```
# Khối mã này cung cấp cách bạn có thể tải các hình ảnh có trong chương sách này.
```

```
from os.path import basename, exists

def download(url):
    filename = basename(url)
    if not exists(filename):
        from urllib.request import urlretrieve

        local, _ = urlretrieve(url, filename)
        print("Downloaded " + str(local))
    return filename

download('https://github.com/AllenDowney/ThinkPython/raw/v3/thinkpython.py');
download('https://github.com/AllenDowney/ThinkPython/raw/v3/diagram.py');

import thinkpython
```

```
Downloaded thinkpython.py
Downloaded diagram.py
```

Hàm trong Python

Trong chương trước, chúng ta đã sử dụng một số hàm được cung cấp bởi Python, như `int` và `float`, và một vài hàm từ mô-đun `math` như `sqrt` và `pow`.

Trong chương này, bạn sẽ học cách tạo ra các hàm của riêng mình và thực thi chúng. Chúng ta cũng sẽ học cách một hàm có thể gọi từ một hàm khác. Ví dụ, chúng ta sẽ hiển thị lời bài hát từ *Monty Python*. Những ví dụ hài hước này sẽ minh họa một đặc điểm quan trọng chính là khả năng tạo ra hàm riêng của bạn là nền tảng của việc lập trình.

Chương này cũng giới thiệu một câu lệnh mới đó là vòng lặp `for`, được sử dụng để **lặp lại** một phép tính.

Định nghĩa hàm

Một **định nghĩa hàm** xác định tên của một hàm mới và chuỗi các câu lệnh sẽ chạy khi hàm đó được gọi. Đây là một ví dụ:

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

`def` là một từ khóa cho biết rằng đây là một định nghĩa hàm. Tên của hàm là `print_lyrics`. Bất cứ thứ gì là tên biến hợp lệ thì cũng có thể là tên hàm hợp lệ.

Dấu ngoặc đơn trống sau tên hàm cho thấy rằng hàm này không nhận bất kỳ tham số nào.

Dòng đầu tiên của định nghĩa hàm được gọi là **phần đầu** -- phần còn lại được gọi là **phần thân**. Phần đầu phải kết thúc bằng dấu hai chấm, và phần thân phải được thụt dòng. Theo quy ước, khoảng thụt dòng luôn là bốn dấu cách. Phần thân của hàm này là hai câu lệnh `print`; nhìn chung, phần thân của một hàm có thể chứa bất kỳ số lượng câu lệnh nào thuộc bất kỳ loại nào.

Định nghĩa một hàm sẽ tạo ra một **đối tượng hàm**, mà chúng ta có thể hiển thị như sau.

```
print_lyrics
```

```
print_lyrics
def print_lyrics()
```

```
<no docstring>
```

Kết quả đầu ra cho thấy `print_lyrics` là một hàm không nhận tham số nào. `__main__` là tên của mô-đun chứa hàm `print_lyrics`.

Bây giờ, khi đã định nghĩa hàm, chúng ta có thể gọi nó theo cách tương tự như khi gọi các hàm tích hợp sẵn trong Python.

```
print_lyrics()
```

```
I'm a lumberjack, and I'm okay.
I sleep all night and I work all day.
```

Khi hàm được chạy, nó thực thi các câu lệnh trong phần thân, hiển thị hai dòng đầu tiên của bài *The Lumberjack Song*.

Các tham số

Một số hàm mà chúng ta đã thấy yêu cầu tham số. Ví dụ, khi bạn gọi hàm `abs` , bạn truyền vào một số làm tham số.

Một số hàm nhận nhiều hơn một tham số. Ví dụ, hàm `math.pow` nhận hai tham số là *cơ số* và *số mũ*.

Dưới đây là định nghĩa cho một hàm nhận một tham số.

```
def print_twice(string):
    print(string)
    print(string)
```

Tên biến trong dấu ngoặc đơn được gọi là **một tham số**. Khi hàm được gọi, giá trị của đối số được gán cho tham số. Ví dụ, chúng ta có thể gọi hàm `print_twice` như sau.

```
print_twice('Dennis Moore, ')
```

```
Dennis Moore,
Dennis Moore,
```

Chạy hàm này có cùng tác dụng như việc gán đối số cho tham số và sau đó thực thi phần thân của hàm, như sau.

```
string = 'Dennis Moore, '
print(string)
print(string)
```

```
Dennis Moore,
Dennis Moore,
```

Bạn cũng có thể sử dụng một biến làm tham số.

```
line = 'Dennis Moore, '
print_twice(line)
```

```
Dennis Moore,
Dennis Moore,
```

Trong ví dụ này, giá trị của biến `line` được gán cho tham số `string` trong khối mã bên trên.

Gọi hàm

Khi bạn đã định nghĩa một hàm, bạn có thể sử dụng nó bên trong một hàm khác. Để minh họa, chúng ta sẽ viết các hàm để in lời bài hát [The Spam Song]

> Spam, Spam, Spam, Spam,
> Spam, Spam, Spam, Spam,
> Spam, Spam,
> (Lovely Spam, Wonderful Spam!)
> Spam, Spam,

Chúng ta sẽ bắt đầu với hàm sau đây, hàm này nhận hai tham số.

```
def repeat(word, n):
    print(word * n)
```

Chúng ta có thể sử dụng hàm này để in dòng đầu tiên của bài hát như sau.

```
spam = 'Spam, '
repeat(spam, 4)
```

```
Spam, Spam, Spam, Spam,
```

Để hiển thị hai dòng đầu tiên, chúng ta có thể định nghĩa một hàm mới sử dụng `repeat` .

```
def first_two_lines():
    repeat(spam, 4)
    repeat(spam, 4)
```

Và sau đó gọi nó như sau.

```
first_two_lines()
```

```
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
```

Để hiển thị ba dòng cuối, chúng ta có thể định nghĩa một hàm khác, cũng sử dụng `repeat`.

```
def last_three_lines():
    repeat(spam, 2)
    print('(Lovely Spam, Wonderful Spam!)')
    repeat(spam, 2)


last_three_lines()
```

```
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,
```

Cuối cùng, chúng ta có thể kết hợp tất cả lại với một hàm để in toàn bộ lời bài hát.

```
def print_verse():
    first_two_lines()
    last_three_lines()


print_verse()
```

```
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,
```

Khi chúng ta chạy `print_verse`, nó sẽ gọi `first_two_lines`, rồi gọi `repeat`, và `repeat` lại gọi `print`. Đó là rất nhiều hàm. Tất nhiên, chúng ta có thể làm điều tương tự với ít hàm hơn, nhưng mục đích của ví dụ này là để cho thấy cách các hàm có thể hoạt động cùng nhau.

Sự lặp lại

Nếu chúng ta muốn hiển thị nhiều hơn một đoạn lời của bài hát, chúng ta có thể sử dụng câu lệnh `for`. Đây là một ví dụ đơn giản.

```
for i in range(2):
  print(i)
```

```
0
1
```

Dòng đầu tiên là tiêu đề kết thúc bằng dấu hai chấm. Dòng thứ hai là nội dung, phải được thụt dòng.

Tiêu đề bắt đầu bằng từ khóa `for`, một biến mới có tên là `i`, và một từ khóa khác là `in`. Nó sử dụng hàm `range` để tạo ra một chuỗi hai giá trị, đó là `0` và `1`. Trong Python, khi chúng ta bắt đầu đếm, chúng ta thường bắt đầu từ `0`.

Khi câu lệnh `for` chạy, nó gán giá trị đầu tiên từ `range` cho `i` và sau đó chạy hàm `print` trong nội dung, hiển thị `0`.

Khi đến cuối nội dung, nó quay lại tiêu đề, đó là lý do tại sao câu lệnh này được gọi là một vòng lặp. Lần thứ hai qua vòng lặp, nó gán giá trị tiếp theo từ `range` cho `i` và hiển thị nó. Sau đó, vì đó là giá trị cuối cùng từ `range`, vòng lặp kết thúc.

Dưới đây là cách chúng ta có thể sử dụng vòng lặp `for` để in hai từ đoạn lời của bài hát.

```
for i in range(2):
    print("Verse", i)
    print_verse()
    print()
```

```
Verse 0
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,

Verse 1
Spam, Spam, Spam, Spam,
Spam, Spam, Spam, Spam,
Spam, Spam,
(Lovely Spam, Wonderful Spam!)
Spam, Spam,
```

Bạn có thể đặt một vòng lặp `for` bên trong một hàm. Ví dụ, `print_n_verses` nhận một tham số có tên là `n`, tham số này phải là một số nguyên, và hiển thị số đoạn đã cho.

```
def print_n_verses(n):
    for i in range(n):
        print_verse()
        print()
```

Trong ví dụ này, chúng ta không sử dụng `i` trong phần thân của vòng lặp, nhưng vẫn cần có một tên biến trong tiêu đề.

Các biến và tham số là cục bộ

Khi bạn tạo một biến bên trong một hàm, nó là **cục bộ**, có nghĩa là nó chỉ tồn tại bên trong hàm đó. Ví dụ, hàm sau đây nhận hai tham số, nối chúng lại với nhau và in kết quả hai lần.

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

Dưới đây là một ví dụ sử dụng nó:

```
line1 = 'Always look on the '
line2 = 'bright side of life.'
cat_twice(line1, line2)
```

```
Always look on the bright side of life.
Always look on the bright side of life.
```

Khi `cat_twice` chạy, nó tạo ra một biến cục bộ có tên là `cat`, và biến này sẽ bị xóa khi hàm kết thúc. Nếu chúng ta cố gắng hiển thị nó, chúng ta sẽ nhận được lỗi `NameError`:

```
%%expect NameError

print(cat)
```

```
NameError: name 'cat' is not defined
```

Ngoài hàm `cat` không được định nghĩa.

Các tham số cũng là cục bộ. Ví dụ, bên ngoài `cat_twice`, không có khái niệm nào về `part1` hoặc `part2`

Sơ đồ ngăn xếp

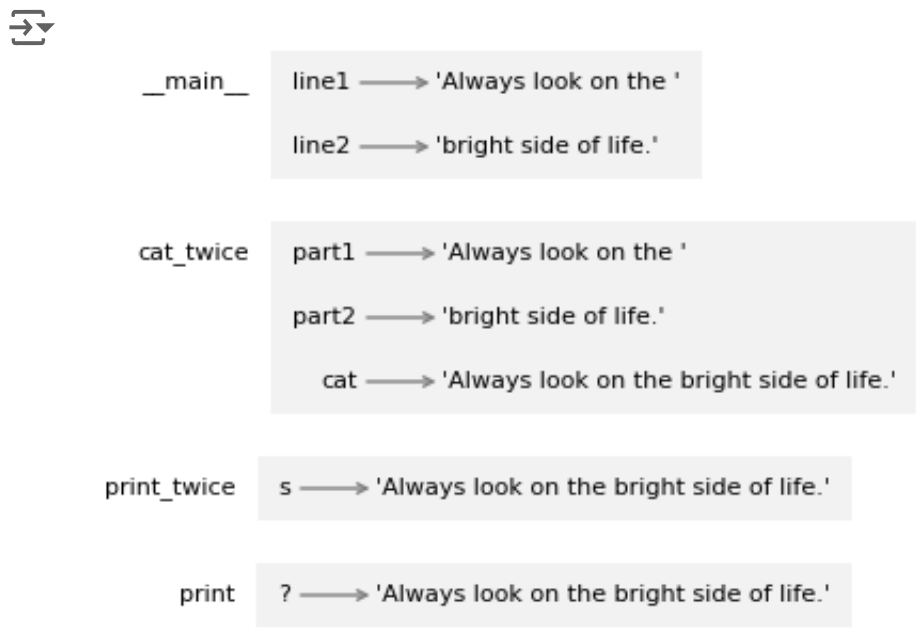
Để theo dõi biến nào có thể được sử dụng ở đâu, đôi khi thật hữu ích khi vẽ một **sơ đồ ngăn xếp**. Giống như sơ đồ trạng thái, sơ đồ ngăn xếp cho thấy giá trị của mỗi biến, nhưng nó cũng cho thấy hàm mà mỗi biến thuộc về.

Mỗi hàm được đại diện bởi **một khung**. Một khung là một hình hộp với tên của hàm ở bên ngoài và các tham số cùng biến cục bộ của hàm ở bên trong.

Dưới đây là sơ đồ ngăn xếp cho ví dụ trước.

```
from diagram import make_frame, Stack

d1 = dict(line1=line1, line2=line2)
frame1 = make_frame(d1, name='__main__', dy=-0.3, loc='left')

d2 = dict(part1=line1, part2=line2, cat=line1+line2)
frame2 = make_frame(d2, name='cat_twice', dy=-0.3,
                    offsetx=0.03, loc='left')

d3 = dict(s = line1+line2)
frame3 = make_frame(d3, name='print_twice',
                    offsetx=-0.28, offsety=-0.3, loc='left')

d4 = {"?": line1+line2}
frame4 = make_frame(d4, name='print',
                    offsetx=-0.28, offsety=0, loc='left')

stack = Stack([frame1, frame2, frame3, frame4], dy=-0.8)
```

```
from diagram import diagram, adjust


width, height, x, y = [3.8, 2.91, 1.15, 2.66]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
#adjust(x, y, bbox)
```



Các khung được sắp xếp theo dạng ngăn xếp cho biết hàm nào đã gọi hàm nào, và cứ như vậy. Đọc từ dưới lên, `print` được gọi bởi `print_twice`, `print_twice` được gọi bởi `cat_twice`, và `cat_twice` được gọi bởi `__main__`, và đây là một tên đặc biệt cho khung ở trên cùng. Khi bạn tạo một biến bên ngoài bất kỳ hàm nào, nó thuộc về `__main__`.

Trong khung của `print`, dấu hỏi cho biết rằng chúng ta không biết tên của tham số. Nếu bạn tò mò, hãy hỏi một trợ lý ảo: *Các tham số của hàm `print()` trong Python là gì?*

Dấu vết lỗi

Khi một lỗi thời gian chạy xảy ra trong một hàm, Python hiển thị tên của hàm đang chạy, tên của hàm đã gọi nó, và cứ như vậy, theo ngăn xếp.

Để xem một ví dụ, tôi sẽ định nghĩa một phiên bản của `print_twice` chứa lỗi. Nó đang cố gắng in `cat`, một biến cục bộ trong hàm khác.

```
def print_twice(string):
    print(cat)            # NameError
    print(cat)
```

Bây giờ, hãy xem điều gì xảy ra khi chúng ta chạy `cat_twice`.

```
# Ô này yêu cầu Jupyter cung cấp thông tin gỡ lỗi chi tiết
# khi xảy ra lỗi thời gian chạy, bao gồm cả thông tin về dấu vết lỗi.

%xmode Verbose
```

```
Exception reporting mode: Verbose
```

```
%%expect NameError

cat_twice(line1, line2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-27-51d38e24da48> in <cell line: 1>()
----> 1 cat_twice(line1, line2)
        global cat_twice = <function cat_twice at 0x79f86d4680d0>
        global line1 = 'Always look on the '
        global line2 = 'bright side of life.'

 ---------------------------------------- 1 frames ----------------------------------------
<ipython-input-25-192e4bfef986> in print_twice(string='Always look on the
bright side of life.')
      1 def print_twice(string):
----> 2     print(cat)            # NameError
        global print = undefined
        global cat = undefined
      3     print(cat)

NameError: name 'cat' is not defined
```

Thông báo lỗi bao gồm **một dấu vết lỗi**, cho thấy hàm nào đang chạy khi lỗi xảy ra, hàm nào đã gọi nó, và cứ như vậy. Trong ví dụ này, nó cho thấy rằng `cat_twice` đã gọi `print_twice`, và lỗi xảy ra trong `print_twice`.

Thứ tự của các hàm trong dấu vết lỗi giống như thứ tự của các khung trong sơ đồ ngăn xếp. Hàm đang chạy nằm ở dưới cùng

Tại sao lại sử dụng hàm?

Có thể bạn chưa rõ tại sao lại dành công sức để chia một chương trình thành các hàm. Có một số lý do:

- Tạo một hàm mới cho bạn cơ hội để đặt tên cho một nhóm các câu lệnh, điều này giúp chương trình của bạn dễ đọc và dễ gỡ lỗi hơn.
- Hàm có thể làm cho chương trình nhỏ hơn bằng cách loại bỏ mã lặp lại. Sau này, nếu bạn cần thay đổi, bạn chỉ cần thực hiện ở một nơi.
- Chia một chương trình dài thành các hàm cho phép bạn gỡ lỗi từng phần một và sau đó lắp ráp chúng thành một tổng thể hoạt động.
- Các hàm được thiết kế tốt thường hữu ích cho nhiều chương trình. Khi bạn viết và gỡ lỗi một hàm, bạn có thể tái sử dụng nó.

Gỡ lỗi

Gỡ lỗi có thể là một quá trình gây khó chịu, nhưng nó cũng đầy thách thức, thú vị và đôi khi thậm chí là vui vẻ. Đây là một trong những kỹ năng quan trọng nhất mà bạn có thể học.

Theo một cách nào đó, gỡ lỗi giống như công việc của một thám tử. Bạn được cung cấp những manh mối và bạn phải suy luận về những sự kiện đã dẫn đến kết quả mà bạn thấy.

Gỡ lỗi cũng giống như khoa học thực nghiệm. Khi bạn có ý tưởng về những gì đang sai, bạn sửa đổi chương trình của mình và thử lại. Nếu giả thuyết của bạn là đúng, bạn có thể dự đoán kết quả của sự sửa đổi, và bạn tiến gần hơn đến một chương trình hoạt động. Nếu giả thuyết của bạn sai, bạn phải đưa ra một giả thuyết mới.

Đối với một số người, lập trình và gỡ lỗi là cùng một việc; tức là, lập trình là quá trình dần dần gỡ lỗi một chương trình cho đến khi nó làm những gì bạn muốn. Ý tưởng là bạn nên bắt đầu với một chương trình hoạt động và thực hiện các sửa đổi nhỏ, gỡ lỗi chúng trong quá trình làm.

Nếu bạn thấy mình dành quá nhiều thời gian để gỡ lỗi, điều đó thường là dấu hiệu cho thấy bạn đang viết quá nhiều mã trước khi bắt đầu thử nghiệm. Nếu bạn thực hiện các bước nhỏ hơn, bạn có thể thấy rằng mình có thể di chuyển nhanh hơn.

Thuật ngữ

- **function definition - định nghĩa hàm:** Một câu lệnh tạo ra một hàm.
- **header - tiêu đề:** Dòng đầu tiên của một định nghĩa hàm.
- **body - nội dung:** Chuỗi các câu lệnh bên trong một định nghĩa hàm.
- **function object - đối tượng hàm:** Một giá trị được tạo ra bởi một định nghĩa hàm. Tên của hàm là một biến tham chiếu đến hàm đối tượng.
- **parameter - tham số:** Một tên được sử dụng bên trong một hàm để tham chiếu đến giá trị được truyền vào như một đối số.
- **loop - vòng lặp:** Một câu lệnh chạy một hoặc nhiều câu lệnh, thường là lặp đi lặp lại.
- **local variable - biến cục bộ:** Một biến được định nghĩa bên trong một hàm và chỉ có thể được truy cập bên trong hàm đó.
- **stack diagram - sơ đồ ngăn xếp:** Một biểu diễn đồ họa của một ngăn xếp các hàm, các biến của chúng, và các giá trị mà chúng tham chiếu đến.
- **frame - khung:** Một hình hộp trong sơ đồ ngăn xếp đại diện cho một cuộc gọi hàm. Nó chứa các biến cục bộ và tham số của hàm.
- **traceback - dấu vết lỗi:** Một danh sách các hàm đang thực thi, được in ra khi một ngoại lệ xảy ra.

Bài tập

```
# Ô này yêu cầu Jupyter cung cấp thông tin gỡ lỗi chi tiết
# Khi xảy ra lỗi thời gian chạy. Chạy nó trước khi làm các bài tập.

%xmode Verbose
```

```
Exception reporting mode: Verbose
```

Hỏi trợ lý ảo

Các câu lệnh trong một hàm hoặc vòng lặp `for` thường được thụt vào bốn khoảng trắng theo quy ước. Nhưng không phải ai cũng đồng ý với quy ước đó. Nếu bạn tò mò về lịch sử của cuộc tranh luận này, hãy hỏi một trợ lý ảo "cho tôi biết về khoảng trắng và tab trong Python".

Trợ lý ảo cũng khá giỏi trong việc viết các hàm nhỏ.

1. Hãy hỏi trợ lý ảo yêu thích của bạn: "Viết một hàm gọi là `repeat` nhận một chuỗi và một số nguyên, và in chuỗi đó ra số lần đã cho."
2. Nếu kết quả sử dụng một vòng lặp `for`, bạn có thể hỏi: "Bạn có thể làm điều đó mà không cần vòng lặp for không?"
3. Chọn bất kỳ hàm nào khác trong chương này và yêu cầu một trợ lý ảo viết nó. Thách thức là mô tả hàm đủ chính xác để nhận được điều bạn muốn. Sử dụng từ vựng mà bạn đã học cho đến nay trong cuốn sách này.

Trợ lý ảo cũng khá giỏi trong việc gỡ lỗi các hàm.

1. Hãy hỏi trợ lý ảo có vấn đề gì với phiên bản này của `print_twice`.

```
def print_twice(string):
    print(cat)
    print(cat)
```

Và nếu bạn gặp khó khăn với bất kỳ bài tập nào dưới đây, hãy xem xét việc hỏi một trợ lý ảo để được giúp đỡ.

Bài tập 1

Viết một hàm có tên là `print_right` nhận một chuỗi có tên là `text` làm tham số và in chuỗi đó với đủ khoảng trắng ở đầu để chữ cái cuối cùng của chuỗi nằm ở cột thứ 40 trên màn hình.

Gợi ý: Sử dụng hàm `len`, toán tử nối chuỗi (`+`) và toán tử lặp chuỗi (`*`).

Dưới đây là một ví dụ cho thấy nó nên hoạt động như thế nào.

```
print_right("Monty")
print_right("Python's")
print_right("Flying Circus")
```

Bài tập 2

Viết một hàm có tên là `triangle` nhận một chuỗi và một số nguyên, và vẽ một kim tự tháp với chiều cao đã cho, được tạo thành từ các bản sao của chuỗi đó. Dưới đây là một ví dụ về kim tự tháp với `5` cấp độ, sử dụng chuỗi `L`.

```
triangle('L', 5)
```

Bài tập 3

Viết một hàm có tên là `rectangle` nhận một chuỗi và hai số nguyên, và vẽ một hình chữ nhật với chiều rộng và chiều cao đã cho, được tạo thành từ các bản sao của chuỗi đó. Dưới đây là một ví dụ về hình chữ nhật với chiều rộng `5` và chiều cao `4`, được tạo thành từ chuỗi `H`.

```
rectangle('H', 5, 4)
```

Bài tập 4

Bài hát "99 Bottles of Beer" bắt đầu với đoạn điệp khúc sau:

> 99 bottles of beer on the wall
> 99 bottles of beer
> Take one down, pass it around
> 98 bottles of beer on the wall

Sau đó, đoạn điệp khúc thứ hai cũng giống như vậy, chỉ khác là bắt đầu với 98 và kết thúc với 97. Bài hát tiếp tục trong một thời gian rất dài cho đến khi không còn chai bia nào.

Viết một hàm có tên là `bottle_verse` nhận một số làm tham số và hiển thị đoạn điệp khúc bắt đầu với số chai đã cho.

Gợi ý: Hãy xem xét việc bắt đầu với một hàm có thể in dòng đầu tiên, thứ hai hoặc dòng cuối cùng của đoạn điệp khúc, và sau đó sử dụng nó để viết `bottle_verse`.

Sử dụng lời gọi hàm bên dưới để hiển thị đoạn điệp khúc đầu tiên.

```
bottle_verse(99)
```

Nếu bạn muốn in toàn bộ bài hát, bạn có thể sử dụng vòng lặp `for` này, vòng lặp đếm ngược từ `99` đến `1`. Bạn không cần phải hoàn toàn hiểu ví dụ này — chúng ta sẽ học thêm về vòng lặp `for` và hàm `range` sau.

```
for n in range(99, 0, -1):
    bottle_verse(n)
    print()
```