Chapter 7

반복

이 장에서는 문장을 되풀이하여 실행하는 기능인 반복을 다룬다. 5.8절에서 재귀문을 사용한 반복의 일종을 살펴봤었다. 4.2절에서 `for` 루프도 봤었다. 이 장에서는 `while`문을 사용한 또 다른 반복문을 배울 것이다. 그 전에 변수 할당에 대해 좀 더 이야기하고 시작하겠다.

7.1 재할당

같은 변수에 한 번 이상 값을 할당해도 괜찮다는 것을 이제 깨달았을 것이다. 기존의 변수에 새로운 값을 할당하면 그 변수는 새로운 값을 가리킨다(이전에 가리키던 값은 더 이상 가리키지 않는다).

```
>>> x = 5
>>> x
5
>>> x = 7
>>> x
7
```

`x`는 처음에 5를 그리고 두 번째 7이라는 값을 갖는다.

그림 7.1의 스택 상태도에 **재할당(reassignment)**를 나타냈다.

현 시점에서 많이들 혼돈하는 것을 집고 넘어가자. Python에서 등호(=)를 사용하는데 이 기호를 수학 명제로서 `a = b`를 해석하려고한다. 수학에서는 `a`와 `b`가 서로 동일하다는 주장이다. 하지만, 그렇게 해석하면 틀린 것이다.

첫째로 등가는 대칭적 관계를 나타내지만 할당은 대칭적이지 않다. 예를 들어보자. 수학에서는 $a = 7$이면 $7 = a$이다. Python에서는 `a = 7` 이라는 문장은 괜찮지만 `7 = a`라는 문장은 틀린 문장이다.

수학에서는 어떤 식이 등가라면 그 주장은 항상 참이다. $a = b$라고 하면 a는 항상 b와 동일하다. Python에서 할당문은 두 변수가 서로 동일하게 만들지만 항상 그대로 유지되는 것은 아니다.

```
>>> a = 5
>>> b = a     # a와 b는 같음
>>> a = 3     # a와 b는 더 이상 같지 않음
>>> b
5
```

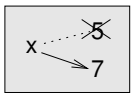

Figure 7.1: 스택 상태도

세 번째 줄은 a를 변경하지만 b는 변경시키지 않는다. 그렇기 때문에 둘은 더 이상 같지 않다.

변수 재할당은 유용하지만 신중하게 사용해야 한다. 변수의 값이 자주 바뀌면 코드 읽기와 디버깅이 어려워진다.

7.2 변수 갱신하기

흔한 할당 방법은 **갱신(update)**이다. 이런 종류는 변수의 새로운 값은 이전 변수의 값에 의존한다.

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

이 문장은 "현재 x의 값에 1을 더하고 그 값으로 x을 갱신한다.

존재하지 않는 변수를 갱신하면 오류가 발생한다. 왜냐하면 Python은 오른쪽의 표현을 먼저 계산하고 x에 값으로 할당하기 때문이다.

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

변수의 값을 갱신하기 전에 변수를 먼저 간단한 할당문으로 **초기화(initialize)**해야 한다.

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

변수의 값에 1을 더하는 것을 **증가(increment)**, 1을 빼는 것을 **감소(decrement)**라 부른다.

7.3 while문

컴퓨터는 반복적인 작업을 자동하는데 주로 사용된다. 동일하거나 유사한 작업을 오류 없이 처리하는 것을 컴퓨터는 잘하지만 사람은 그렇지 못하다. 컴퓨터 프로그램에서 퇴폴이되는 작업을 **반복(iteration)**이라 한다.

우리는 재귀문을 사용하여 반복하는 countdown과 print_n 함수를 봤었다. 반복 작업이 너무나 흔하기 때문에 Python은 언어 차원에서 반복 기능을 제공하고 있다. 그 중 하나는 4.2에서 봤던 for문이다. 다시 한 번 살펴볼 것이다.

또 다른 것은 while문이다. countdown을 while문으로 바꿔서 다시 작성해보자.

```
def countdown(n):
    while n > 0:
        print(n)
        n = n - 1
    print('발사!')
```

while문이 마치 영어처럼 거의 자연스럽게 읽혀진다. 해석하면 "n이 0보다 큰 동안에 n의 값을 표시하고 그 값을 감소시킨다. 0이 되면 발사!라고 표시"가 된다.

좀 더 체계적으로 while문의 실행 흐름을 살펴보자.

1. 조건이 참인지 거짓인지 판단한다.
2. 거짓이면 while문을 끝내고 다음 문장을 실행한다.
3. 조건이 참이면 while의 내용을 실행하고 첫 단계로 간다.

이와 같은 흐름을 루프(loop, 반복)라 부른다. 세 번째 단계가 처음으로 되돌아 반복하기 때문이다.

루프의 내용은 하나 또는 그 이상의 변수를 바꿔야 한다. 그래야 조건이 언젠가는 거짓이 되서 루프가 끝이나기 때문이다. 그렇지 않으면 루프는 무한히 되풀이될 것이다. 이런 경우를 **무한 루프(infinite loop)**이라 부른다. 샴푸의 사용법을 읽어보면 컴퓨터 과학자들은 놀라지 않을 수가 없다. "비누거품, 린스, 반복"라고 써 있는데, 이는 무한 루프이기 때문이다.

countdown의 경우 루프가 끝이 있음을 증명할 수 있다. 만약 n이 0이거나 음수이면 루프는 시작도 안된다. 그 외에는 0이 될 때까지 매번 루프를 돌 때마다 n은 감소된다.

어떤 루프의 경우에는 판단하기가 쉽지않다. 에를 들어보자.

```
def sequence(n):
    while n != 1:
        print(n)
        if n % 2 == 0:        # n은 짝수
            n = n / 2
        else:                 # n은 홀수
            n = n*3 + 1
```

이 루프의 조건은 n != 1이기 때문에 루프는 n이 1이 되어 거짓이 될 때까지 계속된다.

프로그램은 루프를 돌 때마다 n의 값을 표시하고 짝수 또는 홀수 검사를 한다. 짝수라면 n를 2로 나누고, 홀수라면 n은 n*3 +1으로 대체된다. sequence에 인자를 3으로 전달했다고 해보자. n의 결과는 3, 10, 5, 16, 8, 4, 2, 1이 된다.

n가 증가되기도 하고 감소하기도 하기 때문에 1에 반드시 도달한다거나 프로그램이 종료한다는 증명은 간단하지 않다. n이 특정 값인 경우라면 종료한다는 것을 증명할 수 있다. 시작 값이 2의 배수라면 n은 매번 짝수가 되기 때문에 루프를 반복하다보면 1에 도달하게 된다. 16으로 시작한 경우가 그렇게 끝이 난다.

좀 더 어려운 질문은 이 프로그램이 *모든* 양수에 대해서 종료하는가이다. 아직까지는 그 어느 누구도 이것을 증명도 반증하지도 못했다! (참고: http://en.wikipedia.org/wiki/Collatz_conjecture.)

연습 문제로 5.8절의 print_n 함수를 반복문을 사용하여 재작성해보라.

7.4 break문

때로는 루프의 반정도를 실행하기 전까지는 루프를 종료할 때가 되었는지 모를 때가 있다. 그런 경우에 break문을 사용하여 루프 밖으로 나올 수 있다.

사용자가 done을 입력하기 전까지 계속 입력을 받는 예를 살펴보자. 다음과 같이 작성하면 된다.

```
while True:
    line = input('> ')
    if line == 'done':
        break
```

```
    print(line)

print('Done!')
```

루프의 조건이 `True`라고 되어 있기 때문에 항상 참이다. 이 루프는 `break`문을 만나기 전까지 계속 반복된다.

되풀이 될 때마다 꺽쇠 표시로 사용자에게 입력을 요청한다. 사용자가 `done`이라 쓰면 `break`문이 루프를 끝낸다. 그 외에는 사용자가 무엇을 입력하든 그것을 화면에 표시하고 루프의 처음으로 되돌아 간다. 여기 실행 예시가 있다.

```
> not done
not done
> done
Done!
```

`while` 루프를 이렇게 사용하는 두 가지 이유가 있다. 첫 째는 조건이 루프 내 어디서든(루프 맨위에서 한 번이 아니라) 비교가 되기 때문이다. 두 번째는 종료 조건을 부정적으로("이것이 발생하기 전까지 계속") 표현하는 대신 긍정적으로("이것이 발생하면 멈춤") 표현할 수 있기 때문이다.

7.5 제곱 근

루프는 근사치를 구하고 반복적으로 그 값을 더 정확하게 만드는 수치해석 프로그램에서 주로 쓰인다.

예를 들어 보자. 제곱 근을 구하는 방법 중 하나는 뉴튼의 기법(Newton's method)이다. a의 제곱 근을 알고 싶다고 해보자. 어떤 추정치 x에서 시작하든 다음의 식으로 좀 더 정확한 근사치를 얻을 수 있다.

$$y = \frac{x + a/x}{2}$$

a는 4 x는 3이라고 해보자.

```
>>> a = 4
>>> x = 3
>>> y = (x + a/x) / 2
>>> y
2.16666666667
```

정답에 좀 더 가까운 결과를 얻었다($\sqrt{4} = 2$). 이 과정을 새로운 추정치로 반복한다면 좀 더 정답에 가까워진다.

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00641025641
```

몇 번 더 갱신하면 추정치는 거의 정답에 가까워진다.

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.00001024003
>>> x = y
```

```
>>> y = (x + a/x) / 2
>>> y
2.00000000003
```

일반적으로 이 과정을 몇 번이나 거쳐야 정답을 얻게 되는지 모른다. 그렇지만, 정답에 가까워지면 알게된다. 왜냐하면 추정치의 변화하지 않기 때문이다.

```
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> y
2.0
```

`y == x`이면 멈추면 된다. 다음은 초기 추정치 `x`로부터 변화가 없을 때까지 정확도를 높이는 루프이다.

```
while True:
    print(x)
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

대부분의 `a`는 동작을 하지만, `float`형을 등호에 쓸 때는 조심해야 한다. 부동소수점 값은 근사치이기 때문이다. 1/3처럼 유리수인 경우나 $\sqrt{2}$처럼 무리수인 경우의 대부분은 `float`형으로 그 값을 정확하게 표현될 수 없다는 것을 기억해야 한다.

`float`형인 `x`와 `y`가 서로 정확히 일치하는지 등호로 검사하는 대신 내장 함수인 `abs`를 사용하여 그 수의 절대치 또는 크기를 비교하는 것이 좋다.

```
    if abs(y-x) < epsilon:
        break
```

`epsilon`는 `0.0000001`와 같은 값으로 얼마나 작아야 같다고 할 수 있는지 판단하는데 도움이 된다.

7.6 알고리즘

뉴튼의 기법은 **알고리즘(algorithm)**의 예이다. 어떤 종류의 문제(이 경우에는 제곱 근의 계산)를 해결하기 위한 기계적인 과정이다.

알고리즘이 무엇인지 이해하기 위해서는 알고리즘이 아닌 무엇인가로부터 시작하는 것이 좋겠다. 한 자리 수 숫자로 곱셉하는 법을 배웠을 때 구구단을 외웠을 것이다. 사실상은 100개의 서로 다른 해답을 외웠을 뿐이다. 이런 종류의 지식은 알고리즘적 접근이 전혀 아니다.

당신이 만약 "게으르다면" 요령을 배울 필요가 있다. 예를 들어 어떤 수 n과 9의 곱을 구한다고 해보자. 일의 자리에 $n-1$을 쓰고 십의 자리에 $10-n$을 쓰면 된다. 이 요령은 한 자리 수 숫자와 9의 곱에 대한 일반 해이다. 이런 것이 바로 알고리즘이다!

올림이 있는 덧셈이나 빌려서 뺄셈하기와 나눗셈하는 기술들이 모두 알고리즘이다. 알고리즘이 갖는 특성 중 하나는 지적 능력이 전혀 필요 없다는 것이다. 기계적으로 간단한 규칙에 따라 이전 단계에서 다음 단계를 따르면 되기 때문이다.

알고리즘의 실행은 지겨운 일이지만 설계하는 것은 흥미진진하고 지적으로 도전적이다. 뿐만아니라 컴퓨터 과학의 중심이기도 하다.

사람들이 자연스럽게 어려움 없이 또는 깊이 생각하지 않고 하는 것들 중에는 알고리즘으로 표현하기 매우 어려운 것들이 있다. 자연어 인식이 좋은 예이다. 우리는 하지만 아직까지 *어떻게* 그렇게 할 수 있는지 누구도 설명하지 못했다. 최소한 알고리즘으로 풀어내지는 못했다.

7.7 디버깅

프로그램의 크기가 커지다보면 더 많은 시간을 디버깅에 쓰게 되는 것을 보게 된다. 더 많은 코드는 오류를 만들 확률이 많아지고 버그가 숨을 곳이 더 많아진다는 말이다.

디버깅 시간을 줄이는 방법 중 하나는 "등분으로 디버깅하기"이다. 예를 들어 100 줄짜리 프로그램이 있다고 했을 때 한 줄씩 검사한다면 100번 검사해야 한다.

대신에, 문제를 반으로 쪼개보자. 프로그램의 중간 또는 그 근처에 있는 중간 과정의 값 중 검사할 수 있는 것을 확인해보자. `print`문(증명을 도와줄 어떤 것)을 추가한 후 프로그램을 실행하자.

중간 점검이 부정확하다면 전반부에 오류가 있다는 말이다. 정확하다면 문제는 후반부에 있다.

이런 식으로 검사를 할 때마다 검사해야할 줄의 수를 반씩 줄여나가면 된다. 이론상으로는 이 과정을 6번(100번 보다 작다) 반복하면 검사해야 할 줄의 수가 하나 또는 두 줄 정도가 된다.

실제로는 "프로그램의 중간"이 늘 명확하지도 않고 그 부분을 검사한다는 것이 가능하지도 않다. 정확한 중간 지점을 찾기 위해 라인 수를 세서 반으로 나눈다는 것도 말이 안된다. 대신에 오류가 있을만한 부분을 생각해보고 검사 문장을 쉽게 삽입할 수 있는 위치를 생각해보자. 그리고 검사 문장 삽입 위치 전과 그 후에 오류가 있을 확률이 서로 비슷한 지점에 검사 문장을 삽입하자.

7.8 용어 해설

재할당(reassignment): 이미 존재하는 변수에 새로운 값을 할당하는 것.

갱신(update): 변수의 새로운 값이 이전 값에 의존하는 할당문.

초기화(initialization): 갱신될 변수에 초기 값을 할당하는 것.

증가(increment): 변수의 값을 증가 시키는 갱신(주로 1 증가).

감소(decrement): 변수의 값을 감소 시키는 갱신.

반복(iteration): 재귀 함수 호출이나 루프를 사용하여 문장들을 되풀이하여 실행하는 것

무한 루프(infinite loop): 종료 조건이 절대로 충족되지 않는 루프.

알고리즘(algorithm): 어떤 종류의 문제를 해결하는 일반적인 과정.

7.9 연습 문제

문제 7.1. *7.5 절의 루프문을 복사하여* `mysqrt`*라는 함수로 캡슐화하라. 이 때* `a`*를 매개 변수로하고 타당한* `x`*을 선정하자. 그리고* `a`*의 제곱 근의 추정치를 리턴하도록 하라.*

검사를 위해 `test_square_root`*라는 함수를 만들고 다음과 같이 출력하도록 만들자.*

```
a   mysqrt(a)     math.sqrt(a)  diff
-   ---------     ------------  ----
1.0 1.0           1.0           0.0
2.0 1.41421356237 1.41421356237 2.22044604925e-16
3.0 1.73205080757 1.73205080757 0.0
4.0 2.0           2.0           0.0
5.0 2.2360679775  2.2360679775  0.0
6.0 2.44948974278 2.44948974278 0.0
7.0 2.64575131106 2.64575131106 0.0
8.0 2.82842712475 2.82842712475 4.4408920985e-16
9.0 3.0           3.0           0.0
```

첫 번째 열은 숫자 a이다. 두 번째 열은 `mysqrt`*으로 계산된 a의 제곱근이다. 세 번째 열은* `math.sqrt`*으로 계산된 제곱 근이다. 네 번째 열은 두 추정치의 차이의 절대치를 나타낸다.*

문제 7.2. *내장 함수인* `eval`*은 문자열을 받아 Python 인터프리터를 사용하여 계산한다. 다음 예를 살펴보자.*

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<class 'float'>
```

반복저으로 사용자의 입력을 기다리는 `eval_loop` *함수를 작성해보자. 전달받은 값을* `eval` *함수로 계산하고 그 결과를 출력하도록 만들자.*

사용자가 `'done'`*을 입력하기 전까지 계속 동작하고 종료하면서 마지막 계산한 수식의 결과를 표시하도록 만들자.*

문제 7.3. *스리니바사 라마누잔(Srinivasa Ramanujan)은 무한 급수를 발견하였다. 무한 급수는* $1/\pi$*의 근사치를 구하는데 쓰인다.*

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

위의 공식을 사용하여 π*의 추청지를 계산하여 리턴하는* `estimate_pi` *함수를 작성해보자.* `while` *루프를 사용하여 추정치의 마지막 항이* `1e-15`*(Python에서* 10^{-15}*의 표기)보다 작을 때까지 반복하라. 얻은 결과는* `math.pi`*와 비교해 볼 수 있다.*

해답: ***http://thinkpython2.com/code/pi.py****.*