```
# 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');
download('https://github.com/ramalho/jupyturtle/releases/download/2024-03/jupyturtl

import thinkpython
```

Điều kiện và đệ quy

Chủ đề chính của chương này là câu lệnh `if`, câu lệnh này thực thi các đoạn mã khác nhau tùy thuộc vào trạng thái của chương trình. Và với câu lệnh `if`, chúng ta sẽ có thể khám phá một trong những ý tưởng mạnh mẽ nhất trong lĩnh vực tính toán, đó là `đệ quy`.

Nhưng trước tiên, chúng ta sẽ bắt đầu với ba tính năng mới: toán tử chia lấy dư, biểu thức boolean, và các toán tử logic.

Phép chia nguyên và chia lấy phần dư

Hãy nhớ rằng toán tử chia nguyên, `//`, chia hai số và làm tròn xuống đến một số nguyên. Ví dụ, giả sử thời lượng của một bộ phim là 105 phút. Bạn có thể muốn biết thời lượng đó là bao nhiêu giờ. Phép chia thông thường trả về một số thực:

```
minutes = 105
minutes / 60
```

Nhưng chúng ta thường không viết số giờ với dấu thập phân. Phép chia nguyên trả về số giờ nguyên, làm tròn xuống:

```
minutes = 105
hours = minutes // 60
hours
```

Để lấy phần dư, bạn có thể trừ đi một giờ tính bằng phút:

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

Hoặc bạn có thể sử dụng **toán tử chia lấy phần dư**, `%`, toán tử này chia hai số và trả về phần dư.

```
remainder = minutes % 60
remainder
```

Toán tử chia lấy dư hữu ích hơn những gì bạn có thể nghĩ. Ví dụ, nó có thể kiểm tra xem một số có chia hết cho số khác hay không – nếu `x % y` bằng không, thì `x` chia hết cho `y`.

Ngoài ra, nó có thể trích xuất chữ số hoặc các chữ số cuối cùng từ một số. Ví dụ, `x % 10` trả về chữ số cuối cùng của `x` (trong hệ cơ số 10). Tương tự, `x % 100` trả về hai chữ số cuối cùng.

```
x = 123
x % 10
```

```
x % 100
```

Cuối cùng, toán tử chia lấy dư có thể thực hiện "đồng hồ số học". Ví dụ, nếu một sự kiện bắt đầu lúc 11 giờ sáng và kéo dài ba giờ, chúng ta có thể sử dụng toán tử chia lấy dư để tính xem sự kiện kết thúc vào lúc mấy giờ.

```
start = 11
duration = 3
end = (start + duration) % 12
end
```

Sự kiện này sẽ kết thúc vào lúc 2 chiều.

Biểu thức Boolean

Biểu thức Boolean là một biểu thức có giá trị đúng hoặc sai. Ví dụ, các biểu thức sau sử dụng toán tử so sánh bằng, `==`, so sánh hai giá trị và trả về `True` nếu chúng bằng nhau và `False` nếu không:

```
5 == 5
```

```
5 == 7
```

Một lỗi thường gặp là sử dụng một dấu bằng đơn (`=`) thay vì dấu bằng kép (`==`). Hãy nhớ rằng `=` gán một giá trị cho biến, còn `==` so sánh hai giá trị.

```
x = 5
y = 7
```

```
x == y
```

`True` và `False` là các giá trị đặc biệt thuộc kiểu bool; chúng không phải là chuỗi ký tự:

```
type(True)
```

```
type(False)
```

Toán tử `==` là một trong **các toán tử so sánh**; các toán tử khác là:

```
x != y               # x không bằng y
```

```
x > y                # x lớn hơn y
```

```
x < y               # x nhỏ hơn y
```

```
x >= y                # x lớn hơn hoặc bằng y

x <= y                # x nhỏ hơn hoặc bằng y
```

Toán tử logic

Để kết hợp các giá trị `boolean` thành các biểu thức, chúng ta có thể sử dụng các toán tử `logic`. Các toán tử phổ biến nhất là `and`, `or` và `not`. Ý nghĩa của các toán tử này tương tự như ý nghĩa của chúng trong tiếng Anh. Ví dụ, giá trị của biểu thức sau là `True` khi và chỉ khi `x` lớn hơn `0` và nhỏ hơn `10`.

```
x > 0 and x < 10
```

Biểu thức sau là `True` nếu một trong hai điều kiện hoặc cả hai điều kiện đều đúng, tức là nếu số đó chia hết cho `2` hoặc `3`:

```
x % 2 == 0 or x % 3 == 0
```

Cuối cùng, toán tử `not` phủ định một biểu thức `boolean`, vì vậy biểu thức sau là `True` nếu `x > y` là `False`.

```
not x > y
```

Nói một cách chính xác, các toán hạng của một toán tử `logic` nên là các biểu thức `boolean`, nhưng Python không quá nghiêm ngặt. Bất kỳ số nào khác không bằng `0` đều được hiểu là `True`:

```
42 and True
```

Tính linh hoạt này có thể hữu ích, nhưng có một số điểm tinh vi có thể gây nhầm lẫn. Bạn có thể muốn tránh điều này.

Câu lệnh if

Để viết các chương trình hữu ích, chúng ta gần như luôn cần khả năng kiểm tra các điều kiện và thay đổi hành vi của chương trình cho phù hợp. Các câu lệnh điều kiện mang lại cho chúng ta khả năng này. Dạng đơn giản nhất là câu lệnh `if`:

```
if x > 0:
    print('x là số dương')
```

Câu lệnh `if` là một từ khóa trong Python. Các câu lệnh `if` có cấu trúc giống như định nghĩa hàm: một tiêu đề theo sau bởi một câu lệnh hoặc một chuỗi các câu lệnh được gọi là khối lệnh và được thụt dòng.

Biểu thức `boolean` sau `if` được gọi là điều kiện. Nếu điều kiện đúng, các câu lệnh trong khối thụt dòng sẽ được thực thi. Nếu không, chúng sẽ không được thực thi.

Không có giới hạn nào về số lượng câu lệnh có thể xuất hiện trong khối lệnh, nhưng ít nhất phải có một câu lệnh. Đôi khi, việc có một khối lệnh không làm gì cũng rất hữu ích — thường là để giữ chỗ cho mã mà bạn chưa viết. Trong trường hợp đó, bạn có thể sử dụng câu lệnh `pass`, câu lệnh này sẽ không làm gì cả.

```
if x < 0:
    pass          # TODO: Cần xử lý các giá trị âm!
```

Từ `TODO` trong một chú thích là một nhắc nhở thông dụng cho thấy rằng có điều gì đó bạn cần thực hiện sau này.

Câu lệnh else

Một câu lệnh `if` có thể có một phần thứ hai, được gọi là câu lệnh `else`. Cú pháp trông như sau:

```
if x % 2 == 0:
    print('x là số chẵn')
else:
    print('x là số lẻ')
```

Nếu điều kiện đúng, câu lệnh thụt dòng đầu tiên sẽ được thực thi; ngược lại, câu lệnh thụt dòng thứ hai sẽ được thực thi.

Trong ví dụ này, nếu `x` là số chẵn, phần dư khi `x` chia cho `2` là `0`, do đó điều kiện đúng và chương trình sẽ hiển thị "x is even". Nếu `x` là số lẻ, phần dư là `1`, do đó điều kiện sai và chương trình sẽ hiển thị "x is odd".

Vì điều kiện phải đúng hoặc sai, nên chính xác một trong hai nhánh sẽ được thực thi. Các nhánh này được gọi là **các nhánh**.

Các điều kiện nối tiếp

Đôi khi có nhiều hơn hai khả năng và chúng ta cần nhiều hơn hai nhánh. Một cách để diễn đạt một phép tính như vậy là sử dụng **điều kiện nối tiếp**, bao gồm câu lệnh `elif`.

```
if x < y:
    print('x nhỏ hơn y')
elif x > y:
    print('x lớn hơn y')
else:
    print('x và y bằng nhau')
```

Câu lệnh `elif` là viết tắt của "else if". Không có giới hạn nào về số lượng câu lệnh `elif`. Nếu có câu lệnh `else`, nó phải ở cuối, nhưng không nhất thiết phải có câu lệnh này.

Mỗi điều kiện được kiểm tra theo thứ tự. Nếu điều kiện đầu tiên sai, điều kiện tiếp theo sẽ được kiểm tra, và cứ tiếp tục như vậy. Nếu một trong các điều kiện đúng, nhánh tương ứng sẽ được thực thi và câu lệnh `if` sẽ kết thúc. Ngay cả khi có nhiều hơn một điều kiện đúng, chỉ có nhánh đúng đầu tiên được thực thi.

Các điều kiện lồng nhau

Một điều kiện có thể lồng vào trong một điều kiện khác. Chúng ta có thể viết ví dụ trong phần trước như sau:

```
if x == y:
    print('x và y bằng nhau')
else:
    if x < y:
        print('x nhỏ hơn y')
    else:
        print('x lớn hơn y')
```

Câu lệnh `if` bên ngoài có hai nhánh. Nhánh đầu tiên chứa một câu lệnh đơn giản. Nhánh thứ hai chứa một câu lệnh `if` khác, cũng có hai nhánh riêng. Hai nhánh đó đều là các câu lệnh đơn giản, mặc dù chúng cũng có thể là các câu lệnh điều kiện.

Mặc dù việc thụt dòng của các câu lệnh làm cho cấu trúc trở nên rõ ràng, nhưng các điều kiện lồng nhau có thể khó đọc. Tôi gợi ý bạn nên tránh chúng khi có thể.

Các toán tử logic thường cung cấp một cách để đơn giản hóa các câu lệnh điều kiện lồng nhau. Dưới đây là một ví dụ với một câu lệnh điều kiện lồng nhau.

```
if 0 < x:
    if x < 10:
        print('x là một số dương có một chữ số.')
```

Câu lệnh `print` chỉ được thực thi nếu chúng ta vượt qua cả hai điều kiện, vì vậy chúng ta có thể đạt được hiệu ứng tương tự với toán tử `and` .

```
if 0 < x and x < 10:
    print('x là một số dương có một chữ số.')
```

Đối với loại điều kiện này, Python cung cấp một tùy chọn ngắn gọn hơn:

```
if 0 < x < 10:
    print('x là một số dương có một chữ số.')
```

Đệ quy

Việc một hàm gọi chính nó là hợp pháp. Có thể không rõ ràng tại sao điều này lại là điều tốt, nhưng thực tế nó là một trong những điều kỳ diệu nhất mà một chương trình có thể làm.

Dưới đây là một ví dụ.

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

Nếu `n` là `0` hoặc âm, hàm `countdown` sẽ xuất ra từ `'Blastoff!'`. Ngược lại, nó sẽ xuất ra giá trị `n` và sau đó gọi chính nó, truyền `n-1` làm đối số.

Dưới đây là điều gì xảy ra khi chúng ta gọi hàm này với đối số là `3`.

```
countdown(3)
```

Việc thực thi hàm `countdown` bắt đầu với `n=3`, và vì `n` lớn hơn `0`, nó sẽ hiển thị `3` và sau đó gọi chính nó...

> Việc thực thi hàm `countdown` bắt đầu với `n=2`, và vì `n` là lớn hơn `0`, nó sẽ hiển thị `2`, và sau đó gọi chính nó...
>
> > Việc thực thi hàm `countdown` bắt đầu với `n=1`, và vì `n` là lớn hơn `0`, nó sẽ hiển thị `1`, và sau đó gọi chính nó...
> >
> > > Việc thực thi hàm `countdown` bắt đầu với `n=0`, và vì `n` là không lớn hơn `0`, nó sẽ hiển thị `'Blastoff!'` và trả về.
> >
> > Hàm `countdown` với `n=1` sẽ trả về.
>
> Hàm `countdown` với `n=2` sẽ trả về.

Hàm `countdown` với `n=3` sẽ trả về.

Một hàm gọi chính nó được gọi là **đệ quy**. Như một ví dụ khác, chúng ta có thể viết một hàm in một chuỗi `n` lần - `print_n_times`.

```
def print_n_times(string, n):
    if n > 0:
        print(string)
        print_n_times(string, n-1)
```

Nếu `n` là số dương, hàm `print_n_times` sẽ hiển thị giá trị của chuỗi và sau đó gọi chính nó, truyền `string` và `n-1` làm đối số.

Nếu `n` là `0` hoặc âm, điều kiện sẽ sai và hàm `print_n_times` sẽ không làm gì cả.

Dưới đây là cách hoạt động của nó.

```
print_n_times('Spam ', 4)
```

Đối với những ví dụ đơn giản như thế này, có lẽ sẽ dễ hơn nếu sử dụng vòng lặp `for`. Nhưng chúng ta sẽ thấy những ví dụ sau đây khó viết bằng vòng lặp `for` và dễ viết bằng `đệ quy`, vì vậy tốt nhất là nên bắt đầu sớm.

Biểu đồ ngăn xếp cho các hàm đệ quy

Dưới đây là một biểu đồ ngăn xếp cho thấy các khung được tạo ra khi chúng ta gọi hàm `countdown` với `n = 3`.

```
from diagram import make_frame, Stack

frames = []
for n in [3,2,1,0]:
    d = dict(n=n)
    frame = make_frame(d, name='countdown', dy=-0.3, loc='left')
    frames.append(frame)

stack = Stack(frames, dy=-0.5)
```

```
from diagram import diagram, adjust


width, height, x, y = [1.74, 2.04, 1.05, 1.77]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
# adjust(x, y, bbox)
```

Bốn khung `countdown` có các giá trị khác nhau cho tham số `n`. Đáy của ngăn xếp, nơi `n=0`, được gọi là **trường hợp cơ bản**.

Nó không thực hiện cuộc gọi **đệ quy**, vì vậy không còn khung nào nữa.

```
from diagram import make_frame, Stack
from diagram import diagram, adjust

frames = []
for n in [2,1,0]:
    d = dict(string='Hello', n=n)
    frame = make_frame(d, name='print_n_times', dx=1.3, loc='left')
    frames.append(frame)

stack = Stack(frames, dy=-0.5)

width, height, x, y = [3.53, 1.54, 1.54, 1.27]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
# adjust(x, y, bbox)
```

Đệ quy vô hạn

Nếu một phép đệ quy không bao giờ đạt đến trường hợp cơ bản, nó sẽ tiếp tục thực hiện các cuộc gọi đệ quy mãi mãi, và chương trình sẽ không bao giờ kết thúc. Đây được gọi là **đệ quy vô hạn**, và nói chung, đây không phải là một ý tưởng hay. Dưới đây là một hàm tối giản với đệ quy vô hạn.

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

Mỗi lần hàm `recurse` được gọi, nó gọi chính nó, điều này tạo ra một khung khác. Trong Python, có một giới hạn về số lượng khung có thể có trên ngăn xếp cùng một lúc. Nếu một chương trình vượt quá giới hạn này, nó sẽ gây ra lỗi thời gian chạy.

```
%xmode Context
```

```
%%expect RecursionError

recurse()
```

Thông báo lỗi cho biết rằng có gần 3000 khung trên ngăn xếp khi lỗi xảy ra.

Nếu bạn gặp phải đệ quy vô hạn một cách tình cờ, hãy xem xét lại hàm của bạn để xác nhận rằng có một trường hợp cơ bản không thực hiện cuộc gọi đệ quy. Và nếu có một trường hợp cơ bản, hãy kiểm tra xem bạn có đảm bảo sẽ đạt đến nó không.

Nhập từ bàn phím

Các chương trình mà chúng ta đã viết cho đến nay không chấp nhận bất kỳ đầu vào nào từ người dùng. Chúng chỉ thực hiện cùng một thao tác mỗi lần.

Python cung cấp một hàm tích hợp có tên là `input` để tạm dừng chương trình và chờ người dùng nhập vào một cái gì đó. Khi người dùng nhấn phím `Return` hoặc `Enter`, chương trình sẽ tiếp tục và hàm `input` trả về những gì người dùng đã nhập dưới dạng chuỗi.

```
# Đáp án được viết ở đây

text = input()
```

Trước khi nhận đầu vào từ người dùng, bạn có thể muốn hiển thị một thông báo nhắc nhở người dùng nên nhập gì. Hàm `input` có thể nhận một thông báo nhắc như là một tham số:

```
# Đáp án được viết ở đây

name = input('Tên...của bạn là gì\n')
name
```

Chuỗi `\n` ở cuối thông báo nhắc đại diện cho một dòng mới, đây là một ký tự đặc biệt gây ra một dấu ngắt dòng—như vậy, đầu vào của người dùng sẽ xuất hiện bên dưới thông báo nhắc.

Nếu bạn mong đợi người dùng nhập một số nguyên, bạn có thể sử dụng hàm `int` để chuyển đổi giá trị trả về thành kiểu `int`.

```
# Đáp án được viết ở đây
```

```
prompt = 'Vận tốc gió...của một con chim nhạn không mang vật gì là bao nhiêu?\n'
speed = input(prompt)
speed
```

Nhưng nếu họ nhập một cái gì đó không phải là số nguyên, bạn sẽ gặp phải lỗi thời gian chạy.

```
%xmode Minimal
```

```
%%expect ValueError

int(speed)
```

Chúng ta sẽ xem cách xử lý loại lỗi này sau.

Gỡ lỗi

Khi xảy ra lỗi cú pháp hoặc lỗi thời gian chạy, thông báo lỗi chứa rất nhiều thông tin và có thể gây áp lực cho chúng ta. Các phần hữu ích nhất thường là:

- Loại lỗi là gì, và
- Nơi xảy ra lỗi.

Lỗi cú pháp thường dễ tìm, nhưng có một vài điểm cần lưu ý. Các lỗi liên quan đến khoảng trắng và `tab` có thể gây khó khăn vì chúng không nhìn thấy được và chúng ta thường quen với việc bỏ qua chúng.

```
%%expect IndentationError
x = 5
 y = 6
```

Trong ví dụ này, vấn đề là dòng thứ hai bị thụt dòng bởi một khoảng trắng. Nhưng thông báo lỗi lại chỉ vào `y`, điều này có thể gây hiểu nhầm. Các thông báo lỗi chỉ ra nơi vấn đề được phát hiện, nhưng lỗi thực sự có thể nằm ở phần trước đó trong mã.

Điều này cũng đúng với các lỗi thời gian chạy. Ví dụ, giả sử bạn đang cố gắng chuyển đổi một tỷ lệ thành `decibel`, như thế này:

```
%xmode Context


%%expect ValueError
import math
numerator = 9
denominator = 10
ratio = numerator // denominator
decibels = 10 * math.log10(ratio)
```

Thông báo lỗi chỉ ra dòng `5`, nhưng không có gì sai ở dòng đó. Vấn đề nằm ở dòng `4`, nơi sử dụng phép chia nguyên thay vì phép chia dấu phẩy động—kết quả là giá trị của `ratio` là `0`. Khi chúng ta gọi `math.log10`, chúng ta nhận được một `ValueError` với thông báo lỗi "math domain error", vì `0` không nằm trong "miền" các đối số hợp lệ cho `math.log10`, bởi vì `logarithm` của `0` là không xác định.

Nói chung, bạn nên dành thời gian để đọc kỹ các thông báo lỗi, nhưng đừng giả định rằng mọi thứ chúng nói đều chính xác.

Thuật ngữ

- **recursion - đệ quy:** Quá trình gọi hàm đang được thực thi.
- **modulus operator - toán tử chia lấy phần dư:** Một toán tử, `%`, hoạt động trên các số nguyên và trả về số dư khi một số được chia cho số khác.
- **boolean expression - biểu thức boolean:** Một biểu thức có giá trị là `True` hoặc `False`.
- **relational operator - toán tử quan hệ:** Một trong các toán tử so sánh các toán hạng của nó: `==`, `!=`, `>`, `<`, `>=`, và `<=`.
- **logical operator - toán tử logic:** Một trong các toán tử kết hợp các biểu thức boolean, bao gồm and, or, và not.
- **conditional statement - câu lệnh điều kiện:** Một câu lệnh điều khiển luồng thực thi tùy thuộc vào một số điều kiện.
- **condition - điều kiện:** Biểu thức boolean trong câu lệnh điều kiện xác định nhánh nào sẽ được thực hiện.
- **block - khối:** Một hoặc nhiều câu lệnh được thụt dòng để chỉ ra rằng chúng là một phần của một câu lệnh khác.
- **branch - nhánh:** Một trong các chuỗi câu lệnh thay thế trong một câu lệnh điều kiện.
- **chained conditional - câu lệnh điều kiện nối tiếp:** Một câu lệnh điều kiện với một chuỗi các nhánh thay thế.
- **nested conditional - câu lệnh điều kiện lồng nhau:** Một câu lệnh điều kiện xuất hiện trong một trong các nhánh của câu lệnh điều kiện khác.
- **recursive - đệ quy:** Một hàm gọi chính nó thì được gọi là đệ quy.
- **base case - trường hợp cơ bản:** Một nhánh điều kiện trong một hàm đệ quy không thực hiện cuộc gọi đệ quy.
- **infinite recursion - đệ quy vô hạn:** Một phép đệ quy không có trường hợp cơ bản hoặc không bao giờ đạt đến nó. Cuối cùng, đệ quy vô hạn gây ra lỗi thời gian chạy.
- **newline - dòng mới:** Một ký tự tạo ra một dấu ngắt dòng giữa hai phần của một chuỗi.

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

- "Những ứng dụng nào của toán tử chia lấy dư?"
- Python cung cấp các toán tử để tính toán các phép toán logic `and`, `or`, và `not`, nhưng nó không có một toán tử nào tính toán phép toán "hoặc loại trừ", thường được viết là `xor`. Hãy hỏi một trợ lý ảo, "Phép toán logic `xor` là gì và tôi làm thế nào để tính toán nó trong Python?"

Trong chương này, chúng ta đã thấy hai cách để viết một câu lệnh `if` với ba nhánh, sử dụng câu lệnh điều kiện nối tiếp hoặc câu lệnh điều kiện lồng nhau. Bạn có thể sử dụng một trợ lý ảo để chuyển đổi từ cách này sang cách khác. Ví dụ, hỏi một trợ lý ảo, "Chuyển đổi câu lệnh này thành một câu lệnh điều kiện nối tiếp."

```
x = 5
y = 7


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')
```

Hãy hỏi một trợ lý ảo, "Viết lại câu lệnh này với một câu lệnh điều kiện duy nhất."

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

Hãy xem liệu một trợ lý ảo có thể đơn giản hóa sự phức tạp không cần thiết này không.

```
if not x <= 0 and not x >= 10:
    print('x is a positive single-digit number.')
```

Dưới đây là một nỗ lực tạo hàm đệ quy để đếm ngược từng bước 2.

```
def countdown_by_two(n):
    if n == 0:
        print('Blastoff!')
    else:
        print(n)
        countdown_by_two(n-2)
```

Có vẻ như nó hoạt động

```
countdown_by_two(6)
```

Hãy hỏi một trợ lý ảo xem có vấn đề gì và cách khắc phục nó. Dán giải pháp mà nó cung cấp ở đây và thử nghiệm nó.

Bài tập 1

Mô-đun `time` cung cấp một hàm, cũng được gọi là `time`, trả về số giây kể từ "thời đại Unix" (Unix epoch), tức là ngày 1 tháng 1 năm 1970, 00:00:00 UTC (Giờ phối hợp quốc tế).

```
from time import time

now = time()
now
```

Sử dụng phép chia nguyên và toán tử chia lấy dư để tính số ngày kể từ ngày 1 tháng 1 năm 1970 và thời gian hiện tại trong giờ, phút và giây.

Bạn có thể đọc thêm về hàm `time` tại đây https://docs.python.org/3/library/time.html.

Bài tập 2

Nếu bạn được cho ba que, bạn có thể hoặc không thể sắp xếp chúng thành một hình tam giác. Ví dụ, nếu một trong các que dài 12 cm và hai que còn lại dài 1 cm, bạn sẽ không thể khiến các que ngắn gặp nhau ở giữa. Đối với bất kỳ ba độ dài nào, có một bài kiểm tra để xem liệu có thể tạo thành một hình tam giác hay không:

> Nếu bất kỳ độ dài nào trong ba độ dài lớn hơn tổng của hai độ dài còn lại, thì bạn không thể tạo thành một hình tam giác. Ngược lại, bạn có thể. (Nếu tổng của hai độ dài bằng độ dài còn lại, chúng tạo thành một "tam giác được gọi là degenerate".)

Viết một hàm có tên `is_triangle` nhận ba số nguyên làm đối số, và in ra "Yes" hoặc "No", tùy thuộc vào việc bạn có thể hoặc không thể tạo thành một hình tam giác từ các que có độ dài cho trước. Gợi ý: Sử dụng cấu trúc điều kiện nối tiếp.

Kiểm tra hàm của bạn với các trường hợp sau.

```
is_triangle(4, 5, 6)   # nên là Yes

is_triangle(1, 2, 3)   # nên là Yes

is_triangle(6, 2, 3)   # nên là No

is_triangle(1, 1, 12)   # nên là No
```

Bài tập 3

Đầu ra của chương trình sau đây là gì? Vẽ một sơ đồ ngăn xếp thể hiện trạng thái của chương trình khi nó in ra kết quả.

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

recurse(3, 0)
```

Bài tập 4

Các bài tập sau đây sử dụng mô-đun `jupyturtle`, được mô tả trong Chương 4.

Đọc hàm sau đây và xem liệu bạn có thể hiểu nó làm gì không. Sau đó, chạy nó và xem bạn có đúng không. Điều chỉnh các giá trị của `length`, `angle` và `factor` và xem chúng ảnh hưởng đến kết quả như thế nào. Nếu bạn không chắc chắn mình hiểu cách nó hoạt động, hãy thử hỏi một trợ lý ảo.

```
from jupyturtle import forward, left, right, back

def draw(length):
    angle = 50
    factor = 0.6

    if length > 5:
        forward(length)
        left(angle)
        draw(factor * length)
        right(2 * angle)
        draw(factor * length)
        left(angle)
        back(length)
```

Bài tập 5

Hỏi một trợ lý ảo: 'Đường cong Koch là gì?'

Để vẽ một đường cong Koch với độ dài `x`, tất cả những gì bạn cần làm là:

1. Vẽ một đường cong Koch với độ dài `x/3`.
2. Quay sang trái 60 độ.
3. Vẽ một đường cong Koch với độ dài `x/3`.
4. Quay sang phải 120 độ.
5. Vẽ một đường cong Koch với độ dài `x/3`.
6. Quay sang trái 60 độ.
7. Vẽ một đường cong Koch với độ dài `x/3`.

Lưu ý là nếu `x` nhỏ hơn `5`, bạn có thể chỉ cần vẽ một đường thẳng với độ dài `x`.

Viết một hàm có tên koch nhận `x` làm đối số và vẽ một đường cong Koch với độ dài cho trước.

Kết quả nên trông như thế này:

```
make_turtle(delay=0)
koch(120)
```

Khi bạn đã làm cho hàm koch hoạt động, bạn có thể sử dụng vòng lặp này để vẽ ba đường cong Koch tạo thành hình dạng của một bông tuyết.

```
make_turtle(delay=0, height=300)
for i in range(3):
    koch(120)
    right(120)
```

Bài tập 6

Các trợ lý ảo biết về các hàm trong mô-đun `jupyturtle`, nhưng có nhiều phiên bản của những hàm này với các tên khác nhau, vì vậy một trợ lý ảo có thể không biết bạn đang nói đến hàm nào.

Để giải quyết vấn đề này, bạn có thể cung cấp thông tin bổ sung trước khi đặt câu hỏi. Ví dụ, bạn có thể bắt đầu một câu hỏi với "Đây là một chương trình sử dụng mô-đun `jupyturtle`, và sau đó dán vào một trong các ví dụ từ chương này. Sau đó, trợ lý ảo sẽ có khả năng tạo mã sử dụng mô-đun này."

Ví dụ, hãy hỏi một trợ lý ảo về một chương trình vẽ tam giác `Sierpiński`. Mã lệnh bạn nhận được nên là một điểm khởi đầu tốt, nhưng bạn có thể cần phải sửa lỗi. Nếu lần thử đầu tiên không hoạt động, bạn có thể cho trợ lý ảo biết điều đã xảy ra và hỏi để được giúp đỡ — hoặc bạn có thể tự sửa lỗi.

Đây là cách mà kết quả có thể trông như thế nào, mặc dù phiên bản bạn nhận được có thể khác.

```
make_turtle(delay=0, height=200)

draw_sierpinski(100, 3)
```