chap04

September 6, 2023

[47]:
```
from IPython.core.magic import register_cell_magic
from IPython.core.magic_arguments import argument, magic_arguments,␣
 ↪parse_argstring

@register_cell_magic
def expect_error(line, cell):
    try:
        exec(cell)
    except Exception as e:
        %tb


@magic_arguments()
@argument('exception', help='Type of exception to catch')
@register_cell_magic
def expect(line, cell):
    args = parse_argstring(expect, line)
    exception = eval(args.exception)
    try:
        exec(cell)
    except exception as e:
        %tb
```

[48]:
```
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

1 Case study: interface design

This chapter introduces a module called `Turtle`, which allows you to create simple drawings by giving instructions to an imaginary turtle. We will use this module to write functions that draw squares, polygons, and circles – and to demonstrate **interface design**, which is a way of designing functions that work together.

This chapter is the first of several case studies, which means there is less new content than in other chapters, and more exercises.

[49]:
```
import os

if not os.path.exists('Turtle.py'):
    !wget https://gist.github.com/AllenDowney/66a18dec4864b1a19c52cd98bba08f73
```

1.1 The Turtle module

To use the `Turtle` module, we can import it like this.

[50]:
```
import Turtle
```

Now we can use the functions defined in the module, like `make_turtle` and `forward`.

[51]:
```
Turtle.make_turtle()
Turtle.forward(100)
```

```
<IPython.core.display.HTML object>
```

`make_turtle` creates a **canvas**, which is a space on the screen where we can draw, and a turtle, which is represented by a circular shell and a triangular head. The circle shows the location of the turtle and the triangle indicates the direction it is facing.

`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.

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.

[52]:
```
from Turtle import make_turtle, forward
```

This version of the import statement imports `make_turtle` and `forward` from the `Turtle` module so we can call them like this.

[53]:
```
make_turtle()
forward(100)
```

```
<IPython.core.display.HTML object>
```

`Turtle` provides two other functions we'll use, called `left` and `right`. We'll import them like this.

[54]:
```
from Turtle import left, right
```

`left` causes the turtle to turn left. It takes one argument, which is the angle of the turn in degrees. For example, we can make a 90 degree left turn like this.

[55]:
```
make_turtle()
forward(50)
left(90)
forward(50)
```

```
<IPython.core.display.HTML object>
```

This program moves the turtle east and then north, leaving two line segments behind. Before you go on, see if you can modify the previous program to make a square.

1.2 Making a square

Here's one way to make a square.

[56]:
```
make_turtle()

forward(50)
left(90)

forward(50)
left(90)

forward(50)
left(90)

forward(50)
left(90)
```

```
<IPython.core.display.HTML object>
```

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

[57]:
```
make_turtle()
for i in range(4):
    forward(50)
    left(90)
```

```
<IPython.core.display.HTML object>
```

That's everything you need to know about drawing with turtles. Now you can work on the exercises.

1.3 Exercises

The following is a series of exercises using the `Turtle` module. They are meant to be fun, but they also demonstrate a point that I will explain soon. The solutions are in the next section, so don't go on until you have at least attempted each exercise.

1. Put the square-drawing code from the previous section in a function called `square`. Call your function to test it.
2. Add a parameter named `length` to `square`. Test your program with a range of values for `length`.
3. Make a copy of `square` and change the name to `polygon`. Add another parameter named `n` and modify the body so it draws an `n`-sided regular polygon. Hint: The exterior angles of an `n`-sided regular polygon are `360 / n` degrees.

4. Write a function called `circle` that takes a floating-point number called `radius` as a parameters and that draws an approximate circle, with the given radius, by calling `polygon` with an appropriate length and number of sides. Test your function with a range of values of `radius`. Hint: The circumference of a circle with radius `radius` is `2 * math.pi * radius`.

5. Write a function called `arc` that draws an arc of a circle. It should take two parameters: `radius` is the radius of the circle and `angle` is the angle of the arc in degrees. If `angle=180`, `arc` should draw a half circle. If `angle=360`, `arc` should draw a whole circle. Hint: The length of an arc is `2 * math.pi * radius * angle / 360`.

1.4 Encapsulation

The first exercise asks you to put the square-drawing code in a function definition and then call the function. Here is a solution:

[58]:
```
def square():
    for i in range(4):
        forward(50)
        left(90)
```

[59]:
```
make_turtle()
square()
```

```
<IPython.core.display.HTML object>
```

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!

1.5 Generalization

The next step is to add a `length` parameter to `square`. Here is a solution:

[60]:
```
def square(length):
    for i in range(4):
        forward(length)
        left(90)
```

[61]:
```
make_turtle()
square(30)
square(60)
```

```
<IPython.core.display.HTML object>
```

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.

The next exercise is also a generalization. `polygon` is a more general version of `square` that draws regular polygons with any number of sides. Here is a solution:

[62]:
```
def polygon(n, length):
    angle = 360 / n
    for i in range(n):
        forward(length)
        left(angle)
```

[63]:
```
make_turtle()
polygon(7, 30)
```

```
<IPython.core.display.HTML object>
```

This example draws a 7-sided polygon with side length 30.

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:

[64]:
```
make_turtle()
polygon(n=7, length=30)
```

```
<IPython.core.display.HTML object>
```

These are sometimes called "named arguments" because they include the parameter names. But in Python they are more often called **keyword arguments** (not to be confused with Python keywords like `for` and `def`).

This syntax makes the program more readable. It is also a reminder about how arguments and parameters work: when you call a function, the arguments are assigned to the parameters.

1.6 Interface design

The next step is to write `circle`, which takes `radius` as a parameter. Here is a solution that uses `polygon` to draw a 30-sided polygon that approximates a circle.

[65]:
```
import math

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

`circumference` is the circumference of a circle with the given radius. `n` is the number of line segments, so `length` is the length of each segment.

This function might take a long time to run. We can speed it up by calling `make_turtle` with a keyword argument called `speed` that controls the speed of the turtle.

[66]:
```
make_turtle(speed=10)
circle(30)
```

```
<IPython.core.display.HTML object>
```

A limitation of this solution is that `n` is a constant, which means that for very big circles, the line segments are too long, and for small circles, we waste time drawing very small segments. One option is to generalize the function by taking `n` as a parameter. But let's keep it simple for now.

1.7 Refactoring

Now let's get to the last exercise, drawing an arc of a circle. When I wrote `circle`, I was able to re-use `polygon` because a many-sided polygon is a good approximation of a circle. But `arc` is not as cooperative; we can't use `polygon` or `circle` to draw an arc. One approach is to start with a copy of `polygon` and transform it into `arc`. The result might look like this:

[67]:
```
def arc(radius,  angle):
    arc_length = 2 * math.pi * radius * angle / 360
    n = 30
    step_length = arc_length / n
    step_angle = angle / n

    for i in range(n):
        forward(step_length)
        left(step_angle)
```

The second half of this function looks like `polygon`, but we can't re-use `polygon` without changing the interface. We could generalize `polygon` to take an angle as a third argument, but then `polygon` would no longer be an appropriate name! Instead, we'll create the more general function `polyline`:

[68]:
```
def polyline(n, length, angle):
    for i in range(n):
        forward(length)
        left(angle)
```

Now we can rewrite `polygon` and `arc` to use `polyline`:

[69]:
```
def polygon(n, length):
    angle = 360.0 / n
    polyline(n, length, angle)
```

[70]:
```
def arc(radius,  angle):
    arc_length = 2 * math.pi * radius * angle / 360
    n = 30
    step_length = arc_length / n
    step_angle = float(angle) / n
    polyline(n, step_length, step_angle)
```

Finally, we can rewrite `circle` to use `arc`:

[71]:
```
def circle(radius):
    arc(radius,  360)
```

[72]:
```
make_turtle(speed=10)
circle(30)
```

```
<IPython.core.display.HTML object>
```

This process – rearranging a program to improve interfaces and facilitate code re-use – is called **refactoring**. In this case, we noticed that there was similar code in `arc` and `polygon`, so we "factored it out" into `polyline`.

If we had planned ahead, we might have written `polyline` first and avoided refactoring, but often you don't know enough at the beginning of a project to design all the interfaces. Once you start coding, you understand the problem better. Sometimes refactoring is a sign that you have learned something.

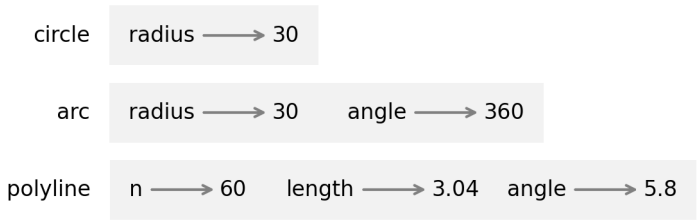
1.8 Stack diagram

When we call `circle`, it calls `arc`, which calls `polyline`. We can use a stack diagram to show this sequence of function calls and the parameters for each one.

[110]:
```
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)
```

[111]:
```
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)
```

[111]:
```
[3.58, 1.31, 0.98, 1.06]
```



Notice that the value of `angle` in `polyline` is difference from the value of `angle` in `arc`. Parameters are local, which means you can use the same parameter name in different functions; it's a different variable in each function, and it can refer to a different value.

1.9 A development plan

A **development plan** is a process for writing programs. The process we used in this case study is "encapsulation and generalization". The steps of this process are:

1. Start by writing a small program with no function definitions.
2. Once you get the program working, identify a coherent piece of it, encapsulate the piece in a function and give it a name. Copy and paste working code to avoid retyping (and re-debugging).
3. Generalize the function by adding appropriate parameters.
4. Repeat Steps 1 to 3 until you have a set of working functions.
5. Look for opportunities to improve the program by refactoring. For example, if you have similar code in several places, consider factoring it into an appropriately general function.

This process has some drawbacks – we will see alternatives later – but it can be useful if you don't know ahead of time how to divide the program into functions. This approach lets you design as you go along.

1.10 Docstrings

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

[75]:
```
def polyline(n, length, angle):
    """Draws line segments with the given length and angle between them.

    n: integer number of line segments
    length: length of the line segments
    angle: angle between segments (in degrees)
    """
    for i in range(n):
        forward(length)
        left(angle)
```

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 does it,
- 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.

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.

1.11 Debugging

An interface is like a contract between a function and a caller. The caller agrees to provide certain arguments and the function agrees to do certain work.

For example, `polyline` requires three arguments: `n` has to be an integer; `length` should be a positive number; and `angle` has to be a number, which is understood to be in degrees.

These requirements are called **preconditions** because they are supposed to be true before the function starts executing. Conversely, conditions at the end of the function are **postconditions**. Postconditions include the intended effect of the function (like drawing line segments) and any side effects (like moving the turtle or making other changes).

Preconditions are the responsibility of the caller. If the caller violates a precondition and the function doesn't work correctly, the bug is in the caller, not the function.

If the preconditions are satisfied and the postconditions are not, the bug is in the function. If your pre- and postconditions are clear, they can help with debugging.

1.12 Glossary

NOTE: None of the glossaries are up-to-date. I will update them when the content of the chapters is settled.

loop:

A part of a program that can run repeatedly.

encapsulation:

The process of transforming a sequence of statements into a function definition.

generalization:

The process of replacing something unnecessarily specific (like a number) with something appropriately general (like a variable or parameter).

keyword argument:

An argument that includes the name of the parameter as a "keyword".

interface:

A description of how to use a function, including the name and descriptions of the arguments and return value.

refactoring:

The process of modifying a working program to improve function interfaces and other qualities of the code.

development plan:

A process for writing programs.

docstring:

A string that appears at the top of a function definition to document the function's interface.

precondition:

A requirement that should be satisfied by the caller before a function starts.

postcondition:

A requirement that should be satisfied by the function before it ends.

1.13 Exercises

For the exercises below, there are a few more turtle functions you might want to use.

- `penup` lifts the turtle's imaginary pen so it doesn't leave a trail when it moves.
- `pendown` puts the pen back down.

The following function uses `penup` and `pendown` to move the turtle without leaving a trail.

[76]:
```
from Turtle import penup, pendown

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

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

1.13.1 Exercise

Write a function called `rectangle` that draws a rectangle with given side lengths. For example, here's a rectangle that's 80 units wide and 40 units tall.

[77]:
```
# Solution

def rectangle(length1, length2):
    """Draw a rectangle with the given lengths.

    length1: length of the first side
    length2: length of the second size
    """
    for i in range(2):
        forward(length1)
        left(90)
        forward(length2)
        left(90)
```

[78]:
```
make_turtle()
rectangle(80, 40)
```

```
<IPython.core.display.HTML object>
```

1.14 Exercise

Write a function called `rhombus` that draws a rhombus with a given side length and a given interior angle. For example, here's a rhombus with side length `50` and an interior angle of `60` degrees.

[79]:
```
def rhombus(length, angle):
    for i in range(2):
        forward(length)
        left(angle)
        forward(length)
        left(180-angle)
```

[80]:
```
make_turtle()
rhombus(50, 60)
```

```
<IPython.core.display.HTML object>
```

1.14.1 Exercise

Now write a more general function called `parallelogram` that draws a quadrilateral with parallel sides. Then rewrite `rectangle` and `rhombus` to use `parallelogram`.

[81]:
```
# Solution

def parallelogram(length1, length2, angle):
    for i in range(2):
        forward(length1)
        left(angle)
        forward(length2)
        left(180-angle)
```

[82]:
```
# Solution

def rectangle(length1, length2):
    """Draw a rectangle with the given lengths.

    length1: length of the first side
    length2: length of the second size
    """
    parallelogram(length1, length2, 90)
```

[83]:
```
# Solution

```

```
def rhombus(length, angle):
    parallelogram(length, length, angle)
```

[84]:
```
make_turtle(speed=8, width=400)
move(-120)

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

```
<IPython.core.display.HTML object>
```

1.14.2 Exercise

Write an appropriately general set of functions that can draw shapes like this.

Hint: Write a function called `triangle` that draws one triangular segment.

[85]:
```
# Solution

def triangle(radius,  angle):
    """Draws an icosceles triangle.

    The turtle starts and ends at the peak, facing the middle of the base.

    radius: length of the equal legs
    angle: half peak angle in degrees
    """
    y = radius * math.sin(angle * math.pi / 180)

    right(angle)
    forward(radius)
    left(90+angle)
    forward(2*y)
    left(90+angle)
    forward(radius)
    left(180-angle)
```

[86]:
```
# Solution

def draw_pie(n, radius):
    """Draws a pie divided into radial segments.

    n: number of segments
    radius: length of the radial spokes
    """
```

```
    angle = 360.0 / n
    for i in range(n):
        triangle(radius,  angle/2)
        left(angle)
```

[87]:
```
make_turtle(speed=12, width=400)
move(-120)

size = 40
draw_pie(5, size)
move(2*size)
draw_pie(6, size)
move(2*size)
draw_pie(7, size)
move(2*size)
draw_pie(8, size)
```

```
<IPython.core.display.HTML object>
```

1.14.3 Exercise

Write an appropriately general set of functions that can draw flowers like this.

Hint: Use `arc` to write a function called `petal` that draws one flower petal.

[88]:
```
# Solution

def petal(radius,  angle):
    """Draws a petal using two arcs.

    t: Turtle
    radius: radius of the arcs
    angle: angle (degrees) that subtends the arcs
    """
    for i in range(2):
        arc(radius,  angle)
        left(180-angle)
```

[89]:
```
# Solution

def flower(n, radius, angle):
    """Draws a flower with n petals.

    n: number of petals
    radius: radius of the arcs
    angle: angle (degrees) that subtends the arcs
    """
    for i in range(n):
```

```
        petal(radius,  angle)
        left(360.0/n)
```

[90]:
```
make_turtle(speed=12, width=400)

move(-100)
flower(7, 60.0, 60.0)

move(100)
flower(10, 40.0, 80.0)

move(100)
flower(14, 140.0, 20.0)
```

```
<IPython.core.display.HTML object>
```

1.14.4 Ask an assistant

There are several modules like `Turtle` in Python, and the one we used in this chapter has been customized for this book. So if you ask a virtual assistant for help, it won't know which module to use. But if you give it a few examples to work with, it can probably figure it out. For example, try this prompt and see if it can write a function that draws a spiral:

```
The following program uses a Turtle module to draw a circle:

from Turtle 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(speed=12)
circle(30)

Write a function that draws a spiral.
```

Keep in mind that the result might use features we have not seen yet, and it might have errors. Copy the code from the VA and see if you can get it working. If you didn't get what you wanted, try modifying the prompt.

[91]:
```
# Solution

def spiral(length, angle):
    turtle = make_turtle(speed=12)
    for _ in range(100):
        forward(length)
        right(angle)
        length += 1  # Increase the length for each segment

spiral(5, 90)
```

```
<IPython.core.display.HTML object>
```

[92]:
```
# Solution

# prompt: make that a circular spiral, and don't change the name of the module

from Turtle import make_turtle, forward, left
import math

def circular_spiral(radius, angle):
    turtle = make_turtle(speed=12)
    rotations = 5
    distance = 2 * math.pi * radius / 360  # Calculate the distance for each␣
 ↪degree of rotation
    for _ in range(rotations * 360):
        forward(distance)
        left(angle)
        radius += 0.1  # Increase the radius for each segment
        distance = 2 * math.pi * radius / 360  # Recalculate the distance for␣
 ↪the updated radius

circular_spiral(50, 1)
```

```
<IPython.core.display.HTML object>
```

[ ]:
```

```