Chapter 4
Functions and Interfaces

A. Downey, *Think Python: How to Think Like a Computer Science,* 3rd ed., O'Reilly, 2024.

https://allendowney.github.io/ThinkPython/

Contents

1. The turtle Module

- https://docs.python.org/3/library/turtle.html
- In Python, turtle graphics provides a representation of a physical "turtle" (a little robot with a pen) that draws on a sheet of paper on the floor.
- Imagine a robotic turtle starting at (0, 0) in the x-y plane. After an `import turtle` statement, give it the command `turtle.forward(15)`, and it moves (on-screen!) 15 pixels in the direction it is facing, drawing a line as it moves. Give it the command `turtle.right(25)`, and it rotates in-place 25 degrees clockwise.

1. The turtle Module

- To use turtle module, we need to import it like this:

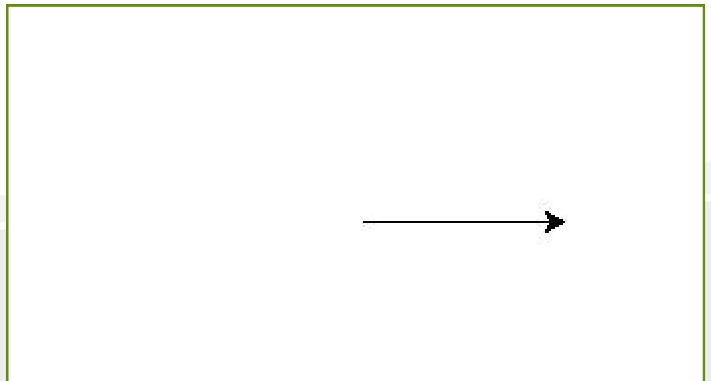
```
import turtle
```

- Now we can use the functions defined in the module, like **forward**():

```
import turtle
```

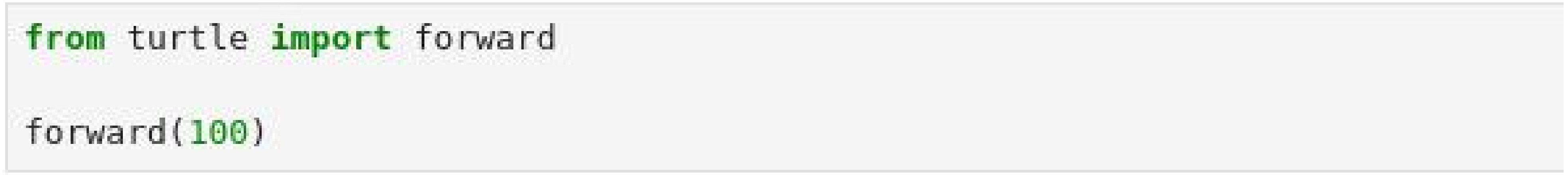
```
turtle.forward(100)
```

  - **forward**() moves the turtle a given distance in the direction it's facing, drawing a line segment along the way. The distance is in arbitrary units—the actual size depends on your computer's screen.



1. The turtle Module

- If we will use functions defined in the **turtle** module many times, so it would be nice if we did not have to write the name of the module every time. That's possible if we import the module like this:

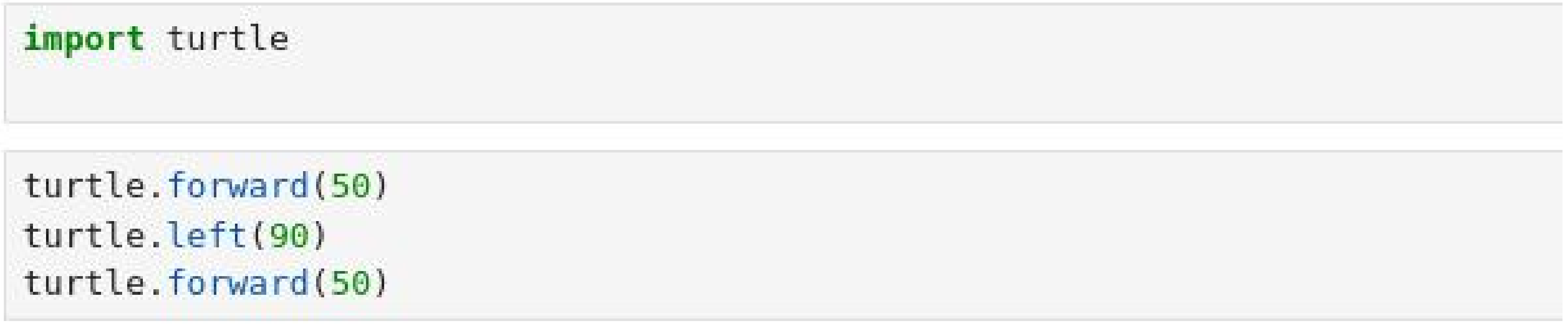


```python
from turtle import forward

forward(100)
```

- turtle.**right**(*angle*)
  - Turn turtle right by angle units. (Units are by default degrees)
- turtle.**left**(*angle*)
  - Turn turtle left by angle units. (Units are by default degrees)

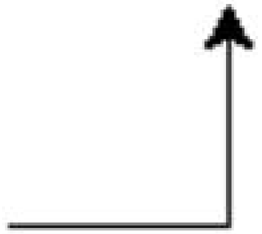
1. The turtle Module

- The following program moves the turtle east and then north, leaving two line segments behind.



```
import turtle
```

```
turtle.forward(50)
turtle.left(90)
turtle.forward(50)
```



2. Making a Square

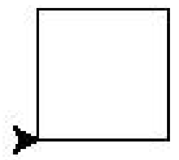
```python
import turtle
```

```python
turtle.forward(50)
turtle.left(90)

turtle.forward(50)
turtle.left(90)

turtle.forward(50)
turtle.left(90)

turtle.forward(50)
turtle.left(90)
```

- Because the program repeats the same pair of lines four times, we can do the same thing more concisely with a for loop:

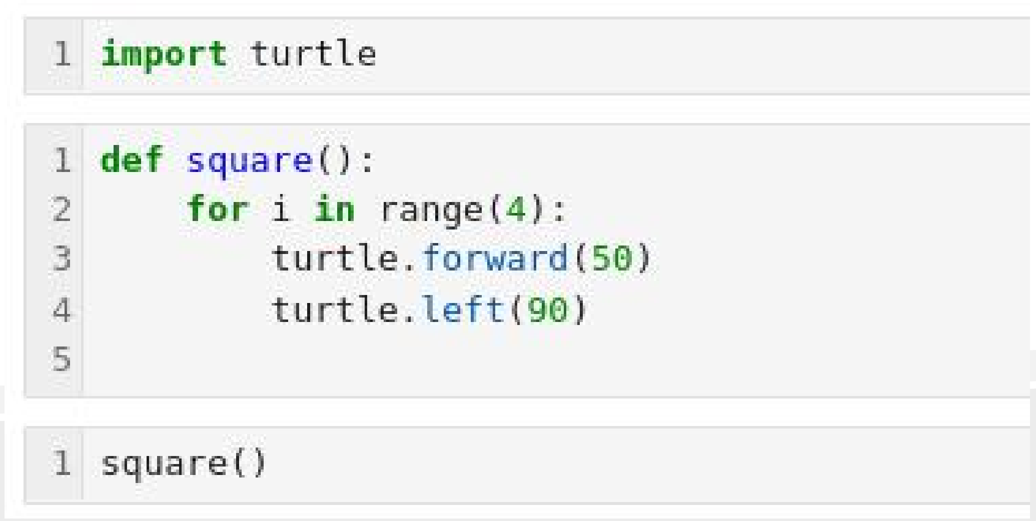
```python
import turtle
```

```python
for i in range(4):
    turtle.forward(50)
    turtle.left(90)

```



3. Encapsulation and Generation

- Define a function square() to draw a quare:
  - Wrapping a piece of code up in a function is called encapsulation. One of the benefits of encapsulation is that it attaches a name to the code, which serves as a kind of documentation.
  - Another advantage is that if you re-use the code, it is more concise to call a function twice than to copy and paste the body!



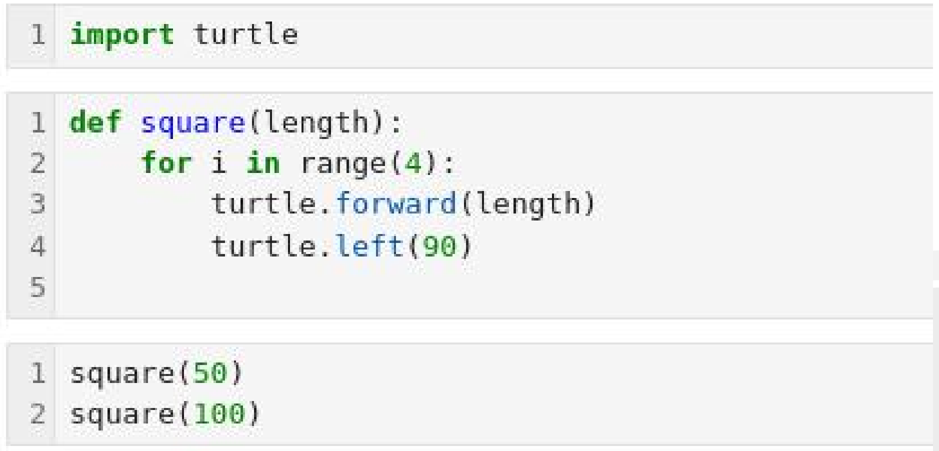
```python
import turtle
```

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

```

```python
square()
```

3. Encapsulation and Generation

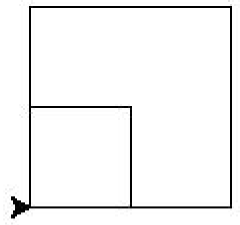
- In the current version, the size of the square is always 50. If we want to draw squares with different sizes, we can take the length of the sides as a parameter:
  - We can draw squares with different sizes.
- Adding a parameter to a function is called **generalization** because it makes the function more general: with the previous version, the square is always the same size; with this version it can be any size.



```
import turtle
```

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

```

```
square(50)
square(100)
```



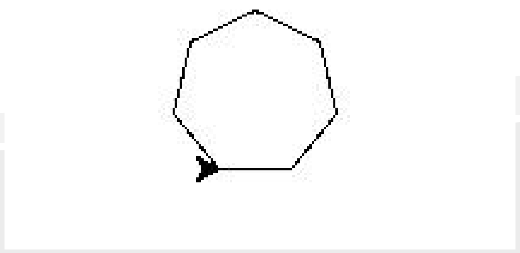
3. Encapsulation and Generation

- If we add another parameter, we can make it even more general. The following function draws regular polygons with a given number of sides:

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

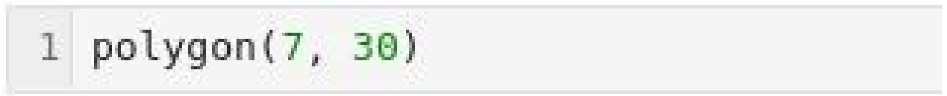
- In a regular polygon with *n* sides, the angle between adjacent sides is 360 / *n* degrees. The following example draws a 7-sided polygon with side length of 30:

```python
polygon(7, 30)
```

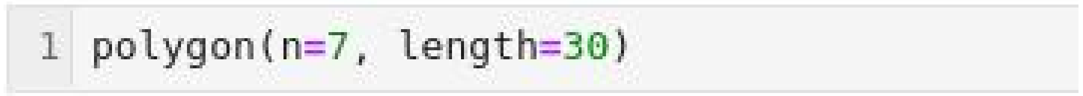


3. Encapsulation and Generation

- When a function has more than a few numeric arguments, it is easy to forget what they are, or what order they should be in. It can be a good idea to include the names of the parameters in the argument list:



```python
polygon(7, 30)
```

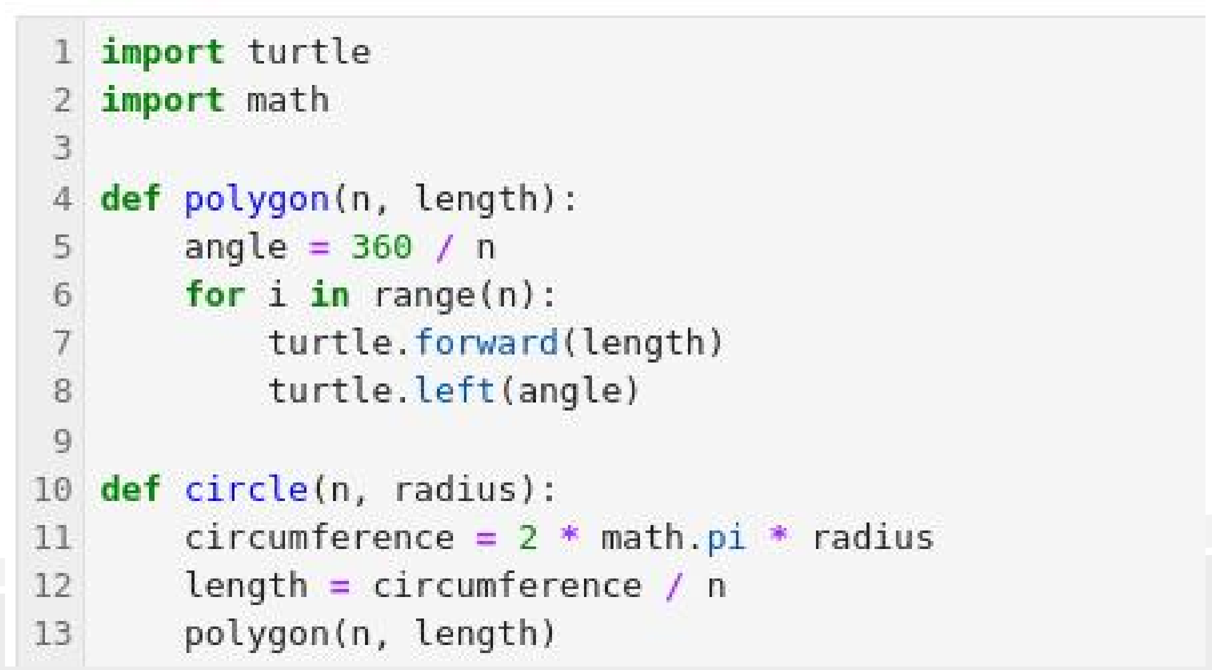


```python
polygon(n=7, length=30)
```

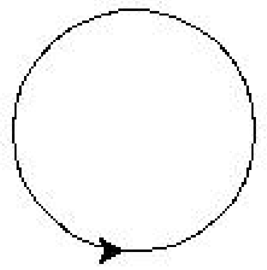
- These are sometimes called “named arguments” because they include the parameter names. But in Python they are more often called **keyword arguments**.

4. Approximating a Circle

- A circle can be approximated by a polygon with a large number of sides, so each side is small enough that it's hard to see. Here is a function that uses polygon to draw a 30-sided polygon that approximates a circle:



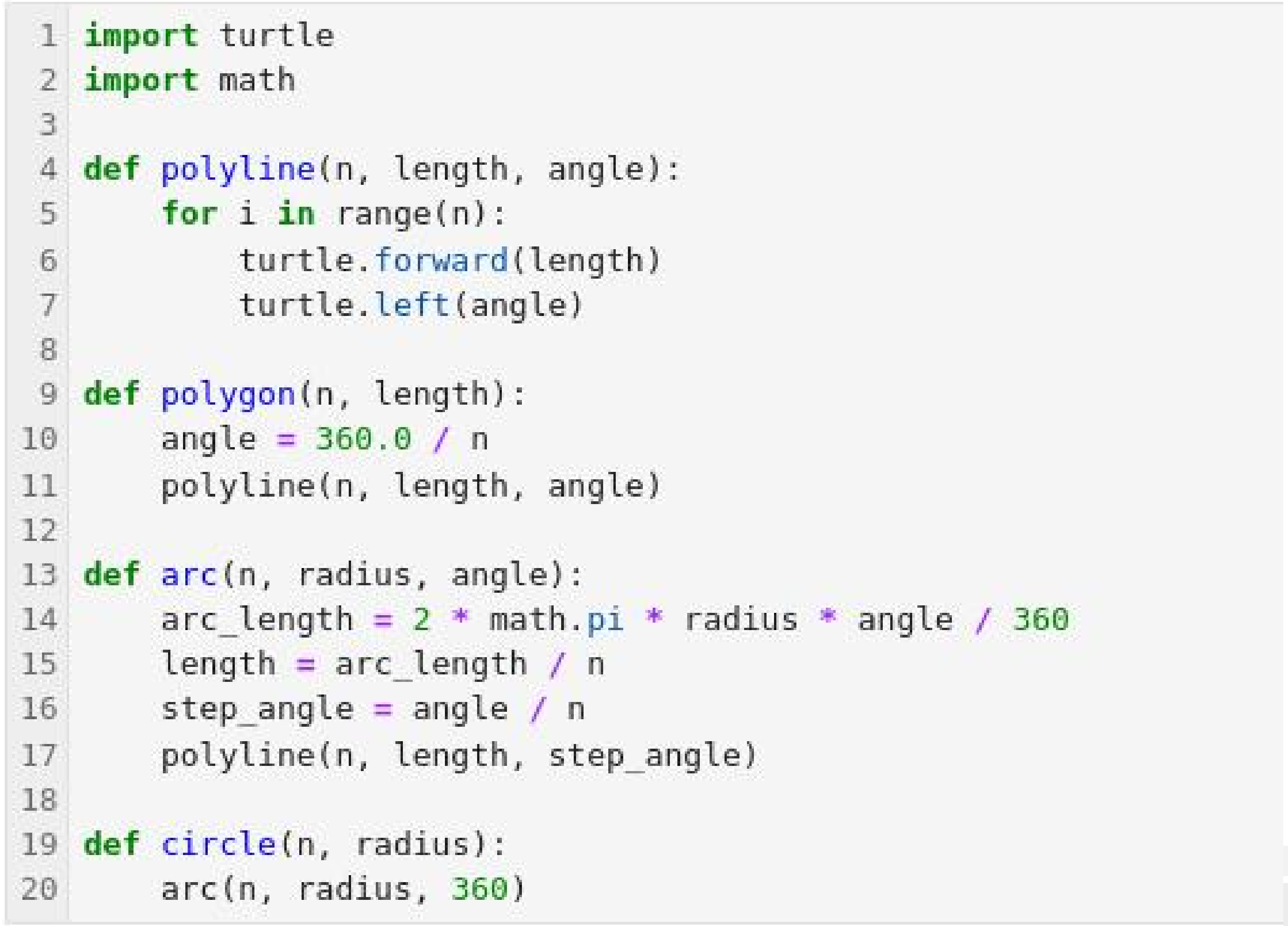
```python
import turtle
import math

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

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



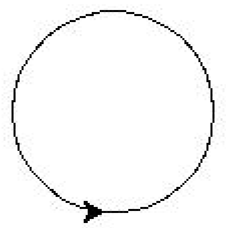
5. Refactoring

- How to draw an arc of a circle?
  - Now let's write a more general version of circle, called arc, that takes a second parameter, angle, and draws an arc of a circle that spans the given angle.
    - For example, if angle is 360 degrees, it draws a complete circle. If angle is 180 degrees, it draws a half circle.
- To write circle, we were able to reuse polygon, because a many-sided polygon is a good approximation of a circle. But we can't use polygon to write arc.
  - Instead, we'll create the more general version of polygon, called polyline.
- In this example, we started with working code and reorganized it with different functions. Changes like this, which improve the code without changing its behavior, are called **refactoring**.

5. Refactoring

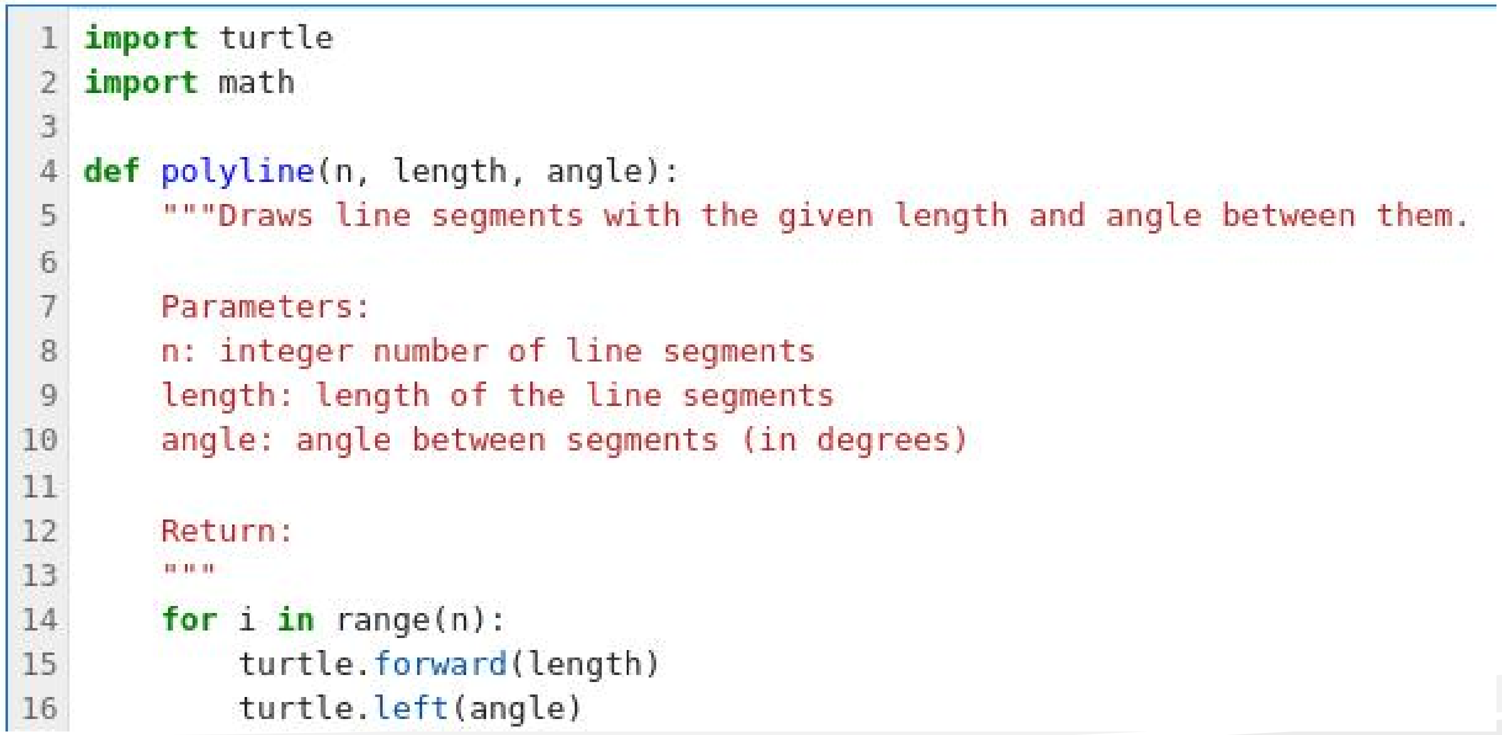


```python
import turtle
import math

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

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

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

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



6. Docstring

- A **docstring** is a string at the beginning of a function that explains the interface (“doc” is short for “documentation”). Here is an example:



```python
import turtle
import math

def polyline(n, length, angle):
    """Draws line segments with the given length and angle between them.

    Parameters:
    n: integer number of line segments
    length: length of the line segments
    angle: angle between segments (in degrees)

    Return:
    """
    for i in range(n):
        turtle.forward(length)
        turtle.left(angle)
```

6. Docstring

- By convention, docstrings are triple-quoted strings, also known as **multiline strings** because the triple quotes allow the string to span more than one line.
- A docstring should:
  - Explain concisely what the function does, without getting into the details of how it works,
  - Explain what effect each parameter has on the behavior of the function, and
  - Indicate what type each parameter should be, if it is not obvious.
  - Explain consiely what the return value is and indicate what type the return value should be.
- Writing this kind of documentation is an important part of interface design. A well-designed interface should be simple to explain; if you have a hard time explaining one of your functions, maybe the interface could be improved.