면 ctime() 함수를 사용하면 됩니다. ctime() 함수의 결과를 확인하면 '2015 년 10 월 11 일 12 시 00 분 50 초'임을 확인할 수 있습니다. 요일은 일요일입니다.

ctime() 함수의 반환값인 'Sun Oct 11 12:00:50 2015'은 문자열처럼 보이긴 하지만 정확한 타입을 확인하기 위해 type() 함수를 사용했습니다. 여기서 type() 함수의 인자로 _를 사용했는데 파이썬 IDLE 에서 _는 가장 최근의 반환값을 바인딩하고 있는 변수입니다. 위 코드에서 type(_) 코드가 나오기 전에 가장 최근의 값은 time.ctime() 함수의 반환값이므로 type(_) 의미는 time.ctime() 함수의 반환값에 대한 타입을 확인하는 것이 됩니다. 예상대로 <class 'str'>, 즉 문자열 타입임을 알 수 있습니다.

time.ctime()의 반환값에서 연도(year)만 구하려면 어떻게 해야 할까요? 앞서 time.ctime()의 반환값이 문자열이었으므로 문자열에서 제공하는 메서드를 사용하면 되겠죠?

다음 코드는 time.ctime() 함수의 반환값을 cur_time 이라는 변수가 바인딩하게 한 후 문자열 객체가 제공하는 split 이라는 메서드를 사용해 공백을 기준으로 문자열을 분리했습니다. 분리된 문자열 중 맨 마지막에 존재하는 원소를 화면에 출력하면 연도가 화면에 출력 됩니다.

```
>>> cur_time = time.ctime()
>>> print(cur_time.split(' ')[-1])
2015
>>>
```

이번에는 time 모듈의 sleep() 함수를 이용해 프로그램을 잠깐 잠재워 보겠습니다. time 모듈의 sleep() 함수는 인자로 전달되는 값에 해당하는 초(sec) 동안 코드의 실행을 멈추는 역할을 합니다. 예를 들어, 다음 코드는 0 부터 9 까지의 숫자를 출력하는데, 1 초 간격으로 숫자를 출력하는 코드입니다. 코드를 실행하면 파이썬 IDLE 에서 숫자가 1 초 간격으로 출력되는 것을 확인할 수 있습니다.

```
>>> for i in range(10):
        print(i)
        time.sleep(1)
```

앞에서 설명한 것처럼 파이썬에서 모듈은 파이썬 파일을 의미합니다. 그러나 모듈의 실행 속도가 중요한 일부 모듈은 파이썬으로 작성되지 않고 C 언어로 작성됩니다. 이 경우에는 해당 모듈의 코드를 직접 볼 수는 없습니다. 다음 코드는 time 모듈과 random 모듈을 임포트한 후 해당 모듈의 위치를 확인하는 코드입니다. time 모듈은 내장 모듈이기 때문에 해당 모듈의 위치가 따로 출력되지는 않지만 random 모듈은 해당 모듈이 위치하는 경로가 출력되는 것을 볼 수 있습니다.

```
>>> import time
>>> time
<module 'time' (built-in)>
>>> import random
>>> random
<module 'random' from 'C:₩₩Anaconda3₩₩lib₩₩random.py'>
>>>
```

파이썬 모듈에는 여러 기능을 수행하는 함수와 변수들이 있을 수 있는데 모듈 안에 어떤 함수나 변수가 있는지 어떻게 확인할 수 있을까요? 가장 간단한 방법은 다음과 같이 dir() 함수를 사용하는 것입니다. 임포트된 모듈에 대해 모듈명을 사용해 dir() 내장 함수를 호출하면 해당 모듈의 구성 요소를 확인할 수 있습니다.

```
>>> import time
>>> dir(time)
['_STRUCT_TM_ITEMS', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'altzone',
'asctime', 'clock', 'ctime', 'daylight', 'get_clock_info', 'gmtime', 'localtime', 'mktime',
'monotonic', 'perf_counter', 'process_time', 'sleep', 'strftime', 'strptime', 'struct_time', 'time',
'timezone', 'tzname']
>>>
```

3-3) OS 모듈

파이썬에는 기본적으로 제공되는 다양한 모듈이 있습니다. 이 모듈들은 모두 유용하게 사용되지만 가장 자주 사용되는 모듈 가운데 os 라는 모듈이 있습니다. os 모듈은 Operating System 의 약자로 운영체제에서 제공되는 여러 기능을 파이썬에서 수행할 수 있게 해줍니다.

예를 들어, 파이썬을 이용해 파일을 복사하거나 디렉터리를 생성하고 특정 디렉터리 내의 파일 목록을 구하고자 할 때 os 모듈을 사용하면 됩니다. os 모듈은 다양한 기능을 제공하는데 이 중 자주 사용되는 몇 가지만 살펴보겠습니다.

먼저 현재 경로를 구하려면 os 모듈의 getcwd() 함수를 사용하면 됩니다. 다음 코드를 보면 현재 경로로 C:₩Anaconda3₩Lib₩idlelib 이 반환되는 것을 확인할 수 있습니다. 이는 현재 파이썬 IDLE 가 해당 경로에서 실행됐기 때문입니다.

```
>>> import os
>>> os.getcwd()
'C:₩₩Anaconda3₩₩Lib₩₩idlelib'
>>>
```

앞으로 프로그래밍하다 보면 getcwd() 함수가 필요가 경우가 종종 있을 것입니다. 일단 getcwd() 함수로 현재 경로를 얻을 수 있다는 것만 기억하기 바랍니다.

이번에는 특정 경로에 존재하는 파일과 디렉터리 목록을 구하는 함수인 listdir() 함수를 사용해보겠습니다. 다음 코드를 보면 현재 경로인 C:₩Anaconda3₩Lib₩idlelib 에 존재하는 파일과 디렉터리 목록이 리스트로 구성된 후 반환되는 것을 확인할 수 있습니다.

```
>>> os.listdir()
['aboutDialog.py', 'AutoComplete.py', 'AutoCompleteWindow.py', 'AutoExpand.py',
'Bindings.py', 'CallTips.py', 'CallTipWindow.py', 'ChangeLog', 'ClassBrowser.py',
'CodeContext.py', 'ColorDelegator.py', 'config-extensions.def', 'config-highlight.def', 'config-
keys.def', 'config-main.def', 'configDialog.py', 'configHandler.py', 'configHelpSourceEdit.py',
'configSectionNameDialog.py', 'CREDITS.txt', 'Debugger.py', 'Delegator.py',
'dynOptionMenuWidget.py', 'EditorWindow.py', 'extend.txt', 'FileList.py', 'FormatParagraph.py',
```

```
'GrepDialog.py', 'help.txt', 'HISTORY.txt', 'HyperParser.py', 'Icons', 'idle.bat', 'idle.py', 'idle.pyw',
'IdleHistory.py', 'idlever.py', 'idle_test', 'IOBinding.py', 'keybindingDialog.py',
'macosxSupport.py', 'MultiCall.py', 'MultiStatusBar.py', 'NEWS.txt', 'ObjectBrowser.py',
'OutputWindow.py', 'ParenMatch.py', 'PathBrowser.py', 'Percolator.py', 'PyParse.py',
'PyShell.py', 'README.txt', 'RemoteDebugger.py', 'RemoteObjectBrowser.py',
'ReplaceDialog.py', 'rpc.py', 'RstripExtension.py', 'run.py', 'ScriptBinding.py', 'ScrolledList.py',
'SearchDialog.py', 'SearchDialogBase.py', 'SearchEngine.py', 'StackViewer.py',
'tabbedpages.py', 'textView.py', 'TODO.txt', 'ToolTip.py', 'TreeWidget.py', 'UndoDelegator.py',
'WidgetRedirector.py', 'WindowList.py', 'ZoomHeight.py', '__init__.py', '__main__.py',
'__pycache__']
>>>
```

listdir() 함수의 인자로 경로를 전달하는 경우 해당 경로에 존재하는 파일과 디렉터리 목록을 구할 수 있습니다. 다음 코드는 listdir() 함수의 인자로 'c:/Anaconda3'을 전달한 경우입니다. 이처럼 listdir() 함수의 인자로 특정 경로를 지정하는 경우 해당 경로에 있는 파일과 디렉터리 목록이 반환됩니다.

```
>>> os.listdir('c:/Anaconda3')
['conda-meta', 'DLLs', 'Doc', 'envs', 'Examples', 'include', 'info', 'Lib', 'Library', 'libs', 'licenses',
'LICENSE_PYTHON.txt', 'Menu', 'node-webkit', 'pkgs', 'python.exe', 'python34.dll',
'pythonw.exe', 'qt.conf', 'Scripts', 'share', 'stock.py', 'tcl', 'Tools', 'Uninstall-Anaconda.exe',
'xlwings32.dll', 'xlwings64.dll', '__pycache__']
>>>
```

윈도우의 파일 탐색기를 이용해 'c:/Anaconda3' 경로로 이동한 후 파일과 디렉터리 목록을 확인하고 위 코드에서 listdir() 함수 반환값과 한 번 비교해 보기 바랍니다.

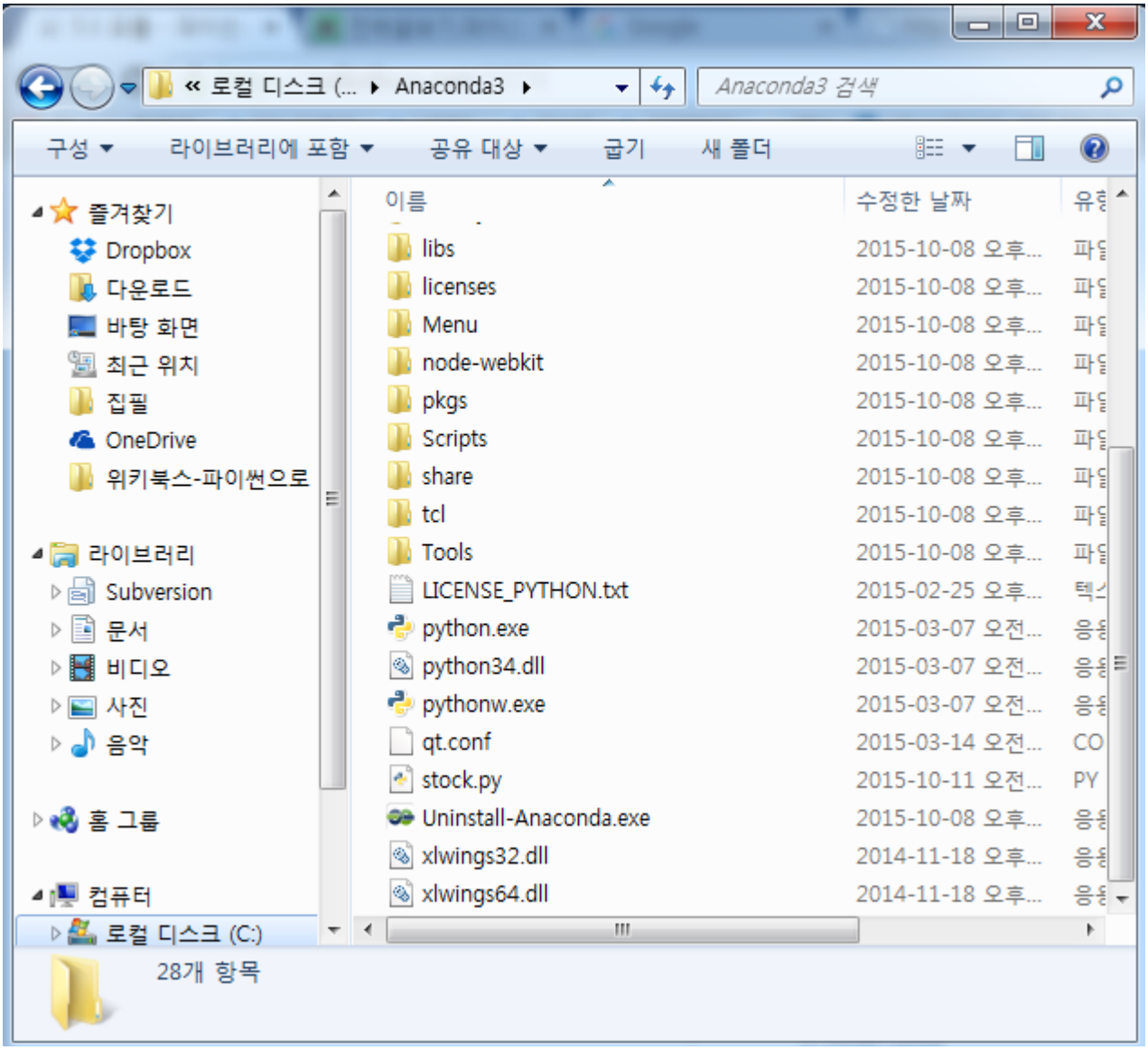

그림 5.12 파일 및 디렉터리 확인

해당 경로에 총 몇 개의 파일 또는 디렉터리가 존재하는지 확인해 보겠습니다. listdir() 함수의 반환값은 파이썬 리스트인데, 리스트, 튜플, 문자열 등에서 원소의 개수를 세기 위해서는 len() 함수를 사용하면 된다고 앞에서 이야기했습니다. 다음은 listdir() 함수의 반환값에 대해 len() 함수를 사용한 예입니다. 해당 경로에는 28 개의 파일 또는 디렉터리가 존재하는 것을 확인할 수 있습니다. 참고로 type() 함수를 이용해 listdir() 함수의 반환값 타입을 확인해보면 리스트임을 확인할 수 있습니다.

```
>>> files = os.listdir('c:/Anaconda3')
>>> len(files)
28
>>> type(files)
<class 'list'>
```

```
>>>
```

이번에는 'c:/Anaconda3'이라는 경로에 있는 파일 중 확장자가 'exe'로 끝나는 파일만 출력하는 코드를 작성해 보겠습니다. 이를 위해서는 listdir() 함수와 if 문, 문자열의 endswith() 메서드를 조합하면 됩니다. 다음을 보면 'c:/Anaconda3' 경로에 있는 파일 목록 중 문자열이 'exe'로 끝나는 경우를 출력하는 것을 확인할 수 있습니다.

```
>>> for x in os.listdir('c:/Anaconda3'):
        if x.endswith('exe'):
            print(x)

python.exe
pythonw.exe
Uninstall-Anaconda.exe
>>>
```

3-4) 모듈을 임포트하는 세가지 방법

지금까지 모듈을 임포트할 때 'import 모듈명'이라는 규칙을 사용했습니다. 그런데 다른 사람이 작성한 파이썬 코드를 보다 보면 다양한 형태로 모듈을 임포트하는 코드를 볼 수 있습니다. 이번 절에서는 모듈을 임포트하는 방식 간의 차이점을 살펴보겠습니다.

먼저 파이썬 IDLE 를 새롭게 실행한 후 dir() 함수를 파이썬 IDLE 에서 호출해 보겠습니다.

```
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__']
>>>
```

이번에는 os 모듈을 임포트한 후 다시 dir() 함수를 호출해 이전 dir() 함수의 결괏값과 비교해 봅니다. 차이점이 보이시나요? os 모듈을 임포트한 후에는 os 라는 이름이 dir() 함수의 결괏값 리스트에 추가된 것을 확인할 수 있습니다. 이처럼 os 라는 이름이 추가됐기 때문에 os 라는 이름을 사용하는 os.listdir()과 같은 표현이 가능했던 것입니다.

```
>>> import os
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'os']
>>>
```

이번에는 모듈을 임포트하는 두 번째 방법을 살펴보겠습니다. 열려 있던 파이썬 IDLE 를 닫고 새로 파이썬 IDLE 를 실행한 후 다음과 같이 실행해보기 바랍니다. 'from os import listdir' 구문의 의미는 "os 모듈로부터(from) listdir 을 임포트하라"입니다. 두 번째 방식으로 모듈을 임포트한 경우 dir()의 결괏값 리스트에는 os 가 아니라 os 모듈에 구현된 listdir 함수만 존재하는 것을 확인할 수 있습니다.

```
>>> from os import listdir
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'listdir']
>>>
```

따라서 다음과 같이 os 라는 모듈명을 사용하거나 os.listdir()과 같이 os 모듈 내의 listdir() 함수를 호출할 경우 오류가 발생합니다. listdir()과 같이 해당 함수를 직접 이용하는 방법만 가능합니다.

```
>>> os
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    os
NameError: name 'os' is not defined
>>> os.listdir()
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    os.listdir()
NameError: name 'os' is not defined
>>> listdir()
```

모듈 내의 함수를 호출할 때 모듈을 임포트하는 첫 번째 방식에서는 '모듈명.함수명' 형태로 사용했는데, 두 번째 방식은 함수 이름만으로도 바로 함수 호출이 가능하므로 프로그래밍할 코드의 수가 적어진다는 장점이 있습니다. 다만 두 번째 방식은 기존에 선언된 변수나 함수와 이름이 충돌할 가능성이 있습니다.

예를 들어, "김철수"라는 이름이 "종로"에도 있고 "종각"에도 있을 때 "종로.김철수", "종각.김철수"와 같은 식으로 표현하면 서로 구분할 수 있습니다. 그런데 지역을 빼고 "김철수"라고 하면 부를 때는 좋지만 누가 누군지 모르는 경우가 발생합니다. listdir 이라는 이름을 여러분이 만든 함수나 변수에서 이미 사용 중이라면 'from os import listdir'을 하는 순간 해당 이름의 변수나 함수가 없어지고 os 모듈의 listdir 로 대체되는데, 이러한 것들이 나중에는 잠재적인 프로그램의 버그가 될 수 있습니다.

모듈을 임포트하는 세 번째 방식은 모듈 내의 모든 것을 임포트하는 것입니다. 이전과 마찬가지로 IDLE 를 새로 실행한 후 다음 코드를 실행해 봅시다. 'from os import *'은 "os 모듈 내의 모든 것을 임포트하라"라는 의미입니다. 따라서 dir() 함수의 결괏값을 보면 os 모듈이 지원하는 함수나 변수가 모두 리스트에 포함된 것을 확인할 수 있습니다.

```
>>> from os import *
>>> dir()
```

['F_OK', 'O_APP**END**', 'O_BINARY', 'O_CREAT', 'O_EXCL', 'O_NOINHERIT', 'O_RANDOM',
'O_RDONLY', 'O_RDWR', 'O_SEQUENTIAL', 'O_SHORT_LIVED', 'O_TEMPORARY', 'O_TEXT',
'O_TRUNC', 'O_WRONLY', 'P_DETACH', 'P_NOWAIT', 'P_NOWAITO', 'P_OVERLAY', 'P_WAIT',
'R_OK', 'SEEK_CUR', 'SEEK_END', 'SEEK_SET', 'TMP_MAX', 'W_OK', 'X_OK', '__builtins__',
'__doc__', '__loader__', '__name__', '__package__', '__spec__', '_exit', 'abort', 'access', 'altsep',
'chdir', 'chmod', '**close**', 'closerange', 'cpu_count', 'curdir', 'defpath', 'device_encoding',
'devnull', 'dup', 'dup2', 'environ', 'error', 'execl', 'execle', 'execlp', 'execlpe', 'execv', 'execve',
'execvp', 'execvpe', 'extsep', 'fdopen', 'fsdecode', 'fsencode', 'fstat', 'fsync', 'get_exec_path',
'get_handle_inheritable', 'get_inheritable', 'get_terminal_size', 'getcwd', 'getcwdb', 'getenv',
'getlogin', 'getpid', 'getppid', 'isatty', 'kill', 'linesep', 'link', 'listdir', 'lseek', 'lstat', 'makedirs',
'mkdir', 'name', '**open**', 'pardir', 'path', 'pathsep', 'pipe', 'popen', 'putenv', '**read**', 'readlink',
'remove', 'removedirs', 'rename', 'renames', '**replace**', 'rmdir', 'sep', 'set_handle_inheritable',
'set_inheritable', 'spawnl', 'spawnle', 'spawnv', 'spawnve', 'startfile', 'stat', 'stat_float_times',
'stat_result', 'statvfs_result', 'strerror', 'supports_bytes_environ', 'symlink', 'system',
'terminal_size', 'times', 'times_result', 'umask', 'uname_result', 'unlink', 'urandom', 'utime',
'waitpid', 'walk', '**write**']
>>>

세 번째 방식으로 os 모듈을 임포트했을 때 listdir()을 사용하려면 두 번째 방식과 동일하게 listdir()을 바로 사용하면 됩니다. 참고로 세 번째 방식도 두 번째 방식과 마찬가지로 기존에 listdir() 이라는 이름의 함수가 이미 있는 경우 이름 충돌과 관련된 문제가 발생할 수 있습니다.

그렇다면 모듈을 임포트하는 가장 좋은 방법 또는 일반적으로 추천되는 스타일은 무엇인지 궁금하실 겁니다. 이에 대한 대답은 바로 첫 번째 방식입니다. 모듈을 임포트할 때는 'import 모듈명'을 사용하고, 모듈 내의 함수 또는 변수를 사용할 때는 '모듈명.함수명(변수)'의 형태를 사용하는 것입니다.

그러나 이따금 모듈명이 너무 긴 경우가 있습니다. 또는 모듈명이 너무 짧아서 모듈의 이름만으로는 정확히 구분하기가 어려워 모듈명을 다르게 사용하고 싶을 때도 있을 것입니다. 이 경우 다음과 같이 모듈을 임포트할 때 이름을 변경할 수도 있습니다.

'import os as winos'의 의미는 "os 모듈을 winos 로 임포트하라"입니다. dir() 함수의 결괏값을 확인해보면 os 대신 winos 항목이 결괏값 리스트에 있는 것을 확인할 수 있습니다. 따라서 os 모듈 내에 있던 함수를 사용할 때도 os.getcwd()가 아니라 winos.getcwd()와 같이 사용해야 합니다.

```
>>> import os as winos
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__', '__spec__', 'winos']
>>> winos.getcwd()
'C:₩₩Anaconda3₩₩Lib₩₩idlelib'
>>>
```

4) 파이썬 내장 함수

파이썬은 자주 사용되는 함수를 내장 함수(Built-in Functions)라는 이름으로 기본적으로 제공합니다. 예를 들어, 절댓값을 구하거나 원소의 개수를 세는 것과 같은 함수가 내장 함수에 해당합니다. 표 5.1 은 파이썬이 제공하는 내장 함수의 목록입니다. 이번 절에서는 이 가운데 자주 사용되는 내장 함수에 대해서만 간단히 알아보겠습니다.

표 5.1 파이썬 내장 함수

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()	

abs(x) 내장 함수는 정수형 또는 실수형 값을 입력받은 후 해당 값의 절댓값을 반환하는 함수입니다. 다음과 같이 내장 함수의 인자로 음수 값을 입력했을 때 입력 값의 절댓값이 반환되는 것을 확인할 수 있습니다. 다만 정수형 또는 실수형이 아닌 문자열 값이 입력되면 오류가 발생합니다.

```
>>> abs(-3)
3
>>> abs(-3.0)
3.0
>>>
>>> abs("hello")
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    abs("hello")
TypeError: bad operand type for abs(): 'str'
>>>
```

chr(i) 내장 함수는 유니코드 값을 입력받은 후 해당 값에 해당하는 문자열을 반환합니다. 예를 들어, 아스키(ASCII) 코드에서 알파벳 대문자 A 는 65, 소문자 a 는 97 에 해당하는데, 해당하는 정숫값을 chr() 함수의 인자로 전달하면 문자열이 반환되는 것을 확인할 수 있습니다.

```
>>> chr(97)
'a'
>>> chr(65)
'A'
>>>
```

enumerate() 내장 함수는 입력으로 시퀀스 자료형(리스트, 튜플, 문자열) 등을 입력받은 후 enumerate 객체를 반환합니다. enumerate() 내장 함수는 다음과 같이 for 문에서 시퀀스 자료형 내의 값과 인덱스를 동시에 구하고 싶을 때 자주 사용됩니다.

```
>>> for i, stock in enumerate(['Naver', 'KAKAO', 'SK']):
```

```
        print(i, stock)
```

```
0 Naver
1 KAKAO
2 SK
>>>
```

다음 코드는 enumerate() 내장 함수를 사용하지 않고 위의 코드를 다시 구현한 것입니다. 이 경우 시퀀스 자료형인 리스트는 따로 인덱스를 갖고 있지 않으므로 i 라는 변수를 통해 따로 인덱스값을 계산해야 하는 번거로움이 있습니다.

```
>>> i = 0
>>> for stock in ['Naver', 'KAKAO', 'SK']:
        print(i, stock)
        i += 1
```

```
0 Naver
1 KAKAO
2 SK
>>>
```

id(object) 내장 함수는 객체를 입력받아 해당 객체의 고윳값을 반환합니다. 고윳값은 파이썬 인터프리터의 구현 방식에 따라 다르지만 보통 객체가 할당된 메모리의 주소 값을 의미합니다. 다음과 같이 a 라는 변수와 b 라는 변수가 모두 1 이라는 값을 바인딩하고 있을 때 a 와 b 가 서로 같은 객체를 바인딩하고 있는지 확인하려면 id() 내장 함수의 반환값이 같은지 확인하면 됩니다. id() 내장 함수의 반환값이 같은 경우 두 변수는 서로 같은 객체를 바인딩하고 있음을 의미합니다.

```
>>> a = 1
>>> b = 1
```

```
>>> id(a)
1918947744
>>> id(b)
1918947744
>>>
```

len(s) 내장 함수는 리스트, 튜플, 문자열, 딕셔너리등을 입력받아 그 객체의 원소 개수를 반환합니다.

```
>>> len(['SK', 'naver'])
2
>>> len('SK hynix')
8
>>> len({1:'SK', 2:'Naver'})
2
>>>
```

list() 내장 함수는 문자열이나 튜플을 입력받은 후 리스트 객체로 만들고 해당 리스트를 반환합니다. 다음과 같이 튜플을 리스트로 변환할 때 자주 사용됩니다.

```
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
>>> list((1,2,3))
[1, 2, 3]
>>>
```

max() 내장 함수는 입력값 중 최댓값을 반환합니다. 반대로 min() 내장 함수는 입력값 중 최솟값을 반환합니다.

```
>>> max(1,2,3)
3
>>> max([1,2,3])
3
>>> min(1,2,3)
1
>>> min([1,2,3])
1
>>>
```

sorted() 내장 함수는 입력값을 정렬한 후 정렬된 결괏값을 '리스트'로 반환합니다.

```
>>> sorted((4,3,1,0))
[0, 1, 3, 4]
>>> sorted([5, 4, 3, 2, 1])
[1, 2, 3, 4, 5]
>>> sorted(['c', 'b', 'a'])
['a', 'b', 'c']
>>>
```

int(x) 내장 함수는 문자열을 인자로 입력받아 해당 문자열을 정수형으로 변환한 후 반환합니다. 반대로 str(x) 내장 함수는 객체를 입력받아 문자열로 변환합니다. int(x)와 str(x) 내장 함수는 다음과 같이 문자열을 정수형으로, 정수형을 문자열로 변환할 때 자주 사용됩니다.

```
>>> int('3')
3
>>> str(3)
```

```
'3'
```

```
>>>
```

1) 연습문제

문제 5-1

두 개의 정수 값을 받아 두 값의 평균을 구하는 함수를 작성하세요.

```
def myaverage(a, b):
    # 함수 구현
```

문제 5-2

함수의 인자로 리스트를 받은 후 리스트 내에 있는 모든 정수 값에 대한 최댓값과 최솟값을 반환하는 함수를 작성하세요.

```
def get_max_min(data_list):
    # 함수 구현
```

문제 5-3

절대 경로를 입력받은 후 해당 경로에 있는 *.txt 파일의 목록을 파이썬 리스트로 반환하는 함수를 작성하세요.

```
def get_txt_list(path):
    # 함수 구현
```

문제 5-4

체질량 지수(Body Mass Index, BMI)는 인간의 비만도를 나타내는 지수로서 체중과 키의 관계로 아래의 수식을 통해 계산됩니다. 여기서 중요한 점은 체중의 단위는 킬로그램(kg)이고 신장의 단위는 미터(m)라는 점입니다.

BMI=체중(kg)신장(m)2BMI=체중(kg)신장(m)2

일반적으로 BMI 값에 따라 다음과 같이 체형을 분류하고 있습니다.

```
BMI <18.5, 마른체형
18.5 <= BMI < 25.0, 표준
```

```
25.0 <= BMI < 30.0, 비만
```

```
BMI >= 30.0, 고도 비만
```

함수의 인자로 체중(kg)과 신장(cm)을 받은 후 BMI 값에 따라 '마른체형', '표준', '비만', '고도 비만' 중 하나를 출력하는 함수를 작성하세요.

문제 5-5

사용자로부터 키(cm)와 몸무게(kg)를 입력받은 후 BMI 값과 BMI 값에 따른 체형 정보를 화면에 출력하는 프로그램을 작성해 보세요. 파이썬에서 사용자로부터의 입력은 input() 함수를 사용하며, 작성된 프로그램은 계속해서 사용자로부터 키와 몸무게를 입력받은 후 BMI 및 체형 정보를 출력해야 합니다(무한 루프 구조).

문제 5-6

삼각형의 밑변과 높이를 입력받은 후 삼각형의 면적을 계산하는 함수를 작성하세요.

```
def get_triangle_area(width, height):
    # 함수 구현
```

문제 5-7

함수의 인자로 시작과 끝 숫자를 받아 시작부터 끝까지의 모든 정수값의 합을 반환하는 함수를 작성하세요(시작값과 끝값을 포함).

```
def add_start_to_end(start, end):
    # 함수 구현
```

문제 5-8

함수의 인자로 문자열을 포함하는 리스트가 입력될 때 각 문자열의 첫 세 글자로만 구성된 리스트를 반환하는 함수를 작성하세요. 예를 들어, 함수의 입력으로 ['Seoul', 'Daegu', 'Kwangju', 'Jeju']가 입력될 때 함수의 반환값은 ['Seo', 'Dae', 'Kwa', 'Jej']입니다.