PythonStudyBook

작성원문: 점프투파이썬(https://wikidocs.net/book/1)
작성자: 김철민&이정우

본 문서는 점프투파이썬 중 일부를 발췌한 것으로 한양대학교 경상대학 학문내실화 사업 '빅데이터 분석역량 강화를 위한 연구모임 및 세미나 운영' 참여 학생들을 위해 작성되었습니다.
연구모임 참여학생들은 본 문서를 통해 파이썬의 기본적인 내용인 자료형과 제어문을 파악하기 바라며 보다 자세한 내용은 책을 통해 학습하길 바랍니다.

☐ 파이썬이란?

파이썬(Python)은 1990년 암스테르담의 귀도 반 로섬(Guido Van Rossum)이 개발한 인터프리터 언어이다. 귀도는 파이썬이라는 이름을 자신이 좋아하는 코미디 쇼인 "몬티 파이썬의 날아다니는 서커스(Monty Python's Flying Circus)"에서 따왔다고 한다. 파이썬의 사전적인 의미는 고대 신화에 나오는 파르나소스 산의 동굴에 살던 큰 뱀을 뜻하며, 아폴로 신이 델파이에서 파이썬을 퇴치했다는 이야기가 전해지고 있다. 대부분의 파이썬 책 표지와 아이콘이 뱀 모양으로 그려져 있는 이유가 여기에 있다.

☐ 파이썬의 특징

1) 인간다운 언어

프로그래밍이란 인간이 생각하는 것을 컴퓨터에 지시하는 행위라고 할 수 있다. 앞으로 살펴볼 파이썬 문법에서도 보게 되겠지만 파이썬은 사람이 생각하는 방식을 그대로 표현할 수 있는 언어이다. 따라서 프로그래머는 굳이 컴퓨터의 사고 체계에 맞추어서 프로그래밍을 하려고 애쓸 필요가 없다. 이제 곧 어떤 프로그램을 구상하자마자 머릿속에서 생각한 대로 술술 써 내려가는 여러분의 모습에 놀라게 될 것이다.

아래 소스 코드를 보면 이 말이 쉽게 이해될 것이다.

```
if 4 in [1,2,3,4]: print("4가 있습니다")
```

위의 예제는 다음처럼 읽을 수 있다:

"만약 4가 1,2,3,4중에 있으면 "4가 있습니다"를 출력한다."
프로그램을 모르더라도 직관적으로 무엇을 뜻하는지 알 수 있지 않는가? 마치 영어 문장을 읽는 듯한 착각에 빠져든다.

2) 쉬운 문법

어려운 문법과 수많은 규칙에 둘러싸인 언어에서 탈피하고 싶지 않은가? 파이썬은 문법 자체가 아주 쉽고 간결하며 사람의 사고 체계와 매우 닮아 있다. 배우기 쉬운 언어, 활용하기 쉬운 언어가 가장 좋은 언어가 아닐까? 유명한 프로그래머인 에릭 레이먼드(Eric Raymond)는 파이썬을 공부한 지 단 하루 만에 자신이 원하는 프로그램을 작성할 수 있었다고 한다.

3) 강력한 기능

오픈 소스1인 파이썬은 당연히 무료이다. 사용료 걱정없이 언제 어디서든 파이썬을 다운로드하여 사용할 수 있다.

또한 프로그래머는 만들고자 하는 프로그램의 대부분을 파이썬으로 만들 수 있다. 물론 시스템 프로그래밍이나 하드웨어 제어와 같은 매우 복잡하고 반복 연산이 많은 프로그램은 파이썬과 어울리지 않는다. 하지만 파이썬은 이러한 약점을 극복할 수 있게끔 다른 언어로 만든 프로그램을 파이썬 프로그램에 포함시킬 수 있다.

파이썬과 C는 찰떡궁합이란 말이 있다. 즉, 프로그램의 전반적인 뼈대는 파이썬으로 만들고, 빠른 실행 속도를 필요로 하는 부분은 C로 만들어서 파이썬 프로그램 안에 포함시키는 것이다(정말 놀라우리만치 영악한 언어가 아닌가). 사실 파이썬 라이브러리2들 중에는 순수 파이썬만으로 제작된 것도 많지만 C로 만들어진 것도 많다. C로 만들어진 것들은 대부분 속도가 빠르다.

4) 간결함

귀도는 파이썬을 의도적으로 간결하게 만들었다. 만약 펄(Perl)과 같은 프로그래밍 언어가 100가지 방법으로 하나의 일을 처리할 수 있다면 파이썬은 가장 좋은 방법 1가지만 이용하는 것을 선호한다. 이 간결함의 철학은 파이썬 문법에도 그대로 적용되어 파이썬 프로그래밍을하는 사람들은 잘 정리되어 있는 소스 코드를 볼 수 있다. 다른 사람이 작업한 소스 코드도 한눈에 들어와 이해하기 쉽기 때문에 공동 작업과 유지 보수가 아주 쉽고 편하다.

다음은 파이썬 프로그램의 예제이다.

```
# simple.py
languages = ['python', 'perl', 'c', 'java']

for lang in languages:
    if lang in ['python', 'perl']:
        print("%6s need interpreter" % lang)
    elif lang in ['c', 'java']:
        print("%6s need compiler" % lang)
    else:
        print("should not reach here")
```

이 예제는 프로그래밍 언어를 판별하여 그에 맞는 문장을 출력하는 파이썬 프로그램 예제이다. 다른 언어들에서 늘 보게 되는 단락을 구분하는 괄호({ }) 문자가 보이지 않는 것을 확인할 수 있다. 또한 줄을 참 잘 맞춘 코드라는 것도 알 수 있다. 파이썬 프로그램은 줄을 맞추지 않으면 실행이 되지 않는다. 코드를 예쁘게 작성하려고 줄을 맞추는 것이 아니라 실행이 되게 하려면 꼭 줄을 맞추어야 하는 것이다. 이렇듯 줄을 맞추어 코드를 작성하는 행위는 가독성에 크게 도움이 된다.

□ 파이썬으로 할 수 있는 것들

1) 시스템 유틸리티 제작

파이썬은 운영체제(윈도우, 리눅스 등)의 시스템 명령어들을 이용할 수 있는 각종 도구를 갖추고 있기 때문에 이를 바탕으로 갖가지 시스템 유틸리티1를 만드는 데 유리하다. 실제로 여러분은 시스템에서 사용 중인 서로 다른 유틸리티성 프로그램들을 하나로 뭉쳐서 큰 힘을 발휘하게 하는 프로그램들을 무수히 만들어낼 수 있다.

2) GUI 프로그래밍

GUI(Graphic User Interface) 프로그래밍이란 쉽게 말해 윈도우 창처럼 화면을 보며 마우스나 키보드로 조작할 수 있는 프로그램을 만드는 것이다. 파이썬으로 GUI 프로그램을 만드는 것은 다른 언어를 이용해 만드는 것보다 훨씬 쉽다. 대표적인 예로 파이썬 프로그램을 설치할때 함께 설치되는 기본 모듈인 Tkinter(티케이인터)를 이용해 만드는 GUI 프로그램을 들 수 있다. 실제로 Tkinter를 이용한 파이썬 GUI 프로그램의 소스 코드는 매우 간단하다. Tkinter를 이용하면 단 5줄의 소스 코드만으로도 윈도우 창을 띄울 수 있다. 놀랍지 않은가!

3) C/C++와의 결합

파이썬은 접착(glue) 언어라고도 부르는데, 그 이유는 다른 언어들과 잘 어울려 다른 언어와 결합해서 사용할 수 있기 때문이다. C나 C++로 만든 프로그램을 파이썬에서 사용할 수 있으며, 파이썬으로 만든 프로그램 역시 C나 C++에서 사용할 수 있다.

4) 웹 프로그래밍

일반적으로 익스플로러나 크롬, 파이어폭스와 같은 브라우저를 이용해 인터넷을 사용하는데, 누구나 한 번쯤 웹 서핑을 하면서 게시판이나 방명록에 글을 남겨 본 적이 있을 것이다. 그러한 게시판이나 방명록을 바로 웹 프로그램이라고 한다. 파이썬은 웹 프로그램을 만들기에 매우 적합한 도구이며 실제로 파이썬으로 제작된 웹사이트는 셀 수 없을 정도로 많다.

5) 수치 연산 프로그래밍

사실 파이썬은 수치 연산 프로그래밍에 적합한 언어는 아니다. 수치가 복잡하고 연산이 많다면 C같은 언어로 하는 것이 더 빠르기 때문이다. 하지만 파이썬에는 Numeric Python이라는 수치 연산 모듈이 제공된다. 이 모듈은 C로 작성되었기 때문에 파이썬에서도 수치 연산을 빠르게 할 수 있다.

6) 데이터베이스 프로그래밍

파이썬은 사이베이스(Sybase), 인포믹스(Infomix), 오라클(Oracle), 마이에스큐엘(MySQL), 포스트그레스큐엘(PostgreSQL) 등의 데이터베이스에 접근할 수 있게 해주는 도구들을 제공한다.

또한 이런 굵직한 데이터베이스를 직접 이용하는 것 외에도 파이썬에는 재미있는 모듈이 하나 더 있다. 바로 피클(pickle)이라는 모듈이다. 피클은 파이썬에서 사용되는 자료들을 변형없이 그대로 파일에 저장하고 불러오는 일들을 맡아 한다. 이 책에서는 외장 함수에서 피클을 어떻게 사용하고 활용하는지에 대해서 알아본다.

7) 데이터 분석, 사물 인터넷

파이썬으로 만들어진 판다스(Pandas)라는 모듈을 이용하면 데이터 분석을 더 쉽고 효과적으로 할 수 있다. 데이터 분석을 할 때 아직까지는 데이터 분석에 특화된 "R"이라는 언어를 많이 사용하고 있지만, 판다스가 등장한 이후로 파이썬을 이용하는 경우가 점점 증가하고 있다. 사물 인터넷 분야에서도 파이썬은 활용도가 높다. 한 예로 라즈베리파이(Raspberry Pi)는 리눅스 기반의 아주 작은 컴퓨터이다. 라즈베리파이를 이용하면 홈시어터나 아주 작은 게임기 등 여러 가지 재미있는 것들을 만들 수 있는데 파이썬은 이 라즈베리파이를 제어하는 도구로 사용된다. 예를 들어 라즈베리파이에 연결된 모터를 작동시키거나 램프에 불이 들어오게 하는 일들을 파이썬으로 할 수 있다.

☐ **파이썬으로 할 수 없는 것들**

1) 시스템과 밀접한 프로그래밍 영역

파이썬으로 도스나 리눅스 같은 운영체제, 엄청난 횟수의 반복과 연산을 필요로 하는 프로그램 또는 데이터 압축 알고리즘 개발 프로그램 등을 만드는 것은 어렵다. 즉, 대단히 빠른 속도를 요구하거나 하드웨어를 직접 건드려야 하는 프로그램에는 어울리지 않는다.

2) 모바일 프로그래밍

파이썬은 구글이 가장 많이 애용하는 언어이지만 파이썬으로 안드로이드 앱(App)을 개발하는 것은 아직 어렵다. 안드로이드에서 파이썬으로 만든 프로그램들이 실행되도록 지원하긴 하지만 이것만으로 앱을 만들기에는 아직 역부족이다. 아이폰 앱을 개발하는 것 역시 파이썬으로는 할 수 없다.

02-1 숫자형

숫자형(Number)이란 숫자 형태로 이루어진 자료형으로, 우리가 이미 잘 알고 있는 것들이다.

우리가 흔히 사용하는 것들을 생각해 보자. 123과 같은 정수, 12.34와 같은 실수, 드물게 사용하긴 하지만 8진수나 16진수 같은 것들도 있다.

아래 표는 숫자들이 파이썬에서 어떻게 사용되는지를 간략하게 보여 준다.

항목	사용 예
정수	123, -345, 0
실수	123.45, -1234.5, 3.4e10
8진수	0o34, 0o25
16진수	0x2A, 0xFF

이제 이런 숫자들을 파이썬에서는 어떻게 만들고 사용하는지 자세히 알아보자.

숫자형은 어떻게 만들고 사용할까?

정수형

정수형(Integer)이란 말 그대로 정수를 뜻하는 자료형을 말한다. 다음 예는 양의 정수와 음의 정수, 숫자 0을 변수 a에 대입하는 예이다.

```
>>> a = 123
>>> a =
-178
>>> a = 0
```

실수형

파이썬에서 실수형(Floating-point)은 소수점이 포함된 숫자를 말한다. 다음 예는 실수를 변수 a에 대입하는 예이다.

```
>>> a = 1.2
>>> a =
-3.45
```

위의 방식은 우리가 일반적으로 볼 수 있는 실수형의 소수점 표현 방식이다.

```
>>> a = 4.24E10
>>> a = 4.24e-
10
```

위의 방식은 "컴퓨터식 지수 표현 방식"으로 파이썬에서는 4.24e10 또는 4.24E10처럼 표현한다(e와 E 둘 중 어느 것을 사용해도 무방하다). 여기서 4.24E10은 4.24*10104.24*1010, 4.24e-10은 4.24*10−104.24*10−10을 의미한다.

8진수와 16진수

8진수(Octal)를 만들기 위해서는 숫자가 0o 또는 0O(숫자 0 + 알파벳 소문자 o 또는 대문자 O)로 시작하면 된다.

```
>>> a =
0o177
```

16진수(Hexadecimal)를 만들기 위해서는 0x로 시작하면 된다.

```
>>> a =
0x8ff
>>> b =
0xABC
```

8진수나 16진수는 파이썬에서 잘 사용하지 않는 형태의 숫자 자료형이니 간단히 눈으로 익히고 넘어가자.

숫자형을 활용하기 위한 연산자

사칙연산

프로그래밍을 한 번도 해본 적이 없는 독자라도 사칙연산(+, -, *, /)은 알고 있을 것이다. 파이썬 역시 계산기와 마찬가지로 아래의 연산자를 이용해 사칙연산을 수행한다.

```
>>> a =
3
>>> b =
4
>>> a +
b
7
>>> a *
b
12
>>> a /
b
0.75
```

[파이썬 2.7에서 3/4를 실행하면 어떻게 될까?]

파이썬 2.7의 경우 위 사칙연산 예제의 `a/b`를 실행하면 0.75가 아닌 0이 출력된다. 파이썬 2.7은 정수형끼리 나눌 경우 정수로만 결과값을 리턴하기 때문이다. 만약 위 예제와 동일한 결과값을 얻고 싶다면 `a/(b*1.0)` 처럼 b를 강제로 실수형으로 변환해야 한다. 이 책에서 사용하는 파이썬 3은 위의 사칙연산 예제에서 볼 수 있듯이 실수형으로 따로 변환해 줄 필요가 없다.

x의 y제곱을 나타내는 ** 연산자

다음으로 알아야 할 연산자로 **라는 연산자가 있다. 이 연산자는 `x ** y` 처럼 사용되었을 때 x의 y제곱(xy) 값을 리턴한다. 다음의 예를 통해 알아보자.

```
>>> a = 3
>>> b = 4
>>> a **
b
81
```

나눗셈 후 나머지를 반환하는 % 연산자

프로그래밍을 처음 접하는 독자라면 % 연산자는 본 적이 없을 것이다. %는 나눗셈의 나머지 값을 반환하는 연산자이다. 7을 3으로 나누면 나머지는 1이 될 것이고 3을 7로 나누면 나머지는 3이 될 것이다. 다음의 예로 확인해 보자.

```
>>> 7 %
3
1
>>> 3 %
7
3
```

나눗셈 후 소수점 아랫자리를 버리는 // 연산자

/ 연산자를 사용하여 7 나누기 4를 하면 그 결과는 예상대로 1.75가 된다.

```
>>> 7 /
4
1.75
```

이번에는 나눗셈 후 소수점 아랫자리를 버리는 // 연산자를 사용한 경우를 보자.

```
>>> 7 //
4
1
```

1.75의 소수점 부분인 0.75가 제거되어 1이 나오는 것을 확인할 수 있다.

// 연산자를 사용할 때는 한가지 주의해야 할 점이 있다. 그것은 다음처럼 음수에 // 연산자를 적용하는 경우이다.

```
>>> -7 / 4
>>> -1.75
>>> -7 //
4
>>> -2
```

-1.75 라는 실수에서 소수점을 버리면 -1이 되어야 할 것 같지만 -7 // 4의 결과는 -2가 되었다. 이렇게 되는 이유는 // 연산자는 나눗셈의 결과에서 무조건 소수점을 버리는것이 아니라 나눗셈의 결과값보다 작은 정수 중, 가장 큰 정수를 리턴하기 때문이다.

02-2 문자열 자료형

문자열(String)이란 문자, 단어 등으로 구성된 문자들의 집합을 의미한다. 예를 들어 다음과 같은 것들이 문자열이다.

```
"Life is too short, You need
Python"
"a"
"123"
```

위의 문자열 예문을 보면 모두 큰따옴표(" ")로 둘러싸여 있다. "123은 숫자인데 왜 문자열이지?"라는 의문이 드는 독자도 있을 것이다. 따옴표로 둘러싸여 있으면 모두 문자열이라고 보면 된다.

문자열은 어떻게 만들고 사용할까?

위의 예에서는 문자열을 만들 때 큰따옴표(" ")만을 사용했지만 이 외에도 문자열을 만드는 방법은 3가지가 더 있다. 파이썬에서 문자열을 만드는 방법은 총 4가지이다.

1. 큰따옴표로 양쪽 둘러싸기

```
"Hello
World"
```

2. 작은따옴표로 양쪽 둘러싸기

```
'Python is
fun'
```

3. 큰따옴표 3개를 연속으로 써서 양쪽 둘러싸기

```
"""Life is too short, You need
python"""
```

4. 작은따옴표 3개를 연속으로 써서 양쪽 둘러싸기

```
'''Life is too short, You need
python'''
```

단순함이 자랑인 파이썬이 문자열을 만드는 방법은 왜 4가지나 가지게 되었을까? 그 이유에 대해서 알아보자.

문자열 안에 작은따옴표나 큰따옴표를 포함시키고 싶을 때

문자열을 만들어 주는 주인공은 작은따옴표(')와 큰따옴표(")이다. 그런데 문자열 안에도 작은따옴표와 큰따옴표가 들어 있어야 할 경우가 있다. 이때는 좀 더 특별한 기술이 필요하다. 예제를 하나씩 살펴보면서 원리를 익혀 보도록 하자.

1) 문자열에 작은따옴표 (') 포함시키기

```
Python's favorite food is
perl
```

위와 같은 문자열을 food라는 변수에 저장하고 싶다고 가정하자. 문자열 중 Python's에 작은따옴표(')가 포함되어 있다.

이럴 때는 다음과 같이 문자열을 큰따옴표(")로 둘러싸야 한다. 큰따옴표 안에 들어 있는 작은따옴표는 문자열을 나타내기 위한 기호로 인식되지 않는다.

```
>>> food = "Python's favorite food is
perl"
```

프롬프트에 food를 입력해서 결과를 확인하자. 변수에 저장된 문자열이 그대로 출력되는 것 을 볼 수 있다.

```
>>> food
"Python's favorite food is
perl"
```

시험 삼아 다음과 같이 큰따옴표(")가 아닌 작은따옴표(')로 문자열을 둘러싼 후 다시 실행해 보자. 'Python'이 문자열로 인식되어 구문 오류(SyntaxError)가 발생할 것이다.

```
>>> food = 'Python's favorite food is
perl'
File "<stdin>", line 1
food = 'Python's favorite food is perl'
^
SyntaxError: invalid syntax
```

2) 문자열에 큰따옴표 (") 포함시키기

```
"Python is very easy." he
says.
```

위와 같이 큰따옴표(")가 포함된 문자열이라면 어떻게 해야 큰따옴표가 제대로 표현될까? 다음과 같이 문자열을 작은따옴표(')로 둘러싸면 된다.

```
>>> say = '"Python is very easy." he
says.'
```

이렇게 작은따옴표(') 안에 사용된 큰따옴표(")는 문자열을 만드는 기호로 인식되지 않는다.

3) \(백슬래시)를 이용해서 작은따옴표(')와 큰따옴표(")를 문자열에 포함시키기

```
>>> food = 'Python\'s favorite food is perl'
>>> say = "\"Python is very easy.\" he
says."
```

작은따옴표(')나 큰따옴표(")를 문자열에 포함시키는 또 다른 방법은 \(백슬래시)를 이용하는 것이다. 즉, 백슬래시(\)를 작은따옴표(')나 큰따옴표(") 앞에 삽입하면 \(백슬래시) 뒤의 작은따옴표(')나 큰따옴표(")는 문자열을 둘

러싸는 기호의 의미가 아니라 문자 ('), (") 그 자체를 뜻하게 된다.

어떤 방법을 사용해서 문자열 안에 작은따옴표(')와 큰따옴표(")를 포함시킬지는 각자의 선택이다. 대화형 인터프리터를 실행시킨 후 위의 예문들을 꼭 직접 작성해 보도록 하자.

여러 줄인 문자열을 변수에 대입하고 싶을 때

문자열이 항상 한 줄짜리만 있는 것은 아니다. 다음과 같이 여러 줄의 문자열을 변수에 대입하려면 어떻게 처리해야 할까?

```
Life is too
short
You need python
```

1) 줄을 바꾸기 위한 이스케이프 코드 \n 삽입하기

```
>>> multiline = "Life is too short\nYou need
python"
```

위의 예처럼 줄바꿈 문자인 \n을 삽입하는 방법이 있지만 읽기에 불편하고 줄이 길어지는 단점이 있다.

2) 연속된 작은따옴표 3개(''') 또는 큰따옴표 3개(""") 이용

위 1번의 단점을 극복하기 위해 파이썬에서는 다음과 같이 작은따옴표 3개(''') 또는 큰따옴표 3개(""")를 이용한다.

```
>>> multiline='''
... Life is too
short
... You need python
... '''
```

작은따옴표 3개를 사용한 경우

```
>>> multiline="""
... Life is too
short
... You need python
... """
```

큰따옴표 3개를 사용한 경우

print(multiline)을 입력해서 어떻게 출력되는지 확인해 보자.

```
>>>
print(multiline)
Life is too short
You need python
```

두 경우 모두 결과는 동일하다. 위 예에서도 확인할 수 있듯이 문자열이 여러 줄인 경우 이스케이프 코드를 쓰는 것보다 따옴표를 연속해서 쓰는 것이 훨씬 깔끔하다.

[이스케이프 코드란?]

문자열 예제에서 여러 줄의 문장을 처리할 때 백슬래시 문자와 소문자 n을 조합한 \n 이스케이프 코드를 사용했다. 이스케이프 코드란 프로그래밍할 때 사용할 수 있도록 미리 정의해 둔 "문자 조합"이다. 주로 출력물을 보기 좋게 정렬하는 용도로 이용된다. 몇 가지 이스케이프 코드를 정리하면 다음과 같다.

코드	설명
\n	개행 (줄바꿈)
\t	수평 탭
\\	문자 "\"
\'	단일 인용부호(')
\"	이중 인용부호(")
\r	캐리지 리턴
\f	폼 피드
\a	벨 소리
\b	백 스페이스
\000	널문자

이중에서 활용빈도가 높은 것은 \n, \t, \\, \', \"이다. 나머지는 프로그램에서 잘 사용되지 않는다.

문자열 연산하기

파이썬에서는 문자열을 더하거나 곱할 수 있다. 이는 다른 언어에서는 쉽게 찾아볼 수 없는 재미있는 기능으로, 우리의 생각을 그대로 반영해 주는 파이썬만의 장점이라고 할 수 있다. 문자열을 더하거나 곱하는 방법에 대해 알아보자.

1) 문자열 더해서 연결하기(Concatenation)

```
>>> head = "Python"
>>> tail = " is
fun!"
>>> head + tail
'Python is fun!'
```

위 소스 코드에서 세 번째 줄을 보자. 복잡하게 생각하지 말고 눈에 보이는 대로 생각해 보자. "Python"이라는 head 변수와 " is fun!"이라는 tail 변수를 더한 것이다. 결과는 'Python is fun!'이다. 즉, head와 tail 변수가 +에 의해 합쳐진 것이다.

직접 실행해 보고 결과값이 제시한 것과 똑같이 나오는지 확인해 보자.

2) 문자열 곱하기

```
>>> a =
"python"
>>> a * 2
'pythonpython'
```

위 소스 코드에서 *의 의미는 우리가 일반적으로 사용하는 숫자 곱하기의 의미와는 다르다. 위 소스 코드에서

a *
2 라는 문장은 a를 두 번 반복하라는 뜻이다. 즉, *는 문자열의 반복을 뜻하는 의미로 사용되었다. 굳이 코드의 의미를 설명할 필요가 없을 정도로 직관적이다.

3) 문자열 곱하기 응용

문자열 곱하기를 좀 더 응용해 보자. 다음과 같은 소스를 에디터로 작성해 실행해 보자.

```
print("=" * 50)
print("My
Program")
print("=" * 50)
```

결과값은 다음과 같이 나타날 것이다.

```
C:\Users>cd C:\Python
C:\Python>python multistring.py
==================================================
My Program
==================================================
```

이런 식의 표현은 앞으로 자주 사용하게 될 것이다. 프로그램을 만들어 실행시켰을 때 출력되는 화면 제일 상단에 프로그램 제목을 이와 같이 표시하면 보기 좋지 않겠는가?

문자열 인덱싱과 슬라이싱

인덱싱(Indexing)이란 무엇인가를 "가리킨다"는 의미이고, 슬라이싱(Slicing)은 무엇인가를 "잘라낸다"는 의미이다. 이런 의미를 생각하면서 다음 내용을 살펴보자.

문자열 인덱싱이란?

```
>>> a = "Life is too short, You need
Python"
```

위 소스 코드에서 변수 a에 저장한 문자열의 각 문자마다 번호를 매겨 보면 다음과 같다.

```
Life is too short, You need
Python
0         1         2         3
01234567890123456789012345678901234567890123
```

"Life is too short, You need Python"이라는 문자열에서 L은 첫 번째 자리를 뜻하는 숫자인 0, 바로 다음인 i는 1 이런 식으로 계속 번호를 붙인 것이다. 중간에 있는 short의 s는 12가 된다.

이제 다음 예를 실행해 보자.

```
>>> a = "Life is too short, You need
Python"
>>> a[3]
'e'
```

a[3]이 뜻하는 것은 a라는 문자열의 네 번째 문자인 e를 말한다. 프로그래밍을 처음 접하는 독자라면 a[3]에서 3이란 숫자가 왜 네 번째 문자를 뜻하는 것인지 의아할 수도 있다. 사실 이 부분이 필자도 가끔 헷갈리는 부분인데, 이렇게 생각하면 쉽게 알 수 있을 것이다.

"파이썬은 0부터 숫자를 센다."

고로 위의 문자열을 파이썬은 다음과 같이 바라보고 있다.

```
a[0]:'L', a[1]:'i', a[2]:'f', a[3]:'e', a[4]:' ',
...
```

0부터 숫자를 센다는 것이 처음에는 익숙하지 않겠지만 계속 사용하다 보면 자연스러워질 것이다. 위의 예에서 볼 수 있듯이 a[번호]는 문자열 내 특정한 값을 뽑아내는 역할을 한다. 이러한 것을 인덱싱이라고 한다.

문자열 인덱싱 활용하기

인덱싱 예를 몇 가지 더 보도록 하자.

```
>>> a = "Life is too short, You need
Python"
>>> a[0]
'L'
>>> a[12]
's'
>>> a[-1]
'n'
```

앞의 a[0]과 a[12]는 쉽게 이해할 수 있는데 마지막의 a[-1]이 뜻하는 것은 뭘까? 눈치 빠른 독자는 이미 알아챘겠지만 문자열을 뒤에서부터 읽기 위해서 마이너스(-) 기호를 붙이는 것이다. 즉 a[-1]은 뒤에서부터 세어 첫 번째가 되는 문자를 말한다. a는 "Life is too short, You needPython"이라는 문장이므로 뒤에서부터 첫 번째 문자는 가장 마지막 문자인 'n'이다.

뒤에서부터 첫 번째 문자를 표시할 때도 0부터 세어 "a[-0]이라고 해야 하지 않을까?"라는 의문이 들 수도 있겠지만 잘 생각해 보자. 0과 -0은 똑같은 것이기 때문에 a[-0]은 a[0]과 똑같은 값을 보여 준다.

```
>>>
a[-0]
'L'
```

계속해서 몇 가지 예를 더 보자.

```
>>>
a[-2]
'o'
>>>
a[-5]
'y'
```

위의 첫 번째 예는 뒤에서부터 두 번째 문자를 가리키는 것이고, 두 번째 예는 뒤에서부터 다섯 번째 문자를 가리키는 것이다.

문자열 슬라이싱이란?

그렇다면 "Life is too short, You need Python"이라는 문자열에서 단순히 한 문자만을 뽑아내는 것이 아니라 'Life' 또는 'You' 같은 단어들을 뽑아내는 방법은 없을까?

다음과 같이 하면 된다.

```
>>> a = "Life is too short, You need
Python"
>>> b = a[0] + a[1] + a[2] + a[3]
>>> b
'Life'
```

위의 방법처럼 단순하게 접근할 수도 있지만 파이썬에서는 더 좋은 방법을 제공한다. 바로 슬라이싱(Slicing) 이라는 기법이다.

위의 예는 슬라이싱 기법으로 다음과 같이 간단하게 처리할 수 있다.

```
>>> a = "Life is too short, You need
Python"
>>> a[0:4]
'Life'
```

a[0:4]가 뜻하는 것은 a라는 문자열, 즉 "Life is too short, You need Python"이라는 문장에서 0부터 4까지의 문자를 뽑아낸다는 뜻이다. 하지만 다음과 같은 의문이 생길 것이다. a[0]은 L, a[1]은 i, a[2]는 f, a[3]은 e니까 a[0:3]으로도 Life라는 단어를 뽑아낼 수 있지 않을까? 다음의 예를 보도록 하자.

```
>>>
a[0:3]
'Lif'
```

이렇게 되는 이유는 간단하다. a[시작 번호:끝 번호]를 지정하면 끝 번호에 해당하는 것은 포함되지 않는다. a[0:3]을 수식으로 나타내면 다음과 같다.

```
0 <= a <
3
```

이 수식을 만족하는 a는 a[0], a[1], a[2]일 것이다. 따라서 a[0:3]은 'Lif'이고 a[0:4]는 'Life'가 되는 것이다. 이 부분이 문자열 연산에서 가장 혼동하기 쉬운 부분이니 장 마지막의 연습문제를 많이 풀어 보면서 몸에 익히기 바란다.

문자열을 슬라이싱하는 방법

슬라이싱의 예를 조금 더 보도록 하자.

```
>>>
a[0:5]
'Life '
```

위의 예는 a[0] + a[1] + a[2] + a[3] + a[4]와 동일하다. a[4]는 공백 문자이기 때문에 'Life'가 아닌 'Life '가 출력되는 것이다. 공백 문자 역시 L, i, f, e 같은 문자와 동일하게 취급되는 것을 잊지 말자. 'Life'와 'Life '는 완전히 다른 문자열이다.

슬라이싱할 때 항상 시작 번호가 '0'일 필요는 없다.

```
>>> a[0:2]
'Li'
>>> a[5:7]
'is'
>>>
a[12:17]
'short'
```

a[시작 번호:끝 번호]에서 끝 번호 부분을 생략하면 시작 번호부터 그 문자열의 끝까지 뽑아낸다.

```
>>> a[19:]
'You need Python'
```

a[시작 번호:끝 번호]에서 시작 번호를 생략하면 문자열의 처음부터 끝 번호까지 뽑아낸다.

```
>>> a[:17]
'Life is too
short'
```

a[시작 번호:끝 번호]에서 시작 번호와 끝 번호를 생략하면 문자열의 처음부터 끝까지를 뽑아낸다.

```
>>> a[:]
'Life is too short, You need
Python'
```

슬라이싱에서도 인덱싱과 마찬가지로 마이너스(-) 기호를 사용할 수 있다.

```
>>>
a[19:-7]
'You need'
```

위 소스 코드에서 a[19:-7]이 뜻하는 것은 a[19]에서부터 a[-8]까지를 말한다. 이 역시 a[-7]은 포함하지 않는다.

슬라이싱으로 문자열 나누기

다음은 자주 사용하게 되는 슬라이싱 기법 중 하나이다.

```
>>> a =
"20010331Rainy"
>>> date = a[:8]
>>> weather = a[8:]
>>> date
'20010331'
>>> weather
'Rainy'
```

a라는 문자열을 두 부분으로 나누는 기법이다. 동일한 숫자 8을 기준으로 a[:8], a[8:]처럼 사용했다. a[:8]은 a[8]이 포함되지 않고, a[8:]은 a[8]을 포함하기 때문에 8을 기준으로 해서 두 부분으로 나눌 수 있는 것이다. 위의 예에서는 "20010331Rainy"라는 문자열을 날짜를 나타내는 부분인 '20010331'과 날씨를 나타내는 부분인 'Rainy'로 나누는 방법을 보여준다.

위의 문자열 "20010331Rainy"를 연도인 2001, 월과 일을 나타내는 0331, 날씨를 나타내는 Rainy의 세 부분으로 나누려면 다음과 같이 할 수 있다.

```
>>> a =
"20010331Rainy"
>>> year = a[:4]
>>> day = a[4:8]
>>> weather = a[8:]
>>> year
'2001'
>>> day
'0331'
>>> weather
'Rainy'
```

위의 예는 4와 8이란 숫자로 "20010331Rainy"라는 문자열을 세 부분으로 나누는 방법을 보여 준다.

지금까지 인덱싱과 슬라이싱에 대해서 살펴보았다. 인덱싱과 슬라이싱은 프로그래밍을 할때 매우 자주 사용되는 기법이니 꼭 반복해서 연습해 두자.

["Pithon"이라는 문자열을 "Python"으로 바꾸려면?]

"Pithon"이라는 문자열을 "Python"으로 바꾸려면 어떻게 해야 할까? 제일 먼저 떠오르는 생각은 다음과 같을 것이다.

```
>>> a =
"Pithon"
>>> a[1]
'i'
>>> a[1] = 'y'
```

요컨대 a라는 변수에 "Pithon"이라는 문자열을 대입하고 a[1]의 값이 i니까 a[1]을 y로 바꾸어 준다는 생 각이다. 하지만 결과는 어떻게 나올까?

당연히 에러가 발생한다. 왜냐하면 문자열의 요소값은 바꿀 수 있는 값이 아니기 때문이다(문자열, 튜플 등 의 자료형은 그 요소값을 변경할 수 없다. 그래서 immutable한 자료형이라고도 부른다). 하지만 앞서 살펴본 슬라이싱 기법을 이용하면 "Pithon"이라는 문자열을 "Python"으로 바꿀 수 있는 방법이 있다.

다음의 예를 보자.

```
>>> a = "Pithon"
>>> a[:1]
'P'
>>> a[2:]
'thon'
>>> a[:1] + 'y' +
a[2:]
'Python'
```

위의 예에서 볼 수 있듯이 슬라이싱을 이용하면 'Pithon'이라는 문자열을 'P' 부분과 'thon' 부분으로 나눌 수 있기 때문에 그 사이에 'y'라는 문자를 추가하여 'Python'이라는 새로운 문자열을 만들 수 있다.

문자열 포매팅

문자열에서 또 하나 알아야 할 것으로는 문자열 포매팅(Formatting)이 있다. 이것을 공부하기에 앞서 다음과 같은

문자열을 출력하는 프로그램을 작성했다고 가정해 보자.

"현재 온도는 18도입니다."

시간이 지나서 20도가 되면 아래와 같은 문장을 출력한다.

"현재 온도는 20도입니다"

위의 두 문자열은 모두 같은데 20이라는 숫자와 18이라는 숫자만 다르다. 이렇게 문자열 내의 특정한 값을 바꿔야 할 경우가 있을 때 이것을 가능하게 해주는 것이 바로 문자열 포매팅 기법이다.

문자열 포매팅이란 쉽게 말해 문자열 내에 어떤 값을 삽입하는 방법이다. 다음의 예들을 직접 실행해 보면서 그 사용법을 알아보자.

문자열 포매팅 따라 하기

1) 숫자 바로 대입

```
>>> "I eat %d apples." %
3
'I eat 3 apples.'
```

위 예제의 결과값을 보면 알겠지만 위의 예제는 문자열 내에 3이라는 정수를 삽입하는 방법을 보여 준다. 문자열 안에서 숫자를 넣고 싶은 자리에 %d라는 문자를 넣어 주고, 삽입할 숫자인 3은 가장 뒤에 있는 % 문자 다음에 써 넣었다. 여기서 %d는 문자열 포맷 코드라고 부른다.

2) 문자열 바로 대입

문자열 내에 꼭 숫자만 넣으라는 법은 없다. 이번에는 숫자 대신 문자열을 넣어 보자.

```
>>> "I eat %s apples." %
"five"
'I eat five apples.'
```

위 예제에서는 문자열 내에 또 다른 문자열을 삽입하기 위해 앞서 사용했던 문자열 포맷 코드 %d 가 아닌 %s를 썼다. 어쩌면 눈치 빠른 독자는 벌써 유추하였을 것이다. 숫자를 넣기 위해서는 %d를 써야 하고, 문자열을 넣기 위해서는 %s를 써야 한다는 사실을 말이다.

(※ 문자열을 대입할 때는 앞에서 배웠던 것처럼 큰따옴표나 작은따옴표를 반드시 써주어야 한다.)

3) 숫자 값을 나타내는 변수로 대입

```
>>> number = 3
>>> "I eat %d apples." %
number
'I eat 3 apples.'
```

1번처럼 숫자를 바로 대입하나 위 예제처럼 숫자 값을 나타내는 변수를 대입하나 결과는 같다.

4) 2개 이상의 값 넣기

그렇다면 문자열 안에 1개가 아닌 여러 개의 값을 넣고 싶을 땐 어떻게 해야 할까?

```
>>> number = 10
>>> day = "three"
>>> "I ate %d apples. so I was sick for %s days." % (number,
day)
'I ate 10 apples. so I was sick for three days.'
```

위의 예문처럼 2개 이상의 값을 넣으려면 마지막 % 다음 괄호 안에 콤마(,)로 구분하여 각각의 변수를 넣어 주면 된다.

문자열 포맷 코드

문자열 포매팅 예제에서는 대입시켜 넣는 자료형으로 정수와 문자열을 사용했으나 이 외에도 다양한 것들을 대입시킬 수 있다. 문자열 포맷 코드로는 다음과 같은 것들이 있다.

코드	설명
%s	문자열 (String)
%c	문자 1개(character)
%d	정수 (Integer)
%f	부동소수 (floating-point)
%o	8진수
%x	16진수
%%	Literal % (문자 % 자체)

여기서 재미있는 것은 %s 포맷 코드인데, 이 코드는 어떤 형태의 값이든 변환해 넣을 수 있다. 무슨 말인지 예를 통해 확인해 보자.

```
>>> "I have %s apples" %
3
'I have 3 apples'
>>> "rate is %s" % 3.234
'rate is 3.234'
```

3을 문자열 안에 삽입하려면 %d를 사용하고, 3.234를 삽입하려면 %f를 사용해야 한다. 하지만 %s를 사용하면 이런 것을 생각하지 않아도 된다. 왜냐하면 %s는 자동으로 % 뒤에 있는 값을 문자열로 바꾸기 때문이다.

[포매팅 연산자 %d와 %를 같이 쓸 때는 %%를 쓴다]

```
>>> "Error is %d%." %
98
```

위 예문의 결과값으로 당연히 "Error is 98%."가 출력될 것이라고 예상하겠지만 파이썬은 값이 올바르지 않다는 값 오류(Value Error) 메시지를 보여 준다.

```
Traceback (most recent call last):
File "<stdin>", line 1, in
<module>
ValueError: incomplete format
```

이유는 문자열 포맷 코드인 %d와 %가 같은 문자열 내에 존재하는 경우, %를 나타내려면 반드시 %%로 써야 하는 법칙이 있기 때문이다. 이 점은 꼭 기억해 두어야 한다. 하지만 문자열 내에 %d 같은 포매팅 연산자가 없으면 %는 홀로 쓰여도 상관이 없다.

따라서 위 예를 제대로 실행하려면 다음과 같이 해야 한다.

```
>>> "Error is %d%%." %
98
'Error is 98%.'
```

포맷 코드와 숫자 함께 사용하기

위에서 보았듯이 %d, %s 등의 포맷 코드는 문자열 내에 어떤 값을 삽입하기 위해서 사용된다. 하지만 포맷 코드를 숫자와 함께 사용하면 더 유용하게 사용할 수 있다. 다음의 예를 보고 따라 해보자.

1) 정렬과 공백

```
>>> "%10s" %
"hi"
'        hi'
```

앞의 예문에서 "%10s"의 의미는 전체 길이가 10개인 문자열 공간에서 hi를 오른쪽으로 정렬하고 그 앞의 나머지는 공백으로 남겨 두라는 의미이다.

그렇다면 반대쪽인 왼쪽 정렬은 "%-10s"가 될 것이다. 확인해 보자.

```
>>> "%-10sjane." %
'hi'
'hi        jane.'
```

hi를 왼쪽으로 정렬하고 나머지는 공백으로 채웠음을 볼 수 있다.

2) 소수점 표현하기

```
>>> "%0.4f" %
3.42134234
'3.4213'
```

3.42134234를 소수점 네 번째 자리까지만 나타내고 싶은 경우에는 위와 같이 사용한다. 즉, 여기서 '.'의 의미는 소수점 포인트를 말하고 그 뒤의 숫자 4는 소수점 뒤에 나올 숫자의 개수를 말한다. 다음의 예를 살펴보자.

```
>>> "%10.4f" %
3.42134234
'    3.4213'
```

위의 예는 3.42134234라는 숫자를 소수점 네 번째 자리까지만 표시하고 전체 길이가 10개인 문자열 공간에서 오른쪽으로 정렬하는 예를 보여준다.

지금까지는 문자열을 가지고 할 수 있는 기본적인 것들에 대해 알아보았다. 이제부터는 문자열을 좀 더 자유자재로 다루기 위해 공부해야 할 것들을 설명할 것이다. 지겹다면 잠시 책을접고 휴식을 취하도록 하자.

문자열 관련 함수들

문자열 자료형은 자체적으로 가지고 있는 함수들이 있다. 이 함수들은 다른말로 문자열 내장함수라고 말한다. 이 내장함수를 사용하려면 문자열 변수 이름 뒤에 '.'를 붙인 다음에 함수 이름을 써주면 된다. 이제 문자열이 자체적으로 가지고 있는 내장 함수들에 대해서 알아보자.

문자 개수 세기(count)

```
>>> a = "hobby"
>>>
a.count('b')
2
```

문자열 중 문자 b의 개수를 반환한다.

위치 알려주기1(find)

```
>>> a = "Python is best
choice"
>>> a.find('b')
10
>>> a.find('k')
-1
```

문자열 중 문자 b가 처음으로 나온 위치를 반환한다. 만약 찾는 문자나 문자열이 존재하지 않는다면 -1을 반환한다.

(※ 파이썬은 숫자를 0부터 세기 때문에 b의 위치는 11이 아닌 10이 된다.)

위치 알려주기2(index)

```
>>> a = "Life is too short"
>>> a.index('t')
8
>>> a.index('k')
Traceback (most recent call last):
File "<stdin>", line 1, in
<module>
ValueError: substring not found
```

문자열 중 문자 t가 맨 처음으로 나온 위치를 반환한다. 만약 찾는 문자나 문자열이 존재하지 않는다면 오류를 발생시킨다. 앞의 find 함수와 다른 점은 문자열 안에 존재하지 않는 문자를 찾으면 오류가 발생한다는 점이다.

문자열 삽입(join)

```
>>> a= ","
>>>
a.join('abcd')
'a,b,c,d'
```

abcd라는 문자열의 각각의 문자 사이에 변수 a의 값인 ','를 삽입한다.

소문자를 대문자로 바꾸기(upper)

```
>>> a = "hi"
>>>
a.upper()
'HI'
```

upper() 함수는 소문자를 대문자로 바꾸어 준다. 만약 문자열이 이미 대문자라면 아무런 변화도 일어나지 않을 것이다.

대문자를 소문자로 바꾸기(lower)

```
>>> a = "HI"
>>>
a.lower()
'hi'
```

lower() 함수는 대문자를 소문자로 바꾸어 준다.

왼쪽 공백 지우기(lstrip)

```
>>> a = " hi
"
>>>
a.lstrip()
'hi '
```

문자열 중 가장 왼쪽에 있는 한 칸 이상의 연속된 공백들을 모두 지운다. lstrip에서 l은 left를 의미한다.

오른쪽 공백 지우기(rstrip)

```
>>> a= " hi "
>>>
a.rstrip()
' hi'
```

문자열 중 가장 오른쪽에 있는 한 칸 이상의 연속된 공백들을 모두 지운다. rstrip에서 r은 right를 의미한다.

양쪽 공백 지우기(strip)

```
>>> a = " hi
"
>>> a.strip()
'hi'
```

문자열 양쪽에 있는 한 칸 이상의 연속된 공백들을 모두 지운다.

문자열 바꾸기(replace)

```
>>> a = "Life is too short"
>>> a.replace("Life", "Your
leg")
'Your leg is too short'
```

replace(바뀌게 될 문자열, 바꿀 문자열)처럼 사용해서 문자열 내의 특정한 값을 다른 값으로 치환해 준다.

문자열 나누기(split)

```
>>> a = "Life is too short"
>>> a.split()
['Life', 'is', 'too',
'short']
>>> a = "a:b:c:d"
>>> a.split(':')
['a', 'b', 'c', 'd']
```

`a.split()`처럼 괄호 안에 아무런 값도 넣어 주지 않으면 공백(스페이스, 탭, 엔터등)을 기준으로 문자열을 나누어 준다. 만약 `a.split(':')`처럼 괄호 안에 특정한 값이 있을 경우에는 괄호 안의 값을 구분자로 해서 문자열을 나누어 준다. 이렇게 나눈 값은 리스트에 하나씩 들어가게 된다. `['Life', 'is', 'too', 'short']`나 `['a', 'b', 'c', 'd']`는 리스트라는 것인데 다음 절에서 자세히 알아볼 것이니 여기서는 너무 신경 쓰지 말도록 하자.

위에서 소개한 문자열 관련 함수들은 문자열 처리에서 사용 빈도가 매우 높은 것들이고 유용한 것들이다. 이 외에도 몇 가지가 더 있지만 자주 사용되는 것들은 아니다.

[고급 문자열 포매팅]

문자열의 format 함수를 이용하면 좀 더 발전된 스타일로 문자열 포맷을 지정할 수 있다. 앞에서 살펴본 문자열 포매팅 예제들을 format 함수를 이용해서 바꾸면 다음과 같다.

숫자 바로 대입하기

```
>>> "I eat {0}
apples".format(3)
'I eat 3 apples'
```

"I eat {0} apples" 문자열 중 {0} 부분이 숫자 3으로 바뀌었다.

문자열 바로 대입하기

```
>>> "I eat {0}
apples".format("five")
'I eat five apples'
```

문자열의 {0} 항목이 five라는 문자열로 바뀌었다.

숫자 값을 가진 변수로 대입하기

```
>>> number = 3
>>> "I eat {0}
apples".format(number)
'I eat 3 apples'
```

문자열의 {0} 항목이 number 변수의 값인 3으로 바뀌었다.

2개 이상의 값 넣기

```
>>> number = 10
>>> day = "three"
>>> "I ate {0} apples. so I was sick for {1} days.".format(number,
day)
'I ate 10 apples. so I was sick for three days.'
```

2개 이상의 값을 넣을 경우 문자열의 {0}, {1}과 같은 인덱스 항목들이 format 함수의 입력값들로 순서에 맞게 바뀐다. 즉, 위 예에서 {0}은 format 함수의 첫 번째 입력값인 number로 바뀌고 {1}은 format 함수의 두 번째 입력값인 day로 바뀐다.

이름으로 넣기

```
>>> "I ate {number} apples. so I was sick for {day} days.".format(number=10,
day=3)
'I ate 10 apples. so I was sick for 3 days.'
```

위 예에서 볼 수 있듯이 {0}, {1}과 같은 인덱스 항목 대신 더 편리한 {name} 형태를 이용하는 방법도 있다. {name} 형태를 이용할 경우 format 함수의 입력값에는 반드시 name=value와 같은 형태의 입력값이 있어야만 한다. 위 예는 문자열의 {number}, {day}가 format 함수의 입력값인 number=10, day=3 값으로 각각 바뀌는 것을 보여 주고 있다.

인덱스와 이름을 혼용해서 넣기

```
>>> "I ate {0} apples. so I was sick for {day} days.".format(10,
day=3)
'I ate 10 apples. so I was sick for 3 days.'
```

위와 같이 인덱스 항목과 name=value 형태를 혼용하는 것도 가능하다.

왼쪽 정렬

```
>>> "{0:
<10}".format("hi")
'hi        '
```

:<10 표현식을 이용하면 치환되는 문자열을 왼쪽으로 정렬하고 문자열의 총 자릿수를 10으로 맞출 수 있다.

오른쪽 정렬

```
>>> "
{0:>10}".format("hi")
'        hi'
```

오른쪽 정렬은 :< 대신 :>을 이용하면 된다. 화살표 방향을 생각하면 어느 쪽으로 정렬이 되는지 금방 알 수 있을 것이다.

가운데 정렬

```
>>> "
{0:^10}".format("hi")
'    hi    '
```

:^ 기호를 이용하면 가운데 정렬도 가능하다.

공백 채우기

```
>>> "
{0:=^10}".format("hi")
'====hi===='
>>> "{0:!
<10}".format("hi")
'hi!!!!!!!!'
```

정렬 시 공백 문자 대신에 지정한 문자 값으로 채워 넣는 것도 가능하다. 채워 넣을 문자 값은 정렬 문자인 <, >, ^ 바로 앞에 넣어야 한다. 위 예에서 첫 번째 예제는 가운데(^)로 정렬하고 빈 공간을 =문자로 채웠고, 두 번째 예제는 왼쪽(<)으로 정렬하고 빈 공간을 !문자로 채웠다.

소수점 표현하기

```
>>> y = 3.42134234
>>> "
{0:0.4f}".format(y)
'3.4213'
```

위 예는 format 함수를 이용해 소수점을 4자리까지만 표현하는 방법을 보여 준다. 이전에 살펴보았던 표현식 0.4f가 그대로 이용된 걸 알 수 있다.

```
>>> "
{0:10.4f}".format(y)
'    3.4213'
```

위와 같이 자릿수를 10으로 맞출 수도 있다. 역시 58쪽에서 살펴보았던 "10.4f"의 표현식이 그대로 이용된걸 알 수 있다.

{ 또는 } 문자 표현하기

```
>>> "{{ and
}}".format()
'{ and }'
```

format 함수를 이용해 문자열 포매팅을 할 경우 {나 }와 같은 중괄호(brace) 문자를 포매팅 문자가 아닌 문자 그대로 사용하고 싶은 경우에는 위 예의 {{와 }}처럼 2개를 연속해서 사용하면 된다.

02-3 리스트 자료형

지금까지 우리는 숫자와 문자열에 대해서 알아보았다. 하지만 숫자와 문자열만으로 프로그래밍을 하기엔 부족한 점이 많다. 예를 들어 1부터 10까지의 숫자 중 홀수 모음인 1, 3, 5, 7, 9라는 집합을 생각해 보자. 이런 숫자 모음을 숫자나 문자열로 표현하기는 쉽지 않다. 파이썬에는 이러한 불편함을 해소할 수 있는 자료형이 존재한다. 그것이 바로 이 절에서 공부하게될 리스트(List)이다.

리스트는 어떻게 만들고 사용할까?

리스트를 이용하면 1, 3, 5, 7, 9라는 숫자 모음을 다음과 같이 간단하게 표현할 수 있다.

```
>>> odd = [1, 3, 5, 7,
9]
```

리스트를 만들 때는 위에서 보는 것과 같이 대괄호([ ])로 감싸 주고 각 요소값들은 쉼표(,)로 구분해 준다.

> *리스트명 = [요소1, 요소2, 요소3, ...]*

여러 가지 리스트의 생김새를 살펴보면 다음과 같다.

```
>>> a = [ ]
>>> b = [1, 2, 3]
>>> c = ['Life', 'is', 'too',
'short']
>>> d = [1, 2, 'Life', 'is']
>>> e = [1, 2, ['Life', 'is']]
```

리스트는 a처럼 아무것도 포함하지 않는, 비어 있는 리스트([ ])일 수도 있고 b처럼 숫자를 요소값으로 가질 수도 있고 c처럼 문자열을 요소값으로 가질 수도 있다. 또한 d처럼 숫자와 문자열을 함께 요소값으로 가질 수도 있으며 e처럼 리스트 자체를 요소값으로 가질 수도 있다. 즉, 리스트 안에는 어떠한 자료형도 포함시킬 수 있다.

(※ 비어 있는 리스트는 a = list()로 생성할 수도 있다.)

리스트의 인덱싱과 슬라이싱

리스트도 문자열처럼 인덱싱과 슬라이싱이 가능하다. 백문이 불여일견. 말로 설명하는 것보다 직접 예를 실행해 보면서 리스트의 기본 구조를 이해하는 것이 쉽다. 대화형 인터프리터로 따라 하며 확실하게 이해하자.

리스트의 인덱싱

리스트 역시 문자열처럼 인덱싱을 적용할 수 있다. 먼저 a 변수에 [1, 2, 3]이라는 값을 설정 한다.

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

a[0]은 리스트 a의 첫 번째 요소값을 말한다.

```
>>>
a[0]
1
```

아래의 예는 리스트의 첫 번째 요소인 a[0]과 세 번째 요소인 a[2]의 값을 더한 것이다.

```
>>> a[0] +
a[2]
4
```

이것은 1 + 3으로 해석되어 4라는 값을 출력한다.

문자열을 공부할 때 이미 살펴보았지만 파이썬은 숫자를 0부터 세기 때문에 a[1]이 리스트 a의 첫 번째 요소가 아니라 a[0]이 리스트 a의 첫 번째 요소임을 명심하자. a[-1]은 문자열에서와 마찬가지로 리스트 a의 마지막 요소값을 말한다.

```
>>>
a[-1]
3
```

이번에는 아래의 예처럼 리스트 a를 숫자 1, 2, 3과 또 다른 리스트인 ['a', 'b', 'c']를 포함하도록 만들어 보자.

```
>>> a = [1, 2, 3, ['a', 'b',
'c']]
```

다음의 예를 따라 해보자.

```
>>> a[0]
1
>>> a[-1]
['a', 'b',
'c']
>>> a[3]
['a', 'b',
'c']
```

예상한 대로 a[-1]은 마지막 요소값인 ['a', 'b', 'c']를 나타낸다. a[3]은 리스트 a의 네 번째 요소를 나타내기 때문에 마지막 요소를 나타내는 a[-1]과 동일한 결과값을 보여 준다.

그렇다면 여기서 리스트 a에 포함된 ['a', 'b', 'c']라는 리스트에서 'a'라는 값을 인덱싱을 이용해 끄집어낼 수 있는 방법은 없을까? 다음의 예를 보자.

```
>>> a[-1]
[0]
'a'
```

위와 같이 하면 'a'를 끄집어낼 수 있다. a[-1]이 ['a', 'b', 'c'] 리스트라는 것은 이미 말했다. 바로 이 리스트에서 첫 번째 요소를 불러오기 위해 [0]을 붙여준 것이다.

아래의 예도 마찬가지 경우이므로 어렵지 않게 이해가 될 것이다.

```
>>> a[-1]
[1]
'b'
>>> a[-1]
[2]
'c'
```

[삼중 리스트에서 인덱싱하기]

조금 복잡하지만 다음의 예를 따라 해보자.

```
>>> a = [1, 2, ['a', 'b', ['Life',
'is']]]
```

리스트 a안에 ['a', 'b', ['Life', 'is']]라는 리스트가 포함되어 있고, 그 리스트 안에 다시 ['Life', 'is']라는 리스트가 포함되어 있다. 삼중 구조의 리스트이다.

이 경우 'Life'라는 문자열만 끄집어내려면 다음과 같이 해야 한다.

```
>>> a[2][2]
[0]
'Life'
```

위의 예는 리스트 a의 세 번째 요소인 리스트 ['a', 'b', ['Life', 'is']]에서 세 번째 요소인 리스트 ['Life', 'is']의 첫 번째 요소를 나타낸다.

이렇듯 리스트를 삼중으로 중첩해서 쓰면 혼란스럽기 때문에 자주 사용하지는 않지만 알아두는 것이 좋다.

리스트의 슬라이싱

문자열과 마찬가지로 리스트에서도 슬라이싱 기법을 적용할 수 있다. 슬라이싱은 "나눈다"는 뜻이라고 했다.

자, 그럼 리스트의 슬라이싱에 대해서 살펴보자.

```
>>> a = [1, 2, 3, 4,
5]
>>> a[0:2]
[1, 2]
```

앞의 예를 문자열에서 슬라이싱했던 것과 비교해 보자.

```
>>> a =
"12345"
>>> a[0:2]
'12'
```

2가지가 완전히 동일하게 사용됨을 눈치챘을 것이다. 문자열에서 했던 것과 사용법이 완전히 동일하다.

몇 가지 예를 더 들어 보도록 하자.

```
>>> a = [1, 2, 3, 4,
5]
>>> b = a[:2]
>>> c = a[2:]
>>> b
[1, 2]
>>> c
[3, 4, 5]
```

b라는 변수는 리스트 a의 첫번째 요소부터 두 번째 요소인 a[1]까지 나타내는 리스트이다. 물론 a[2] 값인 3은 포함되지 않는다. c라는 변수는 리스트 a의 세 번째 요소부터 끝까지 나타내는 리스트이다.

[중첩된 리스트에서 슬라이싱하기]

리스트가 포함된 중첩 리스트 역시 슬라이싱 방법은 똑같이 적용된다.

```
>>> a = [1, 2, 3, ['a', 'b', 'c'], 4,
5]
>>> a[2:5]
[3, ['a', 'b', 'c'], 4]
>>> a[3][:2]
['a', 'b']
```

위의 예에서 a[3]은 ['a', 'b', 'c']를 나타낸다. 따라서 a[3][:2]는 ['a', 'b', 'c']의 첫 번째 요소부터 세 번째 요소 직전까지의 값, 즉 ['a', 'b']를 나타내는 리스트가 된다.

리스트 연산자

리스트 역시 + 기호를 이용해서 더할 수 있고, * 기호를 이용해서 반복할 수 있다. 문자열과 마찬가지로 리스트에서도 되는지 직접 확인해 보자.

1) 리스트 더하기(+)

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

리스트 사이에서 + 기호는 2개의 리스트를 합치는 기능을 한다. 문자열에서 `"abc" + "def" = "abcdef"` 가 되는 것과 같은 이치이다.

2) 리스트 반복하기(*)

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

위에서 볼 수 있듯이 [1, 2, 3]이라는 리스트가 세 번 반복되어 새로운 리스트를 만들어낸다. 문자열에서 `"abc" * 3 = "abcabcabc"` 가 되는 것과 같은 이치이다.

[초보자가 범하기 쉬운 리스트 연산 오류]

다음과 같은 소스 코드를 입력했을 때 결과값은 어떻게 나올까?

```
>>>a = [1, 2,
3]
>>>a[2] + "hi"
```

a[2]의 값인 3과 문자열 hi가 더해져서 3hi가 출력될 것이라고 생각할 수 있다. 하지만 다음의 결과를 보자. 형 오류(TypeError)가 발생했다. 오류의 원인은 무엇일까?

```
Traceback (innermost last):
File "", line 1, in ?
a[2] + "hi"
TypeError: number coercion
failed
```

a[2]에 저장된 값은 3이라는 정수인데 "hi"는 문자열이다. 정수와 문자열은 당연히 서로 더할 수 없기 때문에 형 오류가 발생한 것이다.

만약 숫자와 문자열을 더해서 '3hi'처럼 만들고 싶다면 숫자 3을 문자 '3'으로 바꾸어 주어야 한다. 다음과 같이 할 수 있다.

```
>>>str(a[2]) +
"hi"
```

str()은 정수나 실수를 문자열의 형태로 바꾸어 주는 파이썬의 내장 함수이다.

리스트의 수정, 변경과 삭제

다음의 예들은 서로 연관되어 있으므로 따로따로 실행하지 말고 차례대로 진행해야 한다.

1. 리스트에서 하나의 값 수정하기

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

a[2]의 요소값 3이 4로 바뀌었다.

2. 리스트에서 연속된 범위의 값 수정하기

```
>>> a[1:2]
[2]
>>> a[1:2] = ['a', 'b',
'c']
>>> a
[1, 'a', 'b', 'c', 4]
```

a[1:2]는 a[1]부터 a[2]까지를 말하는데 a[2]는 포함하지 않는다고 했으므로 a = [1, 2, 4]에서 a[1]의 값인 2만을 의미한다. 즉, a[1:2]를 ['a', 'b', 'c']로 바꾸었으므로 a 리스트에서 2라는 값 대신에 ['a', 'b', 'c']라는 값이 대입되었다.

[리스트 수정할 때 주의할 점]

2번 예제에서 리스트를 a[1:2] = ['a', 'b', 'c']로 수정하는 것과 a[1] = ['a', 'b', 'c']로 수정하는 것은 전혀 다른 결과값을 갖게 되므로 주의해야 한다. a[1] = ['a', 'b', 'c']는 리스트 a의 두 번째 요소를 ['a', 'b','c']로 바꾼다는 말이고 a[1:2]는 a[1]에서 a[2] 사이의 리스트를 ['a', 'b', 'c']로 바꾼다는 말이다. 따라서 a[1] = ['a', 'b', 'c']로 수정하게 되면 위와는 달리 리스트 a가 [1, ['a', 'b', 'c'], 4]라는 값으로 변하게 된다.

```
>>> a[1] = ['a', 'b',
'c']
>>> a
[1, ['a', 'b', 'c'], 4]
```

3. [ ] 사용해 리스트 요소 삭제하기

```
>>> a[1:3] = [
]
>>> a
[1, 'c', 4]
```

2번까지 진행한 리스트 a의 값은 [1, 'a', 'b', 'c', 4]였다. 여기서 a[1:3]은 a의 인덱스 1부터 3까지(1<=a<3), 즉 a[1], a[2]를 의미하므로 a[1:3]은 ['a', 'b']이다. 그런데 위의 예에서 볼 수 있듯이 a[1:3]을 [ ]으로 바꿔 주었기 때문에 a에서 ['a', 'b']가 삭제된 [1, 'c', 4]가 된다.

4. del 함수 사용해 리스트 요소 삭제하기

```
>>> a
[1, 'c', 4]
>>> del
a[1]
>>> a
[1, 4]
```

del a[x]는 x번째 요소값을 삭제한다. del a[x:y]는 x번째부터 y번째 요소 사이의 값을 삭제한다. 여기서는 a 리스트에서 a[1]을 삭제하는 방법을 보여 준다. del 함수는 파이썬이 자체적으로 가지고 있는 삭제 함수이며 다음과 같이 사용한다.

> *del 객체*

(※ 객체란 파이썬에서 사용되는 모든 자료형을 말한다.)

리스트 관련 함수들

문자열과 마찬가지로 리스트 변수명 뒤에 '.'를 붙여서 여러 가지 리스트 관련 함수들을 이용할 수 있다. 유용하게 사용되는 리스트 관련 함수 몇 가지에 대해서만 알아보기로 하자.

리스트에 요소 추가(append)

append를 사전에서 검색해 보면 "덧붙이다, 첨부하다"라는 뜻이 있다. 이 뜻을 안다면 아래의 예가 금방 이해가

될 것이다. append(x)는 리스트의 맨 마지막에 x를 추가시키는 함수이다.

```
>>> a = [1, 2,
3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
```

리스트 안에는 어떤 자료형도 추가할 수 있다.

아래의 예는 리스트에 다시 리스트를 추가한 결과이다.

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

리스트 정렬(sort)

sort 함수는 리스트의 요소를 순서대로 정렬해 준다.

```
>>> a = [1, 4, 3,
2]
>>> a.sort()
>>> a
[1, 2, 3, 4]
```

문자 역시 알파벳 순서로 정렬할 수 있다.

```
>>> a = ['a', 'c',
'b']
>>> a.sort()
>>> a
['a', 'b', 'c']
```

리스트 뒤집기(reverse)

reverse 함수는 리스트를 역순으로 뒤집어 준다. 이때 리스트 요소들을 순서대로 정렬한 다음 다시 역순으로 정렬하는 것이 아니라 그저 현재의 리스트를 그대로 거꾸로 뒤집을 뿐이다.

```
>>> a = ['a', 'c',
'b']
>>> a.reverse()
>>> a
['b', 'c', 'a']
```

위치 반환(index)

index(x) 함수는 리스트에 x라는 값이 있으면 x의 위치값을 리턴한다.

```
>>> a =
[1,2,3]
>>> a.index(3)
2
>>> a.index(1)
0
```

위의 예에서 리스트 a에 있는 3이라는 숫자의 위치는 a[2]이므로 2를 리턴하고, 1이라는 숫자의 위치는 a[0]이므로 0을 리턴한다.

아래의 예에서 0이라는 값은 a 리스트에 존재하지 않기 때문에 값 오류(ValueError)가 발생한다.

```
              Traceback (most recent call last):
                File "<stdin>", line 1, in
>>>           <module>
a.index(0)    ValueError: 0 is not in list
```

리스트에 요소 삽입(insert)

insert(a, b)는 리스트의 a번째 위치에 b를 삽입하는 함수이다. 파이썬에서는 숫자를 0부터 센다는 것을 반드시 기억하자.

```
>>> a = [1, 2, 3]
>>> a.insert(0,
4)
[4, 1, 2, 3]
```

위의 예는 0번째 자리, 즉 첫 번째 요소(a[0]) 위치에 4라는 값을 삽입하라는 뜻이다.

```
>>> a.insert(3,
5)
[4, 1, 2, 5, 3]
```

위의 예는 리스트 a의 a[3], 즉 네 번째 요소 위치에 5라는 값을 삽입하라는 뜻이다.

리스트 요소 제거(remove)

remove(x)는 리스트에서 첫 번째로 나오는 x를 삭제하는 함수이다.

```
>>> a = [1, 2, 3, 1, 2,
3]
>>> a.remove(3)
[1, 2, 1, 2, 3]
```

a가 3이라는 값을 2개 가지고 있을 경우 첫 번째 3만 제거되는 것을 알 수 있다.

```
>>>
a.remove(3)
[1, 2, 1, 2]
```

remove(3)을 한 번 더 실행하면 다시 3이 삭제된다.

리스트 요소 끄집어내기(pop)

pop()은 리스트의 맨 마지막 요소를 돌려 주고 그 요소는 삭제하는 함수이다.

```
>>> a =
[1,2,3]
>>> a.pop()
3
>>> a
[1, 2]
```

a 리스트 [1,2,3]에서 3을 끄집어내고 최종적으로 [1, 2]만 남는 것을 볼 수 있다.

pop(x)는 리스트의 x번째 요소를 돌려 주고 그 요소는 삭제한다.

```
>>> a =
[1,2,3]
>>> a.pop(1)
2
>>> a
[1, 3]
```

a.pop(1)은 a[1]의 값을 끄집어낸다. 다시 a를 출력해 보면 끄집어낸 값이 삭제된 것을 확인할 수 있다.

리스트에 포함된 요소 x의 개수 세기(count)

count(x)는 리스트 내에 x가 몇 개 있는지 조사하여 그 개수를 돌려주는 함수이다.

```
>>> a =
[1,2,3,1]
>>> a.count(1)
2
```

1이라는 값이 리스트 a에 2개 들어 있으므로 2를 돌려준다.

리스트 확장(extend)

extend(x)에서 x에는 리스트만 올 수 있으며 원래의 a 리스트에 x 리스트를 더하게 된다.

```
>>> a = [1,2,3]
>>> a.extend([4,5])
>>> a
[1, 2, 3, 4, 5]
>>> b = [6, 7]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 5, 6,
7]
```

a.extend([4,5])는 a += [4,5]와 동일하고, a += [4, 5]는 a = a + [4, 5]와 같은 표현이다.

02-4 튜플 자료형

튜플은 어떻게 만들까?

튜플(tuple)은 몇 가지 점을 제외하곤 리스트와 거의 비슷하며 리스트와 다른 점은 다음과 같다.

- 리스트는 [과]으로 둘러싸지만 튜플은 (과)으로 둘러싼다.
- 리스트는 그 값의 생성, 삭제, 수정이 가능하지만 튜플은 그 값을 바꿀 수 없다.

튜플의 모습은 다음과 같다.

```
>>> t1 = ()
>>> t2 = (1,)
>>> t3 = (1, 2, 3)
>>> t4 = 1, 2, 3
>>> t5 = ('a', 'b', ('ab',
'cd'))
```

리스트와 모습은 거의 비슷하지만 튜플에서는 리스트와 다른 2가지 차이점을 찾아볼 수 있다. t2 = (1,)처럼 단지 1개의 요소만을 가질 때는 요소 뒤에 콤마(,)를 반드시 붙여야 한다는 것과 t4 = 1, 2, 3처럼 괄호()를 생략해도 무방하다는 점이다.

얼핏 보면 튜플과 리스트는 비슷한 역할을 하지만 프로그래밍을 할 때 튜플과 리스트는 구분해서 사용하는 것이 유리하다. 튜플과 리스트의 가장 큰 차이는 값을 변화시킬 수 있는가 없는가이다. 즉, 리스트의 항목값은 변화가 가능하고 튜플의 항목값은 변화가 불가능하다. 따라서 프로그램이 실행되는 동안 그 값이 항상 변하지 않기를 바란다거나 값이 바뀔까 걱정하고싶지 않다면 주저하지 말고 튜플을 사용해야 한다. 이와는 반대로 수시로 그 값을 변화시켜야할 경우라면 리스트를 사용해야 한다. 실제 프로그램에서는 값이 변경되는 형태의 변수가 훨씬 많기 때문에 평균적으로 튜플보다는 리스트를 더 많이 사용하게 된다.

튜플의 요소값을 지우거나 변경하려고 하면 어떻게 될까?

앞서 설명했듯이 튜플의 요소값은 한 번 정하면 지우거나 변경할 수 없다. 다음에 소개하는 2개의 예를 살펴보면 무슨 말인지 알 수 있을 것이다.

1. 튜플 요소값 삭제 시 오류

```
>>> t1 = (1, 2, 'a',
'b')
>>> del t1[0]
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object doesn't support item
deletion
```

튜플의 요소를 리스트처럼 del 함수로 지우려고 한 예이다. 튜플은 요소를 지우는 행위가 지원되지 않는다는 메시지를 확인할 수 있다.

2. 튜플 요소값 변경 시 오류

```
>>> t1 = (1, 2, 'a',
'b')
>>> t1[0] = 'c'
```

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

튜플의 요소값을 변경하려고 해도 마찬가지로 오류가 발생하는 것을 확인할 수 있다.

튜플의 인덱싱과 슬라이싱, 더하기(+)와 곱하기(*)

튜플은 값을 변화시킬 수 없다는 점만 제외하면 리스트와 완전히 동일하므로 간단하게만 살펴보겠다. 다음의 예제는 서로 연관되어 있으므로 차례대로 수행해 보기 바란다.

1. 인덱싱하기

```
>>> t1 = (1, 2, 'a',
'b')
>>> t1[0]
1
>>> t1[3]
'b'
```

문자열, 리스트와 마찬가지로 t1[0], t1[3]처럼 인덱싱이 가능하다.

2. 슬라이싱하기

```
>>> t1 = (1, 2, 'a',
'b')
>>> t1[1:]
(2, 'a', 'b')
```

t1[1]부터 튜플의 마지막 요소까지 슬라이싱하는 예이다.

3. 튜플 더하기

```
>>> t2 = (3, 4)
>>> t1 + t2
(1, 2, 'a', 'b', 3,
4)
```

튜플을 더하는 방법을 보여주는 예이다.

4. 튜플 곱하기

```
>>> t2 * 3
(3, 4, 3, 4, 3,
4)
```

튜플의 곱하기(반복) 예를 보여 준다.

02-5 딕셔너리 자료형

사람은 누구든지 "이름" = "홍길동", "생일" = "몇 월 몇 일" 등으로 구분할 수 있다. 파이썬은 영리하게도 이러한 대응 관계를 나타낼 수 있는 자료형을 가지고 있다. 요즘 사용하는 대부분의 언어들도 이러한 대응 관계를 나타내는 자료형을 갖고 있는데, 이를 연관 배열(Associative array) 또는 해시(Hash)라고 한다.

파이썬에서는 이러한 자료형을 딕셔너리(Dictionary)라고 하는데, 단어 그대로 해석하면 사전이라는 뜻이다. 즉, people이라는 단어에 "사람", baseball이라는 단어에 "야구"라는 뜻이 부합되듯이 딕셔너리는 Key와 Value라는 것을 한 쌍으로 갖는 자료형이다. 예컨대 Key가 "baseball"이라면 Value는 "야구"가될 것이다.

딕셔너리는 리스트나 튜플처럼 순차적으로(sequential) 해당 요소값을 구하지 않고 Key를 통해 Value를 얻는다. 이것이 바로 딕셔너리의 가장 큰 특징이다. baseball이라는 단어의 뜻을 찾기 위해 사전의 내용을 순차적으로 모두 검색하는 것이 아니라 baseball이라는 단어가 있는 곳만 펼쳐 보는 것이다.

딕셔너리는 어떻게 만들까?

다음은 기본적인 딕셔너리의 모습이다.

> *{Key1:Value1, Key2:Value2, Key3:Value3 ...}*

Key와 Value의 쌍 여러 개가 {과 }로 둘러싸여 있다. 각각의 요소는 Key : Value 형태로 이루어져 있고 쉼표(,) 로 구분되어 있다.

(※ Key에는 변하지 않는 값을 사용하고, Value에는 변하는 값과 변하지 않는 값 모두 사용할 수 있다.)

다음의 딕셔너리 예를 살펴보자.

```
>>> dic = {'name':'pey', 'phone':'0119993323', 'birth':
'1118'}
```

위에서 Key는 각각 'name', 'phone', 'birth'이고, 각각의 Key에 해당하는 Value는 'pey', '0119993323', '1118'이 된다.

딕셔너리 dic의 정보

key	value
name	pey
phone	01199993323
birth	1118

다음의 예는 Key로 정수값 1, Value로 'hi'라는 문자열을 사용한 예이다.

```
>>> a = {1:
'hi'}
```

또한 다음의 예처럼 Value에 리스트도 넣을 수 있다.

```
>>> a = { 'a':
[1,2,3]}
```

딕셔너리 쌍 추가, 삭제하기

딕셔너리 쌍을 추가하는 방법과 삭제하는 방법을 살펴보자. 1번은 딕셔너리에 쌍을 추가하는 예이다. 딕셔너리는 순서를 따지지 않는다는 사실을 기억하자. 다음 예에서 알 수 있듯이 추가되는 순서는 원칙이 없다. 중요한 것은 "무엇이 추가되었는가"이다.

다음의 예를 함께 따라 해보자.

1. 딕셔너리 쌍 추가하기

```
>>> a = {1:
'a'}
>>> a[2] = 'b'
>>> a
{2: 'b', 1:
'a'}
```

{1: 'a'}라는 딕셔너리에 a[2] = 'b'와 같이 입력하면 딕셔너리 a에 Key와 Value가 각각 2와'b'인 2 : 'b'라는 딕셔너리 쌍이 추가된다.

```
>>> a['name'] = 'pey'
{'name':'pey', 2: 'b', 1:
'a'}
```

딕셔너리 a에 'name': 'pey'라는 쌍이 추가되었다.

```
>>> a[3] = [1,2,3]
{'name': 'pey', 3: [1, 2, 3], 2: 'b', 1:
'a'}
```

Key는 3, Value는 [1, 2, 3]을 가지는 한 쌍이 또 추가되었다.

2. 딕셔너리 요소 삭제하기

```
>>> del a[1]
>>> a
{'name': 'pey', 3: [1, 2, 3], 2:
'b'}
```

위의 예제는 딕셔너리 요소를 지우는 방법을 보여준다. del 함수를 사용해서 del a[key]처럼 입력하면 지정한 key에 해당하는 {key : value} 쌍이 삭제된다.

딕셔너리를 사용하는 방법

"딕셔너리는 주로 어떤 것을 표현하는 데 사용할까?"라는 의문이 들 것이다. 예를 들어 4명의 사람이 있다고 가정하고, 각자의 특기를 표현할 수 있는 좋은 방법에 대해서 생각해 보자. 리스트나 문자열로는 표현하기가 상당히 까다로울 것이다. 하지만 파이썬의 딕셔너리를 사용한다면 이 상황을 표현하기가 정말 쉽다.

다음의 예를 보자.

```
{"김연아":"피겨스케이팅", "류현진":"야구", "박지성":"축구", "귀도":"파이
썬"}
```

사람 이름과 특기를 한 쌍으로 하는 딕셔너리이다. 정말 간편하지 않은가? 지금껏 우리는 딕셔너리를 만드는 방법에 대해서만 살펴보았는데 딕셔너리를 제대로 활용하기 위해서는 알아야 할 것들이 있다. 이제부터 하나씩 알아보도록 하자.

딕셔너리에서 Key 사용해 Value 얻기

다음의 예를 살펴보자.

```
>>> grade = {'pey': 10, 'julliet':
99}
>>> grade['pey']
10
>>> grade['julliet']
99
```

리스트나 튜플, 문자열은 요소값을 얻어내고자 할 때 인덱싱이나 슬라이싱 기법 중 하나를 이용했다. 하지만 딕셔너리는 단 한 가지 방법뿐이다. 바로 Key를 사용해서 Value를 얻어내는 방법이다. 위의 예에서 'pey'라는 Key의 Value를 얻기 위해 grade['pey']를 사용한 것처럼 어떤 Key의 Value를 얻기 위해서는 "딕셔너리 변수[Key]"를 사용한다.

몇 가지 예를 더 보자.

```
>>> a = {1:'a',
2:'b'}
>>> a[1]
'a'
>>> a[2]
'b'
```

먼저 a라는 변수에 {1:'a', 2:'b'}라는 딕셔너리를 대입하였다. 위의 예에서 볼 수 있듯이 a[1]은 'a'라는 값을 돌려준다. 여기서 a[1]이 의미하는 것은 리스트나 튜플의 a[1]과는 전혀 다르다. 딕셔너리 변수에서 [ ] 안의 숫자 1은 두 번째 요소를 뜻하는 것이 아니라 Key에 해당하는 1을 나타낸다. 앞에서도 말했듯이 딕셔너리는 리스트나 튜플에 있는 인덱싱 방법을 적용할수 없다. 따라서 a[1]은 딕셔너리 {1:'a', 2:'b'}에서 Key가 1인 것의 Value인 'a'를 돌려주게 된다. a[2] 역시 마찬가지이다.

이번에는 a라는 변수에 앞의 예에서 사용했던 딕셔너리의 Key와 Value를 뒤집어 놓은 딕셔너리를 대입해 보자.

```
>>> a = {'a':1,
'b':2}
>>> a['a']
1
>>> a['b']
2
```

역시 a['a'], a['b']처럼 Key를 사용해서 Value를 얻을 수 있다. 정리하면, 딕셔너리 a는 a[Key]로 입력해서 Key에 해당하는 Value를 얻는다.

다음 예는 이전에 한 번 언급했던 딕셔너리인데 Key를 사용해서 Value를 얻는 방법을 잘 보여 준다.

```
>>> dic = {'name':'pey', 'phone':'0119993323', 'birth':
'1118'}
>>> dic['name']
'pey'
>>> dic['phone']
'0119993323'
>>> dic['birth']
'1118'
```

딕셔너리 만들 때 주의할 사항

먼저 딕셔너리에서 Key는 고유한 값이므로 중복되는 Key 값을 설정해 놓으면 하나를 제외한 나머지 것들이 모두 무시된다는 점을 주의해야 한다. 다음 예에서 볼 수 있듯이 동일한 Key가 2개 존재할 경우 1:'a'라는 쌍이 무시된다. 이때 꼭 앞에 쓴 것이 무시되는 것은 아니고 어떤 것이 무시될지는 예측할 수 없다. 결론은 중복되는 Key를 사용하지 말라는 것이다.

```
>>> a = {1:'a',
1:'b'}
>>> a
{1: 'b'}
```

이렇게 Key가 중복되었을 때 1개를 제외한 나머지 Key:Value 값이 모두 무시되는 이유는 Key를 통해서 Value를 얻는 딕셔너리의 특징에서 비롯된다. 즉, 동일한 Key가 존재하면 어떤 Key에 해당하는 Value를 불러야 할지 알 수 없기 때문이다.

또 한 가지 주의해야 할 사항은 Key에 리스트는 쓸 수 없다는 것이다. 하지만 튜플은 Key로 쓸 수 있다. 딕셔너리의 Key로 쓸 수 있느냐 없느냐는 Key가 변하는 값인지 변하지 않는 값인지에 달려 있다. 리스트는 그 값이 변할 수 있기 때문에 Key로 쓸 수 없는 것이다. 아래 예처럼 리스트를 Key로 설정하면 리스트를 키 값으로 사용할 수 없다는 형 오류(TypeError)가 발생한다.

```
>>> a = {[1,2] : 'hi'}
Traceback (most recent call
last):
File "", line 1, in ?
TypeError: unhashable type
```

따라서 딕셔너리의 Key 값으로 딕셔너리를 사용할 수 없음은 당연한 얘기가 될 것이다. 단, Value에는 변하는 값이든 변하지 않는 값이든 상관없이 아무 값이나 넣을 수 있다.

딕셔너리 관련 함수들

딕셔너리를 자유자재로 사용하기 위해 딕셔너리가 자체적으로 가지고 있는 관련 함수들을 사용해 보도록 하자.

Key 리스트 만들기(keys)

```
>>> a = {'name': 'pey', 'phone': '0119993323', 'birth':
'1118'}
>>> a.keys()
dict_keys(['name', 'phone', 'birth'])
```

a.keys()는 딕셔너리 a의 Key만을 모아서 dict_keys라는 객체를 리턴한다.

[파이썬 3.0 이후 버전의 keys 함수, 어떻게 달라졌나?]

파이썬 2.7 버전까지는 a.keys() 호출 시 리턴값으로 dict_keys가 아닌 리스트를 리턴한다. 리스트를 리턴하기 위해서는 메모리의 낭비가 발생하는데 파이썬 3.0 이후 버전에서는 이러한 메모리 낭비를 줄이기 위해 dict_keys라는 객체를 리턴해 준다. 다음에 소개할 dict_values, dict_items 역시 파이썬 3.0 이후 버전에서 추가된 것들이다. 만약 3.0 이후 버전에서 리턴값으로 리스트가 필요한 경우에는 "list(a.keys())"를 사용하면 된다. dict_keys, dict_values, dict_items 등은 리스트로 변환하지 않더라도 기본적인 반복성(iterate) 구문(예: for문)들을 실행할 수 있다.

dict_keys 객체는 다음과 같이 사용할 수 있다. 리스트를 사용하는 것과 차이가 없지만, 리스트 고유의 함수인 append, insert, pop, remove, sort등의 함수를 수행할 수는 없다.

```
>>> for k in
a.keys():
...     print(k)
...
phone
birth
name
```

dict_keys 객체를 리스트로 변환하려면 다음과 같이 하면 된다.

```
>>> list(a.keys())
['phone', 'birth',
'name']
```

Value 리스트 만들기(values)

```
>>> a.values()
dict_values(['pey', '0119993323',
'1118'])
```

Key를 얻는 것과 마찬가지 방법으로 Value만 얻고 싶다면 a.values()처럼 values 함수를 사용하면 된다. values 함수를 호출하면 dict_values 객체가 리턴되는데, dict_values 객체 역시 dict_keys 객체와 마찬가지로 리스트를 사용하는 것과 동일하게 사용하면 된다.

Key, Value 쌍 얻기(items)

```
>>> a.items()
dict_items([('name', 'pey'), ('phone', '0119993323'), ('birth',
'1118')])
```

items 함수는 key와 value의 쌍을 튜플로 묶은 값을 dict_items 객체로 돌려준다.

Key: Value 쌍 모두 지우기(clear)

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

clear() 함수는 딕셔너리 안의 모든 요소를 삭제한다. 빈 리스트를 [ ], 빈 튜플을 ()로 표현하는 것과 마찬가지로 빈 딕셔너리도 { }로 표현한다.

Key로 Value얻기(get)

```
>>> a = {'name':'pey', 'phone':'0119993323', 'birth':
'1118'}
>>> a.get('name')
'pey'
>>> a.get('phone')
'0119993323'
```

get(x) 함수는 x라는 key에 대응되는 value를 돌려준다. 앞서 살펴보았듯이 a.get('name')은 a['name']을 사용했을 때와 동일한 결과값을 돌려받는다.

다만, 다음 예제에서 볼 수 있듯이 a['nokey']처럼 존재하지 않는 키(nokey)로 값을 가져오려고 할 경우 a['nokey']는 Key 오류를 발생시키고 a.get('nokey')는 None을 리턴한다는 차이가 있다. 어떤것을 사용할지는 여러분의 선택이다.

(※ 여기서 None은 "거짓"이라는 뜻이라고만 알아두자.)

```
>>> a.get('nokey')
>>> a['nokey']
Traceback (most recent call last):
File "<stdin>", line 1, in
<module>
KeyError: 'nokey'
```

딕셔너리 안에 찾으려는 key 값이 없을 경우 미리 정해 둔 디폴트 값을 대신 가져오게 하고 싶을 때에는 get(x, '디폴트 값')을 사용하면 편리하다.

```
>>> a.get('foo',
'bar')
'bar'
```

a 딕셔너리에는 'foo'에 해당하는 값이 없다. 따라서 디폴트 값인 'bar'를 리턴한다.

해당 Key가 딕셔너리 안에 있는지 조사하기(in)

```
>>> a = {'name':'pey', 'phone':'0119993323', 'birth':
'1118'}
>>> 'name' in a
True
>>> 'email' in a
False
```

'name'이라는 문자열은 a 딕셔너리의 key 중 하나이다. 따라서 'name' in a를 호출하면 참(True)을 리턴한다. 반대로 'email' 은 a 딕셔너리 안에 존재하지 않는 key이므로 거짓(False)을 리턴하게 된다.

02-6 집합 자료형

집합 자료형은 어떻게 만들까?

집합(set)은 파이썬 2.3부터 지원되기 시작한 자료형으로, 집합에 관련된 것들을 쉽게 처리하기 위해 만들어진 자료형이다.

집합 자료형은 다음과 같이 set 키워드를 이용해 만들 수 있다.

```
>>> s1 =
set([1,2,3])
>>> s1
{1, 2, 3}
```

위와 같이 set()의 괄호 안에 리스트를 입력하여 만들거나 아래와 같이 문자열을 입력하여 만들 수도 있다.

```
>>> s2 =
set("Hello")
>>> s2
{'e', 'l', 'o', 'H'}
```

집합 자료형의 특징

자, 그런데 위에서 살펴본 set("Hello")의 결과가 좀 이상하지 않은가? 분명 "Hello"라는 문자열로 set 자료형을 만들었는데 생성된 자료형에는 l 문자가 하나 빠져 있고 순서도 뒤죽박죽이다. 그 이유는 set에는 다음과 같은 2가지 큰 특징이 있기 때문이다.

- 중복을 허용하지 않는다.
- 순서가 없다(Unordered).

리스트나 튜플은 순서가 있기(ordered) 때문에 인덱싱을 통해 자료형의 값을 얻을 수 있지만 set 자료형은 순서가 없기(unordered) 때문에 인덱싱으로 값을 얻을 수 없다. 이는 마치 02-5절에서 살펴본 딕셔너리와 비슷하다. 딕셔너리 역시 순서가 없는 자료형이라 인덱싱을 지원하지 않는다. 만약 set 자료형에 저장된 값을 인덱싱으로 접근하려면 다음과 같이 리스트나 튜플로 변환한 후 해야 한다.

(※ 중복을 허용하지 않는 set의 특징은 자료형의 중복을 제거하기 위한 필터 역할로 종종 사용되기도 한다.)

```
>>> s1 =
set([1,2,3])
>>> l1 = list(s1)
>>> l1
[1, 2, 3]
>>> l1[0]
1
>>> t1 = tuple(s1)
>>> t1
(1, 2, 3)
>>> t1[0]
1
```

집합 자료형 활용하는 방법

교집합, 합집합, 차집합 구하기

set 자료형이 정말 유용하게 사용되는 경우는 다음과 같이 교집합, 합집합, 차집합을 구할 때이다.

우선 다음과 같이 2개의 set 자료형을 만든 후 따라 해보자. s1은 1부터 6까지의 값을 가지게 되었고, s2는 4부터 9까지의 값을 가지게 되었다.

```
>>> s1 = set([1, 2, 3, 4, 5,
6])
>>> s2 = set([4, 5, 6, 7, 8,
9])
```

1. 교집합

s1과 s2의 교집합을 구해 보자.

```
>>> s1 &
s2
{4, 5, 6}
```

"&" 기호를 이용하면 교집합을 간단히 구할 수 있다.

또는 다음과 같이 intersection 함수를 사용해도 동일한 결과를 리턴한다.

```
>>>
s1.intersection(s2)
{4, 5, 6}
```

s2.intersection(s1)을 사용해도 결과는 같다.

2. 합집합

합집합은 다음과 같이 구할 수 있다. 이때 4, 5, 6처럼 중복해서 포함된 값은 한 개씩만 표현된다.

```
>>> s1 | s2
{1, 2, 3, 4, 5, 6, 7, 8,
9}
```

"|" 기호를 이용한 방법이다.

```
>>> s1.union(s2)
{1, 2, 3, 4, 5, 6, 7, 8,
9}
```

또는 union 함수를 이용하면 된다. 교집합에서 사용했던 intersection 함수와 마찬가지로 s2.union(s1)을 사용해도 동일한 결과를 리턴한다.

3. 차집합

차집합은 다음과 같이 구할 수 있다.

```
>>> s1 -
s2
{1, 2, 3}
>>> s2 -
s1
{8, 9, 7}
```

빼기(-) 기호를 이용한 방법이다.

```
>>>
s1.difference(s2)
{1, 2, 3}
>>>
s2.difference(s1)
{8, 9, 7}
```

difference 함수를 이용해도 차집합을 구할 수 있다.

집합 자료형 관련 함수들

값 1개 추가하기(add)

이미 만들어진 set 자료형에 값을 추가할 수 있다. 1개의 값만 추가(add)할 경우에는 아래와 같이 한다.

```
>>> s1 = set([1, 2,
3])
>>> s1.add(4)
>>> s1
{1, 2, 3, 4}
```

값 여러 개 추가하기(update)

여러 개의 값을 한꺼번에 추가(update)할 때는 다음과 같이 하면 된다.

```
>>> s1 = set([1, 2, 3])
>>> s1.update([4, 5,
6])
>>> s1
{1, 2, 3, 4, 5, 6}
```

특정 값 제거하기(remove)

특정 값을 제거하고 싶을 때는 아래와 같이 하면 된다.

```
>>> s1 = set([1, 2,
3])
>>> s1.remove(2)
>>> s1
{1, 3}
```

지금까지 파이썬의 가장 기본이 되는 자료형인 숫자, 문자열, 리스트, 튜플, 딕셔너리, 집합에 대해서 알아보았다.

여기까지 잘 따라온 독자라면 파이썬에 대해서 대략 50% 정도 습득했다고 보아도 된다. 그만큼 자료형은 중요하고 프로그램의 근간이 되기 때문에 확실하게 해놓지 않으면 좋은 프로그램을 만들 수 없다. 책에 있는 예제들만 따라 하지 말고 직접 여러 가지 예들을 테스트해 보며 02-1부터 02-6절까지의 자료형들에 익숙해지기를 당부한다.

02-7 자료형의 참과 거짓

자료형에 참과 거짓이 있다? 조금 이상하게 들리겠지만 참과 거짓은 분명히 있다. 이는 매우 중요한 특징이며 실제로도 자주 쓰인다.

자료형의 참과 거짓을 구분하는 기준은 다음과 같다.

값	참 or 거짓
"python"	참
""	거짓
[1, 2, 3]	참
[]	거짓
()	거짓
{}	거짓
1	참
0	거짓
None	거짓

문자열, 리스트, 튜플, 딕셔너리 등의 값이 비어 있으면(" ", [ ], ( ), { }) 거짓이 된다. 당연히 비어있지 않으면 참이 된다. 숫자에서는 그 값이 0일 때 거짓이 된다. 위의 표를 보면 None이라는 것이 있는데, 이것에 대해서는 뒷부분에서 배우니 아직은 신경 쓰지 말자. 그저 None은 거짓을 뜻한다는 것만 알아두자.

다음의 예를 보고 참과 거짓이 프로그램에서 어떻게 쓰이는지 간단히 알아보자.

```
>>> a = [1, 2, 3,
4]
>>> while a:
...     a.pop()
...
4
3
2
1
```

먼저 a = [1, 2, 3, 4]라는 리스트를 하나 만들었다.

while문은 3장에서 자세히 다루겠지만 간단히 알아보면 다음과 같다. 조건문이 참인 동안 조건문 안에 있는 문장을 반복해서 수행한다.

```
while 조건문:
    수행할 문
장
```

즉, 위의 예를 보면 a가 참인 경우에 a.pop()을 계속 실행하라는 의미이다. a.pop()이라는 함수는 리스트 a의 마지막 요소를 끄집어내는 함수이므로 a가 참인 동안(리스트 내에 요소가 존재하는 한) 마지막 요소를 계속해서 끄집

어낼 것이다. 결국 더 이상 끄집어낼 것이 없으면 a가 빈 리스트([ ])가 되어 거짓이 된다. 따라서 while문에서 조건이 거짓이 되므로 중지된다. 위에서 본 예는 파이썬 프로그래밍에서 매우 자주 이용하는 기법 중 하나이다.

위의 예가 너무 복잡하다고 생각하는 독자는 다음의 예를 보면 쉽게 이해가 될 것이다.

```
>>> if [ ]:
...     print("True")
... else:
...
print("False")
...
False
```

if문에 대해서 잘 모르는 독자라도 위의 문장을 해석하는 데는 무리가 없을 것이다.

(※ if문에 대해서는 03장에서 자세히 다룬다.)

[ ]는 앞의 표에서 볼 수 있듯이 비어 있는 리스트이므로 거짓이다. 따라서 False란 문자열이 출력된다.

```
>>> if [1, 2, 3]:
...     print("True")
... else:
...
print("False")
...
True
```

위 코드를 해석해 보면 다음과 같다.

> *만약 [1, 2, 3]이 참이면 "True"라는 문자열을 출력하고 그렇지 않으면 "False"라는 문자열을 출력하라.*

위 코드의 [1, 2, 3]은 요소값이 있는 리스트이기 때문에 참이다. 따라서 True를 출력한다.

02-8 자료형의 값을 저장하는 공간, 변수

우리는 앞에서 이미 변수들을 사용해 왔다. 다음 예와 같은 a, b, c를 변수라고 한다.

```
>>> a = 1
>>> b =
"python"
>>> c = [1,2,3]
```

변수를 만들 때는 위의 예처럼 =(assignment) 기호를 사용한다.

C 언어나 JAVA처럼 변수의 자료형을 함께 쓸 필요는 없다. 파이썬은 변수에 저장된 값을 스스로 판단하여 자료형을 알아낸다.

> *변수명 = 변수에 저장할 값*

지금부터 설명할 내용은 프로그래밍 초보자가 얼른 이해하기 어려운 부분이므로 당장 이해가 되지 않는다면 그냥 건너뛰어도 무방하다. 파이썬에 대해서 공부하다 보면 자연스럽게 알게 될 것이다.

변수란?

파이썬에서 사용하는 변수는 객체를 가리키는 것이라고도 말할 수 있다. 객체란 우리가 지금껏 보아 왔던 자료형을 포함해 "파이썬에서 사용되는 모든 것"을 뜻하는 말이다.

```
>>> a =
3
```

만약 위의 코드처럼 a = 3이라고 하면 3이라는 값을 가지는 정수 자료형(객체)이 자동으로 메모리에 생성된다. a는 변수의 이름이며, 3이라는 정수형 객체가 저장된 메모리 위치를 가리키게 된다. 즉, 변수 a는 객체가 저장된 메모리의 위치를 가리키는 레퍼런스(Reference)라고도 할 수 있다.

만약 메모리의 위치를 가리킨다는 말이 잘 이해되지 않는다면 다음처럼 생각해도 무방하다. 즉, a라는 변수는 3이라는 정수형 객체를 가리키고 있다.

> *a --> 3*

[파이썬에서 "3"은 상수가 아닌 정수형 객체이다]

파이썬의 모든 자료형은 객체라고 했다. 그러므로 3은 우리가 일반적으로 생각하는 상수값이 아닌 하나의 "정수형 객체"이다. 따라서 a=3과 같이 선언하면 a.real처럼 내장 함수를 바로 사용할 수 있게 된다. C나 JAVA를 먼저 공부한 사람이라면 이 점이 헷갈릴 수 있다.

3의 자료형을 확인하는 아래 예제를 보면 좀 더 명확히 이해할 수 있을 것이다. type은 입력받은 객체의 자료형이 무엇인지 알려주는 함수이다.

```
>>> type(3)
<class
'int'>
```

다음 예를 보자.

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

a가 3을 가리키고 b도 3을 가리킨다. 즉 a = 3을 입력하는 순간 3이라는 정수형 객체가 생성되고 변수 a는 3이라는 객체의 메모리 주소를 가리킨다. 다음에 변수 b가 동일한 객체인 3을 가리킨다. 이제 3이라는 정수형 객체를 가리키는 변수가2개가 됐다. 이 두 변수는 가리키고 있는 대상이 동일하다. 따라서 동일한 객체를 가리키고 있는지 아닌지에 대해서 판단하는 파이썬 내장 함수인 is 함수를 a is b처럼 실행했을 때 참(True)을 리턴하게 된다.

3이라는 객체를 가리키고 있는 변수의 개수는 2개이다. 이것을 조금 어려운 말로 표현해 레퍼런스 카운트(Reference Count, 참조 개수)가 2개라고 한다. 만약 c = 3이라고 한 번 더 입력한다면 레퍼런스 카운트는 3이 될 것이다.

[a, b, c는 정말 같은 객체를 가리키는 걸까?]

파이썬에는 입력한 자료형에 대한 참조 개수를 알려주는 sys.getrefcount라는 함수가 있다. 이 함수를 이용해 3이라는 정수형 객체에 참조 개수가 몇 개 있는지 살펴보자.

```
>>> import sys
>>>
sys.getrefcount(3)
30
```

맨 처음 인터프리터를 실행한 후 sys.getrefcount(3)을 수행했을 때 참조 개수가 30이 나오는 이유는 파이썬 내부적으로 3이라는 자료형을 이미 사용했기 때문이다.

이후 a = 3, b = 3, c = 3과 같이 3을 가리키는 변수를 늘리면 참조 개수가 증가하는 것을 볼 수 있다.

```
>>> a = 3
>>>
sys.getrefcount(3)
31
>>> b = 3
>>>
sys.getrefcount(3)
32
>>> c = 3
>>>
sys.getrefcount(3)
33
```

변수를 만드는 여러 가지 방법

```
>>> a, b = ('python',
'life')
```

위의 예문처럼 튜플로 a, b에 값을 대입할 수 있다. 이 방법은 다음 예문과 완전히 동일하다.

```
>>> (a, b) = 'python',
'life'
```

튜플 부분에서도 언급했지만 튜플은 괄호를 생략해도 된다.

아래처럼 리스트로 변수를 만들 수도 있다.

```
>>> [a,b] = ['python',
'life']
```

또한 여러 개의 변수에 같은 값을 대입할 수도 있다.

```
>>> a = b =
'python'
```

파이썬에서는 위의 방법을 이용하여 두 변수의 값을 아주 간단히 바꿀 수 있다.

```
>>> a = 3
>>> b = 5
>>> a, b = b,
a
>>> a
5
>>> b
3
```

처음에 a에 3, b에는 5라는 값이 대입되어 있었지만 a, b = b, a라는 문장을 수행한 후에는 그 값이 서로 바뀌었음을 확인할 수 있다.

메모리에 생성된 변수 없애기

a=3을 입력하면 3이라는 정수형 객체가 메모리에 생성된다고 했다. 그렇다면 이 값을 메모리에서 없앨 수 있을까? 3이라는 객체를 가리키는 변수들의 개수를 레퍼런스 카운트라고 하였는데, 이 레퍼런스 카운트가 0이 되는 순간 3이라는 객체는 자동으로 사라진다. 즉, 3이라는 객체를 가리키고 있는 것이 하나도 없을 때 3이라는 객체는 메모리에서 사라지게 되는 것이다. 이것을 어려운 말로 가비지 콜렉션(Garbage collection, 쓰레기 수집)이라고도 한다.

다음은 특정한 객체를 가리키는 변수를 없애는 예이다.

```
>>> a = 3
>>> b = 3
>>>
del(a)
>>>
del(b)
```

위의 예를 살펴보면 변수 a와 b가 3이라는 객체를 가리켰다가 del이라는 파이썬 내장 함수에 의해서 사라진다. 따라서 레퍼런스 카운트가 0이 되어 정수형 객체 3도 메모리에서 사라지게 된다.

(※ 사용한 변수를 del 명령어를 이용하여 일일이 삭제할 필요는 없다. 파이썬이 이 모든 것을 자동으로 해주기 때문이다.)

리스트를 변수에 넣고 복사하고자 할 때

여기서는 리스트 자료형에서 가장 혼동하기 쉬운 "복사"에 대해 설명하려고 한다. 다음 예를 통해 알아보자.

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

앞의 예를 유심히 살펴보면 b라는 변수에 a가 가리키는 리스트를 대입하였다. 그런 다음 a 리스트의 a[1]을 4라는 값으로 바꾸면 a 리스트만 바뀌는 것이 아니라 b 리스트도 똑같이 바뀐다. 그 이유는 a, b 모두 같은 리스트인 [1, 2, 3]을 가리키고 있었기 때문이다. a, b는 이름만 다를 뿐이지 완전히 동일한 리스트를 가리키고 있는 변수이다.

그렇다면 b 변수를 생성할 때 a와 같은 값을 가지도록 복사해 넣으면서 a가 가리키는 리스트와는 다른 리스트를 가리키게 하는 방법은 없을까? 다음의 2가지 방법이 있다.

1. [:] 이용

첫 번째 방법으로는 아래와 같이 리스트 전체를 가리키는 [:]을 이용해서 복사하는 것이다.

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

위의 예에서 볼 수 있듯이 a 리스트 값을 바꾸더라도 b 리스트에는 영향을 끼치지 않는다.

2. copy 모듈 이용

두 번째는 copy 모듈을 이용하는 방법이다. 아래 예를 보면 from copy import copy라는 처음 보는 형태가 나오는데, 이것은 뒤에 설명할 파이썬 모듈 부분에서 자세히 다룬다. 여기서는 단순히 copy라는 함수를 쓰기 위해서 사용되는 것이라고만 알아두자.

```
>>> from copy import
copy
>>> b = copy(a)
```

위의 예에서 b = copy(a)는 b = a[:]과 동일하다.

두 변수가 같은 값을 가지면서 다른 객체를 제대로 생성했는지 확인하려면 다음과 같이 is 함수를 이용하면 된다.

이 함수는 서로 동일한 객체인지 아닌지에 대한 판단을 하여 참과 거짓을 리턴한다.

```
>>> b is
a
False
```

위의 예에서 b is a 가 False를 리턴하므로 b와 a가 서로 다른 객체임을 알 수 있다.

03-1 if문

다음과 같은 상상을 해보자.

> *"돈이 있으면 택시를 타고, 돈이 없으면 걸어 간다."*

우리 모두에게 언제든지 일어날 수 있는 상황 중 하나이다. 프로그래밍도 사람이 하는 것이므로 위의 문장처럼 주어진 조건을 판단한 후 그 상황에 맞게 처리해야 할 경우가 생긴다. 이렇듯 프로그래밍에서 조건을 판단하여 해당 조건에 맞는 상황을 수행하는 데 쓰이는 것이 바로 if 문이다.

위와 같은 상황을 파이썬에서는 다음과 같이 표현할 수 있다.

```
>>> money = 1
>>> if money:
...     print("택시를 타고 가
라")
... else:
...     print("걸어 가라")
...
택시를 타고 가라
```

money에 입력된 1은 참이다. 따라서 if문 다음의 문장이 수행되어 '택시를 타고 가라'가 출력된다.

if문의 기본 구조

다음은 if와 else를 이용한 조건문의 기본 구조이다.

```
if 조건문:
    수행할 문
장1
    수행할 문
장2
    ...
else:
    수행할 문장
A
    수행할 문장
B
    ...
```

조건문을 테스트해서 참이면 if문 바로 다음의 문장(if 블록)들을 수행하고, 조건문이 거짓이면 else문 다음 의 문장(else 블록)들을 수행하게 된다. 그러므로 else문은 if문 없이 독립적으로 사용할 수 없다.

들여쓰기

if문을 만들 때는 if 조건문: 바로 아래 문장부터 if문에 속하는 모든 문장에 들여쓰기(indentation)를 해주어야 한다. 다음과 같이 조건문이 참일 경우 "수행할 문장1"을 들여쓰기 했고 "수행할 문장2"와 "수행할 문장3"도 들여쓰기를 해주었다. 다른 프로그래밍 언어를 사용했던 사람들은 파이썬에서 "수행할 문장"들을 들여쓰기 하는 것을 무시하는 경우가 많으니 더 주의해야 한다.

```
if 조건문:
    수행할 문
장1
    수행할 문
장2
    수행할 문
장3
```

다음과 같이 작성하면 오류가 발생한다. "수행할 문장2"를 들여쓰기 하지 않았기 때문이다.

```
if 조건문:
    수행할 문
장1
수행할 문장2
    수행할 문
장3
```

다음 예제를 살펴보면 무슨 말인지 이해할 수 있을 것이다.

```
>>> money = 1
>>> if money:
...     print("택시를")
... print("타고")
File "<stdin>", line 3
print("타고")
^
SyntaxError: invalid
syntax
```

그렇다면 들여쓰기는 공백[Spacebar]으로 하는 것이 좋을까? 아니면 탭[Tab]으로 하는 것이 좋을까? 이에 대한 논란은 파이썬을 사용하는 사람들 사이에서 아직도 계속되고 있다. 탭으로 하자는 쪽과 공백으로 하자는 쪽 모두가 동의하는 내용은 단 하나, 2가지를 혼용해서 쓰지는 말자는 것이다. 공백으로 할 거면 항상 공백으로 통일하고, 탭으로 할 거면 항상 탭으로 통일해서 사용하자는 말이다. 탭이나 공백은 프로그램 소스에서 눈으로 보이는 것이 아니기 때문에 혼용해서 쓰면 에러의 원인이 되기도 한다. 주의하도록 하자.

(※ 요즘 파이썬 커뮤니티에서는 들여쓰기를 할 때 공백(Spacebar) 4개를 사용하는 것을 권장한다.)

[조건문 다음에 콜론(:)을 잊지 말자!]

if 조건문 뒤에는 반드시 콜론(:)이 붙는다. 어떤 특별한 의미가 있다기보다는 파이썬의 문법 구조이다. 왜 하필 콜론(:)인지 궁금하다면 파이썬을 만든 귀도에게 직접 물어보아야 할 것이다. 앞으로 배우게 될 while이나 for, def, class문에도 역시 문장의 끝에 콜론(:)이 항상 들어간다. 초보자들은 이 콜론(:)을 빠뜨리는 경우가 많으니 특히 주의하자.

파이썬이 다른 언어보다 보기 쉽고 소스 코드가 간결한 이유는 바로 콜론(:)을 사용하여 들여쓰기(indentation)를 하도록 만들었기 때문이다. 하지만 이는 숙련된 프로그래머들이 파이썬을 처음 접할 때 제일 혼란스러워하는 부분이기도 하다. 다른 언어에서는 if문을 { } 기호로 감싸지만 파이썬에서는 들여쓰기로 해결한다는 점을 기억하자.

조건문이란 무엇인가?

if 조건문에서 "조건문"이란 참과 거짓을 판단하는 문장을 말한다. 자료형의 참과 거짓에 대해서 몇 가지만 다시 살펴보면 다음과 같은 것들이 있다.

자료형	참	거짓
숫자	0이 아닌 숫자	0
문자열	"abc"	""
리스트	[1,2,3]	[]
터플	(1,2,3)	()
딕셔너리	{"a":"b"}	{}

따라서 이전에 살펴보았던 택시 예제에서 조건문은 money가 된다.

```
>>> money = 1
>>> if
money:
```

money는 1이기 때문에 참이 되어 if문 다음의 문장을 수행하게 된다.

비교연산자

조건이 참인지 거짓인지 판단할 때 자료형보다는 비교연산자(<, >, ==, !=, >=, <=)를 쓰는 경우가 훨씬 많다.

다음 표는 비교연산자를 잘 설명해 준다.

비교연산자	설명
x < y	x가 y보다 작다
x > y	x가 y보다 크다
x == y	x와 y가 같다
x != y	x와 y가 같지 않다
x >= y	x가 y보다 크거나 같다
x <= y	x가 y보다 작거나 같다

이제 위의 연산자들을 어떻게 사용하는지 알아보자.

```
>>> x =
3
>>> y =
2
>>> x >
y
True
>>>
```

x에 3을, y에 2를 대입한 다음에 `x > y` 라는 조건문을 수행하면 True를 리턴한다. `x > y` 라는 조건문이 참이기 때문이다.

```
>>> x <
y
False
```

위의 조건문은 거짓이기 때문에 False를 리턴한다.

```
>>> x ==
y
False
```

x와 y는 같지 않다. 따라서 위의 조건문은 거짓이다.

```
>>> x !=
y
True
```

x와 y는 같지 않다. 따라서 위의 조건문은 참이다.

앞에서 살펴본 택시 예제를 다음처럼 바꾸려면 어떻게 해야 할까?

> *"만약 3000원 이상의 돈을 가지고 있으면 택시를 타고 그렇지 않으면 걸어 가라"*

위의 상황은 다음처럼 프로그래밍할 수 있다.

```
>>> money = 2000
>>> if money >= 3000:
...     print("택시를 타고 가
라")
... else:
...     print("걸어가라")
...
걸어가라
>>>
```

```
money >=
3000
```
이라는 조건문이 거짓이 되기 때문에 else문 다음의 문장을 수행하게 된다.

and, or, not

조건을 판단하기 위해 사용하는 다른 연산자로는 and, or, not이 있다. 각각의 연산자는 다음처럼 동작한다.

연산자	설명
x or y	x와 y 둘중에 하나만 참이면 참이다
x and y	x와 y 모두 참이어야 참이다
not x	x가 거짓이면 참이다

다음의 예를 통해 or 연산자의 사용법을 알아보자.

> *"돈이 3000원 이상 있거나 카드가 있다면 택시를 타고 그렇지 않으면 걸어 가라"*

```
>>> money = 2000
>>> card = 1
>>> if money >= 3000 or card:
...     print("택시를 타고 가
라")
... else:
...     print("걸어가라")
...
택시를 타고 가라
>>>
```

`money >= 3000 or card` 라는 조건문이 참이 된다. 따라서 if문 다음의 "택시를 타고 가라"라는 문장이 수행된다.

money는 2000이지만 card가 1이기 때문에

x in s, x not in s

더 나아가 파이썬은 다른 프로그래밍 언어에서 쉽게 볼 수 없는 재미있는 조건문들을 제공한다. 바로 다음과 같은 것들이다.

in	not in
x in 리스트	x not in 리스트
x in 튜플	x not in 튜플
x in 문자열	x not in 문자열

in이라는 영어 단어의 뜻이 "~안에"라는 것을 생각해 보면 다음 예들이 쉽게 이해될 것이다.

```
>>> 1 in [1, 2, 3]
True
>>> 1 not in [1, 2,
3]
False
```

앞에서 첫 번째 예는 "[1, 2, 3]이라는 리스트 안에 1이 있는가?"라는 조건문이다. 1은 [1, 2, 3] 안에 있으므로 참이 되어 True를 리턴한다. 두 번째 예는 "[1, 2, 3]이라는 리스트 안에 1이 없는가?"라는 조건문이다. 1은 [1, 2, 3] 안에 있으므로 거짓이 되어 False를 리턴한다.

다음은 튜플과 문자열에 적용한 예이다. 각각의 결과가 나온 이유는 쉽게 유추할 수 있다.

```
>>> 'a' in ('a', 'b',
'c')
True
>>> 'j' not in 'python'
True
```

이번에는 우리가 계속 사용해 온 택시 예제에 in을 적용해 보자.

> *"만약 주머니에 돈이 있으면 택시를 타고, 없으면 걸어 가라"*

```
>>> pocket = ['paper', 'cellphone',
'money']
>>> if 'money' in pocket:
...     print("택시를 타고 가라")
... else:
...     print("걸어가라")
...
택시를 타고 가라
>>>
```

['paper', 'cellphone', 'money']라는 리스트 안에 'money'가 있으므로 'money' in pocket은 참이 된다. 따라서 if문 다음의 문장이 수행된다.

[조건문에서 아무 일도 하지 않게 설정하고 싶다면?]

가끔 조건문의 참, 거짓에 따라 실행할 행동을 정의할 때, 아무런 일도 하지 않도록 설정하고 싶을 때가 있다. 다음의 예를 보자.

> *"주머니에 돈이 있으면 가만히 있고 주머니에 돈이 없으면 카드를 꺼내라"*

이럴 때 사용하는 것이 바로 pass이다. 위의 예를 pass를 적용해서 구현해 보자.

```
>>> pocket = ['paper', 'money',
'cellphone']
>>> if 'money' in pocket:
...     pass
... else:
...     print("카드를 꺼내라")
...
```

pocket이라는 리스트 안에 money라는 문자열이 있기 때문에 if문 다음 문장인 pass가 수행되고 아무런 결과값도 보여 주지 않는다.

다양한 조건을 판단하는 elif

if와 else만으로는 다양한 조건을 판단하기 어렵다. 다음과 같은 예를 보더라도 if와 else만으로는 조건을 판단하는 데 어려움을 겪게 된다.

> *"주머니에 돈이 있으면 택시를 타고, 주머니에 돈은 없지만 카드가 있으면 택시를 타고, 돈도 없고 카드도 없으면 걸어 가라"*

위의 문장을 보면 조건을 판단하는 부분이 두 군데가 있다. 먼저 주머니에 돈이 있는지를 판단해야 하고 주머니에 돈이 없으면 다시 카드가 있는지 판단해야 한다.

if와 else만으로 위의 문장을 표현하려면 다음과 같이 할 수 있다.

```
>>> pocket = ['paper',
'handphone']
>>> card = 1
>>> if 'money' in pocket:
...     print("택시를 타고가라")
... else:
...     if card:
...         print("택시를 타고가라")
...     else:
...         print("걸어가라")
...
택시를 타고가라
>>>
```

언뜻 보기에도 이해하기 어렵고 산만한 느낌이 든다. 이런 복잡함을 해결하기 위해 파이썬에서는 다중 조건 판단을 가능하게 하는 elif라는 것을 사용한다.

위의 예를 elif를 사용하면 다음과 같이 바꿀 수 있다.

```
>>> pocket = ['paper',
'cellphone']
>>> card = 1
>>> if 'money' in pocket:
...     print("택시를 타고가라")
... elif card:
...     print("택시를 타고가라")
... else:
...     print("걸어가라")
...
택시를 타고가라
```

즉, elif는 이전 조건문이 거짓일 때 수행된다. if, elif, else를 모두 사용할 때 기본 구조는 다음과 같다.

```
If <조건문>:
    <수행할 문장1>
    <수행할 문장2>
    ...
elif <조건문>:
    <수행할 문장1>
    <수행할 문장2>
    ...
elif <조건문>:
    <수행할 문장1>
    <수행할 문장2>
    ...
...
else:
   <수행할 문장1>
   <수행할 문장2>
   ...
```

위에서 볼 수 있듯이 elif는 개수에 제한 없이 사용할 수 있다.

[if문을 한 줄로 작성하기]

앞의 pass를 사용한 예를 보면 if문 다음에 수행할 문장이 한 줄이고, else문 다음에 수행할 문장도 한 줄밖에 되지 않는다.

```
>>> if 'money' in pocket:
...     pass
... else:
...     print("카드를 꺼내
라")
...
```

이렇게 수행할 문장이 한 줄일 때 조금 더 간략하게 코드를 작성하는 방법이 있다.

```
>>> pocket = ['paper', 'money',
'cellphone']
>>> if 'money' in pocket: pass
... else: print("카드를 꺼내라")
...
```

if문 다음의 수행할 문장을 콜론(:) 뒤에 바로 적어 주었다. else문 역시 마찬가지이다. 때때로 이렇게 작성하 는 것이 보기 편할 수 있다.

03-2 while문

while문의 기본 구조

반복해서 문장을 수행해야 할 경우 while문을 사용한다. 그래서 while문을 반복문이라고도 부른다.

다음은 while문의 기본 구조이다.

```
while <조건문>:
    <수행할 문
장1>
    <수행할 문
장2>
    <수행할 문
장3>
    ...
```

while문은 조건문이 참인 동안에 while문 아래에 속하는 문장들이 반복해서 수행된다.

"열 번 찍어 안 넘어 가는 나무 없다" 라는 속담을 파이썬 프로그램으로 만든다면 다음과 같이 될 것이다.

```
>>> treeHit = 0
>>> while treeHit < 10:
...     treeHit = treeHit +1
...     print("나무를 %d번 찍었습니다." %
treeHit)
...     if treeHit == 10:
...         print("나무 넘어갑니다.")
...
나무를 1번 찍었습니다.
나무를 2번 찍었습니다.
나무를 3번 찍었습니다.
나무를 4번 찍었습니다.
나무를 5번 찍었습니다.
나무를 6번 찍었습니다.
나무를 7번 찍었습니다.
나무를 8번 찍었습니다.
나무를 9번 찍었습니다.
나무를 10번 찍었습니다.
나무 넘어갑니다.
```

위의 예에서 while문의 조건문은 treeHit < 10 이다. 즉, treeHit가 10보다 작은 동안에 while문 안의 문장들을 계속 수행한다. whlie문 안의 문장을 보면 제일 먼저 treeHit = treeHit + 1로 treeHit 값이 계속 1씩 증가한다. 그리고 나무를 treeHit번만큼 찍었음을 알리는 문장을 출력하고 treeHit가 10이 되면 "나무 넘어갑니다."라는 문장을 출력한다. 그러고 나면 treeHit < 10 이라는 조건문이 거짓이 되므로 while문을 빠져나가게 된다.

(※ treeHit = treeHit + 1 은 프로그래밍을 할 때 매우 자주 사용하는 기법이다. treeHit의 값을 1만큼씩 증가시킬 목적으로 사용되며, treeHit +=1 처럼 사용되기도 한다.)

다음은 while문이 반복되는 과정을 순서대로 정리한 표이다. 이렇게 긴 과정을 소스 코드 단 5줄로 만들 수 있다니 놀랍지 않은가?

treeHit	조건문	조건판단	수행하는 문장	while문
0	0 < 10	참	나무를 1번 찍었습니다.	반복
1	1 < 10	참	나무를 2번 찍었습니다.	반복
2	2 < 10	참	나무를 3번 찍었습니다.	반복
3	3 < 10	참	나무를 4번 찍었습니다.	반복
4	4 < 10	참	나무를 5번 찍었습니다.	반복
5	5 < 10	참	나무를 6번 찍었습니다.	반복
6	6 < 10	참	나무를 7번 찍었습니다.	반복
7	7 < 10	참	나무를 8번 찍었습니다.	반복
8	8 < 10	참	나무를 9번 찍었습니다.	반복
9	9 < 10	참	나무를 10번 찍었습니다. 나무 넘어갑니다.	반복
10	10 < 10	거짓		종료

while문 직접 만들기

다음을 직접 입력해 보자. 여러 가지 선택지 중 하나를 선택해서 입력받는 예제이다. 먼저 다음과 같이 여러 줄짜리 문자열을 만들어 보자.

```
>>> prompt = """
... 1. Add
... 2. Del
... 3. List
... 4. Quit
...
... Enter number:
"""
>>>
```

이어서 number라는 변수에 0을 먼저 대입한다. 이렇게 변수를 먼저 설정해 놓지 않으면 다음에 나올 while문의 조건문인 number != 4에서 변수가 존재하지 않는다는 에러가 발생한다.

```
>>> number = 0
>>> while number != 4:
...     print(prompt)
...     number =
int(input())
...

1. Add
2. Del
3. List
4. Quit

Enter number:
```

while문을 보면 number가 4가 아닌 동안 prompt를 출력하고 사용자로부터 번호를 입력받는다. 다음의 결과 화면

처럼 사용자가 4라는 값을 입력하지 않으면 계속해서 prompt를 출력한다.

(※ 여기서 number = int(input())는 사용자의 숫자 입력을 받아들이는 것이라고만 알아두자. int나 input 함수에 대한 내용은 뒤의 내장 함수 부분에서 자세하게 다룬다.)

```
Enter number:
1

1. Add
2. Del
3. List
4. Quit
```

4를 입력하면 조건문이 거짓이 되어 while문을 빠져나가게 된다.

```
Enter number:
4
>>>
```

while문 강제로 빠져나가기

while문은 조건문이 참인 동안 계속해서 while문 안의 내용을 반복적으로 수행한다. 하지만 강제로 while문을 빠져나가고 싶을 때가 있다. 예를 들어 커피 자판기를 생각해 보자. 자판기 안에 커피가 충분히 있을 때에는 동전을 넣으면 커피가 나온다. 그런데 자판기가 제대로 작동하려면 커피가 얼마나 남았는지 항상 검사해야 한다. 만약 커피가 떨어졌다면 판매를 중단하고 "판매 중지"라는 문구를 사용자에게 보여주어야 한다. 이렇게 판매를 강제로 멈추게 하는 것이 바로 break문이다.

다음의 예는 커피 자판기 이야기를 파이썬 프로그램으로 표현해 본 것이다.

```
>>> coffee = 10
>>> money = 300
>>> while money:
...     print("돈을 받았으니 커피를 줍니다.")
...     coffee = coffee -1
...     print("남은 커피의 양은 %d개입니다." % coffee)
...     if not coffee:
...         print("커피가 다 떨어졌습니다. 판매를 중지합니
다.")
...         break
...
```

money가 300으로 고정되어 있으므로 while money:에서 조건문인 money는 0이 아니기 때문에 항상 참이다. 따라서 무한히 반복되는 무한 루프를 돌게 된다. 그리고 while문의 내용을 한 번 수행할 때마다 coffee = coffee - 1에 의해서 coffee의 개수가 1개씩 줄어든다. 만약 coffee가 0이 되면 if not coffee:라는 문장에서 not coffee가 참이 되므로 if문 다음의 문장인 "커피가 다 떨어졌습니다. 판매를 중지합니다."가 수행되고 break문이 호출되어 while문을 빠져나가게 된다.

break문 이용해 자판기 작동 과정 만들기

하지만 실제 자판기는 위의 예처럼 작동하지는 않을 것이다. 다음은 자판기의 실제 작동 과정과 비슷하게 만들어 본 예이다. 이해가 안 되더라도 걱정하지 말자. 아래의 예는 조금 복잡하니까 대화형 인터프리터를 이용하지 말고 에디터를 이용해서 작성해 보자.

```
coffee = 10
while True:
    money = int(input("돈을 넣어 주세요: "))
    if money == 300:
        print("커피를 줍니다.")
        coffee = coffee -1
    elif money > 300:
        print("거스름돈 %d를 주고 커피를 줍니다." % (money
-300))
        coffee = coffee -1
    else:
        print("돈을 다시 돌려주고 커피를 주지 않습니다.")
        print("남은 커피의 양은 %d개 입니다." % coffee)
    if not coffee:
        print("커피가 다 떨어졌습니다. 판매를 중지 합니다.")
        break
```

위의 프로그램 소스를 따로 설명하지는 않겠다. 여러분이 소스를 입력하면서 무슨 내용인지 이해할 수 있다면 지금껏 배운 if문이나 while문을 이해했다고 보면 된다. 만약 money = int(input("돈을 넣어 주세요:"))라는 문장이 이해되지 않는다면 이 문장은 사용자로부터 입력을 받는 부분이고 입력받은 숫자를 money라는 변수에 대입하는 것이라고만 알아두자.

이제 coffee.py 파일을 저장한 후 프로그램을 직접 실행해 보자. 아래와 같은 입력란이 나타난다.

```
C:\Python>python
coffee.py
돈을 넣어 주세요:
```

입력란에 여러 숫자를 입력해 보면서 결과를 확인하자.

```
돈을 넣어 주세요: 500
거스름돈 200를 주고 커피를 줍니다.
돈을 넣어 주세요: 300
커피를 줍니다.
돈을 넣어 주세요: 100
돈을 다시 돌려주고 커피를 주지 않습니다.
남은 커피의 양은 8개입니다.
돈을 넣어 주세요:
```

[에디터에서 만든 프로그램을 실행할 때 오류가 발생한다면?]

만약 파이썬 2.7 버전을 사용한다면 자판기 예제의 소스 코드에서 input("돈을 넣어 주세요: ") 대신 raw_input("돈을 넣어 주세요: ")로 사용해야 하며 소스 코드 가장 첫 번째 줄에 다음과 같은 문장을 반드시 넣어 주어야만 한다.

파이썬 3 버전을 사용할 때도 아래와 같은 오류가 발생할 수 있다.

```
C:\Python>python coffee.py
File "coffee.py", line 9
SyntaxError: (unicode error) 'utf-8' codec can't decode byte 0xb9 in position
0:
invalid start byte
```

파이썬 3 버전에서 이런 오류가 발생했다면 아마 파일을 저장할 때 파일 인코딩 타입이 utf-8이 아니었을 것이다. 파일을 저장할 때 파일 인코딩을 utf-8로 맞추면 프로그램이 제대로 실행된다.

조건에 맞지 않는 경우 맨 처음으로 돌아가기

while문 안의 문장을 수행할 때 입력된 조건을 검사해서 조건에 맞지 않으면 while문을 빠져나간다. 그런데 프로그래밍을 하다 보면 while문을 빠져나가지 않고 while문의 맨 처음(조건문)으로 다시 돌아가게 만들고 싶은 경우가 생기게 된다. 이때 사용하는 것이 바로 continue문이다.

만약 1부터 10까지의 숫자 중에서 홀수만 출력하는 것을 while문을 이용해서 작성한다고 생각해 보자. 어떤 방법이 좋을까?

```
>>> a = 0
>>> while a < 10:
...     a = a+1
...     if a % 2 == 0:
continue
...     print(a)
...
1
3
5
7
9
```

위의 예는 1부터 10까지의 숫자 중 홀수만을 출력하는 예이다. a가 10보다 작은 동안 a는 1만큼씩 계속 증가한다. `if a % 2 == 0` (a를 2로 나누었을 때 나머지가 0인 경우)이 참이 되는 경우는 a가 짝수일 때이다. 즉, a가 짝수이면 continue 문장을 수행한다. 이 continue문은 while문의 맨 처음(조건문: `a<10`)으로 돌아가게 하는 명령어이다. 따라서 위의 예에서 a가 짝수이면 print(a)는 수행되지 않을 것이다.

무한 루프

이번에는 무한 루프(Loop)에 대해서 알아보자. 무한 루프란 무한히 반복한다는 의미이다. 우리가 사용하는 일반적인 프로그램 중에서 무한 루프의 개념을 사용하지 않는 프로그램은 거의 없다. 그만큼 자주 사용된다는 뜻이다.

파이썬에서 무한 루프는 while문으로 구현할 수 있다. 다음은 무한 루프의 기본적인 형태이다.

```
while True:
    수행할 문장1
    수행할 문장2
    ...
```

while문의 조건문이 True이므로 항상 참이 된다. 따라서 while문 안에 있는 문장들은 무한하게 수행될 것이다.

다음의 무한 루프 예이다.

```
>>> while True:
...     print("Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니
다.")
...
Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.
Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.
Ctrl+C를 눌러야 while문을 빠져나갈 수 있습니다.
....
```

위의 문장은 영원히 출력된다. 하지만 이 예처럼 아무 의미 없이 무한 루프를 돌리는 경우는 거의 없을 것이다. [Ctrl+C]를 눌러 빠져나가도록 하자.

03-3 for문

파이썬의 직관적인 특징을 가장 잘 대변해 주는 것이 바로 이 for문이다. while문과 비슷한 반복문인 for문은 매우 유용하고 문장 구조가 한눈에 쏙 들어온다는 장점이 있다. for문을 잘 사용하면 프로그래밍이 때때로 즐거워질 것이다.

for문의 기본 구조

for문의 기본적인 구조는 다음과 같다.

```
for 변수 in 리스트(또는 튜플, 문자열):
    수행할 문장1
    수행할 문장2
    ...
```

리스트나 튜플, 문자열의 첫 번째 요소부터 마지막 요소까지 차례로 변수에 대입되어 "수행할 문장1", "수행할 문장2" 등이 수행된다.

예제 이용해 for문 이해하기

for문은 예제를 통해서 살펴보는 것이 가장 알기 쉽다. 다음 예제를 직접 입력해 보자.

1. 전형적인 for문

```
>>> test_list = ['one', 'two', 'three']
>>> for i in test_list:
...     print(i)
...
one
two
three
```

['one', 'two', 'three']라는 리스트의 첫 번째 요소인 'one'이 먼저 i 변수에 대입된 후 print(i)라는 문장을 수행한다. 다음에 'two'라는 두 번째 요소가 i 변수에 대입된 후 print(i) 문장을 수행하고 리스트의 마지막 요소까지 이것을 반복한다.

2. 다양한 for문의 사용

```
>>> a = [(1,2), (3,4),
(5,6)]
>>> for (first, last) in a:
...     print(first + last)
...
3
7
11
```

위의 예는 a 리스트의 요소값이 튜플이기 때문에 각각의 요소들이 자동으로 (first, last)라는 변수에 대입된다.

(※ 이 예는 02장에서 살펴보았던 튜플을 이용한 변수값 대입 방법과 매우 비슷한 경우이다.

```
>>> (first, last) = (1,
2)                          )
```

3. for문의 응용

for문의 쓰임새를 알기 위해 다음을 가정해 보자.

> *"총 5명의 학생이 시험을 보았는데 시험 점수가 60점이 넘으면 합격이고 그렇지 않으면 불합격이다. 합격인지 불합격인지 결과를 보여주시오."*

우선 학생 5명의 시험 점수를 리스트로 표현해 보았다.

```
marks = [90, 25, 67, 45,
80]
```

1번 학생은 90점이고 5번 학생은 80점이다.

이런 점수를 차례로 검사해서 합격했는지 불합격했는지 통보해 주는 프로그램을 만들어 보자. 역시 에디터로 작성한다.

```
marks = [90, 25, 67, 45, 80]

number = 0
for mark in marks:
    number = number +1
    if mark >= 60:
        print("%d번 학생은 합격입니다." % number)
    else:
        print("%d번 학생은 불합격입니다." %
number)
```

각각의 학생에게 번호를 붙여 주기 위해 number라는 변수를 이용하였다. 점수 리스트인 marks에서 차례로 점수를 꺼내어 mark라는 변수에 대입하고 for문 안의 문장들을 수행하게 된다. 우선 for문이 한 번씩 수행될 때마다 number는 1씩 증가한다.

이 프로그램을 실행하면 mark가 60 이상일 때 합격 메시지를 출력하고 60을 넘지 않을 때 불합격 메시지를 출력한다.

```
C:\Python>python
marks1.py
1번 학생은 합격입니다.
2번 학생은 불합격입니다.
3번 학생은 합격입니다.
4번 학생은 불합격입니다.
5번 학생은 합격입니다.
```

for문과 continue

while문에서 살펴보았던 continue를 for문에서도 사용할 수 있다. 즉, for문 안의 문장을 수행하는 도중에 continue문을 만나면 for문의 처음으로 돌아가게 된다.

앞서 for문 응용 예제를 그대로 이용해서 60점 이상인 사람에게는 축하 메시지를 보내고 나머지 사람에게는 아무런 메시지도 전하지 않는 프로그램을 에디터를 이용해 작성해 보자.

```
marks = [90, 25, 67, 45, 80]

number = 0
for mark in marks:
    number = number +1
    if mark < 60: continue
    print("%d번 학생 축하합니다. 합격입니다. " %
number)
```

점수가 60점 이하인 학생일 경우에는 mark < 60이 참이 되어 continue문이 수행된다. 따라서 축하 메시지를 출력하는 부분인 print문을 수행하지 않고 for문의 처음으로 돌아가게 된다.

```
C:\Python>python marks2.py
1번 학생 축하합니다. 합격입니다.
3번 학생 축하합니다. 합격입니다.
5번 학생 축하합니다. 합격입니다.
```

for와 함께 자주 사용하는 range함수

for문은 숫자 리스트를 자동으로 만들어 주는 range라는 함수와 함께 사용되는 경우가 많다. 다음은 range 함수의 간단한 사용법이다.

```
>>> a =
range(10)
>>> a
range(0, 10)
```

range(10)은 0부터 10 미만의 숫자를 포함하는 range 객체를 만들어 준다.

시작 숫자와 끝 숫자를 지정하려면 range(시작 숫자, 끝 숫자) 형태를 사용하는데, 이때 끝 숫자는 포함되지 않는다.

```
>>> a = range(1,
11)
>>> a
range(1, 11)
```

range 함수의 예시 살펴보기

for와 range 함수를 이용하면 1부터 10까지 더하는 것을 다음과 같이 쉽게 구현할 수 있다.

```
>>> sum = 0
>>> for i in range(1, 11):
...     sum = sum + i
...
>>> print(sum)
55
```

range(1, 11)은 숫자 1부터 10까지(1 이상 11 미만)의 숫자를 데이터로 갖는 객체이다. 따라서 위의 예에서 i 변수에 리스트의 숫자들이 1부터 10까지 하나씩 차례로 대입되면서 sum = sum + i라는 문장을 반복적으로 수행하고 sum은 최종적으로 55가 된다.

또한 우리가 앞서 살펴보았던 60점 이상이면 합격이라는 문장을 출력하는 예제도 range 함수를 이용해서 바꿀 수 있다. 다음을 보자.

```
marks = [90, 25, 67, 45, 80]
for number in range(len(marks)):
    if marks[number] < 60: continue
    print("%d번 학생 축하합니다. 합격입니다." %
(number+1))
```

여기서 len이라는 함수가 처음 나왔다. len 함수는 리스트 내 요소의 개수를 돌려주는 함수이다. 따라서 len(marks)는 5가 될 것이고 range(len(marks))는 range(5)가 될 것이다. number 변수에는 차례로 0부터 4까지의 숫자가 대입될 것이고, marks[number]는 차례대로 90, 25, 67, 45, 80이라는 값을 갖게 된다. 결과는 marks2.py 예제와 동일하다.

for와 range를 이용한 구구단

for와 range 함수를 이용하면 소스 코드 단 4줄만으로 구구단을 출력할 수 있다. 들여쓰기에 주의하며 입력해 보자.

```
>>> for i in range(2,10):
...     for j in range(1, 10):
...         print(i*j, end="
")
...     print('')
...
2 4 6 8 10 12 14 16 18
3 6 9 12 15 18 21 24 27
4 8 12 16 20 24 28 32 36
5 10 15 20 25 30 35 40 45
6 12 18 24 30 36 42 48 54
7 14 21 28 35 42 49 56 63
8 16 24 32 40 48 56 64 72
9 18 27 36 45 54 63 72 81
```

위의 예를 보면 for문이 두 번 사용되었다. ①번 for문에서 2부터 9까지의 숫자(range(2, 10))가 차례로 i에 대입된다. i가 처음 2일 때 ②번 for문을 만나게 된다. ②번 for문에서 1부터 9까지의 숫자(range(1, 10))가 j에 대입되고 그 다음 문장인 `print(i*j)`를 수행한다.

따라서 i가 2일 때 `2*1, 2*2, 2*3, … 2*9` 까지 차례로 수행되며 그 값을 출력하게 된다. 그 다음으로 i가 3일 때 역시 2일 때와 마찬가지로 수행될 것이고 i가 9일 때까지 계속 반복된다.

[입력 인수 end를 넣어 준 이유는 무엇일까?]

앞의 예제에서 `print(i*j, end="")` 와 같이 입력 인수 end를 넣어 준 이유는 해당 결과값을 출력할 때 다음 줄로 넘기지 않고 그 줄에 계속해서 출력하기 위해서이다. 그 다음에 이어지는 print(' ')는 2단, 3단 등을 구분하기 위해 두 번째 for문이 끝나면 결과값을 다음 줄부터 출력하게 해주는 문장이다.

만약 파이썬 2.7 버전을 사용한다면 `print(i*j, end="")` 대신 `print i*j,`와 같이 콤마(,)를 마지막에 넣

어야 한다.

리스트 안에 for문 포함하기

리스트 안에 for문을 포함하는 리스트 내포(List comprehension)를 이용하면 좀 더 편리하고 직관적인 프로그램을 만들 수 있다. 다음의 예제를 보자.

```
>>> a = [1,2,3,4]
>>> result = []
>>> for num in a:
...
result.append(num*3)
...
>>> print(result)
[3, 6, 9, 12]
```

위 예제는 a라는 리스트의 각 항목에 3을 곱한 결과를 result라는 리스트에 담는 예제이다.

이것을 리스트 내포를 이용하면 아래와 같이 간단히 해결할 수 있다.

```
>>> result = [num * 3 for num in
a]
>>> print(result)
[3, 6, 9, 12]
```

만약 짝수에만 3을 곱하여 담고 싶다면 다음과 같이 "if 조건"을 사용할 수 있다.

```
>>> result = [num * 3 for num in a if num % 2 ==
0]
>>> print(result)
[6, 12]
```

리스트 내포의 일반적인 문법은 다음과 같다. "if 조건" 부분은 앞의 예제에서 볼 수 있듯이 생략할 수 있다.

```
[표현식 for 항목 in 반복가능객체 if 조건]
```

조금 복잡하지만 for문을 2개 이상 사용하는 것도 가능하다. for문을 여러 개 사용할 때의 문법은 다음과 같다.

```
[표현식 for 항목1 in 반복가능객체1 if 조건1
        for 항목2 in 반복가능객체2 if 조건2
        ...
        for 항목n in 반복가능객체n if 조건
n]
```

만약 구구단의 모든 결과를 리스트에 담고 싶다면 리스트 내포를 이용하여 아래와 같이 간단하게 구현할 수도 있다.

```
>>> result = [x*y for x in range(2,10)
...               for y in range(1,10)]
>>> print(result)
[2, 4, 6, 8, 10, 12, 14, 16, 18, 3, 6, 9, 12, 15, 18, 21, 24, 27, 4, 8, 12,
16,
20, 24, 28, 32, 36, 5, 10, 15, 20, 25, 30, 35, 40, 45, 6, 12, 18, 24, 30, 36,
42
, 48, 54, 7, 14, 21, 28, 35, 42, 49, 56, 63, 8, 16, 24, 32, 40, 48, 56, 64,
72,
9, 18, 27, 36, 45, 54, 63, 72, 81]
```

지금껏 우리는 프로그램의 흐름을 제어하는 if문, while문, for문에 대해서 알아보았다. 아마도 여러분은 while문과 for문을 보면서 2가지가 아주 비슷하다는 느낌을 받았을 것이다. 실제로 for문을 사용한 부분을 while문으로 바꿀 수 있는 경우도 많고, while문을 for문으로 바꾸어서 사용할 수 있는 경우도 많다.