```
# 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
```

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

```
import thinkpython

%load_ext autoreload
%autoreload 2
```

Hàm và giao diện

Chương này giới thiệu một mô-đun có tên là `jupyturtle`, cho phép bạn tạo ra những bức tranh đơn giản bằng cách cung cấp các hướng dẫn cho một chú rùa ảo. Chúng ta sẽ sử dụng mô-đun này để viết các hàm vẽ hình vuông, đa giác và hình tròn — và để minh họa thiết kế **giao diện**, một cách thiết kế các hàm hoạt động cùng nhau.

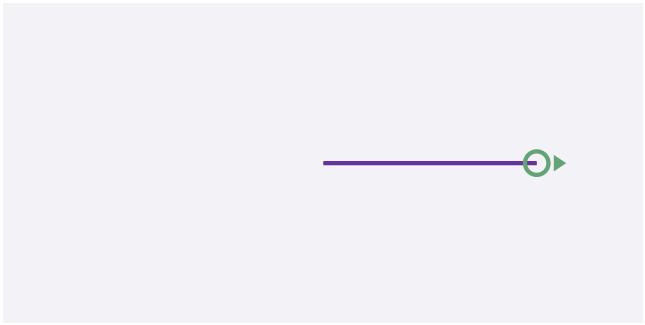
Mô-đun jupyturtle trong Python

Để sử dụng mô-đun `jupyturtle`, chúng ta có thể nhập nó như sau.

```
import jupyturtle
```

Bây giờ chúng ta có thể sử dụng các hàm được định nghĩa trong mô-đun, như `make_turtle` và `forward`.

```
jupyturtle.make_turtle()
jupyturtle.forward(100)
```



`make_turtle` tạo ra một bề mặt vẽ, đó là một không gian trên màn hình nơi chúng ta có thể vẽ, và một chú rùa, được đại diện bởi một hình tròn và một cái đầu hình tam giác. Hình tròn cho thấy vị trí của chú rùa và hình tam giác chỉ ra hướng mà nó đang đối diện.

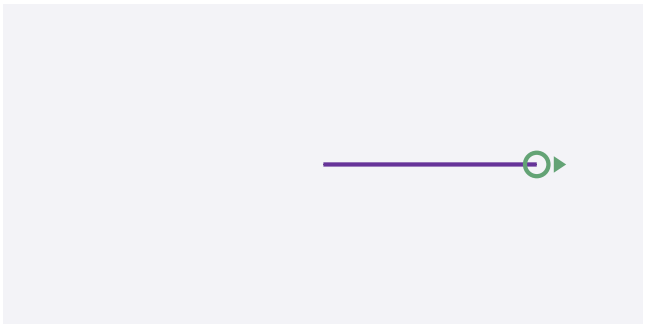
`forward` di chuyển chú rùa một khoảng cách nhất định theo hướng mà nó đang đối diện, vẽ một đoạn thẳng trên đường đi. Khoảng cách được tính bằng đơn vị tùy ý — kích thước thực tế phụ thuộc vào màn hình của máy tính của bạn.

Chúng ta sẽ sử dụng các hàm được định nghĩa trong mô-đun `jupyturtle` nhiều lần, vì vậy sẽ rất tiện lợi nếu chúng ta không phải viết tên của mô-đun mỗi lần. Điều đó có thể thực hiện được nếu chúng ta nhập mô-đun như sau.

```
from jupyturtle import make_turtle, forward
```

Phiên bản này của câu lệnh `import` sẽ nhập `make_turtle` và `forward` từ mô-đun `jupyturtle` để chúng ta có thể gọi chúng như thế này.

```
make_turtle()
forward(100)
```

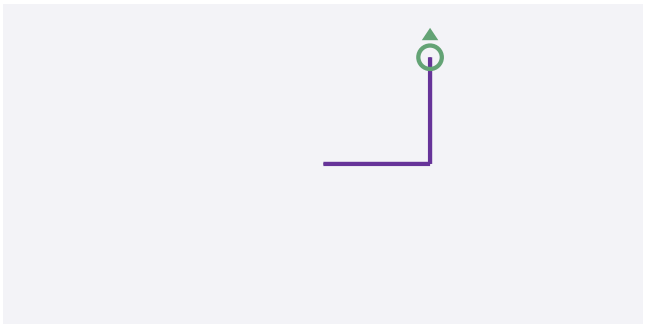


Mô-đun `jupyturtle` cung cấp hai hàm khác mà chúng ta sẽ sử dụng, gọi là `left` và `right`. Chúng ta sẽ nhập chúng như thế này.

```
from jupyturtle import left, right
```

`left` khiến chú rùa quay sang trái. Nó nhận một tham số, đó là góc quay tính bằng độ. Ví dụ, chúng ta có thể làm cho chú rùa quay sang trái `90` độ như sau.

```
make_turtle()
forward(50)
left(90)
forward(50)
```

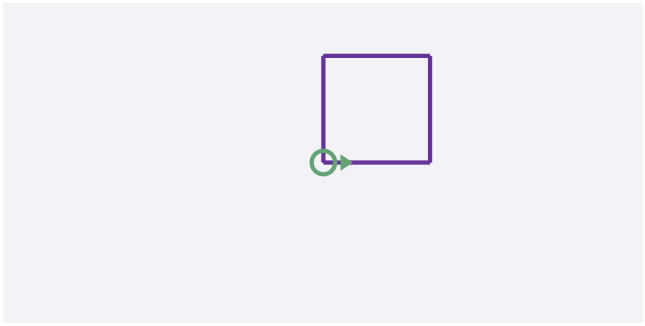


Chương trình này di chuyển chú rùa về phía đông và sau đó là phía bắc, để lại hai đoạn thẳng phía sau. Trước khi tiếp tục, hãy xem bạn có thể sửa đổi chương trình trước đó để tạo thành một hình vuông không.

Tạo một hình vuông

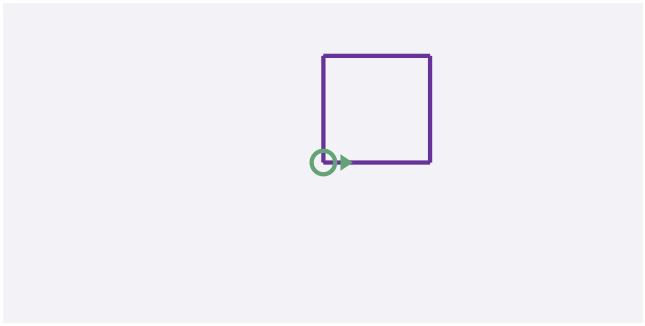
Dưới đây là một cách để tạo ra một hình vuông.

```
make_turtle()

forward(50)
left(90)

forward(50)
left(90)

forward(50)
left(90)

forward(50)
left(90)
```



Bởi vì chương trình này lặp lại cùng một cặp dòng lệnh bốn lần, chúng ta có thể làm điều tương tự một cách ngắn gọn hơn với một vòng lặp `for`.

```
make_turtle()
for i in range(4):
    forward(50)
    left(90)
```



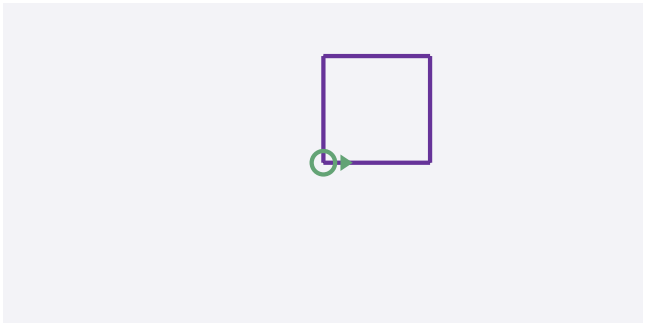
Đóng gói và tổng quát hóa

Hãy lấy đoạn mã vẽ hình vuông từ phần trước và đưa nó vào một hàm có tên là `square`.

```
def square():
    for i in range(4):
        forward(50)
        left(90)
```

Bây giờ, chúng ta có thể gọi hàm như thế này.

```
make_turtle()
square()
```



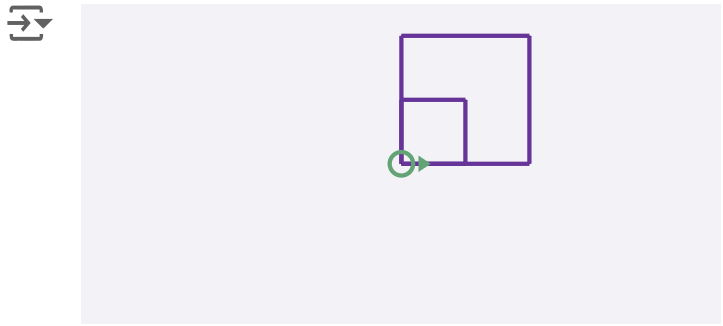
Việc đóng gói một đoạn mã vào trong một hàm được gọi là **đóng gói**. Một trong những lợi ích của đóng gói là nó gán một tên cho đoạn mã, điều này đóng vai trò như một loại tài liệu hướng dẫn. Một lợi thế khác là nếu bạn sử dụng lại mã, việc gọi một hàm hai lần sẽ ngắn gọn hơn là sao chép và dán phần thân mã!

Trong phiên bản hiện tại, kích thước của hình vuông luôn là `50`. Nếu chúng ta muốn vẽ các hình vuông với kích thước khác nhau, chúng ta có thể lấy độ dài các cạnh làm tham số.

```
def square(length):
    for i in range(4):
        forward(length)
        left(90)
```

Bây giờ, chúng ta có thể vẽ các hình vuông với những kích thước khác nhau.

```
make_turtle()
square(30)
square(60)
```



Việc thêm một tham số vào một hàm được gọi là **tổng quát hóa**, vì nó làm cho hàm trở nên tổng quát hơn. Với phiên bản trước, hình vuông luôn có cùng kích thước; với phiên bản này, nó có thể có bất kỳ kích thước nào.

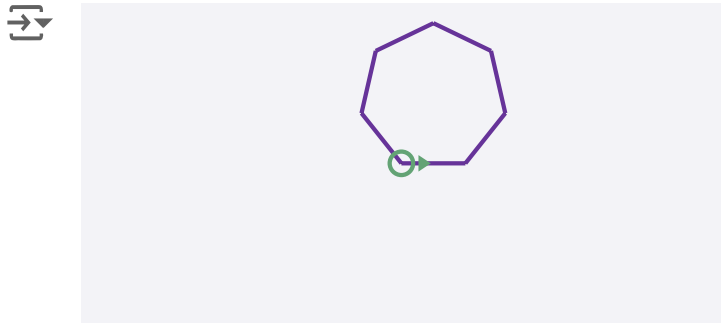
Nếu chúng ta thêm một tham số khác, chúng ta có thể làm cho hàm còn tổng quát hơn nữa. Hàm sau đây vẽ các đa giác đều với số cạnh đã cho.

```
def polygon(n, length):
    angle = 360 / n
    for i in range(n):
        forward(length)
        left(angle)
```

Trong một đa giác đều với n cạnh, góc giữa các cạnh liền kề là $\frac{360}{n}$ độ.

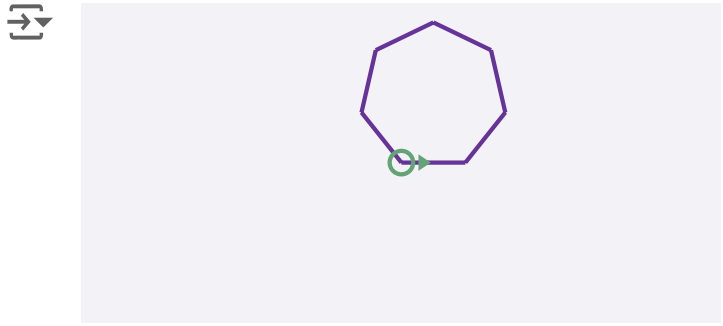
Ví dụ sau đây vẽ một đa giác `7` cạnh với độ dài cạnh là `30`.

```
make_turtle()
polygon(7, 30)
```



Khi một hàm có nhiều hơn một vài tham số số, rất dễ để quên chúng là gì, hoặc thứ tự của chúng nên như thế nào. Việc bao gồm tên của các tham số trong danh sách đối số có thể là một ý tưởng tốt.

```
make_turtle()
polygon(n=7, length=30)
```



Những tham số này đôi khi được gọi là **đối số có tên** vì chúng bao gồm tên của các tham số. Nhưng trong Python, chúng thường được gọi là **đối số từ khóa** (không nên nhầm lẫn với các từ khóa trong Python như `for` và `def`).

Việc sử dụng toán tử gán `=` này nhắc nhở chúng ta về cách thức hoạt động của các đối số và tham số — khi bạn gọi một hàm, các đối số sẽ được gán cho các tham số.

Xấp xỉ một hình tròn

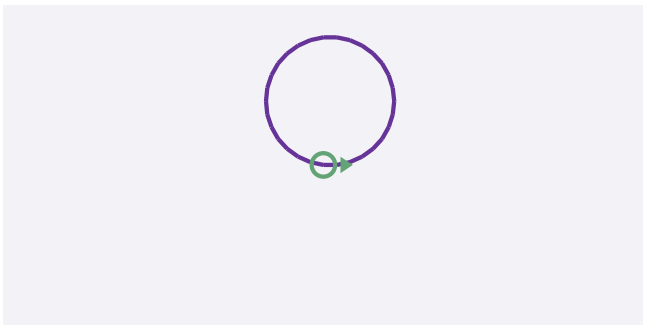
Bây giờ, giả sử chúng ta muốn vẽ một hình tròn. Chúng ta có thể làm điều đó, một cách xấp xỉ, bằng cách vẽ một đa giác với số cạnh lớn, sao cho mỗi cạnh đủ nhỏ để khó nhận thấy. Dưới đây là một hàm sử dụng hàm `polygon` để vẽ một đa giác `30` cạnh, và nó là xấp xỉ một hình tròn.

```
import math

def circle(radius):
    circumference = 2 * math.pi * radius
    n = 30
    length = circumference / n
    polygon(n, length)
```

Hàm `circle` nhận bán kính của hình tròn làm tham số. Nó tính toán chu vi, đó là chu vi của một hình tròn với bán kính đã cho. Biến `n` là số cạnh, vì vậy $\frac{circumference}{n}$ là độ dài của mỗi cạnh.

Hàm này có thể mất nhiều thời gian để chạy. Chúng ta có thể tăng tốc nó bằng cách gọi `make_turtle` với một đối số từ khóa có tên là `delay`, để thiết lập thời gian (tính bằng giây) mà chú rùa chờ sau mỗi bước. Giá trị mặc định là `0,2` giây — nếu chúng ta đặt giá trị này thành `0,02`, nó sẽ chạy nhanh hơn khoảng `10` lần.

```
make_turtle(delay=0.02)
circle(30)
```



Một hạn chế của giải pháp này là `n` là một hằng số, điều này có nghĩa là đối với những hình tròn rất lớn, các cạnh sẽ quá dài, và đối với các hình tròn nhỏ, chúng ta lãng phí thời gian để vẽ các cạnh rất ngắn. Một lựa chọn là tổng quát hóa hàm bằng cách lấy `n` làm tham số. Nhưng hãy giữ cho nó đơn giản trong thời điểm này.

Tổ chức lại mã

Bây giờ, hãy viết một phiên bản tổng quát hơn của hàm `circle`, gọi là `arc`, mà nhận một tham số thứ hai là `angle`, và vẽ một cung tròn trải dài theo góc đã cho. Ví dụ, nếu `angle` là `360` độ, nó sẽ vẽ một hình tròn hoàn chỉnh. Nếu `angle` là `180` độ, nó sẽ vẽ một nửa hình tròn.

Để viết hàm `circle`, chúng ta có thể tái sử dụng hàm `polygon`, vì một đa giác nhiều cạnh là một xấp xỉ tốt của một hình tròn. Nhưng chúng ta không thể sử dụng hàm `polygon` để viết hàm `arc`.

Thay vào đó, chúng ta sẽ tạo ra phiên bản tổng quát hơn của `polygon`, gọi là `polyline`.

```
def polyline(n, length, angle):
    for i in range(n):
        forward(length)
        left(angle)
```

Hàm `polyline` nhận các tham số là số đoạn thẳng cần vẽ là `n`, độ dài của các đoạn là `length`, và góc giữa chúng là `angle`.

Bây giờ, chúng ta có thể viết lại hàm `polygon` để sử dụng `polyline`.

```
def polygon(n, length):
    angle = 360.0 / n
    polyline(n, length, angle)
```

Và chúng ta có thể sử dụng hàm `polyline` để viết hàm `arc`.

```
def arc(radius, angle):
    arc_length = 2 * math.pi * radius * angle / 360
    n = 30
    length = arc_length / n
    step_angle = angle / n
    polyline(n, length, step_angle)
```

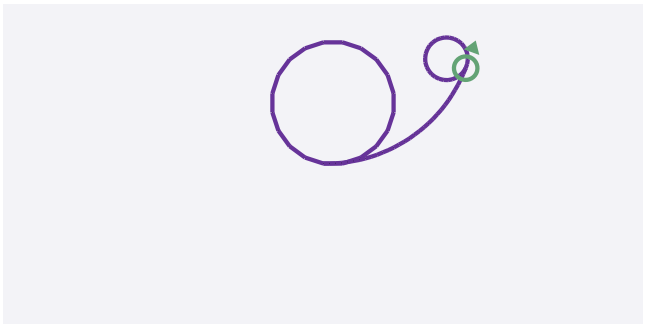
Hàm `arc` tương tự như hàm `circle`, nhưng nó tính toán `arc_length`, đó là một phần của chu vi của hình tròn.

Cuối cùng, chúng ta có thể viết lại hàm `circle` để sử dụng hàm `arc`.

```
def circle(radius):
    arc(radius,  360)
```

Để kiểm tra rằng các hàm này hoạt động như mong đợi, chúng ta sẽ sử dụng chúng để vẽ một hình gì đó giống như một con ốc sên. Với `delay=0`, chú rùa sẽ chạy nhanh nhất có thể.

```
make_turtle(delay=0)
polygon(n=20, length=9)
arc(radius=70, angle=70)
circle(radius=10)
```



Trong ví dụ này, chúng ta đã bắt đầu với mã hoạt động và tổ chức lại nó với các hàm khác nhau. Những thay đổi như vậy, giúp cải thiện mã mà không thay đổi hành vi của nó, được gọi là **tổ chức lại mã**.

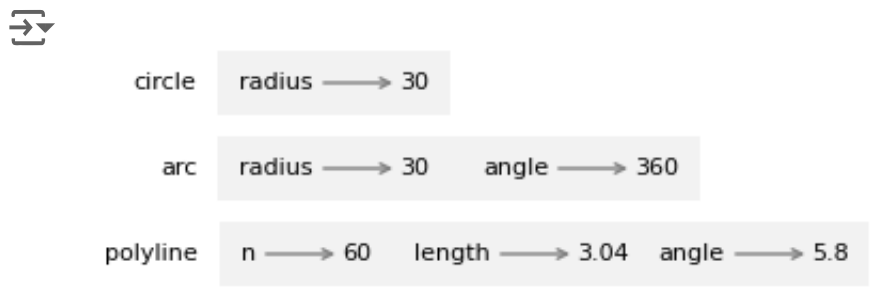
Nếu chúng ta đã lên kế hoạch trước, có thể chúng ta đã viết hàm `polyline` trước và tránh được việc tổ chức lại mã, nhưng thường thì bạn không biết đủ thông tin ngay từ đầu của một dự án để thiết kế tất cả các hàm. Khi bạn bắt đầu lập trình, bạn hiểu rõ hơn về vấn đề. Đôi khi, việc tổ chức lại mã là dấu hiệu cho thấy bạn đã học được điều gì đó.

Sơ đồ ngăn xếp

Khi chúng ta gọi hàm `circle`, nó sẽ gọi hàm `arc`, và hàm `arc` sẽ gọi hàm `polyline`. Chúng ta có thể sử dụng một sơ đồ ngăn xếp để hiển thị chuỗi các cuộc gọi hàm này và các tham số cho mỗi hàm.

```
from diagram import make_binding, make_frame, Frame, Stack

frame1 = make_frame(dict(radius=30), name='circle', loc='left')

frame2 = make_frame(dict(radius=30, angle=360), name='arc', loc='left', dx=1.1)

frame3 = make_frame(dict(n=60, length=3.04, angle=5.8),
                    name='polyline', loc='left', dx=1.1, offsetx=-0.27)

stack = Stack([frame1, frame2, frame3], dy=-0.4)
```

```
from diagram import diagram, adjust

width, height, x, y = [3.58, 1.31, 0.98, 1.06]
ax = diagram(width, height)
bbox = stack.draw(ax, x, y)
#adjust(x, y, bbox)
```



Lưu ý rằng giá trị của tham số `angle` trong hàm `polyline` khác với giá trị của tham số `angle` trong hàm `arc`. Các tham số là biến cục bộ, có nghĩa là bạn có thể sử dụng cùng một tên tham số trong các hàm khác nhau; nó là một biến khác nhau trong mỗi hàm và có thể tham chiếu đến các giá trị khác nhau.

Kế hoạch phát triển

Kế hoạch phát triển là một quy trình để viết chương trình. Quy trình mà chúng ta đã sử dụng trong chương này là "tổ chức lại mã và tổng quát hóa". Các bước của quy trình này là:

1. Bắt đầu bằng cách viết một chương trình nhỏ mà không có định nghĩa hàm.
2. Khi chương trình hoạt động, xác định một phần hợp lý trong nó, tổ chức phần đó thành một hàm và đặt cho nó một tên.
3. Tổng quát hóa hàm bằng cách thêm các tham số thích hợp.
4. Lặp lại các bước 1 đến 3 cho đến khi bạn có một tập hợp các hàm hoạt động.
5. Tìm kiếm cơ hội để cải thiện chương trình bằng cách tổ chức lại mã. Ví dụ, nếu bạn có mã tương tự ở nhiều nơi, hãy xem xét việc phân tách nó thành một hàm tổng quát phù hợp.

Quy trình này có một số nhược điểm - chúng ta sẽ thấy các lựa chọn thay thế sau này - nhưng nó có thể hữu ích nếu bạn không biết trước cách chia chương trình thành các hàm. Cách tiếp cận này cho phép bạn thiết kế trong quá trình làm việc.

Thiết kế của một hàm có hai phần:

- **Giao diện** là cách hàm được sử dụng, bao gồm tên của nó, các tham số mà nó nhận và những gì hàm đó được kỳ vọng thực hiện.
- **Cài đặt** là cách mà hàm thực hiện những gì nó được kỳ vọng làm.

Ví dụ, đây là phiên bản đầu tiên của hàm `circle` mà chúng ta đã viết, sử dụng hàm `polygon`.

```
def circle(radius):
    circumference = 2 * math.pi * radius
    n = 30
    length = circumference / n
    polygon(n, length)
```

Và đây là phiên bản đã được tái cấu trúc sử dụng hàm `arc`.

```
def circle(radius):
    arc(radius,  360)
```

Hai hàm này có cùng một giao diện — chúng nhận các tham số giống nhau và thực hiện cùng một công việc — nhưng chúng có các cài đặt khác nhau.

Docstring trong Python

Tài liệu hướng dẫn là một chuỗi được đặt ở đầu hàm nhằm giải thích giao diện của hàm đó (từ "doc" là viết tắt của "documentation - tài liệu"). Dưới đây là một ví dụ:

```
def polyline(n, length, angle):
    """Vẽ các đoạn thẳng với độ dài và góc giữa chúng đã cho.

    n: số nguyên đại diện cho số đoạn thẳng
    length: độ dài của các đoạn thẳng
    angle: góc giữa các đoạn thẳng (tính bằng độ)
    """
    for i in range(n):
        forward(length)
        left(angle)
```

Theo quy ước, tài liệu hướng dẫn là các chuỗi được đặt trong ba dấu nháy đôi, còn được gọi là chuỗi đa dòng vì ba dấu nháy đôi cho phép chuỗi trải dài qua nhiều dòng.

Một tài liệu hướng dẫn nên:

- Giải thích ngắn gọn về những gì hàm thực hiện, mà không đi vào chi tiết về cách nó hoạt động,
- Giải thích tác động của từng tham số đối với hành vi của hàm, và
- Chỉ ra kiểu của từng tham số nên là gì, nếu điều đó không rõ ràng.

Việc viết loại tài liệu này là một phần quan trọng trong thiết kế giao diện. Một giao diện được thiết kế tốt nên đơn giản để giải thích; nếu bạn gặp khó khăn trong việc giải thích một trong những hàm của mình, có thể giao diện đó cần được cải thiện.

Gỡ lỗi

Giao diện giống như một hợp đồng giữa một hàm và một người gọi. Người gọi đồng ý cung cấp một số tham số nhất định và hàm đồng ý thực hiện một số công việc nhất định.

Ví dụ, `polyline` yêu cầu ba tham số:

- `n` phải là một số nguyên;
- `length` nên là một số dương; và
- `angle` phải là một số, được hiểu là tính bằng độ.

Các yêu cầu này được gọi là **điều kiện tiên quyết** vì chúng phải đứng trước khi hàm bắt đầu thực thi. Ngược lại, các điều kiện ở cuối hàm được gọi là điều kiện hậu kiểm. **Điều kiện hậu kiểm** bao gồm hiệu ứng mong đợi của hàm (như vẽ các đoạn thẳng) và bất kỳ tác động nào (như di chuyển con rùa hoặc thực hiện các thay đổi khác).

Điều kiện tiên quyết là trách nhiệm của người gọi. Nếu người gọi vi phạm một điều kiện tiên quyết và hàm không hoạt động đúng, lỗi nằm ở phía người gọi, không phải hàm.

Nếu các điều kiện tiên quyết được thỏa mãn và các điều kiện hậu kiểm không được đáp ứng, lỗi nằm trong hàm. Nếu các điều kiện tiên quyết và hậu kiểm của bạn rõ ràng, chúng có thể giúp ích cho việc gỡ lỗi.

Thuật ngữ

- **interface design - thiết kế giao diện:** một quy trình để thiết kế giao diện của một hàm, bao gồm các tham số mà hàm đó nên nhận.
- **canvas - bảng vẽ:** một cửa sổ được sử dụng để hiển thị các yếu tố đồ họa bao gồm đường thẳng, hình tròn, hình chữ nhật và các hình dạng khác.
- **encapsulation - đóng gói:** quy trình chuyển đổi một chuỗi các câu lệnh thành một định nghĩa hàm.
- **generalization - tổng quát hóa:** quy trình thay thế điều gì đó quá cụ thể (như một số) bằng điều gì đó tổng quát hơn (như một biến hoặc tham số).
- **keyword argument - tham số từ khóa:** một tham số bao gồm tên của tham số đó.
- **refactoring - tái cấu trúc:** quy trình sửa đổi một chương trình đang hoạt động để cải thiện giao diện hàm và các đặc điểm khác của mã nguồn.
- **development plan - kế hoạch phát triển:** một quy trình để viết các chương trình.
- **docstring - tài liệu hướng dẫn:** một chuỗi xuất hiện ở đầu một định nghĩa hàm để tài liệu hóa giao diện của hàm đó.
- **multiline string - chuỗi nhiều dòng:** một chuỗi được bao quanh bởi dấu ba nháy đôi, có thể trải dài qua nhiều dòng của chương trình.
- **precondition - điều kiện tiên quyết:** một yêu cầu mà người gọi phải thỏa mãn trước khi một hàm bắt đầu.
- **postcondition - điều kiện hậu kiểm:** một yêu cầu mà hàm phải thỏa mãn trước khi kết thúc.

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 một trở lý ảo

Có nhiều mô-đun giống như `jupyturtle` trong Python, và mô-đun mà chúng ta đã sử dụng trong chương này đã được tùy chỉnh cho cuốn sách này. Vì vậy, nếu bạn hỏi một trợ lý ảo để được giúp đỡ, nó sẽ không biết mô-đun nào để sử dụng. Nhưng nếu bạn cung cấp cho nó vài ví dụ để làm việc, nó có thể hiểu ra. Ví dụ, hãy thử lời nhắc này và xem liệu nó có thể viết một hàm vẽ một hình xoắn ốc hay không:

```
Chương trình sau sử dụng một mô-đun đồ họa turtle để vẽ một hình tròn:

from jupyturtle import make_turtle, forward, left
import math

def polygon(n, length):
    angle = 360 / n
    for i in range(n):
        forward(length)
        left(angle)

def circle(radius):
    circumference = 2 * math.pi * radius
    n = 30
    length = circumference / n
    polygon(n, length)

make_turtle(delay=0)
circle(30)

Viết một hàm vẽ một hình xoắn ốc.
```

Hãy nhớ rằng kết quả có thể sử dụng những tính năng mà chúng ta chưa thấy, và nó có thể có lỗi. Sao chép mã từ trợ lý ảo và xem liệu bạn có thể làm cho nó hoạt động. Nếu bạn không đạt được điều mình muốn, hãy thử sửa đổi lời nhắc.

Đối với các bài tập bên dưới, có một vài hàm rùa khác mà bạn có thể muốn sử dụng.

- `penup` nâng bút ảo của con rùa lên để nó không để lại dấu vết khi di chuyển.
- `pendown` hạ bút xuống trở lại.

Hàm sau sử dụng `penup` và `pendown` để di chuyển rùa mà không để lại dấu vết.

```
from jupyturtle import penup, pendown

def jump(length):
    """Move forward length units without leaving a trail.

    Postcondition: Leaves the pen down.
    """
    penup()
    forward(length)
    pendown()
```

Bài tập 1

Viết một hàm có tên là `rectangle` để vẽ một hình chữ nhật với các chiều dài cạnh đã cho. Ví dụ, đây là một hình chữ nhật rộng `80` đơn vị và cao `40` đơn vị.

Bạn có thể sử dụng đoạn mã sau để kiểm tra hàm của bạn.

```
make_turtle()
rectangle(80, 40)
```

Bài tập 2

Viết một hàm có tên là `rhombus` để vẽ một hình thoi với độ dài cạnh cho trước và một góc trong cho trước. Ví dụ, đây là một hình thoi với độ dài cạnh là `50` và góc trong là `60` độ.

Bạn có thể sử dụng đoạn mã sau để kiểm tra hàm của mình.

```
make_turtle()
rhombus(50, 60)
```

Bài tập 3

Bây giờ hãy viết một hàm tổng quát hơn có tên là `parallelogram` để vẽ một tứ giác có các cạnh song song. Sau đó, hãy viết lại các hàm `rectangle` và `rhombus` để sử dụng `parallelogram`.

Bạn có thể sử dụng đoạn mã sau để kiểm tra các hàm của mình.

```
make_turtle(width=400)
jump(-120)

rectangle(80, 40)
jump(100)
rhombus(50, 60)
jump(80)
parallelogram(80, 50, 60)
```

Bài tập 4

Viết một tập hợp các hàm tổng quát phù hợp có thể vẽ các hình dạng như thế này.

Gợi ý: Viết một hàm gọi là `triangle` để vẽ một đoạn hình tam giác, và sau đó viết một hàm gọi là `draw_pie` sử dụng hàm `triangle`.

Bạn có thể sử dụng đoạn mã sau để kiểm tra các hàm của mình.

```
make_turtle(delay=0)
jump(-80)

size = 40
draw_pie(5, size)
jump(2*size)
draw_pie(6, size)
jump(2*size)
draw_pie(7, size)
```

Bài tập 5

Viết một tập hợp các hàm tổng quát phù hợp có thể vẽ những bông hoa như thế này.

Gợi ý: Sử dụng hàm `arc` để viết một hàm gọi là `petal` để vẽ một cánh hoa.

Bạn có thể sử dụng mã sau để kiểm tra các hàm của mình.

Bởi vì giải pháp vẽ nhiều đoạn thẳng nhỏ, nó có xu hướng chậm lại khi thực thi. Để tránh điều đó, bạn có thể thêm tham số từ khóa `auto_render=False` để tránh việc vẽ sau mỗi bước, và sau đó gọi hàm `render` ở cuối để hiển thị kết quả.

Trong khi bạn đang gỡ lỗi, bạn có thể muốn bỏ `auto_render=False`.

```
from jupyturtle import render

make_turtle(auto_render=False)

jump(-60)
n = 7
radius = 60
angle = 60
flower(n, radius, angle)

jump(120)
n = 9
radius = 40
angle = 85
flower(n, radius, angle)

render()
```