Chapter 6

결과가 있는 함수

우리가 사용한 수학 함수와 같은 많은 Python 함수들은 리턴 값이 있지만, 우리가 작성한 함수들은 리턴 값이 없다. 대신 값을 출력하거나 거북이를 이동시키는 것과 같은 영향만 있다. 이 장에서는 결과가 있는 함수에 대해 배울 것이다.

6.1 리턴 값

함수 호출에는 리턴 값이 있는데, 보통은 이 값을 변수에 할당하거나 어떤 수식의 일부로 사용한다.

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

지금까지 우리는 결과가 없는 함수들을 작성했었다. 격식 없게는 리턴 값이 없었고, 좀 정확하게는 그 함수들의 리턴 값은 `None`이 었다.

이 장에서는 (마침내) 결과가 있는 함수를 작성할 것이다. 첫 번째 예제는 전달 받은 반지름으로 원의 넓이를 리턴하는 `area` 함수이다.

```
def area(radius):
    a = math.pi * radius**2
    return a
```

우리가 봤던 결과가 있는 함수에서의 `return`문은 수식이 포함되 있었다. 리턴문은 "리턴 값으로 따라오는 수식을 사용하여 즉시 리턴하라"는 의미를 갖고 있다. 리턴할 수식이 복잡해도 괜찮기 때문에 이 예제의 함수를 좀 더 간결하게 작성 해 볼 수도 있다.

```
def area(radius):
    return math.pi * radius**2
```

그렇기는 하지만, **임시 변수**로 사용한 `a`가 있어 좀 더 쉽게 디버깅할 수도 있다.

때로는 조건문의 분기마다 리턴문이 있는 경우도 있다.

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

각 return문이 선택지마다 있기 때문에 그 중 어느 하나만 실행된다.

리턴문이 실행되면 실행 중이던 함수는 다음 문장을 수행하지 않고 즉시 종료한다. return문 이후의 문장이나 실행의 흐름 상 도달하지 않는 코드를 **죽은 코드(dead code)**라 부른다.

결과가 있는 함수에서는 프로그램의 어떤 실행 흐름이더라도 return문에 도달하도록 하는 것이 좋다. 예를 들어보자.

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

이 함수는 잘못되었다. 만약 x가 0이면 두 선택지 모두 거짓이 된다. 그러면 return문에 도달하지 않고 종료하게 된다. 이 경우의 리턴 값은 None이지 0의 절대치가 아니다.

```
>>> print(absolute_value(0))
None
```

참고로, Python의 내부 함수 중에는 절대치를 계산하는 abs 함수가 있다.

연습 문제로 x와 y 두 수를 받아 x > y이면 1을 리턴하고 x == y이면 0을 그리고 x < y이면 -1을 리턴하는 compare 함수를 작성해보자.

6.2 점진적 개발

큰 함수를 작성하다 보면 디버깅에 더 많은 시간을 쓰는 때가 있다.

엄청나게 복잡한 프로그램을 작성하기 위해 **점진적 개발(incremental development)** 방법론을 시도해 볼 것을 권한다. 점진적 개발 방법론의 목적은 아주 작은 크기의 코드를 추가하고 그 부분에 대한 디버깅을 하여 디버깅에 너무 오랜 시간을 쓰는 것을 피하도록 한다.

예를 들어, (x_1, y_1)과 (x_2, y_2)의 좌표를 갖는 두 점 사이의 거리를 잰다고 해보자. 피타고라스의 정리에 따라 거리는 다음과 같이 계산할 수 있다.

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

첫 단계는 Python에서 distance 함수를 정의하는 것이다. 입력 값(매개 변수)은 무엇인지 출력 값(리턴 값)은 무엇인지를 정해야 한다.

이 경우에는 입력 값은 네 개의 수로 이루어진 두 개의 점이다. 리턴 값은 부동소수점으로 표현되는 거리이다.

입력과 출력 값이 정해지면 함수의 구조를 잡을 수 있다.

```
def distance(x1, y1, x2, y2):
    return 0.0
```

당연하겠지만, 이 버전의 함수는 거리를 계산할 수 없다. 이 함수는 언제는 0만 리턴한다. 문법적으로는 정확하고 실행도 가능하다. 좀 더 복잡해지기 전에 검사가 가능하다는 말이다.

함수를 시험해 보려면 샘플 인자 값으로 호출하면 된다.

```
>>> distance(1, 2, 4, 6)
0.0
```

두 점 사이의 수평 거리가 3 차이나고 수직 거리가 4 차이 나도록 점들을 선택했다. 피타고라스의 수에 따라 직각 삼각형의 세 변의 길이의 비가 3:4:5 이기 때문에 결과는 5가 되야 한다. 정답을 알면 함수 검사가 수월해지는 법이다.

이제 이 함수가 문법적으로 정확하다는 것을 확인했으니 내용에 코드를 추가해도 된다. 다음으로 밟아야 할 순서는 $x_2 - x_1$과 $y_2 - y_1$의 계산으로 점 간의 차를 구하는 것이다. 다음 버전의 함수는 차를 계산하고 임시 변수에 저장한 후 화면에 표시하는 것이다.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print('dx는', dx)
    print('dy는', dy)
    return 0.0
```

함수가 동작한다면, dx는 3 그리고 dy는 4를 출력할 것이다. 그리고 그렇게 출력되었다면 인자도 함수에 제대로 전달했고 첫 계산도 정확히 수행했다는 것을 알 수 있다. 만약 잘못 나왔다면 검사해야 할 코드가 몇 줄 되지 않는 것에 위안을 얻으면 된다.

그 다음 과정은 dx와 dy의 제곱의 합을 계산해야 한다.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print('dsquared is: ', dsquared)
    return 0.0
```

마찬가지로, 변경한 코드를 실행해보고 결과가 맞는지 검사해봐야 한다(결과는 25다). 마지막으로 math.sqrt를 써서 결과를 리턴하면 된다.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

지금까지 제대로 동작했다면, 다 끝났다. 확인하기 원한다면 리턴하기 전에 result를 출력하면 된다.

함수의 최종 버전은 실행했을 때 아무 것도 출력하지 않는다. 그저 값만 리턴할 뿐 이다. print 문은 디버깅 때문에 포함한 것이기 때문에 함수가 동작한다면 제거해야 한다. 이런 류의 코드를 보고 **발판(scaffolding)**이라 한다. 프로그램을 작성하는데 도움은 되지만 최종 결과물의 일부는 아니기 때문이다.

처음 시작할 때는 한 두 줄의 코드만 추가해야 한다. 좀 더 익숙해지면 좀 더 긴 길이의 코드를 작성하고 디버깅할 수 있게 된다. 초보이던 전문가이던, 점진적 개발 방법론은 디버깅 시간을 엄청나게 줄일 수 있다.

방법론의 핵심은 다음과 같다.

1. 동작하는 프로그램으로 부터 시작하여 점진적인 변화를 만들어라. 그래야 어떤 순간이라도 오류를 발견하면 어디에 원인이 있는지 쉽게 찾을 수 있다.

2. 중간 값들을 변수에 저장하여 출력해보고 검사 할 수 있게 만들어라

3. 프로그램이 동작한다면 발판으로 사용한 코드를 삭제하고 여러 줄로 작성한 문장들을 복합문으로 통합해라. 단, 통합했을 때 코드 읽기가 너무 어려워지면 안된다.

연습삼아 hypotenuse라는 함수를 만들어 보자. 이 함수는 직각 삼각형의 빗변을 계산하는 함수로 다른 두 변의 길이를 인자로 받는다. 개발 과정의 각 단계를 거칠 때마다 결과를 표시하도록 해보자.

6.3 합성

이제는 이미 알리라 생각이 되지만, 함수 내에서 다른 함수를 호출하는 것이 가능하다. 원의 중심점과 원주 위의 한 점을 받아 원의 면적을 계산하는 함수로 예를 들어 보자.

xc와 yc에 원의 중심 점을 저장하고 원주 위의 한 점은 xp와 yp에 저장된다고 하자. 첫 번째 단계는 이 두 점을 사용하여 원의 반지름을 계산하는 것이다.이미 작성한 distance 함수를 사용하자.

```
radius = distance(xc, yc, xp, yp)
```

다음으로 반지름을 사용하여 원의 면적을 구하자. 이것도 이미 작성했었다.

```
result = area(radius)
```

이 모든 단계들을 한 함수로 캡슐화 해보자.

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

임시 변수 radius와 result는 개발과 디버깅하는 과정에서만 유용하기 때문에 프로그램이 제대로 동작하면 다음과 같이 간결하게 함수 호출을 합성할 수 있다.

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6.4 Boolean 함수

함수는 bool 값을 리턴할 수 있다. 그렇게 하면 복잡한 검사문을 함수에 감출 수 있기 때문에 유용하다. 예를 살펴 보자.

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

Boolean 함수의 이름은 참/거짓을 묻는 질문처럼 만드는 것이 일반적이다. is_divisible은 x가 y로 나누어 떨어지는지 True 또는 False로 리턴한다.

사용 예를 보자.

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

== 연산자의 결과는 참 또는 거짓이기때문에 그 결과를 바로 리턴하도록 함수를 간결하게 작성할 수 있다.

```
def is_divisible(x, y):
    return x % y == 0
```

Boolean 함수는 조건문에 많이 쓰인다.

```
if is_divisible(x, y):
    print('x is divisible by y')
```

아래와 같이 쓰고 싶은 유혹이 들 수 있다.

```
if is_divisible(x, y) == True:
    print('x is divisible by y')
```

두 번 검사는 불필요하다.

연습삼아 `is_between(x, y, z)` 함수를 작성해보자. 이 함수는 $x \leq y \leq z$이면 `True`를 아니면 `False`를 리턴한다.

6.5 또 다시 재귀문

Python의 일부만을 다뤘지만, 지금까지 다룬 내용이 프로그래밍 언어의 *전부*라는 사실을 알았으면 좋겠다. 지금의 언어로 연산하고 싶은 어떤 것이든 다 표현할 수 있다. 누군가가 만든 프로그램도 지금까지 배운 것들만으로 다시 만들 수 있다(사실 마우스, 디스크같은 것들을 다루려면 다른 명령어들이 필요하겠지만 그게 다다).

이 주장을 증명하는 것은 쉽지 않은 일이긴 하지만 최초의 컴퓨터 과학자인 앨런 튜링(Alan Turing)이 증명했다(그가 수학자라고 주장하는 사람들이 있기는 하지만, 초창기의 대부분의 컴퓨터 과학자들인 수학자로 시작했다). 그의 이름을 따서 튜링 명제라 한다. 이에 관한 완전하고(정확한) 논고는 마이클 싶서(Michael Sipser)의 책 *Introduction to the Theory of Computation*을 읽어보기를 바란다.

지금까지 배운 도구들로 무엇을 할 수 있는지 보이기 위해 재귀적으로 정의된 수학 함수를 살펴보도록 하겠다. 정의하는 대상을 정의에서 다시 사용한다는 점에서 재귀적 정의는 순환 정의와 유사하다. 순환 정의는 그렇게 도움이 되지 않는다.

vorpal: vorpal한 것을 나타내는 형용사

사전에서 이런 단어를 만나면 짜증이 날 것이다. 반면, !로 표기하는 계승(factorial) 함수는 다음과 같이 정의 된다.

$$0! = 1$$
$$n! = n(n-1)!$$

이 정의는 0의 계승은 1이고 어떤 값 n의 계승은 n 곱하기 $n-1$의 계승으로 정의된다.

그래서, 3!은 3 곱하기 2!이고, 다시 2곱하기 1!, 그리고 1 곱하기 0!이 된다. 정리하면 3!은 3 곱하기 2 곱하기 1 곱하기 1이고, 계산하면 6이다.

어떤 것을 재귀문으로 정의하였다면 Pytho 프로그램을 써서 평가해면 된다. 먼저, 매개 변수를 정한다. `factorial`의 경우 당연히 정수를 전달해야 한다.

```
def factorial(n):
```

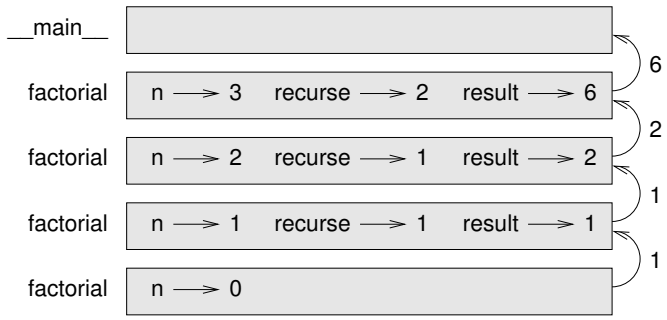

Figure 6.1: 스택 상태도

인자가 0이라면 1을 리턴하면 된다.

```
def factorial(n):
    if n == 0:
        return 1
```

그 외의 경우를 다루는 방법이 흥미로운데, 재귀 호출을 써서 $n-1$의 계승을 계산한 후 n으로 곱하면 된다.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

이 프로그램의 실행의 흐름은 5.8장에서 본 `countdown`의 실행 흐름과 비슷하다. `factorial`의 인자로 3을 주고 시작하면 다음과 같이 동작한다.

3은 0이 아니기 때문에 두 번째 분기를 선택하여 `n-1`의 계승을 계산한다.

> 2은 0이 아니기 때문에 두 번째 분기를 선택하여 `n-1`의 계승을 계산한다.
>
> > 1은 0이 아니기 때문에 두 번째 분기를 선택하여 `n-1`의 계승을 계산한다.
> >
> > > 0은 0이기 때문에 더 이상 재귀 호출을 하지 않고 첫 번째 분기를 선택하여 1을 리턴한다.
> >
> > 리턴 값 1을 n과 곱한 결과를 리턴한다. 이 때, n은 1이다.
>
> 리턴 값 1을 n과 곱한 결과를 리턴한다. 이 때, n은 2이다.

리턴 값 2을 n과 곱한 결과를 리턴한다. 이 때, n은 3이다. 계산 결과 6이 처음 재귀 호출 과정을 시작한 함수의 리턴 값이 된다.

그림 6.1이 함수의 실행 흐름에 따른 스택 상태도를 나타낸다.

리턴 값이 스택 위로 전달되는 것을 볼 수 있다. 각 프레임에 나타난 리턴 값은 `n`과 `recurse`의 곱인 `result`이다.

마지막 프레임에는 `recurse`와 `result`가 없다. 이 변수들을 생성하는 분기가 실행되지 않았기 때문이다.

6.6 믿음의 도약

실행 흐름에 따라 프로그램을 읽는 것도 한 방법이기는 하지만, 코드의 길이에 압도될 수도 있다. 그 대안으로 "믿음의 도약(leap of faith)"을 하는 방법을 소개하겠다. 함수 호출을 만나면 실행 흐름대로 그 함수의 내용을 다 읽는 대신, 그 함수가 제대로 동작하고 리턴 값도 제대로 전달한다고 가정 하자.

사실, 내장 함수은 이미 제대로 동작한다는 믿음을 갖고 사용했었다. `math.cos`나 `math.exp`를 호출 했을 때 그 함수의 내용을 일일이 확인해보지 않았다. 그저 내장 함수들은 훌륭한 프로그래머가 작성했을 것이라 믿고 동작할 것이라 생각했다.

자신이 작성한 함수도 똑같은 믿음을 갖고 사용해도 된다. 6.4절에서 작성한 `is_divisible`을 떠올려 보자. 이 함수가 정확하다고 믿은 순간—코드 확인과 검사를 거친 후—그 함수의 내용을 다시 읽어보지 않고 사용했다.

재귀 프로그램도 다르지 않다. 재귀 호출을 만나면 실행 흐름을 쫓는 대신 재귀 호출이 제대로 동작한다고 믿어보자(그리고 정확한 결과도 리턴한다고 해보자). 그리고 "$n-1$의 계승을 구할 수 있다고 한다면 n의 계승도 계산할 수 있을까?"라고 물어야 한다. n만 곱하면 되기 때문에 당연히 가능하다.

물론, 작성을 다 하지 않은 함수를 제대로 동작한다고 가정하는게 이상할 수 있지만 그렇기 때문에 믿음의 도약이라 부르는 것이다.

6.7 또 하나의 예

`factorial`을 살펴보았는데, 가장 흔한 재귀적으로 정의한 수학 함수는 `fibonacci`이다. 정의는 다음과 같다(참고: `http://en.wikipedia.org/wiki/Fibonacci_number`).

$$\begin{aligned}\text{fibonacci}(0) &= 0\\ \text{fibonacci}(1) &= 1\\ \text{fibonacci}(n) &= \text{fibonacci}(n-1) + \text{fibonacci}(n-2)\end{aligned}$$

Python으로 해석하면 다음과 같다.

```
def fibonacci(n):
    if n == 0:
        return 0
    elif  n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

아무리 작은 n이라 할지라도 이 함수의 실행 흐름을 따라가려면 머리가 터질 것이다. 믿음의 도약을 한다고 생각하고 두개의 재귀 호출이 제대로 동작한다고 믿어보자. 그러면 결과들을 더했을 때 올바른 값을 얻는다는 것이 명확해질 것이다.

6.8 데이터 형 검사

`factorial`을 호출 할 때 인자가 1.5면 어떻게 될까?

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

무한 재귀문 오류인 것처럼 보인다. 어째서 그럴가? `n == 0`이라는 기준 케이스도 있는데 말이다. `n`이 정수가 아니면 기준 케이스에 도달 못하고 영원히 재귀 호출을 하게 된다.

첫 번째 재귀 호출에서 `n`의 값은 0.5이다. 그 다음에는 -0.5가 된다. 그 이후에는 수는 작아지기만 하지 절대로 0은 안된다.

두 가지 선택이 있다. `factorial` 함수를 일반화해서 부동소수점 수도 계산할 수 있게 하거나 `factorial`이 인자의 데이터 형을 검사할 수도 있다. 첫 번째는 감마 함수라고 부르고 이 책이 다룰 수 있는 범위 밖에 있다. 그러니 두 번째 것을 시도해보자.

내부 함수 중 `isinstance`을 사용해서 인자의 데이터 형을 검사할 수 있다. 그와 동시에 인자로 받은 값이 양수인지도 확인할 수 있다.

```
def factorial(n):
    if not isinstance(n, int):
        print('정수에 대해서만 정의되어 있음.')
        return None
    elif n < 0:
        print('음의 정수에 대해서는 정의 안됨.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

첫 기준 케이스는 정수가 아닌 경우를 다루고 두 번째는 음의 정수를 다룬다. 두 경우다 프로그램은 오류 메시지를 출력 하고 무언가 잘못되었다는 것을 표시하기 위해 `None`을 리턴한다.

```
>>> print(factorial('fred'))
정수에 대해서만 정의되어 있음.
None
>>> print(factorial(-2))
음의 양수에 대해서는 정의 안됨.
None
```

두 검사를 통과했다면, *n*은 양수의 정수이거나 0이라는 것을 알기 때문에 재귀문이 종료한다는 것을 증명할 수 있다.

이 프로그램은 **가디언(guardian**, 보호자**)**이라는 패턴을 사용하였다. 첫 두 조건문이 오류가 발생할만한 값들로부터 코드를 보호하는 가디언 역할을 한다. 가디언 패턴으로 코드의 정확성을 증명할 수 있다.

11.4절에서는 오류 메시지를 출력하는 것보다 좀 더 유연한 대안인 엑셉션(exception, 예외) 처리에 대해 살펴 볼 것이다.

6.9 디버깅

큰 프로그램을 작은 함수들로 나누어 작성하는 것은 디버깅을 할 수 있는 지점들을 자연스럽게 만드는 효과가 있다. 함수가 동작하지 않을 때 검사해 볼 세가지가 있다.

- 함수가 전달 받는 인자에 문제가 있다. 사전 조건이 잘못되었다.

- 함수에 문제가 있다. 사후 조건이 잘못되었다.
- 리턴 값 또는 그 값을 사용하는데 문제가 있다.

첫 가능성을 제거하려면 함수 시작부분에 print문을 써서 매개 변수의 값(그리고 데이터 형)을 출력해 보면 된다. 또는 사전 조건이 정확한지 검사하는 코드를 명시적으로 작성하면 된다.

매개 변수가 맞아 보이면 모든 return문 전에 print문을 삽입하여 리턴 값을 검사해보자. 함수 호출의 결과를 수기로도 검산해보자. 검산이 쉬운 값을 넣어 함수를 호출해보자(6.2절을 참고하자).

함수가 제대로 동작하는 것처럼 보이면 호출하는 문장에서 리턴 값을 정확하게 사용하는지(또는 사용은 하는지) 확인해보자.

함수가 시작할 때 그리고 끝날 때 print문을 넣어보면 실행 흐름을 시각화하는데 도움이 된다. 다음은 factorial에 print문을 넣은 버전이다.

```
def factorial(n):
    space = ' ' * (4 * n)
    print(space, 'factorial', n)
    if n == 0:
        print(space, 'returning 1')
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print(space, 'returning', result)
        return result
```

space는 공백 문자열로 결과의 들여쓰기 정도를 나타낸다. 다음은 factorial(4)의 결과이다.

```
                factorial 4
            factorial 3
         factorial 2
     factorial 1
 factorial 0
 returning 1
     returning 1
         returning 2
             returning 6
                 returning 24
```

실행 흐름이 혼란스럽다면 이런 식으로 결과를 표시하는 것도 도움이 된다. 효과적인 발판을 만드는 것은 시간이 걸리지만 작은 발판이라할지라도 디버깅 시간을 크게 줄일 수 있다.

6.10 용어 해설

임시변수(temporary variable): 복잡한 계산의 중간 값을 저장하는데 사용하는 변수

죽은 코드(dead code): 대체적으로 return문 다음에 오는 코드로 프로그램에서 실행되지 않는 부분

점진적 개발(incremental development): 한 번에 작은 크기의 코드를 추가하고 검사하여 디버깅 시간을 줄이려는 프로그램 개발 계획

발판(scaffolding): 프로그램 개발 중에는 사용되나 최종 버전에는 제외되는 코드

가디언(보호자, guardian): 오류를 발생시킬 수도 있는 상황을 조건문으로 처리하거나 검사하는 프로그래밍 패턴

6.11 연습 문제

문제 6.1. *다음 프로그램의 스택 상태도를 그려라. 프로그램은 무엇을 표시하는가?*

```
def b(z):
    prod = a(z, z)
    print(z, prod)
    return prod

def a(x, y):
    x = x + 1
    return x * y

def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square

x = 1
y = x + 1
print(c(x, y+3, x+y))
```

문제 6.2. *아커만 함수 $A(m,n)$은 다음과 같이 정의 되었다.*

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

상세 설명은 http://en.wikipedia.org/wiki/Ackermann_function을 참고하라. 아커만 함수를 계산하는 `ack` *함수를 작성하라.* `ack(3, 4)`*의 값을 구하고 125가 맞는지 확인하라.* `m`*이나* `n`*이 더 큰 값이면 어떻게 되는가? 해답은 http://thinkpython2.com/code/ackermann.py에 있다.*

문제 6.3. *회문(pallindrome)은 "noon"과 "redivider"처럼 단어의 순방향 철자가 역방향 철자와 동일한 단어를 뜻한다. 재귀적으로 표현했을 때, 첫 글자와 마지막 글자가 같고 중간 부분이 회문이면 그 단어는 회문이다.*

다음의 함수들은 문자열을 인자로 받아 첫 글자, 마지막 글자, 그리고 중간의 글자들을 리턴한다.

```
def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

동작 방식은 8장에서 살펴볼 것이다.

1. `palindrome.py`*라는 이름을 함수들을 저장하고 검사해보자. 두 글자 길이의 단어를* `middle` *함수에 인자로 전달하면 어떻게 되는가? 한 글자이면 어떻게 되는가? 빈 문자열* `''`*을 인자로 전달하면 어떻게 되는가?*

2. *문자열을 인자로 받아 그 문자열이 회문이면* `True`*를 아니면* `False`*를 리턴하는* `is_palindrome` *함수를 작성하라. 참고로 문자열의 길이를 확인하기 위해 내장 함수인* `len`*을 쓸 수 있다.*

해답: `http://thinkpython2.com/code/palindrome_soln.py`.

문제 6.4. *어떤 수 a는 b로 나눌 수 있고 a/b가 b의 거듭제곱이면 a는 b의 거듭제곱이다.* `a`*와* `b`*를 매개 변수로 받아* `a`*가* `b`*의 거듭제곱이면* `True`*를 리턴하는* `is_power` *함수를 작성하라. 참고: 기준 케이스 선정이 중요하다.*

문제 6.5. *a와 b의 최대 공약수(the Greatest Common Divisor, GCD)는 나머지 없이 두 수를 나눌 수 있는 가장 큰 수이다.*

최대 공약수를 찾는 한 방법은 a을 b로 나눴을 때의 나머지 r에 대해 $gcd(a,b) = gcd(b,r)$*의 관계가 성립한다는 관찰을 사용하면 된다. 기준 케이스로* $gcd(a,0) = a$*를 사용하라.*

매개 변수 `a`*와* `b`*를 사용하고 최대 공약수를 리턴하는* `gcd` *함수를 작성하라.*

출처: 애벨슨(Abelson)과 서스만(Sussman)의 Structure and Interpretation of Computer Programs*의 예제를 참고하였다.*