Think Python

- Korean ver.2 -

㈜ 비트시스 [대표 김동칠]

동의대학교 블록체인 인력양성 사업단

부산광역시
BUSAN METROPOLITAN CITY

BIT 부산인재평생교육진흥원
Busan Institute for Talent & Lifelong Education

BitSys
행복한 꿈을 공유하다

제 5 장

반복(Iteration)

제 1 절 변수 갱신

대입문의 흔한 패턴은 변수를 갱신하는 대입문이다. 변수의 새로운 값은 이전 값에 의존한다.

```
x = x+1
```

상기 예제는 "현재 값 x에 1을 더해서 x를 새로운 값으로 갱신한다."

만약 존재하지 않는 변수를 갱신하면, 오류가 발생한다. 왜냐하면 x에 값을 대입하기 전에 파이썬이 오른쪽을 먼저 평가하기 때문이다.

```
>>> x = x+1
NameError: name 'x' is not defined
```

변수를 갱신하기 전에 간단한 변수 대입으로 통상 먼저 초기화(initialize)한다.

```
>>> x = 0
>>> x = x+1
```

1을 더해서 변수를 갱신하는 것을 증가(increment)라고 하고, 1을 빼서 변수를 갱신하는 것을 감소(decrement)라고 한다.

제 2 절 while문

종종 반복적인 작업을 자동화하기 위해서 컴퓨터를 사용한다. 오류 없이 동일하거나 비슷한 작업을 반복하는 일은 컴퓨터가 사람보다 잘한다. 반복이 매우 흔한 일이어서, 파이썬에서 반복 작업을 쉽게 하도록 몇가지 언어적 기능을 제공한다.

파이썬에서 반복의 한 형태가 while문이다. 다음은 5 에서부터 거꾸로 세어서 마지막에 "Blastoff(발사)!"를 출력하는 간단한 프로그램이다.

```
n = 5
while n > 0:
    print n
    n = n-1
print 'Blastoff(발사)!'
```

마치 영어를 읽듯이 while을 읽어 내려갈 수 있다. n이 0 보다 큰 동안에 n의 값을 출력하고 n 값에서 1만큼 뺀다. 0 에 도달했을 때, while문을 빠져나가 Blastoff(발사)!"를 화면에 출력한다.

좀더 형식을 갖춰 정리하면, 다음이 while문에 대한 실행 흐름에 대한 정리다.

1. 조건을 평가해서 참(True) 혹은 거짓(False)를 산출한다.
2. 만약 조건이 거짓이면, while문을 빠져나가 다음 문장을 계속 실행한다.
3. 만약 조건이 참이면, 몸통 부문의 문장을 실행하고 다시 처음 1번 단계로 돌아간다.

3번째 단계에서 처음으로 다시 돌아가는 반복을 하기 때문에 이런 종류의 흐름을 루프(loop)이라고 부른다. 매번 루프 몸통 부문을 실행할 때마다, 이것을 반복(iteration)이라고 한다. 상기 루프에 대해서 "5번 반복했다"고 말한다. 즉, 루프 몸통 부문이 5번 수행되었다는 의미가 된다.

루프 몸통 부문은 필히 하나 혹은 그 이상의 변수값을 바꾸어서 종국에는 조건식이 거짓(false)이 되어 루프가 종료되게 만들어야 한다. 매번 루프가 실행될 때마다 상태를 변경하고 언제 루프가 끝날지 제어하는 변수를 반복 변수(iteration variable)라고 한다. 만약 반복 변수가 없다면, 루프는 영원히 반복될 것이고, 결국 무한 루프(infinite loop)에 빠질 것이다.

제 3 절 무한 루프

프로그래머에게 무한한 즐거움의 원천은 아마도 "거품내고, 헹구고, 반복" 이렇게 적혀있는 샴프 사용법 문구가 무한루프라는 것을 알아차릴 때일 것이다. 왜냐하면, 얼마나 많이 루프를 실행해야 하는지 말해주는 반복 변수(iteration variable)가 없어서 무한 반복하기 때문입니다.

숫자를 꺼꾸로 세는 (countdown) 예제는 루프가 끝나는 것을 증명할 수 있다. 왜냐하면 n값이 유한하고, n이 매번 루프를 돌 때마다 작아져서 결국 0 에 도달할 것이기 때문이다. 다른 경우 반복 변수가 전혀 없어서 루프가 명백하게 무한 반복한다.

제 4 절 무한 반복과 break

가끔 몸통 부문을 절반 진행할 때까지 루프를 종료해야하는 시점인지 확신하지 못한다. 이런 경우 의도적으로 무한 루프를 작성하고 break 문을 사용하여 루프를 빠져 나온다.

다음 루프는 명백하게 무한 루프(infinite loop)가 되는데 이유는 while문 논리 표현식이 단순히 논리 상수 참(True)으로 되어 있기 때문이다.

```
n = 10
while True:
    print n,
    n = n - 1
print 'Done!'
```

실수하여 상기 프로그램을 실행한다면, 폭주하는 파이썬 프로세스를 어떻게 멈추는지 빨리 배우거나, 컴퓨터의 전원 버튼이 어디에 있는지 찾아야 할 것이다. 표현식 상수 값이 참(True)이라는 사실로 루프 상단 논리 연산식이 항상 참 값이여서 프로그램이 영원히 혹은 배터리가 모두 소진될 때까지 실행된다.

이것이 역기능 무한 루프라는 것은 사실이지만, 유용한 루프를 작성하기 위해는 이 패턴을 여전히 이용할 것이다. 이를 위해서 루프 몸통 부문에 break문을 사용하여 루프를 빠져나가는 조건에 도달했을 때, 루프를 명시적으로 빠져나갈 수 있도록 주의깊게 코드를 추가해야 한다.

예를 들어, 사용자가 done을 입력하기 전까지 사용자로부터 입력값을 받는다고 가정해서 프로그램 코드를 다음과 같이 작성한다.

```
while True:
    line = raw_input('> ')
    if line == 'done':
        break
    print line
print 'Done!'
```

루프 조건이 항상 참(True)이여서 break문이 호출될 때까지 루프는 반복적으로 실행된다.

매번 프로그램이 꺾쇠 괄호로 사용자에게 명령문을 받을 준비를 한다. 사용자가 done을 타이핑하면, break문이 실행되어 루프를 빠져나온다. 그렇지 않은 경우 프로그램은 사용자가 무엇을 입력하든 메아리처럼 입력한 것을 그대로 출력하고 다시 루프 처음으로 되돌아 간다. 다음 예제로 실행한 결과가 있다.

```
> hello there
hello there
> finished
finished
> done
Done!
```

while 루프를 이와 같은 방식으로 작성하는 것이 흔한데 프로그램 상단에서 뿐만 아니라 루프 어디에서나 조건을 확인할 수 있고 피동적으로 "이벤트가 발생할 때까지 계속 실행" 대신에, 적극적으로 "이벤트가 생겼을 때 중지"로 멈춤 조건을 표현할 수 있다.

제 5 절 continue로 반복 종료

때때로 루프를 반복하는 중간에서 현재 반복을 끝내고, 다음 반복으로 즉시 점프하여 이동하고 싶을 때가 있다. 현재 반복 루프 몸통 부분 전체를 끝내지 않고 다음 반복으로 건너뛰기 위해서 continue문을 사용한다.

사용자가 "done"을 입력할 때까지 입력값을 그대로 복사하여 출력하는 루프 예제가 있다. 하지만 파이썬 주석문처럼 해쉬(#)로 시작하는 줄은 출력하지 않는다.

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print line
print 'Done!'
```

continue문이 추가된 새로운 프로그램을 샘플로 실행했다.

```
> hello there
hello there
> # don't print this
> print this!
print this!
> done
Done!
```

해쉬 기호(#)로 시작하는 줄을 제외하고 모든 줄을 출력한다. 왜냐하면, continue문이 실행될 때,현재 반복을 종료하고 while문 처음으로 돌아가서 다음 반복을 실행하게 되어서 print문을 건너뛴다.

제 6 절 for문을 사용한 명확한 루프

때때로, 단어 리스트나, 파일의 줄, 숫자 리스트 같은 사물의 집합에 대해 루프를 반복할 때가 있다. 루프를 반복할 사물 리스트가 있을 때, for문을 사용해서 *확정 루프(definite loop)*를 구성한다.

while문을 *불확정 루프(indefinite loop)*라고 하는데, 왜냐하면 어떤 조건이 거짓(False)가 될 때까지 루프가 단순히 계혹해서 돌기 때문이다. 하지만, for루프는 확정된 항목의 집합에 대해서 루프가 돌게 되어서 집합에 있는 항목만큼만 실행이 된다.

for문이 있고, 루프 몸통 부문으로 구성된다는 점에서 for루프 구문은 while 루프 구문과 비슷하다.

```
friends = ['Joseph', 'Glenn', 'Sally']
for friend in friends:
    print 'Happy New Year:', friend
print 'Done!'
```

파이썬 용어로, 변수 friends는 3개의 문자열을 가지는 리스트고, for 루프는 리스트를 하나씩 하나씩 찾아서 리스트에 있는 3개 문자열 각각에 대해 출력을 실행하여 다음 결과를 얻게 된다.

```
Happy New Year: Joseph
Happy New Year: Glenn
Happy New Year: Sally
Done!
```

for 루프를 영어로 번역하는 것이 while문을 번역하는 것과 같이 직접적이지는 않다. 하지만, 만약 friends를 집합(set)으로 생각한다면 다음과 같다. friends라고 명명된 집합에서 friend 각각에 대해서 한번씩 for 루프 몸통 부문에 있는 문장을 실행하라.

for 루프를 살펴보면, for와 in은 파이썬 예약어이고 friend와 friends는 변수이다.

```
for friend in friends:
    print 'Happy New Year', friend
```

특히, friend는 for 루프의 반복 변수(iteration variable)다. 변수 friend는 루프가 매번 반복할 때마다 변하고, 언제 for 루프가 완료되는지 제어한다. 반복 변수는 friend 변수에 저장된 3개 문자열을 순차적으로 훑고 간다.

제 7 절 루프 패턴

종종 for문과 while문을 사용하여, 리스트 항목, 파일 콘텐츠를 훑어 자료에 있는 가장 큰 값이나 작은 값 같은 것을 찾는다.

for나 while 루프는 일반적으로 다음과 같이 구축된다.

- 루프가 시작하기 전에 하나 혹은 그 이상의 변수를 초기화
- 루프 몸통부분에 각 항목에 대해 연산을 수행하고, 루프 몸통 부분의 변수 상태를 변경
- 루프가 완료되면 결과 변수의 상태 확인

루프 패턴의 개념과 작성을 시연하기 위해서 숫자 리스트를 사용한다.

7.1 계수(counting)와 합산 루프

예를 들어, 리스트의 항목을 세기 위해서 다음과 같이 for 루프를 작성한다.

```
count = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    count = count + 1
print 'Count: ', count
```

루프가 시작하기 전에 변수 count를 0 으로 설정하고, 숫자 목록을 훑어 갈 for 루프를 작성한다. 반복(iteration) 변수는 itervar라고 하고, 루프에서 itervar 을 사용되지 않지만, itervar는 루프를 제어하고 루프 몸통 부문 리스트의 각 값에 대해서 한번만 실행되게 한다.

루프 몸통 부문에 리스트의 각 값에 대해서 변수 count 값에 1을 더한다. 루프가 실행될 때, count 값은 "지금까지" 살펴본 값의 횟수가 된다.

루프가 종료되면, count 값은 총 항목 숫자가 된다. 총 숫자는 루프 맨마지막에 얻어진다. 루프를 구성해서, 루프가 끝났을 때 기대했던 바를 얻었다.

숫자 집합의 갯수를 세는 또 다른 비슷한 루프는 다음과 같다.

```
total = 0
for itervar in [3, 41, 12, 9, 74, 15]:
    total = total + itervar
print 'Total: ', total
```

상기 루프에서, 반복 변수(iteration variable)가 사용되었다. 앞선 루프에서처럼 변수 count에 1을 단순히 더하는 대신에, 각 루프가 반복을 수행하는 동안 실제 숫자 (3, 41, 12, 등)를 작업중인 합계에 덧셈을 했다. 변수 total을 생각해보면, total은 "지금까지 값의 총계다." 루프가 시작하기 전에 total은 어떤 값도 살펴본 적이 없어서 0 이다. 루프가 도는 중에는 total은 작업중인 총계가 된다. 루프의 마지막 단계에서 total은 리스트에 있는 모든 값의 총계가 된다.

루프가 실행됨에 따라, total은 각 요소의 합계로 누적된다. 이 방식으로 사용되는 변수를 누산기(accumulator)라고 한다.

계수(counting) 루프나 합산 루프는 특히 실무에서 유용하지는 않다. 왜냐하면 리스트에서 항목의 개수와 총계를 계산하는 len()과 sum()가 각각 내장 함수로 있기 때문이다.

7.2 최대값과 최소값 루프

리스트나 열(sequence)에서 가장 큰 값을 찾기 위해서, 다음과 같이 루프를 작성한다.

```
largest = None
print 'Before:', largest
for itervar in [3, 41, 12, 9, 74, 15]:
    if largest is None or itervar > largest :
        largest = itervar
    print 'Loop:', itervar, largest
print 'Largest:', largest
```

프로그램을 실행하면, 출력은 다음과 같다.

```
Before: None
Loop: 3 3
Loop: 41 41
Loop: 12 41
```

```
Loop: 9 41
Loop: 74 74
Loop: 15 74
Largest: 74
```

변수 largest는 "지금까지 본 가장 큰 수"로 생각할 수 있다. 루프 시작 전에 largest 값은 상수 None이다. None은 "빈(empty)" 변수를 표기하기 위해서 변수에 저장하는 특별한 상수 값이다.

루프 시작 전에 지금까지 본 가장 큰 수는 None이다. 왜냐하면 아직 어떤 값도 보지 않았기 때문이다. 루프가 실행되는 동안에, largest 값이 None이면, 첫 번째 본 값이 지금까지 본 가장 큰 값이 된다. 첫번째 반복에서 itervar는 3 이 되는데 largest 값이 None이여서 즉시, largest값을 3 으로 갱신한다.

첫번째 반복 후에 largest는 더 이상 None이 아니다. itervar > largest인지를 확인하는 복합 논리 표현식의 두 번째 부분은 "지금까지 본" 값 보다 더 큰 값을 찾게 될 때 자동으로 동작한다. "심지어 더 큰" 값을 찾게 되면 변수 largest에 새로운 값으로 대체한다. largest가 3에서 41, 41에서 74로 변경되어 출력되어 나가는 것을 확인할 수 있다.

루프의 끝에서 모든 값을 훑어서 변수 largest는 리스트의 가장 큰 값을 담고 있다.

최소값을 계산하기 위해서는 코드가 매우 유사하지만 작은 변화가 있다.

```
smallest = None
print 'Before:', smallest
for itervar in [3, 41, 12, 9, 74, 15]:
    if smallest is None or itervar < smallest:
        smallest = itervar
    print 'Loop:', itervar, smallest
print 'Smallest:', smallest
```

변수 smallest는 루프 실행 전에, 중에, 완료 후에 "지금까지 본 가장 작은" 값이 된다. 루프 실행이 완료되면, smallest는 리스트의 최소 값을 담게 된다.

계수(counting)과 합산에서와 마찬가지로 파이썬 내장함수 max()와 min()은 이런 루프문 작성을 불필요하게 만든다.

다음은 파이썬 내장 min() 함수의 간략 버전이다.

```
def min(values):
    smallest = None
    for value in values:
        if smallest is None or value < smallest:
            smallest = value
    return smallest
```

가장 적은 코드로 작성한 함수 버전은 파이썬에 이미 내장된 min 함수와 동등하게 만들기 위해서 모든 print문을 삭제했다.

제 8 절 디버깅

좀더 큰 프로그램을 작성할 때, 좀더 많은 시간을 디버깅에 보내는 자신을 발견할 것이다. 좀더 많은 코드는 버그가 숨을 수 있는 좀더 많은 장소와 오류가 발생할 기회가 있다는 것을 의미한다.

디버깅 시간을 줄이는 한 방법은 " 이분법에 따라 디버깅(debugging by bisection)" 하는 것이다. 예를 들어, 프로그램에 100 줄이 있고 한번에 하나씩 확인한다면, 100 번 단계가 필요하다.

대신에 문제를 반으로 나눈다. 프로그램 정확히 중간이나, 중간부분에서 점검한다. `print`문이나, 검증 효과를 갖는 상응하는 대용물을 넣고 프로그램을 실행한다.

중간지점 점검 결과 잘못 되었다면 문제는 양분한 프로그램 앞부분에 틀림없이 있다. 만약 정확하다면, 문제는 프로그램 뒷부분에 있다.

이와 같은 방식으로 점검하게 되면, 검토 해야하는 코드의 줄수를 절반으로 계속 줄일 수 있다. 단계가 100 번 걸리는 것에 비해 6번 단계 후에 이론적으로 1 혹은 2 줄로 문제 코드의 범위를 좁힐 수 있다.

실무에서, "프로그램의 중간"이 무엇인지는 명확하지 않고, 확인하는 것도 가능하지 않다. 프로그램 코드 라인을 세서 정확히 가운데를 찾는 것은 의미가 없다. 대신에 프로그램 오류가 생길 수 있는 곳과 오류를 확인하기 쉬운 장소를 생각하세요. 점검 지점 앞뒤로 버그가 있을 곳과 동일하게 생각하는 곳을 중간 지점으로 고르세요.

제 9 절 용어정의

누산기(accumulator): 더하거나 결과를 누적하기 위해 루프에서 사용되는 변수

계수(counter): 루프에서 어떤 것이 일어나는 횟수를 기록하는데 사용되는 변수. 카운터를 0 으로 초기화하고, 어떤 것의 "횟수"를 셀 때 카운터를 증가시킨다.

감소(decrement): 변수 값을 감소하여 갱신

초기화(initialize): 갱신될 변수의 값을 초기 값으로 대입

증가(increment): 변수 값을 증가시켜 갱신 (통상 1씩)

무한 루프(infinite loop): 종료 조건이 결코 만족되지 않거나 종료 조건이 없는 루프

반복(iteration): 재귀함수 호출이나 루프를 사용하여 명령문을 반복 실행

제 10 절 연습문제

Exercise 5.1 사용자가 “done”을 입력할 때까지 반복적으로 숫자를 읽는 프로그램을 작성하세요. “done”이 입력되면, 총계, 갯수, 평균을 출력하세요. 만약 숫자가 아닌 다른 것을 입력하게되면, `try`와 `except`를 사용하여 사용자 실수를 탐지해서 오류 메시지를 출력하고 다음 숫자로 건너 뛰게 하세요.

```
Enter a number: 4
Enter a number: 5
Enter a number: bad data
Invalid input
Enter a number: 7
Enter a number: done
16 3 5.33333333333
```

Exercise 5.2 위에서처럼 숫자 목록을 사용자로부터 입력받는 프로그램을 작성하세요. 평균값 대신에 숫자 목록 최대값과 최소값을 출력하세요.

제 6 장

문자열

제 1 절 문자열은 순서(sequence)다.

문자열은 여러 문자들의 순서다. 꺾쇠 연산자로 한번에 하나씩 문자에 접근한다.

```
>>> fruit = 'banana'
>>> letter = fruit[1]
```

두 번째 문장은 변수 `fruit`에서 1번 위치 문자를 추출하여 변수 `letter`에 대입한다. 꺾쇠 표현식을 인덱스(index)라고 부른다. 인덱스는 순서(sequence)에서 사용자가 어떤 문자를 원하는지 표시한다.

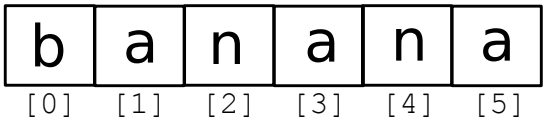
하지만, 여러분이 기대한 것은 아니다.

```
>>> print letter
a
```

대부분의 사람에게 `'banana'`의 첫 분자는 `a`가 아니라 `b`다. 하지만, 파이썬 인텍스는 문자열 처음부터 오프셋(offset)[1] 이다. 첫 글자 오프셋은 0 이다.

```
>>> letter = fruit[0]
>>> print letter
b
```

그래서, `b`가 `'banana'`의 0 번째 문자가 되고 `a`가 첫번째, `n`이 두번째 문자가 된다.



인덱스로 문자와 연산자를 포함하는 어떤 표현식도 사용가능지만, 인덱스 값은 정수여야만 한다. 정수가 아닌 경우 다음과 같은 결과를 얻게 된다.

---

[1] 컴퓨터에서 어떤 주소로부터 간격을 두고 떨어진 주소와의 거리. 기억 장치가 페이지 혹은 세그먼트 단위로 나누어져 있을 때 하나의 시작 주소로부터 오프셋만큼 떨어진 위치를 나타내는 것이다. [네이버 지식백과] 오프셋 [offset] (IT용어사전, 한국정보통신기술협회)

```
>>> letter = fruit[1.5]
TypeError: string indices must be integers
```

제 2 절 len함수 사용 문자열 길이 구하기

len 함수는 문자열의 문자 갯수를 반환하는 내장함수다.

```
>>> fruit = 'banana'
>>> len(fruit)
6
```

문자열의 가장 마지막 문자를 얻기 위해서, 아래와 같이 시도하려 싶을 것이다.

```
>>> length = len(fruit)
>>> last = fruit[length]
IndexError: string index out of range
```

인덱스 오류 (IndexError) 이유는 'banana' 에 6번 인텍스 문자가 없기 때문이다. 0 에서부터 시작했기 때문에 6개 문자는 0 에서부터 5 까지 번호가 매겨졌다. 마지막 문자를 얻기 위해서 length에서 1을 빼야 한다.

```
>>> last = fruit[length-1]
>>> print last
a
```

대안으로 음의 인텍스를 사용해서 문자열 끝에서 역으로 수를 셀 수 있다. 표현식 fruit[-1]은 마지막 문자를 fruit[-2]는 끝에서 두 번째 등등 활용할 수 있다.

제 3 절 루프를 사용한 문자열 운행법

연산의 많은 경우에 문자열을 한번에 한 문자씩 처리한다. 종종 처음에서 시작해서, 차례로 각 문자를 선택하고, 선택된 문자에 임의 연산을 수행하고, 끝까지 계속한다. 이런 처리 패턴을 운행법(traversal)라고 한다. 운행법을 작성하는 한 방법이 while 루프다.

```
index = 0
while index < len(fruit):
    letter = fruit[index]
    print letter
    index = index + 1
```

while 루프가 문자열을 운행하여 문자열을 한줄에 한 글자씩 화면에 출력한다. 루프 조건이 index < len(fruit)이여서, index가 문자열 길이와 같을 때, 조건은 거짓이 되고, 루프의 몸통 부문은 실행이 되지 않는다. 파이썬이 접근한 마지막 len(fruit)-1 인텍스 문자로, 문자열의 마지막 문자다.

Exercise 6.1 문자열의 마지막 문자에서 시작해서, 문자열 처음으로 역진행하면서 한줄에 한자씩 화면에 출력하는 while 루프를 작성하세요.

운행법을 작성하는 또 다른 방법은 `for` 루프다.

```
for char in fruit:
    print char
```

루프를 매번 반복할 때, 문자열 다음 문자가 변수 `char`에 대입된다. 루프는 더 이상 남겨진 문자가 없을 때까지 계속 실행된다.

제 4 절 문자열 슬라이스(slice)

문자열의 일부분을 슬라이스(slice)라고 한다. 문자열 슬라이스를 선택하는 것은 문자를 선택하는 것과 유사하다.

```
>>> s = 'Monty Python'
>>> print s[0:5]
Monty
>>> print s[6:13]
Python
```

`[n:m]` 연산자는 n번째 문자부터 m번째 문자까지의 문자열 - 첫 번째는 포함하지만 마지막은 제외 - 부분을 반환한다.

콜론 앞 첫 인덱스를 생략하면, 문자열 슬라이스는 문자열 처음부터 시작한다. 두 번째 인덱스를 생략하면, 문자열 슬라이스는 문자열 끝까지 간다.

```
>>> fruit = 'banana'
>>> fruit[:3]
'ban'
>>> fruit[3:]
'ana'
```

만약 첫번째 인덱스가 두번째보다 크거나 같은 경우 결과는 인용부호로 표현되는 빈 문자열(empty string)이 된다.

```
>>> fruit = 'banana'
>>> fruit[3:3]
''
```

빈 문자열은 어떤 문자도 포함하지 않아서 길이가 0 이 되지만, 이것을 제외하고 다른 문자열과 동일하다.

Exercise 6.2 `fruit`이 문자열로 주어졌을 때, `fruit[:]`의 의미는 무엇인가요?

제 5 절 문자열은 불변이다.

문자열 내부에 있는 문자를 변경하려고 대입문 왼쪽편에 `[]` 연산자를 사용하고 싶은 유혹이 있을 것이다. 예를 들어 다음과 같다.

```
>>> greeting = 'Hello, world!'
>>> greeting[0] = 'J'
TypeError: object does not support item assignment
```

이 경우 "객체(object)"는 문자열이고, 대입하고자 하는 문자는 "항목(item)"이다. 지금으로서 객체는 값과 동일하지만, 나중에 객체 정의를 좀더 상세화할 것이다. 항목은 순서 값 중의 하나다.

오류 이유는 문자열은 불변(immutable)이기 때문이다. 따라서 기존 문자열을 변경할 수 없다는 의미다. 최선의 방법은 원래 문자열을 변형한 새로운 문자열을 생성하는 것이다.

```
>>> greeting = 'Hello, world!'
>>> new_greeting = 'J' + greeting[1:]
>>> print new_greeting
Jello, world!
```

새로운 첫 문자에 `greeting` **문자열 슬라이스를 연결한다. 원래 문자열에는 어떤 영향도 주지 않는 새로운 문자열을 생성되었다.**

제 6 절 루프 돌기(looping) 계수(counting)

다음 프로그램은 문자열에 문자 `a`**가 나타나는 횟수를 계수한다.**

```
word = 'banana'
count = 0
for letter in word:
    if letter == 'a':
        count = count + 1
print count
```

상기 프로그램은 계수기(counter)라고 부르는 또다른 연산 패턴을 보여준다. 변수 `count`**는 0 으로 초기화 되고, 매번** `a`**를 찾을 때마다 증가한다. 루프를 빠져나갔을 때,** `count`**는 결과 값 즉,** `a`**가 나타난 총 횟수를 담고 있다.**

Exercise 6.3 문자열과 문자를 인자(argument)로 받도록 상기 코드를 `count`라는 함수로 캡슐화(encapsulation)하고 일반화하세요.

제 7 절 `in` 연산자

연산자 `in` **은 불 연산자로 두 개의 문자열을 받아, 첫 번째 문자열이 두 번째 문자열의 일부이면** 참`(True)`**을 반환한다.**

```
>>> 'a' in 'banana'
True
>>> 'seed' in 'banana'
False
```

제 8 절 문자열 비교

비교 연산자도 문자열에서 동작한다. 두 문자열이 같은지를 살펴보다.

```
if word == 'banana':
    print  'All right, bananas.'
```

다른 비교 연산자는 단어를 알파벳 순서로 정렬하는데 유용하다.

```
if word < 'banana':
    print 'Your word,' + word + ', comes before banana.'
elif word > 'banana':
    print 'Your word,' + word + ', comes after banana.'
else:
    print 'All right, bananas.'
```

파이썬은 사람과 동일하는 방식으로 대문자와 소문자를 다루지 않는다. 모든 대문자는 소문자 앞에 온다.

```
Your word, Pineapple, comes before banana.
```

이러한 문제를 다루는 일반적인 방식은 비교 연산을 수행하기 전에 문자열을 표준 포맷으로 예를 들어 모두 소문자, 변환하는 것입니다. 경우에 따라서 "Pineapple"로 무장한 사람들로부터 여러분을 보호해야하는 것을 명심하세요.

제 9 절 string 메쏘드

문자열은 파이썬 객체(objects)의 한 예다. 객체는 데이터(실제 문자열 자체)와 메쏘드(methods)를 담고 있다. 메쏘드는 객체에 내장되고 어떤 객체의 인스턴스(instance)에도 사용되는 사실상 함수다.

객체에 대해 이용가능한 메쏘드를 보여주는 dir 함수가 파이썬에 있다. type 함수는 객체의 자료형(type)을 보여 주고, dir은 객체에 사용될 수 있는 메쏘드를 보여준다.

```
>>> stuff = 'Hello world'
>>> type(stuff)
<type 'str'>
>>> dir(stuff)
['capitalize', 'center', 'count', 'decode', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> string

    Return a copy of the string S with only its first character
    capitalized.
>>>
```

dir 함수가 메쏘드 목록을 보여주고, 메쏘드에 대한 간단한 문서 정보는 help를 사용할 수 있지만, 문자열 메쏘드에 대한 좀더 좋은 문서 정보는 **docs.python.org/library/string.html**에서 찾을 수 있다.

인자를 받고 값을 반환한다는 점에서 메쏘드(method)를 호출하는 것은 함수를 호출하는 것과 유사하지만, 구문은 다르다. 구분자로 점을 사용해서 변수명에 메쏘드명을 붙여 메쏘드를 호출한다.

예를 들어, upper 메쏘드는 문자열을 받아 모두 대문자로 변환된 새로운 문자열을 반환한다.

함수 구문 upper(word) 대신에, word.upper() 메쏘드 구문을 사용한다.

```
>>> word = 'banana'
>>> new_word = word.upper()
>>> print new_word
BANANA
```

이런 형태의 점 표기법은 메쏘드 이름(upper)과 메쏘드가 적용되는 문자열 이름(word)을 명세한다. 빈 괄호는 메쏘드가 인자가 없다는 것을 나타낸다.

메쏘드를 부르는 것을 호출(invocation)이라고 부른다. 상기의 경우, word에 upper 메쏘드를 호출한다고 말한다.

예를 들어, 문자열안에 문자열의 위치를 찾는 find라는 문자열 메쏘드가 있다.

```
>>> word = 'banana'
>>> index = word.find('a')
>>> print index
1
```

상기 예제에서, word 문자열의 find 메쏘드를 호출하여 매개 변수로 찾고자 하는 문자를 넘긴다.

find 메쏘드는 문자뿐만 아니라 부속 문자열(substring)도 찾을 수 있다.

```
>>> word.find('na')
2
```

두 번째 인자로 어디서 검색을 시작할지 인텍스를 넣을 수 있다.

```
>>> word.find('na', 3)
4
```

한 가지 자주 있는 작업은 strip 메쏘드를 사용해서 문자열 시작과 끝의 공백(공백 여러개, 탭, 새줄)을 제거하는 것이다.

```
>>> line = '  Here we go  '
>>> line.strip()
'Here we go'
```

startswith 메쏘드는 참, 거짓 같은 불 값(boolean value)을 반환한다.

```
>>> line = 'Please have a nice day'
>>> line.startswith('Please')
True
>>> line.startswith('p')
False
```

startswith가 대소문자를 구별하는 것을 요구하기 때문에 `lower` **메쏘드를 사용해서 검증을 수행하기 전에, 한 줄을 입력받아 모두 소문자로 변환하는 것이 필요하다.**

```
>>> line = 'Please have a nice day'
>>> line.startswith('p')
False
>>> line.lower()
'please have a nice day'
>>> line.lower().startswith('p')
True
```

마지막 예제에서 문자열이 문자 "p"로 시작하는지를 검증하기 위해서, `lower` **메쏘드를 호출하고 나서 바로** `startswith` **메쏘드를 사용한다. 주의깊게 순서만 다룬다면, 한 줄에 다수 메쏘드를 호출할 수 있다.**

Exercise 6.4 앞선 예제와 유사한 함수인 `count`로 불리는 문자열 메쏘드가 있다. `docs.python.org/library/string.html`에서 `count` 메쏘드에 대한 문서를 읽고, 문자열 `'banana'`의 문자가 몇 개인지 계수하는 메쏘드 호출 프로그램을 작성하세요.

제 10 절 문자열 파싱(Parsing)

종종, 문자열을 들여다 보고 특정 부속 문자열(substring)을 찾고 싶다. 예를 들어, 아래와 같은 형식으로 작성된 일련의 라인이 주어졌다고 가정하면,

`From stephen.marquard@` **uct.ac.za** `Sat Jan 5 09:14:16 2008`

각 라인마다 뒤쪽 전자우편 주소(즉, `uct.ac.za`**)만 뽑아내고 싶을 것이다.** `find` **메쏘드와 문자열 슬라이싱(string sliceing)을 사용해서 작업을 수행할 수 있다.**

우선, 문자열에서 골뱅이(@, at-sign) 기호의 위치를 찾는다. 그리고, 골뱅이 기호 *뒤* **첫 공백 위치를 찾는다. 그리고 나서, 찾고자 하는 부속 문자열을 뽑아내기 위해서 문자열 슬라이싱을 사용한다.**

```
>>> data = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> atpos = data.find('@')
>>> print atpos
21
>>> sppos = data.find(' ',atpos)
>>> print sppos
31
>>> host = data[atpos+1:sppos]
>>> print host
uct.ac.za
>>>
```

find 메쏘드를 사용해서 찾고자 하는 문자열의 시작 위치를 명세한다. 문자열 슬라이싱(slicing)할 때, 골뱅기 기호 뒤부터 빈 공백을 *포함하지 않는* 위치까지 문자열을 뽑아낸다.

find 메쏘드에 대한 문서는 **docs.python.org/library/string.html** 에서 참조 가능하다.

제 11 절 서식 연산자

서식 연산자(format operator), %는 문자열 일부를 변수에 저장된 값으로 바꿔 문자열을 구성한다. 정수에 서식 연산자가 적용될 때, %는 나머지 연산자가 된다. 하지만 첫 피연산자가 문자열이면, %은 서식 연산자가 된다.

첫 피연산자는 서식 문자열 format string로 두번째 피연산자가 어떤 형식으로 표현되는지를 명세하는 하나 혹은 그 이상의 서식 순서 format sequence를 담고 있다. 결과값은 문자열이다.

예를 들어, 형식 순서 '%d'의 의미는 두번째 피연산자가 정수 형식으로 표현됨을 뜻한다. (d는 "decimal"을 나타낸다.)

```
>>> camels = 42
>>> '%d' % camels
'42'
```

결과는 문자열 '42'로 정수 42와 혼동하면 안 된다.

서식 순서는 문자열 어디에도 나타날 수 있어서 문장 중간에 값을 임베드(embed)할 수 있다.

```
>>> camels = 42
>>> 'I have spotted %d camels.' % camels
'I have spotted 42 camels.'
```

만약 문자열 서식 순서가 하나 이상이라면, 두번째 인자는 튜플(tuple)이 된다. 서식 순서 각각은 순서대로 튜플 요소와 매칭된다.

다음 예제는 정수 형식을 표현하기 위해서 '%d', 부동 소수점 형식을 표현하기 위해서 '%g', 문자열 형식을 표현하기 위해서 '%s'을 사용한 사례다. 여기서 왜 부동 소수점 형식이 '%f'대신에 '%g'인지는 질문하지 말아주세요.

```
>>> 'In %d years I have spotted %g %s.' % (3, 0.1, 'camels')
'In 3 years I have spotted 0.1 camels.'
```

튜플 요소 숫자는 문자열 서식 순서의 숫자와 일치해야 하고, 요소의 자료형(type)도 서식 순서와 일치해야 한다.

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dollars'
TypeError: illegal argument type for built-in operation
```

상기 첫 예제는 충분한 요소 개수가 되지 않고, 두 번째 예제는 자료형이 맞지 않는다.

서식 연산자는 강력하지만, 사용하기가 어렵다. 더 많은 정보는 **docs.python.org/lib/typesseq-strings.html**에서 찾을 수 있다.

제 12 절 디버깅

프로그램을 작성하면서 배양해야 하는 기술은 항상 자신에게 질문을 하는 것이다. "여기서 무엇이 잘못 될 수 있을까?" 혹은 "내가 작성한 완벽한 프로그램을 망가뜨리기 위해 사용자는 무슨 엄청난 일을 할 것인가?"

예를 들어 앞장의 반복 `while` 루프를 시연하기 위해 사용한 프로그램을 살펴봅시다.

```
while True:
    line = raw_input('> ')
    if line[0] == '#' :
        continue
    if line == 'done':
        break
    print line

print 'Done!'
```

사용자가 입력값으로 빈 공백 줄을 입력하게 될 때 무엇이 발생하는지 살펴봅시다.

```
> hello there
hello there
> # don't print this
> print this!
print this!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if line[0] == '#' :
```

빈 공백줄이 입력될 때까지 코드는 잘 작동합니다. 그리고 나서, 0 번째 문자가 없어서 트레이스백(traceback)이 발생한다. 입력줄이 비어있을 때, 코드 3번째 줄을 "안전"하게 만드는 두 가지 방법이 있다.

하나는 빈 문자열이면 거짓(`False`)을 반환하도록 `startswith` 메쏘드를 사용하는 것이다.

```
    if line.startswith('#') :
```

가디언 패턴(guardian pattern)을 사용한 `if`문으로 문자열에 적어도 하나의 문자가 있는 경우만 두번째 논리 표현식이 평가되도록 코드를 작성한다.

```
    if len(line) > 0 and line[0] == '#' :
```

제 13 절 용어정의

계수기(counter): 무언가를 계수하기 위해서 사용되는 변수로 일반적으로 0 으로 초기화하고 나서 증가한다.

빈 문자열(empty string): 두 인용부호로 표현되고, 어떤 문자도 없고 길이가 0 인 문자열.

서식 연산자(format operator): 서식 문자열과 튜플을 받아, 서식 문자열에 지정된 서식으로 튜플 요소를 포함하는 문자열을 생성하는 연산자, `%`.

서식 순서(format sequence): `%d`처럼 어떤 값의 서식으로 표현되어야 하는지를 명세하는 "서식 문자열" 문자 순서.

서식 문자열(format string): 서식 순서를 포함하는 서식 연산자와 함께 사용되는 문자열.

플래그(flag): 조건이 참인지를 표기하기 위해 사용하는 불 변수(boolean variable)

호출(invocation): 메쏘드를 호출하는 명령문.

불변(immutable): 순서의 항목에 대입할 수 없는 특성.

인덱스(index): 문자열의 문자처럼 순서(sequence)에 항목을 선택하기 위해 사용되는 정수 값.

항목(item): 순서에 있는 값의 하나.

메쏘드(method): 객체와 연관되어 점 표기법을 사용하여 호출되는 함수.

객체(object): 변수가 참조하는 무엇. 지금은 "객체"와 "값"을 구별없이 사용한다.

검색(search): 찾고자 하는 것을 찾았을 때 멈추는 운행법 패턴.

순서(sequence): 정돈된 집합. 즉, 정수 인덱스로 각각의 값이 확인되는 값의 집합.

슬라이스(slice): 인덱스 범위로 지정되는 문자열 부분.

운행법(traverse): 순서(sequence)의 항목을 반복적으로 훑기, 각각에 대해서는 동일한 연산을 수행.

제 14 절 연습문제

Exercise 6.5 다음 문자열을 파이썬 코드를 작성하세요.

```
str = 'X-DSPAM-Confidence: 0.8475'
```

find 메쏘드와 문자열 슬라이싱을 사용하여 콜론(:) 문자 뒤 문자열을 뽑아내고 float 함수를 사용하여 뽑아낸 문자열을 부동 소수점 숫자로 변환하세요.

Exercise 6.6 https://docs.python.org/2.7/library/stdtypes.html#string-methods에서 문자열 메쏘드 문서를 읽어보세요. 어떻게 동작하는가를 이해도를 확인하기 위해서 몇개를 골라 실험을 해보세요. strip과 replace가 특히 유용합니다.

문서는 좀 혼동스러울 수 있는 구문을 사용합니다. 예를 들어, find(sub[, start[, end]])의 꺾쇠기호는 선택(옵션) 인수를 나타냅니다. 그래서, sub은 필수지만, start은 선택 사항이고, 만약 start가 인자로 포함된다면, end는 선택이 된다.

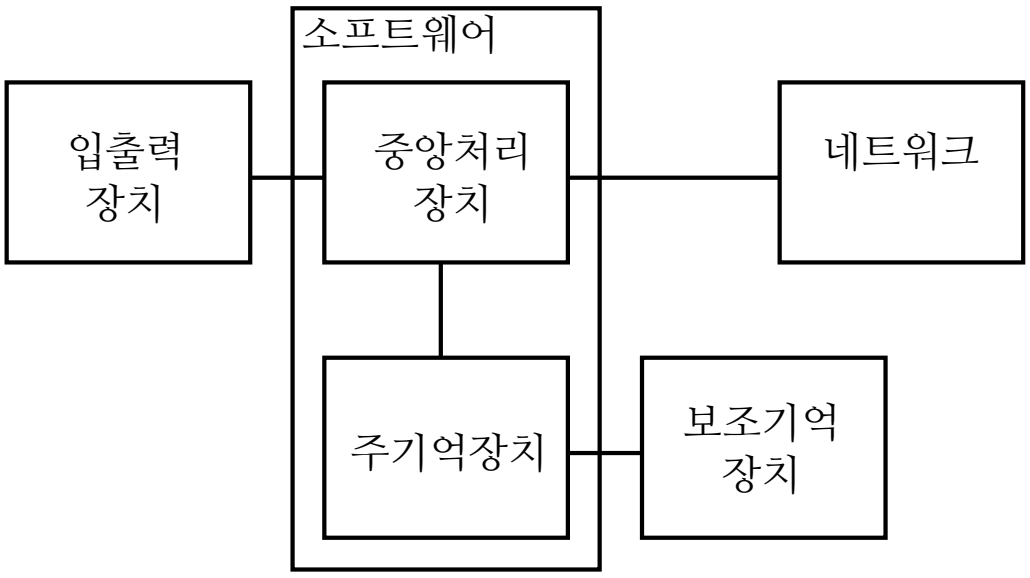
제 7 장

파일

제 1 절　영속성(Persistence)

지금까지, 프로그램을 어떻게 작성하고 조건문, 함수, 반복을 사용하여 중앙처리장치(CPU, Central Processing Unit)에 프로그래머의 의도를 커뮤니케이션하는지 학습했다. 주기억장치(Main Memory)에 어떻게 자료구조를 생성하고 사용하는지도 배웠다. CPU와 주기억장치는 소프트웨어가 동작하고 실행되는 곳이고, 모든 "생각(thinking)"이 발생하는 장소다.

하지만, 앞서 하드웨어 아키텍처를 논의했던 기억을 되살린다면, 전원이 꺼지게 되면, CPU와 주기억장치에 저장된 모든 것이 지워진다. 지금까지 작성한 프로그램은 파이썬을 배우기 위한 일시적으로 재미로 연습한 것에 불과하다.



이번 장에서 보조 기억장치(Secondary Memory) 혹은 파일을 가지고 작업을 시작한다. 보조 기억장치는 전원이 꺼져도 지워지지 않는다. 혹은, USB 플래쉬 드라이브를 사용한 경우에는 프로그램으로부터 작성한 데이터는 시스템에서 제거되어 다른 시스템으로 전송될 수 있다.

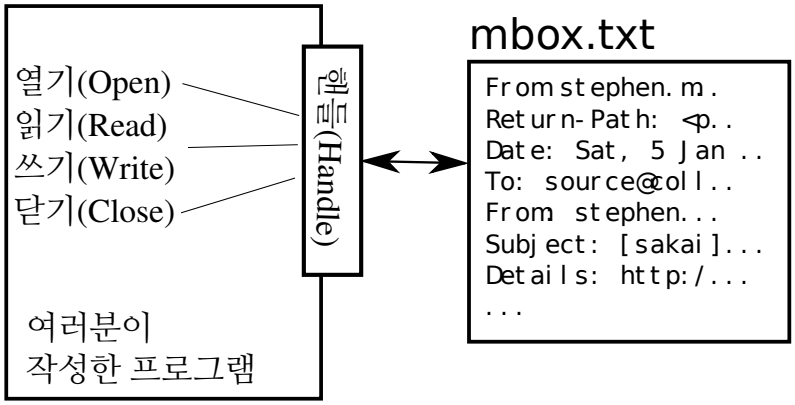
우선 텍스트 편집기로 작성한 텍스트 파일을 읽고 쓰는 것에 초점을 맞출 것이다. 나중에 데이터베이스 소프트웨어를 통해서 읽고 쓰도록 설계된 바이너리 파일 데이터베이스를 가지고 어떻게 작업하는지를 살펴볼 것이다.

제 2 절 파일 열기

하드 디스크 파일을 읽거나 쓸려고 할 때, 파일을 열여야(open) 한다. 파일을 열 때 각 파일 데이터가 어디에 저장되었는지를 알고 있는 운영체제와 커뮤니케이션 한다. 파일을 열 때, 운영체제에 파일이 존재하는지 확인하고 이름으로 파일을 찾도록 요청한다. 이번 예제에서, 파이썬을 시작한 동일한 폴더에 저장된 `mbox.txt` 파일을 연다. **`www.py4inf.com/code/mbox.txt`** 에서 파일을 다운로드할 수 있다.

```
>>> fhand = open('mbox.txt')
>>> print fhand
<open file 'mbox.txt', mode 'r' at 0x1005088b0>
```

`open`이 성공하면, 운영체제는 파일 핸들(file handle)을 반환한다. 파일 핸들(file handle)은 파일에 담겨진 실제 데이터는 아니고, 대신에 데이터를 읽을 수 있도록 사용할 수 있는 "핸들(handle)"이다. 요청한 파일이 존재하고, 파일을 읽을 수 있는 적절한 권한이 있다면 이제 핸들이 여러분에게 주어졌다.



파일이 존재하지 않는다면, `open`은 역추적(traceback) 파일 열기 오류로 실패하고, 파일 콘텐츠에 접근할 핸들도 얻지 못한다.

```
>>> fhand = open('stuff.txt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'stuff.txt'
```

나중에 `try`와 `except`를 가지고, 존재하지 않는 파일을 열려고 하는 상황을 좀더 우아하게 처리할 것이다.

제 3 절 텍스트 파일과 라인

파이썬 문자열이 문자 순서(sequence)로 간주 되듯이 마찬가지로 텍스트 파일은 라인 순서(sequence)로 생각될 수 있다. 예를 들어, 다음은 오픈 소스 프로젝

트 개발 팀에서 다양한 참여자들의 전자우편 활동을 기록한 텍스트 파일 샘플이다.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
Return-Path: <postmaster@collab.sakaiproject.org>
Date: Sat, 5 Jan 2008 09:12:18 -0500
To: source@collab.sakaiproject.org
From: stephen.marquard@uct.ac.za
Subject: [sakai] svn commit: r39772 - content/branches/
Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
...
```

상호 의사소통한 전자우편 전체 파일은 **www.py4inf.com/code/mbox.txt** 에서 접근 가능하고, 간략한 버젼 파일은 **www.py4inf.com/code/mbox-short.txt**에서 얻을 수 있다. 이들 파일은 다수 전자우편 메시지를 담고 있는 파일로 표준 포맷으로 되어 있다. "From "으로 시작하는 라인은 메시지 본문과 구별되고, "From: "으로 시작하는 라인은 본문 메시지의 일부다. 더 자세한 정보는 **en.wikipedia.org/wiki/Mbox**에서 찾을 수 있다.

파일을 라인으로 쪼개기 위해서, 새줄(newline) 문자로 불리는 "줄의 끝(end of the line)"을 표시하는 특수 문자가 있다.

파이썬에서, 문자열 상수 역슬래쉬-n(`\n`)으로 새줄(newline) 문자를 표현한다. 두 문자처럼 보이지만, 사실은 단일 문자다. 인터프리터에 ”stuff”에 입력한 후 변수를 살펴보면, 문자열에 `\n`가 있다. 하지만, `print`문을 사용하여 문자열을 출력하면, 문자열이 새줄 문자에 의해서 두 줄로 쪼개지는 것을 볼 수 있다.

```
>>> stuff = 'Hello\nWorld!'
>>> stuff
'Hello\nWorld!'
>>> print stuff
Hello
World!
>>> stuff = 'X\nY'
>>> print stuff
X
Y
>>> len(stuff)
3
```

문자열 `'X\nY'`의 길이는 *3* 이다. 왜냐하면 새줄(newline) 문자도 한 문자이기 때문이다.

그래서, 파일 라인을 볼 때, 라인 끝을 표시하는 새줄(newline)로 불리는 눈에 보이지 않는 특수 문자가 각 줄의 끝에 있다고 *상상*할 필요가 있다.

그래서, 새줄(newline) 문자는 파일에 있는 문자를 라인으로 분리한다.

제 4 절 파일 읽어오기

파일 핸들(file handle)이 파일 자료를 담고 있지 않지만, `for` 루프를 사용하여 파일 각 라인을 읽고 라인수를 세는 것을 쉽게 구축할 수 있다.

```
fhand = open('mbox.txt')
count = 0
for line in fhand:
    count = count + 1
print 'Line Count:', count

python open.py
Line Count: 132045
```

파일 핸들을 `for` 루프 순서(sequence)로 사용할 수 있다. `for` 루프는 단순히 파일 라인 수를 세고 전체 라인수를 출력한다. `for` 루프를 대략 일반어로 풀어 말하면, "파일 핸들로 표현되는 파일 각 라인마다, `count` 변수에 1 씩 더한다"

`open` 함수가 전체 파일을 바로 읽지 못하는 이유는 파일이 수 기가 바이트 파일 크기를 가질 수도 있기 때문이다. `open` 문장은 파일 크기에 관계없이 파일을 여는데 시간이 동일하게 걸린다. 실질적으로 `for` 루프가 파일로부터 자료를 읽어오는 역할을 한다.

`for` 루프를 사용해서 이 같은 방식으로 파일을 읽어올 때, 새줄(newline) 문자를 사용해서 파일 자료를 라인 단위로 쪼갠다. 파이썬에서 새줄(newline) 문자까지 각 라인 단위로 읽고, `for` 루프가 매번 반복할 때마다 `line` 변수에 새줄(newline)을 마지막 문자로 포함한다.

`for` 루프가 데이터를 한번에 한줄씩 읽어오기 때문에, 데이터를 저장할 주기억장치 저장공간을 소진하지 않고, 매우 큰 파일을 효과적으로 읽어서 라인을 셀 수 있다. 각 라인별로 읽고, 세고, 그리고 나서 폐기되기 때문에, 매우 적은 저장공간을 사용해서 어떤 크기의 파일도 상기 프로그램을 사용하여 라인을 셀 수 있다.

만약 주기억장치 크기에 비해서 상대적으로 작은 크기의 파일이라는 것을 안다면, 전체 파일을 파일 핸들로 `read` 메쏘드를 사용해서 하나의 문자열로 읽어올 수 있다.

```
>>> fhand = open('mbox-short.txt')
>>> inp = fhand.read()
>>> print len(inp)
94626
>>> print inp[:20]
From stephen.marquar
```

상기 예제에서, `mbox-short.txt` 전체 파일 콘텐츠(94,626 문자)를 변수 `inp`로 바로 읽었다. 문자열 슬라이싱을 사용해서 `inp`에 저장된 문자열 자료 첫 20 문자를 출력한다.

파일이 이런 방식으로 읽혀질 때, 모든 라인과 새줄(newline)문자를 포함한 모든 문자는 변수 `inp`에 대입된 매우 큰 문자열이다. 파일 데이터가 컴퓨터 주기억장치가 안정적으로 감당해 낼 수 있을때만, 이런 형식의 `open` 함수가 사용될 수 있다는 것을 기억하라.

만약 주기억장치가 감당해 낼 수 없는 매우 파일 크기가 크다면, `for`나 `while` 루프를 사용해서 파일을 쪼개서 읽는 프로그램을 작성해야 한다.

제 5 절 파일 검색

파일 데이터를 검색할 때, 흔한 패턴은 파일을 읽고, 대부분 라인은 건너뛰고, 특정 기준을 만족하는 라인만 처리하는 것이다. 간단한 검색 메카니즘을 구현하기 위해서 파일을 읽는 패턴과 문자열 메쏘드를 조합한다.

예를 들어, 파일을 읽고, "From:"으로 시작하는 라인만 출력하고자 한다면, startswith 문자열 메쏘드를 사용해서 원하는 접두사로 시작하는 라인만을 선택한다.

```
fhand = open('mbox-short.txt')
for line in fhand:
    if line.startswith('From:') :
        print line
```

이 프로그램이 실행하면 다음 출력값을 얻는다.

```
From: stephen.marquard@uct.ac.za

From: louis@media.berkeley.edu

From: zqian@umich.edu

From: rjlowe@iupui.edu
...
```

"From:"으로만 시작하는 라인만 출력하기 때문에 출력값은 훌륭해 보인다. 하지만, 왜 여분으로 빈 라인이 보이는 걸까? 원인은 눈에 보이지 않는 새줄(newline) 문자 때문이다. 각 라인이 새줄(newline)로 끝나서 변수 line에 새줄(newline)이 포함되고 `print`문이 추가로 새줄(newline)을 추가해서 결국 우리가 보기에는 두 줄 효과가 나타난다.

마지막 문자를 제외하고 모든 것을 출력하기 위해서 라인 슬라이싱(slicing)을 할수 있지만, 좀더 간단한 접근법은 다음과 같이 문자열 오른쪽 끝에서부터 공백을 벗겨내는 rstrip 메쏘드를 사용하는 것인다.

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.startswith('From:') :
        print line
```

프로그램을 실행하면, 다음 출력값을 얻는다.

```
From: stephen.marquard@uct.ac.za
From: louis@media.berkeley.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: zqian@umich.edu
From: rjlowe@iupui.edu
From: cwen@iupui.edu
...
```

파일 처리 프로그램이 점점 더 복잡해짐에 따라 continue를 사용해서 검색 루프(search loop)를 구조화할 필요가 있다. 검색 루프의 기본 아이디어는 "흥미로운" 라인을 집중적으로 찾고, "흥미롭지 않은" 라인은 효과적으로 건너뛰는 것이다. 그리고 나서 흥미로운 라인을 찾게되면, 그 라인에서 특정 연산을 수행하는 것이다.

다음과 같이 루프를 구성해서 흥미롭지 않은 라인은 건떠뛰는 패턴을 따르게 한다.

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    # Skip 'uninteresting lines'
    if not line.startswith('From:') :
        continue
    # Process our 'interesting' line
    print line
```

프로그램의 출력값은 동일하다. 흥미롭지 않는 라인은 "From:"으로 시작하지 않는 라인이라 continue문을 사용해서 건너뛴다. "흥미로운" 라인 (즉, "From:"으로 시작하는 라인)에 대해서는 연산처리를 수행한다.

find 문자열 메쏘드를 사용해서 검색 문자열이 라인 어디에 있는지를 찾아주는 텍스트 편집기 검색기능을 모사(simulation)할 수 있다. find 메쏘드는 다른 문자열 내부에 검색하는 문자열이 있는지 찾고, 문자열 위치를 반환하거나, 만약 문자열이 없다면 -1을 반환하기 때문에, "@uct.ac.za"(남아프리카 케이프 타운 대학으로부터 왔다) 문자열을 포함하는 라인을 검색하기 위해 다음과 같이 루프를 작성한다.

```
fhand = open('mbox-short.txt')
for line in fhand:
    line = line.rstrip()
    if line.find('@uct.ac.za') == -1 :
        continue
    print line
```

출력결과는 다음과 같다.

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
X-Authentication-Warning: set sender to stephen.marquard@uct.ac.za using -f
From: stephen.marquard@uct.ac.za
Author: stephen.marquard@uct.ac.za
From david.horwitz@uct.ac.za Fri Jan  4 07:02:32 2008
X-Authentication-Warning: set sender to david.horwitz@uct.ac.za using -f
From: david.horwitz@uct.ac.za
Author: david.horwitz@uct.ac.za
...
```

제 6 절 사용자가 파일명을 선택하게 만들기

매번 다른 파일을 처리할 때마다 파이썬 코드를 편집하고 싶지는 않다. 매번 프로그램이 실행될 때마다, 파일명을 사용자가 입력하도록 만드는 것이 좀더 유

용할 것이다. 그래서 파이썬 코드를 바꾸지 않고, 다른 파일에 대해서도 동일한 프로그램을 사용하도록 만들자.

다음과 같이 raw_input을 사용해서 사용자로부터 파일명을 읽어 프로그램을 실행하는 것이 단순하다.

```
fname = raw_input('Enter the file name: ')
fhand = open(fname)
count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
print 'There were', count, 'subject lines in', fname
```

사용자로부터 파일명을 읽고 변수 fname에 저장하고, 그 파일을 연다. 이제 다른 파일에 대해서도 반복적으로 프로그램을 실행할 수 있다.

```
python search6.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search6.py
Enter the file name: mbox-short.txt
There were 27 subject lines in mbox-short.txt
```

다음 절을 엿보기 전에, 상기 프로그램을 살펴보고 자신에게 다음을 질문해 보자. "여기서 어디가 잘못될 수 있는가?" 혹은 "이 작고 멋진 프로그램에 트레이스백(traceback)을 남기고 바로 끝나게 하여, 결국 사용자 눈에는 좋지 않은 프로그램이라는 인상을 남길 수 있도록 우리의 친절한 사용자는 무엇을 할 수 있을까?

제 7 절 try, except, open 사용하기

제가 여러분에게 엿보지 말라고 말씀드렸습니다. 이번이 마지막 기회입니다. 사용자가 파일명이 아닌 뭔가 다른 것을 입력하면 어떻게 될까요?

```
python search6.py
Enter the file name: missing.txt
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
IOError: [Errno 2] No such file or directory: 'missing.txt'

python search6.py
Enter the file name: na na boo boo
Traceback (most recent call last):
  File "search6.py", line 2, in <module>
    fhand = open(fname)
IOError: [Errno 2] No such file or directory: 'na na boo boo'
```

웃지마시구요, 사용자는 결국 여러분이 작성한 프로그램을 망가뜨리기 위해 고의든 악의를 가지든 가능한 모든 수단을 강구할 것입니다. 사실, 소프트웨어 개

발팀의 중요한 부분은 품질 보증(Quality Assurance, QA)이라는 조직이다. 품질보증 조직은 프로그래머가 만든 소프트웨어를 망가뜨리기 위해 가능한 말도 안 되는 것을 합니다.

사용자가 소프트웨어를 제품으로 구매하거나, 주문형으로 개발하는 프로그램에 대해 월급을 지급하던지 관계없이 품질보증 조직은 프로그램이 사용자에게 전달되기 전까지 프로그램 오류를 발견할 책임이 있다. 그래서 품질보증 조직은 프로그래머의 최고의 친구다.

프로그램 오류를 찾았기 때문에, `try`/`except` 구조를 사용해서 오류를 우아하게 고쳐봅시다. `open` 호출이 잘못될 수 있다고 가정하고, `open` 호출이 실패할 때를 대비해서 다음과 같이 복구 코드를 추가한다.

```
fname = raw_input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print 'File cannot be opened:', fname
    exit()

count = 0
for line in fhand:
    if line.startswith('Subject:') :
        count = count + 1
print 'There were', count, 'subject lines in', fname
```

`exit`함수가 프로그램을 끝낸다. 결코 돌아오지 않는 함수를 호출한 것이다. 이제 사용자 혹은 품질 보증 조직에서 올바르지 않거나 어처구니 없는 파일명을 입력했을 때, “catch”로 잡아서 우아하게 복구한다.

```
python search7.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python search7.py
Enter the file name: na na boo boo
File cannot be opened: na na boo boo
```

파이썬 프로그램을 작성할 때 `open` 호출을 보호하는 것은 `try`, `except`의 적절한 사용 예제가 된다. ”파이썬 방식(Python way)”으로 무언가를 작성할 때, ”파이썬스러운(Pythonic)”이라는 용어를 사용한다. 상기 파일을 여는 예제는 파이썬스러운 방식의 좋은 예가 된다고 말한다.

파이썬에 좀더 자신감이 생기게 되면, 다른 파이썬 프로그래머와 동일한 문제에 대해 두 가지 동치하는 해답을 가지고 어떤 접근법이 좀더 "파이썬스러운지"에 대한 현답을 찾는데도 관여하게 된다.

"좀더 파이썬스럽게" 되는 이유는 프로그래밍이 엔지니어링적인 면과 예술적인 면을 동시에 가지고 있기 때문이다. 항상 무언가를 단지 작동하는 것에만 관심이 있지 않고, 프로그램으로 작성한 해결책이 좀더 우아하고, 다른 동료에 의해서 우아한 것으로 인정되기를 또한 원합니다.

제 8 절 파일에 쓰기

파일에 쓰기 위해서는 두 번째 매개 변수로 'w' 모드로 파일을 열어야 한다.

```
>>> fout = open('output.txt', 'w')
>>> print fout
<open file 'output.txt', mode 'w' at 0xb7eb2410>
```

파일이 이미 존재하는데 쓰기 모드에서 파일을 여는 것은 이전 데이터를 모두 지워버리고, 깨끗한 파일 상태에서 다시 시작되니 주의가 필요하다. 만약 파일이 존재하지 않는다면, 새로운 파일이 생성된다.

파일 핸들 객체의 write 메쏘드는 데이터를 파일에 저장한다.

```
>>> line1 = 'This here's the wattle,\n'
>>> fout.write(line1)
```

다시 한번, 파일 객체는 마지막 포인터가 어디에 있는지 위치를 추적해서, 만약 write 메쏘드를 다시 호출하게 되면, 새로운 데이터를 파일 끝에 추가한다.

라인을 끝내고 싶을 때, 명시적으로 새줄(newline) 문자를 삽입해서 파일에 쓰도록 라인 끝을 필히 관리해야 한다.

print문은 자동적으로 새줄(newline)을 추가하지만, write 메쏘드는 자동적으로 새줄(newline)을 추가하지는 않는다.

```
>>> line2 = 'the emblem of our land.\n'
>>> fout.write(line2)
```

파일 쓰기가 끝났을 때, 파일을 필히 닫아야 한다. 파일을 닫는 것은 데이터 마지막 비트까지 디스크에 물리적으로 쓰여져서, 전원이 나가더라도 자료가 유실되지 않는 역할을 한다.

```
>>> fout.close()
```

파일 읽기로 연 파일을 닫을 수 있지만, 몇개 파일을 열어 놓았다면 약간 단정치 못하게 끝날 수 있습니다. 왜냐하면 프로그램이 종료될 때 열린 모든 파일이 닫혀졌는지 파이썬이 확인하기 때문이다. 파일에 쓰기를 할 때는, 파일을 명시적으로 닫아서 예기치 못한 일이 발생할 여지를 없애야 한다.

제 9 절 디버깅

파일을 읽고 쓸 때, 공백 때문에 종종 문제에 봉착한다. 이런 종류의 오류는 공백, 탭, 새줄(newline)이 눈에 보이지 않기 때문에 디버그하기도 쉽지 않다.

```
>>> s = '1 2\t 3\n 4'
>>> print s
1 2  3
 4
```

내장함수 repr이 도움이 될 수 있다. 인자로 임의 객체를 잡아 객체 문자열 표현으로 반환한다. 문자열 공백문자는 역슬래쉬 순서(sequence)로 나타냅니다.

```
>>> print repr(s)
'1 2\t 3\n 4'
```

디버깅에 도움이 될 수 있다.

여러분이 봉착하는 또 다른 문제는 다른 시스템에서는 라인 끝을 표기하기 위해서 다른 문자를 사용한다는 점이다. 어떤 시스템은 `\n` **으로 새줄(newline)을 표기하고, 다른 시스템은** `\r`**으로 반환 문자(return character)를 사용한다. 둘 다 모두 사용하는 시스템도 있다. 파일을 다른 시스템으로 이식한다면, 이러한 불일치가 문제를 야기한다.**

대부분의 시스템에는 A 포맷에서 B 포멧으로 변환하는 응용프로그램이 있다. `wikipedia.org/wiki/Newline` **에서 응용프로그램을 찾을 수 있고, 좀더 많은 것을 읽을 수 있다. 물론, 여러분이 직접 프로그램을 작성할 수도 있다.**

제 10 절 용어정의

잡기(catch): `try`**와** `except` **문을 사용해서 프로그램이 끝나는 예외 상황을 방지하는 것.**

새줄(newline): 라인의 끝을 표기 위한 파일이나 문자열에 사용되는 특수 문자.

파이썬스러운(Pythonic): 파이썬에서 우아하게 작동하는 기술. "try와 catch를 사용하는 것은 파일이 없는 경우를 복구하는 파이썬스러운 방식이다."

품질 보증(Quality Assurance, QA): 소프트웨어 제품의 전반적인 품질을 보증하는데 집중하는 사람이나 조직. 품질 보증은 소프트웨어 제품을 시험하고, 제품이 시장에 출시되기 전에 문제를 확인하는데 관여한다.

텍스트 파일(text file): 하드디스크 같은 영구 저장소에 저장된 일련의 문자 집합.

제 11 절 연습문제

Exercise 7.1 파일을 읽고 한줄씩 파일의 내용을 모두 대문자로 출력하는 프로그램을 작성하세요. 프로그램을 싱행하면 다음과 같이 보일 것입니다.

```
python shout.py
Enter a file name: mbox-short.txt
FROM STEPHEN.MARQUARD@UCT.AC.ZA SAT JAN  5 09:14:16 2008
RETURN-PATH: <POSTMASTER@COLLAB.SAKAIPROJECT.ORG>
RECEIVED: FROM MURDER (MAIL.UMICH.EDU [141.211.14.90])
 BY FRANKENSTEIN.MAIL.UMICH.EDU (CYRUS V2.3.8) WITH LMTPA;
 SAT, 05 JAN 2008 09:14:16 -0500
```

`www.py4inf.com/code/mbox-short.txt`에서 파일을 다운로드 받으세요.

Exercise 7.2 파일명을 입력받아, 파일을 읽고, 다음 형식의 라인을 찾는 프로그램을 작성하세요.

`X-DSPAM-Confidence:` **0.8475**

"X-DSPAM-Confidence:"로 시작하는 라인을 만나게 되면, 부동 소수점 숫자를 뽑아내기 위해 해당 라인을 별도로 보관하세요. 라인 수를 세고, 라인으로부터 스팸 신뢰값의 총계를 계산하세요. 파일의 끝에 도달할 했을 때, 평균 스팸 신뢰도를 출력하세요.

```
Enter the file name: mbox.txt
Average spam confidence: 0.894128046745

Enter the file name: mbox-short.txt
Average spam confidence: 0.750718518519
```

`mbox.txt`와 `mbox-short.txt` 파일에 작성한 프로그램을 시험하세요.

Exercise 7.3 때때로, 프로그래머가 지루해지거나, 약간 재미를 목적으로, 프로그램에 무해한 **부활절 달걀**(Easter Egg, `en.wikipedia.org/wiki/Easter_egg_(media)`)을 넣습니다. 사용자가 파일명을 입력하는 프로그램을 변형시켜, 'na na boo boo'로 파일명을 정확하게 입력했을 때, 재미있는 메시지를 출력하는 프로그램을 작성하세요. 파일이 존재하거나, 존재하지 않는 다른 모든 파일에 대해서도 정상적으로 작동해야 합니다. 여기 프로그램을 실행한 견본이 있습니다.

```
python egg.py
Enter the file name: mbox.txt
There were 1797 subject lines in mbox.txt

python egg.py
Enter the file name: missing.tyxt
File cannot be opened: missing.tyxt

python egg.py
Enter the file name: na na boo boo
NA NA BOO BOO TO YOU - You have been punk'd!
```

프로그램에 부활절 달걀을 넣도록 격려하지는 않습니다. 단지 연습입니다.