

---

# 함수형 프로그래밍 HOWTO

출시 버전 3.10.18

Guido van Rossum  
and the Python development team

7월 08, 2025

## Contents

<b>1</b>	<b>소개</b>	<b>2</b>
1.1	형식적 증명 가능성	3
1.2	모듈성	3
1.3	디버깅과 테스트 용이성	4
1.4	결합성	4
<b>2</b>	<b>이터레이터</b>	<b>4</b>
2.1	이터레이터를 지원하는 데이터형	5
<b>3</b>	<b>제너레이터 표현식과 리스트 컴프리헨션</b>	<b>6</b>
<b>4</b>	<b>제너레이터</b>	<b>8</b>
4.1	제너레이터에 값 전달하기	9
<b>5</b>	<b>내장 함수</b>	<b>11</b>
<b>6</b>	<b>itertools 모듈</b>	<b>12</b>
6.1	새로운 이터레이터 만들기	13
6.2	요소에 대한 함수 호출	14
6.3	요소 선택하기	14
6.4	조합 함수	15
6.5	요소 분류	16
<b>7</b>	<b>functools 모듈</b>	<b>16</b>
7.1	operator 모듈	18
<b>8</b>	<b>작은 함수와 람다 표현식</b>	<b>18</b>
<b>9</b>	<b>개정내역 및 감사의 글</b>	<b>19</b>
<b>10</b>	<b>참고 문헌</b>	<b>20</b>
10.1	일반	20
10.2	파이썬 특정	20
10.3	파이썬 설명서	20

저자 A. M. Kuchling

버전 0.32

이 문서에서는, 함수형 방식으로 프로그램을 구현하는데 적합한 파이썬의 특성에 대해 알아볼 것입니다. 함수형 프로그래밍의 개념을 소개한 뒤에, 이터레이터, 제너레이터와 같은 언어의 특성과 `itertools`, `functools`와 같은 관련 라이브러리 모듈을 살펴볼 것입니다.

## 1 소개

이 절에서는 함수형 프로그래밍의 기본적인 개념을 설명합니다; 만약 단순히 파이썬의 언어적 특성에 관해서만 관심이 있으시다면, 이터레이터 절로 건너뛰세요.

프로그래밍 언어들은 다음과 같이 각각 다른 방식으로 문제를 더 작은 부분으로 분할하는 방법을 지원합니다:

- 대부분의 프로그래밍 언어들은 **절차적**입니다: 프로그램은 컴퓨터에 프로그램의 입력을 어떻게 할지 알려주는 명령 목록입니다. C, 파스칼, 유닉스 셸과 같은 것들은 절차적 언어입니다.
- **선언적** 언어에서는 해결해야 할 문제를 설명하는 명세서를 작성하고, 언어 구현은 계산을 효과적으로 수행하는 방법을 파악합니다. SQL은 가장 친숙한 선언적 언어입니다; SQL 질의는 검색하고 싶은 데이터 세트를 설명하고, SQL 엔진은 테이블을 스캔하거나 인덱스를 사용할 것인지, 어떤 하위 구문을 먼저 수행해야 하는지 등을 결정합니다.
- **객체지향** 프로그램은 객체들의 컬렉션을 다룹니다. 객체는 내부적인 상태를 갖고 있으며 이 내부적인 상태를 어떤 방식으로 가져오거나 수정하는 메서드를 제공합니다. 스몰토크와 자바는 객체지향 언어입니다. C++와 파이썬은 객체지향 프로그래밍을 지원하는 언어이지만, 객체 지향적인 특성들을 사용하도록 강제하지는 않습니다.
- **함수형** 프로그래밍은 함수들의 세트로 문제를 분해합니다. 이상적으로 말하면, 함수들은 입력을 받아서 출력을 만들어내기만 하며, 주어진 입력에 대해 생성된 출력에 영향을 끼칠만한 어떠한 내부적인 상태도 가지지 않습니다. 잘 알려진 함수형 언어로는 ML 계열(Standard ML, OCaml 및 다른 변형)과 하스켈이 있습니다.

일부 컴퓨터 언어의 설계자들은 프로그래밍에 대한 한 가지의 특별한 접근 방식을 강조합니다. 이것은 종종 다른 접근 방식으로 프로그램을 작성하는 것을 어렵게 만듭니다. 다른 언어들은 다양한 접근 방법을 지원하는 다중 패러다임 언어입니다. Lisp, C++, 파이썬 등은 다중 패러다임 언어입니다; 이러한 언어에서는 절차적, 객체 지향적 혹은 함수형으로 프로그램이나 라이브러리를 작성할 수 있습니다. 거대한 프로그램에서, 각 구역은 서로 다른 접근 방법을 사용하여 작성될 수 있습니다; 예를 들어 처리 로직이 절차적 혹은 함수형으로 작성되었을 때, GUI는 객체 지향적으로 작성될 수 있습니다.

함수형 프로그램에서, 입력은 여러 함수의 세트를 통해 흘러 다닙니다. 각 함수는 입력으로부터 동작해서 출력을 만들어냅니다. 함수형 방식은 내부 상태를 수정하거나 함수의 반환 값에서 보이지 않는 다른 변경사항들을 만드는 부작용이 있는 함수를 사용하지 않습니다. 부작용이 전혀 없는 함수를 **순수 함수**라고 합니다. 부작용을 피한다는 것은 프로그램이 실행될 때 수정될 수 있는 자료 구조를 사용하지 않는다는 의미입니다; 모든 함수의 출력은 입력에만 의존해야 합니다.

어떤 언어는 순수성에 대해 매우 엄격하며,  $a=3$  혹은  $c = a + b$ 와 같은 대입문조차 없지만, 화면에 출력하거나 디스크 파일에 쓰는 작업과 같은, 모든 부작용을 피하는 것은 어렵습니다. 다른 예로, `print()` 혹은 `time.sleep()` 함수를 호출하면 쓸모 있는 값을 반환하지 않습니다. 이 함수들은 화면에 문자열을 보내거나 잠시 동안 실행을 일시 중지하는 작업과 같은 부작용을 위해 호출합니다.

함수형 방식으로 작성된 파이썬 프로그램은 보통 극단적으로 모든 I/O 혹은 대입문을 회피하는 방식으로 나아가지는 않습니다; 대신 함수형처럼 보이는 인터페이스를 제공하며 내부적으로는 함수형이 아닌 기능들을

사용합니다. 예를 들어 함수의 구현은 여전히 지역 변수에 값을 대입하는 방식이 사용되지만 전역 변수를 수정하거나 다른 부작용을 발생시키지는 않습니다.

함수형 프로그래밍은 객체 지향 프로그래밍의 반대라고 생각할 수 있습니다. 객체는 내부 상태들을 갖고 있으며 이 상태들을 수정할 수 있는 메서드의 호출 모음이 포함된 작은 캡슐이며, 프로그램은 올바른 상태 변경 집합을 구성합니다. 함수형 프로그래밍은 가능한 한 상태 변경을 피하고자 하며 함수 간의 데이터 흐름을 사용합니다. 파이썬에서는 응용 프로그램의 객체를 나타내는 인스턴스(전자 우편 메시지, 트랜잭션 등)를 가져와서 반환하는 함수를 작성함으로써 두 가지 접근 방식을 결합할 수 있습니다.

함수형 설계는 동작 방식에 이상한 제약이 있는 것처럼 보일 수 있습니다. 왜 객체와 부작용을 피해야만 할까요? 함수형 방식은 이론적으로도, 실질적으로도 다음과 같은 장점이 있습니다:

- 형식적 증명 가능성.
- 모듈성.
- 결합성.
- 디버깅과 테스트 용이성.

## 1.1 형식적 증명 가능성

이론적인 장점은 함수형 프로그램이 정확하다는 수학적 증명을 만드는 것이 더 쉽다는 것입니다.

오랫동안 연구자들은 수학적으로 프로그램이 정확하다는 것을 증명하는 방법을 찾는 데 관심을 보여왔습니다. 이것은 수많은 입력에 대해 프로그램을 테스트하고 출력이 정확하다고 결론짓거나, 프로그램의 소스코드를 읽어보고 코드가 올바르다고 결론짓는 것과는 다릅니다; 그들의 목표는 입력 가능한 모든 것에 대해 프로그램이 올바른 결과를 산출한다는 엄격한 증거를 찾는 것입니다.

프로그램이 올바른지 증명하기 위해 사용하는 기술은 항상 참인 입력 데이터와 프로그램의 변수라는 특성을 지닌 **불변자**를 작성하는 것입니다. 각 코드 행에 대해, 그 행이 실행되기 **전에** 불변자 X와 Y가 참이라면, 그 행이 실행된 **후에** 약간 다른 불변자 X' 및 Y'가 참이라는 것을 보여줍니다. 이 작업은 프로그램이 종료될 때까지 계속되며, 종료 시점에서 불변자는 프로그램의 출력으로써 원하는 조건과 일치해야만 합니다.

함수형 프로그래밍에서 값 대입을 피하려는 이유는 값 대입이 이러한 기법을 활용하는 것을 어렵게 만들기 때문입니다; 대입은 다음 단계로 나아갈 수 있는 새 불변자를 만들지 않은 채로, 대입전에 참이었던 불변자를 무너뜨릴 수 있습니다.

불행하게도, 정확한 프로그램임을 증명하는 것은 실제로는 비실용적이며 파이썬 소프트웨어와 관련이 없습니다. 사소한 프로그램일지라도 여러 페이지 분량의 증명이 필요합니다; 적당히 복잡한 프로그램에 대한 정확성의 증명은 엄청난 양일 것이며, 매일 사용하는 프로그램(파이썬 인터프리터, XML 파서, 웹 브라우저)의 정확성은 거의 증명이 불가능할 수도 있습니다. 만약 증명을 작성하거나 만들었더라도, 그 증명이 검증된 것인지 의구심이 들 것입니다; 어쩌면 그 증명에 오류가 있을 수도 있고, 프로그램의 정확성이 증명되었다고 잘못 믿고 있을 수도 있습니다.

## 1.2 모듈성

함수형 프로그래밍의 실질적인 이점은 문제를 작은 조각으로 분해하도록 강제한다는 점입니다. 결과적으로 프로그램은 더욱 모듈화가 됩니다. 복잡한 변환을 수행하는 거대한 함수보다, 한 가지 작업을 수행하는 작은 함수를 명시하고 작성하기가 더 쉽습니다. 작은 함수는 읽기에도 더 쉽고 오류를 확인하기도 쉽습니다.

## 1.3 디버깅과 테스트 용이성

함수형 방식 프로그램은 테스트하고 디버깅하는 것이 더 쉽습니다.

일반적으로 함수가 작고 분명하게 명시되기 때문에 디버깅이 단순화됩니다. 프로그램이 동작하지 않는다면, 각 함수는 데이터가 올바른지 확인할 수 있는 접점이 됩니다. 중간 지점의 입력과 출력을 살펴보면 버그가 있는 함수를 빠르게 분간할 수 있습니다.

각 함수는 잠재적으로 단위 테스트의 대상이기 때문에 테스트가 더 쉽습니다. 함수는 테스트를 실행하기 전에 복제해야 하는 시스템 상태에 의존하지 않습니다; 올바른 입력을 만들고 결과가 예상과 일치하는지 확인만 하면 됩니다.

## 1.4 결합성

함수형 방식의 프로그램을 만들 때, 다양한 입력과 출력으로 여러 가지 함수를 작성하게 됩니다. 이러한 함수 중 일부는 불가피하게 특정 응용 프로그램에 특화될 수 있지만, 대체로 다양한 프로그램에서 유용하게 사용할 수 있습니다. 예를 들어 디렉터리 경로를 받아서 그 디렉터리 내의 모든 XML 파일을 반환하는 함수나, 혹은 파일명을 받아서 그 내용을 반환하는 함수는 다양한 상황에 적용할 수 있습니다.

시간이 흐르면, 여러분은 개인적인 유틸리티 라이브러리를 구성하게 될 것입니다. 보통, 새로운 구성으로 기존 함수를 배치하고 현재 작업에 특화된 몇 가지 함수만을 작성해서 새로운 프로그램을 구성하게 됩니다.

## 2 이터레이터

함수형 방식의 프로그램을 작성하는 중요한 토대가 되는 파이썬 언어의 기능을 살펴보겠습니다: 이터레이터.

이터레이터는 데이터 스트림을 나타내는 객체입니다; 이 객체는 한 번에 한 요소씩 데이터를 반환합니다. 파이썬 이터레이터는 반드시 `__next__()` 라는 메서드를 지원해야 합니다. 이 메서드는 인자를 취하지 않고 항상 스트림의 다음 요소를 반환합니다. 만약 스트림에 더는 요소가 없다면, `__next__()` 는 `StopIteration` 예외를 발생시켜야 합니다. 이터레이터가 유한할 필요는 없습니다; 무한한 데이터 스트림을 생성하는 이터레이터를 작성하는 것도 합리적인 방법입니다.

내장 함수 `iter()` 는 임의의 객체를 취하여 객체의 내용이나 요소를 반환하는 이터레이터를 반환합니다. 객체가 이터레이션을 지원하지 않으면 `TypeError` 를 발생시킵니다. 파이썬의 내장 데이터형 중 몇 가지가 이터레이션을 지원하는데, 가장 일반적인 것은 리스트와 딕셔너리입니다. 이터레이터를 얻을 수 있는 객체는 이터러블 이라고 불립니다.

수동으로 이터레이션 인터페이스를 실험해볼 수 있습니다:

```
>>> L = [1, 2, 3]
>>> it = iter(L)
>>> it
<...iterator object at ...>
>>> it.__next__() # same as next(it)
1
>>> next(it)
2
>>> next(it)
3
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

파이썬은 여러 다른 컨텍스트에서 이터러블 객체를 기대하며, 가장 중요한 것은 for 문입니다. for X in Y 문에서 Y는 반드시 이터레이터 혹은 iter() 가 이터레이터를 생성할 수 있는 객체이어야 합니다. 다음 두 문장은 같은 의미입니다:

```
for i in iter(obj):
    print(i)

for i in obj:
    print(i)
```

이터레이터는 list() 또는 tuple() 생성자 함수를 사용하여 리스트나 튜플로 나타낼 수 있습니다:

```
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

시퀀스 언패킹 또한 이터레이터를 지원합니다: 이터레이터가 N개의 요소를 반환한다는 것을 알고 있다면, 그것들을 N-튜플로 언패킹할 수 있습니다:

```
>>> L = [1, 2, 3]
>>> iterator = iter(L)
>>> a, b, c = iterator
>>> a, b, c
(1, 2, 3)
```

max() 및 min() 과 같은 내장 함수는 하나의 이터레이터 인자를 취할 수 있으며 가장 큰 혹은 가장 작은 요소를 반환합니다. "in" 과 "not in" 연산자 또한 이터레이터를 지원합니다: 이터레이터가 반환한 스트림에서 X가 발견되면 X in iterator는 참입니다. 이터레이터가 무한하다면 명백한 문제에 부딪힙니다; max() 와 min() 는 영원히 결과를 반환하지 않으며, 요소 X가 스트림에서 나타나지 않으면 "in" 과 "not in" 연산자 역시 영원히 결과를 반환하지 않을 것입니다.

이터레이터에서는 오직 앞으로만 나아갈 수 있다는 점에 유의하세요; 이전 요소를 가져오거나, 이터레이터를 재설정하거나, 사본을 만들 방법은 없습니다. 이터레이터 객체는 선택적으로 이러한 추가 기능을 제공할 수 있지만, 이터레이터 프로토콜은 \_\_next\_\_() 메서드만 명시해두었습니다. 함수는 모든 이터레이터의 출력을 소비할 수 있으므로 같은 스트림에서 다른 작업을 수행해야 하는 경우 새로운 이터레이터를 만들어야 합니다.

## 2.1 이터레이터를 지원하는 데이터형

리스트와 튜플이 이터레이터를 어떻게 지원하는지 이미 살펴보았습니다. 실제로 문자열과 같은 파이썬 시퀀스형은 이터레이터의 생성을 자동으로 지원합니다.

iter() 를 딕셔너리에 적용하면 딕셔너리의 키를 반복하는 이터레이터를 반환합니다:

```
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun': 6,
...      'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}
>>> for key in m:
...     print(key, m[key])
Jan 1
Feb 2
Mar 3
Apr 4
May 5
Jun 6
Jul 7
```

(다음 페이지에 계속)

```
Aug 8
Sep 9
Oct 10
Nov 11
Dec 12
```

파이썬 3.7부터는, 딕셔너리 이터레이션 순서가 삽입 순서와 같음을 보장합니다. 이전 버전에서는, 동작이 지정되지 않았고 구현마다 다를 수 있었습니다.

`iter()` 를 딕셔너리에 적용하는 것은 항상 키를 반복하지만, 딕셔너리에는 다른 이터레이터를 반환하는 메서드가 있습니다. 값이나 키/값 쌍을 반복하는 경우에는 명시적으로 `values()` 혹은 `items()` 메서드를 사용하여 적절한 이터레이터를 얻을 수 있습니다.

`dict()` 생성자는 (키, 값) 튜플의 유한한 스트림을 반환하는 이터레이터를 받을 수 있습니다:

```
>>> L = [('Italy', 'Rome'), ('France', 'Paris'), ('US', 'Washington DC')]
>>> dict(iter(L))
{'Italy': 'Rome', 'France': 'Paris', 'US': 'Washington DC'}
```

또한 파일은 더는 새로운 줄이 없을 때까지 `readline()` 메서드를 호출하여 이터레이션을 지원합니다. 즉, 다음과 같이 파일의 각 행을 읽을 수 있습니다:

```
for line in file:
    # do something for each line
    ...
```

집합은 이터러블에서 내용을 가져와서 집합의 원소를 반복할 수 있습니다:

```
>>> S = {2, 3, 5, 7, 11, 13}
>>> for i in S:
...     print(i)
2
3
5
7
11
13
```

### 3 제너레이터 표현식과 리스트 컴프리헨션

이터레이터의 출력에 대한 두 가지 일반적인 연산은 1) 모든 요소에 대해 어떤 연산을 수행하고, 2) 어떤 조건을 만족하는 요소의 부분 집합을 선택하는 것입니다. 예를 들어 문자열 리스트가 있으면 각 줄에서 후미 공백을 제거하거나, 주어진 부분 문자열을 포함하는 모든 문자열을 추출할 수 있습니다.

리스트 컴프리헨션과 제너레이터 표현식(줄임말: “listcomps” 및 “genexps”)은 함수형 프로그래밍 언어 하스켈(<https://www.haskell.org/>)에서 빌린 이러한 작업을 위한 간결한 표기법입니다. 다음 코드를 사용하여 문자열 스트림에서 모든 공백을 제거할 수 있습니다:

```
>>> line_list = [' line 1\n', 'line 2 \n', ' \n', '']

>>> # Generator expression -- returns iterator
>>> stripped_iter = (line.strip() for line in line_list)

>>> # List comprehension -- returns list
>>> stripped_list = [line.strip() for line in line_list]
```

"if" 조건을 추가하여 특정 요소만 선택할 수도 있습니다:

```
>>> stripped_list = [line.strip() for line in line_list
...                  if line != ""]
```

리스트 컴프리헨션을 사용하면 파이썬 리스트를 얻을 수 있습니다; `stripped_list` 는 이터레이터가 아니라 결과 행을 담고 있는 리스트입니다. 제너레이터 표현식은 필요에 따라 값을 계산하는 이터레이터를 반환하며 모든 값을 한 번에 구체화할 필요가 없습니다. 즉, 무한 스트림이나 매우 많은 양의 데이터를 반환하는 이터레이터로 작업하는 경우 리스트 컴프리헨션은 유용하지 않습니다. 제너레이터 표현식은 이러한 상황에서 유용합니다.

제너레이터 표현식은 괄호("(")로 묶여 있으며 리스트 컴프리헨션은 대괄호("[")로 묶여 있습니다. 제너레이터 표현식은 다음과 같은 형식입니다:

```
( expression for expr in sequence1
    if condition1
    for expr2 in sequence2
    if condition2
    for expr3 in sequence3
    ...
    if condition3
    for exprN in sequenceN
    if conditionN )
```

다시 말하면, 리스트 컴프리헨션을 위해서는 바깥쪽 괄호만 다릅니다(괄호 대신 대괄호).

생성된 출력의 요소는 `expression` 의 연속적인 값이 될 것입니다. `if` 절은 모두 선택적입니다; `if` 절이 존재한다면, `expression` 은 `condition` 이 참일 때만 평가되고 결과에 추가됩니다.

제너레이터 표현식은 항상 괄호 안에 작성해야 하지만 함수 호출을 알리는 괄호도 포함됩니다. 함수에 즉시 전달되는 이터레이터를 만들고 싶다면 다음과 같이 작성할 수 있습니다:

```
obj_total = sum(obj.count for obj in list_all_objects())
```

`for...in` 절은 반복할 시퀀스를 포함합니다. 시퀀스는 왼쪽에서 오른쪽으로 반복되며 병렬로 처리되지 않기 때문에 같은 길이일 필요는 없습니다. `sequence1` 의 각 요소에 대해 `sequence2` 는 처음부터 반복됩니다. `sequence3` 은 `sequence1` 과 `sequence2` 의 각각 모든 결과에 대해 반복됩니다.

다른 식으로 표현하면, 리스트 컴프리헨션 혹은 제너레이터 표현식은 다음 파이썬 코드와 같습니다:

```
for expr1 in sequence1:
    if not (condition1):
        continue # Skip this element
    for expr2 in sequence2:
        if not (condition2):
            continue # Skip this element
        ...
    for exprN in sequenceN:
        if not (conditionN):
            continue # Skip this element

    # Output the value of
    # the expression.
```

이것은 여러 개의 `for...in` 절이 있지만 `if` 절이 없을 때 결과 출력의 길이가 모든 시퀀스 길이의 곱과 같음을 의미합니다. 길이가 3인 두 개의 리스트가 있는 경우 출력 목록의 길이는 9개입니다:



```
>>> seq1 = 'abc'
>>> seq2 = (1, 2, 3)
>>> [(x, y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3),
 ('b', 1), ('b', 2), ('b', 3),
 ('c', 1), ('c', 2), ('c', 3)]
```

파이썬 문법의 모호함을 피하고자, `expression` 이 튜플을 생성하고 있다면, 괄호로 묶어야 합니다. 아래의 첫 번째 리스트 컴프리헨션은 구문 오류이며, 두 번째는 올바릅니다:

```
# Syntax error
[x, y for x in seq1 for y in seq2]
# Correct
[(x, y) for x in seq1 for y in seq2]
```

## 4 제너레이터

제너레이터는 이터레이터를 작성하는 작업을 단순화하는 특별한 클래스의 함수입니다. 일반 함수는 값을 계산하여 반환하지만, 제너레이터는 값의 스트림을 반환하는 이터레이터를 반환합니다.

파이썬이나 C에서 정규 함수 호출이 어떻게 작동하는지 잘 알고 있을 것입니다. 함수를 호출하면 지역 변수가 생성되는 비공개 이름 공간이 생깁니다. 함수가 `return` 문에 도달하면 지역 변수가 소멸하고 그 값이 호출자에게 반환됩니다. 같은 함수를 나중에 호출하면 새로운 비공개 이름 공간과 새로운 지역 변수 집합이 만들어집니다. 그러나 지역 변수가 함수를 빠져나갈 때 버려지지 않으면 어떻게 될까요? 나중에 중단했던 곳에서 함수를 다시 시작할 수 있다면 어떨까요? 이것이 제너레이터가 제공하는 것입니다; 그들은 재개 가능한 함수라고 생각할 수 있습니다.

다음은 제너레이터 함수의 가장 간단한 예입니다:

```
>>> def generate_ints(N):
...     for i in range(N):
...         yield i
```

`yield` 키워드를 포함하는 함수는 제너레이터 함수입니다; 이것은 파이썬의 바이트 코드 컴파일러에 의해 감지됩니다. 결과적으로 컴파일러는 특별하게 함수를 컴파일합니다.

제너레이터 함수를 호출하면 단일 값을 반환하지 않습니다; 대신 이터레이터 프로토콜을 지원하는 제너레이터 객체를 반환합니다. `yield` 표현식을 실행하면 제너레이터는 `return` 문과 비슷하게 `i` 의 값을 출력합니다. `yield` 와 `return` 의 큰 차이점은 `yield` 에 도달하면 제너레이터의 실행 상태가 일시 중단되고 지역 변수가 보존된다는 것입니다. 제너레이터의 `__next__()` 메서드가 다음에 실행될 때, 함수가 다시 실행됩니다.

다음은 `generate_ints()` 제너레이터의 사용 예입니다:

```
>>> gen = generate_ints(3)
>>> gen
<generator object generate_ints at ...>
>>> next(gen)
0
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
Traceback (most recent call last):
  File "stdin", line 1, in <module>
```

(다음 페이지에 계속)



```
File "stdin", line 2, in generate_ints
StopIteration
```

여러분은 똑같이 `for i in generate_ints(5)` 또는 `a, b, c = generate_ints(3)` 라고 쓸 수 있습니다.

제너레이터 함수 내에서, `return value` 는 `__next__()` 메서드에서 `StopIteration(value)` 를 발생시킵니다. 이런 일이 발생하거나 함수의 맨 아래에 도달하면 값의 행렬이 끝나고 제너레이터는 더는 값을 산출할 수 없습니다.

직접 클래스를 작성하고 제너레이터의 모든 지역 변수를 인스턴스 변수로 저장하여 제너레이터의 효과를 수동으로 얻을 수 있습니다. 예를 들어, 정수 리스트를 반환하는 것은 `self.count` 를 0으로 설정하고 `__next__()` 메서드로 `self.count` 를 증가시켜 반환하는 식으로 수행할 수 있습니다. 그러나, 다소 복잡한 제너레이터의 경우에는 해당 클래스를 작성하는 것이 훨씬 더 복잡할 수 있습니다.

파이썬의 라이브러리인 `Lib/test/test_generators.py` 에 포함된 테스트 묶음에는 더 많은 흥미로운 예제들이 있습니다. 제너레이터를 재귀적으로 사용하여 트리를 중위 순회하는 것을 구현하는 하나의 제너레이터가 있습니다.

```
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x

        yield t.label

        for x in inorder(t.right):
            yield x
```

`test_generators.py` 의 다른 두 가지 예는, N-여왕 문제( $N \times N$  체스판에 서로 다른 왕비를 위협할 수 없도록 N개의 왕비를 배치하는 문제)와 기사의 여행(기사가  $N \times N$  체스판의 모든 칸을 정확히 한 번씩 갈 수 있도록 하는 방법을 찾는 문제)입니다.

## 4.1 제너레이터에 값 전달하기

파이썬 2.4 및 그 이전 버전에서 제너레이터는 출력만 생성했습니다. 제너레이터의 코드가 이터레이터를 만들기 위해 호출된 후에는 그 실행이 다시 시작될 때 함수에 새로운 정보를 전달할 방법이 없었습니다. 제너레이터가 전역 변수를 보거나 호출자가 수정할 수 있는 변경 가능한 객체를 전달함으로써 이 기능을 해킹할 수 있지만, 이러한 접근법은 지저분한 방식입니다.

파이썬 2.5에서는 제너레이터에 값을 전달하는 간단한 방법이 있습니다. `yield` 는 표현식이 되어 변수에 대입하거나 다른 식으로 조작할 수 있는 값을 반환합니다:

```
val = (yield i)
```

위 예제처럼 반환 값으로 무엇인가를 할 때 `yield` 표현식 주위에 항상 괄호를 넣는 것이 좋습니다. 괄호는 항상 필요한 것은 아니지만 필요한 시점을 기억하지 않고 항상 추가하기가 더 쉽습니다.

(PEP 342 는 정확한 규칙을 설명합니다. 이것은 대입의 오른쪽에 있는 최상위 표현식에서 발생하는 경우를 제외하고 항상 `yield` 표현식을 괄호로 묶어야 한다는 것입니다. `val = yield i` 라고 쓸 수도 있지만, `val = (yield i) + 12` 처럼 연산이 있을 때는 괄호를 써야합니다.)

값은 `send(value)` 메서드를 호출하여 제너레이터로 보내집니다. 이 메서드는 제너레이터의 코드를 다시 시작하고 `yield` 표현식은 지정된 값을 반환합니다. 만약 정규 `__next__()` 메서드가 호출되면 `yield` 는 `None` 을 반환합니다.

다음은 1씩 증가하며 내부 카운터값을 변경할 수 있는 간단한 카운터입니다.

```
def counter(maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
```

다음은 카운터 변경의 예시입니다:

```
>>> it = counter(10)
>>> next(it)
0
>>> next(it)
1
>>> it.send(8)
8
>>> next(it)
9
>>> next(it)
Traceback (most recent call last):
  File "t.py", line 15, in <module>
    it.next()
StopIteration
```

yield 가 종종 None 을 반환할 것이므로, 항상 이 경우를 확인해야 합니다. send() 메서드가 제너레이터 함수를 다시 시작하는데 사용되는 유일한 메서드가 아니라면, 표현식의 결과값을 확인없이 사용하지 마세요.

send() 외에도 제너레이터에 대한 두 가지 다른 메서드가 있습니다:

- throw(value) 는 제너레이터 내에서 예외를 발생시키는 데 사용됩니다; 예외는 제너레이터의 실행이 일시 중지된 yield 표현식에 의해 발생합니다.
- close() 는 생성자 내에서 GeneratorExit 예외를 발생시켜 이터레이션을 종료합니다. 이 예외가 발생하면 제너레이터의 코드는 GeneratorExit 또는 StopIteration 을 발생시켜야 합니다; 예외를 받고도 다른 작업을 하는 것은 금지되어 있으며 RuntimeError 를 촉발합니다. close() 는 제너레이터가 가비지로 수거될 때 파이썬의 가비지 수거기에 의해 호출될 것입니다.

GeneratorExit 이 발생할 때 정리 작업을 위한 코드를 실행해야 한다면 GeneratorExit 를 잡는 대신 try: ... finally: 를 사용하는 것이 좋습니다.

이러한 변화의 누적 효과는 제너레이터를 일방적인 정보 생산자에서 생산자와 소비자 모두로 전환하는 것입니다.

제너레이터는 코루틴 이 되어 더 일반적인 형태의 서브루틴이 됩니다. 서브루틴은 한 지점에서 시작되고 다른 한 지점(함수의 맨 위와 return 문)에서 빠져나옵니다. 그러나 여러 다른 지점에서 코루틴을 시작하고 빠져나오고 다시 시작할 수 있습니다(yield 문).

## 5 내장 함수

이터레이터에서 자주 사용되는 내장 함수를 자세히 살펴보겠습니다.

파이썬의 두 가지 내장 함수인 `map()` 와 `filter()` 는 제너레이터 표현식의 기능을 복제합니다:

`map(f, iterA, iterB, ...)` 은 다음과 같은 시퀀스에 대한 이터레이터를 반환합니다. `f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]), ....`

```
>>> def upper(s):
...     return s.upper()
```

```
>>> list(map(upper, ['sentence', 'fragment']))
['SENTENCE', 'FRAGMENT']
>>> [upper(s) for s in ['sentence', 'fragment']]
['SENTENCE', 'FRAGMENT']
```

물론 리스트 컴프리헨션으로 같은 효과를 얻을 수 있습니다.

`filter(predicate, iter)` 는 특정 조건을 만족하는 모든 시퀀스 요소에 대한 이터레이터를 반환하며, 마찬가지로 리스트 컴프리헨션에 의해 복제됩니다. **predicate** 는 어떤 조건의 진릿값을 반환하는 함수입니다; `filter()` 와 함께 사용하는 경우, `predicate` 는 단일 값을 받아들여야 합니다.

```
>>> def is_even(x):
...     return (x % 2) == 0
```

```
>>> list(filter(is_even, range(10)))
[0, 2, 4, 6, 8]
```

또한 이것은 리스트 컴프리헨션으로 작성될 수 있습니다:

```
>>> list(x for x in range(10) if is_even(x))
[0, 2, 4, 6, 8]
```

`enumerate(iter, start=0)` 는 카운트(*start* 부터)와 각 요소를 포함하는 2-튜플을 반환하는 이터러블의 요소를 계산합니다.

```
>>> for item in enumerate(['subject', 'verb', 'object']):
...     print(item)
(0, 'subject')
(1, 'verb')
(2, 'object')
```

`enumerate()` 는 리스트를 반복하고 특정 조건이 충족되는 인덱스를 기록할 때 자주 사용됩니다:

```
f = open('data.txt', 'r')
for i, line in enumerate(f):
    if line.strip() == '':
        print('Blank line at line #%i' % i)
```

`sorted(iterable, key=None, reverse=False)` 는 이터러블의 모든 요소를 리스트로 모으고, 리스트를 정렬하고, 정렬된 결과를 반환합니다. *key* 와 *reverse* 인자는 생성된 리스트의 `sort()` 메서드로 전달됩니다.

```
>>> import random
>>> # Generate 8 random numbers between [0, 10000)
>>> rand_list = random.sample(range(10000), 8)
```

(다음 페이지에 계속)

(이전 페이지에서 계속)

```
>>> rand_list
[769, 7953, 9828, 6431, 8442, 9878, 6213, 2207]
>>> sorted(rand_list)
[769, 2207, 6213, 6431, 7953, 8442, 9828, 9878]
>>> sorted(rand_list, reverse=True)
[9878, 9828, 8442, 7953, 6431, 6213, 2207, 769]
```

(정렬에 대한 자세한 설명은 [sortinghowto](#) 를 참고하세요.)

`any(iter)` 및 `all(iter)` 내장 함수는 이터러블의 진릿값을 봅니다. `any()` 는 이터러블의 어떤 요소가 참이면 `True` 를 반환하고, `all()` 은 모든 요소가 참이면 `True` 를 반환합니다:

```
>>> any([0, 1, 0])
True
>>> any([0, 0, 0])
False
>>> any([1, 1, 1])
True
>>> all([0, 1, 0])
False
>>> all([0, 0, 0])
False
>>> all([1, 1, 1])
True
```

`zip(iterA, iterB, ...)` 은 각 이터러블에서 하나의 요소를 취하여 튜플로 반환합니다:

```
zip(['a', 'b', 'c'], (1, 2, 3)) =>
('a', 1), ('b', 2), ('c', 3)
```

이 함수는 결과를 반환하기 전에 메모리 내의 리스트를 구성하거나 모든 입력 이터레이터를 처리하지 않습니다; 대신 튜플은 요청된 경우에만 생성하여 반환합니다. (이 동작의 전문 용어는 *느긋한 평가* 입니다.)

이 이터레이터는 모두 같은 길이의 이터러블과 함께 사용하기 위한 것입니다. 이터러블의 길이가 다른 경우 결과 스트림은 가장 짧은 이터러블과 같은 길이가 됩니다.

```
zip(['a', 'b'], (1, 2, 3)) =>
('a', 1), ('b', 2)
```

더 긴 이터레이터에서 나머지 요소는 버려질 수 있기 때문에 이런 방식은 피해야 합니다. 즉, 삭제된 요소를 건너뛰는 위험이 있으므로 이터레이터를 계속 사용할 수 없습니다.

## 6 itertools 모듈

`itertools` 모듈은 공통적으로 사용되는 많은 이터레이터와 몇몇 이터레이터를 결합하기 위한 함수를 포함합니다. 이 절에서는 작은 예제를 보여줌으로써 모듈의 내용을 소개합니다.

모듈의 기능은 몇 가지 광범위한 클래스로 분류됩니다:

- 기존 이터레이터를 기반으로 새로운 이터레이터를 만드는 함수.
- 이터레이터의 요소를 함수 인자로 처리하는 함수.
- 이터레이터의 출력 부분을 선택하는 함수.
- 이터레이터의 출력을 분류하는 함수.

## 6.1 새로운 이터레이터 만들기

`itertools.count(start, step)` 는 균등하게 간격을 둔 값들의 무한한 스트림을 반환합니다. 선택적으로 기본값이 0인 시작 번호와 기본값이 1인 숫자 사이의 간격을 제공할 수 있습니다:

```
itertools.count() =>
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
itertools.count(10) =>
10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
itertools.count(10, 5) =>
10, 15, 20, 25, 30, 35, 40, 45, 50, 55, ...
```

`itertools.cycle(iter)` 은 제공된 이터러블의 내용 사본을 저장하고 처음부터 마지막까지 요소를 반환하는 새로운 이터레이터를 반환합니다. 새로운 이터레이터는 이러한 요소를 무한히 반복합니다.

```
itertools.cycle([1, 2, 3, 4, 5]) =>
1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...
```

`itertools.repeat(elem, [n])` 는 제공된 요소를  $n$  번 반환하거나,  $n$  이 제공되지 않으면 끝없이 요소를 반환합니다.

```
itertools.repeat('abc') =>
abc, abc, abc, abc, abc, abc, abc, abc, abc, abc, ...
itertools.repeat('abc', 5) =>
abc, abc, abc, abc, abc
```

`itertools.chain(iterA, iterB, ...)` 은 임의의 수의 이터러블을 입력으로 취하여, 첫 번째 이터러블의 모든 요소를 반환한 다음 두 번째 요소의 모든 요소를 반환하고, 모든 이터러블이 다 소모될 때까지 이 동작을 반복합니다.

```
itertools.chain(['a', 'b', 'c'], (1, 2, 3)) =>
a, b, c, 1, 2, 3
```

`itertools.islice(iter, [start], stop, [step])` 는 이터레이터의 조각 스트림을 반환합니다. 단일 *stop* 인자를 사용하면 처음 *stop* 개 요소가 반환됩니다. 시작 인덱스를 지정하면 *stop-start* 요소가 생기고, *step* 에 값을 지정하면 요소는 그에 따라 생략됩니다. 파이썬의 문자열 및 리스트 슬라이싱과 달리, *start*, *stop*, *step* 에 음수값을 사용할 수 없습니다.

```
itertools.islice(range(10), 8) =>
0, 1, 2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8) =>
2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8, 2) =>
2, 4, 6
```

`itertools.tee(iter, [n])` 는 이터레이터를 복제합니다; 원본 이터레이터의 내용을 모두 반환하는  $n$  개의 독립적인 이터레이터를 반환합니다.  $n$  에 대한 값을 제공하지 않으면 기본값은 2입니다. 이터레이터를 복제하려면 원본 이터레이터의 일부 내용을 저장해야 하므로 이터레이터가 크고 새로운 이터레이터 중 하나가 다른 것보다 많이 소비된다면 이것은 상당한 메모리를 소비할 수 있습니다.

```
itertools.tee(itertools.count()) =>
iterA, iterB

where iterA ->
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```

(다음 페이지에 계속)

```
and iterB ->
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```

## 6.2 요소에 대한 함수 호출

operator 모듈은 파이썬의 연산자에 대응하는 함수 집합을 포함합니다. 예를 들어 operator.add(a, b) (두 개의 값을 더하기), operator.ne(a, b) (a != b 와 동일) 및 operator.attrgetter('id') (.id 이트리뷰트를 가져오는 콜러블을 반환)와 같은 함수가 있습니다.

itertools.starmap(func, iter) 은 이터러블이 튜플의 스트림을 반환할 것이라고 가정하고, 이 튜플을 인자로 사용하여 *func* 를 호출합니다:

```
itertools.starmap(os.path.join,
                  [('/bin', 'python'), ('/usr', 'bin', 'java'),
                   ('/usr', 'bin', 'perl'), ('/usr', 'bin', 'ruby')])
=>
/bin/python, /usr/bin/java, /usr/bin/perl, /usr/bin/ruby
```

## 6.3 요소 선택하기

또 다른 함수 모음은 서술자에 기초하여 이터러블 요소의 부분 집합을 선택합니다.

itertools.filterfalse(predicate, iter) 는 filter() 의 반대이며, predicate가 거짓을 반환하는 모든 요소를 반환합니다:

```
itertools.filterfalse(is_even, itertools.count()) =>
    1, 3, 5, 7, 9, 11, 13, 15, ...
```

itertools.takewhile(predicate, iter) 은 predicate가 참을 반환하는 한, 요소를 반환합니다. predicate가 거짓을 반환하면 이터레이터는 결과의 종료를 알립니다.

```
def less_than_10(x):
    return x < 10

itertools.takewhile(less_than_10, itertools.count()) =>
    0, 1, 2, 3, 4, 5, 6, 7, 8, 9

itertools.takewhile(is_even, itertools.count()) =>
    0
```

itertools.dropwhile(predicate, iter) 은 predicate가 참을 반환하는 동안 요소를 버리고, 나머지 이터러블의 결과를 반환합니다.

```
itertools.dropwhile(less_than_10, itertools.count()) =>
    10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...

itertools.dropwhile(is_even, itertools.count()) =>
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
```

itertools.compress(data, selectors) 는 두 개의 이터레이터를 취하고 selectors의 해당 요소가 참인 data의 요소만을 반환하고, 한쪽이 고갈될 때마다 중단합니다:

```
itertools.compress([1, 2, 3, 4, 5], [True, True, False, False, True]) =>
    1, 2, 5
```

## 6.4 조합 함수

`itertools.combinations(iterable, r)` 는 *iterable* 에 포함된 모든 요소의 가능한 *r*-튜플 조합을 제공하는 이터레이터를 반환합니다.

```
itertools.combinations([1, 2, 3, 4, 5], 2) =>
(1, 2), (1, 3), (1, 4), (1, 5),
(2, 3), (2, 4), (2, 5),
(3, 4), (3, 5),
(4, 5)

itertools.combinations([1, 2, 3, 4, 5], 3) =>
(1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 4), (1, 3, 5), (1, 4, 5),
(2, 3, 4), (2, 3, 5), (2, 4, 5),
(3, 4, 5)
```

각 튜플 내의 원소들은 *iterable* 이 반환한 것과 같은 순서로 유지됩니다. 예를 들어 위의 예시에서 숫자 1은 항상 2, 3, 4 또는 5 앞에 옵니다. 비슷한 함수인 `itertools.permutations(iterable, r=None)` 은 제약 조건을 제거하여 길이 *r*의 가능한 모든 순열을 반환합니다:

```
itertools.permutations([1, 2, 3, 4, 5], 2) =>
(1, 2), (1, 3), (1, 4), (1, 5),
(2, 1), (2, 3), (2, 4), (2, 5),
(3, 1), (3, 2), (3, 4), (3, 5),
(4, 1), (4, 2), (4, 3), (4, 5),
(5, 1), (5, 2), (5, 3), (5, 4)

itertools.permutations([1, 2, 3, 4, 5]) =>
(1, 2, 3, 4, 5), (1, 2, 3, 5, 4), (1, 2, 4, 3, 5),
...
(5, 4, 3, 2, 1)
```

*r*에 값을 지정하지 않으면 이터러블의 길이가 사용됩니다. 즉, 모든 요소가 치환됩니다.

이 함수는 위치별로 가능한 모든 조합을 생성하며 *iterable*의 내용이 고유해야 할 필요는 없습니다:

```
itertools.permutations('aba', 3) =>
('a', 'b', 'a'), ('a', 'a', 'b'), ('b', 'a', 'a'),
('b', 'a', 'a'), ('a', 'a', 'b'), ('a', 'b', 'a')
```

같은 튜플 ('a', 'a', 'b')가 두 번 발생하지만, 두 개의 'a' 문자열은 다른 위치에서 왔습니다.

`itertools.combinations_with_replacement(iterable, r)` 함수는 다른 제약을 완화합니다: 요소는 단일 튜플 내에서 반복될 수 있습니다. 개념적으로 요소는 각 튜플의 첫 번째 위치에 대해 선택되고 두 번째 요소가 선택되기 전에 대체됩니다.

```
itertools.combinations_with_replacement([1, 2, 3, 4, 5], 2) =>
(1, 1), (1, 2), (1, 3), (1, 4), (1, 5),
(2, 2), (2, 3), (2, 4), (2, 5),
(3, 3), (3, 4), (3, 5),
(4, 4), (4, 5),
(5, 5)
```



## 6.5 요소 분류

마지막으로 소개할 `itertools.groupby(iter, key_func=None)` 함수는 가장 복잡합니다. `key_func(elem)` 는 이터러블에 의해 반환된 각 요소에 대한 키값을 계산할 수 있는 함수입니다. 키 함수를 제공하지 않으면 키는 단순히 각 요소 자체입니다.

`groupby()` 는 이터러블 내부에서 키값이 같은 연속된 모든 요소를 수집하여 키값과 해당 키를 가진 요소의 이터러블을 포함하는 2-튜플의 스트림을 반환합니다.

```
city_list = [('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL'),
             ('Anchorage', 'AK'), ('Nome', 'AK'),
             ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ'),
             ...
            ]

def get_state(city_state):
    return city_state[1]

itertools.groupby(city_list, get_state) =>
    ('AL', iterator-1),
    ('AK', iterator-2),
    ('AZ', iterator-3), ...

where
iterator-1 =>
    ('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL')
iterator-2 =>
    ('Anchorage', 'AK'), ('Nome', 'AK')
iterator-3 =>
    ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ')
```

`groupby()` 는 이터러블 내부의 내용이 키에 따라 이미 정렬되었다고 가정합니다. 반환된 이터레이터 역시 이터러블 내부를 사용하므로 이터레이터-2와 해당 키를 요청하기 전에 이터레이터-1의 결과를 소진해야 합니다.

## 7 functools 모듈

The `functools` module in Python 2.5 contains some higher-order functions. A **higher-order function** takes one or more functions as input and returns a new function. The most useful tool in this module is the `functools.partial()` function.

함수형 방식으로 작성된 프로그램의 경우, 일부 매개 변수가 채워진 기존 함수의 변형이 필요한 경우가 있습니다. 파이썬 함수 `f(a, b, c)` 를 고려해보세요; 파이썬 함수인 `f(1, b, c)` 에 해당하는 새로운 함수 `g(b, c)` 를 만들 수 있습니다; 이를 “부분적 함수 적용” 이라고 합니다.

`partial()` 의 생성자는 (`function, arg1, arg2, ..., kwarg1=value1, kwarg2=value2`) 와 같은 인자를 취합니다. 결과 객체는 콜러블이므로, 채워진 인자로 `function` 을 실행하기 위해서는 결과 객체를 호출하면 됩니다.

작지만 현실적인 예가 있습니다:

```
import functools

def log(message, subsystem):
    """Write the contents of 'message' to the specified subsystem."""
    print('%s: %s' % (subsystem, message))
```

(다음 페이지에 계속)

```
...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

`functools.reduce(func, iter, [initial_value])` 는 모든 이터러블 요소에 대해 누적 연산을 수행하므로 무한 이터러블에 적용할 수 없습니다. *func* 는 두 요소를 사용하여 하나의 값을 반환하는 함수이어야 합니다. `functools.reduce()` 는 이터레이터가 반환한 처음 두 요소 A와 B를 취해 `func(A, B)` 를 계산합니다. 그다음 세 번째 요소인 C를 취해 `func(func(A, B), C)` 를 계산하고, 이 결과를 반환된 네 번째 요소와 결합해 이터러블이 소진될 때까지 계속합니다. 이터러블이 전혀 값을 반환하지 않으면 `TypeError` 예외가 발생합니다. 초기값이 제공되면 시작점으로 사용되며 `func(initial_value, A)` 가 첫 번째로 계산됩니다.

```
>>> import operator, functools
>>> functools.reduce(operator.concat, ['A', 'BB', 'C'])
'ABBC'
>>> functools.reduce(operator.concat, [])
Traceback (most recent call last):
...
TypeError: reduce() of empty sequence with no initial value
>>> functools.reduce(operator.mul, [1, 2, 3], 1)
6
>>> functools.reduce(operator.mul, [], 1)
1
```

`operator.add()` 를 `functools.reduce()` 와 함께 사용하면 이터러블의 모든 요소를 합합니다. 이 경우는 매우 일반적이어서 이를 계산하기 위해 `sum()` 이라는 특수 내장 함수가 제공됩니다:

```
>>> import functools, operator
>>> functools.reduce(operator.add, [1, 2, 3, 4], 0)
10
>>> sum([1, 2, 3, 4])
10
>>> sum([])
0
```

그렇지만 `functools.reduce()` 를 사용하는 많은 경우에 명백하게 `for` 루프만 작성하는 것이 더 명확할 수 있습니다:

```
import functools
# Instead of:
product = functools.reduce(operator.mul, [1, 2, 3], 1)

# You can write:
product = 1
for i in [1, 2, 3]:
    product *= i
```

A related function is `itertools.accumulate(iterable, func=operator.add)`. It performs the same calculation, but instead of returning only the final result, `accumulate()` returns an iterator that also yields each partial result:

```
itertools.accumulate([1, 2, 3, 4, 5]) =>
1, 3, 6, 10, 15
```

```
itertools.accumulate([1, 2, 3, 4, 5], operator.mul) =>
1, 2, 6, 24, 120
```

## 7.1 operator 모듈

`operator` 모듈은 이전에 언급되었습니다. 여기에는 파이썬 연산자에 해당하는 함수 집합이 포함되어 있습니다. 단일 연산을 수행하는 사소한 함수를 작성하지 않아도 되므로 이러한 함수는 함수형 방식 코드에서 유용합니다.

이 모듈의 몇몇 함수는 다음과 같습니다:

- 수학 연산: `add()`, `sub()`, `mul()`, `floordiv()`, `abs()`, ...
- 논리 연산: `not_()`, `truth()`.
- 비트 연산: `and_()`, `or_()`, `invert()`.
- 비교: `eq()`, `ne()`, `lt()`, `le()`, `gt()`, `ge()`.
- 객체 아이덴티티: `is_()`, `is_not()`.

전체 목록은 연산자 모듈의 문서를 참고하세요.

## 8 작은 함수와 람다 표현식

함수형 방식의 프로그램을 작성할 때, 서술자로 동작하거나 어떤 식으로든 요소를 결합하는 작은 함수가 필요할 것입니다.

파이썬 내장 함수나 적당한 모듈 함수가 있다면, 새로운 함수를 정의할 필요가 전혀 없습니다:

```
stripped_lines = [line.strip() for line in lines]
existing_files = filter(os.path.exists, file_list)
```

필요한 기능이 없다면 작성해야 합니다. 작은 함수를 작성하는 한 가지 방법은 `lambda` 표현식을 사용하는 것입니다. `lambda` 는 여러 매개 변수와 이들 매개 변수를 결합하는 표현식을 취해 표현식의 값을 반환하는 익명의 함수를 만듭니다:

```
adder = lambda x, y: x+y

print_assign = lambda name, value: name + '=' + str(value)
```

다른 방법은 `def` 문을 사용하고 일반적인 방식으로 함수를 정의하는 것입니다:

```
def adder(x, y):
    return x + y

def print_assign(name, value):
    return name + '=' + str(value)
```

어떤 대안이 바람직할까요? 이것은 스타일에 대한 질문입니다; 필자가 평소에 사용하는 방법은 `lambda` 사용을 피하는 것입니다.

필자가 선호하는 방식에 대한 이유 중 하나는 `lambda` 가 정의할 수 있는 함수가 상당히 제한적이기 때문입니다. 결과는 단일 표현식으로 계산할 수 있어야 합니다. 즉, `if... elif... else` 비교 또는 `try... except` 문을 가질 수 없습니다. `lambda` 문에서 너무 많은 것을 하려고 하면, 읽기 어려운 복잡한 표현으로 끝날 것입니다. 다음 코드가 무엇을 하는지 빠르게 알아보세요.

```
import functools
total = functools.reduce(lambda a, b: (0, a[1] + b[1]), items)[1]
```

여러분은 이해할 수 있지만, 어떻게 동작하는지 이해하기 위해 표현식을 풀어내는 데 시간이 걸립니다. 짧게 중첩된 def 문을 사용하면 좀 더 나은 것을 만들 수 있습니다:

```
import functools
def combine(a, b):
    return 0, a[1] + b[1]

total = functools.reduce(combine, items)[1]
```

그러나 단순히 for 루프를 사용했다면 가장 좋았을 것입니다:

```
total = 0
for a, b in items:
    total += b
```

혹은 sum() 내장 함수와 제너레이터 표현식이었어도 좋았을 것입니다:

```
total = sum(b for a, b in items)
```

functools.reduce() 를 사용하는 많은 경우, for 루프로 작성했을 때 더 명확합니다.

Fredrik Lundh는 한때 lambda 사용법의 리팩토링을 위해 다음과 같은 규칙 집합을 제안했습니다:

1. 람다 함수를 작성하세요.
2. 람다가 하는 일에 관해 설명하는 글을 쓰세요.
3. 잠깐 설명을 검토하고 설명의 본질을 포착하는 이름을 생각해 보세요.
4. 해당 이름을 사용하여 람다를 def 문으로 변환합니다.
5. 설명을 삭제하세요.

필자는 이 규칙을 정말 좋아하지만, 여러분은 이렇게 람다가 없는 방식이 더 나은지에 대해 동의하지 않을 수 있습니다.

## 9 개정내역 및 감사의 글

필자는 이 글의 다양한 초안을 제안하고 수정하고 도와준 다음 사람들에게 감사하고 싶습니다: Ian Bicking, Nick Coghlan, Nick Efford, Raymond Hettinger, Jim Jewett, Mike Krell, Leandro Lameiro, Jussi Salmela, Collin Winter, Blake Winton.

버전 0.1: 2006년 6월 30일 게시.

버전 0.11: 2006년 7월 1일 게시. 오타 수정.

버전 0.2: 2006년 7월 10일 게시. 제너레이터 표현식과 리스트 컴프리헨션 섹션을 하나로 통합. 오타 수정.

버전 0.21: 튜터 메일링 리스트에서 추천된 참고 문헌을 추가.

버전 0.30: Collin Winter가 작성한 functional 모듈에 대한 섹션 추가; 연산자 모듈에 대한 짧은 섹션 추가; 몇 가지 다른 편집.

## 10 참고 문헌

### 10.1 일반

**Structure and Interpretation of Computer Programs**, by Harold Abelson and Gerald Jay Sussman with Julie Sussman. Full text at <https://mitpress.mit.edu/sicp/>. In this classic textbook of computer science, chapters 2 and 3 discuss the use of sequences and streams to organize the data flow inside a program. The book uses Scheme for its examples, but many of the design approaches described in these chapters are applicable to functional-style Python code.

<https://www.defmacro.org/ramblings/fp.html>: 오랜 역사적 소개와 함께 자바 예제를 사용한 함수형 프로그래밍에 대한 일반적인 개론.

[https://en.wikipedia.org/wiki/Functional\\_programming](https://en.wikipedia.org/wiki/Functional_programming): 함수형 프로그래밍을 설명하는 일반 위키피디아 항목.

<https://en.wikipedia.org/wiki/Coroutine>: 코루틴에 대한 항목.

<https://en.wikipedia.org/wiki/Currying>: 커링 개념에 대한 항목.

### 10.2 파이썬 특정

<https://gnosis.cx/TPiP/>: David Mertz의 책 *Text Processing in Python*의 첫 번째 장에서는 “Utilizing Higher-Order Functions in Text Processing” 절에서 텍스트 처리를 위한 함수형 프로그래밍에 관해 설명합니다.

Mertz는 또한 IBM의 DeveloperWorks 사이트에서 함수형 프로그래밍 기사 시리즈 3부작을 작성했습니다; [part 1](#), [part 2](#), [part 3](#),

### 10.3 파이썬 설명서

`itertools` 모듈에 대한 설명서

`functools` 모듈에 대한 설명서

`operator` 모듈에 대한 설명서

**PEP 289**: “제너레이터 표현식”

**PEP 342**: “개선된 제너레이터를 통한 코루틴”은 파이썬 2.5의 새로운 제너레이터 기능을 설명합니다.

## 색인

### Y

파이썬 항상 제안

PEP 289, 20

PEP 342, 9, 20