Python 01 시작하기

유현조

2022년 3월 9일

차 례

1

시작하기

1.1 대화형 모드

보통 프로그래밍을 한다고 하면 소스 코드 파일을 작성하여 프로그램을 완성하고 이것을 실행시키는 방식으로 진행한다. 파이썬은 이런 방식 외에 대화형 모드(interactive mode)를 지원한다. 대화형 모드에서는 사용자가 하나의 명령을 입력하고 엔터를 치면 즉시 그 명령이 수행된다. 파이썬을 처음 배울 때 연습용으로 사용할 수 있다. 파이썬이 각 명령 마다 어떤 방식으로 돌아가는지 확인하며 파이썬과 친해질 수 있다. 대화형 모드는 1960년대 Lisp 언어을 위한 고안된 REPL (Read-Eval-Print Loop)이 시초라고 할 수 있다. 파이썬이 대화형 모드를 지원한 것은 매우 성공적이었고 이후 현대적인 언어들은 대부분 대화형 모드를 지원하는하고 있다. 처음에는 교육용 목적으로 지원된 것이라 할 수 있으나 데이터과학의 인기와 함께 탐색적 프로그래밍이 중요해 지며 대화형 모드는 필수적인 기능이 되었다.

다음과 같이 아무 인자없이 파이썬 인터프리터를 실행시키면 대화형 환경이 시작된다. `python`을 실행하면 다음과 같이 간략한 버전 정보가 출력된 후 사용자의 명령을 기다리는 프롬프트가 나온다. 파이썬의 프롬프트는 >>> 이다.

```
$ python
Python 3.8.1 (default, Dec 27 2019, 18:05:50)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

여기서 `python`은 Python 인터프리터의 이름이다. 즉 실행 파일의 이름이다. 시스템에 따라 `python -i`와 같이 옵션을 주어야 대화형 모드로 실행되기도 한다.

1.2 계산기로 사용하기

1.2.1 산술 연산자

파이썬 프로그램 짜는 법을 전혀 모르더라도 대화형 모드에서는 계산기로 유용하게 사용할 수 있다. 계산의 기본은 사칙연산이다. 덧셈(+), 뺄셈(-), 곱셈(*), 나눗셈(/) 기호를 사용하면 된다. 숫자와 연산자 사이에 공백을 두는 것이 일반적인 관습이고 보기에 좋다.

```
>>> 2 + 3
5
>>> 11 - 4
7
>>> 2 * 3
6
>>> 3 / 2
1.5
```

거듭제곱을 계산하려면 **을 사용한다. 다른 프로그래밍 언어에서는 거듭제곱을 ^로 표기하기도 한다. 이 기호는 삿갓표로 불리기도 하고 영어의 캐럿(caret)이라는 용어를 그대로 사용하기도 한다. 파이썬에서 이 기호는 비트별 배타적 논리합(Bitwise Exclusive Or)이라는 전혀 다른 의미로 사용되니 다른 언어 사용자는 혼동하지 않도록 주의하자.

```
>>> 2**10
1024
```

몫은 // 연산자를 이용하여 나머지는 % 연산자를 이용하여 구할 수 있다. 나머지라는 개념은 유용하게 사용되는 경우가 많다. 프로그래밍에서 사칙연산과 더불어 가장 기본적인 산술 연산자의 하나로 취급된다.

```
>>> 10 // 3
3
>>> 10 % 3
1
```

1.2.2 연산자 우선 순위

연산자 우선 순위라는 개념은 모두 잘 알고 있을 것이다. $3 + 4 \times 2$와 같은 식이 주어진다면 곱하기를 먼저하고 더하기를 나중에 하는 것 말이다. 즉 명시적으로 표현하면 $3 + (4 \times 2)$와 같이 쓸 수 있다. 파이썬에서도 수학 시간에 배운 것과 같은 연산자 우선순위가 지켜진다.

```
>>> 3 + 4 * 2
11
```

이러한 연산 순서는 $(3+4)\times 2$와 같이 괄호를 이용하면 원하는 대로 바꿀 수도 있다. 파이썬에서도 이와 같은 원칙이 그대로 적용된다.

```
>>> (3 + 4) * 2
14
```

단, 수학 시간에는 다양한 형태의 괄호를 사용해도 되었지만 파이썬에서는 오로지 소괄호만 이용해야 한다. 소괄호 `()`, 중괄호 `{}`, 대괄호 `[]`는 파이썬에서 각각 다른 의미를 가진다. 프로그래밍에서는 이들 괄호에 소, 중, 대라는 크기의 의미가 없다. 괄호의 모양에 따라 소괄호를 둥근 괄호 또는 그냥 괄호, 대괄호를 꺾쇠괄호 또는 네모괄호 등으로 다양하게 불린다. 중괄호는 따로 많이 사용되는 이름이 없지만 영어의 'curly bracket'과 유사하게 꼬불괄호 정도로 부르자는 의견도 있다.

수식으로는 시각적으로 쉽게 이해할 수 있는 경우에도 파이썬에서는 많은 괄호의 중첩 때문에 이해가 어려운 경우가 있다. $n=3$이 주어졌을 때

$$\frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+n}}}}$$

을 파이썬에서 계산해보자.

```
>>> 1 / (1 + 1 / (1 + 1 /(1 + 1/(1 + 3))))
0.6428571428571428
```

괄호를 정확히 쳐주지 않으면 오류가 나거나 엉뚱한 결과를 얻을 수 있다. 오류가 나면 사실 다행이다. 고칠 기회가 주어지기 때문이다. 하지만 오류가 나지 않게 잘못 괄호를 치고 프로그램이 잘 돌아간다면 문제다. 이런 작은 실수가 재앙적 결과를 가져오는 일들이 실제로 발생한다. 프로그래밍에서 괄호의 중첩은 오류의 주요 원인 중의 하나이며 프로그래머가 괄호의 짝이 맞는지 육안으로 확인하는 것은 쉽지 않다. 이 때문에 프로그래밍을 위한 텍스트 에디터들은 대부분 괄호의 짝을 시각적으로 표시하는 기능을 제공한다. 괄호가 많은 식을 써야 한다면 항상 주의하자. 가능하면 하나의 식에 괄호가 너무 많아지지 않도록 분리해서 쓰는 것이 더 좋다.

연습 1.1 다음 식의 값을 파이썬을 이용하여 계산해 보자.

1. $7+4\times 3$
2. $13-\frac{19}{5+6}$
3. $5+\frac{5\times(2+\frac{1}{2})-7}{3}$

1.2.3 수학 모듈

지수, 로그, 삼각 함수 등 수학 함수가 필요하다면 `math` 모듈을 사용해야 한다. 프로그래밍에서 어떤 기능을 미리 만들어 모아 놓은 것을 라이브러리(library)라고 한다. 대개의 프로그래밍 언어에서 기본적이고 필수적이고 중요하다고 생각되는 기능을 모아 제공하며 이것을 표준 라이브러리(standard library)라고 부른다. 파이썬 표준 라이브러리(Python Standard Library)에 어떤 기능이 있는지 미리 둘러본다면 나중에 유용하게 쓸 수 있을 것이다. 반드시 사이트를 방문해 보자. 파이썬에서는 라이브러리에 들어있는 각각의 개별 기능을 제공하는 단위를 모듈(module)이라고 부른다. 모듈을 모아놓은 묶음은 패키지(package)라고 한다. 모듈, 패키지, 라이브러리라는 용어는 비슷한 의미로 혼란스럽게 사용되는 경우가 많으나 적절히 이해하자. 파이썬에서 모듈은 하나의 소스 코드 파일로 작성된 것을 말하고 이것들을 묶어놓은 것을 패키지, 패키지 여러 개를 모아 놓은 것을 라이브러라고 구분하기는 한다. 모듈을 이용하기 위해서는 우선 `import` 명령으로 불러들어야 한다.

```
>>> import math
>>> math.sqrt(2)
1.4142135623730951
>>> math.log(3)
1.0986122886681098
```

위 예는 `math` 모듈을 `import` 한 후에 이 모듈에 들어있는 `log()` 함수를 사용한 것이다. `math.log()` 처럼 마침표를 찍어 구분한다. 어떤 라이브러리 안에 패키지 안에 모듈 안에 함수가 들어있을 때 각 단계의 이름 사이에 점을 찍어 연결한다.

1.3 소스 코드 파일 작성하기

대화형 모드는 명령 하나씩 즉시 그 결과를 알 수 있다는 점에서 편리하지만 본격적으로 프로그램을 작성할 수 있는 환경은 아니다. 정말 프로그램을 짜려면 소스 코드를 파일로 만들어야 한다.

소스 코드 파일의 파일 형식은 텍스트 파일에 불과하다. 메모장에서 작성해도 된다. 파이썬 소스 코드는 보통 파일명 뒤에 `.py` 확장자를 덧붙인다. 이것은 단지 이름일 뿐 여전히 텍스트 파일이다. 확장자를 안 붙이거나 다르게 붙인다고 해서 실행이 안 되는 것은 아니다. 확장자는 편의상의 약속일 뿐이다. 하지만 약속이니 지키는 것이 좋다.

우선 간단하게 “Hello World”를 출력하는 프로그램을 작성해 보자. 예를 들어 `hello.py`라는 파일에 다음 내용을 그대로 넣어보자.

```
print('Hello World')
```

문자를 그대로 사용하고 싶을 때에는 따옴표를 쳐주어야 한다. print는 출력하라는 의미의 명령이다. 위와 같은 내용의 파일을 저장하고 실행은 다음과 같이 한다.

```
$ python hello.py
```

대화형 모드에서는 3 + 4와 같이 어떤 식을 입력하고 엔터를 치면 바로 그 식의 값인 7이 출력이 되지만 파일로 작성한 프로그램에서는 명시적으로 출력 명령을 써주어야 결과가 출력이 된다. 예를 들어,

```
print(3 + 4)
```

라고 해 주어야 이 소스 코드 파일을 실행했을 때 7이 출력되는 것을 볼 수 있다.

1.3.1 주석 달기

주석(comment)은 컴퓨터에게, 즉, python 인터프리터에게 무시하고 지나가도록 지정한 부분을 말한다. 컴퓨터에게는 필요없고 사람에게 필요한 부분이다. 프로그램을 작성하다 보면 무엇인가 추가 설명을 하는 것이 유용할 때가 있다. 다른 사람이 소스 코드를 읽을 때 이해하는 데에 도움이 될 만한 것들이 있다. 그런 내용을 주석으로 작성한다. 파이썬에서 샵(#) 문자를 사용하면 그 이후부터 행의 끝까지 주석으로 처리된다. 주석으로 처리된다는 것은 파이썬 인터프리터가 무시한다는 뜻이다.

다음처럼 파일의 가장 앞부분에 이름과 프로그램에 대한 필요한 설명을 써주고 이어서 저작권과 사용권(라이선스)을 명기하는 것이 일반적이다.

```
# hello.py
# Hello World Program
#
# Copyright (c) 2016 Foobar
#
# This program is free software: ...
print("Hello World")
```

1.3.2 실행 파일처럼 작동하게 만들기

마이크로소프트 Windows 운영체제에서 파이썬 프로그래밍을 하고 있다면 이 내용은 건너뛰어도 된다. 유닉스 계열의 운영체제에서는 매우 유용하게 사용된다. 주석 중에는

매우 특별한 것으로 'Hashbang'이라 불리는 주석이 있다. 소스 코드의 첫 행(경우에 따라 둘째 행)에만 쓸 수 있으며 #!로 시작한다. 해당 스크립트를 실행시키는 데 필요한 인터프리터의 경로를 명시해 준다. 예를 들면:

```
#!/usr/bin/env python3
# hello.py
# Hello World Program

print("Hello World")
```

위와 같이 hashbang 주석이 있을 경우 다음과 같이 실행 퍼미션을 주면

```
$ chomd +x hello.py
```

다음처럼 스크립트 파일 이름만 써서 실행시킬 수 있다.

```
$ ./hello.py
```

이런 경우 확장자를 생략하고 파일 이름을 hello로 바꾸면 다음처럼 실행시킬 수 있게 된다.

```
$ ./hello
```

이제 이 파일을 PATH에 넣으면 다른 디렉토리에서도 간편하게

```
$ hello
```

로 실행시킬 수 있다. 대개의 경우 자신의 홈 디렉토리 아래 ~/bin 디렉토리에 넣으면 된다.

1.4 프로그램을 구성하는 기본 요소

1.4.1 자료형와 변수

파이썬은 여러 가지 자료, 즉 여러 가지 값들을 받아들일 수 있다. 그런 값들을 유형에 따라 구분한 것을 자료형(data type)이라고 한다. '자료'라는 말을 떼고 그냥 '타입(type)'이라고 하는 경우도 많다. 계산하기에서 본 것처럼 3과 같은 정수 값과 3.14와 같은 소수 값을 구별하는데 정수 값은 int 형, 소수 값은 float 형이라고 한다. 그리고 하나 더 자주 쓰이는 자료형이 있는데 문자로 된 자료이다. 다음처럼 따옴표로 둘러싸서 표현한다.

```
>>> 'Python'
'Python'
```

이런 자료를 문자들이 줄 지어 있는 것이라는 의미에서 문자열이라고 하는데 영어로는 'string'이라고 한다. 파이썬에서는 `str` 형이라고 한다.

어떤 값에 이름을 부여하면 더 편리하게 계산할 수도 있다. 이러한 것을 변수(variable)라고 한다. 새로운 변수 또는 기존의 변수에 값을 넣는 방법은 다음과 같다.

```
>>> x = 3
>>> y = 4
>>> x + y
7
>>> x = 5
>>> x
5
```

변수는 단순 계산을 넘어 프로그래밍이라는 것을 가능하게 해주는 첫단계이다. 변수는 단순히 특정값을 기억하는 장소를 넘어 추상적인 논의를 가능하게 해준다. $3+4$이라는 산수를 넘어 $x+y$라는 추상적인 식을 논할 수 있게 해주는 것이다.

때로는 자료가 여러 개의 값으로 이루어진 경우가 있다. 예를 들어, 5일간 최저 기온을 기록한 자료가 있다고 생각해 보자. 이 값들에 어떻게 이름을 붙여야 할까? 다음처럼?

```
x = 13.8
y = 15.7
z = 17.0
v = 18.1
w = 18.5
```

만약 100개의 값에 이름을 붙여야 한다면 영어 소문자로는 모자랄 것이다. 아마도 다음과 같은 방법을 선택하게 될 것이다.

```
t1 = 13.8
t2 = 15.7
t3 = 17.0
t4 = 18.1
t5 = 18.5
```

컴퓨터과학자들은 이 보다 더 우아한 해결책을 만들어 놓았다. 파이썬에서는 꺾쇠괄호로 둘러싸고 쉼표로 값들을 나열한 후 이 자료 전체에 하나의 변수 이름만 부여할 수 있다.

```
t = [13.8, 15.7, 17.0, 18.1]
```

이렇게 만들어진 변수 이름 뒤에 꺾쇠를 치고 번호를 넣으면 개별 값을 불러올 수도 있고 바꿔 넣을 수도 있다.

```
>>> t = [13.8, 15.7, 17.0, 18.1]
>>> t
[13.8, 15.7, 17.0, 18.1]
>>> t[0]
13.8
>>> t[1] = 16.7
```

```
>>> t
[13.8, 16.7, 17.0, 18.1]
```

번호가 0부터 시작한다는 점에 유의하자. 이렇게 꺾쇠괄호로 표현된 자료를 파이썬에서 list 형의 자료라고 한다. 리스트에는 새로운 값을 추가할 수도 있다.

```
>>> t.append(18.5)
>>> t
[13.8, 16.7, 17.0, 18.1, 18.5]
```

여기서 append()와 같은 것을 메소드(method)라고 부른다는 정도만 알아 두자. 객체 지향 프로그래밍에서 사용하는 용어이다. 리스트 a가 있을 때 이름 다음에 점을 찍고 a.append(x)라고 하면 a에 x를 추가해 준다.

여러 개의 값으로 이루어진 데이터가 있을 때 방금 본 리스트는 번호를 붙여 값들을 나열하는 방식이었다. 그런데 어떤 경우에는 값들에 번호를 붙이는 것보다 이름을 붙이는 것이 적절할 때가 있다. 이를 위해 파이썬에서는 dict라는 자료형을 제공한다. 중괄호로 묶고 이름과 값의 짝을 쌍점으로 대응시키고 쉼표로 구분하여 나열한다.

```
>>> temp = {'seoul' :  3.0, 'incheon' : 2.7, 'daejeon': 6.5, 'busan' : 9.9}
>>> temp
{'seoul' :  3.0, 'incheon' : 2.7, 'daejeon': 6.5, 'busan' : 9.9}
```

특정 값을 찾기 위해서는 꺾쇠괄호 안에 이름을 넣어 주면 된다.

```
>>> temp['seoul']
3.0
>>> temp['busan']
9.9
```

새로운 이름-값의 짝을 추가할 수도 있다.

```
>>> temp['jeju'] = 13.3
>>> temp
{'seoul' :  3.0, 'incheon' : 2.7, 'daejeon': 6.5, 'busan' : 9.9, 'jeju' : 13.3}
```

1.4.2 함수

프로그래밍 언어들은 사용자 스스로 새로운 함수를 만들 수 있는 기능을 제공한다. 예를 들어, 다음과 같이 두 수의 합을 계산하는 함수를 만들어 사용할 수 있다. 파이썬에서 함수를 작성할 때 def, return, 쌍점(:)과 같은 형식이 사용된다는 점을 눈여겨 보자.

```
>>> def add(x, y) : return x + y
>>> add(3, 4)
7
```

함수는 변수와 더불어 프로그래밍의 가장 중요한 요소이다. 프로그래밍의 기초 단계는 대개 함수를 작성하는 과정이라고 할 수 있다. 어떤 관점에서는 프로그램 자체를 하나의 함수로 보기도 한다.

대화형 모드로 함수를 새로 만들어 사용하는 일은 많지 않다. 함수로 만든다는 것은 나중에 두고 두고 쓰려는 의도를 가지고 있는 것이기 때문에 대개 파일로 작성하게 된다. 위 함수를 작동 예와 함께 하나의 파일로 작성해 보자.

```
def add(x, y):
    return x + y

x = 3
y = 4
s = add(x, y)
print(s)
```

함수를 작성할 때 함수의 내용에 해당하는 부분은 들여쓰기를 한다. 지금은 간단한 예로 내용이 한 줄인 함수를 보았지만 보통은 여러 줄로 작성하게 되며 들여쓰기를 함으로써 어디서부터 어디가 하나의 함수인지 눈으로 쉽게 구별할 수 있게 된다.

대부분의 프로그래밍을 위한 텍스트 에디터에서는 자동으로 들여쓰기를 해주며 탭 키를 이용하여 들여쓰기 단계를 조절할 수 있다. 탭 키를 사용한다고 해서 탭 문자를 사용하라는 의미는 아니다. 들여쓰기를 할 때 탭을 칠 것인가 스페이스를 사용할 것인가하는 것이 문제가 되기도 한다. 파이썬에서는 공식적으로 스페이스로 들여쓰는 것이 권장된다. 기존에 탭으로 들여쓰기가 되어있다면 반드시 탭만 사용해야 한다. 탭과 스페이스를 섞어서 쓰면 안 된다. 파이썬을 지원하는 텍스트 에디터들에서는 대개 탭 키를 치면 자동으로 스페이스 4개를 입력해 주는 방식으로 작동한다. 진짜 탭 문자를 입력하는 것과는 다르다. 메모장 같은 경우에는 탭 키를 치면 정말 탭 문자가 입력되지만 프로그래밍을 위한 에디터들에서 탭 키는 상황에 따라 다르게 작동하는 경우가 많다.

연습 1.2 전세계 거의 모든 나라가 온도를 나타낼 때 섭씨를 사용하는 데에 비해 미국에서는 화씨를 사용한다. 섭씨를 화씨로 바꾸는 함수 `cels2fahr()`와 화씨를 섭씨로 바꾸는 함수 `fahr2cels()`를 작성해 보아라. 섭씨 온도 C와 화씨 온도 F 사이에는 다음과 같은 관계가 있다.

$$C = \frac{5}{9}(F - 32), \quad F = \frac{9}{5}C + 32$$

이 함수를 작성하면 다음과 같이 사용할 수 있게 된다.

```
>>> fahr2cels(100)
37.77777777777778
>>> cels2fahr(18)
64.4
```

1.4.3 조건문

컴퓨터가 상황에 따라 다르게 작동하도록 해주는 것이 조건문이라는 형식이다. 마치 컴퓨터가 지능을 가진 것처럼 행동하도록 해주는 부분이라고 할 수 있다.

여기에서는 간단하게 조건에 따라서 다른 값을 가지는 수학 함수를 예로 들어 조건문 사용법을 알아보자. 다음과 같은 함수를 생각해 보자.

$$f(x) = \begin{cases} 1, & x > 0 \\ 0, & \text{otherwise} \end{cases}$$

파이썬에서 조건문은 `if`와 `else`를 기본으로 하여 구성된다. 이 함수를 정의 그대로 파이썬으로 작성한다면 다음과 같다.

```
def f(x) :
    if x > 0 : return 1
    else     : return 0
```

보통은 조건문 안의 실행 내용이 길어지기 때문에 다음처럼 줄을 바꾸어 쓰게 된다.

```
def f(x) :
    if x > 0 :
        return 1
    else :
        return 0
```

조건문의 실행 내용에 해당되는 부분은 조건보다 한 단계 들여써야 한다. 결국 논리적인 프로그램 덩어리와 들여쓰기 구조가 일치하게 된다.

1.4.4 반복문

컴퓨터는 계산기라는 그 이름 그대로 정확하고 빠른 계산 능력으로 인간을 초월했다. 아마도 그 다음으로는 동일하거나 규칙적인 동작을 정확히 반복해 내는 것이 기계가 인간보다 강점을 가지는 영역일 것이다. 우리가 컴퓨터에게 시키려는 일은 대부분 무엇인가를 반복하는 일이다.

파이썬에서 반복문은 `for in` 구문을 이용하여 작성한다. 예를 들어, 리스트 자료의 값을 한 행에 하나씩 출력해 보자.

```
temperatures =  [13.8, 15.7, 17.0, 18.1, 18.5]

for t in temperatures:
    print(t)
```

여기에서 '`for` x `in` S'라는 구문을 보자. S라는 목록에 들어있는 원소를 차례대로 하나씩 꺼내면서 그것을 x라고 이름을 붙여서 처리하겠다는 의미이다.

이제 조건문과 반복문을 조합하여 사용해 보자. 이것이 바로 컴퓨터를 이용한 데이터 처리의 핵심 구조이라고 할 수 있다. 온도가 18도 보다 낮으면 온도와 함께 'cold'라고 출력하고 아니면 'warm'이라고 출력하는 프로그램은 다음과 같이 작성할 수 있다.

```
temperatures =  [13.8, 15.7, 17.0, 18.1, 18.5]

for t in temperatures:
    if t < 18 :
        print(t, 'cold')
    else :
        print(t, 'warm')
```

연습 1.3 섭씨를 화씨로 바꾸는 함수 `cels2fahr()`를 이용하여 리스트에 들어 있는 섭씨 온도를 화씨 온도로 바꾸어 출력하는 프로그램을 작성해 보아라.

1.5 입력과 출력

프로그래밍에서 입출력(I/O, Input/Output) 기능은 완성된 프로그램의 사용이라는 측면에서 매우 중요한 기능이다. 예를 들어 다음과 같은 프로그램은 하나의 온전한 프로그램이다.

```
x = 3
y = 4
m = (x + y) / 2
```

이 코드를 다음과 같이 함수를 이용하여 작성하면 더 일반화되고 활용도가 높은 코드가 된다는 것도 배웠다. 일련의 계산과정을 정리하여 입력과 출력이 있는 함수 형태로 만들었다.

```
def average(x, y):
    m = (x + y) / 2
    return m

x = 3
y = 4
m = average(x, y)
```

하지만 이 프로그램을 실행시켰을 때 우리는 아무 것도 얻을 수 없다. 계산된 결과를 알기 위해서는 출력 기능을 이용해야 한다는 것을 앞서 배웠다. 컴퓨터에서 출력의 형태는 다양할 수 있지만 가장 대표적인 것은 모니터 화면으로 출력하거나 파일로 저장하는 것이다. 우리는 `print()` 함수를 이용해 화면으로 출력하는 것을 배웠다.

```
def average(x, y):
    m = (x + y) / 2
    return m

x = 3
y = 4
m = average(x, y)
print(m)
```

그런데 위와 같이 프로그램을 작성한 후에 우리가 5와 6의 평균을 구해야 한다면? 소스 코드를 다음과 같이 고쳐서 저장한 후에 실행해야 할 것이다.

```
def average(x, y):
    m = (x + y) / 2
    return m

x = 5
y = 6
m = (x + y) / 2
print(m)
```

또 다른 숫자들의 평균을 내야 한다면? 그 때마다 소스 코드를 고치는 것은 비효율적일 뿐만 아니라 위험하다. 우리에게 필요한 것은 프로그램을 작성하는 순간에는 아직 정해지지 않은 정보, 그리고 프로그램이 실행되는 순간에 주어질 정보를 처리할 수 있는 장치가 필요하다. 그것이 프로그램의 외부에 있는 정보를 입력 받는 기능이다.

프로그램 외부에서 주어지는 자료를 받아서 처리하는 방법에는 여러가지가 있다. 대부분의 경우 파일 형태로 자료가 주어지므로 파일 입출력 기능을 이용해 데이터를 읽어들이고 결과를 저장할 수 있다. 여기에서는 파이썬에서 파일을 직접 다루는 방법은 잠시 유보하고 키보드 입력을 받는 방법, 명령행에서 데이터를 받는 방법과 표준 입력으로 파일의 내용을 받는 방법을 살펴보자.

1.5.1 키보드 입력 받기

사용자가 키보드로 입력한 내용을 `input()` 명령을 이용하여 받아 사용할 수 있다. 다음처럼 소스 코드를 작성해 보자.

```
# hello.py
print("What's your name?")
name = input()
print("Hello", name)
```

실행시키면 'What's your name?'이라는 메시지가 출력되고 커서가 멈추어 있는 것을 보게 될 것이다. 여기에 이름을 입력하면 'Hello'와 함께 입력한 이름이 출력된다.

```
$ python hello.py
What's your name?
Alice
Hello Alice
```

1.5.2 숫자로 바꾸기

숫자를 입력 받는 것은 어떨까? 다음과 같이 숫자 3을 입력 받아 보자.

```
>>> x = input()
3
```

그런데 덧셈을 하려고 하면 오류가 발생한다.

```
>>> x + 4
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

오류 메시지를 읽어보자. 오류 메시지를 반드시 읽는 습관을 들이자. `TypeError`라는 말은 타입, 즉, 자료형에 문제가 있음을 알려준다. 이어지는 상세 정보에서는 `int`를 `str`로 바꿀 수 없다라는 말이 나온다. 컴퓨터가 이렇게 말하는 것이다. '지금 나한테 인간이 정수(int)를 주었는데 필요한 것은 문자열(str)이다. 그래서 인간이 준 정수를 조용히(implicitly) 인간에게 물어보지 않고 문자열로 바꾸려고 시도해 봤는데 안 된다.'

어떤 값이 들어있는지 살펴보면 숫자에 따옴표가 쳐져 있는 것을 볼 수 있다. ≫〉 x '3' #+END$_{\text{EXAMPLE}}$ 따옴표로 둘러싸여 있다는 것은 숫자가 아니라 문자로 처리되었다는 의미이다. 오류 메시지를 다시 보자. 변수 x에 문자열이 들어있을 때 파이썬에서는 다른 문자열과 덧셈(+)을 하는 것이 가능하다. 하지만 문자열에 정수를 더할 수는 없다. 이 때문에 파이썬은 4라는 정수를 문자열로 바꾸려고 시도했던 것이다.

외부 입력은 기본적으로 문자로 처리된다. 이것을 숫자로 바꾸기 위해 정수 형태라면 `int()`, 소수 형태라면 `float()`를 이용한다. 예를 들면 다음처럼:

```
>>> x = '3'
>>> n = int(x)
>>> n
3
```

연습 1.4 화씨 온도를 키보드로 입력한 것을 `input()`으로 받은 후에 섭씨 온도로 바꾸어 출력하는 프로그램을 작성하여라.

1.5.3 명령행 인자 받기

명령행 인자를 받기 위해서는 sys 모듈이 필요하다. 명령행에 주어진 데이터는 sys.argv를 통해 사용할 수 있다.

```
# hello2.py
import sys

name = sys.argv[1]
print("Hello", name)
```

위 프로그램은 명령행에 주어진 첫단어를 name이라는 변수에 저장하고 그것을 출력해 준다. 다음과 같이 실행해보자.

```
$ python hello2.py Alice
Hello Alice
```

이 때 명령행 인자는 기본적으로 공백문자에 의해 구분된다. 따라서 다음과 같이 명령행 인자를 주면 첫번째 인자만 프로그램에 의해 처리된다.

```
$ python hello2.py Lewis Carroll
Hello Lewis
```

두번째 인자는 sys.argv[2]로 접근할 수 있으나 우리가 작성한 프로그램에서 이것에 대해 어떤 처리도 하지 않았으므로 무시된다. 만일 공백문자가 포함된 전체를 하나의 인자로 주고 싶다면 다음과 같이 따옴표를 치면 된다.

```
$ python hello2.py "Lewis Carroll"
Hello Lewis Carroll
```

연습 1.5 화씨 온도를 명령행에서 받아 섭씨 온도로 바꾸어 출력하는 프로그램을 작성하여라. 다음처럼 작동한다.

```
$ python3 fahr2cels.py 100
37.77777777777778
```

1.5.4 표준 입력 읽기

데이터 파일이 주어졌을 때 파이썬의 파일 입출력 기능을 이용하여 직접 파일을 읽거나 쓸 수도 있지만 표준 입력을 이용할 수도 있다. 셸에서 제공하는 redirection 기능을 이용하면 된다. 물론 redirection을 하지 않으면 원래의 기본 입력인 키보드로부터 데이터를 받는다. 다음과 같이 소스 파일을 작성해보자.

```
# cat.py
import sys

for line in sys.stdin:
    sys.stdout.write(line)
```

여기서 sys.stdin은 표준 입력을 의미하고 sys.stdout은 표준 출력을 의미한다. 표준 입력 sys.stdin을 구성하는 단위는 행이다. 그러므로 위 코드는 표준 입력에서 받은 텍스트에 들어있는 각각의 행을 그대로 표준 출력으로 내보내는 것이다. 다음과 같은 내용을 담은 fishes.txt라는 텍스트 파일이 있다고 하자

```
cod
eel
mackerel
salmon
trout
```

다음과 같이 실행해 보자. 텍스트 파일의 내용이 그대로 화면으로 출력될 것이다.

```
$ python cat.py < fishes.txt
```

다음처럼 입력을 주지 않으면

```
$ python cat.py
_
```

아무 내용도 출력되지 않고 커서가 멈추어 있는 것을 보게 될 것이다. 이것은 입력을 기다리고 있는 것이다. 키보드로 다음처럼 입력을 하고 엔터 키를 쳐보자.

```
$ python cat.py
apple
```

프로그램을 종료하기 위해서는 Control+D를 누르자.

참고로 마이크로소프트 PowerShell에서는 < 입력을 지원하지 않으므로 다음과 같은 방법을 사용해야 한다.

```
PS> Get-Content | python cat.py
apple
```

연습 1.6 지구의 기온 변화에 관한 보고서를 작성하려는 친구가 도움을 청한다. 세계 각국의 기온 자료를 수집했는데 미국 자료는 모두 화씨로 되어 있어 곤란한 상황이다. 이 자료는 미국의 국립환경정보센터(National Center for Environmental Information, https://www.ncei.noaa.gov)에서 받은 것이라고 한다. 매일 최고기온이 시간순으로 한 행에 하나씩 기록된 파일을 받았다. 파일의 앞부분을 보면 다음과 같다.

```
$ head -6 tempmax_fahr.txt
36
33
40
29
18
35
```

이 파일을 읽어서 모두 섭씨로 바꾸어 새로운 파일에 저장하여 친구에게 전달하는 것이 여러분이 할 일이다. 표준 입력으로 이 파일을 읽고 앞서 작성한 `fahr2cels`()를 이용하여 섭씨 온도로 바꾼 후 출력하는 프로그램을 작성하여라. 다음과 같이 사용할 것이다.

```
$ python3 tempconv.py < tempmax_fhar.txt > tempmax_cels.txt
```

참고로 마이크로소프트 PowerShell에서는 다음과 같이 해야한다.

```
PS> Get-Content tempmax_fhar.txt ¦ python tempconv.py > tempmax_cels.txt
```

파일을 파이썬에서 직접 읽어서 처리하는 것은 나중에 문자열을 자세히 다룰 때 소개할 것이다. 파일 입출력의 장점도 있지만 기본 입출력의 redirection을 이용한 방법의 장점도 있다. 명령행에서는 여러 도구들의 입출력을 연결하여 사용하는 일이 많으며 이를 위해서는 기본 입출력을 처리하는 도구가 유용할 수 있다.

차 례

2

프로그래밍의 기본 요소

2.1 자료형과 연산

자료형(data type)이란 컴퓨터에서 자료의 종류를 구분하여 다른 연산(operation)을 적용하기 위해 필요한 개념이다. 간단히 타입(type)이라고 부르는 경우도 많다. 파이썬에서는 수많은 자료형이 있을 수 있다. 무엇이든지 새로 추가할 수도 있다. 날짜나 시간 같은 것이 별도의 자료형으로 다룬다면 편리한 자료의 대표적인 예이다.

여기에서는 파이썬 프로그램을 구성하는 가장 기본적인 자료형과 연산자에 대해서 알아볼 것이다. 기본 자료형(basic type), 내장 자료형(built-in type), 원시 자료형(primitive type) 등 조금씩 다른 개념의 용어가 사용되며 프로그래밍 언어마다 구체적으로 어떤 자료형을 기본으로 보는지도 다르다. 파이썬에서는 원시 자료형이라는 개념은 사용하지 않는다. 여기에서는 파이썬에서 내장 자료형(built-in type)으로 제공되며 일반적으로 프로그래밍 언어들에서 공통적으로 기본적인 자료형으로 다루어지는 것들을 알아볼 것이다.

2.1.1 수치형 자료와 산술 연산

정수형과 실수형

파이썬에서는 정수(int)형과 실수(float)형을 구분한다. 복소수(complex)도 있지만 우리가 자주 사용할 것 같지는 않다. 우리에게는 그냥 같은 숫자로 보이지만 컴퓨터에게 '2'과 '2.0'은 완전히 다른 데이터이다. 후자를 편의상 '실수'라고 부르는 일이 많지만 엄밀하게 수학적인 의미에서 실수를 지칭하는 것은 아니다. 컴퓨터는 물리적 한계를 가지고 있기 때문에 모든 실수를 정확히 표현할 수는 없다. 파이썬에서는 가장 널리 사용되는 이배 정밀도 부동소숫점 형식(double-precision floating-point format)을 이용하여 실수를 표현한다. 이 이름에서 'float'라는 명칭이 나온 것이다.

연산자

컴퓨터 프로그래밍 언어에서 연산(operation)은 일종의 함수를 연산자(operator)라는 특수한 형태의 문법으로 호출하는 방식이라고 할 수 있다. 예를 들어, add(x, y)는 함수의 형태이고 x + y는 연산의 형태이다. neg(x)는 함수의 형태이고 -x는 연산의 형태이다. 피연산자가 하나만 있는 -x와 같은 경우를 1항 연산이라고 하고 피연산자가 둘 있는 x + y와 같은 경우를 2항 연산이라고 한다.

산술 연산자에는 덧셈(+), 뺄셈(-), 곱셈(*), 나눗셈(/), 제곱(**), 몫(//), 나머지(%) 등이 있다. 사용법을 다시 예시하면 다음과 같다.

```
>>> 12 + 5
17
>>> 12 - 5
7
>>> 12 * 5
60
>>> 12 / 5
2.4
>>> 12 // 5
2
>>> 12 % 5
2
>>> 2**10
1024
```

참고로 파이썬2에서는 12 / 5의 결과가 2로 나온다는 점에 주의해야 한다. 파이썬2에서는 정수 끼리의 연산에서는 결과가 무조건 정수로 나온다.

이 연산자들은 기본적으로는 산술 연산자로 사용되지만 연산자 좌우에 어떤 자료형이 오느냐에 따라 다른 의미를 가지기도 한다. 뒤에 나오는 문자형 자료에서 구체적인 예가 소개될 것이다.

정수

파이썬3에서는 정수를 한계없이 표현할 수 있다. 다음 명령은 컴퓨터 성능에 따라 매우 오래 걸릴 수도 있다는 점에 주의해서 실행하여야 한다. 30만 자리가 넘는 수가 출력될 것이다.

```
>>> 2 ** 1000000
```

다른 대부분의 프로그래밍 언어에서는 이렇게 자유로운 정수 표현 기능을 기본으로 제공하지 않는다. 파이썬은 정수를 다루는 데에 매우 강력한 기능을 기본으로 제공하는 드문 프로그래밍 언어이다. 나중에 다른 프로그래밍 언어를 배우게 된다면 정수를 사용할 때 파이썬과 다르다는 점에 주의해야 한다.

참고로 파이썬2는 정수 표현에 일부 제약이 있다. 파이썬2에서는 정수형을 `int`와 `long`으로 구별한다. 파이썬2로 프로그래밍을 해야 하는 경우에 `int`는 시스템에 따라 다르기는 하나 일정한 한계를 가지므로 큰 정수를 다룰 때에는 `long`으로 처리해야 한다는 점을 기억하자.

실수

수학에서의 실수라는 개념을 컴퓨터로 그대로 다루지는 못한다. 컴퓨터에서 사용하는 것은 소수의 형식을 빌어 표현하는 것이기 때문에 '부동 소숫점 수'라고 부르는 것이 정확하겠지만 편의상 실수라고 부르겠다. 실수와 정수는 완전히 다른 존재이다.

```
>>> 2
2
>>> 2.0
2.0
```

파이썬의 내장함수 중 `type()`을 이용하면 내부에서 어떻게 구별하고 있는지 확인할 수 있다.

```
>>> type(2)
<class 'int'>
>>> type(2.0)
<class 'float'>
```

여기에서 float는 64비트를 이용해 하나의 숫자를 표현하는 이배 정밀도 부동소숫점 형식이다. 1 비트는 음양 부호, 11비트는 거듭제곱, 53비트는 유효숫자를 표현하는 데에 사용된다. 십진수로 이야기하면

$$-5.342 \times 10^{-3}$$

과 같은 형식으로 숫자를 표현하고 음양 구분을 부하는 − 부분에 1비트, 5.342에 53비트, −3에 11비트를 사용하는 것이다. 이와 같은 숫자를 파이썬에 입력할 때에는 '-5.342e-3'라고 하면 된다. 이것을 흔히 과학적 표기(scientific notation)이라고 한다. 십진수로 표현했을 때 유효숫자는 15–17자리가 된다.

```
>>> 0.12345678901234567890123456789O
0.12345678901234568
>>> import math
>>> math.pi
3.141592653589793
>>> 2.0 / 3.0
0.6666666666666666
```

유효숫자의 개수만 제한이 있는 것이 아니라 숫자의 절대 크기에도 제한이 있다. 거듭제곱을 11비트를 사용하므로 그 이상의 수는 표현하지 못한다. 정수형을 이용할 때에는 2^{1024}를 정확하게 계산했지만 실수형은 이배 정밀도의 한계를 넘어서기 때문에 계산할 수 없다.

```
>>> 2.0**1023
8.98846567431158e+307
>>> 2.0**1024
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: (34, 'Result too large')
>>> 1e308
1e+308
>> 2e308
inf
```

영에 가까운 작은 수를 표현하는 데에도 제한이 있다.

```
>>> 5e-324
5e-324
>>> 4e-324
5e-324
>>> 3e-324
```

```
5e-324
>>> 2e-324
0.0
```

시스템의 한계를 알고 싶다면 다음과 같이 확인할 수 있다.

```
>>> import sys
>>> sys.float_info
sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313e-16,
radix=2, rounds=1)
```

복소수

파이썬은 기본적으로 복소수도 지원한다. 허수부는 `j`를 붙여 표현한다. 다음과 같다.

```
>>> 1 + 2j
(1+2j)
>>> 3 + 4j
(3+4j)
>>> 1 + 2j + 3 + 4j
(4+6j)
```

형 변환

파이썬에서는 `3`이라고 하면 정수로 처리되고 `3.0`이라고 하면 실수로 처리된다. 컴퓨터에게 둘은 완전히 다른 존재인데도 정수와 실수 사의 연산은 자연스럽게 이루어진다.

```
>>> 3 / 2.0
1.5
```

우리에게는 당연하고 자연스러워 보이지만 컴퓨터에는 그렇지 않다. 정수 3을 실수 2.0으로 직접 나눌 수는 없다. 파이썬은 우선 정수 3을 실수 3.0으로 바꾼 후에 나눗셈을 수행한다. 이렇게 자료형을 바꾸는 것을 형 변환(type conversion)이라고 한다.

많은 경우에 사람에게 자연스러운 방식으로 자동으로 형 변환을 해주지만 어떤 경우에는 프로그래머가 명시적으로 강제로 형 변환을 해야할 때도 있다. 이때에 타입의 이름에 해당하는 `int`, `float`를 사용한다.

```
>>> float(3)
3.0
>>> int(3.0)
```

```
3
>>> int(3.14)
3
```

이것은 무한을 표현하고 싶을 때에도 유용하다.

```
>>> float('inf')
inf
```

여기서 `inf`에는 따옴표를 쳐주어야 한다. 숫자가 아니라 문자로 입력한 것이기 때문이다.

2.1.2 논리형 자료와 논리 연산

불 자료형

논리형 자료은 참(`True`)과 거짓(`False`)라는 두가지 값을 가질 수 있으며 파이썬에서는 불(`bool`)이라고 부른다. 불 대수(Boolean algebra)의 체계를 따르는 자료형이라는 의미이다. 영국의 수학자이자 논리학자인 조지 불(George Boole)의 이름을 딴 것이다.

```
>>> True
True
>>> False
False
```

대문자로 `True`, `False`를 쓴다는 점에 주의하자.

논리 연산자

논리형 자료에 대한 연산에는 연접(conjunction)을 위한 `and`, 이접(disjunction)을 위한 `or`, 부정(negation)을 위한 `not` 연산자가 제공된다. 예를 들어, 다음과 같이 사용한다.

```
>>> True and False
False
>>> not False
True
```

이러한 and, or, not은 파이썬 내부적으로 의미가 이미 정해진 예약어이므로 사용자 정의 변수 등의 다른 용도로 사용할 수 없다. True와 False도 다른 용도로 사용할 수 없다.

연습 2.1 논리학에서는 연접, 이접, 부정을 각각 $\wedge$, $\vee$, $\neg$ 기호로 표현하는 경우가 많다. 명제 p는 참, q는 거짓일 때 다음의 진리값을 파이썬을 이용하여 계산해 보자.

1. $p \rightarrow q$
2. $p \wedge (\neg p \rightarrow q)$
3. $(p \wedge q) \vee (\neg p \wedge q)$

여기서 화살표($\rightarrow$)는 함의(implication) 또는 조건(conditional)을 나타내는 것으로 $p \rightarrow q$의 값은 $\neg p \vee q$의 값과 동치임을 이용해 구할 수 있다.

형 변환

불 자료형에 대한 형 변환은 개념적으로 별로 좋지 않은 경우도 있지만 가능하다. 숫자 값 0은 False에 해당하고 나머지 숫자는 True에 해당한다.

```
>>> bool(0)
False
>>> bool(1)
True
>>> bool(-3.0)
True
```

거꾸로 불 자료형을 강제로 숫자로 바꿀 때에 True는 1이 된다.

```
>>> int(True)
1
>>> float(True)
1.0
>>> int(False)
0
```

2.1.3 비교 연산

산술 연산, 논리 연산과 함께 기본적인 연산으로 비교 연산이 있다. 기본적으로는 수치형 자료의 크기를 비교하는 데에 사용된다. 같다(==), 다르다(!=), 크다(>), 크거나 같다(>=), 작다(<), 작거나 같다(<=)의 사용법을 예시하면 다음과 같다.

```
>>> 1 == 2
False
>>> 1 != 2
True
>>> 1 > 2
False
>>> 1 >= 2
False
>>> 1 < 2
True
>>> 1 <= 2
True
```

파이썬에서는 비교 연산자를 연쇄적으로 사용하는 것을 허락하고 있다. 즉, 다음과 같이 사용할 수 있다.

```
>>> 1 < 2 < 3
True
>>> 1 < 2 < 3 < 4 < 5
True
>>> 1 < 2 < 2 < 3 < 4
False
```

이 때 `x < y < z`의 의미는 `x < y and y < z`와 같다. 대부분의 다른 프로그래밍 언어에서는 파이썬과 같이 비교 연산자를 연쇄하는 것을 허락하지 않는다. 즉, 반드시 다음처럼 써주어야 한다.

```
>>> 1 < 2 and 2 < 3
True
```

연산의 순서

연산의 순서 혹은 연산 우선 순위(operator precedence)란 연산자가 여러 개 함께 있는 식에서 어떤 연산을 먼저할 것인가를 결정하는 규칙을 말한다. 초등학교 수학에서 배운 '곱셈은 먼저하고 덧셋은 나중에 한다', '곱셈이 여러 개 있으면 앞(왼쪽)에 있는 것부터 먼저 계산한다'와 같은 규칙을 말한다.

연산의 순서를 임의로 바꾸고 싶을 때는 수학에서처럼 괄호를 이용한다. 단 파이썬에서는 원괄호만 이용한다.

```
>>> (4 + 1) * 3
15
```

파이썬은 대부분의 프로그래밍 언어들이 따르고 있는 연산 우선 순위 규칙을 동일하게 따르고 있다. 산술 연산은 수학 시간에 배운 것과 똑같은 순서로 계산한다. 산술 연산자 이외에 다른 연산자들에 대해서도 계산 순서에 대한 규칙이 존재한다. 여러 연산자가 섞여있을 때 산술 연산자, 비교 연산자, 논리 연산자 순서로 계산된다. 다음 식을 생각해 보자.

```
>>> 4 - 1 == 3 and 4 + 1 == 5
True
```

괄호를 명시적으로 사용하면 다음과 같다.

```
>>> ((4 - 1) == 3) and ((4 + 1) == 5)
True
```

불필요한 괄호를 지나치게 많이 사용하는 것은 코드를 읽기 어렵게 만들고 오류의 원인이 될 수 있으니 피하는 것이 좋다. 뒤에서 배울 변수를 적절히 이용하면 괄호를 잔뜩 지치 않고 깔끔한 코드를 사용할 수 있다.

동일성

동일성 판정을 위한 `is`와 `is not` 연산자도 있다. 값이 값다(`==`), 다르다(`!=`), 즉 등가를 판별해 주는 연산과 다르다. 수치형 `int`와 `float`, 문자열 `str`, 논리형 `bool`의 경우에는 `==` 연산과 `is` 연산의 결과가 항상 동일하다.

```
>>> a = 3
>>> b = 3
>>> a == b
True
>>> a is b
True
```

그러나 일반적으로 등가성 비교와 동일성 비교에 차이가 있다. 다음의 리스트 자료형의 예를 보자. 다음 `a`와 `b`는 등가이지만 동일한 것은 아니다. 비유하자면 쌍동이가 똑같이 생겼더라도 동일한 사람은 아니라는 것이다.

```
>>> a = [0, 1, 2]
>>> b = [0, 1, 2]
>>> a == b
True
>>> a is b
False
```

연습 2.2 다음 식의 참과 거짓을 판단하여 보자.

1. $7 + 4 \times 3 \geq 20$
2. $\pi > 3$
3. $2^{100} > 10^{30}$

참고로 파이썬에서 원주율은 `math.pi`로 얻을 수 있다. 우선 `math` 모듈을 `import` 해야 한다.

#+BEGINexercise 명제 p, q, r이 다음과 같을 때

$$\begin{aligned} p: &\quad 7 + 4 \times 3 \geq 20 \\ q: &\quad \pi > 3 \\ r: &\quad 10^{30} < 2^{100} < 10^{31} \end{aligned}$$

다음 식의 진리값을 계산하여라.

$$(p \to q) \wedge (q \to r)$$

#+ENDexercis

2.1.4 문자형 자료

문자열

프로그래밍에서 문자(character)가 나열된 형태를 문자열(string)이라고 부른다. 파이썬에서는 문자열 자료를 위해 `str`를 제공하고 있다. 파이썬에서 문자열은 작은따옴

표(') 또는 큰따옴표(")로 둘러싸서 표현한다. 수치형과 문자형의 구분에 주의하자. 다음 예에서 따옴표 안에 있으면 문자열이고 그렇지 않으면 숫자이다.

```
>>> "3"
'3'
>>> 3
3
```

문자열 연산

어떤 자료형인가에 따라 연산도 달라진다. 문자열끼리 더하는 것과 숫자끼리 더하는 것은 다른 결과를 가져온다.

```
>>> 3 + 3
6
>>> "3" + "3"
'33'
```

두 문자열을 더하면 두 문자열이 연결된 새로운 문자열이 만들어진다. 더하기(+) 연산자가 좌우에 어떤 자료형이 오느냐에 따라 다르게 작동한다. 이런 작동 방식을 연산자 오버로드(operator overloading)이라고 한다.

곱셈도 문자열이 개입할 경우 그 의미가 달라진다. 문자열과 정수를 곱하면 지정된 수 만큼 문자열이 반복되어 이어진 새로운 문자열이 만들어진다.

```
>>> "abc" * 10
'abcabcabcabcabcabcabcabcabcabc'
>>> 10 * "abc"
'abcabcabcabcabcabcabcabcabcabc'
```

자료형에 따라 연산에 제한이 있다. 문자열과 숫자를 더할 수는 없다. 문자열과 문자열을 곱할 수도 없다. 이러한 경우 오류가 발생한다.

```
>>> "3" + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> "a" * "b"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

오류가 발생하면 오류 메시지가 출력된다. 오류 메시지를 잘 읽는 것은 큰 프로그램을 작성하기 위해 필수적인 요건이다. 대화형 모드에서 연습하면서 오류 메시지를 읽고 어떤 의미인지 이해하는 연습을 해두는 것이 좋다.

형 변환

데이터를 처리하다 보면 문자열과 숫자 사이의 변환이 필요할 때가 많다. 문자열 자료를 숫자로 바꿀 때에는 int()나 float(), 숫자 또는 다른 자료형을 문자열로 바꿀 때에는 str()을 이용한다.

가장 흔한 경우가 문자열로 된 것을 숫자로 바꾸어야 하는 경우이다.

```
>>> a = "3"
>>> a
'3'
>>> int(a)
3
```

이 방법을 이용하면 앞에서 본 에러를 피할 수 있다.

```
>>> a + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> int(a) + 3
6
```

숫자를 문자열로 바꾸어야 하는 경우도 종종 발생한다. 문자와 숫자를 합쳐 새로운 문자열을 만들려고 하는 경우가 있다.

```
>>> year = 1987
>>> "year" + str(year)
'year1987'
```

2.2 변수와 할당

변수는 어떤 데이터에 이름을 붙인 것이다. 어떤 프로그래밍 언어에서 내부적으로 그 값이 지정되어 있는 변수도 있는데 이러한 변수를 내장 변수(built-in variable)이라고 한다. 이미 지정되어 있는 변수 이름을 사용하면 문제가 될 수 있으므로 주의해야 한다. 특수 목적의 프로그래밍 언어의 경우에는 기본적으로 다수의 내장 변수가 지정하기도 하나 파이썬과 같은 범용 프로그래밍 언어에서 내장 변수를 찾아보기는 어렵다. 변수가 필요할 때 프로그래머가 새로운 이름을 붙여 사용하게 되는데 이것을 사용자 정의 변수(user-defined variable)라고 한다.

2.2.1 변수의 이름

변수명에는 영문자, 숫자, 밑줄(_)을 사용할 수 있다. 영문자의 대문자와 소문자는 구별한다. 변수명의 첫글자가 숫자이어서는 안 된다. 파이썬3에서는 한글을 포함하여 유니코드(Unicode)의 많은 문자들을 변수명에 사용할 수 있지만 일반적으로 권장되는 관습은 아니다.

변수명은 기본 제약 아래에서 마음대로 붙일 수 있으나 아무렇게 붙이는 것은 좋은 프로그램을 작성하는 데에 걸림돌이 될 수 있다. 이름 붙이기에 관련된 여러가지 관습이 존재하는데 변수의 첫글자는 영문자 소문자로 하는 것이 일반적이다. 긴 변수명을 쓰기 위한 오래된 방법의 하나는 단어를 밑줄로 연결하는 것이다.

```
>>> weight = 55.0
>>> height = 160.0
>>> body_mass_index = weight / (height/100)**2
```

객체지향 프로그래밍에서는 낙타 대문자(camel case)라고 하여 단어들을 붙여쓰되 단어의 시작을 대문자로 하여 단어 단위가 구별되도록 쓰는 관습을 따르는 경우도 많다. 이에 따르면 `bodyMassIndex`이 될 것이다.

변수 이름 이외에도 여러가지 스타일 지침이 존재한다. 반드시 지켜야 하는 스타일이 존재하는 것은 아니다. 어떤 특정한 스타일이 항상 옳은 것도 아니다. 프로그래밍 언어마다 관습의 차이가 있기도 하다. 파이썬의 경우 PEP (Python Enchancement Proposals)에서 제안된 파이썬 스타일 지침이 있으니 한번 참조해 보면 좋을 것이다.

- `https://www.python.org/dev/peps/pep-0008/`

2.2.2 할당 연산

변수에 어떤 값을 넣을 때 등호(=)를 사용한다. 등호의 왼쪽에 변수를 쓰고 오른쪽에는 값을 쓴다. 이것을 할당(assignment)라고 한다. 이때 사용되는 등호의 의미를 등가(equivalence)와 혼동하지 않도록 주의해야 한다. 수학에서는 $x = y$가 의미하는 바가 x와 y가 같은 값을 가진다는 의미이다. 하지만 파이썬을 비롯한 대부분의 프로그래밍 언어에서 '`x = y`'의 의미는 `y`라는 곳에 저장된 값을 `x`라는 곳에 할당하라는 의미이다.

할당 연산자의 왼쪽에는 변수, 오른쪽에는 값을 쓴다는 점을 다시 한번 강조한다. 다음 할당문에서 변수 x에 '3 + 4'라는 식이 기억되는 것이 아니라 그 식의 연산 결과 얻어진 최종값인 7이 기억된다.

```
>>> x = 3 + 4
```

다음과 같이 오른쪽의 식에 변수가 포함된 경우도 마찬가지이다.

```
>>> x = 3
>>> y = 4
>>> z = x + y
```

여기에서 z에 x + y라는 식을 할당하는 것이 아니라 이 식을 계산한 결과인 7이라는 값을 할당하는 것이다. 할당 연산자의 오른쪽에 있는 식이 우선 모두 계산이 된 후에 최종값이 왼쪽 변수에 할당이 된다.

증강 할당

파이썬에는 =이외에 +=, -=, *=, /= 등의 추가적인 할당 연산자가 존재한다. 이러한 할당 연산을 증강 할당(augmented assignment) 또는 복합 할당(compound assignment)이라고 한다.

할당 연산이 왼쪽에는 변수, 오른쪽에는 값을 취한다는 기본 원리를 염두에 두고 다음 할당문이 어떤 결과를 가져오는지 생각해 보자.

```
>>> x = 3
>>> x = x + 1
```

여기서 x = x + 1이 실행될 때 우선 오른쪽의 x + 1의 값이 계산되어야 한다. x의 값은 3이므로 x + 1은 4가 된다. 따라서 최종적으로 x = 4라는 의미가 된다. 즉 이제 x의 값은 4가 된다. 마치 x가 1이 증가하는 것과 같은 효과를 얻게 된다. 왼쪽과 오른쪽에 동일한 변수를 사용했기 때문에 좀 혼동스러울 수도 있지만 할당문의 작동 방식을 생각해 보면 당연한 결과이다.

이렇게 할당문 양쪽에 동일한 변수를 사용하면서 어떤 값을 더하거나 빼거나 하는 등의 단순 연산이 필요할 때 증강 할당 연산자를 이용하면 편리하다. 즉 x = x + 1 대신에 다음과 같이 할당문을 작성하는 것이다.

```
>>> x += 1
```

이것은 x를 1만큼 증가시키라는 의미로 이해할 수 있다. 변수 이름을 한번만 써도 된다는 점에서 편리하며 그 의미가 직관적으로 받아들여진다는 점에서 인지적으로도 더 편하다.

2.2.3 참조

파이썬에서는 변수(variable)보다는 참조(reference), 값(value)보다는 객체(object)라는 용어를 써서 이야기하는 것이 더 정확할 수 있다. 여기서는 객체는 단순한 값이 아니라 무엇인가 많은 정보가 들어있는 덩어리라는 정도로만 생각해 보자.

할당 연산자의 오른쪽에 있는 것이 값이라는 것에 대해 좀더 이야기해 보자. 방금 이야기했듯이 'y = x'라는 할당문에서 우선 x가 가리키는 값인 3으로 바꾼 후에 y = 3이 실행되는 것이다.

```
>>> x = 3
>>> y = x
>>> y
3
```

이 때 y는 x와는 다른 변수이고 둘이 가리키는 값이 같게 될 뿐이다. 이제 y의 값을 바꾸어 보자.

```
>>> y = 4
>>> x
3
>>> y
4
```

그다지 놀라울 것 없이 x의 값은 그대로 3이고 y의 값은 4로 바뀌었다.

이번에는 리스트의 경우를 살펴보자. 등가성과 동일성의 차이가 기억나는가?

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a == b
True
>>> a is b
False
```

여기서 a와 b는 값은 같지만 서로 다른 데이터이다. 똑같이 생겼지만 다른 존재인 쌍동이로 비유할 수 있다고 했다. 이제 다음 경우를 보자.

```
>>> c = [1, 2, 3]
>>> d = c
>>> c == d
True
>>> c is d
True
```

여기서 c와 d는 동일한 데이터를 가리키는 변수이다. 비유하자면 한 사람이 두 개의 이름을 가지고 있는 것이다. 두 변수 c와 d가 동일한 데이터를 가리키고 있기 때문에 다음과 같은 일이 벌어진다.

```
>>> c[1] = 4
>>> c
[1, 4, 3]
>>> d
[1, 4, 3]
```

두 변수가 동일한 데이터를 가리키기 때문에 c의 값을 바꾸면 d의 값도 바뀐다.

마치 정수와 리스트에 다른 원리가 적용되는 것처럼 이야기했지만 그렇지 않다. 위의 예제에서 정수는 그 값을 통째로 바꾸었고 리스트는 그 일부인 원소 하나만 값을 바꾸었다. 리스트도 그 값을 통째로 바꾼다면 마찬가지이다.

```
>>> c = [1, 2, 3]
>>> d = c
>>> d = [1, 4, 3]
>>> c
[1, 2, 3]
>>> d
[1, 4, 3]
```

이런 일이 일어나는 이유는 정수는 변할 수 없는 자료이고 리스트는 변할 수 있는 자료이기 때문에 그렇다. 예를 들어 정수 '3'이라는 자료가 주어졌을 때 그 내용을 바꾼다는 생각을 아예 할 수 없다. 하지만 [1, 2, 3]이라는 리스트가 주어졌을 때 그 내용을 바꾸어 [1, 4, 3]으로 만들 수 있다. 이러한 차이를 구별할 때 불변(immutable), 가변(mutable)이라는 용어를 사용한다.

2.3 함수

프로그래밍에서 함수(function)를 수학에서 함수와 형식적으로 동일한 것으로 엄격하게 바라보는 입장도 있지만 일반적으로는 좀 차이가 있다. 사실 '함수'보다는 어떤

1.3 함수

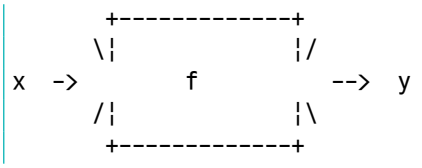
프로그래밍에서 함수(function)를 수학에서 함수와 형식적으로 동일한 것으로 엄격하게 바라보는 입장도 있지만 일반적으로는 좀 차이가 있다. 사실 '함수'보다는 어떤 '기능'이라고 하는 것이 더 가까울 수 있다. 고전적인 프로그래밍에서는 서브루틴(subroutine)이나 프로시저(procedure)라고 부르기도 하고 객체지향 프로그래밍에서는 메소드(method)라고 부르는 일도 있다. 사실 용어만 다르게 부르는 것이 아니라 조금씩 개념의 차이도 있다.

1.3.1 함수의 기본 형식

여기서 우리는 순수한 함수(pure function)에 집중하여 이야기하려고 한다. 순수한 함수는 수학에서 함수에 대응되는 것이다. 예를 들면, 수식으로

$$y = f(x) = x + 1$$

와 같은 형식으로 표현되는 것이다. 이런 함수는 다음과 같은 그림으로 표현할 수도 있다.



어떤 값을 입력으로 받아서 어떤 계산을 수행한 후에 어떤 값을 내보내는 기계 상자로서 함수를 생각할 수 있다. 파이썬에서 이 함수는 다음과 같이 작성한다.

```
def f(x) :
    return x + 1
```

함수를 작성하기 위해서는 다음 요소들이 필요하다.

- 함수의 이름: 새로운 함수를 정의하기 위해서 def라는 키워드로 시작을 하여 함수의 이름을 지정한다.

- 함수의 인자: 함수의 입력 부분에 해당하는 요소를 인자 또는 매개변수(parameter)라고 한다. 영어에서 'argument'와 'parameter'라는 용어를 구별하기도 하는데 이때 'argument'는 '인수'라고 번역하자는 견해도 있으며 현재 용어의 사용은 혼란스러운 상황이다. 함수를 정의할 때 사용하는 변수는 'parameter', 함수를 사용할 때 값으로 넣어주는 것은 'argument'로 구별하는 것이 일반적이다. 함수 이름 바로 뒤의 괄호 안에 인자의 이름을 쉼표로 구분하여 나열한다.
- 함수의 반환값: 함수의 출력 부분에 해당하는 요소를 말한다. 흔히 '리턴값'이라고 한다. `return`이라는 키워드를 이용하여 되돌려 줄 값을 지정한다.
- 함수의 본체: 함수가 하는 일은 `def` 보다 한 단계 들여쓰기를 하여 작성한다. 이것을 함수의 본체라고 한다.

예제 1.1

파이썬으로 함수를 구현해 보자.

$$f(n) = \frac{1}{1+\frac{1}{1+\frac{1}{1+\frac{1}{1+n}}}}$$

와 같은 함수가 있다면 $f(3)$, $f(4)$ 등을 편리하게 구할 수 있을 것이다. 다음과 같이 새로운 함수를 정의할 수 있다.

```
def f(n):
    ans = 1 / (1 + 1 / (1 + 1 /(1 + 1/(1 + n))))
    return ans
```

이때 `def`, 쌍점(`:`), `return`은 형식적으로 고정된 틀이다. 함수 이름 `f`, 함수 인자 `n`, 함수 값은 마음대로 바꿀 수 있다. 이제 다음처럼 함수를 사용할 수 있다.

```
>>> f(3)
0.6428571428571428
>>> f(4)
0.6470588235294118
```

1.3.2 반환값이 없는 함수

프로그래밍 언어마다 용어 사용에 차이가 있지만 반환값, 즉 결과값 또는 리턴값이 있는 경우를 함수, 반환값이 없는 경우를 서브루틴으로 구별하기도 한다. 반환값이 없는 경우는 엄밀하게 말해 함수가 아니라는 것이다. 파이썬의 함수는 엄격한 의미의 함수이지만 동시에 파이썬에서는 `return` 문이 없는 함수를 정의하는 것을 허락한다. 간단한 예를 들면 다음과 같다.

```
def hello(name):
    print("Hello, " + name + "!")
```

이것이 가능한 이유는 정확히 말해 `return` 문이 없는 것이지 반환값이 없는 것은 아니기 때문이다. 형식적으로 리턴값이 존재하는 엄격한 함수이다. 이러한 함수를 사용할 때 리턴값은 `None`이라는 특별한 값이 된다. `None`은 값이 없음을 표현하기 위한 특별한 값이다.

1.3.3 지역 변수와 전역 변수

함수를 작성할 때 변수를 지역(local)과 전역(global)이라는 개념으로 구별하여 이해하는 것이 도움이 될 수 있다. 말 그대로 '전역'은 어떤 변수가 소스 코드 전체에 걸쳐 사용된다는 것이고 '지역'은 특정한 일부분에서만 사용된다는 것이다. 정확한 이해를 위해서는 변수 영역(scope), 이름공간(namespace) 등의 개념도 함께 이해할 필요가 있다.

여기에서는 함수 내부에 존재하는 변수와 함수 외부에 존재하는 변수라는 차원에서만 이야기해 보자. 다음과 같이 x라는 이름의 변수가 함수 밖에도 있고 함수 안에도 있는 경우를 생각해 보자.

```
x = 3

def f(x) :
    return x

print(f(4))
print(x)
```

출력 명령을 통해서 각각 4와 3이 얻어지는 것을 확인할 수 있다. 즉, f(4)를 실행한다고 하더라고 함수 밖에 있는 x 변수의 값이 4로 바뀌지 않는다. 이렇게 x 변수는 함수 내부에만 존재하고 함수 밖에는 영향을 미치지 않는다. 이런 것을 지역 변수라고 한다. 이렇게 변수가 유효한 영역이 제한되어 있기 때문에 우리는 함수 내부에서 변수명을 자유롭고 안전하게 사용할 수 있다.

조금 다른 경우를 생각해 보자. 이번에는 f 함수의 인자로 y라는 이름의 변수를 제공하지도 않고 함수 내부에서 새로 만들지도 않으면서 y라는 변수를 사용했다.

```
x = 3
y = 10

def f(x) :
    return x + y

print(f(4))
```

여기에서 14라는 결과가 출력되는 것을 확인해 보자. 즉, 함수 외부에 있는 변수 y가 사용된 것이다. 여기서 y는 전역 변수로 작동하고 있는 것이다. 일반적으로 이렇게 전역 변수를 사용하는 것은 기대하지 못한 오류를 일으킬 수 있는 위험성을 가지고 있기 때문에 좋지 않은 습관으로 여겨진다. 물론 이런 것을 허락하는 이유는 경우에 따라 전역 변수가 쓸모가 있기 때문이다. 하지만 우리는 절대로 전역 변수를 사용하지 않을 것이다. 좀더 정확히 말해 함수를 작성할 때 함수 밖에 있는 변수를 절대로 사용하지 않을 것이다.

1.3.4 내장 함수

함수도 크게 내장형 함수와 사용자 정의 함수로 구분할 수 있다. 범용 프로그래밍 언어에서 내장 변수를 미리 정의해 놓는 경우는 드물지만 내장 함수는 종종 제공된다. 정말 기본적이고 필수적이라고 생각되는 함수들은 기본 내장 함수로 제공되고 그 이외의 널리 이용되는 함수들은 표준 라이브러리로 제공하는 것이 일반적이다.

파이썬에서도 아무런 추가적인 장치없이 사용할 수 있는 함수들을 수십여개 제공하고 있다. 새로운 변수나 사용자 정의 함수를 만들 때 내장 함수와 동일한 이름을 사용하면 혼동이 생길 여지가 있으니 주의해야한다. 파이썬의 기본적인 내장 함수의 목록은 파이썬 공식 문서에서 찾아볼 수 있다.

- http://docs.python.org/library/functions.html

우선 내장 함수에는 주어진 데이터 x의 자료형이 무엇인지 알려주는 type(x)를 비롯하여 해당 자료형을 만들거나 강제변환하는 함수로 bool(), int(), float(), complex(), str(), tuple(), list(), set(), dict() 가 있다. 문자와 관련해서는 아스키 코드 o와 아스키 문자 c 사이의 변환를 해주는 chr(o), ord(c)가 있으며 숫자 x에 대해 진수변환을 해주는 bin(x), oct(x), hex(x)와 절대값을 계산해주는 abs(x)가 있다. 숫자들의 목록 또는 집합 S가 주어졌을 때 합 sum(S), 최소값 min(S), 최대값 max(S)를 구할 수도 있다. 입출력을 위한 print(), input()와 파일을 열기 위한 open()도 유용하게 사용되는 내장 함수이다. 대화형 모드에서 무엇인가에 대해 도움말을 보고 싶다면 help()를 이용해 보자.

예제 1.2

몸무게(kg)과 키(cm)를 넣으면 신체질량지수(BMI, Body Mass Index)를 계산해주는 함수를 작성해보자. 신체질량지수를 구하는 공식은 다음과 같다.

$$\text{BMI} = \frac{\text{weight (kg)}}{(\text{height (m)})^2}$$

키는 cm로 입력 받고 몸무게는 kg으로 입력받는다고 가정하고 작성해 보자. 파이썬에서 함수로 작성하고 키 160cm, 몸무게 55kg인 경우를 계산하여 출력하는 프로그램은 다음과 같다.

```
# bmi.py
def bmi(height, weight) :
    bmi = weight / (height/100)**2
    return bmi

print(bmi(160, 55))
```

위 프로그램을 실행시켜보자. 다음과 같이 BMI 값이 출력되는지 확인해보자.

```
$ python bmi.py
21.484374999999996
```

연습 1.4 위에서 작성한 프로그램은 키 160cm, 몸무게 55kg인 경우만 계산해 준다. 다른 경우를 계산하기 위해서는 프로그램 자체를 수정해야 하는 불편함이 있다. `input()`으로 키와 몸무게를 입력받아 BMI를 계산해 주는 프로그램을 작성하여라.

1.4 조건문

조건문은 조건에 따라 다르게 작동을 하도록 해주는 구문이다. 조건문 어떤 조건식의 참거짓을 평가하여 그 결과에 따라 다르게 작동하도록 해 준다. 파이썬에서 조건문을 작성할 때에는 `if`, `elif`, `else`를 사용한다.

if문의 구조

간단한 예를 통해 파이썬 조건문의 형식을 살펴보자. 다음과 같이 주어진 실수값의 부호를 알려주는 함수를 생각해 보자.

$$\text{sign}(x) = \begin{cases} 1, & x > 0 \\ 0, & x = 0 \\ -1, & x < 0 \end{cases}$$

파이썬으로 위 함수를 그대로 옮겨 쓰면 다음과 같다.

```
def sign(x) :
    if x > 0 :  val = 1
    if x == 0 : val = 0
    if x < 0 :  val = -1
    return val
```

이 함수는 우리가 기대하는 대로 작동하고 특별한 문제를 일으키지 않는다. 하지만 필요없는 계산을 하고 있다. 세 개의 조건이 서로 배타적이므로 하나를 만족하면 나머지 조건은 따져볼 필요가 없다. 예를 들어, x가 1인 경우 첫번째 조건문을 만족하므로 나머지 두 조건문은 굳이 실행할 필요가 없다. 하지만 위와 같이 작성하면 항상 세 조건식을 모두 계산하게 된다.

파이썬의 `elif`는 바로 앞선 조건식을 만족하지 않는 경우에만 실행되도록 하여 쓸모없는 계산을 피할 수 있게 해준다. 다음과 같이 고칠 수 있다.

```
def sign(x) :
    if x > 0 :    val = 1
    elif x == 0 : val = 0
    elif x < 0 :  val = -1
    return val
```

이제 x가 1인 경우에 첫번째 조건을 만족하므로 나머지 두 elif 부분은 계산하지 않고 지나간다. 이것으로 훌륭하지만 아직 고칠 부분이 있다. 우리는 세 조건이 서로 배타적이면서 동시 모든 가능한 경우를 나타낸다는 사실을 알고 있다.[1] 따라서 처음 두 조건이 거짓이면 세번째 조건은 무조건 참이 된다. 이런 경우 세번째 조건식을 계산하는 것은 낭비이다. 다음과 같이 else를 이용하면 이러한 쓸모없는 계산을 피할 수 있다.

```
def sign(x) :
    if x > 0 :    val = 1
    elif x == 0 : val = 0
    else :        val = -1
    return val
```

연습 1.5 신체질량지수(BMI) 값은 그 범위에 따라 저체중, 정상체중, 과체중을 판정하는 데에 사용된다. 일반적으로 18.5이하는 저체중, 25이상은 과체중, 두 값 사이에 속하면 정상체중으로 판정한다. BMI값을 입력으로 받아 판정결과를 'Underweight', 'Normal', 'Overweight'로 출력해주는함수 `bmicat()`를 작성하여라.

연습 1.6 지금까지 작성한 코드를 모두 종합하여 키와 몸무게를 입력 받아 BMI와 저체중/과체중 판정여부를 출력해 주는 프로그램을 작성하여라.

1.5 반복문

반복문은 흔히 루프(loop)라고 한다. 완벽하게 동일한 동작을 단순 반복할 수도 있지만 대개는 조금씩 다른 동작을 반복하기 위해 사용한다. 어떤 대상 집단의 각 원소들에 동일한 동작을 적용하려고 할 때에 사용하는 경우와 단계적으로 일정한 규칙에 따라 진행해 나가는 경우가 많다.

[1] 수학에서는 이러한 것을 분할(partition)이라고 한다. 부분 집합들이 서로 배타적이면서 부분 집합들의 합집합이 전체 집합이 될 때를 말한다.

1.5 반복문

반복문은 흔히 루프(loop)라고 한다. 완벽하게 동일한 동작을 단순 반복할 수도 있지만 대개는 조금씩 다른 동작을 반복하기 위해 사용한다. 어떤 대상 집단의 각 원소들에 동일한 동작을 적용하려고 할 때에 사용하는 경우와 단계적으로 일정한 규칙에 따라 진행해 나가는 경우가 많다.

파이썬에서는 반복을 위해 `while`문과 `for`문을 제공하고 있다. 일반적으로 대상 집합이 정해진 경우나 반복 횟수를 미리 알 수 있는 경우에 `for`문을 사용하며 끝나는 조건은 알고 있지만 언제 끝나는지는 미리 알 수 없을 때 `while`문을 많이 사용한다.

for

파이썬의 `for`문은 항상 `in`과 함께 짝을 이루어 사용된다. 형식은 다음과 같다.

```
items = ['apple', 'banana', 'carrot']

for item in items:
    print(item)
```

위의 예에서 `in` 다음에 리스트를 사용했다. 이렇게 `in` 다음에는 어떤 목록이나 집합이 와야 한다. 전문 용어로 말하면 나열(sequence)형 또는 반복가능(iterable)형 자료가 와야 한다. 문자열, 튜플, 리스트 같은 것이 대표적인 나열형 자료이다.

range()

프로그램을 작성하다 보면 숫자를 나열하고 싶을 때가 있다. C, C++, Java와 같은 프로그래밍 언어에서는 i를 1부터 100까지 2씩 증가시키면서 무엇을 하라는 형태로 반복문을 작성한다. 파이썬에는 이런 형태의 문법을 제공하지 않는다. 대신 파이썬에서는 `range()` 함수를 이용한다.

```
>>> for i in range(0, 5):
...         print i
...
0
1
2
3
4
```

범위를 지정하기 위해 사용되는 range(i, j) 함수는 두 개의 정수값을 받아 i에서 $j-1$까지의 정수의 목록을 표현한다. 마지막 수가 포함되지 않는다는 점에 주의하자. 더 간단하게 range(n)라고 하면 0에서 시작해서 n개의 정수의 목록을 제공해 준다.

enumerate()

파이썬에서는 나열형 자료에 대해 for in 구문을 쓰는 것이 일반적이다. 다음과 같이 인덱스를 루프로 돌리는 것은 파이썬답지 않은 방식이다.

```
items = ['apple', 'banana', 'carrot']

for i in range(0, 3):
    print(items[i])
```

하지만 때로는 나열형 자료의 번호가 필요할 때가 있다. 리스트와 함께 원소들의 번호가 필요할 때 enumerate()를 쓸 수 있다.

```
for i, item in enumerate(items):
    print(i, item)
```

Aggregation

반복문을 개별 항목에 대한 어떤 작업을 하는 목적으로 사용되는 경우도 있지만 반복문을 통해 여러 개의 원소를 하나로 통합하는 목적으로 사용되는 경우도 있다. 전형적으로 반복문과 증강 연산이 함께 사용하는 경우이다. 예를 들어, 리스트의 원소들의 합을 다음과 같이 구할 수 있다.

```
counts = [2, 5, 9, 3, 2, 7, 8]

total = 0
for x in counts:
   total += x

print(total)
```

물론 파이썬에는 sum()과 같은 함수가 있기 때문에 위 코드를 사용할 필요는 없다. 그러나 여러가지 다양한 형태의 집계를 위해서 위와 같은 논리가 사용될 수 있다.

break와 continue

반복문이 실행되는 도중에 중단하기 위해서 break, 다음 단계로 건너뛰기 위해서 continue를 사용할 수 있다.

예를 들어 1부터 100사이의 수 중에 3 또는 4의 배수는 제외한 나머지 숫자만 출력하는 프로그램을 생각해 보자.

```
for n in range(1, 101):
    if n % 3 == 0 or n % 4 == 0:
        continue

    print(n)
```

물론 이 경우에 위의 예처럼 continue를 사용하는 것보다 다음처럼 하는 것이 더 깔끔할 수도 있다.

```
for n in range(1, 101):
    if n % 3 != 0 and n % 4 != 0:
        print(n)
```

대부분의 경우 continue를 사용하지 않고도 동일하게 작동하는 코드를 작성할 수 있다. 하지만 continue를 사용하는 것이 더 보기 좋은 경우도 있다. 위 문제에서 단순히 출력하는 것이 아니라 무엇인가 복잡한 일을 해야한다고 해보자. 다음과 같이 될 것이다.

```
for n in range(1, 101):
    if n % 3 == 0 or n % 4 == 0:
        continue

    #
    # 여기에 긴 코드 작성
    #
```

다른 버전으로 continue를 사용하지 않는 경우는 다음과 같다.

```
for n in range(1, 101):
    if n % 3 != 0 and n % 4 != 0:
        #
        # 여기에 긴 코드 작성
        #
```

긴 코드를 모두 두 단계 들여쓰기를 해야 하고 코드를 이해하는 데에도 인지적 부담이 더 크다.

조금 다른 예로 1부터 100사이의 수 중에 12로도 나누어지고 8로도 나누어지는 것 중 가장 작은 수를 구하는 문제를 생각해 보자. 최소공배수이다. 가장 작은 수만 구하면 되므로 조건에 부합하는 순간 반복문을 종료하는 것이 효율적일 것이다.

```
for n in range(1, 101):
    if n % 12 == 0 and n % 8 == 0 :
        break

print(n)
```

중첩된 반복문

반복문은 중첩하여 사용할 수 있다. 중첩된 반복문은 프로그래밍에서 가장 흔히 사용되는 구문 중의 하나이다. 두 겹, 세 겹 얼마든지 겹쳐질 수 있다. 모든 가능한 순서쌍이나 조합을 다룰 때 기본적으로 사용되며 수많은 알고리즘들이 반복문의 중첩 형태로 이루어져 있다.

간단히 구구단 표를 출력하는 프로그램을 생각해보자. 1에서 9까지의 숫자들의 모든 가능한 순서쌍과 이 둘을 곱한 결과를 출력하는 프로그램을 작성해 보자.

```
for n in range(1, 10):
    for m in range(1, 10):
        print(n, '*', m, '=', n*m)
```

서로 독립적인 두 개의 반복문이 중첩되어 있는 형태이다. 외부와 내부의 반복문을 자리를 바꾸어도 동일하게 작동한다.

연습 1.7 다음 식의 값을 파이썬을 이용하여 계산하여 보아라.

$$\sum_{n=1}^{10}\sum_{m=1}^{10}\frac{n}{m} = \left(\frac{1}{1}+\frac{1}{2}+\cdots+\frac{1}{10}\right)+\cdots+\left(\frac{10}{1}+\frac{10}{2}+\cdots+\frac{10}{10}\right)$$

조금 다르게 1에서 9까지의 숫자가 쓰여 있는 카드 중 2장을 고르는 모든 방법을 나열하는 프로그램을 생각해 보자. 모든 조합을 찾는 것이다. 1, 9와 9, 1은 같은 것이므로 하나만 출력하도록 하자.

```
for n in range(1, 10):
    for m in range(n, 10):
        print(n, m)
```

이렇게 내부의 반복문이 외부의 반복문에 종속적인 형태로 작성될 수 있다. 이 예에서 보듯이 반복의 범위를 꼼꼼하게 따져 보아야 한다. 이러한 반복문은 기계적으로 내부와 외부 반복문의 자리를 바꿀 수 없다는 점에 주의해야 한다.

연습 1.8 다음 식의 값을 파이썬을 이용하여 계산하여 보아라.

$$\sum_{n=1}^{10}\sum_{m=1}^{n}\frac{n}{m} = \left(\frac{1}{1}\right) + \left(\frac{2}{1}+\frac{2}{2}\right) + \cdots + \left(\frac{10}{1}+\frac{10}{2}+\cdots+\frac{10}{10}\right)$$

while

앞선 본 break에서와 같이 반복을 하다가 어떤 조건에 도달하는 경우 중단하는 코드를 작성하는 경우가 많다. while문은 이에 대응되는 구문이라 할 수 있다. 조건과 행위를 제시하는 형태로 작성된다. 조건을 만족하는 동안 주어진 행위를 반복하고 조건을 만족하지 않는 순간 반복문을 빠져나간다. 이 때문에 조건 루프(conditional loop)라고도 불린다.

```
>>> while x < 5:
...         print(x)
...         x = x + 1
...
0
1
2
3
4
```

무한 루프

무한 루프란 종료되지 않고 한없이 반복되는 것이다. 흔히 '무한 루프에 빠졌다'고 하여 실수로 끝나는 지점이나 조건을 설정하지 않아 생기는 오류를 지칭할 때 사용하기도 한다. 그러나 무한 루프는 의도적으로 사용될 수도 있는 것이고 항상 오류인 것은 아니다. for 문을 사용할 때는 반복 대상 목록을 지정하여 주지만 while 문에서는 추상적인 조건의 형태로 종료 조건이 제시되기 때문에 실수로 무한 루프에 빠지지 않도록 조심해야 한다.

연습 1.9 앞서 만든 BMI 프로그램이 하나의 입력값을 받아 끝나는 것이 아니라 하나의 입력에 대해 계산이 끝나면 다음 데이터를 입력 받아 계산하는 방식으로 계속 반복되도록 만들어보자. 사용자가 'quit'라고 입력하고 프로그램이 종료되도록 하자.

연습 1.10 앞서 만든 BMI 계산 함수를 이용하여 행에는 키, 열에는 몸무게를 배치한 BMI 표를 출력하는 프로그램을 작성하여라. 키의 범위와 몸무게의 범위, 그리고 등급의 간격을 적절히 설정하여라.

연습 1.11 앞의 연습 1.10의 결과를 HTML의 표를 이용하여 출력하는 프로그램을 작성하라. BMI 판정 결과에 따라 저체중, 보통체중, 과체중의 색깔을 달리하여 출력하여 보라. HTML에 관해서는 다음 링크 참조.

- `http://www.w3schools.com/html/`

연습 1.12 제곱근을 계산해 주는 함수를 만들어 보자. 즉, 임의의 양수 x가 주어졌을 때 $x = y^2$을 만족하는 양수 y를 구하는 것이다. 제곱근은 다음과 같은 절차를 반복함으로써 구할 수 있다. 주어진 양수 x의 제곱근의 참값이 y라고 할 때

- 적당히 제곱근이 될 수 있는 수를 하나 생각하고 g라고 하자.
- g와 x/g의 평균을 계산하여 이것을 다시 g라고 하자
- g^2과 x를 비교하자.
  - 그 차이가 너무 크다면 다시 2번으로 돌아가 반복하자.
  - 그 차이가 충분히 작다면 반복을 중단하고 g를 y의 근사값으로 결론 내리자.

함수의 이름은 `sqrt()`라고 하자.

Python 03 문자열

유현조

2022년 3월 2일

차 례

3

문자열

문자열(string)은 문자(character)들의 연쇄로 이루어진 자료형(data type)을 말한다. 숫자(numeric)와 함께 프로그래밍에서 기본적으로 사용되는 자료형이다. 우선 문자열 리터럴과 기본적인 문자열 관련 내장 함수를 사용하는 방법을 알아 본 후에 문자열에 대한 연산과 문자열의 메소드를 이용하여 문자열을 조작하는 방법을 소개할 것이다. 하지만 매뉴얼처럼 모든 기능과 옵션을 전부 나열하지는 않을 것이다. 언제나 도움말이나 파이썬 홈페이지의 문서를 참조하는 것이 좋다.

리터럴

컴퓨터 프로그래밍에서 리터럴(literal)이란 소스 코드에 어떤 값을 직접 써 넣은 것을 말한다. 파이썬에서 `3`이라고 써 주면 정수, `3.0`이라고 써 주면 소수를 의미한다. `0x88`이라고 써 주면 십육진수 정수, `0b10001000`라고 쓰면 이진수 정수를 의미한다. 이런 `3`, `3.0`, `0x88`, `0b10001000`과 같이 값을 소스 코드에 직접 써 준 것을 리터럴이라고 한다. 이와 마찬가지로 문자열 값을 소스 코드에 직접 써 넣은 것을 문자열 리터럴이라고 한다.

```
>>> 3
3
>>> 3.0
3.0
>>> 0x88
136
>>> 0b1000100
68
```

내장 함수

내장 함수(built-in function)는 한 프로그래밍 언어에서 아무런 추가적인 조작 없이 기본적으로 제공되는, 항상 사용 가능한 함수를 말한다. 일반적으로 범용 프로그래밍 언어에서는 극소수의 내장 함수만을 제공한다. 파이썬3에서는 70여개의 내장 함수를 제공하고 있다. 참조: `https://docs.python.org/3/library/functions.html`

내장 함수는 문법의 일부는 아니기 때문에 사용자가 덮어쓸 수 있다. 하지만 가능하면 그런 일은 피하는 것이 좋을 것이다. 예를 들어 `if`와 같은 예약어는 문법의 일부이기 때문에 사용자가 다른 용도로 사용하려고 하면 오류가 발생한다. 하지만 `max`와 같은 내장 함수의 이름은 사용자가 새롭게 정의하여 사용할 수 있다.

```
>>> if = 3
  File "<stdin>", line 1
    if = 3
       ^
SyntaxError: invalid syntax
>>> max
<built-in function max>
>>> max = 3
>>> max
3
```

함수, 연산, 메소드

연산(operation)은 수학에서의 사칙 연산 같은 것을 생각해도 좋다. 연산자(operator)를 이용해 표현한다. 메소드(method)는 객체지향 프로그래밍에서 사용하는 개념이다. 일단은 한 객체 안에 들어있는 함수라고 생각하자. 여기서 함수, 연산, 메소드의 개념을 이론적으로 구별할 필요는 없다. 어떤 객체 s, t가 있고 함수 `fun`, 연산자 `op`, 메소드 `met`가 있을 때 다음과 같이 사용하는 형식이 다르다는 점만 기억하자.

함수	`fun`(s, t)
연산	s `op` t
메소드	s.`met`(t)

도움말

파이썬 인터프리터를 대화형으로 실행한 후 `help()`로 도움말을 찾아 볼 수 있다. 문자열(`str`)에 대한 도움말은 `help(str)` 명령으로 찾을 수 있다.

파이썬 문서

파이썬3에 관련된 문서들을 볼 수 있는 곳 `https://docs.python.org/3/` 사이트를 둘러보거나 인터넷 검색을 하면 된다. 인터넷 상의 정보를 볼 때는 파이썬2와 파이썬3에

차이가 있다는 점에 주의하면서 정보를 취사 선택할 필요가 있다. 가능하면 파이썬 공식 사이트 python.org에 있는 정보를 보는 것이 좋다.

3.1 문자열 만들기

3.1.1 문자열 리터럴

따옴표

문자열은 겉모습으로 말하자면 따옴표로 둘러싼 것을 말한다. 예를 들어 다음은 숫자이다.

```
>>> n = 3
```

하지만 다음은 문자열이다.

```
>>> s = '3'
```

문자열을 만들기 위해서는 작은따옴표(') 또는 큰따옴표(")[1]로 둘러싸서 문자열임을 나타내야 한다.

```
>>> s = "Hello World"
```

작은따옴표와 큰따옴표 사이에 의미의 차이는 없다. 둘 중의 어느 것을 쓸 것인가는 취향의 문제이다. 파이썬은 둘을 전혀 구별하지 않지만 용도를 구별해서 쓰는 사람들도 있다. 문자열에 따옴표가 포함되어야 할 때 유용하기도 하다. 예를 들어, 다음과 같이 문자열을 만들려고 하면 오류가 난다.

```
>>> title = 'Alice's Adventures in Wonderland'
  File "<stdin>", line 1
    title = 'Alice's Adventures in Wonderland'
                   ^
SyntaxError: invalid syntax
```

위의 오류 메시지를 잘 생각해 보아라. 왜 s 위치에서 에러가 났는지. 이런 오류는 다음과 같이 간단히 피할 수 있다.

```
>>> title = "Alice's Adventures in Wonderland"
>>> title
"Alice's Adventures in Wonderland"
```

[1]이 두 기호는 아스키 문자로 유니코드의 공식적인 명칭은 아포스트로피(apostrophe, U+0027, '), 따옴표(quotation mark, U+0022, ")이다. 일반적인 문서 작성할 때 사용하는 작은따옴표(‘)와 큰따옴표(“”)를 사용해서는 안 된다. 그리고 키보드에 있는 역따옴표(`)도 이런 기능은 없다. 영어로 이 문자의 유니코드 공식 명칭은 ‘grave accent’이며 프로그래머들은 흔히 ‘backquote’ 또는 ‘backtick’이라고 한다. 한국어로는 ‘grave accent’를 ‘역음 부호’라고 하는데 프로그래머들이 이런 용어를 쓰지는 않는다. 이 문자는 파이썬2에서는 별도의 의미가 있었지만 파이썬3에서는 제거되었다.

확장 문자

아포스트로피를 문자열에 포함하기 위한 다른 방법도 있다. 아포스트로피 앞에 역빗금(\)[2]을 붙여주는 것이다. 예를 들면 다음과 같다.

```
>>> title = 'Alice\'s Adventures in Wonderland'
>>> title
"Alice's Adventures in Wonderland"
```

이렇게 특정한 문자열을 입력하기 위해 사용되는 연쇄를 확장열(escape sequence)이라고 하고 이런 확장열을 불러내는 기호를 확장 문자 또는 탈출 문자(escape character)라고 한다.

파이썬에서 탈출 문자는 역빗금(\)이다. 이 문자 자체를 문자열에 포함시키고 싶을 때에는 \\를 이용한다. 따옴표 두 가지 모두 문자열에 포함시키고 싶을 때 탈출 문자를 이용한다.

```
>>> s = '\\\'\"'
>>> print(s)
\'"
```

확장 문자를 이용하여 특수한 문자들을 입력할 수 있다. 자주 사용되는 것은 새줄(newline) 문자를 \n으로 나타내고 탭 문자을 \t로 나타내는 것이다.

```
>>> table = "apple\t10\norange\t5"
>>> table
'apple\t10\norange\t5'
>>> print(table)
apple        10
orange        5
```

문자열에서 자주 사용되는 확장열에는 다음과 같은 것들이 있다.

확장열	의미
\\	역빗금(\)
\'	아포스트로피(')
\"	따옴표(")
\n	새줄 문자(LF, 아스키 10번 문자)
\t	탭 문자(TAB, 아스키 9번 문자)
\u*hhhh*	16비트 16진수 값 *hhhh* 번에 해당하는 유니코드 문자
\x*hh*	8비트 16진수 값 *hh* 번에 해당하는 유니코드 문자

[2] 이 문자의 유니코드 공식 이름은 'reverse solidus'이다. 빗금(solidus) 문자 /의 반대 모양이라는 것이다. 흔히 역슬래시 또는 백슬래시(backslash)라고도 한다.

날 문자열

파이썬에서는 날 문자열(raw string)이라고 하여 따옴표 앞에 r자를 덧붙여 문자열을 표현하는 방법을 제공하고 있다. 이 문자열은 탈출 문자를 확장하지 않는 문자열이다. 이런 장치는 역빗금(\)이 혼란스럽게 많아지는 문제를 해결하기 위해서 도입된 것이다. 예를 들어, 문자열에서 탭을 표현하기 위해 확장열 \t를 사용한다.

```
>>> s = 'a\tb'
>>> s
'a\tb'
>>> print(s)
a	b
```

만약 문자열에서 말 그대로 역빗금과 문자 t가 연달아 있는 것을 표현하고 싶다면 \\를 사용해야 한다.

```
>>> r = 'a\\tb'
>>> print(r)
a\tb
```

이런 경우 날 문자열을 사용하면 편리하다. 탈출 문자로 시작하는 연쇄를 확장하지 않기 때문에 있는 그대로 입력할 수 있다.

```
>>> r = r'a\tb'
>>> print(r)
a\tb
```

이것이 그다지 무슨 쓸모가 있을까 하는 생감이 들 수 있겠다. 하지만 정규표현(regular expression)이라는 도구를 사용할 때에는 정말 유용하게 사용된다.

여러 줄 문자열

문자열은 따옴표 세 개로 둘러쌀 수도 있다. 이렇게 세겹의 따옴표를 사용할 때는 큰따옴표만 사용하는 것을 권장하고 있다.

```
>>> """Hello World"""
'Hello World'
>>> '''Hello World'''
'Hello World'
```

이것은 주로 여러 줄로 이루어진 문자열을 입력할 때 사용한다.

```
msg = """first line
second line
third line"""
```

이것을 따옴표 하나인 문자열로 작성하려면 반드시 다음과 같이 확장열로 새줄 문자를 대신해야 한다.

```
msg = "first line\nsecond line\nthird line"
```

참고로 세겹 따옴표 문자열은 특수한 용도로 사용되기도 한다. 파이썬에서 '문서화 문자열(docstring)'이라고 불리는 것으로 소스 코드 안에 설명을 추가할 때 사용된다. 일반적으로 권장되는 것은 아니지만 주석이 여러 줄 이어질 때 세겹 따옴표를 사용하는 사람도 있다. 예를 들어 다음과 같이 함수에 설명을 다는 것을 문서화 문자열이라고 한다. 이때에는 큰따옴표만 사용하는 것이 파이썬의 공식적인 지침이다.

```
def add(x, y) :
    """calculate sum of x and y"""
    return x + y
```

이것은 기호(#)로 시작하는 주석과 달리 파이썬 인터프리터에 의해 처리가 되고 `add.__doc__`로 그 내용을 확인할 수 있다.

3.1.2 수치형과 문자형

문자열이 숫자를 표현하고 있을 때 `int()`, `float()` 함수를 이용하여 수치형 자료로 바꿀 수 있다.

```
>>> c = "299792458"
>>> int(c)
299792458
```

이 함수들은 외부에서 입력 받은 데이터가 숫자일 때 유용하게 사용할 수 있다.

반대로 `str()`을 이용하여 숫자를 문자열로 바꿀 수도 있다.

```
>>> str(3)
'3'
>>> L = 1.86498e-38
>>> str(L)
'1.86498e-38'
```

수치형 자료 이외에 다른 많은 것들을 문자열로 바꿀 수 있다. 당장은 정수 `3`을 문자열 `'3'`으로 바꾸는 것이 무슨 쓸모가 있는지 의문이 들 수도 있겠지만 파이썬에서는 유용하게 널리 사용되는 함수 중의 하나이다.

3.1.3 기본 입출력

입력

내장 함수 중 `input()`은 한 행을 입력 받아 문자열로 바꾸어 준다.

```
>>> s = input()
python
>>> s
'python'
```

이 함수로 받은 결과는 문자열이기 때문에 숫자를 입력 받고 싶다면 int() 또는 float()를 이용하여 숫자로 바꾸어야 한다.

```
>>> x = int(input())
3
>>> x / 2
1.5
```

출력

내장 함수 중 print()는 기본 출력을 담당하고 있다. 문자열을 인자로 주면 당연히 있는 그대로 출력된다.

```
>>> print('Hello World')
Hello World
```

문자열이 아닌 것을 인자로 주면 어떨까? 숫자를 인자로 주면 있는 그대로 출력되는 것처럼 보인다.

```
>>> print(3.14)
3.14
>>> 3.14
3.14
>>> str(3.14)
'3.14'
```

하지만 사실은 먼저 문자열로 변환된 후에 출력된 것이다. 어떤 대상에 str()를 적용했을 때 얻어지는 형태로 출력이 된다.

출력할 대상이 여럿일 때에는 쉼표로 구분하여 여러 개의 인자를 나열하면 된다. 인자의 개수에 제한이 없이 사용할 수 있다.

```
>>> print('hello', 'world')
hello world
>>> print("year", 2012)
year 2012
```

위의 결과에서 볼 수 있듯이 print()는 문자열 마지막에 새줄 문자를 추가하고 쉼표로 나열한 인자들 사이에 스페이스 하나를 구분자로 넣어 출력해 준다.

구분자를 스페이스 대신 다른 것을 사용할 수도 있다. sep 옵션을 이용하여 구분자를 지정해 주면 된다.

```
>>> print(21, 34, 55, sep='-')
21-34-55
```

출력할 때 새줄 문자를 더하고 싶지 않을 때도 있다. end 옵션을 사용하여 줄 끝에 어떤 문자를 넣을지 지정할 수 있다.

```
>>> for n in [21, 34, 55]:
...     print(n, end='')
...
213455>>>
```

새줄 문자가 추가되지 않았기 때문에 마지막에 파이썬 프롬프트가 출력 결과 뒤에 붙어서 나왔다.

예제 3.1

아스키 문자를 출력하는 프로그램을 작성해 보자. 아스키 문자를 번호순으로 전체 목록을 출력해 보자. 반복문을 이용하여 프로그램을 작성하자.

우선 이 문제를 풀기에 앞서 반복문을 사용하는 두 가지 방법을 알아 보자. 반복할 대상 목록이 주어진 경우에는 다음과 같은 형식으로 반복문을 사용할 수 있다.

```
>>> cs = ['A', 'B', 'C']
>>> for c in cs:
...    print(x, end='')
...
ABC
```

목록은 없고 숫자 범위만 주어진 경우를 다루기 위해 파이썬에서는 '범위(range)'라는 타입을 제공한다. 이것은 range() 함수로 만들 수 있다. 다음과 같은 형식으로 사용한다.

```
>>> for i in range(0, 4):
...     print(i, end='')
...
0123
```

위 출력 결과에서 볼 수 있듯이 range(0, 4)라고 하면 0, 1, 2, 3을 의미한다. 4는 포함되지 않는다.

이제 아스키 문자를 출력해 보자. 아스키 문자 중 32번부터 126번까지 한 행에 출력하는 프로그램을 작성해 보자.

```
>>> for i in range(32, 127):
...     print(chr(i), end='')
...
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`
abcdefghijklmnopqrstuvwxyz{|}~>>>
```

여기에서 두 줄로 나온 것은 화면이 좁아서 그런 것일 뿐이다. '한 행'이라는 의미는 중간에 줄을 바꾸어 주는 문자가 없다는 뜻이다.

3.1.4 문자코드와 인코딩

아스키 문자

지금까지 의도적으로 문자열에 사용하는 문자를 영문 키보드를 통해 직접 입력할 수 있는 것으로 한정하여 이야기하였다. 이 문자들을 아스키(ASCII) 문자라고 한다. 아스키라는 말은 'American Standard Code for Information Interchange'의 줄임말이다. 아스키 문자는 7비트 인코딩으로 모두 128(= 2^7)개의 문자로 이루어져 있다. 33개의 출력 불가능한 제어 문자와 95개의 출력 가능 문자로 구분된다. 출력 가능 문자는 영문 대소문자 52개, 숫자 10개, 문장 부호 및 기호 32개, 스페이스 1개이다.

유니코드 문자

이제 아스키를 벗어나 한글을 포함한 다른 문자들을 사용해 보자. 파이썬3의 문자열은 유니코드[3] 문자열이다. 세계 모든 문자를 자유롭게 사용할 수 있다.

한 문자 c의 유니코드 값은 `ord`(c)를 이용하여 알 수 있다. 코드 값 n을 알면 `chr`(n)을 이용하여 문자를 얻을 수 있다. 다음 예에서 영어 소문자 'a'가 97번 문자라는 것을 알 수 있다.

```
>>> ord('a')
97
>>> chr(97)
'a'
```

한글도 가능하다. 다음 예에서 한글 '가'의 유니코드 번호가 44032번임을 알 수 있다.

```
>>> ord('가')
44032
>>> chr(44032)
'가'
```

유니코드 번호는 일반적으로 16진수로 표현한다. 10진수에 대응되는 16진수 표현을 알고 싶다면 `hex()` 함수를 이용하고 16진수를 입력하고 싶다면 `0x`$hhhh$ 형식을 이용한다. 예를 들면 다음과 같다.

```
>>> hex(44032)
'0xac00'
>>> 0xac00
44032
```

유니코드 번호를 이용할 때에는 십진수를 이용하는 것보다 십육진수를 이용하는 것이 편리할 때가 많다. 이때 `0x`$hhhh$ 형식을 이용한다.

[3] 앞에서 이미 '유니코드(Unicode)'라는 말이 몇번 등장하였다. 무엇인지 모른다면 검색해 보자. 유니코드는 모든 문자를 컴퓨터에서 일관성 있게 호환가능하도록 만든 표준이다.

```
>>> hex(ord('가'))
'0xac00'
>>> chr(0xac00)
'가'
```

파이썬2

참고로 파이썬2의 문자열은 아스키이고 유니코드 문자열은 별도로 취급하므로 파이썬2에서 유니코드 문자열을 다룰 때에는 주의해야 한다. 파이썬2에서 한글처럼 아스키 범위를 벗어나는 문자는 유니코드 문자열로 만들어야 한다. 이때는 따옴표 앞에 u를 붙여서 유니코드 문자열임을 알려주어야 한다.

```
>>> u = u"안녕하세요"
```

파이썬3에서도 파이썬2와의 호환을 위해서 위와 같은 접두사 u를 허락하고 있지만 사용할 필요는 없다.

소스 코드 인코딩

유니코드 덕분에 세계의 다양한 문자를 쉽게 이용하게 되었지만 아직 인코딩[4]이라는 문제가 남아 있다. 동일하게 유니코드를 쓰더라도 몇가지 서로 다른 인코딩을 이용하여 파일을 저장할 수 있다. 파이썬을 비롯하여 컴퓨터 프로그래밍에서 널리 사용되는 인코딩은 UTF-8이다. 파이썬 소스 코드는 UTF-8로 작성하는 것이 좋다.

현재 Linux나 macOS에서는 기본적으로 시스템 자체가 UTF-8을 기본값으로 사용도록 되어있어 특별히 신경을 쓰지 않아도 대부분의 경우 문제가 생기지 않는다. Microsoft Windows의 경우에는 나라마다 다른 인코딩이 기본값으로 되어 있다. 한국어 Windows의 경우에는 CP949라는 인코딩이 기본값으로 메모장 등에서 한글이 포함된 파일을 작성할 경우 CP949로 저장된다. 이 경우 문제가 생길 수 있으므로 파일을 저장할 때 UTF-8을 사용하는 습관을 들이는 것이 좋다. 한편 cmd 등 터미널도 CP949 인코딩을 사용하기 때문에 문제가 생길 수 있다. 대신에 Cygwin Bash, Git Bash 등과 같이 UTF-8을 기본으로 하고 있는 터미널을 쓰는 것이 좋다.

파이썬 소스 코드를 반드시 UTF-8로 작성해야 하는 것은 아니다. 다음과 같이 파일의 첫번째 또는 두번째 행에 다음과 같은 형식으로 인코딩을 지정해 주면 원하는 인코딩을 사용할 수 있다.

```
# -*- coding : cp949 -*-
```

파이썬3에서 UTF-8을 사용할 경우 인코딩을 명시할 필요가 없다. 가능하면 다른 인코딩을 사용하지 않도록 하자.

---

[4] 문자 인코딩(character encoding)이란 간단하게는 문자를 컴퓨터에서 표현하기 위한 방법이라고 말할 수 있다. 자세한 내용은 검색해 보자.

연습 3.1 다음과 같이 한글 호환 자모의 문자표를 출력하는 프로그램을 작성해 보자. 세 열로 구성되어 있다. 유니코드 코드 값에 해당하는 십진수, 십육진수, 문자를 출력한 것이다. 구분자로는 탭 문자를 사용하였다.

```
12592	0x3130
12593	0x3131	ㄱ
12594	0x3132	ㄲ
12595	0x3133	ㄳ
12596	0x3134	ㄴ
...
12683	0x318b	ㆇ
12684	0x318c	ㆈ
12685	0x318d	ㆍ
12686	0x318e	ㆎ
12687	0x318f
```

참고로 정수 n을 십육진수를 표현하는 문자열로 바꾸고 싶을 때에는 hex(n)를 이용한다.

연습 3.2 아래 예시와 같은 형식으로 유니코드 문자표를 한 행에 16개씩 출력하는 프로그램을 작성해 보자. 명령행 인자로 출력한 범위를 입력 받자. 하지만 정확하게 주어진 범위만 출력하는 것이 아니라 주어진 범위를 포함하는 최소의 직사각형 형태로 출력하도록 하자.

```
$ python unitab.py 32 127
0x2          !"#$%&'()*+,-./
0x3         0123456789:;<=>?
0x4         @ABCDEFGHIJKLMNO
0x5         PQRSTUVWXYZ[\]^_
0x6         `abcdefghijklmno
0x7         pqrstuvwxyz{¦}~

$ python unitab.py 0x3131 0x3180
0x313           ㄱ ㄲ ㄳ ㄴ ㄵ ㄶ ㄷ ㄸ ㄹ ㄺ ㄻ ㄼ ㄽ ㄾ ㄿ
0x314         ㅀ ㅁ ㅂ ㅃ ㅄ ㅅ ㅆ ㅇ ㅈ ㅉ ㅊ ㅋ ㅌ ㅍ ㅎ ㅏ
0x315         ㅐ ㅑ ㅒ ㅓ ㅔ ㅕ ㅖ ㅗ ㅘ ㅙ ㅚ ㅛ ㅜ ㅝ ㅞ ㅟ
0x316         ㅠ ㅡ ㅢ ㅣ   ㅥ ㅦ ㅧ ㅨ ㅩ ㅪ ㅫ ㅬ ㅭ ㅮ ㅯ
0x317         ㅰ ㅱ ㅲ ㅳ ㅴ ㅵ ㅶ ㅷ ㅸ ㅹ ㅺ ㅻ ㅼ ㅽ ㅾ ㅿ
0x318         ㆀ ㆁ ㆂ ㆃ ㆄ ㆅ ㆆ ㆇ ㆈ ㆉ ㆊ ㆋ ㆌ ㆍ ㆎ
```

명령행 인자를 십육진수로 받았을 때에도 int()로 처리할 수 있다. base=0 옵션을 주면 정수 리터럴을 알아서 처리해 준다.

```
>>> s = '0xac00'
>>> int(s, base=0)
44032
```

3.1.5 파일 입출력

파일은 텍스트 파일과 이진(binary) 파일로 구별된다. 어떤 파일을 텍스트 파일로 읽는다는 것은 그것이 어떤 인코딩으로 되어있느냐에 따라 적절하게 복호화하여 문자들의 연쇄, 즉, 문자열을 얻는다는 의미이다. 어떤 파일을 이진 파일로 읽는다는 것은 파일 안에 나열된 0과 1의 연쇄를 그대로 바이트 단위의 연쇄, 즉 8비트 단위의 연쇄, 즉 0부터 255까지의 정수들의 연쇄로 읽는다는 뜻이다. 예를 들면,

`00000000 10101100`	파일의 내용
`'가'`	UTF-16-LE 텍스트 파일로 읽었을 때
`b'\x00\xac'`	이진 파일로 읽었을 때

여기서 따옴표 앞에 `b`라고 붙인 것은 파이썬에서 '`bytes`'라고 하는 바이트열을 표시하기 위한 접두사이다.

파이썬에서는 파일을 읽기 위한 내장 함수 `open`()을 제공하고 있으며 이 함수는 파일을 여는 방식을 특별히 지정하지 않는 한 기본적으로 주어진 파일을 읽기 전용 텍스트로 읽는다. 파일 여는 방식을 지정하면 쓰기를 위해 열 수도 있고 이진 파일로 열 수도 있다.

텍스트 파일로 열 때에 인코딩을 명시적으로 지정할 수 있다. 인코딩을 지정하지 않으면 플랫폼에 따라 달라진다. 예를 들어 UTF-8 텍스트 파일을 읽으려면 다음과 같이 파일 경로와 인코딩을 지정한다.

```
>>> file = open('filename.txt', encoding='utf8')
```

파일 열기 함수 `open`()은 그 결과를 파일 객체로 돌려준다. 파일 객체는 특정 파일을 조작할 수 있는 기능을 제공한다. 다음과 같이 읽고 쓰고 닫는 기본적인 기능을 포함하여 여러 가지 기능을 제공한다.

file.`read`()
file.`readline`()
file.`readlines`()
file.`write`()
file.`close`()

파일을 읽으려고 할 때 반드시 위의 읽기 함수를 써야 하는 것은 아니다. 텍스트 파일을 읽을 때에는 흔히 한 행씩 차례대로 읽으면서 처리하는 경우가 많다. 이때에는 다음과 같은 형식으로 반복문을 사용하는 것이 편리하다.

`for` *line* `in` *file* `:`

위와 같이 반복문을 사용하면 파일에서 행 단위로 내용을 읽어들인다. 이 코드는 보기에도 깔끔할 뿐만 아니라 메모리 사용과 속도 측면에서 효율적이다.

예제 3.2

한 행에 하나의 숫자가 들어 있는 텍스트 파일을 읽어 합을 구하는 프로그램을 작성해 보자. 예를 들어 numbers.txt라는 파일에 다음 내용이 들어 있을 때,

```
1
2
3
4
```

다음과 같이 작동하는 프로그램이다. 파일의 이름을 명령행 인자로 받아서 읽도록 하자.

```
$ python sum.py numbers.txt
10.0
```

이 프로그램을 작성하기에 앞서 숫자들의 나열이 주어졌을 때 합을 구하는 방법을 알아보자. 다음과 같은 구문을 이용한다.

```
xs = [1, 2, 3, 4, 5]

total = 0
for x in xs:
    total += x
```

우선 합을 저장하기 위한 변수를 하나 만들어 초기화를 한다. 위의 경우 `total`이라는 변수이다. 합을 구하기 위한 것이기 때문에 0으로 초기화한다. 다음으로 반복문을 돌리면서 목록에 들어있는 원소를 하나씩 꺼내서 `total`에 누적해서 더한다. 누적해서 더할 때에는 `+=` 연산을 이용한다.

이제 우리가 짜려던 프로그램을 만들어 보자. 파일 이름을 명령행 인자에서 받기 위해서는 `sys.argv`를 이용하면 된다.

```
# sum.py
import sys

file = open(sys.argv[1])

total = 0
for line in file:
    total += float(line)

print(total)
```

파일에서 읽어 들인 것은 문자열이므로 합을 구하기 위해서는 우선 숫자로 바꾼 후 계산해야 한다. 숫자로 바꿀 때에는 정수는 `int()`, 소수는 `float()`를 사용한다.

예제 3.3

텍스트 파일을 읽어 행 번호를 붙여서 출력해 주는 프로그램을 작성해 보자. 명령행 인자로 파일 이름을 받도록 하자. 이 프로그램은 다음처럼 작동한다.

```
$ python nl.py nl.py
1 : # nl.py - numer lines of files
2 : import sys
3 :
4 : file = open(sys.argv[1])
5 :
6 : n = 0
7 : for line in file:
8 :     n += 1
9 :     print(n, ':', line, end='')
10 :
```

번호를 나타내는 데에 쓸 n이라는 변수를 0으로 초기화한 후에 반복문 안에 'n += 1' 이라는 문장을 써서 1회 반복할 때마다 n이 1씩 증가하는 효과를 내고 있다.

3.1.6 인코딩

텍스트 파일을 읽고 쓸 때에는 open()에 인코딩을 지정해 파일을 열면 주면 파일의 내용이 일괄적으로 인코딩이 처리된다. 파일에 들어 있는 이진 데이터를 읽을 때 문자열로 바꾸는 것을 복호화(decoding)이라고 하고 문자열을 파일에 쓸 때 이진 데이터로 바꾸는 것을 부호화(encoding)이라고 한다.

이진 데이터는 바이트의 연쇄로 표현할 수 있고 파이썬에서 이것을 바이트열(bytes)이라고 한다. 따라서 바이트열을 문자열로 바꾸는 것이 복호화, 문자열을 바이트열로 바꾸는 것이 부호화에 해당한다. 파이썬에서는 문자열(str)의 encode() 메소드와 바이트열(bytes)의 decode() 메소드를 이용하여 다양한 인코딩을 다룰 수 있다.

str.**encode**(*encoding*)	문자열을 바이트열로
bytes.**decode**(*encoding*)	바이트열을 문자열로

예를 들면 다음과 같다. 한 문자열이 주어졌을 때 지정된 인코딩에 맞추어 바이트열로 부호화할 수 있다.

```
>>> s = '가'
>>> s.encode('utf-16-le')
b'\x00\xac'
```

여기서 따옴표 앞에 붙은 접두사 b는 바이트열을 표시하기 위한 것이다. 한 바이트열이 주어졌을 때 지정된 인코딩에 맞추어 문자열로 복호화할 수 있다.

```
>>> b = b'\x00\xac'
>>> b.decode('utf-16-le')
'가'
```

3.2 문자열은 나열형

파이썬의 문자열은 나열형 자료의 하나다. 문자들이 나열된 것으로 보는 것이다. 여기서 설명할 기능들은 문자열 뿐만 아니라 다른 나열형 자료에도 동일하게 사용할 수 있다.

3.2.1 길이

문자열의 길이는 len()을 이용하여 알 수 있다. 문자열은 문자의 나열이므로 길이는 성분 문자의 개수가 된다.

```
>>> s = 'python programming'
>>> len(s)
18
```

문자열은 유니코드 문자의 나열이므로 각각의 유니코드 문자가 1개로 처리된다.

```
>>> h = "안녕하세요"
>>> len(h)
5
```

어떤 유니코드 문자를 이용하여 입력했는지 주의해야 한다. 눈에 보이는 것과는 다를 수 있다. 특히 한글의 경우에는 유니코드에 음절문자(Hangul Syllable)와 자모(Hangul Jamo)가 따로 들어있어 혼동을 일으키는 경우가 있다. 다음은

```
>>> s = "\uac00"
>>> s
'가'
>>> len(s)
1
>>> j = "\u1100\u1161"
>>> j
'가'
>>> len(j)
2
```

3.2.2 이어 붙이기

더하기

두 문자열을 이어 붙여서 하나로 만드는 것을 접합(concatenation)이라고 한다. 파이썬에서는 문자열 s와 t를 더하기 기호를 이용하여 $s + t$로 연결하여 이어 붙인 새로운 문자열을 얻을 수 있다.

```
>>> 'alice' + '@' + 'wonderland'
'alice@wonderland'
```

이러한 연산들은 그 연산의 의미가 정의되어 있을 때만 가능하다. 문자열 사이의 곱, 문자열과 부동소수점수와의 곱이나 문자열과 숫자형의 합 등은 불가능하다. 문자열과 숫자를 연결시켜야 하는 경우는 자주 발생한다.

```
>>> "year" + 2012
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

이런 경우 str()을 이용하여 숫자를 문자열로 바꾼 후 이어 붙이기를 할 수 있다.

```
>>> "year" + str(2012)
'year2012'
```

곱하기: 반복해서 이어 붙이기

문자열 s와 정수 n을 곱하면 문자열 s를 n번 반복하여 이어 붙여 새로운 문자열을 만들 수 있다. 문자열과 정수를 쓰는 순서는 관계없다.

```
>>> 'abc' * 10
'abcabcabcabcabcabcabcabcabcabc'
>>> '*' * 20
'********************'
>>> '12345' * 4
'12345123451234512345'
```

연습 3.3 (1) 행의 수 n이 지정되면 다음과 같이 별표를 출력하는 프로그램을 만들어 보아라. 예를 들어 $n = 5$이면,

```
*
**
***
****
*****
```

(2) 행의 수 n이 지정되면 다음과 같이 별표를 출력하는 프로그램을 만들어 보아라. 예를 들어 $n = 4$이면,

```
   *
  ***
 *****
*******
```

(3) 정수의 리스트 x가 주어지면 각 원소들의 수 많큼 별표를 이용하여 막대 그래프를 그려주는 프로그램을 만들어 보아라. 예를 들어 $x = [7, 19, 21, 13, 5]$이면,

```
*******
*******************
*********************
*************
*****
```

(4) 정수 n이 주어지면 다음과 같이 숫자를 출력하는 프로그램을 만들어 보아라. 예를 들어 $n = 6$이면,

```
     1
    1 2
   1 2 3
  1 2 3 4
 1 2 3 4 5
1 2 3 4 5 6
```

3.2.3 번호로 찾아보기

문자열을 구성하는 각 문자는 번호를 통해 접근할 수 있다. 이렇게 어떤 자료 모음에서 특정 값을 찾아줄 수 있게 하는 장치를 일반적으로 인덱스(index)라고 한다. 파이썬에서는 0부터 시작하는 번호를 인덱스로 사용한다.

문자열 뒤에 꺽쇠 괄호를 치고 번호를 지정하면 해당 번호에 위치한 문자가 무엇인지 찾아볼 수 있다. 예를 들면 다음과 같다.

```
>>> s = "안녕하세요!"
>>> s[0]
'안'
>>> s[1]
'녕'
>>> s[-1]
'!'
>>> s[-2]
'요'
```

위 예에서 보듯이 음수는 뒤에서부터 매긴 번호를 의미한다. 맨 뒤에서 첫번째 문자는 -1번, 맨 뒤에서 두번째 문자는 -2번이 되는 식이다. 문자열을 구성하는 특정 문자를 바꾸는 것은 불가능하다.

```
>>> h = "가나다"
>>> h[0] = "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

3.2.4 부분 문자열 조각

주어진 문자열의 일부 조각만 잘라내어 새로운 문자열을 얻는 방법도 있다. 이렇게 어떤 큰 덩어리 자료의 일부를 잘라내는 것을 슬라이싱(slicing)이라고 한다. 파이썬에서 부분 문자열 조각은 꺽쇠 괄호 안에 시작 번호와 끝 번호를 지정하여 얻을 수 있다. 예를 들면 다음과 같다.

```
>>> s = "abcdefghij"
>>> han[0:3]
'abc'
>>> han[3:5]
'de'
```

위에서 보듯이 'n:m'이라고 번호를 주면 n번부터 $m-1$번까지의 문자로 이루어진 조각이 얻어진다. m번이 포함되지 않는다는 점에 주의해야 한다. 이때에도 역시 음수 번호를 사용할 수 있다.

```
>>> s[-3:-1]
'hi'
```

번호를 비워놓을 수도 있는데 앞자리가 비어있으면 처음부터, 뒷자리가 비어있으면 마지막까지라는 의미가 된다.

```
>>> s[:3]
'abc'
>>> s[-3:]
'hij'
```

그리고 당연히

```
>>> s[:]
'abcdefghij'
```

그리고 동의 안 할 수도 있지만

```
>>> s[2:0]
''
```

그리고 따져보면 당연한 것이지만 직관적으로 편하게 받아들여지지 않는 것으로

```
>>> s[2:3]
'c'
>>> s[2]
'c'
```

3.2.5 반복문에서

문자열은 나열형 자료이므로 for 구문의 in 자리에 사용할 수 있다. 이때에는 개별 성분 문자를 하나씩 돌아가며 반복하라는 의미가 된다.

```
>>> s = "abc"
>>> for c in s:
...     print(c, hex(ord(c)))
...
a 0x61
b 0x62
c 0x63
```

예제 3.4

텍스트 파일을 읽어서 문자의 목록을 만들어 주는 프로그램을 작성하여라. 쉽게 말해 한 문자마다 엔터를 쳐주는 프로그램이다. 다음처럼 작동한다.

```
$ cat hello.txt
안녕! Hi!


$ python charlist.py hello.txt
안
녕
!

H
i
!
```

이 프로그램과 함께 텍스트 처리 도구 sort, uniq를 이용하면 주어진 텍스트 파일에 들어 있는 문자의 빈도표를 작성할 수도 있다. Microsoft Windows에서는 Cygwin 또는 Git Bash 등을 설치하면 두 명령을 사용할 수 있다.

연습 3.4 카이사르 암호[5]를 구현해보자. 아스키 영문자 대문자 A–Z만 사용하는 경우로 한정하자. 암호화 열쇠가 3이라면 카이사르 암호는 다음과 같은 치환암호가 된다.

```
평문:   ABCDEFGHIJKLMNOPQRSTUVWXYZ
복문:   DEFGHIJKLMNOPQRSTUVWXYZABC
```

다음과 같이 작동하는 함수를 만들어 보자.

```
>>> caesar.cipher('HOCBELLUMESTFINIS', 3)
'KRFEHOOXPHVWILQLV'
>>> caesar.decipher('KRFEHOOXPHVWILQLV', 3)
'HOCBELLUMESTFINIS'
```

카이사르 암호를 형식적으로 표현하면 다음과 같다. 한 언어에서 사용하는 문자가 모두 n개라고 하자. 문자들을 적절한 순서로 배열하고 번호를 붙이자. 이제 문자 대신 번호를 사용할 수 있다. 번호들의 집합은 $C = \{0, 1, \cdots, n-1\}$가 된다. 암호화 열쇠를 k, 평문(plaintext)의 문자를 p라고 하자. 평문의 문자 p를 열쇠 k를 통해 암호문으로 바꾸어 주는 함수를 $ci(p, k, n)$, 암호문의 문자 c를 열쇠 k를 통해 평문으로 바꾸어 주는 함수를 $de(c, k, n)$라고 하면 다음과 같은 간단한 수식으로 나타낼 수 있다.

$$ci(p, k, n) = (p + k) \mod n$$
$$de(c, k, n) = (c - k) \mod n$$

여기서 mod 연산은 나머지를 구하는 연산이라는 의미로 사용되었다.

3.2.6 비교

같음

두 문자열이 같은지 다른지 확인할 때에는 ==와 !=를 이용한다. 성분 문자가 완전히 동일해야 한다.

```
>>> 'a' == 'a'
True
>>> 'python' != 'Python'
True
```

당연한 이야기이지만 눈으로 보이는 문자의 모양으로 판단해서는 안 된다.

[5] http://en.wikipedia.org/wiki/Caesar_cipher

```
>>> s = '\u00e1'
>>> t = 'a\u0301'
>>> s
'á'
>>> t
'á'
>>> s == t
False
```

실제 텍스트 데이터를 파이썬에서 읽어서 처리하려고 할 때 이런 문제가 발생할 수 있다는 점을 잊지 말자.

순서

부등호도 이용할 수 있다. 문자열에서는 '크다'와 '작다'라는 말보다는 '먼저'와 '나중'이라는 말으로 이야기하는 것이 자연스러울 것이다.

```
>>> 'a' < 'b' < 'c'
True
```

이것은 문자열을 정렬했을 때의 순서와 개념적으로 일치한다. 여러 문자로 구성된 문자열을 비교하면 왼쪽에 있는 문자부터 순서대로 비교한다.

```
>>> 'cat' < 'dog'
True
>>> 'dog' < 'deer'
False
```

사전에 단어를 배열하는 순서를 떠올리면 이해하기 편할 것이다. 참고로 `sorted()` 함수를 이용하면 문자열의 리스트를 정렬할 수 있다.

```
>>> sorted(['dog', 'cat', 'deer'])
['cat', 'deer', 'dog']
```

포함

두 문자열에 대해 x `in` s 연산은 문자열 x가 s에 들어있으면, 다시 말해 문자열 s의 일부가 x와 동일하면 참이 되는 연산이다.

```
>>> s = "hello world"
>>> "hello" in s
True
>>> "word" in s
False
```

연습 3.5 텍스트 파일에서 특정 문자열이 들어있는 행을 찾아 행 번호와 함께 출력하는 프로그램을 작성하여라. 예를 들어, 다음과 같이 alice.txt라는 텍스트 파일에서 'Cheshire'라는 문자열이 포함된 행을 출력한다.

```
$ python find.py Cheshire alice.txt
1347 : 'It's a Cheshire cat,' said the Duchess, 'and that's why. Pig!'
1353 : 'I didn't know that Cheshire cats always grinned; in fact, I didn't know
1467 : was a little startled by seeing the Cheshire Cat sitting on a bough of a
1474 : 'Cheshire Puss,' she began, rather timidly, as she did not at all know
2102 : 'It's the Cheshire Cat: now I shall have somebody to talk to.'
2134 : 'It's a friend of mine--a Cheshire Cat,' said Alice: 'allow me to
2178 : When she got back to the Cheshire Cat, she was surprised to find quite a
```

참고로 이상한 나라의 앨리스의 영문 텍스트는 gutenberg.org 프로젝트에서 다운로드 받을 수 있다.

3.3 문자열 조작

3.3.1 껍질 벗기기

어떤 문자열의 바깥 부분, 즉 왼쪽과 오른쪽 부분에 있는 특정한 문자들을 벗겨 내고 가운데에 들어있는 알맹이 문자열만 빼내어 새로운 문자열을 만들 수 있다. 양쪽을 다 벗겨 낼 때에는 strip(), 왼쪽만 벗길 때는 lstrip(), 오른쪽만 벗길 때는 rstrip()을 사용한다.

벗기기 메소드는 파일에서 데이터를 읽거나 사용자의 입력을 받는 등에 외부에서 데이터를 가져올 때 유용하게 사용된다. 외부에서 가져온 문자열의 양 끝의 의미없는 탭, 스페이스, 새줄 문자 등 공백 문자를 지우는 것이 주된 쓰임새이다. 예를 들어, 외부에서 어떤 문자열을 받았는데 다음처럼 왼쪽에는 스페이스가 네 개, 오른쪽에는 탭이 세 개 있는 문자열이었다고 해보자. 여기에 아무 인자 없이 strip()을 하면 알맹이 문자열만 꺼낼 수 있다.

```
>>> s
'    Hello World\t\t\t'
>>> s.strip()
'Hello World'
```

항상은 아니지만 눈에 쉽게 띄는 왼쪽 공백 문자는 데이터로서 의미가 있고 눈에 띄지 않는 오른쪽 공백 문자는 의미가 없는 경우가 많다. 이런 경우에 rstrip()을 사용한다.

없애려는 문자의 목록을 담은 문자열을 인자로 지정하면 해당되는 문자로 이루어진 껍질은 모두 벗겨진다.

```
>>> s = "#$#?Hello World######"
>>> s.strip("#$?")
'Hello World'
```

단순한 지우기가 아니라 바깥 껍질을 벗겨 내는 것이라는 점에 주의해야 한다. 안쪽에 있는 있는 것은 사라지지 않는다.

```
>>> s = " Hello     World   "
>>> s.strip()
'Hello     World'
>>> s = "#Hello#World#"
>>> s.strip("#")
'Hello#World'
```

3.3.2 조각내고 합치고

조각내기

문자열을 특정한 문자열을 기준으로 잘라서 조각 문자열의 리스트로 만들 수 있다. 이때 자르는 기준이 되는 특정한 문자열을 분리자(separator) 또는 구분자(delimiter)라고 한다. 이를 위한 메소드로 `split()`, `rsplit()`, `partition()`, `rpartition()`이 있다. 여기서 `r` 접두사는 오른쪽이란 의미이다. 이들 기능은 유사하기 때문에 여기에서는 `split()`만 살펴볼 것이다.

주어진 문자열을 단어의 목록으로 만들고 싶을 때 아무런 인자없이 `split()`을 사용한다. 인자를 주지 않으면 공백 문자를 기준으로 쪼갠다. 공백 문자에는 스페이스, 탭(`\t`), 새줄문자(`\n`) 등이 있다.

```
>>> s = "Hello World"
>>> s.split()
['Hello', 'World']
```

공백 문자가 연달아 있는 경우 하나의 구분자로 처리한다.

```
>>> s = "Hello      World"
>>> s.split()
['Hello', 'World']
```

구분자를 명시적으로 지정할 수도 있다.

```
>>> s = "alice@wonderland"
>>> s.split('@')
['alice', 'wonderland']
```

문자열을 행 단위로 잘라 행의 리스트를 만들어 주는 `splitlines()`도 있다.

```
>>> 'first line\nsecond line'.splitlines()
['first line', 'second line']
```

단순한 함수이기는 하지만 행을 구분하는 구분자가 시스템마다 다양하기 때문에 유용하게 사용될 수 있다.

합치기

문자열을 조각낼 때 사용하는 split() 메소드와 짝을 지어 생각할 수 있는 합치기 메소드도 있다. join() 메소드이다. 여러 개의 문자열이 목록으로 나열되어 있을 때 이들을 합쳐 하나의 문자열로 만들 수 있다. 사용 형식이 좀 낯설게 보일 수 있다. 다음과 같은 형식으로 사용한다.

```
>>> '@'.join(['alice', 'wonderland'])
'alice@wonderland'
```

이 메소드는 흔히 구분자 없이 여러 문자열을 이어붙이려고 할 때 사용된다. 구분자를 빈 문자열로 지정하면 된다.

```
>>> lst = ['pro', 'gram', 'ming']
>>> ''.join(lst)
'programming'
```

예제 3.5

주어진 텍스트 파일을 단어의 목록으로 바꾸어 주는 프로그램을 만들어 보자. 쉽게 말해 한 단어마다 엔터를 쳐 주는 프로그램이다. 여기서 단어란 공백 문자로 구분된 단위를 말한다. 다음처럼 작동하도록 만들 것이다.

```
$ cat sent.txt
I heard every word you fellows were saying.

$ python wordlist.py sent.txt
I
heard
every
word
you
fellows
were
saying.
```

이 프로그램을 텍스트 처리 도구인 sort, uniq와 함께 사용하면 주어진 텍스트 파일에 들어 있는 단어들의 빈도를 계산할 수 있다.

예제 3.6

아래 왼쪽과 같이 단어 빈도 정보를 담고 있는 파일이 있다. 이 파일을 읽어 단어의 길이와 로그 빈도 값을 계산하여 오른쪽과 같은 형식으로 출력하는 프로그램을 작성하여라. 출력할 때 구분자는 탭을 사용하여라.

```
1664 the                the         3        1664        7.416979621381154
 780 and                and         3         780        6.659293919683638
 773 to                 to          2         773        6.650279048587422
 662 a                  a           1         662        6.495265555937008
 596 of                 of          2         596        6.39024066706535
 484 she                she         3         484        6.182084906716632
 416 said               said        4         416        6.030685260261263
 401 in                 in          2         401        5.993961427306569
 356 it                 it          2         356        5.87493073085203
 329 was                was         3         329        5.796057750765372
```

예제 3.7

자연 언어 텍스트가 가지는 어휘적 다양성을 측정하기 위한 개념으로 타입토큰비(TTR, Type-Token Ratio)라는 것이 있다. 타입의 수를 토큰의 수로 나눈 것을 말한다. 타입의 수란 텍스트에 사용된 단어의 종류가 몇 가지인가를 말하고 토큰의 수란 텍스트에 사용된 전체 단어의 수를 말한다.

자연 언어 텍스트의 단어는 아니지만 다음과 같은 간단한 예를 들어 보자.

```
gtgcgactccttctatgagtagaccacacctt
```

여기에서 문자의 빈도를 조사하면 다음과 같다.

```
      7 a
     10 c
      6 g
      9 t
```

총 4가지 문자가 사용되고 있으므로 타입의 수는 4이다. 전체 텍스트가 총 32개 문자로 이루어져 있으므로 토큰의 수는 32이다. 따라서 TTR은 $4/32 = 0.125$이다.

아래 왼쪽과 같은 단어 빈도 정보 파일이 주어졌을 때 타입토큰비를 계산하여 오른쪽처럼 출력하는 프로그램을 작성하여라.

```
$ head -6 wordfreq.txt
   1664 the
    780 and
    773 to
    662 a
    596 of
    484 she
```

```
$ python ttr.py wordfreq.txt
Number of Types = 6014
Number of Tokens = 29459
Type-Token Ratio = 0.20414813809022708
```

즉 왼쪽 빈도표 파일에 몇 행이 들어있는지 세어 타입의 수를 구하고 첫번째 컬럼의 숫자를 모두 더하여 토큰의 수를 구하면 된다.

예제 3.8

직사각형 형태로 수치들이 배열되어 있는 파일이 있다. 수학에서 이렇게 배열된 자료를 행렬(matrix)이라고 한다. 행의 수와 열의 수는 고정되어 있지 않다. 파일마다 달라질 수 있다. 예를 들면 다음과 같다.

```
$ cat mat.txt
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
```

이 파일을 읽어 행의 합을 구하는 프로그램을 작성하여라. 다음과 같이 작동한다.

```
$ python3 rowsums.py mat.txt
10.0
26.0
42.0
58.0
```

각 열의 합을 구하는 프로그램도 작성할 수 있겠는가?

3.3.3 찾고 바꾸고

시작과 끝

문자열의 시작 또는 끝 부분이 특정한 문자열인지 아닌지 확인할 필요가 종종 있다. `startswith()`와 `endswidth()`를 사용한다.

```
>>> s = 'python'
>>> s.startswith('p')
True
>>> s.startswith('py')
True
>>> s.endswith('on')
True
```

찾기

문자열을 대상으로 부분 문자열이 어디에 있는지 찾을 수도 있다. 이를 위한 메소드로 `find()`와 `index()`가 있다. `find()`는 찾지 못했을 경우 `-1`을 돌려주는 데에 비해 `index()`는 오류를 발생시키는다는 데에 차이가 있다. 각각 오른쪽에서부터 찾아주는 `rfind()`와 `rindex()` 메소드도 있다.

어떤 부분 문자열이 들어있는지 아닌지를 아는 것이 목적일 때에는 `in` 연산자를 사용해야 한다.

```
>>> "o" in "python"
True
```

어디에 있는지 알고 싶을 때에는 `find()`나 `index()`를 이용한다.

```
>>> s = "python"
>>> s.find("o")
4
```

이것으로 "python"의 4번 문자가 "o"라는 것을 알 수 있다. 부분 문자열이 시작하는 첫번째 위치를 알려준다.

```
>>> s = "python"
>>> s.find("on")
4
```

부분 문자열이 여러 번 등장하는 경우에는 첫번째 일치하는 위치를 알려준다.

```
>>> s = "python programming"
>>> s.find("o")
4
```

이때 다음과 같이 찾기 시작 위치를 지정해 주면 두번째 "o"의 위치를 찾을 수 있다.

```
>>> s.find("o", 5)
9
```

개수

문자열 내에 특정 부분 문자열이 몇 번 등장하는지 개수를 셀 수 있다. `count()`를 이용한다.

```
>>> s = 'python programming'
>>> s.count('p')
2
```

바꾸기

문자열의 일부분을 바꾸어 새로운 문자열을 만들 수 있다. 이를 위하여 `replace`(*old*, *new*) 메소드를 제공한다. 원래 문자열이 변경되는 것이 아니라 새로운 문자열이 만들어지는 것이라는 점에 주의하자.

```
>>> s = 'Hello World'
>>> s.replace('Hello', 'Hi')
'Hi World'
>>> s
'Hello World'
```

문자의 범주

우선 파이썬 문자열에서는 대문자와 소문자를 변환하는 메소드를 제공하고 있다. 유니코드에서는 라틴, 그리스, 키릴, 아르메니아, 조지아 문자 등에 대해 대소문자 구별 정보를 제공하고 있다. 주어진 문자열을 모두 소문자로 바꿀 때에는 `lower()`, 모두 대문자로 바꿀 때에는 `upper()`을 사용한다.

```
>>> "python".upper()
'PYTHON'
>>> "PYTHON".lower()
'python'
```

이와 함께 대소문자 변환 관련하여 몇가지 메소드가 더 있다. 다음과 같은 것들이 있다. 이름을 보면 대략의 기능을 유추할 수 있을 것이다. 정확한 기능은 도움말이나 파이썬 홈페이지의 문서에서 확인하는 것이 좋다.

- `lower()`
- `upper()`
- `casefold()`
- `swapcase()`
- `capitalize()`
- `title()`

다음으로 주어진 문자열이 어떤 범주에 해당하는지 검사해 주는 메소드들이 여럿 제공된다. 문자의 범주에는 숫자, 알파벳, 공백 문자 같은 것들이 있다. 다음과 같은 메소드들이 있다. 역시 이름을 보면 대략 기능을 유추할 수 있을 것이다.

- `isalnum()`
- `isalpha()`
- `isdecimal()`
- `isdigit()`
- `isnumeric()`
- `istitle()`
- `islower()`
- `isupper()`
- `isprintable()`
- `isspace()`
- `isidentifier()`

변환

주어진 문자열의 문자들을 변환표에 주어진 규칙에 따라 다른 문자로 바꾸어 새로운 문자열을 만들어 주는 translate()라는 메소드가 있다. 이 메소드를 사용하기 위해서는 변환표를 만들어 인자로 넣어 주어야 한다. 직접 변환표를 만들 수도 있지만 str.maketrans()를 이용하면 편리하게 변화표를 만들 수 있다.

다음은 룬 문자[6]를 로마자에 대응시킨 변환표를 만드는 예이다.

```
>>> tbl = str.maketrans("ᚠᚢᚦᚨᚱᚲᚷᚹᚺᚾᛁᛃᛇᛈᛉᛊᛏᛒᛖᛗᛚᛜᛞᛟ:", "fuþarkgwhnijïpzstbemlŋdo:")
>>> "ᚨᚨᚱᛈᚨᚨ:ᛁᚾᚨᚾᚨᚾᚨᛚᛟᛉᚾᚨᛒᛟᛉ".translate(tbl)
'aarpaa:inananaloznaboz'
```

문자 변환표를 str.maketrans()를 이용하지 않고 직접 만들면 일대일 문자 대응이 아닌 경우도 다룰 수 있다.

예제 3.9

명령행에서 인자로 주어진 숫자들의 합을 계산해 주는 프로그램을 만들어 보자. 다음과 같이 작동하는 프로그램이다.

```
$ python sum.py 1 2 3 4 5
15.0
```

명령해에 주어진 인자를 파이썬 안으로 가져오기 위해서는 sys.argv를 이용하면 된다.

```
# sum.py
import sys

total = 0
for x in sys.argv[1:]:
    total += float(x)

print(total)
```

주의할 것은 sys.argv의 0번 원소는 프로그램의 이름이기 때문에 덧셈의 대상이 아니라는 것이다. 1번부터 시작해야 한다. 그리고 sys.argv 안에 들어있는 것들은 문자열이므로 수치형 자료로 바꾼 뒤에 계산해야 한다.

[6] https://en.wikipedia.org/wiki/Hogganvik_runestone

3.3.4 끼워 넣기와 문자열 서식

끼워넣기 연산자

퍼센트(%) 기호를 이용한 연산은 문자열 중간에 다른 자료를 지정된 형식에 맞추어 끼워 넣을 수 있게 해준다. 전문 용어를 사용하면, 한 문자열 내부에 다른 것을 끼워 넣는다는 뜻에서 '내삽(interpolation)'이라고 하고 특정한 형식으로 꾸미다는 차원에서 '서식(formatting)'이라고 한다. 간단한 예를 들면 다음과 같다.

```
>>> msg = "Hello, %s!"
>>> name = "Alice"
>>> msg % name
'Hello, Alice!'
```

퍼센트 기호가 두 군데에 이용되고 있다. 둘은 모양만 퍼센트 기호로 동일할 뿐 전혀 다른 것이다. 문자열 `msg` 안에 들어 있는 퍼센트 기호로 시작하는 부분 `%s`는 서식을 지정하는 기능을 한다. 탈출 문자로 시작하는 확장열 `\n`, `\t`처럼 특별한 의미를 가지는 것이다. 두 문자열의 연산 `msg % name`의 퍼센트 기호는 왼쪽에 있는 문자열의 내부에 오른쪽에 있는 자료를 끼워 넣는 연산을 한다.

흔히 이렇게 문자열 내부에 `%`로 시작하는 연쇄를 이용하여 서식을 적용하는 방법을 'printf-style'이라고 한다. C 언어와 그 영향을 받은 여러 프로그래밍 언어에서 어떤 서식에 맞추어 출력하는 기능을 'print formatted'라는 의미에서 `printf`라는 이름의 함수로 제공하여온 전통 때문이다. 파이썬에는 `printf` 함수는 제공하지 않고 대신 끼워 넣기 연산자를 제공한다. 그리고 `%`를 이용한 서식은 파이썬에서도 동일하게 제공한다. 파이썬은 끼워 넣기 연산자가 있기 때문에 별도의 출력 함수는 필요없다. 그냥 `print` 함수를 사용하면 된다.

```
>>> print(msg % name)
Hello, Alice!
```

다른 프로그래밍 언어를 배운 후 파이썬을 배우기 시작한 사람들이 종종 파이썬에서 `printf`를 어떻게 하느냐고 묻는 경우가 있다. 이때에는 위와 같이 끼워 넣기 연산자를 사용하라고 알려 주면 된다.

서식 문자열에서는 다양한 기능을 제공한다. 여기에서는 기본적이고 널리 쓰이는 다음 네 가지만 간략하게 알아 보자.

- `%%` 퍼센트 기호 자체를 넣고 싶을 때
- `%s` 문자열로. `str()`을 적용한 결과
- `%d` 정수형 십진수로
- `%f` 소수형 십진수로

사용 예를 들면 다음과 같다.

```
>>> "내일 강수확률은 %d%%입니다" % 65
'내일 강수확률은 65%입니다'
>>> "현재 풍속은 %fm/s입니다" % 1.5
'현재 풍속은 1.500000m/s입니다'
```

퍼센트 기호와 서식 유형 문자 사이에 숫자를 넣어서 문자열의 길이 또는 숫자의 자릿수를 조정할 수도 있다. 예를 들어 **%3.1f**는 전체 길이가 3자이고 소숫점 아래 1자리가 되도록 서식을 설정하는 것이다.

```
>>> "현재 풍속은 %3.1fm/s입니다" % 1.5
'현재 풍속은 1.5m/s입니다'
```

자릿수를 나타내는 숫자를 0으로 시작하면 빈자리가 0으로 채워진다. 몇가지 다른 예를 더 보면 다음과 같다.

```
>>> "%s" % "hello"
'hello'
>>> "%10s" % "hello"
'     hello'
>>> "%8d" % 123
'     123'
>>> "%08d" % 123
'00000123'
>>>"%8.5f" % 3.14
' 3.14000'
>>> "%08.05f" % 3.14
'03.14000'
```

이렇게 문자수를 맞추는 기능은 명령행에서 사용하는 프로그램에서 결과를 보기 좋게 출력하려고 할 때 자주 사용된다.

여러 개의 자료를 동시에 끼워 넣을 수도 있다. 서식 퍼센트 기호를 여러 개 사용하면 된다.

```
>>> loc = "서울"
>>> temp = 11
>>> "내일 %s의 최저 기온은 %d도로 예상됩니다." % (loc, temp)
'내일 서울의 최저 기온은 11도로 예상됩니다.'
```

서식 메소드

문자열의 **format**() 메소드는 **%** 연산을 이용한 끼워 넣기에 대응되는 것이다. 끼워 넣기 연산자를 이용한 것이 구식, **format**() 메소드를 이용한 것이 신식이다. 간단한 예제를 통해 두 방식을 비교해 보자. 옛날 방식은 다음과 같다.

```
>>> 'My name is %s.' % 'Alice'
'My name is Alice.'
```

새로운 방식으로 쓰면 다음과 같다.

```
>>> 'My name is {}.'.format('Alice')
'My name is Alice.'
```

끼워 넣기할 대상의 개수는 중괄호의 개수와 일치한다. 인자에 주어진 순서대로 중괄호가 있는 위치에 끼워 넣기가 된다.

```
>>> loc = '서울'
>>> temp = 11
>>> msg = '내일 {}의 최저 기온은 {}도로 예상됩니다.'
>>> msg.format(loc, temp)
'내일 서울의 최저 기온은 11도로 예상됩니다.'
```

중괄호 안에 데이터의 이름을 지정하는 주면 더 이해하기 쉬운 코드를 작성할 수 있다.

```
>>> msg = '내일 {location}의 최저 기온은 {temperature}도로 예상됩니다.'
>>> msg.format(location='서울', temperature=18)
'내일 서울의 최저 기온은 18도로 예상됩니다.'
```

문자열 좌우 맞춤

문자열 서식에 해당하는 것은 아니지만 부수적인 메소드 하나를 추가로 살펴 보자.

문자열 좌우에 스페이스 또는 지정된 문자를 채워 넣어 가운데 맞춤 `center`(), 왼쪽 맞춤 `ljust`(), 오른쪽 맞춤 `rjust`()을 할 수 있다. 숫자를 위해 사용되는 것으로 왼쪽에 0을 채워 넣는 `zfill`() 메소드를 사용할 수도 있다. 예를 들어, 전체 길이가 20이 되도록 좌우에 별표를 넣고 가운데 맞춤을 하려면:

```
>>> "hello".center(20, "*")
'*******hello********'
```

이런 기능은 명령행에서 작동하는 프로그램을 만들 때 편리하게 쓰일 수 있다.

Python 04 자료구조

유현조

2020년 5월 11일

차 례

4

자료 구조

4.1 구조를 가진 자료형

자료 구조란 컴퓨터에서 정보를 구성하는 방식을 말한다. 이때 어떻게 하면 저장 공간을 효율적으로 사용하고, 어떻게 하면 정보를 읽고 쓰고 고치는 등의 조작을 빠르게 할 수 있을까 하는 것이 중요한 과제가 된다. 이 장은 자료 구조에 대한 근본적인 이해보다는 자료 구조에 대한 연구를 통해 이미 완성되어 파이썬에서 기본적으로 제공되고 있는 자료형들을 사용하는 방법을 이해하는 것을 목적으로 한다.

하나의 값을 위해 자료 구조가 필요하지는 않을 것이다. 여러 개의 값으로 이루어진 정보를 조직하기 위해서 자료 구조가 필요한 것이다. 파이썬에서는 단일한 값을 위한 정수형(`int`), 실수형(`float`), 논리형(`bool`) 자료형을 제공하듯이 여러 개의 값으로 이루어진 정보를 다루기 위한 자료형도 제공하고 있다. 이러한 자료형은 흔히 합성(composite), 복합(compound), 종합(aggregate) 등의 용어로 지칭된다. 이것은 자료 구조를 구현한 것이며 일종의 구조를 가진 자료형이라고 할 수 있다.

파이썬에서 제공되는 것을 겉모습에 따라 크게 구분하면 나열(sequence)형[1], 사전(dictionary)형[2], 집합(set)형이 있다. 나열형으로 `tuple`, `list`, `range`라는 세 가지 자료형을 제공하고 있다. 사전형으로 `dict`, 집합형으로 `set`, `frozenset`을 제공한다. 이들

[1] 나열, 서열, 배열(array), 목록(list), 열거(enumeration) 등 많은 용어들이 사용된다. 컴퓨터에 내부적으로 어떻게 구현되었느냐에 따라 이들 용어의 의미를 구별하기도 한다. 수학에서 수열(sequence)이라는 개념에 대응된다.

[2] 사전, 맵(map), 연상/연관 배열(associative array) 등 용어가 사용된다. 수학에서 사상(mapping)이라고 하는 개념에 대응된다.

자료형은 변경가능(mutable)한 것과 변경불가능(immutable)한 것으로 구별하여 볼 필요가 있다.

		자료형	예
나열형	변경불가	tuple	(0, 1, 2)
	변경가능	list	[0, 1, 2]
	변경불가	range	range(0, 3)
사전형	변경가능	dict	{'a' : 1, 'b' : 2, 'c' : 3}
집합형	변경가능	set	{1, 2, 3}
	변경불가	fronzenset	frozenset({1, 2, 3})

여기에서는 다루지 않겠지만 파이썬에서는 문자열과 바이트열도 넓은 의미에서 나열형 자료에 해당한다. 문자열을 위해서 str, 바이트열을 위해서 bytes, bytearray, memoryview와 같은 자료형이 제공된다.

4.2 나열형 자료

4.2.1 나열형 자료의 기본 특성

나열형 자료는 어떤 값들을 일렬로 늘어 놓은 구조를 가진다. 어떤 값들을 일렬로 늘어 놓았으므로 순서가 있다. 각각의 값들을 나열형 자료의 원소라고 부를 수 있다. 우리는 어렵지 않게 원소들이 차례대로 줄지어 있는 1차원의 구조를 떠올릴 수 있다. 더 나아가 원소 값이 또다시 복합형 자료가 되는 것을 허락하면 매우 복잡한 구조를 표현할 수도 있다.

파이썬에서는 3가지 나열형 자료를 기본적으로 제공하고 있다. 변경불가능한 순서짝 tuple과 변경가능한 목록 list로 임의의 원소들의 나열로 이루어진 자료를 표현할 수 있다. 추가적으로 반복문에서 유용하게 사용되는 범위 range가 있다.

범위 range

우선 나머지와 성격이 좀 다른 자료형인 범위 range부터 살펴보자. 이 자료형은 정수들이 지정된 구간에 일정한 간격으로 늘어선 자료를 위한 것이다. 다음과 같이 원하는 범위를 만들 수 있다.

사용법	예	의미
`range`($stop$)	`range(5)`	$0, 1, 2, 3, 4$
`range`($start$, $stop$)	`range(3, 7)`	$3, 4, 5, 6$
`range`($start$, $stop$, $step$)	`range(0, 11, 3)`	$0, 3, 6, 9$

범위를 표현하는 range는 실제 값을 입력한 list와 같은 것이 아니다. 우선 대화형 모드에서 출력을 보면 다른 형식으로 정보를 제공한다.

```
>>> a = [0, 1, 2]
>>> r = range(0, 3)
>>> a
[0, 1, 2]
>>> r
range(0, 3)
```

각각의 타입을 확인하여 보면 list와 range인 것을 확인할 수 있다.

```
>>> type(a)
<class 'list'>
>>> type(r)
<class 'range'>
```

둘 사이에 비교 연산자를 써보아도 전혀 다른 영역의 존재임을 알 수 있다.

```
>>> a = [0, 1, 2]
>>> b = [0, 1, 2]
>>> a == b
True
>>> r = range(0, 3)
>>> s = range(0, 3)
>>> r == s
True
>>> a == r
False
```

범위는 일반적으로 반복문과 함께 사용된다. 어떤 일을 n번 반복하려면 range(n)을 사용하면 된다. 예를 들어, 'hello'라는 메시지를 10번 출력하고 싶다면 다음과 같이 할 수 있다.

```
for _ in range(10):
    print('hello')
```

반복문 안에서 특정한 숫자 범위를 필요로 할 때에는 range에 범위를 지정해 주는 형식을 사용하면 된다. 이때 끝을 지정하는 값은 범위에 포함되지 않는다는 점에 주의해야 한다. 예를 들어, 다음 예에서는 유니코드 65번에서 90번까지의 문자를 출력해 준다.

```
>>> for n in range(65, 91):
...     print(chr(n), end='')
...
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

이런 경우에 list를 사용해서는 안 된다. 범위 range(65, 91)은 실제로 나열된 숫자들을 가지고 있는 것이 아니라 숫자를 나열하기 위한 규칙만 가지고 있다. 이에 비해 리스트라면 [65, 66, 67, 68, ..., 90]과 같이 실제로 숫자들을 나열하는 것이다. 원소가 몇개 되지 않을 때에는 범위와 리스트가 별다른 차이가 없어보일 수도 있지만 매우 큰 범위, 예를 들어, {range(0, 1000000)} 대신에 list를 쓰는 것은 자원의 낭비일 것이다.

순서짝 tuple

순서짝(tuple)은 정해진 개수의 값이 순서대로 나열된 것으로 수학에서는 n짝이라고도 한다. 수학에서 순서쌍(ordered pair)이라고 하면 두 개씩 순서를 지어 짝을 지어 놓은 것을 말하고 (x, y)와 같은 형식으로 표시한다. 이 개념을 n개의 짝으로 확장하여 $(x_1, x_2, \cdots, x_n)$으로 묶어 놓은 것이 순서짝이다. 파이썬에서 순서짝은 원소의 개수가 정해져 있고 그 값을 바꿀 수 없는 자료이다. 파이썬에서 tuple은 둥근 괄호로 묶고 쉼표로 구별하여 원소를 나열한다. 간단한 예로 3차원 공간 상의 좌표 또는 벡터를 순서짝으로 표현하면 다음과 같다.

```
>>> v = (3, 4, 5)
```

이때 순서짝의 각 원소를 사용하기 위해서는 원소 번호를 이용한다. 원소 번호는 0부터 시작한다. 꺽쇠 괄호 안에 번호를 지정하는 형식을 사용한다.

```
>>> v[0]
3
>>> v[1]
4
```

순서짝은 변경불가능한 자료형이다. 한번 순서짝을 만들면 그 원소를 다른 것으로 바꿀 수 없다. 원소를 다른 것으로 바꾸려고 하면 오류가 발생한다.

```
>>> v[0] = 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

위와 같이 순서짝은 원소 할당(item assignment)을 지원하지 않는다는 메시지를 볼 수 있다. 이런 성격을 가지므로 변경불가능(immutable)이라고 하는 것이다.

나열형 자료에 대해서 이야기할 때 중요한 것 중의 하나는 그 원소들의 종류, 즉 자료형이 무엇인가 하는 것이다. 파이썬의 tuple에서는 원소의 자료형을 제한하지 않는

다. 서로 이질적인 자료들을 하나의 tuple로 묶는 것이 가능하며 오히려 서로 이질적인 값들을 묶는 것이 tuple의 전형적인 용도라고 할 수 있다.

```
>>> person1 = ('Guido', 'van Rossum', 1956, 'https://gvanrossum.github.io/')
>>> person2 = ('Larry', 'Wall', 1954, 'http://www.wall.org/~larry/')
```

위 예의 경우에 0번 원소는 이름, 1번 원소는 성, 2번 원소는 태어난 해, 3번 원소는 홈페이지 주소이다. 일반화하여 말하자면, 특정한 위치(번호)에 특정한 유형의 정보를 배치하는 방식으로 사용하는 것이다.

더 나아가 순서짝의 원소는 복잡한 자료형이 될 수도 있다. 순서짝의 원소가 또다른 순서짝이 될 수도 있다. 이런 경우를 특히 중첩된(nested) 구조라고 하며 널리 사용된다. 다음은 $a \oplus b$라는 연산을 $(\oplus, a, b)$의 순서짝을 이용하여 표현하였을 때 $x/y + y/x$라는 식을 순서짝을 이용해 표현한 것이다.

```
expr = ("+", ("/", "x", "y"), ("/", "y", "x"))
```

중첩된 구조에 관해서는 뒤에서 자세히 다룰 것이다.

목록 list

목록(list)은 말 그대로 원소들의 목록을 표현하기 위한 자료형이다. 프로그래밍에서는 '목록'이라는 용어보다 '리스트'라는 용어를 자주 접할 수 있다. 리스트라는 용어는 일반적인 의미에서 순서대로 나열된 원소의 목록을 의미하기 보다는 컴퓨터에서 이러한 목록을 구현하는 특정한 방법을 지칭하기 위해 주로 사용된다. 특히 연결 리스트(linked list)라는 특정한 자료 구조를 지칭하는 경우가 많다. 하지만 파이썬의 list는 연결 리스트가 아니라 동적 배열(dynamic array)이라는 자료 구조에 해당한다. 이어지는 내용에서 오해의 여지가 없다면 파이썬의 list를 리스트라고 지칭할 것이다.

파이썬에서 리스트를 만들 때에는 꺾쇠 괄호로 묶고 쉼표로 구분하여 원소를 나열한다. 각 원소는 0번부터 시작하는 번호를 통해 접근할 수 있고 특정 원소를 다른 것으로 바꿀 수도 있다.

```
>>> a = [1, 2, 3]
>>> a[0]
1
>>> a[1]
2
>> a[0] = 100
>>> a
[100, 2, 3]
```

이와 같이 원소를 바꿀 수 있다는 의미에서 list를 변경가능(mutable)한 자료형이라고 한다. 이것이 변경불가능한 자료형인 tuple과의 차이다.

개념적으로도 순서짝과 목록은 차이가 있다. 순서짝이 서로 이질적인 정보를 하나로 묶는 것이라면 배열은 서로 동질적인 정보를 나열하는 것이다. 파이썬의 tuple과 list는 원소의 자료형에 대한 제약이 없으므로 자유롭게 사용할 수 있다. 다음과 같이 사용하는 것이 가능하다.

```
>>> person = ['Guido', 'van Rossum', 1956, 'https://gvanrossum.github.io/']
```

그러나 배열은 개념적으로 동일한 종류의 자료를 순서대로 나열하기 위한 것이라고 할 수 있으며 파이썬의 리스트도 주로 그러한 목적으로 사용된다. 동일한 유형의 자료를 모아놓았을 때 우리는 주로 각 원소에 대한 동일한 처리를 반복하려는 경우가 많다. 이 때문에 리스트는 반복문과 함께 널리 사용된다.

```
xs = [3, 7, 2, 1, 9, 5, 1, 2]
for x in xs:
    print('*' * x)
```

나열형 자료 다루기

나열형 자료에 해당하는 tuple과 list는 전자는 둥근 괄호, 후자는 꺾쇠 괄호를 쓴다는 점 이외에 겉보기에 동일해 보이지만 변경가능성에서 차이가 있으며 변경가능성의 차이는 많은 기능적인 차이를 낳는다. 변경가능한 리스트가 더 많은 기능을 제공한다. 리스트의 사용법에 대해서는 뒤에 이어서 자세히 다룰 것이다.

나열형 자료에 대해 사용할 수 있는 명령들을 요약하면 다음 표와 같다. 다음은 나열형 자료에 공통적으로 사용할 수 있는 명령들로 객체의 내용을 변경하지 않는 것들이다. 이 명령들은 문자열 str과 바이트열 bytes에 대해서도 사용할 수 있다.

a[i]	번호로 찾아보기
a[i:j]	부분 목록
a[i:j:k]	부분 목록
for x in a: $f(x)$	a의 원소들을 하나씩 함수 f로 처리
a + b	이어붙이기
a * n	a를 n번 반복해서 이어붙이기
len(a)	길이. 원소의 개수
sum(a)	원소의 합
min(a)	최소 원소
max(a)	최대 원소
a.index(x)	a에서 x 원소가 처음 나타나는 위치
a.count(x)	a에 들어 있는 x 원소의 개수
sorted(a)	a의 원소들을 정렬하여 만든 리스트
reversed(a)	a의 원소들을 거꾸로 읽어주는 객체
enumerate(a)	a의 원소에 번호를 붙인 순서짝의 목록 객체
for i, x in enumerate(a):	i는 원소의 번호, x는 원소의 값
zip(a, b, $\cdots$)	$a, b, \cdots$의 같은 번호의 원소 끼리 묶은 순서짝의 목록 객체
for x, y in zip(a, b):	x는 a의 원소, y는 b의 원소.
a.copy()	a의 복제 (list에서만 사용 가능)

다음의 객체의 내용을 변경하는 명령들은 리스트에서만 사용할 수 있다.

a[i] = x	원소 할당
a.pop()	a의 마지막 원소 빼내기
a.sort()	a의 원소들을 정렬
a.reverse()	a의 원소들의 순서를 거꾸로
a.append(x)	a에 x를 추가
a.remove(x)	a에서 첫번째로 등장하는 x 원소를 제거
a.extend(b)	a에 b를 이어붙여서 확장
a.clear()	a의 원소들을 모두 제거
a.insert(i, x)	a의 i번째 원소 앞에 x를 삽입

여기에서 명령이라는 모호한 용어를 사용하였는데 크게 연산(operation), 내장함수(built-in function), 메소드(method)로 나눌 수 있다. 위 표에서 $a[i]$, $a+b$ 등은 연산에 해당하고, len(a)와 같은 것은 내장함수, a.append(x)와 같은 것은 메소드에 해당한다.

4.2.2 리스트 만들기

리스트와 원소

리스트(list)는 순서대로 나열된 원소로 이루어진 자료구조이다. 다음처럼 리스트 a를 만들 수 있다. 꺽쇠 괄호를 원소들을 묶고 쉼표로 원소들을 구별한다.

```
>>> a = [4, 8, 1, 5, 1, 6, 2, 3, 4, 2]
```

리스트는 순서가 있는 목록이므로 그 원소들에는 번호가 있다. 그 번호 인덱스(index)라고 한다. 리스트에 들어 있는 원소를 지칭하려면 리스트 이름 다음에 꺽쇠 괄호 안에 해당 리스트의 인덱스(index)를 써 주면 된다. 번호는 0부터 시작한다. 음수는 뒤에서부터의 번호를 뜻한다.

```
>>> a = [4, 8, 1, 5, 1, 6, 2, 3, 4, 2]
>>> a[0]
4
>>> a[3]
5
>>> a[-1]
2
>>> a[-3]
3
```

원소 개수

리스트를 만들 때 그리고 주어진 리스트를 사용할 때 구성 원소의 개수는 가장 먼저 파악해야 할 정보이다. 리스트의 길이, 즉 원소의 개수는 내장 함수 len()를 이용하여 얻을 수 있다.

```
>>> a = [4, 8, 1, 5, 1, 6, 2, 3, 4, 2]
>>> len(a)
10
```

원소 변경

리스트는 원소를 변경하는 것을 허락한다. 기존의 리스트의 특정한 위치에 있는 값을 바꿀 수 있다.

```
>>> a = [1, 3, 5, 7]
>>> a[0] = 3
>>> a
[3, 3, 5, 7]
```

원소 추가

리스트에 새로운 원소를 추가하려면 append()라는 메소드를 이용하면 된다.

```
>>> a
[0, 1, 2, 3, 4, 5]
>>> a.append(6)
>>> a
[0, 1, 2, 3, 4, 5, 6]
>>> a.append(100)
>>> a
[0, 1, 2, 3, 4, 5, 6, 100]
```

반복문의 형태로 여러 개의 원소를 리스트에 추가할 수도 있다. 기본적으로는 다음과 같은 형식이 된다.

```
>>> a = []
>>> for i in xrange(0, 10):
        a.append(i)
>>> a
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

반복문을 돌리기 전에 '`a = []`'로 빈 리스트를 만들어주어야 한다는 것을 잊지 말자. 이렇게 변수를 초기화하는 것을 잊으면 에러가 난다.

부분 리스트

주어진 리스트의 일부분을 잘라내어 새로운 리스트를 만들 수도 있다. 영어로는 'slicing'이라고 한다. 꺽쇠괄호 안에 잘라낼 범위를 지정해 주면 된다. 잘라낼 부분의 시작과 끝 인덱스를 콜론으로 구별하여 지정해준다.

```
>>> weekdays = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"]
>>> weekdays[0:5]
['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
```

범위 중 시작 또는 끝 인덱스를 비워놓으면 원래 리스트의 처음 또는 마지막을 의미한다.

```
>>> weekdays[:5]
['Mon', 'Tue', 'Wed', 'Thu', 'Fri']
>>> weekdays[-2:]
['Sat', 'Sun']
```

잘라내기에서 세 개의 값을 지정하기도 하는 데 이때 마지막 숫자는 인덱스의 증가 단위를 뜻한다.

```
>>> weekdays[0:5:2]
['Mon', 'Wed', 'Fri']
```

이어붙이기

리스트 끼리 덧셈이 가능하다. 두 리스트를 이어붙여 새로운 리스트를 만들 수 있다.

```
>>> a = [1, 2, 3, 4]
>>> b = [5, 6, 7, 8]
>>> a + b
[1, 2, 3, 4, 5, 6, 7, 8]
```

리스트에 원소를 더할 수는 없다.

```
>>> a = [1, 2, 3, 4]
>>> a + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
```

하나의 원소를 추가하고 싶다면 원소 하나인 리스트를 만들어 더하면 된다.

```
>>> a + [3]
[1, 2, 3, 4, 3]
```

반복하여 이어붙이기

리스트에 정수를 곱하면 해당 리스트가 주어진 정수만큼 반복이 되어 연결된 리스트가 만들어진다.

```
>>> a = [1, 2, 3]
>>> 3 * a
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

별칭과 복제

리스트를 다룰 때 주의할 점 중의 하나는 할당 기호를 사용할 때이다. 우선 단순 자료형인 정수형의 경우를 보자.

```
>>> x = 3
>>> y = x
>>> x, y
(3, 3)
>>> y = 4
>>> x, y
(3, 4)
```

y=x라는 할당문은 x의 값을 새로운 변수 y에 할당하므로 x와 y는 값은 같지만 서로 다른 변수이다. y를 변경해도 x는 변하지 않는다. 리스트는 그렇게 작동하지 않는다.

```
>>> a = [1, 2, 3, 4]
>>> b = a
```

이때 b=a라는 할당문은 a라는 이름으로 지칭되던 리스트에 또다른 이름 b를 붙여줄 뿐이다. 새로운 리스트가 생성되는 것이 아니다. 하나의 리스트에 이름만 두 개가 있는 것이다. 따라서 다음처럼 작동한다.

```
>>> b[0] = 3
>>> b
[3, 2, 3, 4]
>>> a
[3, 2, 3, 4]
```

이러한 것을 흔히 별칭(alias)라고 한다.

리스트를 복사할 때 잘라내기 기법을 사용하기도 한다. 다음에서 리스트 b는 a의 값을 그대로 복사한 것이므로 등가이지만 동일한 리스트는 아니다.

```
>>> a = [1, 2, 3, 4]
>>> b = a[:]
>>> a == b
True
>>> a is b
False
```

4.2.3 원소들의 순서

최소와 최대

내장함수 min()과 max()를 이용하여 최대와 최소를 구할 수 있다.

```
>>> a = [4, 8, 1, 5, 1, 6, 2, 3, 4, 2]
>>> max(a)
8
>>> min(a)
1
```

숫자가 아닌 경우에는 먼저와 나중의 개념으로 생각할 수 있다. 정렬을 했을 때 가장 먼저 나오는 것이 최소, 가장 나중에 나오는 것이 최대이다.

```
>>> animals = ['dog', 'cat', 'dolphin', 'bat', 'zebra', 'bear']
>>> min(animals)
'bat'
>>> max(animals)
'zebra'
```

정렬

내장 함수 sorted()를 사용하면 정렬이 된 새로운 리스트를 만들어 준다.

```
>>> b = [3, 5, 2, 6, 9, 7]
>>> sorted(b)
[2, 3, 5, 6, 7, 9]
>>> b
[3, 5, 2, 6, 9, 7]
```

리스트의 정렬 메소드 sort()를 사용할 때에는 주의해야 한다. 리스트 자체를 변경한다.

```
>>> b = [3, 5, 2, 6, 9, 7]
>>> b.sort()
>>> b
[2, 3, 5, 6, 7, 9]
```

정렬 메소드는 정렬된 리스트를 만들어 돌려주는 함수의 형태가 아니다. 정렬 메소드는 해당 리스트 자체를 변화시키므로 원래의 리스트를 보존하고 싶다면 그 내용을 새로운 리스트에 복제한 후에 정렬해야한다.

거꾸로

원소들의 순서를 뒤집을 수도 있다. reverse()라는 메소드를 이용한다. 리스트 자체가 변경된다는 점에 주의해야 한다.

```
>>> b
[2, 3, 5, 6, 7, 9]
>>> b.reverse()
>>> b
[9, 7, 6, 5, 3, 2]
```

이와 달리 내장 함수 reversed()를 사용하면 원래 리스트는 그대로 둔 채 순서를 뒤집을 수 있다.

```
>>> b = [3, 5, 2, 6, 9, 7]
>>> reversed(b)
<list_reverseiterator object at 0x1064f2da0>
```

다만 순서가 뒤집어진 리스트를 새로 만드는 것이 아니라 리스트를 거꾸로 읽어 주는 반복자(iterator)가 만들어진 것이다. 다음과 같이 반복문에서 사용할 수 있다.

```
>>> for i in reversed(b): print(i)
...
7
9
6
2
5
3
```

4.2.4 스택과 큐

리스트에서 원소를 하나씩 뺄 수도 있다. pop()을 이용하면 원소가 순서대로 하나씩 빠져나온다.

```
>>> a
[0, 1, 2, 3, 4, 5, 6, 100]
>>> a.pop()
100
>>> a
[0, 1, 2, 3, 4, 5, 6]
>>> a.pop()
6
>>> a
[0, 1, 2, 3, 4, 5]
```

이 메소드가 작동하는 방법을 잘 살펴보라. 메소드의 리턴값으로 리스트의 마지막 원소의 값을 돌려주면서 동시에 리스트 자체도 변화시킨다.

자료구조 중 스택(stack)과 큐(queue)이라는 개념은 프로그래밍을 배우는 사람이라면 상식 수준에서라도 알고 있는 것이 좋다. 스택은 LIFO (Last In First Out), 즉 마지막에 들어온 것이 먼저 빠져나가는 구조를 말하며 흔히 쌓아놓은 접시을 사용하는 것에 비유된다. 큐는 말그대로 '줄서기'에 해당되는 것으로 FIFO (First In Fist Out)의 자료구조이다. 까페테리아에 가서 줄을 서면 먼저 줄을 선 사람이 먼저 밥을 받는 것에 비유된다.

파이썬의 list에서 append(x)와 pop()을 이용하면 스택처럼 작동하게 할 수 있고 pop(0)을 이용하면 원소를 앞에서부터 빼내어 큐처럼 작동하게 할 수 있다. 파이썬의 list는 객체의 참조(reference)를 원소로 하는 동적 배열(dynamic array)인데 이 자료구조에서 마지막 원소를 빼내는 pop()이나 끝부분에 새로운 원소를 추가하는 append(x)는 효율적이지만 맨 앞에 원소를 빼는 pop(0)은 비효율적이다. 이 때문에 list를 큐로 사용하는 경우에는 효율성이 떨어지므로 필요하다면 collections 모듈의 deque를 사용하는 것이 좋다. 중간에 원소를 끼워넣는 insert($\{i, x\}$)와 원소를 제거하는 remove(x)도 비효율적으므로 이러한 연산이 주로 이루어지는 자료를 처리하는 경우에 list를 사용해서는 안 된다.

4.2.5 반복문

리스트는 주로 반복문과 함께 사용된다. 리스트와 반복문을 함께 사용하는 유형은 크게 세 가지로 나누어 생각해 볼 수 있다.

대응

한 리스트의 각 원소에 대해 개별적으로 어떤 함수를 적용하여 결과를 내는 것이 리스트와 반복문의 가장 전형적인 사용 유형이다. 이때 결과를 리스트로 만들 수도 있다. 원래 리스트와 결과 리스트가 길이가 같고 원소들이 순서대로 대응된다. 수학에서 사상(mapping) 개념에 해당되며 함수형 프로그래밍에서는 맵(map)이라는 용어를 사용한다.

```
a = [20, 16, 5, 12, 19, 67, 9, 30]

b = []
for x in a:
    y = x + 1
    b.append(y)
```

축약

한 리스트에 들어 있는 원소들을 함께 묶어 어떤 연산을 적용하는 것으로 대개 축약(reduce)하거나 집계(aggregate)하는 결과를 가져온다. 함수형 프로그래밍에서는 리스트를 접는다(fold)는 표현을 사용하기도 한다. 가장 전형적인 예는 숫자 리스트에 들어 있는 원소들의 합을 구하는 것이다.

```
a = [20, 16, 5, 12, 19, 67, 9, 30]

total = 0
for x in a:
    total += x
```

선택

주어진 리스트에서 특정한 조건을 만족하는 원소만 골라낼 수 있다. 골라낸 원소를 새로운 리스트에 담으면 특정 조건을 만족하는 원소로 이루어진 리스트를 얻을 수 있다. 함수형 프로그래밍에서는 원소를 걸러낸다(filter)는 표현을 사용한다. 간단한 예시를 하면 다음과 같다. 주어진 리스트에서 10보다 큰 수만 골라내어 새로운 리스트로 만들어 준다.

```
a = [20, 16, 5, 12, 19, 67, 9, 30]

b = []
for x in a:
    if x > 10 : b.append(x)
```

4.3 사전형 자료

파이썬에서는 사전(dictionary)라고 하는 자료형 dict를 제공한다. 이와 동일한 의미로 맵(map) 또는 연상 배열(associative array)이라는 용어가 주로 사용된다. 수학에서 사상(map)이라는 개념을 통해 이해할 수도 있다. 사상(map)은 수학에서는 함수(function)와 동일한 의미로 사용되거나 일대일 대응, 일대다 대응과 같은 대응의 개념을 지칭하기 위해 사용된다.

사전형 자료는 열쇠(key)와 값(value)의 짝으로 이루어진 자료형이다. 열쇠를 보고 값을 찾아갈 수 있는 구조이다. 하나의 사전에 동일한 열쇠가 중복될 수 없다. 다르게 말하면 하나의 열쇠에 서로 다른 두 개의 값이 연결될 수 없다. 하지만 서로 다른 열쇠에 같은 값이 연결되는 데에는 아무런 문제가 없다. 즉, 일대다 대응에 해당한다.

4.3.1 사전 만들기

파이썬의 dict는 중괄호로 둘러싸고 쉼표로 원소를 구분하며 하나의 원소는 콜론으로 구분된 열쇠와 값으로 구성된다. 예를 들면 다음과 같다. 사전을 초기화하면서 원소들도 넣고 싶다면 다음과 같이 열쇠와 값을 콜론(:)으로 짝을 지어 쉼표로 구분하여 나열하면 된다.

```
>>> brix = {"grape" : 12, "apple" : 10, "lemon" : 6, "pineapple" : 14}
```

어떻게 초기화되었는지 확인해 보자.

```
>>> brix
{'grape': 12, 'apple': 10, 'lemon': 6, 'pineapple': 14}
```

이때 파이썬 3.7 이전 버전에서는 입력한 순서가 그대로 유지되지 않는다는 점에 주의하자. 이전 버전에서의 예를 보면 다음과 같다.

```
>>> person = { "name" : "guido", "birthyear" : 1956, "sex" : 'male' }
>>> person
{'name': 'guido', 'sex': 'male', 'birthyear': 1956}
```

입력한 내용과 파이썬이 돌려준 결과에 차이가 있다. 사전 자료는 열쇠를 이용해서 원소를 찾아가므로 원소의 순서는 사용자가 원하는 대로 고정될 필요는 없다.

4.3.2 열쇠와 값

열쇠로 값에 접근하기

목록에서는 번호를 이용하여 원소 값에 접근할 수 있었다. 사전에서는 원하는 원소의 열쇠를 이용하여 원소의 값에 접근할 수 있다. 사전의 이름 뒤에 꺾쇠 괄호를 치고 열쇠를 넣어 주면 된다.

```
>>> person['name']
'guido'
>>> person['name'] = 'Guido'
>>> person
{'name': 'Guido', 'sex': 'male', 'birthyear': 1956}
```

열쇠가 있나?

이것은 해당 열쇠가 존재하는지 여부를 알아볼 때 사용한다.

```
>>> 'john' in grade
True
>>> 'peter' in grade
False
```

열쇠가 없으면?

열쇠를 이용해 그것에 대응하는 값을 얻을 수 있다. 리스트에서 인덱스를 이용하여 값을 찾아내듯이 사전에서는 열쇠를 이용해 값을 얻는다.

```
>>> brix["apple"]
10
```

이 방법 대신에 열쇠를 사용해서 값을 리턴하는 `get()` 메소드를 이용할 수도 있다.

```
>>> brix.get("apple")
10
```

리스트와 달리 이런 메소드를 제공하는 데에 이유가 있다. 사전을 좀더 사용하다 보면 `get()` 메소드의 필요성을 느끼게 될 것이다. 일단 두 방법이 차이나 나는 점을 살펴보자. 사전 자료는 어떤 열쇠가 사전 안에 있는지 없는지 모르는 상태에서 그 열쇠를 사용해야 하는 경우가 종종 발생한다. 그런데 없는 열쇠를 사용하면 다음과 같은 에러가 발생한다.

```
>>> brix["mango"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'mango'
```

리스트에서 없는 인덱스를 호출할 때 에러가 발생하는 것과 마찬가지이다.

```
>>> l = [5, 4, 7]
>>> l[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

리스트는 원소를 순서대로 번호를 매겨서 가지고 있기 때문에 이와 같이 없는 인덱스를 사용할 때 에러가 나는 것이 당연하게 생각되고 프로그래밍을 할 때 이것 때문에 골치 아픈 일이 생기지는 않는다. 하지만 사전의 경우에는 원소가 어떤 열쇠가 있는지 없는지 자체를 미리 파악할 수 없는 일이 자주 생긴다. get 메소드를 사용할 때 부가적인 인자를 하나 더 사용하면 이런 에러를 막을 수 있다.

```
>>> brix.get('mango', 0)
0
```

위와 같이 열쇠가 없을 때 리턴할 기본값을 설정해 놓을 수 있다.

원소 추가

원소를 추가는 다음처럼 간단히 할 수 있다. 눈여겨 볼 것인 새로 추가한 원소가 순서대로 뒤쪽에 들어가는 게 아니라는 점이다.

```
>>> brix["mango"] = 6
>>> brix
{'grape': 12, 'apple': 10, 'lemon': 6, 'pineapple': 14, 'mango' : 6}
```

원소의 값 변경

주어진 열쇠에 대응되는 값을 변경하는 것도 단순하다.

```
>>> brix["mango"] = 10
>>> brix
{'grape': 12, 'apple': 10, 'lemon': 6, 'pineapple': 14, 'mango' : 10}
```

원소 제거

사전의 원소를 지울 때는 다음처럼 특수한 예약어를 사용한다.

```
>>> del brix["lemon"]
>>> brix
{'grape': 12, 'apple': 10, 'pineapple': 14, 'mango': 10}
```

다음과 같이 clear() 메소드을 이용하여 모든 원소를 지우고 빈 사전으로 만들 수도 있다.

```
>>> brix.clear()
>>> brix
{}
```

연습 4.1 예제 파일 sales.txt는 어느 자동차 판매사의 실적 데이터이다. 첫번째 컬럼에는 직원의 이름, 두번째 컬럼에는 연도, 세번째 컬럼에는 판매액(1000 USD)가 들어있다.

```
john        2006        500
john        2007        1000
john        2008        780
mary        2006        70
mary        2007        150
mary        2008        550
mark        2006        1700
mark        2007        1300
mark        2008        500
```

파이썬의 사전을 이용하여 직원별 총 판매액과 평균 판매액을 계산하라. ■

연습 4.2 예제 파일 emp.dat는 어느 패스트푸드점의 직원 데이터이다. 이름, 시급, 주당근무시간, 나이, 성별이 기록되어 있다.

```
AIDAN   7       19      22
AMELIA  10.5    16      22
NOAH    9       24      25
ISABELLA        8.5     17      25
LIAM    9       23      23
AVA     11      24      25
CAYDEN  12      21      28
SOPHIA  6.5     20      20
ETHAN   9.5     2       30
OLIVIA  7       1       18
JACKSON 11.5    2       29
MADELINE        9.5     2       36
LANDON  9.5     11      38
LILY    8       14      42
JACOB   12      7       37
ABIGAIL 9.5     12      31
CALEB   10.5    7       49
CHLOE   10      17      60
LUCAS   12.5    13      51
EMMA    13.5    10      48
```

가장 어린 작원은 18세, 가장 나이가 많은 직원은 60세이다. 직원의 나이에 따라 10대, 20대, 30대, 40대, 50대, 60대로 나누고 각 집단별 평균 시급을 계산하여라. ■

4.3.3 반복문

다음과 같은 사전을 만들어 보자. 사람의 이름을 열쇠로 하여 나이를 알려주는 사전이다.

```
>>> age = {'john' : 29, 'mary': 45, 'mark' : 39, 'alice' : 17}
```

사전은 원소들의 순서를 사용자가 마음대로 제어할 수 없다. 파이썬 3.7 이전 버전에서는 위와 같은 순서로 초기화를 했는데 어떤 원소들이 들어있는지 확인해 보면 우리가 입력한 것과는 다른 순서로 바뀌어 있음을 알 수 있다.

```
>>> age
{'john': 29, 'mary': 45, 'alice': 17, 'mark': 39}
```

우리가 입력 순서가 유지되지 않는다고 해도 사전에 원소들에 순서가 없는 것은 아니다. 제어할 수 없을 뿐이다. 사전을 `for` 루프로 돌리면 위에서 보여준 순서대로 나열된다. `for ... in ...` 구문에서 열쇠가 주어진다는 점에 유의하자.

```
for key in age:
    print(key, age[key])
```

위에서 알 수 있듯이 x는 열쇠가 된다. 다음과 동일하게 작동한다.

```
for name in age.keys():
    print(name, age[name])
```

열쇠들

다음과 같이 keys() 메소드를 이용하면 열쇠의 목록을 얻을 수 있다.

```
>>> age.keys()
dict_keys(['john', 'mary', 'alice', 'mark'])
```

앞서 보았듯이 사전을 looping한다는 것은 정확하게는 사전의 열쇠들을 looping하는 것이다.

```
for name in age.keys():
    print(name, age[name])
```

값들

다음과 같이 values() 메소드를의 이용하연 값(value)의 목록만 얻을 수 있다.

```
>>> age.values()
dict_values([29, 45, 17, 39])
```

이것을 이용하면 값만 뽑아서 무언가 할 수 있다. 예를 들어, 네 사람의 나이의 평균을 구하고 싶다면:

```
>>> sum(age.values()) / len(age)
32
```

원소들

사전에 대해 items() 메소드를 사용하면 (key, value) 순서짝을 원소로 하는 목록을 얻을 수 있다.

```
>>> age.items()
dict_items([('john', 29), ('mary', 45), ('alice', 17), ('mark', 39)])
```

4.3.4 정렬하기

열쇠로 정렬하기

다음 사전을 보자. 이 사전을 이름 순으로 정렬하려고 한다.

```
>>> age
{'john': 29, 'mary': 45, 'alice': 17, 'mark': 39}
>>> for name in age:
...         print(name, age[name])
...
john 29
mary 45
alice 17
mark 39
```

열쇠로 사전을 정렬하려면 우선 열쇠의 정렬된 목록을 얻어야 한다. 사전에 내장함수 sorted()를 적용하면 정렬된 열쇠의 리스트를 얻을 수 있다.

```
>>> names = sorted(age)
>>> names
['alice', 'john', 'mark', 'mary']
```

이제 정렬된 열쇠 리스트를 이용하여 루프를 돌리면 된다. 주어진 age라는 사전의 내부 원소의 순서가 어떻든 관계없이 names라는 열쇠 리스트의 원소 순서에 따라 루프가 돌아가게 된다.

```
>>> for name in names:
...         print(name, age[name])
...
alice 17
john 29
mark 39
mary 45
```

값으로 정렬하기

값(value)으로 정렬하는 것은 열쇠로 정렬하는 것에 비해 복잡하다. 열쇠를 기준으로 정렬할 때는 사전에서 열쇠만 뽑아내서 정렬하고 그 정렬된 열쇠를 이용하여 값을 불러내면 그만이었다. 하지만, 사전에서 값만 뽑아내고 정렬을 하게 되면 그 값에 해당하는 열쇠를 찾아낼 방법이 없어진다. 예를 들어, 다음과 같이 값들의 목록을 정렬할 수는 있지만 그것으로 끝이다.

```
>>> a = age.values()
>>> sorted(a)
[17, 29, 39, 45]
```

우리가 필요로 하는 것은 값을 기준으로 열쇠를 포함한 원소를 정렬하는 것이다. 이를 위해서는 sorted() 함수 안에 정렬 기준을 정의해야 한다. 이때 흔히 `lambda`가 사용된다. 우선 다음과 같은 방식으로 할 수 있다는 사실만 알아두자.

```
sorted_items =  sorted(age.items(), key=lambda x: x[1])
for key, val in sorted_items:
    print(key, val)
```

연습 4.3 앞서 사용했던 예제 파일 `emp.dat`을 다시 보자. 이름, 시급, 주당근무시간, 나이, 성별이 기록되어 있다.

```
AIDAN   3.5     19      22      M
AMELIA  5.25    16      22      F
NOAH    4.5     24      25      M
ISABELLA        4.25    17      25      F
LIAM    4.5     23      23      M
AVA     5.5     24      25      F
```

각 직원별로 주급을 계산하고 주급이 높은 순으로 정렬하여 이름과 주급을 출력하여라. 단, 주급은 시급과 주당근무시간의 곱으로 계산한다. ■

4.3.5 사전에 사용할 수 있는 명령들

사전과 함께 사용할 수 있는 구문, 내장함수, 메소드에는 다음과 같은 것들이 있다.

m[k]	열쇠로 찾아보기
k in m	m에 열쇠 k의 존재 여부
m.get(k)	$m[k]$. 만약 k 열쇠가 없으면 None
m.get(k, d)	$m[k]$. 만약 k 열쇠가 없으면 d 값
len(m)	길이. 원소의 개수
sum(m)	열쇠들의 합. 열쇠가 숫자일 때만 사용 가능
min(m)	열쇠들의 최소
max(m)	열쇠들의 최대
sorted(m)	열쇠들을 정렬한 리스트
m.items()	m의 열쇠와 값의 순서짝들
for k, v in m.items():	
m.keys()	m의 원소의 열쇠들
for k in m.keys():	
m.values()	m의 원소의 값들
from v in m.values():	
m.copy()	m의 복제본
m[k] = v	열쇠 k에 대응되는 값으로 v를 할당
m.update(m^*)	사전형 자료 m^*를 이용해 m의 원소들을 갱신
m.popitem()	한 원소 빼내기
m.pop(k)	m에서 열쇠 k인 항목 빼내면서 $m[k]$를 돌려 줌
m.pop(k, d)	열쇠 k가 없을 때 d를 돌려 줌.
m.fromkeys(a)	열쇠들의 목록 a를 이용하여 새로운 사전 생성
m.clear()	m의 원소를 모두 제거

4.4 집합형 자료

집합은 수학에서 이야기하는 것과 같이 원소들 사이에 순서가 없으며 동일한 원소가 중복되지 않은 자료를 말한다. 파이썬에서는 집합형 자료로 set과 frozenset를 제공하고 있다. 파이썬에서 set는 중괄호로 둘러싸서 표시한다.

```
>>> {0, 1, 2, 3}
{0, 1, 2, 3}
```

집합은 중복된 원소를 저절로 제거하여 하나로 만들어 준다.

```
>>> {0, 1, 1,  2, 1, 3, 2}
{0, 1, 2, 3}
```

빈 집합은 다음과 같이 생성한다.

```
>>> set()
```

집합에 원소를 추가할 때에는 `add`(x) 메소드를 이용한다.

```
>>> s = {1, 2, 3}
>>> s.add(4)
```

두 집합의 교집합과 합집합을 계산할 수도 있다.

```
>>> s = {1, 2, 3}
>>> u = {2, 3, 4, 5}
>>> s.intersection(u)
{2, 3}
>>> s.union(u)
{1, 2, 3, 4, 5}
```

4.5 중첩된 구조

4.5.1 중첩된 리스트

리스트의 원소에 들어갈 수 있는 자료형에는 제약이 없다. 리스트 안에 리스트가 들어갈 수도 있다. 리스트 안에 리스트가 들어있는 것을 중첩된 리스트(nested list)라고 한다. 중첩된 리스트는 매우 다양한 형태의 데이터를 표현할 수 있는 가능성을 제공한다.

행렬과 다차원 배열

가장 단순한 형태는 동일한 길이의 리스트들을 모아서 만든 리스트일 것이다. 수학에서 행렬(matrix) 또는 배열(array)이라고 하는 형태를 중첩 리스트를 이용하여 표현할 수 있다. 행렬은 2차원 배열이다. 예들 들어, 다음과 같은 중첩 리스트는

```
mat =  [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

다음과 같은 행렬을 표현하는 데에 사용할 수 있다.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

원소에 접근하기 위해서는 꺾쇠 괄호 안에 번호를 두 번 사용하면 된다. 즉, `mat`[i][j]와 같은 형식을 사용하고 i는 행 번호, j는 열 번호로 생각할 수 있다.

```
>>> mat[0]
[1, 2, 3]
>>> mat[0][0]
1
```

중첩이 여러 번 되는 것도 가능하므로 다차원 배열을 표현할 수도 있다. 예를 들어, 리스트를 3중으로 겹치면 3차원 배열을 표현할 수 있다.

```
>>> arr = [[[1, 2],[3, 4], [5, 6]], [[7, 8], [9, 10], [11, 12]]]
>>> arr[0]
[[1, 2], [3, 4], [5, 6]]
>>> arr[0][0]
[1, 2]
>>> arr[0][0][0]
1
```

테이블

스프레드시트(spreadsheet) 프로그램을 비롯하여 데이터를 처리하는 많은 소프트웨어서 사용되는 테이블(table), 즉 표는 행과 열로 이루어진 직사각형 형태로 자료를 정리하여 저장하고 처리하는 데에 사용되는 형식이다. 테이블 데이터는 직사각형이므로 행렬과 마찬가지로 리스트의 리스트로 표현할 수 있다. 예를 들어, 환자의 이름과 나이로 이루어진 데이터를 다음처럼 중첩된 리스트로 표현할 수 있다.

```
>>> patients = [['john', 23], ['mary', 18], ['mark', 39]]
```

이때 각각의 행(레코드)를 구성하는 필드들이 그 순서와 값이 고정이 되어있어서 개별적으로 변경될 수 없는 것이라면 순서짝의 리스트로 표현할 수도 있다.

```
>>> patients = [('john', 23), ('mary', 18), ('mark', 39)]
```

트리

중첩된 리스트가 반드시 배열과 같은 직사각형 형태일 필요는 없다. 일반적으로 트리(tree)와 같이 생긴 구조를 생각할 수 있다.

```
동물
    파충류
        악어
        뱀
    포유류
        개
        고양이
```

이러한 구조는 적절한 약속을 통해 다음과 같은 중첩된 리스트로 표현가능하다.

```
['동물', ['파충류', '악어', '뱀'], ['포유류', '개', '고양이']]
```

트리 구조를 항상 위와 같은 리스트로 표현해야 한다는 뜻은 아니다. 어떻게 약속하냐에 따라 달라질 수 있다. 위에서는 리스트의 첫번째 원소를 부모, 나머지는 자식으로 하는 약속을 따른 것이다.

대부분의 데이터는 파일의 형태로 제공된다. 파일에 들어있는 데이터를 읽어 어떠한 처리를 하려고 할 때 두가지 서로 다른 접근 방법이 있을 수 있다. 처리해야하는 일이 특정한 단위 내에서만 이루어질 때에는 파일에서 해당 부분을 읽는 즉시 처리할 수 있다. 이때에는 파일의 모든 데이터를 기억하고 있을 필요는 없다. 이에 비해 파일 전체 데이터가 개입하는 처리를 해야하는 경우에는 모든 데이터를 기억하고 있어야하며 이때 리스트의 형태로 데이터를 기억하는 것이 유용하다.

테이블형 테이터는 가장 흔한 형태의 데이터 중의 하나이다. 각각의 행에는 레코드(record)가 저장이 되고 각각의 열로 필드(field)가 구분된다. 레코드는 대개 줄바꿈 문자로 구분이 되고 필드는 공백문자로 구분이 된다. 이렇게 구성된 파일을 읽에서 각각의 필드를 하나의 리스트로 만들어 사용할 수 있다. 기본적으로 다음과 같은 형태로 사용하게 된다.

```
file = open('data.txt')

lst = []
for line in file:
        fields = line.split()
        lst.append(fields[0])
```

□

연습 4.4 다음 `class.txt` 파일에는 한 반 학생의 번호, 성별, 나이, 키(cm), 몸무게(kg)가 기록되어 있다.

```
$ head -6 class.txt
1        M        20        175        77
2        F        25        160        50
3        F        23        155        45
4        F        24        160        46
5        M        24        170        70
6        M        23        170        55
```

나이, 키, 몸무게에 대하여 평균, 최소값, 최대값을 구하라. 이때 각 열(필드)를 하나의 리스트로 만드는 방법으로 프로그래을 작성하라.

연습 4.5 위 `class.txt` 데이터에서 성별에 따라 각각 키와 몸무게의 평균을 구하는 프로그램을 작성하라.

연습 4.6 앞서 다룬 `emp.dat` 데이터를 다시 보자. 이름, 시급, 주당노동시간, 나이, 성별을 담고 있다. 이 데이터 파일을 읽어 중첩된 리스트로 만들어보라. 그 후 다음을 수행하여라.

1. 이름과 주급(시급과 주당노동시간의 곱)을 출력하라.
2. 주급이 가장 높은 직원을 찾아라.

4.6 조건제시법

파이썬에서 가장 사랑받는 도구 중의 하나인 리스트 조건제시법(list comprehension)을 살펴보자. 이것은 기존의 리스트에 기반해서 새로운 리스트를 만들 수 있게 해주는 구문을 말한다. 수학에서 집합을 기존의 집합에 기반해서 정의하는 것을 본 적이 있을 것이다.

$$\{f(x) : x \in N\}$$

파이썬의 리스트 조건제시법은 이것과 형식적으로 동일하다. 다음과 같은 형태로 사용한다. 리스트 안에서 `for` 문을 사용한다.

```
[func(x) for x in lst]
```

예를 들어, 주어진 리스트의 원소들의 제곱으로 이루어진 새로운 리스트를 만들어보자.

```
>>> lst = [1, 2, 3, 4, 5]
>>> [x**2 for x in lst]
[1, 4, 9, 16, 25]
```

조건제시법에서는 `if` 문을 사용할 수도 있다. 다음과 같은 형태로 사용한다.

```
[x for x in lst if func(x)]
```

예를 들어, 주어진 리스트에서 짝수만 골라 새로운 리스트를 만들어보자.

```
>>> lst = [1, 2, 3, 4, 5]
>>> [x for x in lst if x % 2 == 0]
[2, 4]
```

중첩된 조건제시법

중첩된 리스트가 있듯이 조건제시법도 중첩될 수도 있다. 다음은 주어진 행렬에서 행과 열을 바꾸는 방법의 하나이다.

```
>>> mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> [[row[i] for row in mat] for i in [0, 1, 2]]
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

for 루프를 두번 사용하면 다음과 같이 주어진 행렬 lst의 원소들의 모든 가능한 tuple로 이루어진 리스트를 만들 수도 있다.

```
>>> lst = [1, 2, 3, 4, 5]
>>> [(x,y) for x in lst for y in lst]
[(1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (1, 2), (2, 2), (3, 2), (4, 2), (5, 2),
 (1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4),
 (1, 5), (2, 5), (3, 5), (4, 5), (5, 5)]
```

중첩된 리스트를 이용하면 결과를 행렬 형태로 만들 수도 있다.

```
>>> [[(x,y) for x in lst] for y in lst]
[[(1, 1), (2, 1), (3, 1), (4, 1), (5, 1)],
 [(1, 2), (2, 2), (3, 2), (4, 2), (5, 2)],
 [(1, 3), (2, 3), (3, 3), (4, 3), (5, 3)],
 [(1, 4), (2, 4), (3, 4), (4, 4), (5, 4)],
 [(1, 5), (2, 5), (3, 5), (4, 5), (5, 5)]]
```

따라서 이런 것도 할 수 있다.

```
>>> [[x+y for x in lst] for y in lst]
[[2, 3, 4, 5, 6],
 [3, 4, 5, 6, 7],
 [4, 5, 6, 7, 8],
 [5, 6, 7, 8, 9],
 [6, 7, 8, 9, 10]]
```

연습 4.7 주어진 텍스트 파일에 들어 있는 단어의 빈도를 계산하여 빈도순으로 정렬하여 출력하는 프로그램을 작성하여라. 여기서 단어는 공백문자로 구분되는 단위를 뜻한다. 예를 들어, 다음과 같이 작동한다.

```
$ python wordfreq.py alice.txt ¦ head
the         1664
and         780
to         773
a         662
of         596
she         484
said          416
in         401
it         356
was         329
```

4.7 데이터 모음 collections

지금까지 파이썬에서 기본적으로 제공되는 내장 자료구조로 tuple, list, dict, set을 살펴보았다. 리스트와 사전이 주로 널리 사용되는 것이다. 파이썬 표준 라이브러리에는

collections라는 모듈이 존재하며 추가적인 자료구조를 제공한다. 이들을 사용할 때에는 어떤 형태의 작업을 하는 것에 따라 성능(performance)을 고려하여 적절한 것을 선택해야 한다.

지금까지 다룬 내장 자료구조들의 쓰임을 두 가지로 나누어 생각해 볼 수 있는데 우선 다음과 같은 경우이다.

```
person = ("Guido", 1956, "Male")
```

또는

```
person = {"name" : "Guido", "year" : 1956, "sex" : "Male"}
```

여기에서는 어떤 하나의 대상이 가지는 속성 또는 구성 요소를 묶기 위한 목적으로 자료구조를 이용하고 있다. 복잡한 내적 구조를 가지는 대상을 표현하는 방법으로 사용된 것이다. 이 때에는 체계적이면서 이해하고 사용하기 쉽게 표현하는 문제가 중요할 것이다. 파이썬과 같은 언어에서는 이러한 목적은 클래스(class)라는 객체 지향 프로그래밍 기법을 통해 구현하는 것이 더 좋을 때가 많다.

다른 한 용법으로는 동일한 종류의 값들을 모아놓기 위한 목적으로 사용되는 것이다. 예를 들자면 다음과 같은 것이 있을 것이다.

```
programmers = ["Guido", "Larry", "James", "Bjarne"]
```

또는

```
programmers = { "Guido" : "Python" , "Larry" : "Perl" , "James" : "Java", "Biarne" : "C++" }
```

이러한 경우 일반적으로 원소의 수가 고정되어 있지 않으며 아마도 원소의 수가 많을 것이라고 기대된다. 이 때에는 원소를 빠르게 찾고 수정하고 추가하고 제거하는 등의 작업을 얼마나 효율적으로 할 수 있는가가 중요할 것이다. 이렇게 동일한 유형의 값들을 여러 개 모아놓은 경우에 주로 'container', 'collection' 등의 용어를 사용하기도 한다.

4.7.1 이름 붙인 속성들 namedtuple

방금 보았듯이 구조를 가진 데이터를 tuple을 이용하거나 dict를 이용해서 표현할 수 있다. 파이썬의 dict에 해당하는 맵을 사용하여 데이터를 표현하는 다음과 같은 방법은 것은 매우 광범위하게 사용되는 방법의 하나이다.

```
person1 = {"name" : "Guido", "year" : 1956, "sex" : "Male"}
```

이런 데이터가 하나로 끝나는 것이 아니라 여러 개의 서로 다른 값을 가지는 데이터를 다룰 필요가 있으며 그 틀은 똑같이 공유를 하는 것이 일반적이다. 예를 들어, 위에서 name, year, sex라는 틀은 똑같게 유지하며 그 값들은 다양할 수 있다. 그런데 다음과 같이 데이터를 생성하면 어떤 일이 벌어질까?

```
person2 = {"name" : "Guido", "age" : 63, "sex" : "Male"}
```

맵은 무엇이든 표현할 수 있는 가능성을 가지고 있지만 이 예에서 보듯이 어떤 고정된 틀을 프로그램 수준에서 일관성 있게 유지해 줄 수 있는 장치가 없다. 객체 지향 프로그래밍 기법을 이용하면 체계적인 프로그래밍이 가능하다.

다루고자 하는 대상이 매우 단순할 때에는 클래스까지 가는 것이 번거로울 수 있다. 이러한 경우에 namedtuple를 이용할 수 있다. 다음과 같은 방법으로 사용한다.

```
>>> from collections import namedtuple
>>> Programmer = namedtuple('Programmer', ['name', 'year', 'sex'])
>>> p = Programmer("Guido", "1956", "Male")
>>> p
Programmer(name='Guido', year='1956', sex='Male')
>>> p.name
'Guido'
>>> p.year
'1956'
>>> p.sex
'Male'
>>> type(p) is Programmer
True
```

값을 입력하여 데이터를 생성할 때 일정한 틀을 유지하는 장치가 마련되어 있으며 속성 값에 접근할 때에도 p.name과 같이 간결한 표현을 사용할 수 있다는 장점이 있다.

연습 4.8 기하학적 도형과 넓이를 다루는 프로그램을 구현해 보자. 2차원의 점, 원, 직사각형을 namedtuple로 구현하고 넓이를 계산하여 주는 함수를 작성하여라. 다음처럼 사용할 수 있도록 작성하여라.

```
p1 = Point(x=1, y=0)
p2 = Point(x=3, y=4)

c1 = Circle(center=p1, radius=1)
c2 = Circle(center=p2, radius=2)
rect = Rectangle(p1, p2)

s1 = area(c1)                          #-> 3.141592653589793
s2 = area(c2)                          #-> 12.566370614359172
s3 = area(rect)                        #-> 8
```

4.7.2 개수를 셀 때 Counter

데이터에서 반복해서 등장하는 요소가 있을 때 이들의 개수를 세는 작업이 필요할 때가 많다. 예를 들어, 다음과 같은 데이터에서 각 문자의 개수를 세는 작업을 생각해 보자.

```
sample = "acgtaagcccgatttcataaacacagatgccgatcaaaccagattgatcactaggtctgg"
```

파이썬의 사전을 이용하면 빈도를 측정하는 프로그램을 작성할 수 있다. collections 모듈의 Counter를 이용하면 쉽게 이러한 목적을 달성할 수 있다.

```
from collections import Counter

sample = "acgtaagcccgatttcataaacacagatgccgatcaaaccagattgatcactaggtctgg"

cnt = Counter()
for b in sample:
    cnt[b] += 1
```

이 프로그램에서 cnt에 다음과 같은 결과를 얻을 수 있다.

```
>>> cnt
Counter({'a': 20, 'c': 15, 't': 13, 'g': 12})
```

연습 4.9 주어진 텍스트 파일에서 단어의 빈도를 세어 빈도가 높은 단어부터 낮은 단어 순으로 고빈도 단어 n개의 목록을 출력하는 프로그램을 작성하여라. gutenberg.org 프로젝트에서 다운로드 받은 Moby Dick 파일에서 고빈도 단어 10개의 목록을 출력하면 다음과 같다.

```
$ python wordfreq.py moby.txt 10
the                   13851
of                     6638
and                    6000
a                      4549
to                     4529
in                     3904
that                   2692
his                    2428
I                      1723
with                   1695
```

4.7.3 양쪽 끝에서 빠르게 빼고 더하는 deque

파이썬 리스트를 이야기하면서 스택(stack)과 큐(queue)의 개념을 배웠다. 여기에서는 흔히 스택과 큐를 합쳐놓은 것이라고 하는 덱(deque)에 대해서 알아보자. 이 이름은

'double ended queue'의 줄임말이다. 양 끝에서, 즉, 앞과 뒤에서 삭제와 삽입이 빠른 자료구조이다.

파이썬 표준 라이브러리의 collections에서 deque를 제공한다. 오른쪽(뒤)과 왼쪽(앞)에 추가하는 append(), appendleft()와 삭제하는 pop(), popleft()를 사용할 수 있다. 간단한 예를 보면 다음과 같다.

```
>>> from collections import deque
>>> deq = deque()
>>> deq.append("a")
>>> deq.append("b")
>>> deq.append("c")
>>> deq
deque(['a', 'b', 'c'])
>>> deq.appendleft("d")
>>> deq.appendleft("e")
>>> deq
deque(['e', 'd', 'a', 'b', 'c'])
>>> deq.pop()
'c'
>>> deq.popleft()
'e'
>>> deq
deque(['d', 'a', 'b'])
```

여러 개의 원소를 한번에 추가하는 것도 가능하다. extend()와 extendleft() 메소드를 이용하면 된다.

```
>>> deq = deque()
>>> deq.extend([1, 2, 3, 4, 5])
>>> deq
deque([1, 2, 3, 4, 5])
>>> deq.extendleft([6, 7, 8, 9])
>>> deq
deque([9, 8, 7, 6, 1, 2, 3, 4, 5])
```

한편 deque는 rotate() 메소드를 이용해 원소들의 위치를 바꾸는 기능과 최대 길이를 제한하는 기능도 제공하고 있다. 다음은 회전의 예이다.

```
>>> deq.rotate()
>>> deq
deque(['b', 'd', 'a'])
```

길이를 제한하면 새로운 원소를 추가했을 때 반대쪽 원소가 밀려나간다.

```
>>> deq = deque(maxlen=5)
>>> deq.extend("abcde")
>>> deq
deque(['a', 'b', 'c', 'd', 'e'], maxlen=5)
>>> deq.extend("fgh")
>>> deq
```

```
deque(['d', 'e', 'f', 'g', 'h'], maxlen=5)
>>> deq.appendleft("k")
>>> deq
deque(['k', 'd', 'e', 'f', 'g'], maxlen=5)
```

리스트에서 유용하게 사용되는 slice를 이용하여 일부분을 뽑아내는 기능 사용할 수 없다.

```
>>> deq = deque([1, 2, 3, 4, 5])
>>> deq[0:3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence index must be integer, not 'slice'
```

이러한 기능이 필요하다면 `itertools` 모듈의 `islice`를 이용할 수 있다.

```
>>> from itertools import islice
>>> islice(deq, 0, 3)
<itertools.islice object at 0x10df09138>
>>> list(islice(deq, 0, 3))
[1, 2, 3]
```

연습 4.10 일렬로 원소들이 늘어서 있는 데이터에서 특정한 요소를 찾아 좌우에 무엇이 나타나는지를 함께 살펴보아야 하는 경우가 종종 있다. 일반적으로 검색은 이러한 형태로 이루어지는 경우가 많다. 자연 언어 텍스트에 대해 검색을 할 때 문맥과 함께 검색어를 보여주는 형식의 하나로 KWIC (KeyWord In Context)라는 방식이 있다. 다음과 같이 텍스트 파일과 검색어를 지정하면 KWIC 형식으로 출력하는 프로그램을 구현하여라. 이 예에서 alice.txt 파일은 `gutenberg.org` 프로젝트에서 다운로드 받은 것이다.

```
$ python kwic.py alice.txt Cheshire
01314      grins like that?' 'It's a    Cheshire     cat,' said the Duchess, 'and
01320   again:-- 'I didn't know that    Cheshire     cats always grinned; in fact,
01434  little startled by seeing the    Cheshire     Cat sitting on a bough
02069      said to herself 'It's the    Cheshire     Cat: now I shall have
02101      'It's a friend of mine--a    Cheshire     Cat,' said Alice: 'allow me
02145            she got back to the    Cheshire     Cat, she was surprised to
```

왼쪽의 숫자는 검색 파일 파일에서 해당 검색어가 나타난 행의 번호이다. 검색어를 중심으로 좌우에 5개의 단어를 출력하였다.

Python 05 함수형 프로그래밍

유현조

2022년 2월 24일

차 례

5

함수형 프로그래밍

이 장은 함수형 프로그래밍 방법론을 소개하는 것을 목적으로 하지만 본격적인 이야기에 앞서 '함수'라는 것을 프로로그래밍에서 작성하는 기초적인 내용부터 시작할 것이다. 파이썬에서 지원되는 형식을 소개하고 있지만 파이썬에만 해당되는 이야기는 아니다. 물론 다른 프로그래밍 언어에서 동일한 개념을 지원하더라도 코드를 작성하는 형식은 파이썬과 다를 수 있다. 파이썬이라는 특정 언어를 배우는 것이 목적이 아니라 프로그래밍에서 '함수'라는 개념을 활용하는 방법을 이해하는 것이 목적이라는 점을 잊지 맡고 공부하도록 하자.

파이썬은 절차형(procedural), 객체지향(obeject-oriented), 함수형(functional) 프로그래밍이 모두 가능한 multi-paradigm 언어 중의 하나이다. 함수형 프로그래밍 방법론은 수학적이고 이론적인 성격이 강하여 학술적으로 주로 논의되고 실무에서는 잘 쓰이지 않는 경향이 있었다. 일반적으로 상용 소프트웨어 개발에서는 복잡한 기능의 어플리케이션을 체계적으로 설계하는 데에 객체 지향 프로그래밍 방법론이 널리 사용되었다. 함수형 프로그래밍은 소프트웨어 개발 분야에서는 활용도가 높지 않았다. 그러나 최근에 빅데이터, 데이터과학이 컴퓨팅과 프로그래밍의 중요한 분야로 부각되면서 함수형 프로그래밍 방법론이 큰 인기를 끌고 있다. 함수형 프로그래밍은 알고리즘, 자료구조, 고성능 컴퓨팅 등과 밀접하게 연계되어 논의되는 방법론이다. 데이터 처리에서는 함수형 프로그래밍이 매우 간결한 코드를 작성할 수 있게 해주기도 한다. 요즘에는 고급 수준의 프로그래밍으로 함수형 프로그래밍을 가장 중요한 방법론으로 다루는 경우가 많다.

그러나 파이썬은 본격적인 함수형 언어는 아니다. 함수형 기법은 주로 간단하게 파편적으로 이용된다. 파이썬 언어 사용자로서 함수형 프로그래밍에 대한 배경 지식을 얻고 싶다면 다음 문서의 도입 부분을 읽어보라.

- http://docs.python.org/dev/howto/functional.html

5.1 함수

5.1.1 사용자 정의 함수 만들기

우선 파이썬에서 함수를 만들고 사용하는 기본적인 방법을 되돌아보자. 사용자 정의 함수를 만들기 위해서는 def라는 키워드를 이용한다. 함수를 만드는 기본 원리는 많은 프로그래밍 언어에서 유사하다. 함수의 정의는 기본적으로 함수의 이름, 인자의 목록, 리턴값으로 구성된다. 예를 들어, 다음은 두 수의 차를 구하는 함수이다. 함수의 이름, 인자의 목록, 리턴값이 어떻게 명시되었는지 확인해보라.

```
def diff(x, y):
    return x - y
```

이번에는 어떤 리스트가 주어지면 그 리스트에 들어있는 원소의 합을 구하는 함수를 만들어 보자. 내장 함수인 sum()은 다음처럼 작동한다.

```
>>> a = [1, 2, 3, 4, 5]
>>> sum(a)
15
```

이 함수를 직접 만들어 보자. 반복문을 이용하여 리스트의 원소의 합을 구하는 코드를 이용하여 사용자 정의 함수 정의를 작성해 보자[1].

```
def sum(a):
    total = 0
    for i in a:
        total = total + i
    return total
```

평균을 구하는 함수도 다음처럼 작성할 수 있다.

```
def avg(a):
    total = 0
    for i in a:
        total = total + i
    return total / len(a)
```

만약 이미 sum() 함수를 만들었다면 그 함수를 이용해 다음처럼 간단하게 만들 수도 있다.

[1] 사용자 정의 함수 sum()을 만들면 이와 동일한 이름의 내장 함수 sum()은 덮어쓰기가 되어 더 이상 사용할 수 없게 된다. del sum 명령으로 사용자 정의 함수를 지우면 다시 내장 함수 sum()을 쓸 수 있게 된다.

```
def avg(a):
        return sum(a) / len(a)
```

함수는 결국 하나의 프로그램이라고 할 수 있다. 프로그램이 어떤 반복적인 일을 처리하기 위해 작성된 것이라면 함수 또한 그렇다. 위와 같이 리스트의 합을 구할 일이 많을 때 그것을 함수로 만들어 놓으면 간단하게 처리되고 또 그것을 다른 함수를 만들 때 불러서 사용할 수 있다.

이렇게 어떠한 개별적인 기능의 단위를 하나의 함수로 묶는 것이 프로그래밍에서 중요한 과제 중의 하나이다. 프로그래밍을 한다는 것은 함수를 작성하는 것이라고 생각할 수도 있다.

5.1.2 함수의 개념

프로그래밍에서 함수(function)는 수학에서 함수와 유사하지만 동일한 개념은 아니다. 좀더 일반적인 용어로 기능(funciton)이라고 할 수도 있으나 어떤 특정한 형식을 갖춘 것이라는 점에서 단순히 '기능'이라는 용어를 사용하는 것은 부적절하다.

함수와 비슷한 것들

함수를 경우에 따라 서브루틴(subroutine)이나 프로시저(procedure)라고 부르기도 하고 객체지향 프로그래밍에서는 메소드(method)라고 부르기도 한다. 이런 용어들은 동의어로 사용되기도 하지만 조금씩 개념을 구별하여 사용하기도 한다.

용어 사용에 완벽한 합의가 존재한다고는 할 수 없으나 흔히 서브루틴이라고 하면 실행할 명령들을 묶어놓은 덩어리를 지칭한다. 일반적으로 함수는 인자을 입력으로 받아 어떤 결과를 리턴값으로 돌려주는 것을 말한다. 서브루틴은 그러한 형식을 갖추지 않아도 되는 것이다. 다른 말로 하면 프로그램의 일부분을 무엇이든 한 단위로 묶어서 이름을 붙여 놓았다가 필요할 때 불러 쓸 수 있게 만든 것은 모두 서브루틴이라고 할 수 있다. 파이썬에서는 `return` 문을 사용하지 않고 함수를 정의하는 것을 허락한다. 예를 들면 다음과 같다.

```
def say_hello():
    print('Hello')
```

사실 엄밀하게 말하면 파이썬에서 위와 같이 정의하면 리턴 값이 `None`인 함수가 된다. 파이썬에서는 서브루틴, 즉, 리턴 값이 없는 명령의 묶음은 만들 수 없는 셈이다.

좀 더 일반적인 개념으로 어떤 일을 하는 명령 덩어리를 '불러 쓸 수 있다'는 의미에서 호출가능(callable)한 것이라는 용어를 사용하기도 한다. 파이썬에서도 `callable()`이라는 내장함수를 이용하여 어떤 객체가 호출가능한 것인지 확인할 수 있다.

순수한 함수

프로그래밍에서는 함수를 순수한 함수(pure function)와 불순한 함수(impure function)이라는 관점에서 구분하기도 한다.

순수한 함수란 동일한 인자를 주었을 때 항상 동일한 결과가 나오고 결과 값을 내는 것 이외에는 다른 일을 하지 않는 함수를 말한다. 함수 내부에 어떤 숨겨진 장치가 없으며 인자로 받은 데이터 이외에 다른 정보를 외부에서 받아 오지 않는다. 인자로 주어진 것에 변화를 가하거나 결과 값을 리턴하는 것 이외의 다른 출력을 발생시키지 않는다. 산술 계산 함수가 전형적인 순수한 함수에 해당한다. 내장 함수 중 sum()을 예로 들 수 있다.

```
>>> a = [1, 2, 3, 4, 5]
>>> sum(a)
15
```

불순한 함수

불순한 함수(impure function)는 순수하지 못한 함수인데 무엇인가 외부적인 개입이 있는 것이라는 정도로 이해할 수 있다. 예를 들어, 다음 함수에서는 a라는 변수를 사용하고 있는데 이 변수는 함수 내부에 정의되어 있지 않다. 함수 외부에 있는 정보를 이용하는 것이다.

```
def inc(x) :
    return x + a
```

불순한 함수에는 부작용(side effect)을 일으키는 함수도 있다. 함수가 결과값을 내는 것 이외에 함수 밖에 있는 무엇인가에 영향을 미치는 것을 부작용라고 한다. 이 말에 들어 있는 부정적인 의미를 피하고 중립적인 의미로 이해하고 싶다면 '부수적 효과' 정도로 표현할 수 있다. 예를 들어, 다음과 같은 함수를 생각해 보자.

```
def setitem(arr, ind, val):
    arr[ind] = val
    return val
```

이 함수를 사용하면 어떤 일이 생기는지 알아보자. 다음에서 리스트 a는 함수 밖에 존재하는 것인데 함수를 사용하고 나면 a의 값이 바뀐다.

```
>>> a = [1, 2, 3]
>>> setitem(a, 0, 100)
100
>>> a
[100, 2, 3]
```

부작용이 있는 함수를 작성하거나 사용할 때는 주의해야 한다.

5.1.3 함수의 인자

인자의 이름

파이썬에서는 함수를 호출(call)할 때에도 인자의 이름을 사용할 수 있다. 무슨 말인지 잘 이해가 안 갈 수도 있고 당연한 것이 아닌가하는 생각이 들 수도 있겠다. 예를 들어 살펴보자. 그 전에 당연한 것은 아니라는 점은 이야기해 두고 진행하자. 함수를 호출할 때 인자의 이름을 지원하지 않는 프로그래밍 언어도 많다.

함수를 호출한다는 것은 함수를 사용할 때를 말한다. 함수를 만드는 것을 '정의'한다고 하고 사용하는 것을 '호출'한다고 한다. 다음 함수의 정의를 보자.

```
def rep(x, n):
    return [x] * n
```

여기에서 인자에 x, n라는 이름을 붙였다. x 값을 n번 반복하여 리스트로 돌려주는 함수이다. 사용하는 경우를 보자.

```
>>> rep(3, 4)
[3, 3, 3, 3]
>>> rep(4, 3)
[4, 4, 4]
>>> rep('a', 4)
['a', 'a', 'a', 'a']
```

사용할 때에는 인자의 이름 x, n이 드러나지 않는다. rep(3, 4)에서 3과 4는 인자의 위치에 의해서 그 의미가 결정된다. 이것이 일반적인 프로그래밍 언어들에서 공통적으로 지원하는 함수의 사용 방식이다. 인자의 의미는 그 위치에 의해 결정된다. 인자의 이름은 정의할 때에만 사용된다.

인자가 많은 경우에, 많지 않더라도 rep(3, 4)와 같은 코드를 검토할 때를 생각해 보자. 이 함수의 사용법을 외우고 있지 않는 한 3을 4번 반복하라는 것인지, 3번 4를 반복하라는 것인지는 rep(3, 4)라는 명령만 보아서는 알 수가 없다. 다음처럼 인자의 이름을 함께 써준다면 의미를 떠올리는 데에 힌트가 된다.

```
>>> rep(x = 3, n = 4)
[3, 3, 3, 3]
>>> rep(x = 4, n = 3)
[4, 4, 4]
>>> rep(x = 'a', n = 4)
['a', 'a', 'a', 'a']
```

보통 인자가 1개나 2개 혹은 3개인 경우까지도 인자 이름 없이 사용하는 데에 큰 혼동이 발생하지 않는다. 그러나 인자의 개수가 4개, 5개로 늘어나기 시작하면 인자 이름과 함께 함수를 호출하는 것이 유용함을 알게 될 것이다.

더 나아가 이름을 지정해 준다면 순서를 바꾸는 것도 가능하다.

```
>>> rep(x = 3, n = 4)
[3, 3, 3, 3]
>>> rep(n = 4, x = 3)
[3, 3, 3, 3]
>>> rep(n = 4, x = 'a')
['a', 'a', 'a', 'a']
```

물론 함수를 정의할 때 인자의 이름을 더 설명적으로 부여하면 코드의 의미가 더 명시적으로 드러날 것이다. 예를 들면, 다음처럼 말이다.

```
def rep(value, times):
    return [value] * times
```

이름 붙이기에 절대적인 기준은 없다. 간결성과 설명력 사이에서 적절한 선택을 하면 된다.

인자의 기본값

함수를 정의할 때 인자의 기본값을 지정할 수도 있다. 예를 들면, 다음과 같이 정의된 함수를 생각해 보자.

```
def incr(x, by=1):
    return x + by
```

인자의 기본값이 정의되어 있는 경우 호출할 때 해당 인자를 지정하지 않고 함수를 사용할 수 있다.

```
>>> incr(3)
4
>>> incr(4)
5
```

기본값이 아닌 다른 값이 필요한 때에는 해당 인자를 사용하면 된다.

```
>>> incr(3, 2)
5
>>> incr(3, by=2)
5
```

인자의 기본값을 지정해 줌으로써 해당 인자를 선택적(optional)인 것으로 만들 수 있는 셈이다. 이에 대비해 기본값을 지정해 주지 않은 인자는 필수적인 것이다. 기본값이 지정된 인자와 기본값이 지정되지 않은 인자라는 구분 대신에 선택적 인자와 필수 인자라는 구분도 가능하다. 다음과 같은 경우를 생각해 보자.

```
import datetime

def timestamp(msg=None):
    stamp = datetime.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
```

```
    if msg is not None:
        stamp += ' ' + msg
    return stamp
```

이 함수는 다음처럼 이용할 수 있다.

```
>>> timestamp()
'2020-06-02 01:26:24'
>>> timestamp('START')
'2020-06-02 01:26:51 START'
>>> timestamp('END')
'2020-06-02 01:26:55 END'
```

이와 같이 기본값을 None으로 주는 정의는 바로 이러한 선택적 인자라는 개념에 부합한다. 기본값이 없는데 기본값을 주는 것이므로 모순적으로 보일 수 있지만 해당 인자를 선택적인 것으로 만드는 형식적인 장치인 것이다.

인자의 개수가 가변적인 경우

지금까지 파이썬 프로그래밍을 공부하며 print 함수에서 무엇인가 이상하다는 점을 느낀 적이 있는가? 어떻게 이 출력 함수는 인자를 한 개, 두 개, 세 개 주어도 작동하는가? 파이썬에서는 함수의 인자를 가변적으로 만들 수 있는 장치를 제공하고 있다.

```
x = 1
y = 2
z = 3
print(x)
print(x, y)
print(x, y, z)
```

우선 함수의 인자가 tuple처럼 생겼다는 점을 상기해 보자. 파이썬의 tuple은 괄호를 반드시 쳐주지 않아도 된다는 점을 생각하면 함수의 인자는 더욱 tuple처럼 생겼다. 다음처럼 쉼표로 나열하면 반드시 괄호가 없어도 tuple임을 나타내는 것으로 약속되어 있다.

```
>>> 1,
(1,)
>>> 1, 2,
(1, 2)
>>> 1, 2, 3
(1, 2, 3)
```

하지만 함수의 인자 목록이 tuple은 아니다. 다음과 같은 함수를 생각해 보자.

```
def sum(arg):
    total = 0
    for x in arg:
        total += x
    return total
```

이것은 arg로 tuple을 받는다고 생각하고 작성한 함수이다. 이 함수를 사용하려고 다음과 같이 써주면 tuple로 처리되지 않고 함수의 인자로 처리되기 때문에 에러가 난다.

```
>>> sum(1,)
TypeError: 'int' object is not iterable
>>> sum(1, 2)
TypeError: sum() takes 1 positional argument but 2 were given
```

오류가 나지 않게 사용하려면 다음과 같이 해야 한다.

```
>>> sum((1,))
1
>>> sum((1, 2))
3
>>> sum((1, 2, 3))
6
```

괄호가 두 번 겹쳐있는 것이 보기 좋지 않다. 이 함수를 다음처럼 작동하게 작성할 수는 없을까?

```
>>> sum(1)
1
>>> sum(1, 2)
3
>>> sum(1, 2, 3)
6
```

다음처럼 해 주면 된다. 인자 이름 앞에 별표를 하나 붙여주면 인자의 목록을 tuple처럼 사용할 수 있게 된다.

```
def sum(*arg):
    total = 0
    for x in arg:
        total += x
    return total
```

이것과 같은 맥락에 있는 문법으로 함수를 호출할 때 인자에 별표를 붙여서 쓰는 경우가 있다. 위에서 본 것은 함수를 정의할 때 인자에 별표를 붙여서 표현하는 문법이었다. 이번에는 함수가 다음처럼 평범하게 정의된 경우이다.

```
def add(x, y):
    return x + y
```

이 함수를 사용하는 경우에 입력하려는 데이터가 tuple로 주어진 경우를 생각해 보자.

```
p = (3, 4)
add(p[0], p[1])
```

번거롭지 않는가? 다음처럼 할 수 있다.

```
add(*p)
```

위 두가지를 혼동하지 말자. 첫번째 경우는 함수를 정의할 때 인자에 별표를 붙인 것이고 두번째 경우는 함수를 사용할 때 인자에 별표를 붙인 것이다.

인자의 이름이 가변적인 경우

함수를 사용할 때 인자의 이름을 붙여서 호출하는 경우를 보자.

```
f(x=1, y=2)
```

이렇게 인자의 이름과 값을 제시하는 것은 다음과 같이 사전으로 정보를 전달하는 것과 유사하지 않은가?

```
f({'x' : 1, 'y' : 2})
```

인자의 이름 앞에 별표를 두 개 붙이면 함수의 이름이 있는 인자의 목록을 내부에서 사전으로 사용할 수 있게 해 준다.

```
def f(**kwargs):
    print(kwargs)
```

이 함수를 사용해 보면 다음과 같이 임의의 인자들을 사용할 수 있다.

```
>>> f(x=1, y=2)
{'y': 2, 'x': 1}
>>> f(z=5, x=3)
{'z': 5, 'x': 3}
```

이것은 가변적인 옵션을 함수에 전달하려고 할 때 유용하게 사용할 수 있다. 기본값이 있는 인자는 옵션의 개수가 많지 않으면서 고정되어 있는 경우에 사용한다. 이에 비해 `**kwargs` 형식의 인자는 옵션의 개수가 매우 많은 경우에 더 편리할 수 있다. 물론, 인자의 종류 자체가 결정되어 있지 않은 경우에도 사용할 수 있다. 이 경우는 사전을 인자로 받는 것과 마찬가지이다.

단순화하면 함수를 정의할 때 인자에 별을 하나 붙이면 인자를 리스트로 받게 되고 별을 두 개 붙이면 인자를 사전으로 받게 된다.

함수 정의	함수 호출	함수 내부 변수 값
`fun(*args)`	`f(1, 2, 3)`	`args = [1, 2, 3]`
`fun(**kwargs)`	`f(x=1, y=2, z=3)`	`kwargs = {'x' : 1, 'y' : 2, 'z' : 3}`

5.2 재귀

재귀(recursion)는 컴퓨터 과학에서 문제를 해결하는 핵심적인 방법의 하나이다. 함수형 프로그래밍에서 흔히 함수의 재귀적인 정의를 통해 문제를 해결하는 방법을 시작으로 이야기를 풀어가는 경우가 많으나 재귀가 반드시 함수형 프로그래밍에만 해당되는 것은 아니다. 다만 함수형 프로그래밍에서는 루프(loop) 보다 재귀를 선호하는 경우가 많고 어떤 경우에는 루프 자체를 아예 사용하지 않고 오로지 재귀만 사용하기도 한다.

5.2.1 factorial 계산의 예

재귀의 전형적인 예로 예로 factorial을 계산하는 함수를 생각해 보자. factorial은 다음과 같이 정의된다.

$$n! = \prod_{i=1}^{n} i = n(n-1)(n-2)\cdots 2 \cdot 1$$

이것을 수열로 생각해도 되고

$$a_n = n(n-1)(n-2)\cdots 2 \cdot 1$$

또는 n의 함수로 생각해도 된다.

$$f(n) = n(n-1)(n-2)\cdots 2 \cdot 1$$

이 $f(n)$ 함수를 그대로 받아들여 파이썬으로 구현한다면 다음과 같이 될 것이다.

```
def factorial(n) :
   total = 1
   for i in range(1, n+1):
      total *= i
   return total
```

이것은 절차형 프로그래밍으로 문제를 해결한 것이다. $n!$을 구하려면 무엇을 해야하는지를 순서대로 컴퓨터에게 지시하는 방식이다. 이 코드에서 명시적으로 드러나거나 의

도한 것은 아니지만 운좋게도 $0! = 1$이라는 정의에 맞추어 `factorial(0)`은 1로 계산이 된다. `n`이 양의 정수일 때만 계산하고 나머지 경우에는 에러 메시지를 출력하고 싶다면 그에 맞추어 코드가 더 추가되어야 할 것이다. 여기에서는 올바른 입력만 주어진다고 가정하고 논의를 이어가겠다.

점화식이라는 것을 들어보았을 것이다. 인접한 항 사이의 관계식을 제시하고 초항을 명시함으로써 수열을 나타내는 방식이다. factorial은 다음과 같이 표현할 수 있다.

$$a_0 = 1$$
$$a_n = na_{n-1}$$

또는

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ nf(n-1) & \text{if } n > 0 \end{cases}$$

이것을 그대로 파이썬에서 구현할 때 재귀(recursion)의 형태를 띄게 된다. 재귀라는 것은 어떤 함수가 그 내부에서 자기 자신을 호출하는 것을 말한다. 기본적으로 $f(n) = nf(n-1)$은 다음과 같은 형태로 구현될 것이다. `factorial` 함수의 정의에서 자기 자신을 다시 호출하고 있으므로 재귀라고 하는 것이다.

```
def factorial(n):
    return n * factorial(n-1)
```

위 코드에는 초항이 제시되어 있지 않고 `n`의 한계도 정해져 있지 않으므로 한없이 자기 자신을 호출하면서 `n`도 그에 따라 한없이 작아진다. 즉, 계산 결과가 나오지 않고 무한한 재귀적 순환에 빠지게 된다. 초항을 지정하기 위하여 위의 수식으로 표현된 그대로 구현하면 다음과 같다.

```
def factorial(n):
    if n == 0 : return 1
    else :      return n * factorial(n-1)
```

조건문 대신에 불 연산자를 이용하여 다음과 같이 프로그래밍할 수도 있다. 반복문도 없고 조건문도 없는 이러한 프로그래밍이 더 함수형다운 스타일이라고 할 수 있다.

```
def factorial(n):
    return (n==0 and 1) or n * factorial(n-1)
```

5.2.2 리스트의 원소의 합

다음으로 주어진 리스트의 원소의 합을 구하는 문제를 생각해 보자. 이때 리스트는 숫자를 원소로 가지는 리스트라고 하자. 예를 들어 다음과 같은 것이 될 것이다.

```
>>> lst = [0, 1, 2, 3, 4, 5]
```

절차형 사고로 이 문제를 해결한다면 다음과 같이 될 것이다. 하나의 리스트가 인자로 주어지면 그것의 원소들을 하나씩 나열해가며 더한 결과를 리턴하는 함수이다.

```
def sum(lst):
   total = 0
   for i in lst:
       total += i
   return total
```

재귀적 사고로 리스트의 원소의 합을 구하는 함수를 정의하면 기본적으로는 다음과 같은 형태가 된다. 어떤 리스트의 원소의 합은 그 리스트의 첫번째 원소와 나머지 리스트의 원소의 합이 된다.

```
def sum(lst):
    return lst[0] + sum(lst[1:])
```

위와 같은 코드만으로는 에러가 난다. 그 이유는 재귀가 끝나는 지점을 정해주지 않았기 때문이다. 이 문제의 경우에는 리스트에 더 이상 원소가 남지 않았을 때 어떻게 할지를 정해주어야 한다. 다음처럼 텅 빈 리스트의 합을 0으로 정의해 주면 문제가 해결된다.

```
def sum(lst):
    if lst == []: return 0
    else:         return lst[0] + sum(lst[1:])
```

리스트의 pop() 메소드를 이용하면 다음처럼 표현할 수도 있다.

```
def sum(lst):
    if lst == []: return 0
    else:         return lst.pop() + sum(lst)
```

지금까지 살펴본 예들은 절차형 사고로도 반복문 적절하게 이용하면 쉽게 해결될 수 있는 문제들이었다. 루프를 만드는 for라는 문법이 제공되지 않았다면 해결되기 어려운 문제였을 것이다. 하지만 파이썬에는 for가 있으므로 굳이 재귀를 사용할 필요는 없다. 단, 재귀를 이용하면 명시적 루프 없이도 반복이 필요한 계산을 처리할 수 있다는 점은 기억해두자.

5.2.3 중첩된 리스트의 원소의 합

이번에는 중첩된 리스트(nested list)의 경우를 생각해보자. 중첩된 리스트란 리스트 안에 또 리스트가 들어있는 것을 말한다. 이러한 리스트는 트리 구조를 나타낼 때 사용되기도 한다.

```
>>> nl = [20, [5, [2, 5]], [15, [1, 6]]]
```

우선 파이썬에 기본적으로 제공되는 내장 함수 sum을 사용해 보자. 단순 리스트를 인자로 주었을 때는 합을 계산해 주지만 중첩된 리스트를 인자로 넣어주면 에러가 난다.

```
>>> l = [0, 1, 2, 3, 4, 5]
>>> nl = [20, [5, [2, 5]], [15, [1, 6]]]
>>> sum(l)
15
>>> sum(nl)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

이제 우리는 중첩된 리스트의 원소들의 합을 구하는 함수를 생각해 보자. 이러한 함수를 for 루프를 사용하여 만들 수 있을까? 위의 주어진 nl이라면 다음과 같이 가능하다. 한 원소씩 루프를 돌리면서 그것이 리스트인지 아닌지를 체크해서 리스트이면 또 for 루프를 돌리고 아니면 합을 계산하는 것이다.

```
total = 0
for i in nl:
  if type(i) is list:
      for j in i:
          if type(j) is list:
              for k in j:
                  total += k
          else:
              total += j
  else:
      total += i
```

이것이 가능한 이유는 주어진 nl에 리스트가 중첩된 만큼은 for 문을 중첩해서 사용했기 때문이다. 만약 더 중첩된 리스트가 주어지면 그것의 합을 구하기 위해서는 리스트가 중첩된 수 만큼은 for 문을 중첩해서 사용해야한다. 어떤 리스트가 주어질지 모르는 상태에서 루프를 사용해서는 이 문제를 '본질적으로' 해결할 수 없다.

재귀를 이용하면 이 문제는 쉽게 풀린다. 앞서 작성했던 단순 리스트의 원소의 합을 구하는 함수와 별로 달라지지 않는다. 계산하려는 원소의 데이터 타입이 리스트인지 아닌지 판단하는 부분만 추가되면 된다.

```
def sum(lst):
    if lst == []:                return 0
    elif type(lst[0]) is list:   return sum(lst[0]) + sum(lst[1:])
    else:                        return lst[0] + sum(lst[1:])
```

이 함수는 중첩이 몇 겹으로 이루어졌는지 알 수 없는 리스트가 주어져도 계산할 수 있다는 실행 수준에서의 장점이 있다. 하지만 그보다 더 중요한 것은 논리 구조가 투명하고 아름다우며 코드도 간결하다는 프로그래밍 수준에서의 장점일 것이다.

5.2.4 뉴턴의 방법으로 제곱근 구하기

뉴턴의 방법(Newton's method)은 방정식의 근, 또는 어떤 함수가 0이 되는 근사값을 점진적으로 찾아가는 방법의 하나이다. 처음에 예측치로 x_0를 적절히 선택한 후에 다음 식을 따라서 $x_1, x_2, \cdots$를 구해나가면 해에 가까운 값을 구할 수 있다는 것이다.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

어떤 수 a의 제곱근을 찾는 것은 다음과 같은 함수가 0이 되는 해를 찾는 것이라고 할 수 있다.

$$f(x) = x^2 - a$$

따라서 초기 예측치 x_0를 선택하고 다음 식을 따라서 점진적으로 계산해 나가면 a의 제곱근에 가까운 값을 얻을 수 있게 된다. 초기값은 간편하게 1로 시작할 수 있다.

$$x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{x_n + a/x_n}{2}$$

이러한 뉴턴 법에 따라 제곱근을 구하는 절차를 파이썬에서 그대로 표현하면 다음과 같다. 여기서 x는 위 식에서 x_n에 대응한다. 아래 함수의 리턴값이 위 식의 x_{n+1}에 대응한다. a는 고정된 숫자이므로 파이썬의 함수에서 재귀적으로 호출할 때 a를 하위 프로세스에 그대로 전달한다.

```
def sqrt_approx(x, a):
    return sqrt_approx((x+a/x)/2, a)
```

위와 같은 코드는 훌륭하지만 역시 끝나지 않는다는 문제가 있다. 언제 끝날지를 정해 주어야한다. 끝나는 지점은 우리가 얼마나 정확한 값을 원하느냐에 따라 달라질 것이다. 예를 들어, 근사적인 해 x_n을 제곱해 보았을 때 a와의 차이가 0.001 이내라면 충분하다고 하면 파이썬 코드는 다음과 같이 될 것이다.

```
def sqrt_approx(x, a):
    if abs(x**2 - a) < 0.001 : return x
    else :                     return sqrt_approx((x+a/x)/2, a)
```

이 함수를 좀더 편리하게 사용하기 위하여 초기값을 무조건 1로 지정하자. 초기 예측값을 잘 정하면 계산량을 줄일 수 있겠지만 다음처럼 인자 하나를 받는 변수가 훨씬 편리할 것이다.

```
def sqrt(a):
    return sqrt_approx(1.0, a)
```

이것은 지금 논의하고 있는 재귀와는 무관하다. 껍데기에 불과한 함수이다. 하지만 우리가 만든 함수를 나중에 사용할 때 편의성을 생각한다면 중요한 장치이다.

5.3 함수는 일급 객체

파이썬에서 함수는 일급 객체(first-class object)이다. 일반적으로는 '일급 시민(first-class citizen)'이라는 용어를 쓰기도 한다. 이 말을 보았을 때 '이급(sencond-class)'도 있을 것이라고 예상될 것이다. 숫자처럼 변수에 할당할 수 있는 것을 '일급', 함수나 프로시저처럼 불러서 실행할 수만 있는 것을 '이급'으로 구분하는 개념은 1960년대부터 있었다. 간단히 말해 `f(x)`와 같은 형식에서 함수 `f`와 변수 `x`는 차원이 다른 존재라는 것이다.

파이썬에서 함수가 일급 객체라는 것은 함수가 일반 변수와 동일한 취급을 받는다는 것이다. 함수를 일반 변수와 동일하게 사용할 수 있다는 것을 의미한다. 즉 다른 함수의 인자로 함수를 넘길 수 있고 리턴할 수도 있으며 다른 변수에 할당될 수도 있다는 뜻이다. 함수형 프로그래밍 기법에서 필요로 하는 요소 중의 하나이다.

파이썬에서 우리는 `def` 키워드를 이용하여 함수를 정의할 수 있다.

```
>>> def minus(x) : return -x
```

정의된 함수의 이름을 인터랙티브 모드에서 입력해보면 함수임을 알 수 있다.

```
>>> minus
<function minus at 0xb7e2479c>
```

이 함수는 다음처럼 다른 변수에 할당될 수 있고 새로운 이름의 함수를 사용할 수 있다.

```
>>> neg = minus
>>> neg
<function minus at 0xb7e2479c>
>>> neg(4)
-4
```

5.3.1 람다 대수

람다 대수(λ-calculus)는 함수를 정의하기 위한 형식 체계의 하나이다. 다음과 같이 def 구문으로 정의된 함수는

```
>>> def f(x): return x + 3
...
>>> f(4)
7
```

람다 함수를 만들어주는 lambda 키워드를 이용하면 다음처럼 쓸 수 있다.

```
>>> f = lambda x: x + 3
>>> f(4)
7
```

람다가 유용한 것은 아주 간단한 이름이 없는 함수(anonymous function)를 다른 함수의 인자로 주거나 리턴값으로 사용할 때이다. 람다 함수는 다음처럼 그 함수에 이름을 붙이지 않고도 사용할 수 있다.

```
>>> (lambda x: x + 3)(4)
7
```

람다 함수를 이용하는 구체적인 예로 리스트의 sort() 메소드를 다시 한번 살펴 보자. sort()에 아무런 인자도 주지 않으면 숫자형 데이터는 크기 순서대로, 문자열 데이터는 철자순으로 정렬해 준다.

```
>>> members = ['john', 'alice', 'tom']
>>> members.sort()
>>> members
['alice', 'john', 'tom']
```

정렬 기준을 지정해 줄 수도 있다. 다음처럼 해 주면 문자열을 그 길이에 따라 정렬해 준다. 길이가 짧은 것부터 짧은 것 순으로 정렬된다.

```
>>> members.sort(key=lambda x: len(x))
>>> members
['tom', 'john', 'alice']
```

다음처럼 해 주면 길이가 긴 것부터 짧은 것 순으로 정렬된다.

```
>>> members.sort(key=lambda x: len(x), reverse=True)
>>> members
['alice', 'john', 'tom']
```

sort()의 인자로 주는 것은 정렬에 필요한 기준을 제공하는 함수를 써주는 것이다. 위에서 만들어지는 함수는 문자열 x를 입력으로 받아서 길이를 리턴값으로 돌려주는 함수이다. sort는 이 함수를 이용하여 리스트 내의 원소들의 쌍을 비교한다.

□

뉴턴의 방법을 이용하여 방정식의 해를 구하는 함수를 만들어보자. 뉴턴의 방법은 다음처럼 나타낼 수 있다.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

우선 다음을 만족하는 x값을 구해보자. 물론 우리가 의도하는 것은 $f(x)$가 다르게 정의되어도 $f(x) = 0$을 만족하는 x의 해를 푸는 것이다.

$$f(x) = 4x^3 - 6x^2 - 16x - 6 = 0$$

이를 위해 일단 이 함수를 파이썬에서 다음과 같이 정의할 수 있다.

```
def f(x):
    return 4*x**3 - 6*x**2 - 16*x -6
```

인수분해를 해보면 알겠지만 $x = 3$일 때 0이된다. 따라서 다음과 같이 3을 인자로 주면 0이 나오는 것을 확인할 수 있다.

```
>>> f(3)
0
```

뉴턴의 방법을 사용하기 위해서는 우선 도함수를 만들어야 한다. 여기에서는 직접 미분을 하는 대신에 Δx를 작게 해서 다음 값을 구하는 것으로 도함수를 대신하자.

$$f'(x) \approx \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

파이썬에서 이것을 구현해 보자. derivative라는 함수를 만들 것이다. 이 함수는 어떤 함수 f를 인자로 받아서 그것의 도함수 f'를 리턴해 준다. 다음과 같은 형태가 될 것이다.

```
def derivative(f) :
    return lambda x: (f(x+0.001) - f(x)) / 0.001
```

위 코드에서는 0.001 자리에는 필요에 따라 적당히 작은 값을 사용할 수 있다. 이것을 h라는 인자로 나타내고 편의상 기본값을 0.001로 지정하여 보자.

```
def derivative(f, h=0.001) :
    return lambda x: (f(x+h) - f(x)) / h
```

앞서 예로 삼은 $f(x)$를 실제로 미분해보면:

$$f'(x) = 12x^2 - 12x - 16$$

우리가 작성한 derivative() 함수가 제대로 작동하는지 테스트해 보자. 위 식에 따르면 $f'(0) = -16$이고 $f'(1) = -16$이다. 우리가 만든 함수가 상당히 가까운 근사값을 계산해 주는 것을 확인할 수 있다.

```
>>> derivative(f)(0)
-16.005995999999634
>>> derivative(f)(1)
-15.993995999998845
```

이제 $f(x) = 0$을 만족하는 해를 구하는 함수 solve()를 만들어보자. 앞서 만들어 보았던 제곱근을 구하는 경우와 별로 다를 것이 없다.

```
def solve(x, f, e=0.001):
    if abs(f(x)) < e : return x
    else :             return solve(x - f(x)/derivative(f)(x), f)
```

초기값은 사용자가 지정해 주어야한다. $f(x) = 0$의 해가 여러 개가 있을 수 있고 초기값을 어떻게 지정하느냐에 따라 가까운 해를 구해줄 것이다. 다음과 같이 초기값으로 0, 100, -100을 넣어 세 개의 해 $-0.5, -1, 3$을 구했다.

```
>>> solve(0, f)
-0.49999863035692499
>>> f(-0.5)
0.0
>>> solve(100, f)
3.000000053907216
>>> f(3)
0
>>> solve(-100, f)
-0.99999944978695066
>>> f(-1)
0
```

처음에 우리가 정한 $f(x)$가 아니더라도 얼마든지 풀 수 있다. 예를 들어, $x^2 - 3x + 2 = 0$의 해를 구하고 싶다면:

```
>>> solve(0, lambda x: x**2 - 3*x +2)
0.9999891617701554
```

다시 한번 필요한 코드를 요약하면

```
def derivative(f, h=0.001) :
    return lambda x: (f(x+h) - f(x)) / h
```

```
def solve(x, f, e=0.001):
    if abs(f(x)) < e : return x
    else :             return solve(x - f(x)/derivative(f)(x), f)
```

이와 같은 간단한 두 개의 함수로 임의의 방정식의 해를 구하는 프로그램을 완성할 수 있다. 물론 여기서 더 중요한 것은 함수형 프로그래밍 기법이 아니라 뉴턴의 방법이다. 파이썬과 함수형 프로그래밍에게 감사하기 전에 뉴턴에게 먼저 감사해야 할 것이다. ■

5.3.2 함수의 합성

함수의 합성을 예로 한 함수를 다른 함수의 인자로 넘기고 함수를 리턴하는 방법을 살펴보자. 두 함수를 인자로 받아 합성해 주는 함수 compose()를 다음과 같이 정의할 수 있다. 아래의 예는 간단하지만, 합수를 인자로 넘기고, 람다 식을 이용하여 이름 없는 함수를 만들고, 함수를 리턴하는 예를 모두 포함하고 있다.

```
>>> def compose(f, g): return lambda x: f(g(x))
```

이제, 예를 들어, 다음과 같이 어떤 수의 제곱을 해 주는 함수가 정의되었다고 하자.

```
>>> def square(x): return x**2
```

이제 앞서 만든 minus() 함수와 square() 함수를 합성해 보자. 다음처럼 두 함수를 합성해서 새로운 람다 함수가 만들어진 것을 확인할 수 있다.

```
>>> compose(square, minus)
<function <lambda> at 0xb7e24764>
```

합성된 함수가 어떻게 작동하는지 테스트해보자.

```
>>> compose(square, minus)(4)
16
>>> compose(minus, square)(4)
-16
```

물론 합성된 함수에 이름을 붙여서 사용할 수도 있다.

```
>>> h = compose(minus, square)
>>> h(4)
-16
```

5.4 고차 함수

앞서 함수를 인자로 취하고 함수를 리턴하는 매우 단순한 형태의 예들을 살펴보았다. 이렇게 다른 함수를 인자로 취하거나 함수를 리턴하는 함수를 고차 함수(higher-order function)라고 한다. 파이썬에서는 자주 사용되는 유형의 고차 함수들을 제공하고 있다. `map()`, `filter()`, `reduce()`가 있다.

5.4.1 map

이런 문제를 생각해 보자. 어떤 정수들의 리스트가 주어졌는데 각 정수들을 제곱한 리스트를 만들고 싶다. 즉, 다음과 같이 바꾸고 싶다.

```
[1, 2, 3, 4, 5] -> [1, 4, 9, 16, 25]
```

주어진 리스트 `lst`에서 각 원소가 제곱된 리스트 `lst2`를 다음과 같이 다음과 같이 만들 수 있을 것이다.

```
lst = [1, 2, 3, 4, 5]
lst2 = []
for i in lst:
    lst2.append(square(i))
```

이것을 `map` 함수를 이용하면 다음과 같이 쓸 수 있다.

```
>>> lst = [1, 2, 3, 4, 5]
>>> map(square, lst)
<map object at 0x103dd3978>
```

여기서 map object는 실제로 lst의 각 원소를 제곱하여 새롭게 리스트를 만든 것은 아니다. 반복자(iterator)라고 하는 타입으로 나열형 자료에 대한 반복을 가능하게 해준다. 리스트로 바꾸고 싶다면 다음과 같이 할 수도 있다.

```
>>> list(map(square, lst))
[1, 4, 9, 16, 25]
```

반복문에 넣어 돌리고 싶다면 다음 구문을 이용한다.

```
>>> for item in map(square, lst):
...     print(item)
```

이때 `map`의 첫번째 인자로 줄 함수가 이미 구비되어 있지 않은 경우가 많다. 그때 바로 람다를 유용하게 사용할 수 있다. 예들 들어, 주어진 리스트의 제곱의 합을 구하려고 할 때 제곱해주는 함수는 따로 정의한 후에 `map`에 넣어주는 것은 매우 번거로운 일이다. 람다 식을 이용하면 다음 처럼 간편하게 써줄 수 있다.

```
>>> lst = [1, 2, 3, 4, 5]
>>> sum(map(lambda x: x**2, lst))
55
```

분산도 다음처럼 구할 수 있다.

```
>>> m = sum(lst) / len(lst)
>>> sum(map(lambda x: (x-m)**2, lst)) / len(lst)
2
```

5.4.2 filter

주어진 리스트에서 특정 조건을 만족시키는 것만 골라서 새로운 리스트로 만들고 싶을 때가 있다. 예를 들어, 다음과 같은 숫자들의 리스트에서 5보다 큰 값만 골라 새로운 리스트를 만들고 싶다.

```
[10, 1, 8, 1, 9, 6, 10, 10, 6, 5] ->  [10, 8, 9, 6, 10, 10, 6]
```

이를 위해 다음처럼 for 루프로 리스트의 원소들을 탐색하면서 if 문으로 조건을 만족시키는 경우에만 새로운 리스트에 추가하는 방식으로 이 문제를 해결할 수 있을 것이다.

```
lst = [10, 1, 8, 1, 9, 6, 10, 10, 6, 5]
lst2 = []
for i in lst:
    if i > 5: lst2.append(i)
```

동일한 결과를 다음과 같이 `filter()`를 이용하여 간단하게 얻을 수 있다. `filter`(f, $\mathbf{x}$)의 형태로 사용한다. 이때 f는 참(`True`) 또는 거짓(`False`)를 리턴하는 판별 함수이고 $\mathbf{x}$는 시퀀스 자료이다.

```
>>> xs = [10, 1, 8, 1, 9, 6, 10, 10, 6, 5]
>>> filter(lambda x: x > 5, xs)
<filter object at 0x104033400>
```

판별하는 과정이 간단하지 않다면 따로 함수를 정의해서 사용하는 것이 좋다. 여기에서는 지면상 간단한 것으로 짝수를 판별하는 함수를 따로 정의하는 경우를 생각해보자. 짝수를 판별하는 함수를 풀어서 쓰면 다음과 같다.

```
def isEven(x):
    if x % 2 == 0: return True
    else:          return False
```

이 함수는 간단하게 다음처럼 정의하는 것이 정상이다.

```
>>> def isEven(x): return x % 2 == 0
...
>>> isEven(2)
True
>>> isEven(3)
False
```

이제 다음처럼 filter()할 때 사용할 수 있다.

```
>>> filter(isEven, xs)
```

5.4.3 reduce

다음으로 살펴볼 reduce()는 파이썬2에서는 기본으로 지원하였으나 파이썬3에서는 빠졌다. 대신 functools 모듈에서 지원한다. 2항 함수를 시퀀스의 왼쪽 원소들부터 차례로 적용시켜 나간다. 즉, 두 개의 인자를 취하는 함수 f와 5개의 원소로 이루어진 리스트 $\mathbf{x}$가 주어졌을 때 reduce(f, $\mathbf{x}$)는 다음처럼 실행된다. 결과적으로 리스트가 하나의 값으로 줄어들게 된다는 뜻에서 reduce라는 이름이 붙었다.

$$\texttt{reduce}(f, [x_1, x_2, x_3, x_4, x_5]) \longrightarrow f(f(f(f(x_1, x_2), x_3), x_4), x_5)$$

주어진 리스트의 원소들의 합은 다음과 같이 구할 수 있다.

```
>>> from functools import reduce
>>> lst = [1, 2, 3, 4, 5]
>>> reduce(lambda x, y: x + y, lst)
15
```

곱은 다음과 같이 구할 수 있다.

```
>>> reduce(lambda x, y: x * y, lst)
120
```

참고로 operator라는 모듈을 import하면 다음과 같이 할 수도 있다.

```
>>> import operator
>>> reduce(operator.add, lst)
15
>>> reduce(operator.mul, lst)
120
```

5.5 조건제시법

파이썬의 가장 강력한 도구 중의 하나인 리스트 조건제시법(list comprehension)을 살펴보자. 이것은 기존의 리스트에 기반해서 새로운 리스트를 만들 수 있게 해주는 구문을 말한다. 수학에서 집합을 기존의 집합에 기반해서 정의하는 것을 본 적이 있을 것이다.

$$\{x^2 : x \in N\}$$

파이썬의 리스트 조건제시법은 이것과 형식적으로 동일하다.

앞서 `map()` 함수를 사용하는 방법을 배웠는데 동일한 결과를 list comprehension을 이용해서 얻을 수도 있다. 다음 두 표현은 동일한 결과를 낸다. 리스트 안에서 `for` 문을 사용한다.

```
map(func, lst)           # higher-order function map()
[func(x) for x in lst]   # list comprehension
```

예를 들어, 주어진 리스트의 원소들의 제곱으로 이루어진 새로운 리스트를 만들어보자.

```
>>> lst = [1, 2, 3, 4, 5]
>>> [x**2 for x in lst]
[1, 4, 9, 16, 25]
```

또 다른 함수인 `filter()`로 할 수 있는 일도 list comprehension으로 할 수 있다. 리스트 안에서 `if` 문을 사용한다.

```
filter(func, lst)             # higher-order function filter()
[x for x in lst if func(x)]   # list comprehension
```

예를 들어, 주어진 리스트에서 짝수만 골라 새로운 리스트를 만들어보자.

```
>>> lst = [1, 2, 3, 4, 5]
>>> [x for x in lst if x % 2 == 0]
[2, 4]
```

중첩된 리스트가 있듯이 list comprehension도 중첩될 수도 있다. 다음은 주어진 행렬에서 행과 열을 바꾸는 방법의 하나이다.

```
>>> mat = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> [[row[i] for row in mat] for i in [0, 1, 2]]
[[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

이 경우 리스트 함수인 `zip()`을 사용하는 것이 더 편리하다.

```
>>> zip(*mat)
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Python 06 객체 지향 프로그래밍

유현조

2022년 2월 8일

차 례

6

객체 지향 프로그래밍

6.1 테이블형 데이터

테이블은 행과 열로 이루어져있다. 행은 레코드(record), 열은 필드(field)라고 불리기도 한다. 이런 용어는 데이터베이스에서 흔히 사용된다. 필드 중의 하나는 레코드를 식별하는 용도로 사용될 수 있는데 이런 필드는 흔히 키(key)라고 불리기도 한다. 하나의 레코드는 서로 연관이 있는 값들의 튜플(tuple)로 이루어지게 되는데 이것은 흔히 실세계의 어떤 대상(object)을 지칭하고 그 대상이 가진 속성(attribute)의 값(value)을 묶어놓은 것이다. 테이블 형태와 카드 형태를 생각해보라. 카드 형태는 하나의 카드가 실세계의 하나의 대상에 대응하는 형태라 직관적 이해가 잘 부합한다. 그러나 카드형 데이터를 관리하는 것은 불편하다. 관계형 데이터베이스가 널리 사용되는 데에서도 알 수 있듯이 데이터는 테이블 형태로 만들어놓는 것이 다루기 편하다.

```
name        phone       type
---------------------------
aardvark    555-5553    A
barfly      555-7685    B
bites       555-1675    B
camelot     555-0542    A
```

$\Longleftrightarrow$

```
 name = aardvark
phone = 555-5553
 type = A
```

객체 지향 프로그래밍을 본격적으로 들어가기에 앞서 이런 테이블형 데이터를 다루는 방법들을 비교하며 객체(object)를 이용하는 방법의 장점을 생각해보자.

(1) 각 열(컬럼, 필드)을 하나의 리스트로:

```
names = ['aardvark', 'barfly', 'bites', 'camelot']
phones = ['555-5553', '555-7685', '555-1675', '555-0542']
types = ['A', 'B', 'B', 'A']
```

다음과 같이 데이터에 접근할 수 있다.

```
>>> names[0]
'aardvark'
>>> phones[0]
'555-5553'
>>> for name, phone, type in zip(names, phones, types):
...     print(name, phone, type)
...
aardvark 555-5553 A
barfly 555-7685 B
bites 555-1675 B
camelot 555-0542 A
```

각 리스트는 어떤 개체의 속성을 나타내고 있다. 각 리스트에서 첫번째 값들은 첫번째 대상의 속성을 나타내고 있다. 즉, 'aardvark'의 전화번가 '555–5553'이고 서비스 유형이 'A'이다. 그러나 각 속성을 독립적인 리스트로 만들면 이러한 관계가 보장되지 않는다.

(2) 하나의 행(레코드)를 하나의 순서짝으로:

```
customers = [ ('aardvark', '555-5553', 'A'),
              ('barfly',   '555-7685', 'B'),
              ('bites',    '555-1675', 'B'),
              ('camelot',  '555-0542', 'A')]
```

이때 다음과 같이 데이터에 접근할 수 있다.

```
>>> customers[0]
('aardvark', '555-5553', 'A')
>>> customers[0][0]
'aardvark'
>>> for customer in customers:
...     print(customer[0], customer[1], customer[2])
...
aardvark 555-5553 A
barfly 555-7685 B
bites 555-1675 B
camelot 555-0542 A
```

하나의 개체(레코드)에 대한 속성(필드)이 묶여 있다는 장점이 있다. 속성을 인덱스로 표시하고 있다. 0번 인덱스는 이름, 1번 인덱스는 전화번호, 2번 인덱스는 서비스 유형과 같은 식이다. 명시적으로 무슨 속성인지 드러나지 않으므로 불편하다.

(3) 하나의 개체(레코드)를 속성-값(attribute-value)의 짝으로 하는 사전으로:

```
customers = [ {'name' : 'aardvark', 'phone' : '555-5553', 'type' : 'A'},
              {'name' : 'barfly',   'phone' : '555-7685', 'type' : 'B'},
              {'name' : 'bites',    'phone' : '555-1675', 'type' : 'B'},
              {'name' : 'camelot',  'phone' : '555-0542', 'type' : 'A'}]
```

다음과 같이 데이터에 접근할 수 있다.

```
>>> customers[0]
{'phone': '555-5553', 'type': 'A', 'name': 'aardvark'}
>>> customers[0]['name']
'aardvark'
>>> customers[0]['phone']
>>> for customer in customers:
...     print(customer['name'], customer['phone'], customer['type'])
...
aardvark 555-5553 A
barfly 555-7685 B
bites 555-1675 B
camelot 555-0542 A
```

속성의 값을 속성의 이름으로 접근할 수 있으므로 편리하다. `customers[0]['name']`은 0번 인덱스의 고객의 이름이라는 의미로 직관적으로 이해될 수 있다.

(4) 하나의 개체를 하나의 객체(object)로:

```
class Customer:
    def __init__(self, name, phone, type):
        self.name = name
        self.phone = phone
        self.type = type

customers = [ Customer('aardvark', '555-5553', 'A'),
              Customer('barfly',   '555-7685', 'B'),
              Customer('bites',    '555-1675',  'B'),
              Customer('camelot',  '555-0542', 'A')]
```

다음과 같이 객체에 접근할 수 있다.

```
>>> customers[0].name
'aardvark'
>>> customers[0].phone
'555-5553'
>>> for c in customers:
...     print(c.name, c.phone, c.type)
...
aardvark 555-5553 A
barfly 555-7685 B
bites 555-1675 B
camelot 555-0542 A
```

참고로 객체를 이렇게 속성값을 구조화하는 용도로만 사용한다면 클래스 대신에 `collections` 모듈의 `namedtuple`를 사용해도 된다.

6.2 클래스와 인스턴스

파이썬은 객체 지향 프로그래밍(OOP, Object-Oriented Programming) 기법을 지원한다. 파이썬의 객체 지향 프로그래밍에서는 클래스(class)를 설계한 후에 그것의 인스턴스(instance)를 생성하여 사용한다. 클래스는 추상적인 부류(class)이고 인스턴스는 그 부류에 속하는 구체적인 사례(instance)이다. 비유하자면, 클래스가 자동차의 설계도라면 인스턴스는 그 설계도에 따라 제작된 실제 자동차다. 하나의 설계도로 여러 대의 자동차를 제작할 수 있다. 이렇게 제작된 자동차는 하나의 설계도에서 나왔기 때문에 겉모습이 완벽하게 동일할 수도 있지만 서로 다른 물리적 공간을 차지하는 서로 다른 인스턴스다.

6.2.1 클래스 정의

새로운 클래스를 정의할 때는 `class`라는 키워드를 사용한다. 클래스의 이름은 대문자로 시작하는 것이 일반적인 관례이다. `pass`라는 것은 당장은 실제 내용을 작성하지 않고 넘어가려고 할 때 사용한다.

```
>>> class Car:
...         pass
```

이제 우리는 `Car`라는 이름의 클래스를 가지게 되었다. 다음처럼 `Car`가 정의되어 있음을 확인할 수 있다.

```
>>> Car
<class '__main__.Car'>
```

6.2.2 인스턴스 생성

이제 이 `Car`의 인스턴스를 만들어보자. 다음처럼 클래스 이름 뒤에 괄호를 붙이면 인스턴스를 만들어 준다.

```
>>> c = Car()
```

이제 정말로 Car의 인스턴스가 제대로 만들어졌는지 확인해 보자. 아래 출력된 십육진수 숫자는 객체의 id라고 하는 고유번호이다.

```
>>> c
<__main__.Car object at 0x10a6704e0>
>>> hex(id(c))
'0x10a6704e0'
```

이제 Car의 인스턴스를 하나 더 만들어 보자. 위에서 만든 Car 객체와 id가 다른 것을 확인할 수 있다.

```
>>> d = Car()
>>> d
<__main__.Car object at 0x10a6705c0>
```

지금 생성한 두 객체는 같은 클래스에서 만들어진 것이지만 서로 다른 존재이다. 따라서 is 연산자로 비교하면 거짓이 나온다.

```
>>> c is d
False
```

두 객체의 등가성에 대해 따로 정의하지 않는 한 id를 이용하여 비교하므로 다른 값을 가지는 것으로 판정된다.

```
>>> c == d
False
```

6.3 속성

객체는 속성(property)을 가질 수 있다. 같은 부류에 속하면서 다른 속성을 가지는 인스턴스가 존재할 수 있다는 것이 속성이라는 개념의 주요한 필요성일 것이다. 방금 클래스도 만들어보고 그것의 인스턴스도 생성하여 보았다. 하지만 클래스는 아무 내용도 없는 것이었고 따라서 인스턴스도 텅 빈 것이었다. 앞서 클래스는 추상적이고 인스턴스는 구체적인 것이라고 설명하였다. 예를 들어, 자동차를 설계할 때 다른 부분은 동일하지만 색깔은 달리 할 수 있는 경우를 생각해 보자. 한 설계도에서 빨간 자동차도 만들 수 있고 까만 자동차도 만들 수 있을 것이다. 이럴 때 자동차가 색깔이라는 속성을 가진다고 한다.

6.3.1 변수

속성은 객체 내부의 변수로 표현된다. 객체 내부에 들어 있는 변수를 필드(field), 멤버 변수(member variable) 등으로 부른다. 때로는 클래스 변수와 인스턴스 변수라는 개념을 도입하여 구별할 필요도 있다.

여기에서는 각 인스턴스마다 가지는 고유한 속성을 표현하기 위해 객체 내부에 변수를 사용하는 경우만을 이야기해 보자. 인스턴스의 이름 뒤에 점을 찍고 속성을 표현하기 위한 변수를 추가할 수 있다. 예를 들면 다음과 같다.

```
>>> c = Car()
>>> c.color = 'red'
>>> c.color
'red'
```

새로운 속성을 위와 같은 방식으로 얼마든지 추가할 수 있다. 이런 자유로움은 편리할 수도 있지만 어떤 객체에 무엇이 들어있는지 예상할 수 없기 때문에 일관성 있게 체계적으로 프로그래밍하는 것을 어렵게 만든다. 일반적으로 객체 지향 프로그래밍에서 이런 방식으로 새로운 속성을 도입하지는 않다.

6.3.2 초기화

한 객체가 가질 수 있는 속성을 아무 때나 필요할 때 임기응변적으로 추가하는 것보다는 클래스 설계에 반영하고 인스턴스를 생성할 때 필수적인 속성의 값을 초기화하는 것이 좋다. 앞서 인스턴스를 생성할 때 클래스 이름 뒤에 둥근괄호를 붙였다. 다시 써보면 다음과 같다.

```
>>> c = Car()
```

이때 괄호는 그냥 붙인 것이 아니라 사실은 클래스 안에 들어 있는 어떤 메소드가 실행된 것이다. 인스턴스를 만들 때 특별히 약속된 이름의 메소드가 실행되되어 객체의 상태를 초기화한다. 예를 들어, 다음과 같이 색깔 속성을 가지는 자동차 클래스를 설계할 수 있다.

```
>>> class Car:
...     def __init__(self, color):
...         self.color = color
...
```

위 코드에서 `Car` 클래스 안에 `__init__`라는 함수가 보인다. 표기 때문에 어렵게 느낄 수 있으나 단지 약속일 뿐이다. 초기화(initialization)라는 의미에서 그렇게 이름을 붙인

것이며 양쪽에 밑줄 문자를 두 개 붙인 것은 프로그래머가 자유롭게 만들 수 있는 메소드 이름과 가능한 충돌을 피하기 위한 장치이다.

이제 다음과 같이 인스턴스를 만들 때 속성을 초기화하여 생성할 수 있다. 클래스 이름 뒤에 괄호를 붙여서 함수처럼 보이는 `Car()`를 사용하면 `__init__()` 메소드가 실행되는 것이다.

```
>>> c = Car('red')
```

예제 6.1

한 클래스로부터 인스턴스를 생성할 때 속성값을 부여해서 인스턴스를 만드는 것이 일반적이다. 클래스를 설계할 때 먼저 어떻게 사용할 것인지부터 생각하는 것으 좋을 때가 많다. 프로그램을 짜기 전에 어떻게 쓰면 좋을지 생각을 해보는 것이다.

어떤 패스트푸드 체인에서 매장 관리 프로그램을 개발하는 경우를 예로 들어 보자. 직원 관리 기능이 있다. 각 직원을 하나의 객체로 설계하는 것은 자연스러운 생각일 것이다. 직원 클래스 `Employee`를 정의하려고 한다. 이때 직원 클래스가 이미 있다면 어떻게 사용할지 생각해 보는 것이다. 이름이 'John'이고 나이가 25세인 직원 인스턴스를 만들라는 의미로

```
>>> john = Employee("John", 25)
```

라고 쓸 수 있다면 좋을 것이다. 이와 같이 사용하고 싶다면 클래스를 그에 맞추어 정의하면 된다. 다음처럼 클래스를 정의하면 된다.

```
class Employee:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

이럴 때 `__init__()` 메소드를 사용하는 것이다. 이 메소드 이름이 낯설어서 처음에는 어렵게 느껴질 수 있지만 클래스, 인스턴스, 속성의 개념 이상 알아야 할 것은 없다. 그러한 개념을 실제로 언어로 써서 구현하려고 하다 보니 여러가지 형식적인 약속이 생겨나게 된 것이다.

```
>>> Employee("John", 25)
```

라는 식으로 클래스 이름, 여기에서는 `Employee` 뒤에 괄호를 붙여서 마치 함수인 것처럼 사용하면 실제로는 `__init__` 함수가 불려지는 것이다.

6.3.3 self

파이썬의 클래스 정의에서는 self라는 키워드가 사용된다. 이 키워드는 인스턴스 자신을 가리키기 위한 장치이다. 클래스를 설계할 때는 아직 존재하지 않지만 나중에 이 클래스에서 인스턴스가 생성될 것이다.

```
>>> c = Car('red')
>>> d = Car('black')
```

여기서 c, d가 가리키고 있는 객체는 Car의 인스턴스이다. 하지만 Car 클래스를 설계할 때에는 그것이 c가 가리키는 객체가 될지, d가 가리키는 객체가 될지, 아니면 또다른 무엇이 될지 미리 알 수 없다. 한 클래스에서 여러 개의 인스턴스를 생성할 수도 있다. self는 이렇게 생성된 각각의 인스턴스가 자기 자신을 가리킬 수 있게 해주는 위한 장치이다.

앞서 작성한 Car의 클래스 설계에서 self.color이라고 되어 있다는 것은 인스턴스를 만들었을 때 다음과 같이 쓸 수 있다는 의미이다.

```
>>> c = Car('red')
>>> c.color
```

단순하게 생각하면 self 자리에 인스턴스의 이름을 쓸 수 있는 것이다. 그리고 클래스를 설계할 때, 즉 Car를 정의할 때에는 인스턴의 이름을 알 수 없으므로 self라는 특별한 키워드를 이용하는 것이다.

예제 6.2

패스트푸트 매장 관리 시스템 예제를 다시 생각해 보자. 다음과 같이 정의하였다.

```
class Employee:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

그리고 다음처럼 사용할 수 있었다.

```
>>> john = Employee("John", 25)
```

이렇게 새로운 인스턴스를 만들어서 john이라는 변수에 할당을 했다면

```
>>> john.name
'John'
>>> john.age
25
```

라고 쓰고 싶은 것이 인지상정일 것이다. 이것을 가능하게 해 주는 것이 바로 self이다. 클래스의 정의 내용에서 self.name과 self.age가 정의되어 있기 때문에 john이라는 인스턴스 변수에 대해서 john.name, john.age라고 쓸 수 있는 것이다.

속성은 변수이다. 일반 변수를 사용하듯이 사용할 수 있다. 변수값을

```
>>> name = 'John'
>>> age = 25
```

라고 초기화하듯이 인스턴스를 만들 때 인스턴스의 속성들은 __init__ 함수에 의해 초기화된다.

```
>>> e = Employee('John', 25)
```

라고 써주었을 때 name이나 age 같은 속성 변수의 이름이 명시적으로 드러나지는 않지만 그 e.name, e.age 변수들의 값이 초기화되는 것이다. 따라서

```
>>> e.name
'John'
>>> e.age
25
```

처럼 그 값을 사용할 수 있다.

이제 우리는 클래스(class), 인스턴스(instance), 속성(attribute)이라는 개념을 배웠고 그것을 파이썬에서 구현할 때 사용하는 class, self라는 키워드를 배웠다. 키워드 def는 함수를 정의할 때 사용하는 키워드로 새로운 것이 아니다. __init__() 함수의 이름은 단순히 이름붙이기 약속 중의 하나일 뿐이다.

6.3.4 속성 값에 접근하기

속성 변수 값을 초기화하여 객체를 생성하고 나면 그 값을 자유롭게 읽고 쓸 수 있다. 나중에도 얼마든지 그 값을 바꿀 수 있다. 즉,

```
>>> c = Car('red')
>>> c.color
'red'
>>> c.color = 'black'
>>> c.color
'black'
```

그러나 인스턴스의 속성 값을 마음대로 바꾸는 것은 커다란 프로그램을 작성할 때는 일관성에 문제를 일으킬 수 있으므로 주의해야 한다. 객체 지향 프로그래밍에서는 일반적으로 객체의 속성을 외부에서 자유롭게 바꾸는 것을 위험하다고 생각한다. 지금은

파이썬에서도 이러한 직접 접근을 막는 방법이 존재한다는 사실만 기억하고 뒤에서 이야기하도록 하자.

6.3.5 구조를 가지는 자료형

실제 데이터를 다루다 보면 하나의 클래스에서 인스턴스를 여러 개 만들게 된다. 만약 단 하나만 존재하는 대상이라면 굳이 클래스를 만들 필요가 없다. 클래스/인스턴스 개념을 도입한 이유는 동일한 부류에 속하는 개체가 여럿 존재하는 세계를 모사하기 위한 것이다. 그리고 동일한 부류에 속하더라도 개체마다 조금씩 다른 성격을 가지는 문제는 속성 개념을 통해 해결될 수 있다.

아래의 문구점 영수증 데이터를 생각해 보자. 하나의 항목은 메뉴, 단가, 수량, 금액으로 구성되어 있다. 이 영수증의 한 항목을 표현하기 위한 구조를 가진 자료형으로서 클래스를 이용할 수 있다.

메뉴	단가	수량	금액
연필깍이2홀	2000	1	2000
유니볼	1600	5	8000
랜케이블	2000	1	2000
5핀케이블	2800	2	5600
컷팅방안자	3200	1	3200

위 데이터의 한 항목을 표현하기 위한 클래스는 아래와 같이 정의할 수 있을 것이다. 이 클래스는 단순히 몇 개의 값을 묶어 놓은 것에 불과하다. 리스트, 순서짝(tuple), 사전(dict)으로 표현해도 사용법에 사소한 차이가 있을 뿐 본질적으로 차이가 없으며 namedtuple은 바로 이러한 경우를 위한 편리한 도구이다. 하지만 클래스를 만들어 보자.

```
class Item:
    def __init__(self, name, price, quantity, amount) :
        self.name = name
        self.price = price
        self.quantity = quantity
        self.amount = amount
```

데이터에는 여러 개의 인스턴스가 들어있으므로 리스트에 객체를 넣어 사용한다면 좋을 것이다. 예를 들면, 다음과 같다.

```
>>> shopping_list = []
>>> e1 = Item('연필깎이2홀', 2000, 1, 2000)
>>> shopping_list.append(e1)
>>> e2 = Item('유니볼', 1600, 5, 8000)
>>> shopping_list.append(e2)
```

이제 리스트 안에 들어있는 인스턴스를 하나 씩 꺼내서 그 속성값들에 대해 원하는 작업을 할 수 있다. 일단 다음처럼 리스트에서 인스턴스 꺼내서 제대로 들어있는지 확인해 보자.

```
>>> shopping_list[0].name
'연필깎이2홀'
```

그런데 여러 개의 인스턴스를 만들어서 리스트에 넣는 과정을 보면 각각의 인스턴스를 변수에 할당하였고 그러기 위해서 매번 변수의 이름을 생각해 내어야했다. 사실 리스트에 넣고 나면 변수의 이름은 필요가 없는 것이다. 이를 위해 이름이 없는 인스턴스(anonymous instance)를 만드는 것이 가능하다. 예를 들면, 다음과 같다.

```
>>> shopping_list = []
>>> shopping_list.append(Item('연필깎이2홀', 2000, 1, 2000))
>>> shopping_list.append(Item('유니볼', 1600, 5, 8000))
```

연습 6.1 영수증 데이터가 텍스트 파일로 주어졌을 때 파일을 읽어서 객체의 리스트를 만드는 프로그램을 작성하여라.

예제 6.3

패스트푸드 매장 관리 시스템 예제를 다시 보자. 이제 다음과 같은 직원 데이터 `emp.dat`를 읽어서 직원 인스턴스를 만들고 리스트에 넣어 보자. 이름, 시급, 주당근무시간, 나이로 구성된 데이터이다.

```
AIDAN    3.5     19      22
AMELIA   5.25    16      22
NOAH     4.5     24      25
ISABELLA         4.25    17      25
LIAM     4.5     23      23
```

앞선 예제에서 본 `Employee` 클래스를 확장해 보자. 다음처럼 할 수 있을 것이다.

```
class Employee:
    def __init__ (self, name, rate, hours, age):
        self.name = name
        self.rate = rate
        self.hours = hours
        self.age = age
```

클래스를 정의하면 일반적으로 독립적인 파일에 모듈로 만든 후에 `import` 해서 사용한다. 지금은 간단한 예제이니 그러한 부분은 생략하고 이야기해 보자. 한 파일에 클래스 정의와 다음 실행 코드를 함께 써서 테스트해 보아도 좋다.

```
file = open('emp.dat')

employees = []
for line in file:
    (name, rate, hours, age) = line.strip().split('\t')
    e = Employee(name, float(rate), float(hours), int(age))
    employees.append(e)
```

조금 전에 본 것처럼 이름없는 인스턴스도 가능하지만 보기 좋게 나누어서 써 보았다. 만약 인스턴스 생성과 리스트에 넣기 사이에 이 인스턴스에 해서 추가적인 작업을 해야 한다면 위처럼 나누어서 쓰는 수밖에 없다.

이제 데이터 파일의 내용을 모두 읽어서 리스트에 넣는 작업이 끝났다. 하고 싶은 일을 할 수 있다. 아마도 가장 기본적인 일은 데이터가 제대로 들어갔는지 확인해 보는 일일 것이다.

```
for e in employees:
    print(e.name, e.rate, e.hours, e.age)
```

속성들 중 rate와 hours는 숫자 데이터이므로 계산도 할 수 있다. 시급(rate)과 주당근무시간(hours)를 곱하면 주급이 계산된다. 다음과 같이 코드를 바꾸면 주급을 계산한 결과가 출력될 것이다.

```
for e in employees:
    print(e.name, e.rate * e.hours, e.age)
```

지금까지 구조를 가진 데이터로서 객체를 살펴보았다. 실세계의 존재하는 대상을 흉내내는 것이라는 차원에서 객체의 개념을 이해할 수 있다. 그런데 실세계의 존재들도 결국은 데이터이다. 그리고 구조를 가진 데이터이다. 테이블형의 데이터를 예로 들어 살펴보았기 때문에 무엇인가 매우 특수한 좁은 범위의 대상만을 다루는 것처럼 느껴질 수도 있다. 하지만 테이블이라는 형식의 표현력은 대단하다. 사실 어떤 대상, 어떤 정보, 어떤 데이터라도 테이블로 표현할 수 있다. 물론 우리는 점차 객체가 테이블보다 더 강력한 표현 도구임을 알게 될 것이다. 하지만 우선은 어떤 대상을 테이블로 표현해 보는 데에서 시작하는 것이 나쁘지 않을 것이다.

6.4 행위

6.4.1 속성과 행위

객체 지향 프로그래밍이라는 것은 실세계에 존재하는 객체(object)들을 컴퓨터 속에서 모사하는 방식의 프로그래밍이다. 예를 들어, 고양이가 객체가 될 수 있다. 우리가 동물이 나오는 게임을 만든다고 해보자. 그리고 고양이가 주요 동물이라고 하자. 실세계의 고양이는 품종, 털 색깔, 무늬, 눈 색깔, 등등 여러가지 속성을 가질 수 있다. 프로그램을 만들기 위해서는 그 중 의미있다고 생각되는 정보들을 골라 구현을 하게 된다. 우리의 게임에 등장하는 고양이는 색깔만 차이가 있고 고양이마다 이름이 있다고 하자. 다음처럼 될 것이다.

```
class Cat:
    def __init__ (self, color, name):
        self.color = color
        self.name = name
```

그런데, 실세계에 존재하는 프로그래밍의 대상(object)들을 보면 속성만 가지는 것이 아니다. 예를 들어, 고양이는 여러가지 행위(action)을 할 수 있다. 우리 게임의 고양이는 색깔과 이름이 있는 것이 다가 아니라 달릴 수도 있고, 잠을 잘 수도 있고, '야옹'하고 울 수도 있고, 무언가를 먹을 수도 있다고 하자. 속성의 경우와 마찬가지로 실세계의 고양이를 모사하기 위하여 우리가 의미 있다고 생각되는 행위(action)을 골라서 프로그램으로 구현하게 될 것이다.

6.4.2 메소드

한 객체의 속성은 변수를 이용해 프로그래밍하고 행위는 함수를 이용해 프로그래밍한다. 객체 내부에 들어 있는 함수를 메소드(method)라고 한다. 그냥 '함수'라고 부르지 않고 '메소드'라는 용어를 따로 이름을 붙인 것은 어느 정도 타당한 측면이 있다. 객체의 메소드는 엄밀한 의미에서의 순수한 함수와는 거리가 있다. 메소드들은 객체의 상태를 변화시킬 수 있다. 순수한 함수가 외부에 아무런 변화도 주지 않느 것과는 차이가 있다. 물론 객체의 상태를 변화시키지 않는 메소드도 있다. 이 경우에도 객체 안의 속성을 참조하여 무언가를 할 수 있다. 외부에서 보자만 인자를 통해 제공하지 않은 어떤 정보를 이용하는 셈이다. 전형적인 함수가 인자로 받은 정보만을 이용해서 어떤 일을 하고 그 결과를 리턴하는 완전히 닫힌 형태인 것과는 차이가 있는 것이다.

이제 고양이 클래스에 행위를 추가해 보자. 메소드는 파이썬에서 함수를 만들 때 사용하는 것과 동일한 형식으로 만들 수 있다. 차이가 있다면 인자로 `self`라는 것을 써준다는 것이다. 우선은 어떤 식으로 작동하는지 살펴보자.

```
class Cat:
    def __init__ (self, color, name):
        self.color = color
        self.name = name

    def run (self):
        print(self.name + " is running!")

    def sleep (self):
        print("Zzzzz")

    def mew(self):
        print("Mew~ mew~")

    def eat(self, food):
        print(self.name + " is eating " + food)
```

위 내용을 그대로 옮겨 animal.py라는 파일로 저장한 후 대화형 모드로 들어가서 다음 내용을 테스트해 보자.

```
>>> from animal import *
>>> felix = Cat("black", "Felix")
>>> felix.color
'black'
>>> felix.name
'Felix'
>>> felix.run()
Felix is running!
>>> felix.sleep()
Zzzzz
>>> felix.mew()
Mew~ mew~
>>> felix.eat("sardine")
Felix is eating sardine
```

지금은 단순하게 메소드를 간단한 메시지를 출력하는 형태로 작성했지만 실제 게임 프로그램을 만든다면 각각의 메소드의 내용에 여러가지 그래픽과 사운드를 제어하는 코드를 작성하여 해당 고양이 인스턴스가 화면 상에서 움직이고 소리를 내게 할 수 있을 것이다.

6.4.3 self

메소드를 정의하기 위한 형식적인 측면을 한번 살펴보자. 코드의 일부분을 보자. 아래 예에서 점 세 개(...)는 정말로 점을 세 개 찍으라는 뜻이 아라 다른 코드가 있는데 생략했다는 표시이다.

```
class Cat:

    ## (...)

    def run (self):
        print(self.name + " is running!")
```

라고 클래스를 작성했기 때문에

```
>>> felix.run()
Felix is running!
```

라고 인스턴스를 사용할 수 있다. 앞서 self 키워드를 속성에 붙여 사용하는 것을 설명했는데 메소드에서도 동일한 목적으로 사용하는 것이다. 파이썬의 객체 지향 시스템은 인스턴스 이름 다음에 점을 찍고 메소드를 사용하면 자기 자신을 자동으로 첫번째 인자로 준다. 이에 보조를 맞추기 위해 클래스에서 메소드를 정의할 때 함수의 첫번째 인자로 self를 주는 것이다. run() 메소드는 인스턴스를 인자로 넘겨 받기 때문에 인스턴스가 가지고 있는 속성들을 이용하여 프로그램을 짤 수 있다.

6.4.4 메소드를 이용한 자료 처리

앞서 보았던 문구점 영수증 데이터를 다시 생각해 보자. 데이터 파일을 읽어서 하나의 레코드를 하나의 인스턴스로 만들고 리스트에 넣었다. 한 레코드의 필드는 각각 그 인스턴스의 속성이 되었다.

메뉴	단가	수량	금액
연필깍이2홀	2000	1	2000
유니볼	1600	5	8000
랜케이블	2000	1	2000
5핀케이블	2800	2	5600
컷팅방안자	3200	1	3200

그런데 여기에서 금액은 사실 단가와 수량의 곱으로 얻어지는 것이다. 이 값을 직접 입력하는 것은 적절하지 않다. 단가와 수량만 객체의 속성으로 주고 금액은 계산하여 얻는 것이 적절하다. 이러한 경우 금액은 메소드를 이용하여 표현할 수 있다.

```
class Item:
    def __init__(self, name, price, quantity, amount) :
        self.name = name
        self.price = price
        self.quantity = quantity
```

```
    def amount(self) :
        return self.price * self.quantity
```

예제 6.4

패스트푸드 매장 문제를 이어서 보자. 앞에서 우리는 속성만 가지는 클래스를 만들었었다. 그런데 우리는 일부 필드를 이용하여 연산을 하고 싶었다. 시급(rate)와 주당근무시간(hours)를 곱하여 주급을 계산하고 싶었다. 그리고 이 계산을 객체의 외부에서 수행했다.

이런 계산이 Employee 클래스에 대해 일반적으로 반복적으로 행해져야 하는 일이라면 메소드로 만드는 것이 좋다. 다음과 같다. 점 세 개는 생략 표시이다. 앞서 작성한 코드가 들어간다는 의미이다. 그 뒤에 다음 메소드 정의 부분만 추가하면 된다.

```
class Employee:
    # (...)
    def getSalary(self):
        return self.rate * self.hours
```

메소드의 이름을 붙이는 데에 파이썬 자체의 형식적인 제약이 있는 것은 아니다. 함수를 작성할 때 이름을 마음대로 붙일 수 있듯이 메소드의 이름도 마음대로 붙일 수 있다. 물론 밑줄(__)로 시작하는 이름은 파이썬 언어 자체에서 특별히 형식적으로 약속이 된 것이니 마음대로 쓰면 안 된다.

이와 같이 주급을 계산하는 메소드를 정의하고 나면 앞서 for 문을 돌릴 때 직접 계산했던 부분을 다음과 같이 바꿀 수 있다.

```
for e in employees:
    print e.name, e.getSalary(), e.age
```

이 예는 너무 단순한 계산이라서 메소드를 만들어서 사용하는 것을 통해 그리 많은 것을 얻을 수 없다. 하지만, 십여 개의 속성값을 가지고 있는 객체에서 속성들 사이의 복잡한 계산을 거쳐서 어떤 특성값을 계산해야 하는 경우라면 메소드를 만들어 사용하는 것이 편리할 것이다.

메소드의 이름을 어떻게 붙일 것인가 하는 문제를 이야기해 보자. 일반적인 객체 지향 프로그래밍의 관습을 따르면 어떤 값을 알려주는 메소드의 이름을 getAmount(), getSalary() 등과 같이 붙인다. 관습일 뿐이기 때문에 지키지 않아도 에러가 나지는 않는다. 하지만 관습을 지킨다면 의사소통이 편리하니 따르는 것이 좋다. 한편 이런 관습이란 많은 프로그래머들이 오랜 경험을 통해 이렇게 하는 것이 제일 좋더라라는 합의에 도달해 성립된 것이다. 실질적인 측면에서도 관습을 따르는 것이 효율적이다. 어떤 객체에서 주로 속성값을 뽑아보려고 할 때 메소드 이름을 get으로 시작하는 관행이

있다. 이 관행은 널리 퍼져 있기 때문에 심지어 'get 함수'라는 용어를 사용하는 사람도 많다. 'get 함수'란 어떤 객체가 가지고 있는 특정 정보의 값을 돌려주는 메소드를 말한다.

다만 여기에서 일종의 '경고'를 하나 하자면, 파이썬에서는 'get'을 붙이는 것과는 다른 방법을 제공한다는 것이다. 나중에 이 부분은 따로 이야기할 것이다. 덧붙이면, 오랜 관습이라는 것이 편리하고 남들과도 충돌하지 않고 다 그렇게 하는 이유가 있고 효율적이고 등등 말하지만 어떤 관습은 너무 형식에 얽매여서 과도한 의식을 치른다는 의심이 드는 경우가 있다. 우리는 현실 세계에서 그러한 일이 벌어진다는 것을 알고 있다. 프로그래밍에서도 마찬가지여서 때로는 기존의 관습에서 불편함이 발견될 때 그것을 타파하기 위한 새로운 아이디어가 나타난다. 파이썬에서는 이런 'get' 함수를 대체하는 새로운 아이디어가 도입되었다. 뒤에서 이야기해 보자.

6.5 객체지향 이론

클래스를 정의하고 그것으로부터 인스턴스를 생성하여 사용함으로써 객체지향 프로그래밍을 할 수 있다. 객체가 가지는 속성은 변수로, 객체가 할 수 있는 행위는 메소드로 작성할 수 있다. 형식적으로는 이것이 전부라고도 할 수 있지만 이런 틀을 이용하여 어떻게 프로그래밍을 할 것인가에 대해서 이야기를 해야한다. 객체지향 프로그래밍을 하기 위한 이론의 핵심으로 널리 이야기되는 것은 추상화(abstraction), 캡슐화(encapsulation), 상속(inheritance), 다형성(polymorphism)의 개념이다.

6.5.1 추상화

추상화라는 것은 실세계의 어떤 대상을 컴퓨터를 이용하여 흉내내기 위해서 거쳐야할 과정이다. 객체지향 방법론이 아니더라도 실세계의 문제를 프로그래밍을 통해서 해결하려고 한다면 추상화가 필요하다. 예를 들어, 실제 고양이는 여러가지 속성을 가지고 있고 여러가지 행위를 할 수 있지만 그중 우리에게 주어진 문제를 해결하기 위해서 필요한 정보가 단지 색깔뿐이라고 한다면 다음과 같이 고양이 클래스를 디자인할 수 있을 것이다.

```
class Cat:
    def __init__(self, color):
        self.color = color

felix = Cat('black')
```

고양이의 색깔이라는 정보를 표현하기 위해서 `color` 변수를 사용하였다. 추상화라는 것은 일종의 이름 붙이기라고도 할 수 있다.

6.5.2 캡슐화

캡슐화라는 것은 캡슐 안에 집어넣는다는 의미이다. 클래스라는 것이 여러 개의 변수와 메소드들을 하나로 묶는다는 점에서 캡슐화라고 이해할 수도 있을 것이다. 필요한 것들을 모두 모아 한 클래스에 담아놓았다는 점에서 클래스를 캡슐이라 할 수 있는 것이다. 그러나 일반적으로 캡슐화는 단순히 모아놓았다는 차원을 넘어서 그것이 닫혀 있음 또는 숨겨져 있음을 뜻한다. 한 객체가 감싸고 있는 데이터를 외부에서 마음대로 손댈 수 없도록 감추고 보호해야 한다는 의미이다. 데이터에 대한 접근이나 수정이 외부에서 직접적으로 이루어져서는 안 되며 주어진 메소드를 통해서만 이루어지도록 하는 것이 좋은 습관이다. 파이썬에서는 기본적으로 객체 외부에서 변수들을 마음대로 제어하는 것이 가능하며 간단한 프로그램을 작성할 때에는 자유로운 제어가 편리한 경우도 있다. 그러나 조금이라도 프로그램이 커지고 복잡해 지면 캡슐화의 필요성이 대두된다.

예를 들어 사각형 객체가 있고 가로와 세로 길이 속성을 가지며 그에 따라 넓이를 구할 수 있다고 하자. 다음과 같이 사각형 클래스를 설계하였다고 하자.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height
```

다음 예는 외부에서 한 객체의 속성을 직접 건드릴 때 생길 수 있는 문제를 보여준다. 초기화할 때 넓이가 계산되고 이후에 너비나 높이 속성을 바꿀 때에는 넓이가 재계산되지 않는다. 게다가 넓이에 아무 값이나 강제로 넣을 수도 있다.

```
>>> r1 = Rectangle(5, 6)
>>> r1.area
30
>>> r1.width = 3
>>> r1.width, r1.height
(3, 6)
>>> r1.area
30
>>> r1.area = 1000
>>> r1.area
1000
```

이 문제를 피하기 위하여 다음과 같이 넓이를 속성이 아니라 메소드로 정의할 수 있다. 너비나 높이 속성을 바꾸어도 문제가 되지 않는다. 넓이 메소드는 항상 현재의 너비와 높이를 이용하여 계산되기 때문이다.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
```

```
        self.height = height

    def area(self):
        return self.width * self.height
```

이 방법은 틀린 계산 결과를 내지는 않지만 비효율적이다. 사각형의 넓이를 사용하려고 할 때마다 매번 다시 계산을 하기 때문이다. 너비와 높이가 변하지 않았다면 넓이도 변하지 않으므로 다시 계산할 필요가 없다.

이런 문제를 피하기 위하여 객체의 속성값을 외부에서 직접 접근하지 않고 특정한 메소드를 이용하여 접근하는 방법이 있다. 보통 속성 값을 읽을 수 있게 해주는 것을 getter, 속성 값을 바꿔 쓸 수 있게 해주는 것을 setter라고 부른다. 예를 들면 다음과 같이 될 것이다.

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.area = width * height

    def getWidth(self):
        return self.width

    def getHeight(self):
        return self.height

    def getArea(self):
        return self.area

    def setWidth(self, width):
        self.width = width
        self.update()

    def setHeight(self, height):
        self.height = height
        self.update()

    def update(self):
        self.area = self.width * self.height
```

사각형의 너비와 높이를 직접 바꾸지 않고 set 함수를 통해서만 바꾸면 넓이는 항상 올바른 값으로 갱신된다. 그러나 위와 같은 코드는 사용자가 너비와 높이를 강제로 바꾸는 것을 막지는 못한다. 한편 사용자가 속성에 접근하기 위하여 항상 메소드 형태를 사용해야 하는 것도 불편하다. 이것은 다른 객체 지향 프로그래밍 언어들에서는 전형적인 형태의 코드이다. 그리고 사용자가 너비와 높에 직접 접근하지 못하게 막는 장치도 제공한다. 파이썬에서는 이와 같은 코드를 선호하지 않는다.

파이썬에서는 캡슐화를 위한 장치로 우선 변수나 메소드 이름을 밑줄 문자 두 개로 시작하면 외부에서 사용할 수 없도록 약속하고 있다. 그리고 이렇게 숨겨진 속성을 실제로는 메소드를 통해서 접근하지만 사용자는 마치 속성 변수인 것처럼 사용할 수 있게 해주는 장치를 마련해 놓고 있다. 속성 읽기 메소드 앞에는 `@property`를 붙이고 속성 쓰기 메소드 앞에는 `@`*attributename*`.=setter=`를 붙이면 된다.

```
class Rectangle:
    def __init__(self, width, height):
        self.__width = width
        self.__height = height
        self.__area = width * height

    @property
    def height(self):
        return self.__height

    @property
    def width(self):
        return self.__width

    @property
    def area(self):
        return self.__area

    @height.setter
    def height(self, height):
        self.__height = height
        self.__update()

    @width.setter
    def width(self, width):
        self.__width = width
        self.__update()

    def __update(self):
        self.__area = self.__height * self.__width
```

이렇게 정의된 사각형 클래스를 사용해 보자. 클래스 정의에서는 메소드로 작성되어 있지만 외부에서는 마치 속성 변수인 것처럼 보인다.

```
>>> r = Rectangle(3, 4)
>>> r.height, r.width, r.area
(4, 3, 12)
```

이번에는 높이를 바꾸어 보자. 넓이도 함께 자동으로 갱신된다.

```
>>> r.height = 10
>>> r.height, r.width, r.area
(10, 3, 30)
```

넓이를 강제로 바꾸어 보자. 넓이를 위한 setter가 정의되어 있지 않으므로 불가능하다.

```
>>> r.area = 1000
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

연습 6.2 패스트푸드 매장 문제의 Employee 클래스를 캡슐화해 보아라. 앞에서는 객체지향 프로그래밍에서 오랜 관습에 따라 getSalary()라는 메소드를 만들었었다. 여기에서는 salary라는 속성으로 접근할 수 있도록 설계해 보아라. 시급이나 시간이 변경되는 것에도 대응할 수 있어야 한다.

6.5.3 상속

클래스를 체계적으로 디자인할 수 있도록 해주는 개념의 하나가 상속(inheritance)이다. 기존에 정의된 클래스를 부모로 하여 그것의 필드(속성)와 메소드를 물려받는다는 의미에서 상속이라고 한다. 코드의 재사용에 해당된다. 상속이라는 관계를 설명하기 위해서는 부모 클래스 또는 상위 클래스라는 개념과 자식 클래스 또는 파생 클래스라는 개념이 필요하다.

예를 들어, 앞서 정의하였던 고양이 클래스를 생각해보자.

```
class Cat:
    def __init__ (self, name, color):
        self.color = color
        self.name = name

    def run (self):
        print(self.name + " is running!")

    def sleep (self):
        print("Zzzzz")

    def mew(self):
        print("Mew~ mew~")

    def eat(self, food):
        print(self.name + " is eating " + food)
```

이때 페르시아 고양이, 터키 고양이, 체셔 고양이라는 클래스가 필요하며 이들은 고양이라는 공통적인 속성과 메소드를 공유하면서 나름대로의 독자적인 특성도 가진다고 하자. 페르시아 고양이는 말을 할 수 있고, 터키 고양이는 수영을 할 수 있고, 체셔 고양이는 몸이 안 보이게 사라졌다 나타나는 능력이 있다고 하자. 다음과 같이 Cat를 부모 클래스로 하는 자식 클래스를 정의할 수 있다.

```
class PersianCat(Cat):
    def speak(self, words):
        print(self.name, "says:", words)

class TurkishCat(Cat):
    def swim(self):
        print(self.name + " loves swimming")

class CheshireCat(Cat):
    def __init__(self, name, color):
        Cat.__init__(self, name, color)
        self.status = 'visible'

    def disappear(self):
        self.status = 'invisible'
        print(self. name + " is dispearing slowly ...")

    def appear(self):
        self.status = 'visible'
        print(self.name + " is visible now.")
```

이제 이 클래스를 사용하여 인스턴스를 만들어 사용해 보자. 다음과 같이 PersianCat 객체는 Cat 객체가 가진 속성과 메소드를 모두 물려받았을 뿐만 아니라 자신만의 메소드도 가지고 있다.

```
>>> tinkles = PersianCat('Mr. Tinkles', 'white')
>>> tinkles.mew()
Mew~ mew~
>>> tinkles.run()
Mr. Tinkles is running!
>>> tinkles.speak("Evil does not wear a bonnet!")
Mr. Tinkles says: Evil does not wear a bonnet!
```

이와 다른 CheshireCat 객체 역시 Cat이 가지고 있는 것들을 물려받았을 뿐만 아니라 자신만의 속성과 메소드를 추가적으로 가지고 있다. 그리고 당연히 PersianCat 객체의 고유한 메소드는 가지고 있지 않다.

```
>>> chess = CheshireCat('Chess', 'tabby')
>>> chess.disappear()
Chess is dispearing ...
>>> chess.status
'invisible'
>>> chess.appear()
Chess is visible now.
>>> chess.status
'visible'
>>> chess.speak("We're all mad here.")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: CheshireCat instance has no attribute 'speak'
```

6.5.4 다형성

다형성은 다수의 형태를 가진다는 의미이다. 하나의 표현이 상황에 따라 그 의미가 달라질 수 있는 경우를 가리킨다. 우리는 이미 다형성을 잘 알고 있다. 덧셈의 의미가 무엇에 대한 것이냐에 따라 달라지는 것이 바로 다형성의 예이다.

```
>>> 3 + 4
7
>>> "abc" + "def"
'abcdef'
```

객체지향에서 다형성은 흔히 하위유형에서의 다형성을 가리킨다. 다음과 같이 동일한 이름의 메소드를 하위 클래스에 따라 다르게 정의할 수 있다.

```
class Canis:
    def bark(self):
        pass

class Dog(Canis):
    def bark(self):
        print("woof")

class Wolf(Canis):
    def bark(self):
        print("howl")
```

다음과 같이 다르게 작동한다.

```
>>> dog = Dog()
>>> dog.bark()
woof
>>> wolf = Wolf()
>>> wolf.bark()
howl
```

서로 다른 여러 객체에 대해 동일한 메소드를 사용하되 다르게 작동하는 것은 다음과 같은 경우에 그 잇점이 있다.

```
>>> animals = [Dog(), Wolf(), Wolf(), Dog(), Dog()]
>>> for a in animals:
...     a.bark()
...
woof
howl
howl
woof
woof
```

6.6 객체 합성

객체지향 프로그램에서 객체 합성(object composition)은 단순한 객체들을 모아 더 복잡한 객체를 만드는 방법이다. 예를 들어, 자동차라는 대상을 흉내내려는 경우를 생각해보자. 자동차는 엔진, 바퀴, 운전대, 문, 좌석 등 여러가지 부분(component)으로 이루어져있다. 자동차라는 대상을 단 하나의 클래스로 추상화하는 것은 매우 복잡할 것이다. 엔진을 위한 클래스, 바퀴를 위한 클래스, 운전대를 위한 클래스를 따로 디자인하고 이들을 합쳐서 자동차라는 클래스를 만든다면 덜 복잡할 것이다. 그리고 이렇게 요소들을 분리해 놓는다면 어떤 바퀴 클래스를 서로 다른 자동차를 조립하는 데 이용할 수도 있을 것이다. 즉, 단순히 자동차 클래스가 덩치가 크기 때문에 분리해서 코드를 작성하는 것이 아니다. 실세계에서 그러하듯 모듈화를 하여 부품을 따로 만들 수 있게 설계를 하고 표준화된 각 부품들은 여러 자동차에서 사용할 수 있게 하는 것이 목적이다. 자동차를 다른 부분들로 이루어진 합성/구성(composition)으로 설계한 예를 보자.

```
class Car:
    def __init__(self, color, fule, speed):
        self.body = CarBody(color)
        self.engine = Engine(fule, speed)
        self.wheel = Wheel(wheel_factor)

    @property
    def speed(self) :
        return self.engine.speed

class CarBody:
    def __init__(self, color):
        self.color = color

class Engine:
    def __init__(self, fule, speed):
        self.fule = fule
        self.speed = speed
```

이것을 다음과 같이 사용할 수 있다.

```
>>> mycar = Car("black", "gasoline", 100)
>>> mycar.speed
100
```

합성(composition)을 집합(aggregation)이라는 개념과 대비하여 구별하기도 한다. 어떤 객체들을 모아놓은 집합 객체를 만드는 경우이다. 예를 들어, 동물원이라는 클래스가 있다고 하자. 동물원이 코끼리, 기린, 호랑이, 사자들로 이루어져 있다고 하자. 그러나, 동물원과 코끼리의 관계는 자동차와 바퀴의 관계와는 다르다. 코끼리는 동물원이라는 집합의 한 원소일 뿐이다. 코끼리 객체는 그 자체로서 독립성을 가진다.

```
class Zoo:
    def __init__(self, animals):
        self.__animals = animals

    def append(self, animal):
        self.__animals.append(animal)

    def pop(self):
        self.__animals.pop()

    def show(self):
        for animal in self.__animals:
            print(animal.name)

class Animal:
    def __init__(self, name):
        self.name = name

class Elephant(Animal):
    pass

class Giraff(Animal):
    pass

class Monkey(Animal):
    pass

class Lion(Animal):
    pass
```

다음과 같이 사용할 수 있다.

```
>>> nala = Elephant("Nala")
>>> twiga = Giraff("Twiga")
>>> furaha = Monkey("Furaha")
>>> kiara =  Lion("Kiara")
>>> zoo = Zoo([nala, twiga, furaha])
>>> zoo.append(kiara)
>>> zoo.show()
Nala
Twiga
Furaha
Kiara
>>> zoo.pop()
>>> zoo.show()
Nala
Twiga
Furaha
```

합성(composition)과 집합(aggregation)을 자동차와 동물원이라는 예를 들어 설명

했지만 컴퓨터는 자동차와 엔진의 관계와 동물원과 코끼리의 관계가 어떻게 다른지 모른다. 자동차를 몸체, 엔진, 바퀴의 집합으로 프로그래밍할 수도 있고 동물원을 코끼리, 사자의 합성으로 프로그래밍할 수도 있다. 합성과 집합의 차이는 기술적으로는 어디에서 인스턴스가 만들어지느냐에 따라 구별되는 것이다. 합성의 예에서는 자동차 개체의 내부에서 엔진의 인스턴스가 만들어졌기 때문에 자동차가 사라지는 순간 엔진도 사라진다. 동물원의 예의 경우에는 코끼리 Nala는 스스로 존재한다. 단지 동물원의 구성원으로 참여할 뿐이다. 동물원이 사라지더라도 코끼리 Nala는 여전히 존재한다.

연습 6.3 패스트푸드 매장 문제이다. 이번에는 각 매장을 `Restaurant`라는 클래스로 만들어 보자. 다음과 같은 형태가 될 것이다.

```
class Restaurant:
    def __init__(self, name):
        self.name = name
        self.employees = []

    def append(self, emp):
        self.employees.append(emp)

class Employee:
    # (...)
```

앞선 예제서 보았던 직원 정보 텍스트 파일을 읽어 `Employee` 객체들을 만들고 직원 객체들로 구성된 `Restaurant` 객체를 만들어 보아라.

직원들로 구성된 매장 객체를 만드는 것에 성공하였다면 이제 `Restaurant` 객체에 기능을 추가해 보자. 즉, 필요한 메소드들을 추가해 보자. 직원의 수를 알려주는 메소드, 시급의 평균을 알려주는 메소드, 지급해야할 임금의 총액을 알려주는 메소드 등을 생각할 수 있을 것이다. 어떤 기능을 넣을지 먼저 생각해 보아라. 그리고 그 기능을 위한 메소드의 인터페이스를 생각해 보라. 사용을 어떻게 할 것인지 먼저 생각해 보라는 의미이다. 그 후에 실제로 구현해 보아라.

6.7 디자인 패턴

디자인 패턴(design pattern)은 원래는 건축학의 개념으로 Christopher Alexander에 의해 제안된 것이다. 이것은 전산학자들이 수용하여 재사용 가능한 객체 지향 소프트웨어를 개발하는 방법론으로 발전시켰다. 객체 지향 이론만으로 프로그램을 잘 설계하는 것은 쉽지 않다. 프로그래밍을 할 때 자주 부딪히게 되는 문제들을 패턴으로 정리해놓은 것이라고 할 수 있다.

굳이 번역하자면 '설계 유형'이다. 또는 옷을 만들 때 본을 뜬다고 하는 말을 따르면 '설계 본'이다. 비슷한 것을 통해 이야기해 보자면, 워드프로세서 프로그램에서 템플릿 혹은 양식 같은 것이다. 워드프로세서의 기능을 속속들이 알고 있는 사람이라도 텅 빈 문서를 띄워놓고 어떤 문서를 밑바닥부터 만들어 가는 것은 쉽지 않을 것이다. 그래서 우리는 '서식' 파일을 사용한다. 이력서, 월간 일정표, 주간 일정표, 견적석, 초정장, 등등 미리 만들어진 서식에 필요한 내용을 채워넣어 문서를 만들 수 있다.

객체를 설계(디자인)할 때 그 목적과 용도에 따라 미리 만들어진 틀(패턴)을 가져다 쓴다는 것이다. 프로그래밍할 때 자주 등장하는 문제의 유형과 해결책을 정리하여 놓고 그 중에서 골라서 쓴다는 것이 디자인 패턴의 아이디어이다. 클래스의 이름과 변수와 메소드의 인터페이스 등을 자신이 필요로 하는 객체에 맞추어 바꾸지만 전체 틀은 정해진 것을 사용하는 것이다. 물론 문서 서식처럼 빈칸만 채워넣는 정도로 가능한 것은 아니다. 어떤 디자인 패턴이 필요한지 이해하고 그 구조에 맞게 자신의 객체를 설계할 수 있어야한다.

실세계의 문제를 해결하는 상당한 규모의 소프트웨어를 객체지향 프로그래밍으로 개발하려고 한다면 디자인 패턴을 익혀두는 것이 도움이 될 것이다.