Chapter 3

함수

프로그래밍이라는 문맥에서 **함수(function)**은 연산을 수행하기 위한 일련의 문장들에 이름을 붙여 놓은 것이다. 함수를 정의할 때는 이름과 문장들 지정해야 한다. 그 다음에 함수를 지정한 이름으로 "호출(call)"할 수 있다 .

3.1 함수 호출

우리는 이미 **함수 호출(function call)**의 예를 보았다:

```
>>> type(42)
<class 'int'>
```

함수의 이름은 type이다. 괄호 안의 표현은 함수의 **인자(argument)**라고 불린다. 이 함수의 결과로 인자의 분류를 알려준다.

흔히 함수가 인자를 "받아서(take)" 결과를 "리턴(return)"한다고 말한다. 그리고 함수의 결과를 **리턴 값(return value)**이라고 부른다.

Python은 어떤 값을 한 분류에서 다른 분류로 변환하는 함수들을 제공한다. int 함수는 어떤 값이든 받아서 정수로 변환할 수 있으면 변환하고 그렇지 못하는 경우는 불평을 한다.

```
>>> int('32')
32
>>> int('Hello')
ValueError: invalid literal for int(): Hello
```

int 함수는 부동 소수점 숫자를 정수로 변환할 수 있지만 반올림을 하지 않는다. 소수점 부분을 잘라 버리기만 한다.

```
>>> int(3.99999)
3
>>> int(-2.3)
-2
```

float 함수는 정수와 문자열을 부동 소수점 숫자로 변환한다.

```
>>> float(32)
32.0
>>> float('3.14159')
3.14159
```

마지막으로 str 함수는 인자를 문자열로 변환한다.

```
>>> str(32)
'32'
>>> str(3.14159)
'3.14159'
```

3.2 수학 함수

Python은 대부분의 익숙한 수학 함수들을 포함하는 수학 모듈을 갖고 있다. **모듈(module)**은 관련된 함수들을 모아 놓은 파일이다.

모듈에 포함된 함수를 사용하기 전에 **import 문**으로 읽어들여야 한다.

```
>>> import math
```

이 문장은 math라고 불리는 **모듈 객체(module object)**를 생성한다. 모듈 객체를 표시하도록 하면 객체에 관한 정볼르 얻을 수 있다.

```
>>> math
<module 'math' (built-in)>
```

모듈 객체는 모듈에 정의된 함수들과 변수들을 포함하고 있다. 모듈에 포함된 함수를 사용하고 싶으면 모듈과 사용하기 원하는 함수의 이름을 구두점(닷, dot)으로 구분하여 지정해야 한다. 이와 같은 형식을 **닷 표기법(dot notation)**이라고 부른다.

```
>>> ratio = signal_power / noise_power
>>> decibels = 10 * math.log10(ratio)

>>> radians = 0.7
>>> height = math.sin(radians)
```

첫 예제에서는 math.log10을 사용하여 signal_power 와 noise_power의 변수가 정의되어 있다고 가정했을 때의 신호대잡음비를 데시벨로 변환한다. 수학 모듈은 밑 수를 e로 하는 log도 제공한다.

두 번째 예제에서는 radians 변수에 대한 사인 값을 계산한다. sin, cos 그리고 tan와 같은 삼각함수와 관련된 함수들의 인자 값은 이 예제에서 사용한 변수 명처럼 라디안을 쓴다. 도를 라디안으로 변환하려면 180으로 나누고 π로 곱하면 된다.

```
>>> degrees = 45
>>> radians = degrees / 180.0 * math.pi
>>> math.sin(radians)
0.707106781187
```

math.pi라고 쓰면 pi 변수를 수학 모듈에서 가져온다. 이 값은 소수점 15자리까지 정확한 근사값이다.

삼각함수 식에 따라 45도의 사인 값은 2분의 루트 2이기 때문에 예제의 결과가 맞는지 비교해 볼 수 있다.

```
>>> math.sqrt(2) / 2.0
0.707106781187
```

3.3 구성

지금까지 변수와 수식 그리고 문장이라는 프로그램의 요소들을 독립적으로만 다루었기 때문에 구성 요소들을 화합하여 쓰는 것에 대해서는 다루지 않았다.

프로그래밍 언어의 가장 유익한 기능 중 하나는 작은 단위의 필요한 부분들을 모아서 **구성(compose)**할 수 있다는 것이다. 예를 들면, 함수의 인자로 산술 연산자들을 포함하는 어떤 수식이라도 다 받을 수 있다.

```
x = math.sin(degrees / 360.0 * 2 * math.pi)
```

함수 호출도 할 수 있다.

```
x = math.exp(math.log(x+1))
```

거의 모든 곳에 값을 넣을 수 있으며 임의의 수식을 쓸 수 있다. 단, 한 가지 예외가 있다. 할당문의 왼쪽에는 변수의 이름이 있어야 한다. 왼쪽에 다른 어떤 수식이라도 오면 문법 오류가 발생한다 (이것에 대한 예외 상황을 이후에 살펴 보자).

```
>>> minutes = hours * 60                # 맞음
>>> hours * 60 = minutes                # 틀림!
SyntaxError: can't assign to operator
```

3.4 새로운 함수의 추가

지금까지는 Python이 제공하는 함수들만 사용했었는데, 새로운 함수도 추가 할 수 있다. **함수 정의(function definition)**로 새로운 함수에 이름을 부여하고 그 함수를 호출했을 때 실행될 일련의 문장들을 지정할 수 있다.

여기 그 예가 있다.

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")
```

`def`라는 키워드가 함수 정의를 표시한다. 이 함수의 이름은 `print_lyrics`이다. 함수 이름에 대한 규칙은 변수명에 대한 규칙과 똑같다. 글자와 숫자 그리고 밑줄 표시는 사용 가능하지만 숫자가 첫 글자로 오면 안된다. 키워드로 예약되어 있는 단어들은 함수명으로 쓸 수 없으며 변수와 함수를 동일한 이름으로 짓는 것은 피해야 한다.

함수명 다음에 빈 괄호를 사용한 것은 이 함수가 아무런 인자도 사용하지 않는다는 것을 뜻한다.

함수 정의의 첫 줄을 **헤더(header)**라고 부르고 그 나머지 부분을 **내용(바디, body)**라고 부른다. 헤더의 끝은 콜론이고 내용은 들여쓰기를 해야 한다. 관례적으로 항상 4개의 공백으로 들여쓰기를 한다. 내용에는 몇 개의 문장이든 포함될 수 있다.

`print`문에 문자열은 큰따옴표로 싸여있다. 작은 따옴표나 큰따옴표나 동일하다. 대부분의 사람들은 작은 따옴표를 사용하지만, 예외적으로 작은 따옴표(아포스트로피 또는 생략 기호)가 문자열 중에 포함된 경우에만 큰따옴표를 사용한다.

모든 인용부호(작은 따옴표와 큰따옴표)는 대체적으로 키보드의 엔터 키 옆에 있는 "반듯한 인용부호"를 사용해야 한다. 이 문장에서 쓰고 있는 "휘어진 인용부호"는 Python에서 쓸 수 없다.

대화식 모드에서 함수 정의를 입력하면 인터프리터는 함수의 내용 부분에 점(...)을 표시하여 정의가 끝나지 않았다는 것을 표시한다.

```
>>> def print_lyrics():
...     print("I'm a lumberjack, and I'm okay.")
...     print("I sleep all night and I work all day.")
...
```

함수의 정의를 끝내려면 빈 줄을 입력하면 된다. 이렇게 정의된 함수 객체는 function으로 분류된다.

```
>>> print(print_lyrics)
<function print_lyrics at 0xb7e99e9c>
>>> type(print_lyrics)
<class 'function'>
```

새로운 함수를 호출하는 문법은 내장된 함수들을 호출하는 방식과 똑같다.

```
>>> print_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

이렇게 정의된 함수는 다른 함수 내에서 사용될 수 있다. 예를 들어 방금 출력한 후렴구를 반복적으로 호출하도록 repeat_lyrics라는 함수를 작성해 보자

```
def repeat_lyrics():
    print_lyrics()
    print_lyrics()
```

그리고 repeat_lyrics을 호출해보자.

```
>>> repeat_lyrics()
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

실제 노래가 이렇게 흘러가지는 않는다.

3.5 정의와 활용

이전 절에서 사용한 코드를 다시 가져와 보자. 전체 코드는 다음과 같다.

```
def print_lyrics():
    print("I'm a lumberjack, and I'm okay.")
    print("I sleep all night and I work all day.")

def repeat_lyrics():
    print_lyrics()
    print_lyrics()

repeat_lyrics()
```

이 프로그램은 print_lyrics 와 repeat_lyrics라는 두 개의 함수를 정의하고 있다. 함수 정의는 다른 문장처럼 실행되지만, 실행의 결과로 함수 객체만 생성이 된다. 실제 함수 정의의 내용은 그 함수가 호출되기 전까지는 실행되지 않으며 함수 정의 자체는 어떠한 결과도 출력하지 않는다.

예상한 것처럼, 함수를 생성한 후에야 함수를 실행할 수 있다. 다시 말하면, 함수가 호출되기 이전에 함수가 정의되어야 한다.

연습을 해보자. 두 번째 함수의 마지막 줄을 이 프로그램의 가장 처음으로 옮겨보자. 프로그램을 실행해서 함수 정의보다 먼저 함수 호출 문장이 나타나면 어떤 오류 메시지를 출력하는지 보자.

함수 호출 문장을 원상복귀 한 후, 이번에는 `repeat_lyrics` 다음에 `print_lyrics`를 배치해보자. 이 상태로 프로그램을 실행시키면 어떻게 되나?

3.6 실행의 흐름

함수를 정의한 후에 사용되도록 만들려면 어떤 순서로 문장들이 실행되는지를 알아야 한다. 이를 **실행의 흐름(flow of execution)**이라고 부른다.

실행의 순서는 언제 프로그램의 첫 문장부터 시작된다. 가장 위에서부터 밑까지 문장은 한 번에 하나씩 실행된다.

함수 정의는 프로그램의 실행의 흐름을 변경시키지 않지만 함수 정의 내의 문장들은 해당 함수가 호출되기 이전에는 실행되지 않는다는 것을 기억해야 한다.

함수 호출은 실행 흐름의 우회로와 같다. 다음 문장을 실행시키는 대신 함수의 내용으로 이동하여 그 안의 문장들을 실행시킨 후 이전에 멈췄던 부분부터 실행한다.

매우 간단하게 들리겠지만, 함수가 또 다른 함수를 호출 할 수 있다는 것을 되새겨 보면 그렇지 않다는 것을 깨달을 것이다. 함수의 중간에서 다른 함수의 문장들을 호출할 수 있다. 그러는 중에 또 다른 함수를 호출할 수도 있다!

다행스러운 것은, Python은 현재 실행 중이던 위치를 파악하는 것에 능숙하다. 함수의 실행이 완료되면 호출했던 함수에서 마지막 실행 중이던 위치를 찾아 계속 진행할 수 있다. 마침내 프로그램의 끝에 도달하면 종료한다.

정리하면, 프로그램을 읽을 때에는 프로그램의 처음부터 한 줄 씩 읽으려 할 필요가 없다. 오히려 실행의 흐름을 따르는 것이 이해하기 더 쉬울 수 있다.

3.7 매개 변수와 인자값

어떤 함수들은 인자를 사용한다는 것을 보았다. 예를 들어 `math.sin`을 쓸 때 숫자를 인자로 입력했었다. 어떤 함수는 하나 이상의 인자를 받기도 한다. `math.pow`는 밑수와 지수라는 두 개의 인자를 받는다.

함수 내에서는 전달 받은 인자를 **매개 변수(파라미터, parameter)**라는 변수로 할당된다. 인자를 받는 어떤 함수의 정의를 살펴보자.

```
def print_twice(bruce):
    print(bruce)
    print(bruce)
```

이 함수는 전달 받은 인자를 `bruce`라는 매개 변수로 할당한다. 함수가 호출되면 그 매개 변수가 무엇이든 간에 두 번 출력하고 있다.

다음의 함수는 출력 가능한 모든 값에 쓸 수 있다.

```
>>> print_twice('Spam')
Spam
Spam
>>> print_twice(42)
```

```
42
42
>>> print_twice(math.pi)
3.14159265359
3.14159265359
```

내장된 함수들에 적용되었던 구성에 관한 규칙들은 프로그래머가 정의한 함수들에도 똑같이 적용된다. 그러므로, print_twice의 인자 값으로 어떤 수식이라도 쓸 수 있다.

```
>>> print_twice('Spam '*4)
Spam Spam Spam Spam
Spam Spam Spam Spam
>>> print_twice(math.cos(math.pi))
-1.0
-1.0
```

함수가 호출되기 전에 인자 값을 계산한다. 예에서 사용한 'Spam'*4와 math.cos(math.pi)라는 수식은 한 번씩 계산된다.

변수 역시도 인자로 사용할 수 있다.

```
>>> michael = 'Eric, the half a bee.'
>>> print_twice(michael)
Eric, the half a bee.
Eric, the half a bee.
```

인자로 전달한 (michael)이라는 병수명은 함수를 정의할 때 사용한 매개 변수(bruce)와는 아무런 관계가 없다. 호출되기 전 코드(호출자, caller) 어떤 값이었든 상관없다. print_twice 함수 내에서는 bruce라 부른다.

3.8 변수와 매개 변수의 지역성

함수 내에서 변수를 생성하면 **지역 또는 로컬(local)** 변수라고 부른다. 함수 내에서만 존재하기 때문이다. 예를 들어 보자.

```
def cat_twice(part1, part2):
    cat = part1 + part2
    print_twice(cat)
```

이 함수는 두 개의 인자를 받아서 연결하고, 그 결과를 두 번 출력한다. 이 함수 정의를 쓰는 예를 살펴 보자.

```
>>> line1 = 'Bing tiddle '
>>> line2 = 'tiddle bang.'
>>> cat_twice(line1, line2)
Bing tiddle tiddle bang.
Bing tiddle tiddle bang.
```

cat_twice 가 종료하면 함수 내에서 사용되었던 변수 cat은 없어진다. 출력해보려고 하면 예외처리 된다.

```
>>> print(cat)
NameError: name 'cat' is not defined
```

매개 변수도 지역에서만 정의된다. 예를 들어 print_twice 밖에서는 bruce라는 것은 존재하지 않는다.

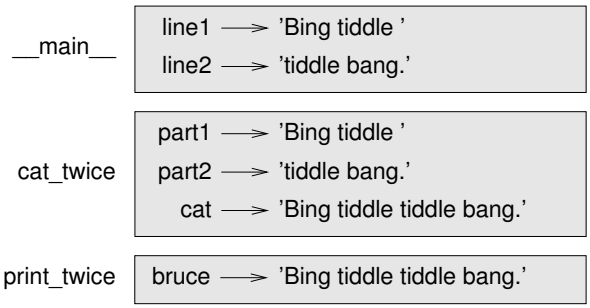

Figure 3.1: 스택 상태도.

3.9 스택 상태도

각 변수의 사용 범위 파악에 **스택 상태도(stack diagram)**을 그려보는 것이 도움이 된다. 상태도와 유사하게 스택 상태도는 각 변수의 값을 나타낸다. 추가적으로 각 변수가 어떤 함수에 포함되는지도 나타낸다.

각 함수는 **프레임(frame)**이라는 단위로 구분된다. 여기서 프레임은 하나의 상자로 함수의 이름이 곁에 적혀 있고 상자 내부에는 변수와 매개 변수가 적혀 있다. 앞서 본 예제의 스택 상태도는 그림 3.1에 나타나 있다.

어떤 함수가 어떤 함수를 호출 했는지를 알아 볼 수 있도록 프레임들이 스택에 정리되어 있다. 이 예제에서는 `print_twice` 함수는 `cat_twice` 함수에 의해 호출되었으며 `cat_twice` 함수는 `__main__`에 의해 호출이 되었다. `__main__`라는 함수는 최상위 프레임에게 부여하는 특별한 이름이다. 함수 밖에서 생성된 변수는 `__main__` 함수에 속해 있다.

각 매개 변수는 해당 인자가 갖고 있는 값과 똑같은 값을 갖고 있다. 그렇기 때문에 `part1`가 같고 있는 값은 `line1`이 갖고 있는 값고 똑같으며, `part2`가 같고 있는 값은 `line2`와 동일하다. 그리고, `bruce`는 `cat`과 똑같다.

함수 호출 중에 오류가 발생하면 Python은 함수의 이름과 호출을 시도한 함수명을 출력한다. 그리고 다시 그 함수를 호출한 상위 프레임의 함수명을 출력한다. 이 과정을 `__main__`에 도달 할 때까지 반복한다.

예를 들어, `print_twice` 내에서 `cat`를 접근하려고 시도 한다면 `NameError`이라는 오류 메시지를 받을 것이다.

```
Traceback (innermost last):
  File "test.py", line 13, in __main__
    cat_twice(line1, line2)
  File "test.py", line 5, in cat_twice
    print_twice(cat)
  File "test.py", line 9, in print_twice
    print(cat)
NameError: name 'cat' is not defined
```

이런 식으로 함수의 리스트를 보여주는 것을 **추적(트레이스백, traceback)**이라고 부른다. 오류가 발생했을 때 어떤 프로그램 파일에서 오류가 발생했는지 그리고 실행 중이던 함수와 오류를 일으킨 줄 번호에 대한 정보를 보여준다. 그리고 오류를 일으킨 코드도 보여준다.

트레이스백의 함수의 순서는 스택 상태도에서 나타난 프레임의 순서와 똑같다. 현재 실행 중이던 함수가 가장 아래에 표시된다.

3.10 열매가 있는 함수들과 비어 있는 함수

수학 함수들과 같은 우리가 사용해본 함수들은 결과를 리턴한다. 이런 종류의 함수를 구분하는 좋은 이름이 따로 없어서 **열매가 있는 함수(fruitful function)**이라고 부른다. `print_twice`와 같은 류의 다른 함수들은 어떤 동작을 하지만 결과 값을 리턴하지는 않는다. 이런 류는 **비어 있는 함수(void function)**라고 부른다.

열매가 있는 함수를 호출하면 거의 모든 경우에 리턴받은 결과를 활용하기를 원한다. 예를 들어, 그 결과에 변수를 할당하거나 수식의 일부로 사용하기도 한다.

```
x = math.cos(radians)
golden = (math.sqrt(5) + 1) / 2
```

대화식 모드에서 함수를 호출할 때 Python은 결과를 표시한다.

```
>>> math.sqrt(5)
2.2360679774997898
```

스크립트로 실행했을 때, 열매가 있는 함수를 그 자체로만 호출하면 리턴 받은 결과는 영원히 잃어버리고 만다!

```
math.sqrt(5)
```

이 스크립트는 루트 5를 계산하지만 결과를 저장하지도 표시하지도 않기 때문에 그렇게 유용하지는 않다.

비어 있는 함수는 화면에 무언가를 표시하거나 다른 어떤 영향이 있는 것처럼 보이지만 리턴할 결과 값이 없다. 만약 결과 값을 변수에 할당하려고 시도한다면 `None`이라는 특수한 값을 돌려받는다.

```
>>> result = print_twice('Bing')
Bing
Bing
>>> print(result)
None
```

`None`이라는 값은 `'None'`이라는 문자열과 같은 것이 아니다. 그 자체가 특수한 값을 같은 또 다른 분류이다.

```
>>> type(None)
<class 'NoneType'>
```

지금까지 우리가 작성한 모든 함수들은 모두 비어 있는 함수들이었다. 이제 앞으로 다룰 장들에서 열매가 있는 함수들을 작성하기 시작할 것이다.

3.11 왜 함수인가?

프로그램의 내용을 왜 함수들로 나눠야 하는지에 대한 이유가 아직은 명확하지 않을 것이다. 그럴만한 이유는 많이 있다.

- 새로운 함수를 생성하면 문장들의 묶음에 이름을 지어줄 수 있다. 그러면 프로그램을 읽거나 디버깅하기가 쉬워진다.
- 함수들을 활용하면 반복적인 코드들을 제거할 수 있기 때문에 프로그램의 길이가 짧아진다. 나중에 반복되는 코드를 수정을 할 일이 생긴다면 한 곳에서만 수정을 하면 된다.
- 긴 프로그램을 함수들로 나누어 놓으면 한 번에 한 부분씩 디버깅을 할 수 있으며, 디버깅이 완료되면 제대로 동작하는 전체로 다시 모을 수있다.
- 잘 설계된 함수들은 여러 프로그램들에 유용하게 사용될 수 있다. 하나를 잘 작성해 놓고 디버깅을 해 놓으면 다른 곳에서도 그 부분을 재활용할 수 있다.

3.12 디버깅

가장 중요한 기술 중에 하나인 디버깅 기술을 가져야 한다. 좌절스럽게 만들 때도 있지만 디버깅은 지적으로 풍부하고 도전적고 또한 프로그래밍을 흥미롭게 만드는 것이기도 하다.

디버깅은 어떤 면에서 탐정이 수사를 펼치는 것과 같다. 현재 발생한 결과를 이끌어 낸 과정과 사건들을 유추할 있는 여러 단서들과 만나게 된다.

디버깅은 실험에 기반한 과학과도 같다. 무엇인 잘못된 것 같단 아이디어가 떠 오르면 프로그램을 수정해보고 다시 시도 해 보면 된다. 가정이 옳다면 수정에 대한 결과를 예측해 볼 수 있게 되고 동작하는 프로그램으로 한 걸음 더 가까이 다가갈 수 있게 된다. 만약 가정이 틀렸다면 다른 방법을 시도해봐야 한다. 셜록 홈즈가 말하듯이 "불가능한 것들을 하나씩 제거한 뒤에 남은 것이 아무리 불가능해 보일지라도 그게 사실일 것이다."(A. 코난 도일, *4개의 서명*)

어떤 이들에게는 프로그래밍과 디버깅은 똑같은 것이다. 그들에게 프로그래밍이라는 것은 원하는 동작을 할 때까지 프로그램을 조금씩 디버깅해 나가는 것이다. 동작 가능한 프로그램을 먼저 작성한 후에 조금씩 변형을 만들어 내고 디버깅하는 것이 기본 접근 방식이다.

예를 들어, 리눅스라는 운영체제는 지금은 수 백만 줄의 코드로 이루어져있지만, 최초에는 리누스 토발즈가 인텔 80386 CPU를 사용하는 간단한 프로그램으로 시작되었다. 래리 그린필드에 의하면 "리누스의 초기 프로젝트 중에 하나는 AAAA를 BBBB로 변환하는 프로그램을 만드는 것이었다. 이게 나중에는 발전하여 리눅스가 되었다." (*The Linux Users' Guide(리눅스 사용자 가이드)* 베타 1판).

3.13 용어 해설

(function): 이름이 있는 일련의 문장들로 유용한 작업을 수행한다. 함수는 인자를 받을 수도 안 받을 수도 있으며 실행하였을 때 결과 돌려 줄 수도 있고 그렇지 않을 수도 있음

함수 정의(function definition): 새로운 함수를 생성하는 문장으로 함수의 이름과 매개 변수들을 지정하고 그리고 문장들을 포함함

함수 객체(function object): 함수 정의에 의해 생성되는 값. 함수의 이름이 함수 객체를 가리키는 변수임

헤더(header): 함수 정의의 첫 줄

내용(바디, body): 함수 정의 내의 일련의 문장들

매개 변수(parameter): 함수 내에서 인자로 전달된 값을 가리키는 이름

함수 호출(function call): 함수를 실행시키는 문장. 함수의 이름과 괄호로 싸여있는 인자들의 목록으로 구성되어 있음

인자(argument): 함수가 호출되었을 때 함수에 제공되는 값. 이 값은 해당 함수에 매개 변수에 할당됨

지역 변수(local variable): 함수 내에서 정의된 변수. 지역 변수는 정의된 함수 내에서만 사용될 수 있음

리턴 값(return value): 함수의 결과. 함수 호출이 수식의 일부로 사용되었다면 리턴 값은 수식이 사용하는 값이 됨

열매가 있는 함수(fruitful function): 결과를 리턴하는 함수

비어 있는 함수(void function): None을 리턴하는 함수

None**:** 비어 있는 함수가 리턴하는 특수한 값

모듈(module): A file that contains a collection of related functions and other definitions.

읽어들이기 문장(import statement): 모듈 파일을 읽어서 모듈 객체를 생성하는 문장

모듈 객체(module object): import문으로 생성되는 값으로 모듈에서 정의한 값들을 사용할 수 있도록 함

닷 표기법(dot notation): 다른 모듈의 함수를 호출하는 문법으로 모듈의 이름과 함수의 이름을 점(닷, 구두점)으로 연결하는 표기법

구성(composition): 더 큰 수식의 일부로 수식을 쓰거나 더 큰 문장의 일부로 문장을 작성하는 것

실행의 흐름(flow of execution): 문장들의 실행 순서

스택 상태도(stack diagram): 함수가 사용하고 있는 스택의 상태를 그림으로 표기하는 방법으로 변수와 각 변수가 가리키는 값을 나타나냄

프레임(frame): 스택 상태도의 상자로 함수 호출을 나타냄. 상자는 지역 변수와 함수의 매개 변수를 포함함.

트레이스백(추적, traceback): 실행 중인 함수들의 목록으로 예외가 발생할 때 출력이 됨

3.14 연습 문제

문제 3.1. right_justify*(오른쪽 정렬이라는 의미)라는 이름과 문자열* s*를 매개 변수로 사용하는 함수를 작성하여라. 문자열 앞에 충분한 공백을 두어서 문자열의 마지막 글자가 70번째 열에 보이도록 하라.*

```
>>> right_justify('monty')
                                                                monty
```

힌트: 문자열 연결과 반복을 사용하라. Python은 len*이라는 내장 함수를 갖고 있다. 이 함수는 문자열의 길이를 리턴한다.* len('monty')*의 값은 5이다.*

문제 3.2. *함수 객체는 값이기 때문에 변수에 할당할 수도 있고 인자로 전달할 수도 있다. 예를 들어,* do_twice*는 함수 객체를 인자로 전달 받아 두 번 호출하는 함수이다.*

```
def do_twice(f):
    f()
    f()
```

다음 예제는 do_twice*를 사용하여* print_spam*라는 함수를 두 번 호출한다.*

```
def print_spam():
    print('spam')

do_twice(print_spam)
```

1. *이 예제를 스크립트로 작성하여 제대로 동작하는지 확인해보라*
2. do_twice*를 두 개의 인자를 받도록 수정하여라. 함수 객체와 값을 전달받아 해당 함수는 두 번 호출하고 값은 인자로 전달하도록 만들라.*

3. *이 장의 처음에 다뤘던* `print_twice` *함수의 정의를 복사하여 스크립트에 포함시켜라*

4. *수정한* `do_twice` *함수를 사용하여* `print_twice`*를 두 번 호출하여라. 이 때,* `print_twice` *함수의 인자는* `'spam'`*을 사용하여라.*

5. *함수 객체와 값을 전달 받아 해당 함수를 4번 호출하고 값은 매개 변수로 사용하는* `do_four` *라는 새로운 함수를 정의하여라. 이 함수의 내용에는 네 개가 아니라 두 개의 문장만 있어야 한다.*

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

문제 3.3. *메모: 이 연습 문제는 지금까지 우리가 배운 문장들과 기능들만을 사용하여 해결해야 한다.*

1. *아래와 같은 격자 무늬를 만드는 함수를 작성하라:*

```
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
|         |         |
|         |         |
|         |         |
|         |         |
+ - - - - + - - - - +
```

   힌트: 한 줄에 하나 이상의 값을 출력하려면 값을 쉼표로 구분지으면 된다.

```
print('+', '-')
```

   기본적으로 `print`*는 출력이 끝나면 다음 줄로 넘어간다. 이 같은 동작을 수정하려면 다음과 같은 방법으로 동작을 변경할 수 있다.*

```
print('+', end=' ')
print('-')
```

   이 문장들의 결과는 `'+ -'`*이다.*

   `print`*문에 인자가 없다면 현재 출력 중이던 줄을 종료하고 다음 줄로 넘어간다.*

2. *네 개의 행과 열이 있는 격자 무늬를 그리는 함수를 작성하여라.*

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

*출처: 이 예제는 스티브 오우알린의 (*Practical C Programming, 3판, *오라일리 출판사, 1997), 연습 문제를 기초로 하였다.*