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

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
```

Trả về giá trị

Trong các chương trước, chúng ta đã sử dụng các hàm được tích hợp sẵn — như `abs` và `round` — và các hàm trong mô-đun toán học — như `sqrt` và `pow`. Khi bạn gọi một trong những hàm này, nó sẽ trả về một giá trị mà bạn có thể gán cho một biến hoặc sử dụng như một phần của biểu thức.

Các hàm mà chúng ta đã viết cho đến nay thì khác. Một số hàm sử dụng hàm `print` để hiển thị giá trị, và một số hàm sử dụng các hàm của mô-đun `turtle` để vẽ hình. Nhưng chúng không trả về giá trị mà chúng ta gán cho các biến hoặc sử dụng trong biểu thức.

Trong chương này, chúng ta sẽ xem cách viết các hàm trả về giá trị.

Một số hàm có trả về giá trị

Khi bạn gọi một hàm như `math.sqrt`, kết quả được gọi là **trả về giá trị**. Nếu lời gọi hàm xuất hiện ở cuối một ô, `Jupyter` sẽ hiển thị trả về giá trị ngay lập tức.

```
import math
```

```
math.sqrt(42 / math.pi)
```

Nếu bạn gán trả về giá trị cho một biến, nó sẽ không được hiển thị.

```
radius = math.sqrt(42 / math.pi)
```

Nhưng bạn có thể hiển thị nó sau đó.

```
radius
```

Hoặc bạn có thể sử dụng trả về giá trị như một phần của biểu thức.

```
radius + math.sqrt(42 / math.pi)
```

Dưới đây là một ví dụ về một hàm trả về một giá trị.

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

Hàm `circle_area` nhận bán kính làm tham số và tính diện tích của một hình tròn với bán kính đó.

Dòng cuối cùng là một câu lệnh `return` trả về giá trị của `area`.

Nếu chúng ta gọi hàm như thế này, `Jupyter` sẽ hiển thị trả về giá trị.

```
circle_area(radius)
```

Chúng ta có thể gán trả về giá trị cho một biến.

```
a = circle_area(radius)
```

Hoặc sử dụng nó như một phần của biểu thức.

```
circle_area(radius) + 2 * circle_area(radius / 2)
```

Sau đó, chúng ta có thể hiển thị giá trị của biến mà chúng ta đã gán kết quả cho nó.

```
a
```

Nhưng chúng ta không thể truy cập vào `area`.

```
%%expect NameError

area
```

`area` là một biến cục bộ trong một hàm, vì vậy chúng ta không thể truy cập nó từ bên ngoài hàm.

Và một số có None

Nếu một hàm không có câu lệnh `return`, nó sẽ trả về `None`, đây là một giá trị đặc biệt giống như `True` và `False`. Ví dụ, đây là hàm `repeat` từ Chương 3.

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

Nếu chúng ta gọi nó như thế này, nó sẽ hiển thị dòng đầu tiên của bài hát "Finland" của Monty Python.

```
repeat('Finland, ', 3)
```

Hàm này sử dụng hàm `print` để hiển thị một chuỗi, nhưng nó không sử dụng câu lệnh `return` để trả về một giá trị. Nếu chúng ta gán kết quả cho một biến, nó vẫn hiển thị chuỗi đó.

```
result = repeat('Finland, ', 3)
```

Và nếu chúng ta hiển thị giá trị của biến, chúng ta không nhận được gì.

```
result
```

Thực ra, `result` có một giá trị, nhưng Jupyter không hiển thị nó. Tuy nhiên, chúng ta có thể hiển thị nó như thế này.

```
print(result)
```

Trả về giá trị từ hàm `repeat` là `None`.

Bây giờ, đây là một hàm tương tự như `repeat` nhưng có một trả về giá trị.

```
def repeat_string(word, n):
    return word * n
```

Lưu ý rằng chúng ta có thể sử dụng một biểu thức trong câu lệnh `return`, không chỉ là một biến.

Với phiên bản này, chúng ta có thể gán kết quả cho một biến. Khi hàm chạy, nó không hiển thị gì cả.

```
line = repeat_string('Spam, ', 4)
```

Nhưng sau đó, chúng ta có thể hiển thị giá trị được gán cho biến `line`.

```
line
```

Một hàm như thế này được gọi là **hàm thuần khiết** vì nó không hiển thị gì hoặc có bất kỳ hiệu ứng nào khác — ngoài việc trả về một giá trị.

Trả về giá trị và các câu lệnh điều kiện

Nếu Python không cung cấp hàm `abs`, chúng ta có thể viết nó như thế này.

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

Nếu `x` là số âm, câu lệnh `return` đầu tiên sẽ trả về `-x` và hàm sẽ kết thúc ngay lập tức. Ngược lại, câu lệnh `return` thứ hai sẽ trả về `x` và hàm sẽ kết thúc. Vì vậy, hàm này là đúng.

Tuy nhiên, nếu bạn đặt các câu lệnh `return` trong một điều kiện, bạn phải đảm bảo rằng mọi đường đi có thể xảy ra trong chương trình đều đi đến một câu lệnh `return`. Ví dụ, đây là một phiên bản không chính xác của hàm `absolute_value`.

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

Đây là điều gì sẽ xảy ra nếu chúng ta gọi hàm này với `0` làm đối số.

```
absolute_value_wrong(0)
```

Chúng ta không nhận được gì cả! Đây là vấn đề: khi `x` bằng `0`, không điều kiện nào là đúng, và hàm kết thúc mà không gặp câu lệnh `return`, điều này có nghĩa là giá trị trả về là `None`, vì vậy Jupyter không hiển thị gì.

Như một ví dụ khác, đây là một phiên bản của hàm `absolute_value` với một câu lệnh `return` bổ sung ở cuối.

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

    return 'This is dead code'
```

Nếu `x` là số âm, câu lệnh `return` đầu tiên sẽ chạy và hàm sẽ kết thúc. Ngược lại, câu lệnh `return` thứ hai sẽ chạy và hàm sẽ kết thúc. Dù bằng cách nào, chúng ta cũng không bao giờ đến câu lệnh `return` thứ ba — vì vậy nó không bao giờ chạy.

Mã không bao giờ chạy được gọi là mã chết. Nói chung, mã chết không gây hại gì, nhưng nó thường chỉ ra một sự hiểu lầm, và nó có thể gây nhầm lẫn cho ai đó đang cố gắng hiểu chương trình.

Phát triển theo từng bước

Khi bạn viết các hàm lớn hơn, bạn có thể thấy mình dành nhiều thời gian hơn cho việc gỡ lỗi. Để đối phó với các chương trình ngày càng phức tạp, bạn có thể muốn thử phát triển theo từng bước, đây là cách thêm và kiểm tra chỉ một lượng mã nhỏ tại một thời điểm.

Như ví dụ, giả sử bạn muốn tìm khoảng cách giữa hai điểm được biểu diễn bởi các tọa độ (x_1, y_1) và (x_2, y_2). Theo định lý Pythagore, khoảng cách là:

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

Bước đầu tiên là xem xét hàm `distance` sẽ trông như thế nào trong Python — tức là, các tham số (đầu vào) là gì và trả về giá trị (đầu ra) là gì?

Đối với hàm này, các đầu vào là tọa độ của các điểm. Trả về giá trị là khoảng cách. Ngay lập tức, bạn có thể viết một dàn bài cho hàm:

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

Phiên bản này chưa tính toán khoảng cách — nó luôn trả về `zero`. Nhưng nó là một hàm hoàn chỉnh với giá trị `return`, có nghĩa là bạn có thể kiểm tra nó trước khi làm cho nó phức tạp hơn.

Để kiểm tra hàm mới, chúng ta sẽ gọi nó với các đối số mẫu:

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

Tôi đã chọn những giá trị này để khoảng cách ngang là `3` và khoảng cách dọc là `4`. Như vậy, kết quả là `5`, cạnh huyền của một tam giác vuông lần lượt là `3–4–5`. Khi kiểm tra một hàm, thật hữu ích khi biết câu trả lời đúng.

Tại thời điểm này, chúng ta đã xác nhận rằng hàm chạy và trả về một giá trị, và chúng ta có thể bắt đầu thêm mã vào thân hàm. Một bước tiếp theo tốt là tìm các hiệu số `x2 – x1` và `y2 – y1`. Dưới đây là một phiên bản lưu các giá trị đó vào các biến tạm và hiển thị chúng.

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

Nếu hàm hoạt động, nó sẽ hiển thị `dx` là `3` và `dy` là `4`. Nếu vậy, chúng ta biết rằng hàm đang nhận các đối số đúng và thực hiện phép tính đầu tiên chính xác. Nếu không, chỉ có vài dòng để kiểm tra.

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

Mọi việc vẫn diễn ra tốt đẹp. Tiếp theo, chúng ta tính tổng bình phương của `dx` và `dy`:

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

Một lần nữa, chúng ta có thể chạy hàm và kiểm tra đầu ra và nó nên là `25`.

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

Cuối cùng, chúng ta có thể sử dụng `math.sqrt` để tính khoảng cách:

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

Và tiếp tục kiểm tra nó.

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

Kết quả là chính xác, nhưng phiên bản này của hàm hiển thị kết quả thay vì trả về nó, vì vậy trả về giá trị là `None`.

Chúng ta có thể sửa điều đó bằng cách thay thế hàm `print` bằng một câu lệnh `return`.

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

Phiên bản này của hàm `distance` là một hàm thuần khiết. Nếu chúng ta gọi nó như thế này, chỉ có kết quả được hiển thị.

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

Và nếu chúng ta gán kết quả cho một biến, sẽ không có gì được hiển thị.

```
d = distance(1, 2, 4, 6)
```

Các câu lệnh `print` mà chúng ta đã viết rất hữu ích cho việc gỡ lỗi, nhưng khi hàm đã hoạt động, chúng ta có thể xóa chúng. Mã như vậy được gọi là **giàn giáo** vì nó hữu ích cho việc xây dựng chương trình nhưng không phải là một phần của sản phẩm cuối cùng.

Ví dụ này minh họa cho phát triển theo từng bước. Các khía cạnh chính của quá trình này là:

1. Bắt đầu với một chương trình hoạt động, thực hiện các thay đổi nhỏ và kiểm tra sau mỗi thay đổi.
2. Sử dụng các biến để lưu trữ các giá trị trung gian để bạn có thể hiển thị và kiểm tra chúng.
3. Khi chương trình hoạt động, hãy gỡ bỏ giàn giáo đi.

Vào bất kỳ thời điểm nào, nếu có lỗi, bạn sẽ có ý tưởng rõ ràng về vị trí của nó. Phát triển theo từng bước có thể giúp bạn tiết kiệm rất nhiều thời gian gỡ lỗi.

Các hàm Boolean

Các hàm có thể trả về các giá trị boolean là `True` và `False`, điều này thường rất tiện lợi cho việc đóng gói một bài kiểm tra phức tạp trong một hàm. Ví dụ, hàm `is_divisible` kiểm tra xem `x` có chia hết cho `y` mà không có số dư hay không.

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

Đây là cách chúng ta sử dụng nó.

```
is_divisible(6, 4)
```

```
is_divisible(6, 3)
```

Bên trong hàm, kết quả của toán tử `==` là một giá trị boolean, vì vậy chúng ta có thể viết hàm một cách ngắn gọn hơn bằng cách trả về nó trực tiếp.

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

Các hàm boolean thường được sử dụng trong các câu lệnh điều kiện.

```
if is_divisible(6, 2):
    print('divisible')
```

Có thể bạn sẽ bị cám dỗ để viết một cái gì đó như thế này:

```
if is_divisible(6, 2) == True:
    print('divisible')
```

Nhưng phép so sánh là không cần thiết.

Đệ quy với trả về giá trị

Bây giờ, chúng ta có thể viết các hàm với trả về giá trị, chúng ta có thể viết các hàm đệ quy với trả về giá trị, và với khả năng đó, chúng ta đã vượt qua một ngưỡng quan trọng — tập hợp Python mà chúng ta có bây giờ là **Turing hoàn chỉnh**, có nghĩa là chúng ta có thể thực hiện bất kỳ phép toán nào có thể được mô tả bởi một thuật toán.

Để minh họa đệ quy với trả về giá trị, chúng ta sẽ đánh giá một vài hàm toán học được định nghĩa đệ quy. Một định nghĩa đệ quy tương tự như một định nghĩa vòng tròn, nghĩa là định nghĩa đó tham chiếu đến chính nó. Một định nghĩa thực sự vòng tròn không hữu ích lắm:

> vorpal: Tính từ dùng để miêu tả một cái gì đó là rất sắc bén.

Nếu bạn thấy định nghĩa đó trong từ điển, bạn có thể sẽ cảm thấy khó chịu. Mặt khác, nếu bạn tra cứu định nghĩa của hàm giai thừa, được ký hiệu bằng dấu $!$, bạn có thể nhận được một cái gì đó như sau:

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

Định nghĩa này nói rằng giai thừa của 0 là 1, và giai thừa của bất kỳ giá trị nào khác, n, là n nhân với giai thừa của $n-1$.

Nếu bạn có thể viết một định nghĩa đệ quy cho một thứ gì đó, bạn có thể viết một chương trình Python để đánh giá nó. Theo quy trình phát triển từng bước, chúng ta sẽ bắt đầu với một hàm nhận `n` làm tham số và luôn trả về `0`.

```
def factorial(n):
    return 0
```

Bây giờ, hãy thêm phần đầu tiên của định nghĩa — nếu đối số là `0`, tất cả những gì chúng ta cần làm là trả về `1`:

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

Bây giờ, hãy hoàn thành phần thứ hai — nếu `n` không phải là `0`, chúng ta cần thực hiện một cuộc gọi đệ quy để tìm giai thừa của `n-1` và sau đó nhân kết quả với `n`:

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

Luồng thực thi cho chương trình này tương tự như luồng của hàm `countdown` trong Chương 5. Nếu chúng ta gọi `factorial` với giá trị `3`:

Vì `3` không phải là `0`, chúng ta sẽ đi vào nhánh thứ hai và tính giai thừa của `n-1` ...

> Vì `2` không phải là `0`, chúng ta sẽ đi vào nhánh thứ hai và tính toán giai thừa của `n-1` ... giai thừa của `n-1` ...
>
> > Vì `1` không phải là `0`, chúng ta sẽ đi vào nhánh thứ hai và tính toán giai thừa của `n-1` ... giai thừa của `n-1` ...
> >
> > > Vì `0` bằng `0`, chúng ta sẽ đi vào nhánh đầu tiên và trả về `1` mà không cần... thực hiện thêm bất kỳ cuộc gọi đệ quy nào.
> >
> > Trả về giá trị, `1`, được nhân với `n`, là `1`, và kết quả được trả về.
>
> Trả về giá trị, `1`, được nhân với `n`, là `2`, và kết quả được trả về.

Trả về giá trị `2` được nhân với `n`, là `3`, và kết quả, `6`, trở thành trả về giá trị của cuộc gọi hàm đã khởi đầu toàn bộ quá trình.

Hình dưới đây cho thấy sơ đồ ngăn xếp cho chuỗi các cuộc gọi hàm này.

```
from diagram import Frame, Stack, make_binding

main = Frame([], name='__main__', loc='left')
frames = [main]

ns = 3, 2, 1
recurses = 2, 1, 1
results = 6, 2, 1

for n, recurse, result in zip(ns, recurses, results):
    binding1 = make_binding('n', n)
    binding2 = make_binding('recurse', recurse)
    frame = Frame([binding1, binding2],
                  name='factorial', value=result,
                  loc='left', dx=1.2)
    frames.append(frame)

binding1 = make_binding('n', n)
frame = Frame([binding1], name='factorial', value=1,
              shim=1.2, loc='left', dx=1.4)
frames.append(frame)

stack = Stack(frames, dy=-0.45)


from diagram import diagram, adjust

width, height, x, y = [2.74, 2.26, 0.73, 2.05]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
# adjust(x, y, bbox)
```

Các trả về giá trị được hiển thị đang được truyền ngược lên ngăn xếp. Trong mỗi khung, trả về giá trị là tích của `n` và `recurse`.

Trong khung cuối cùng, biến cục bộ `recurse` không tồn tại vì nhánh tạo ra nó không chạy.

Bước nhảy của niềm tin

Theo dõi luồng thực thi là một cách để đọc chương trình, nhưng nó có thể nhanh chóng trở nên quá tải. Một lựa chọn khác là điều mà tôi gọi là "bước nhảy của niềm tin". Khi bạn đến một cuộc gọi hàm, thay vì theo dõi luồng thực thi, bạn giả định rằng hàm đó hoạt động chính xác và trả về kết quả đúng.

Thực tế, bạn đã thực hành bước nhảy của niềm tin này khi sử dụng các hàm tích hợp sẵn. Khi bạn gọi `abs` hoặc `math.sqrt`, bạn không xem xét phần thân của các hàm đó — bạn chỉ giả định rằng chúng hoạt động.

Điều tương tự cũng đúng khi bạn gọi một trong những hàm của chính mình. Ví dụ, trước đó chúng ta đã viết một hàm gọi là `is_divisible` để xác định xem một số có chia hết cho số khác hay không. Khi chúng ta thuyết phục bản thân rằng hàm này là chính xác, chúng ta có thể sử dụng nó mà không cần xem lại phần thân nữa.

Điều tương tự cũng đúng đối với các chương trình đệ quy. Khi bạn đến cheo dõi luồng thực thiuộc gọi đệ quy, thay vì t, bạn nên giả định rằng cuộc gọi đệ quy hoạt động và sau đó tự hỏi mình, "Giả sử tôi có thể tính giai thừa của $n-1$, tôi có thể tính giai thừa của n không?" Định nghĩa đệ quy của giai thừa ngụ ý rằng bạn có thể, bằng cách nhân với n.

Tất nhiên, thật hơi kỳ lạ khi giả định rằng hàm hoạt động chính xác khi bạn chưa hoàn thành việc viết nó, nhưng đó là lý do tại sao nó được gọi là bước nhảy của niềm tin!

Số fibonacci

Sau hàm `factorial`, ví dụ phổ biến nhất về một hàm đệ quy là `fibonacci`, có định nghĩa như sau:

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

Khi được dịch sang Python, nó trông như thế này:

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

Nếu bạn cố gắng theo dõi luồng thực thi ở đây, ngay cả với các giá trị nhỏ của n, bạn sẽ cảm thấy choáng váng. Nhưng theo bước nhảy của niềm tin, nếu bạn giả định rằng hai cuộc gọi đệ quy hoạt động chính xác, bạn có thể tự tin rằng câu lệnh `return` cuối cùng là đúng.

Nhân tiện, cách tính số Fibonacci này rất kém hiệu quả. Trong Chương 10 tôi sẽ giải thích tại sao và đề xuất một cách để cải thiện nó.

Kiểm tra kiểu dữ liệu

Điều gì xảy ra nếu chúng ta gọi `factorial` và đưa cho nó 1.5 làm đối số?"

```
%%expect RecursionError

factorial(1.5)
```

Nó trông như một cuộc gọi đệ quy vô hạn. Làm sao có thể như vậy? Hàm có một trường hợp cơ bản — khi `n == 0`. Nhưng nếu n không phải là một số nguyên, chúng ta có thể bỏ lỡ trường hợp cơ bản và đệ quy mãi mãi.

Trong ví dụ này, giá trị ban đầu của `n` là `1.5`. Trong cuộc gọi đệ quy đầu tiên, giá trị của `n` là `0.5`. Trong cuộc gọi tiếp theo, nó là `-0.5`. Từ đó, nó sẽ nhỏ hơn (âm hơn), nhưng nó sẽ không bao giờ bằng `0`.

Để tránh đệ quy vô hạn, chúng ta có thể sử dụng hàm tích hợp sẵn `isinstance` để kiểm tra kiểu của đối số. Đây là cách chúng ta kiểm tra xem một giá trị có phải là một số nguyên hay không

```
isinstance(3, int)
```

```
isinstance(1.5, int)
```

Bây giờ, đây là một phiên bản của hàm `factorial` với kiểm tra lỗi.

```
def factorial(n):
    if not isinstance(n, int):
        print('factorial is only defined for integers.')
        return None
    elif n < 0:
        print('factorial is not defined for negative numbers.')
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Đầu tiên, nó kiểm tra xem `n` có phải là một số nguyên hay không. Nếu không, nó hiển thị một thông báo lỗi và trả về `None`.

```
factorial('crunchy frog')
```

Sau đó, nó kiểm tra xem `n` có phải là số âm hay không. Nếu có, nó hiển thị một thông báo lỗi và trả về `None`.

```
factorial(-2)
```

Nếu chúng ta vượt qua cả hai kiểm tra, chúng ta biết rằng `n` là một số nguyên không âm, vì vậy chúng ta có thể tự tin rằng quá trình đệ quy sẽ kết thúc. Kiểm tra các tham số của một hàm để đảm bảo rằng chúng có kiểu và giá trị đúng được gọi là **kiểm tra đầu vào**.

Gỡ lỗi

Chia một chương trình lớn thành các hàm nhỏ hơn tạo ra những điểm kiểm tra tự nhiên để gỡ lỗi. Nếu một hàm không hoạt động, có ba khả năng cần xem xét:

- Có điều gì đó sai với các đối số mà hàm nhận được — tức là, một điều kiện tiên quyết bị vi phạm.
- Có điều gì đó sai với hàm — tức là, một điều kiện hậu kiểm bị vi phạm. Người gọi đang làm sai điều gì đó với trả về giá trị.
- Để loại trừ khả năng đầu tiên, bạn có thể thêm một câu lệnh `print` ở đầu hàm để hiển thị các giá trị của các tham số (và có thể cả kiểu của chúng). Hoặc bạn có thể viết mã kiểm tra rõ ràng các điều kiện tiên quyết.

Nếu các tham số trông ổn, bạn có thể thêm một câu lệnh `print` trước mỗi câu lệnh `return` và hiển thị trả về giá trị. Nếu có thể, hãy gọi hàm với các đối số mà giúp dễ dàng kiểm tra kết quả.

Nếu hàm có vẻ hoạt động, hãy xem xét cuộc gọi hàm để đảm bảo rằng trả về giá trị được sử dụng đúng cách — hoặc ít nhất là đã được sử dụng!

Thêm các câu lệnh `print` ở đầu và cuối của một hàm có thể giúp làm cho luồng thực thi trở nên rõ ràng hơn. Ví dụ, đây là một phiên bản của hàm giai thừa với các câu lệnh `print`:

```
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
```

`space` là một chuỗi các ký tự khoảng trắng dùng để điều khiển độ thụt dòng của đầu ra. Dưới đây là kết quả của `factorial(3)`:

```
factorial(3)
```

Nếu bạn cảm thấy bối rối về luồng thực thi, loại đầu ra này có thể hữu ích. Việc phát triển một hệ thống giàn giáo hiệu quả sẽ mất nhiều thời gian, nhưng một chút giàn giáo có thể tiết kiệm rất nhiều thời gian gỡ lỗi.

Thuật ngữ

- **return value - trả về giá trị:** Kết quả của một hàm. Nếu một cuộc gọi hàm được sử dụng như một biểu thức, giá trị trả về là giá trị của biểu thức đó.
- **pure function - hàm thuần túy:** Một hàm không hiển thị bất cứ điều gì hoặc không có bất kỳ tác động nào khác, ngoài việc trả về một giá trị.
- **dead code - mã chết:** Phần của một chương trình không bao giờ chạy, thường vì nó xuất hiện sau một câu lệnh `return`.
- **incremental development - phát triển từng bước:** Kế hoạch phát triển chương trình nhằm tránh gỡ lỗi bằng cách thêm và kiểm tra chỉ một lượng mã lệnh nhỏ mỗi lần.
- **scaffolding - giàn giáo:** Mã lệnh được sử dụng trong quá trình phát triển chương trình nhưng không phải là một phần của phiên bản cuối cùng.
- **Turing complete - Turing hoàn chỉnh:** Một ngôn ngữ, hoặc tập con của một ngôn ngữ, được gọi là Turing hoàn chỉnh nếu nó có thể thực hiện bất kỳ phép tính nào có thể được mô tả bằng một thuật toán.
- **input validation - kiểm tra đầu vào**: Kiểm tra các tham số của một hàm để đảm bảo rằng chúng có kiểu và giá trị đúng.

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
```

Hỏi trợ lý ảo

Trong chương này, chúng ta đã thấy một hàm không chính xác có thể kết thúc mà không trả về giá trị nào.

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

Và một phiên bản của cùng một hàm có mã chết ở cuối.

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

    return 'This is dead code.'
```

Và chúng ta đã thấy ví dụ sau, mặc dù đúng nhưng không mang tính chất hiệu quả.

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

Hãy hỏi một trợ lý ảo xem có gì sai với từng hàm này và xem nó có thể phát hiện lỗi hoặc cải thiện phong cách không.

Sau đó hỏi: "Viết một hàm nhận tọa độ của hai điểm và tính toán khoảng cách giữa chúng." Hãy xem kết quả có giống phiên bản hàm `distance` mà chúng ta đã viết trong chương này không.

Bài tập 1

Sử dụng phát triển từng bước để viết một hàm có tên là `hypot` trả về độ dài của cạnh huyền của một tam giác vuông, với độ dài của hai cạnh còn lại là đối số.

Lưu ý: Có một hàm trong mô-đun toán học gọi là `hypot` thực hiện cùng một chức năng, nhưng bạn không nên sử dụng nó cho bài tập này!

Ngay cả khi bạn có thể viết hàm chính xác ngay từ lần đầu tiên, hãy bắt đầu với một hàm luôn trả về `0` và thực hành thực hiện các thay đổi nhỏ, kiểm tra từng bước. Khi bạn hoàn thành, hàm chỉ nên trả về một giá trị — nó không nên hiển thị bất kỳ điều gì.

Bài tập 2

Viết một hàm boolean, `is_between(x, y, z)`, trả về `True` nếu $x < y < z$, hoặc nếu $z < y < x$, và trả về `False` trong các trường hợp khác.

Bạn có thể sử dụng những ví dụ này để kiểm tra hàm của mình.

```
is_between(1, 2, 3)  # nên là True

is_between(3, 2, 1)  # nên là True

is_between(1, 3, 2)  # nên là False

is_between(2, 3, 1)  # nên là False
```

Bài tập 3

Hàm Ackermann, $A(m, n)$, được định nghĩa như sau:

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

Viết một hàm có tên là `ackermann` để đánh giá hàm Ackermann. Điều gì xảy ra nếu bạn gọi ackermann(5, 5)?

Bạn có thể sử dụng những ví dụ này để kiểm tra hàm của mình.

```
ackermann(3, 2)  # nên là 29
```

```
ackermann(3, 3)  # nên là 61
```

```
ackermann(3, 4)  # nên là 125
```

Nếu bạn gọi hàm này với các giá trị lớn hơn `4`, bạn sẽ nhận được một lỗi `RecursionError`.

```
%%expect RecursionError
```

```
ackermann(5, 5)
```

Để xem lý do, hãy thêm một câu lệnh `print` ở đầu hàm để hiển thị các giá trị của các tham số, sau đó chạy lại các ví dụ.

Bài tập 4

Một số, a, là lũy thừa của b nếu nó chia hết cho b và $\frac{a}{b}$ là lũy thừa của b. Viết một hàm có tên là `is_power` nhận các tham số `a` và `b` và trả về `True` nếu `a` là lũy thừa của `b`. Lưu ý: bạn sẽ cần suy nghĩ về trường hợp cơ sở.

Bạn có thể sử dụng những ví dụ sau để kiểm tra hàm của mình.

```
is_power(65536, 2)   # nên là True
```

```
is_power(27, 3)  # nên là True
```

```
is_power(24, 2)  # nên là False
```

```
is_power(1, 17)   # nên là True
```

Bài tập 5

Ước chung lớn nhất (GCD) của a và b là số lớn nhất chia hết cho cả hai mà không có dư.

Một cách để tìm GCD của hai số dựa trên nhận xét rằng nếu r là phần dư khi a chia cho b, thì $gcd(a, b) = gcd(b, r)$. Làm trường hợp cơ sở, ta có $gcd(a, 0) = a$.

Viết một hàm có tên là gcd nhận các tham số a và b và trả về ước chung lớn nhất của chúng.

Bạn có thể sử dụng những ví dụ sau để kiểm tra hàm của mình.

```
gcd(12, 8)    # nên là 4

gcd(13, 17)   # nên là 1
```