Chapter 5

조건문과 재귀문

프로그램의 상태에 따라 다른 코드를 실행하는 `if`문이 이 장의 핵심 주제이다. 그 전에 내림 나눗셈 연산자와 나머지 연산자를 살펴보자.

5.1 내림 나눗셈과 나머지 연산자

내림 나눗셈 연산자 `//`는 두 수를 나누어 얻은 결과 값의 정수 값을 취한다. 예를 들어, 105분 길이의 영화가 있다고 했을 때 몇 시간 짜리 영화인지 궁금할 때가 있다. 이런 경우 일반 나눗셈을 사용하면 부동소수점 값을 얻는다.

```
>>> minutes = 105
>>> minutes / 60
1.75
```

하지만, 일반적으로 시간을 소수점으로 표현하지 않는다. 내림 나눗셈은 소수점 부분을 버리고 정수부분만 리턴한다.

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

나머지를 원하면 60분에서 계산된 시간 값을 제외한 나머지 분을 빼면 된다.

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```

이런 복잡한 계산 대신 나눗셈 후 나머지 값을 리턴하는 **나머지 연산자** `%`를 사용할 수 있다.

```
>>> remainder = minutes % 60
>>> remainder
45
```

나머지 연산자는 보기보다 매우 유용한다. 한 가지 예를 들어 보면, 나머지 연산자는 어떤 값이 서로 나누어 떨어지는 지 확인할 때 쓸 수있다. `x % y`가 영이라면 `x`는 `y`로 나누어 떨어진다.

또한, 어떤 숫자의 가장 오른쪽 자리수를 얻을 때 나머지 연산자를 쓸 수 있다. `x % 10`을 계산하면 `x`(십진수라 가정)의 1의 자리 값을 얻는다. 마찬가지로 `x % 100`을 하면 마지막 두 자리 수를 얻는다.

Python 2를 사용 중이라면 나눗셈은 다르게 동작한다. 나눗셈 연산자 /은 두 수가 모두 정수이면 내림 나눗셈을하고 두 값 중 하나라도 실수 형면 부동소수점 나눗셈을 한다.

5.2 Boolean 표현식

Boolean 표현식(boolean expression)은 참또는 거짓을 나타내는 표현식이다. 다음의 예들은 두 피연산자를 비교하는 == 연산자를 사용한다. 두 값이 동일하면 `True(참)`을 다르면 `False(거짓)`이다.

```
>>> 5 == 5
True
>>> 5 == 6
False
```

이 값은 문자열이 아니고, `bool(불)` 데이터 형에 속한`True`와 `False`는 특별한 값이다.

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

== 연산자는 **관계 연산자(relational operator)** 중 하나이다. 나머지는 다음과 같다.

```
x != y               # x 는 y와 같지 않음
x > y                # x 는 y보다 큼
x < y                # x 는 y보다 작음
x >= y               # x 는 y보다 크거나 같음
x <= y               # x 는 y보다 작거나 같음 y
```

이 연산자들이 익숙해 보일 수 있지만, Python에서 사용하는 기호가 수학 기호와 다르다. 가장 흔한 실수는 하나의 등호(=)를 써서 두 값이 같은지 비교(==)하는 것이다. =는 할당 연산자이고 ==가 관계 연산자인 것을 기억해야 한다. 그리고 =<나 =>와 같은 연산자는 없다.

5.3 논리 연산자

세 개의 **논리 연산자(logical operator)**가 있다. 각각은 `and`, `or`, 그리고 `not`이다. 이 연산자들의 의미는 영어에서의 각 단어의 의미와 같다. 예를 들어, `x > 0 and x < 10` 은 `x`가 0보다 *크고* 10보다 10보다 작을 때 참이 된다.

`n%2 == 0 or n%3 == 0` 은 *둘 중 하나만* 또는 *둘 다* 참일 때 참이 된다. 즉, 2 또는 3으로 나누어지거는 경우이다.

마지막으로 `not` 연산자는 boolean 표현식의 반대 값을 취한다. `not (x > y)`는 `x > y`가 거짓일 때 참이 된다. 즉, `x`가 `y`보다 작거나 같은 경우이다.

정확하게 말하면, 논리 연산자의 피연산자는 boolean 표현식이어야하지만, Python이 그렇게 엄격하지는 않다. 0이 아닌 어떤 수이기만 하면 `True`으로 인식한다.

```
>>> 42 and True
True
```

이러한 유연성이 도움이 될 때도 있지만, 미묘한 부분도 있기 때문에 혼돈이 있을 수도 있다. 이렇게 사용하는 것은 피하는게 좋다(뭘 하는지 정확히 알고 있다면 모를까).

5.4 조건부 실행

프로그램을 유용하게 만들려면 프로그램이 상황에 맞게 동작할 수 있도록 조건을 검사하는 능력이 반드시 필요하다. **조건문(conditional statement)**이 바로 그 능력을 갖고 있다. 그 중 가장 간단한 것은 `if`문이다.

```
if x > 0:
    print('x is positive')
```

`if`문 이후의 boolean 표현식은 **조건(condition)**이라 부른다. 참이라면, 의도한 문장이 실행되고 그렇지 않다면 아무 일도 일어 나지 않는다.

헤더와 들여쓰기된 내용으로 구성된 함수 정의처럼 `if`문도 헤더와 내용을 갖고 있다. 이런 류의 문장들을 **복합문(compound statement)**라고 부른다.

내용에 포함될 수 있는 문장의 수가 정해지지 않았지만 최소한 하나는 있어야 한다. 때로는 어떤 문장도 없는 내용이 필요할 때도 있다(아직 적지 않은 코드를 대신해서 자리만 차지하는 기호를 쓸 수도 있다). 그 때에는 아무 일도 하지 않는 `pass`문을 쓰면 된다.

```
if x < 0:
    pass          # TODO: 음수 처리해야 함!
```

5.5 선택적 실행

`if`문의 두 번째 형식은 "선택적 실행(aternative execution)"이다. 두 가지 대안이 존재하고 조건 검사를 통해 대안 중 하나를 선택하여 실행한다. 문법은 다음과 같다.

```
if x % 2 == 0:
    print('x는 짝수')
else:
    print('x는 음수')
```

`x`를 2로 나눈 나머지가 0이라면 `x`가 짝수이고, 프로그램은 걸맞는 메시지를 표시한다. 조건이 거짓이라면, 두 번째 부분의 문장이 실행된다. 조건은 참이거나 거짓이기 때문에 둘 중에 정확히 하나만 선택적으로 실행된다. 실행의 흐름이 분리되므로 이러한 종류의 선택지를 **분기(branch)**라고 부른다.

5.6 연쇄 조건문

두 개 이상의 가능성이 있는 경우 두개 이상의 분기가 필요하다. 이런 류의 연산을 **연쇄 조건문(chained conditional)**이다.

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

`elif`는 "else if"의 약어이다. 다시 한 번 말하지만, 선택된 분기가 실행된다. `elif`문이 몇 개가 있던 상관없다. `else` 절을 만나면 거기가 마지막이지만, 꼭 필요한 것은 아니다.

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

각 조건은 순차적으로 검사된다. 처음 조건이 거짓이면 다음 조건을 검사하는 식이다. 참인 조건을 만나면 해당 분기가 실행되고 if문은 종료한다. 여러 개의 조건이 참이 될 수는 있는지 첫 번째 참이 조건만 실행이 된다.

5.7 중첩된 조건문

조건문은 다른 조건문 내에 포함될 수 있다. 전 절에서 봤던 예제를 다음과 같이 적을 수도 있다.

```
if x == y:
    print('x and y are equal')
else:
    if x < y:
        print('x is less than y')
    else:
        print('x is greater than y')
```

처음 만나는 조건문은 분기가 두개다. 첫 번째 분기는 간단한 printf문이다. 두 번째 분기는 또 다른 if문을 포함하고 있다. 내부의 if문도 두 개의 분기를 갖고 있다. 각 분기의 내용도 간단한 문장이지만, 이 문장들도 또 다른 조건문이었을 수도 있다.

문장 들여쓰기를 하면 구조가 시각적으로 명확해지긴 하지만, **중첩 조건문(nested conditionals)**을 사용하면 읽기가 어려워진다. 가능하면 쓰지 않는 방법을 찾는 것이 좋다.

논리 연산자를 쓰면 중첩된 조건문이 간단해지는 경우가 많다. 다음의 코드를 하나의 조건문으로 바꿔보자.

```
if 0 < x:
    if x < 10:
        print('x is a positive single-digit number.')
```

print문은 두 개의 조건문을 모두 통과 했을 때만 실행이 된다. 이와 같은 경우에 and 연산자를 쓰면 동일한 효과를 볼 수 있다.

```
if 0 < x and x < 10:
    print('x is a positive single-digit number.')
```

이와 같은 조건문에 Python은 더 간단한 표기법을 제공한다.

```
if 0 < x < 10:
    print('x is a positive single-digit number.')
```

5.8 재귀문

어느 한 함수가 또 다른 함수를 부르는 것은 전혀 이상하지 않다. 그렇기 때문에 어떤 함수가 자기 자신을 부르는 것도 가능하다. 함수가 자신을 부르는 것이 유용한 것인지 확신이 서지 않을 수도 있다. 나중에 알게되겠지만, 프로그램이 할 수 있는 가장 마법같은 일 중 하나가 재귀적 호출이다. 예로 다음의 함수를 살펴보자.

```
def countdown(n):
    if n <= 0:
        print('발사!')
    else:
        print(n)
        countdown(n-1)
```

n이 0이거나 음수이면 "발사!"고 표시하고 그 외의 경우에는 n을 화면에 적고 n-1을 인자로하여 자기 자신(countdown)을 호출한다.

다음과 같이 함수를 호출하면 어떻게 될까?

```
>>> countdown(3)
```

countdown은 n=3에서 실행된다. n은 0보다 크기 때문에 3을 출력하고 자신을 호출한다.

> countdown은 n=2에서 실행된다. n은 0보다 크기 때문에 2을 출력하고 자신을 호출한다.
>
> > countdown은 n=1에서 실행된다. n은 0보다 크기 때문에 1을 출력하고 자신을 호출한다.
> >
> > > countdown은 n=0에서 실행된다. n은 0이기 때문에 "발사!"라고 화면에 표시하고 리턴한다.
>
> n=2을 받은 countdown이 리턴한다.

n=3을 받은 countdown이 리턴한다.

그리고 나면, __main__으로 돌아온다. 최종적으로 다음과 같이 표시된다.

```
3
2
1
발사!
```

함수가 자기 자신을 호출하는 것을 보고 **재귀적(recursive)**이라고 한다. **재귀문(recursion)**은 함수를 재귀적으로 실행시키는 과정을 말한다.

또 다른 예로, 어떤 문자열을 n번 표시하는 함수를 작성해보자.

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```

n <= 0이면 **리턴문(return statement)**으로 함수가 종료된다. 실행의 흐름은 호출한 함수에게 즉시 리턴된다. 그리고 함수의 나머지 부분은 실행되지 않는다.

함수의 나머지 부분은 countdown과 유사하다. s를 표시하고 $n - 1$번 s를 출력하도록 자기 자신을 다시 호출한다. 결과로 표시되는 줄의 수가 1 + (n - 1)이기 때문에 총 n번이 된다.

이처럼 간단한 예에서는 for 루프를 사용하는게 더 쉬울 수 도 있다. 하지만 우리가 이후에 보게 될 좀 더 복잡한 예제들은 for 루프로 표현하기가 어렵다. 어려운 거라면 미리 시작해서 익숙해지는 것이 좋다.

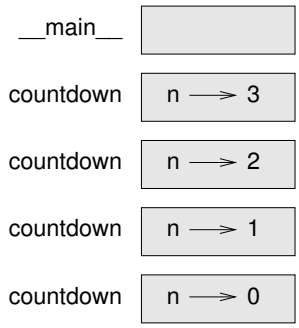

Figure 5.1: 스택 상태도.

5.9 재귀 함수의 스택 상태도

3.9절에서 함수 호출 중에 프로그램의 상태를 스택 상태도로 표현하였었다. 똑같은 그림을 재귀문을 이해하는데도 쓸 수 있다.

함수가 호출될 때마다 Python은 프레임을 생성하여 함수의 지역 변수와 매개 변수를 저장한다. 재귀문을 쓰게 되면 스택에 하나 이상의 프레임이 동시에 여러 개가 존재할 수 있다.

그림 5.1은 `countdown`을 `n = 3`을 인자로 하여 호출했을 때의 스택 상태도를 나타낸다.

스택의 최상위 프레임에 `__main__`가 있다. 프레임에 아무 것도 없는 이유는 `__main__`가 어떤 변수도 생성하지 않았고 인자도 전달받지 못했기 때문이다.

네 개의 `countdown` 프레임들은 매개 변수 `n`이 모두 다르다. `n = 0`이 있는 스택 프레임을 **기준 케이스(base case)**라 부른다. 더 이상 재귀 호출을 하지 않기 때문에 그 이하로는 프레임이 없다.

연습삼아 `s = 'Hello'`와 `n=2`를 인자로 `print_n`가 호출되었을 때의 스택 상태도를 그려보자. 그리고 함수 객체와 `n`을 인자로 하는 `do_n` 함수를 만들어서 그 함수를 `n`번 호출해 보자.

5.10 무한 재귀문

재귀문이 기준 케이스에 도달하지 못한다면 재귀문은 끝없이 호출되고 프로그램은 절대로 종료하지 않는다. 이것이 **무한 재귀문(infinite recursion)**이고 일반적으로 나쁘다. 무한 재귀문의 한 짧은 예를 살펴보자.

```
def recurse():
    recurse()
```

대부분의 프로그래밍 환경에서 무한 재귀문을 갖는 프로그램이 무한히 동작하는 일은 없다. 프로그램이 최대 재귀 가능 횟수에 도달하면 Python은 오류 메시지를 보고 한다.

```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                  .
                  .
                  .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

이 트레이스백은 이 전 장에서 살펴보았던 보다 좀 더 길다. 오류가 발생하면서 스택에 1,000개의 `recurse` 프레임이 생겼다.

실수로 무한 재귀문을 만나게 되면 재귀 호출에 기준 케이스가 있는지 함수를 확인해야 한다. 기준 케이스가 있다면 그 부분에 도달 가능한지 확인해야 한다.

5.11 키보드 입력

지금까지 작성한 모든 프로그램은 사용자 입력이 없었다. 실행하면 똑같은 동작만 했다.

Python은 `input`이라는 내장 함수를 제공한다. 이 함수는 잠시 멈춰서 사용자가 입력하기를 기다린다. 사용자가 Return 또는 Enter키를 누르면 프로그램은 계속 동작하고 `input`은 사용자가 입력한 문자열을 리턴한다. Python 2에서는 똑같은 일을 하는 함수를 `raw_input`이라 부른다.

```
>>> text = input()
What are you waiting for?
>>> text
'What are you waiting for?'
```

사용자로 부터 입력을 받기 전에 어떤 것을 입력할지 알려주는 것이 좋다. `input`는 인자로 프롬프트의 내용을 갖는다.

```
>>> name = input('당신은 누구죠?\n')
당신은 누구죠?
영국의 왕, 아서!
>>> name
'영국의 왕, 아서!'
```

프롬프트의 마지막 부분의 `\n`은 **새 줄(newline)**을 나타내는 특수 기호로 줄을 바꾼다. 그렇기 때문에 사용자 입력 위치가 프롬프트 아래에 있다.

정수를 입력하기를 바란다면 리턴 값을 `int`으로 변환할 수 있다.

```
>>> prompt = '짐을 메달고 있지 않은 제비의 속도는?\n'
>>> speed = input(prompt)
짐을 메달고 있지 않은 제비의 속도는?
42
>>> int(speed)
42
```

하지만 사용자가 숫자 말고 다른 것을 입력하면 오류가 발생한다.

```
>>> speed = input(prompt)
짐을 메달고 있지 않은 제비의 속도는?
그 제비가 아프리카 제비야 아니면 영국 제비야?
>>> int(speed)
ValueError: invalid literal for int() with base 10
```

이런 류의 오류를 다루는 법을 나중에 보도록 하겠다.

5.12 디버깅

문법 오류나 실행 중에 발생하는 오류에는 많은 정보가 담겨 있다. 때로는 벅차게 많을 수도 있다. 그 중에 중요한 정보들은 다음과 같다.

- 어떤 종류의 오류인가
- 어디서 발생했는가

문법 오류는 대체적으로 찾기가 쉽지만 몇 개는 눈에 잘 안 띄기도 한다. 사이띄기와 탭 문자 같은 공백과 관련한 오류들은 보이지 않기 때문에 쉽게 지나치곤 한다. 그래서 찾기가 쉽지 않다.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

이 예제에서의 문제는 두 번째 줄이 한 칸 들여쓰기되어 있다는 것이다. 그렇지만 오류 메시지는 y를 가리키고 있기 때문에 오해하기 쉽다. 일반적으로 오류 메시지는 오류가 발견된 시점을 나타내기 때문에 실제 오류는 코드의 이전에서 이미 있었을 수도 있다. 심지어는 전 줄에 오류가 있었을 수도 있다.

실행 중 발생 오류에서도 마찬가지다. 신호대잡음비를 데시벨로 계산한다고 해보자. 계산식은 $SNR_{db} = 10\log_{10}(P_{signal}/P_{noise})$이다. Python에서는 이 식을 작성하면 다음과 같다.

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

이 프로그램을 실행하면 예외처리가 된다.

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

이 오류 메시지는 5번 줄을 가리키고 있지만 그 줄에는 문제가 없다. 실제 오류를 찾으려면 `ratio`의 값을 출력해보는게 좋다. 출력해보면 0 값을 리턴 받는다. 실제 문제는 내림 나눗셈을 하는 4번 줄에 있다. 소수점 단위의 나눗셈을 했어야 했다.

오류 메시지를 읽는데 충분한 시간을 들여야 하겠지만, 그 하는 말을 모두 참이라 믿어서는 안 된다.

5.13 용어 해설

내림 나눗셈(floor division): //로 표시되는 연산자로서 두 수를 나눈 결과를 정수가 되도록 소수점을 버림

나머지 연산자(modulus operator): 퍼센트(%) 기호를 갖는 연산자로 두 정수를 나눈 나머지 값을 돌려줌

Boolean 표현식(boolean expression): 결과가 항상 참(`True`)이거나 거짓(`False`)인 표현식

관계 연산자(relational operator): 피연산자를 비교하는 연산자로서 다음의 종류가 있음: ==, !=, >, <, >=, <=.

논리 연산자(logical operator): Boolean 표현식들을 서로 연결하는 연산자: `and`, `or`, `not`이 있음

조건문(conditional statement): 어떤 조건에 따라 실행의 흐름을 결정하는 문장

조건(condition): 조건문에서 어떤 분기가 실행될지를 결정하는 boolean 표현식

복합문(compound statement): 헤더와 내용으로 구성된 문장으로 헤더는 콜론(:)으로 끝이 나나. 복합문의 내용은 헤더 위치에서 상대적으로 들여쓰기함

분기(branch): 조건문에서 선택적으로 실행되는 일련의 문장

연쇄 조건문(chained conditional): 여러 개의 선택적 분기문으로 구성된 조건문

중첩 조건문(nested conditional): 어떤 조건문의 분기 내있는 또 다른 조건문

리턴문(return statement): 함수를 호출한 문장으로 즉시 리턴하는 문장

재귀문(recursion): 현재 실행 중인 함수를 호출하는 과정

기준 케이스(base case): 재귀문에서 재귀 호출을 하지 않는 조건 분기

무한 재귀문(infinite recursion): 기준 케이스가 없거나 절대 도달하지 못하는 재귀문. 무한 재귀문을 실행하면 실행 중 오류가 발생하게 됨

5.14 연습 문제

문제 5.1. `time` *모듈은* `time` *함수를 제공한다. 임의의 기준 시점에서 시작하는 "에포크(the epoch)"를 기준으로 현재 그린위치 평균시를 리턴함. UNIX 시스템에서는 에포크는 1970 1월 1일이다.*

```
>>> import time
>>> time.time()
1437746094.5735958
```

현재 시간을 읽어서 시, 분, 초, 그리고 에포크로부터 현재까지 지난 날 수를 리턴하는 스크립트를 작성하라.

문제 5.2. *페르마의 마지막 정리는 n이 2보다 큰 경우 a, b, c의 양의 정수에 대해 다음의 방정식은 해를 갖지 않는다는 정리이다.*

$$a^n + b^n = c^n$$

1. *네 개의 변수—*`a`, `b`, `c`, `n`*—를 받아 n이 2보다 큰 경우에 대해 페르마의 마지막 정리가 참인지 검사하는* `check_fermat` *함수를 작성하라.*

   $$a^n + b^n = c^n$$

   위의 식이 참이라면 "어머나, 페르마가 틀렸었네!", 거짓이라면 "역시 없구나."를 출력하도록 만들어라.

2. *사용자가* `a`, `b`, `c`, `n` *값을 정하도록 하고, 입력 받은 값을 정수로 변환하여* `check_fermat`*의 입력으로 쓰는 함수를 만들어라. 입력 받은 값으로 페르마의 정리가 참인지 확인해보자.*

문제 5.3. *임의의 세 개의 막대기의 길이에 따라 삼각형 만들기 가능 여부가 결정된다. 예를 들어, 한 막대기가 12인치이고 다른 두 개가 각각 1인치라고 해보자. 작은 두 개의 막대기로는 중간에서 만날 수가 없다. 임의의 세 길이가 있을 때 삼각형 만들기가 가능한지 확인하는 간단한 테스트가 있다.*

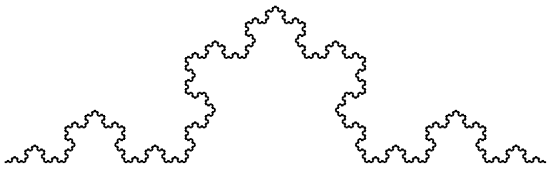

Figure 5.2: 코흐 곡선.

세 선분 중 하나의 선분이 나머지 두 선분의 합보다 길면 삼각형을 만들 수 없다. 짧으면 삼각형을 만들 수 있다. (두 선분의 길의의 합이 세 번째 길이와 같으면 "퇴화 (degenerate)" 삼각형이라고 한다.)

1. *세 개의 정수를 인자로하여 주어진 수로 삼각형 만들기가 가능한지 여부를 판단하는* `is_triangle` *함수를 작성하라. 가능하면 "Yes" 불가능하면 "No"를 출력하도록 하라.*

2. *사용자로 부터 세 정수를 입력받아, 그 수를 정수로 변환하고,* `is_triangle`*를 사용하여 주어진 정수로 삼각형을 만들 수 있는지 판단하는 함수를 작성하라.*

문제 5.4. *다음 프로그램의 결과가 무엇인가? 프로그램이 결과를 표시할 때의 스택 상태도를 그려보아라.*

```
def recurse(n, s):
    if n == 0:
        print(s)
    else:
        recurse(n-1, n+s)

recurse(3, 0)
```

1. *이 함수를* `recurse(-1, 0)`*으로 호출하면 어떻게 될까?*

2. *이 함수를 사용하기 위해 필요한 모든 정보를 담은 설명 문자열을 작성하라.*

다음의 문제들은 4장에서 설명한 `turtle` 모듈을 사용한다.

문제 5.5. *다음의 함수를 읽고 어떤 동작을 하는지 생각해보자(필요하면 4장의 예제를 읽어보자). 생각한 데로 동작하는지 실행해보자.*

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

문제 5.6. *코흐 곡선(The Koch curve)는 그림 5.2처럼 보이는 프랙탈 곡선이다. 길이 x를 사용하여 코흐 곡선을 그려라. 그리는 방법은 다음과 같다.*

1. *x/3의 길이로 코흐 곡선을 그린다.*

2. *왼쪽으로 60도 회전한다.*
3. *x/3의 길이로 코흐 곡선을 그린다.*
4. *오른쪽으로 120도 회전한다.*
5. *x/3의 길이로 코흐 곡선을 그린다.*
6. *왼쪽으로 60도 회전한다.*
7. *x/3의 길이로 코흐 곡선을 그린다.*

만약, x가 3보다 작으면 x만큼의 길이로 직선을 그리면 된다.

1. *거북이 객체와 길이를 매개 변수로 하는* `koch` *함수를 작성하라. 이 함수로 주어진 길이로 거북이를 사용하여 코흐 곡선을 그려보아라.*
2. *세 개의 코흐 곡선으로 눈송이의 외각선을 그리는* `snowflake` *함수를 작성하라.*

   해답: http://thinkpython2.com/code/koch.py.
3. *코흐 곡선을 일반화하는 방법은 여러 개가 있다. http://en.wikipedia.org/wiki/Koch_snowflake 에 나와 있는 것 중 마음에 드는 것을 골라 만들어 보자.*