Chapter 5

Conditionals and recursion

The main topic of this chapter is the `if` statement, which executes different code depending on the state of the program. But first I want to introduce two new operators: floor division and modulus.

5.1   Floor division and modulus

The **floor division** operator, `//`, divides two numbers and rounds down to an integer. For example, suppose the run time of a movie is 105 minutes. You might want to know how long that is in hours. Conventional division returns a floating-point number:

```
>>> minutes = 105
>>> minutes / 60
1.75
```

But we don't normally write hours with decimal points. Floor division returns the integer number of hours, rounding down:

```
>>> minutes = 105
>>> hours = minutes // 60
>>> hours
1
```

To get the remainder, you could subtract off one hour in minutes:

```
>>> remainder = minutes - hours * 60
>>> remainder
45
```

An alternative is to use the **modulus operator**, `%`, which divides two numbers and returns the remainder.

```
>>> remainder = minutes % 60
>>> remainder
45
```

The modulus operator is more useful than it seems. For example, you can check whether one number is divisible by another—if `x % y` is zero, then `x` is divisible by `y`.

Also, you can extract the right-most digit or digits from a number. For example, `x % 10` yields the right-most digit of `x` (in base 10). Similarly `x % 100` yields the last two digits.

If you are using Python 2, division works differently. The division operator, `/`, performs floor division if both operands are integers, and floating-point division if either operand is a `float`.

5.2   Boolean expressions

A **boolean expression** is an expression that is either true or false. The following examples use the operator ==, which compares two operands and produces `True` if they are equal and `False` otherwise:

```
>>> 5 == 5
True
>>> 5 == 6
False
```

`True` and `False` are special values that belong to the type `bool`; they are not strings:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

The == operator is one of the **relational operators**; the others are:

```
    x != y                  # x is not equal to y
    x > y                   # x is greater than y
    x < y                   # x is less than y
    x >= y                  # x is greater than or equal to y
    x <= y                  # x is less than or equal to y
```

Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols. A common error is to use a single equal sign (=) instead of a double equal sign (==). Remember that = is an assignment operator and == is a relational operator. There is no such thing as =< or =>.

5.3   Logical operators

There are three **logical operators**: `and`, `or`, and `not`. The semantics (meaning) of these operators is similar to their meaning in English. For example, `x > 0 and x < 10` is true only if `x` is greater than 0 *and* less than 10.

`n%2 == 0 or n%3 == 0` is true if *either or both* of the conditions is true, that is, if the number is divisible by 2 *or* 3.

Finally, the `not` operator negates a boolean expression, so `not (x > y)` is true if `x > y` is false, that is, if `x` is less than or equal to `y`.

Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as `True`:

```
>>> 42 and True
True
```

This flexibility can be useful, but there are some subtleties to it that might be confusing. You might want to avoid it (unless you know what you are doing).

5.4   Conditional execution

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. **Conditional statements** give us this ability. The simplest form is the `if` statement:

```
if x > 0:
    print('x is positive')
```

The boolean expression after `if` is called the **condition**. If it is true, the indented statement runs. If not, nothing happens.

`if` statements have the same structure as function definitions: a header followed by an indented body. Statements like this are called **compound statements**.

There is no limit on the number of statements that can appear in the body, but there has to be at least one. Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet). In that case, you can use the `pass` statement, which does nothing.

```
if x < 0:
    pass          # TODO: need to handle negative values!
```

5.5   Alternative execution

A second form of the `if` statement is "alternative execution", in which there are two possibilities and the condition determines which one runs. The syntax looks like this:

```
if x % 2 == 0:
    print('x is even')
else:
    print('x is odd')
```

If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays an appropriate message. If the condition is false, the second set of statements runs. Since the condition must be true or false, exactly one of the alternatives will run. The alternatives are called **branches**, because they are branches in the flow of execution.

5.6   Chained conditionals

Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

```
if x < y:
    print('x is less than y')
elif x > y:
    print('x is greater than y')
else:
    print('x and y are equal')
```

elif is an abbreviation of "else if". Again, exactly one branch will run. There is no limit on the number of elif statements. If there is an else clause, it has to be at the end, but there doesn't have to be one.

```
if choice == 'a':
    draw_a()
elif choice == 'b':
    draw_b()
elif choice == 'c':
    draw_c()
```

Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch runs and the statement ends. Even if more than one condition is true, only the first true branch runs.

5.7   Nested conditionals

One conditional can also be nested within another. We could have written the example in the previous section like this:

```
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')
```

The outer conditional contains two branches. The first branch contains a simple statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both simple statements, although they could have been conditional statements as well.

Although the indentation of the statements makes the structure apparent, **nested conditionals** become difficult to read very quickly. It is a good idea to avoid them when you can.

Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

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

The print statement runs only if we make it past both conditionals, so we can get the same effect with the and operator:

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

For this kind of condition, Python provides a more concise option:

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

5.8 Recursion

It is legal for one function to call another; it is also legal for a function to call itself. It may not be obvious why that is a good thing, but it turns out to be one of the most magical things a program can do. For example, look at the following function:

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

If n is 0 or negative, it outputs the word, "Blastoff!" Otherwise, it outputs n and then calls a function named countdown—itself—passing n-1 as an argument.

What happens if we call this function like this?

```
>>> countdown(3)
```

The execution of countdown begins with n=3, and since n is greater than 0, it outputs the value 3, and then calls itself...

> The execution of countdown begins with n=2, and since n is greater than 0, it outputs the value 2, and then calls itself...
>
> > The execution of countdown begins with n=1, and since n is greater than 0, it outputs the value 1, and then calls itself...
> >
> > > The execution of countdown begins with n=0, and since n is not greater than 0, it outputs the word, "Blastoff!" and then returns.
> >
> > The countdown that got n=1 returns.
>
> The countdown that got n=2 returns.

The countdown that got n=3 returns.

And then you're back in __main__. So, the total output looks like this:

```
3
2
1
Blastoff!
```

A function that calls itself is **recursive**; the process of executing it is called **recursion**.

As another example, we can write a function that prints a string n times.

```
def print_n(s, n):
    if n <= 0:
        return
    print(s)
    print_n(s, n-1)
```
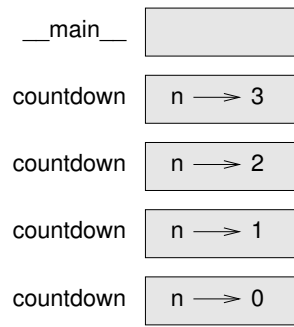
Figure 5.1: Stack diagram.

If `n <= 0` the **return statement** exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function don't run.

The rest of the function is similar to `countdown`: it displays `s` and then calls itself to display `s` $n-1$ additional times. So the number of lines of output is `1 + (n - 1)`, which adds up to `n`.

For simple examples like this, it is probably easier to use a `for` loop. But we will see examples later that are hard to write with a `for` loop and easy to write with recursion, so it is good to start early.

5.9   Stack diagrams for recursive functions

In Section 3.9, we used a stack diagram to represent the state of a program during a function call. The same kind of diagram can help interpret a recursive function.

Every time a function gets called, Python creates a frame to contain the function's local variables and parameters. For a recursive function, there might be more than one frame on the stack at the same time.

Figure 5.1 shows a stack diagram for `countdown` called with `n = 3`.

As usual, the top of the stack is the frame for `__main__`. It is empty because we did not create any variables in `__main__` or pass any arguments to it.

The four `countdown` frames have different values for the parameter n. The bottom of the stack, where `n=0`, is called the **base case**. It does not make a recursive call, so there are no more frames.

As an exercise, draw a stack diagram for `print_n` called with `s = 'Hello'` and `n=2`. Then write a function called `do_n` that takes a function object and a number, n, as arguments, and that calls the given function n times.

5.10   Infinite recursion

If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not a good idea. Here is a minimal program with an infinite recursion:

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

In most programming environments, a program with infinite recursion does not really run forever. Python reports an error message when the maximum recursion depth is reached:

```
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
  File "<stdin>", line 2, in recurse
                  .
                  .
                  .
  File "<stdin>", line 2, in recurse
RuntimeError: Maximum recursion depth exceeded
```

This traceback is a little bigger than the one we saw in the previous chapter. When the error occurs, there are 1000 `recurse` frames on the stack!

If you encounter an infinite recursion by accident, review your function to confirm that there is a base case that does not make a recursive call. And if there is a base case, check whether you are guaranteed to reach it.

5.11   Keyboard input

The programs we have written so far accept no input from the user. They just do the same thing every time.

Python provides a built-in function called `input` that stops the program and waits for the user to type something. When the user presses Return or Enter, the program resumes and `input` returns what the user typed as a string. In Python 2, the same function is called `raw_input`.

```
>>> text = input()
What are you waiting for?
>>> text
'What are you waiting for?'
```

Before getting input from the user, it is a good idea to print a prompt telling the user what to type. `input` can take a prompt as an argument:

```
>>> name = input('What...is your name?\n')
What...is your name?
Arthur, King of the Britons!
>>> name
'Arthur, King of the Britons!'
```

The sequence `\n` at the end of the prompt represents a **newline**, which is a special character that causes a line break. That's why the user's input appears below the prompt.

If you expect the user to type an integer, you can try to convert the return value to int:

```
>>> prompt = 'What...is the airspeed velocity of an unladen swallow?\n'
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
42
>>> int(speed)
42
```

But if the user types something other than a string of digits, you get an error:

```
>>> speed = input(prompt)
What...is the airspeed velocity of an unladen swallow?
What do you mean, an African or a European swallow?
>>> int(speed)
ValueError: invalid literal for int() with base 10
```

We will see how to handle this kind of error later.

5.12   Debugging

When a syntax or runtime error occurs, the error message contains a lot of information, but it can be overwhelming. The most useful parts are usually:

- What kind of error it was, and

- Where it occurred.

Syntax errors are usually easy to find, but there are a few gotchas. Whitespace errors can be tricky because spaces and tabs are invisible and we are used to ignoring them.

```
>>> x = 5
>>>  y = 6
  File "<stdin>", line 1
    y = 6
    ^
IndentationError: unexpected indent
```

In this example, the problem is that the second line is indented by one space. But the error message points to y, which is misleading. In general, error messages indicate where the problem was discovered, but the actual error might be earlier in the code, sometimes on a previous line.

The same is true of runtime errors. Suppose you are trying to compute a signal-to-noise ratio in decibels. The formula is $SNR_{db} = 10 \log_{10}(P_{signal}/P_{noise})$. In Python, you might write something like this:

```
import math
signal_power = 9
noise_power = 10
ratio = signal_power // noise_power
decibels = 10 * math.log10(ratio)
print(decibels)
```

When you run this program, you get an exception:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
    decibels = 10 * math.log10(ratio)
ValueError: math domain error
```

The error message indicates line 5, but there is nothing wrong with that line. To find the real error, it might be useful to print the value of `ratio`, which turns out to be 0. The problem is in line 4, which uses floor division instead of floating-point division.

You should take the time to read error messages carefully, but don't assume that everything they say is correct.

5.13 Glossary

floor division: An operator, denoted //, that divides two numbers and rounds down (toward negative infinity) to an integer.

modulus operator: An operator, denoted with a percent sign (%), that works on integers and returns the remainder when one number is divided by another.

boolean expression: An expression whose value is either `True` or `False`.

relational operator: One of the operators that compares its operands: ==, !=, >, <, >=, and <=.

logical operator: One of the operators that combines boolean expressions: `and`, `or`, and `not`.

conditional statement: A statement that controls the flow of execution depending on some condition.

condition: The boolean expression in a conditional statement that determines which branch runs.

compound statement: A statement that consists of a header and a body. The header ends with a colon (:). The body is indented relative to the header.

branch: One of the alternative sequences of statements in a conditional statement.

chained conditional: A conditional statement with a series of alternative branches.

nested conditional: A conditional statement that appears in one of the branches of another conditional statement.

return statement: A statement that causes a function to end immediately and return to the caller.

recursion: The process of calling the function that is currently executing.

base case: A conditional branch in a recursive function that does not make a recursive call.

infinite recursion: A recursion that doesn't have a base case, or never reaches it. Eventually, an infinite recursion causes a runtime error.

5.14 Exercises

Exercise 5.1. *The* `time` *module provides a function, also named* `time`*, that returns the current Greenwich Mean Time in "the epoch", which is an arbitrary time used as a reference point. On UNIX systems, the epoch is 1 January 1970.*

```
>>> import time
>>> time.time()
1437746094.5735958
```

Write a script that reads the current time and converts it to a time of day in hours, minutes, and seconds, plus the number of days since the epoch.

Exercise 5.2. *Fermat's Last Theorem says that there are no positive integers a, b, and c such that*

$$a^n + b^n = c^n$$

for any values of n greater than 2.

1. *Write a function named* `check_fermat` *that takes four parameters—a, b, c and n—and checks to see if Fermat's theorem holds. If n is greater than 2 and*

$$a^n + b^n = c^n$$

   the program should print, "Holy smokes, Fermat was wrong!" Otherwise the program should print, "No, that doesn't work."

2. *Write a function that prompts the user to input values for a, b, c and n, converts them to integers, and uses* `check_fermat` *to check whether they violate Fermat's theorem.*

Exercise 5.3. *If you are given three sticks, you may or may not be able to arrange them in a triangle. For example, if one of the sticks is 12 inches long and the other two are one inch long, you will not be able to get the short sticks to meet in the middle. For any three lengths, there is a simple test to see if it is possible to form a triangle:*

> *If any of the three lengths is greater than the sum of the other two, then you cannot form a triangle. Otherwise, you can. (If the sum of two lengths equals the third, they form what is called a "degenerate" triangle.)*

1. *Write a function named* `is_triangle` *that takes three integers as arguments, and that prints either "Yes" or "No", depending on whether you can or cannot form a triangle from sticks with the given lengths.*

2. *Write a function that prompts the user to input three stick lengths, converts them to integers, and uses* `is_triangle` *to check whether sticks with the given lengths can form a triangle.*

Exercise 5.4. *What is the output of the following program? Draw a stack diagram that shows the state of the program when it prints the result.*

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

recurse(3, 0)
```

1. *What would happen if you called this function like this:* `recurse(-1, 0)`*?*

2. *Write a docstring that explains everything someone would need to know in order to use this function (and nothing else).*

The following exercises use the `turtle` module, described in Chapter 4:
Exercise 5.5. *Read the following function and see if you can figure out what it does (see the examples in Chapter 4). Then run it and see if you got it right.*
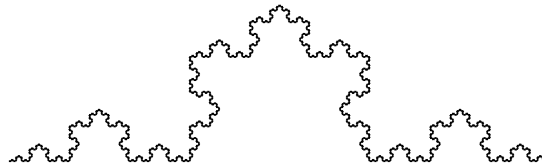
Figure 5.2: A Koch curve.

```
def draw(t, length, n):
    if n == 0:
        return
    angle = 50
    t.fd(length*n)
    t.lt(angle)
    draw(t, length, n-1)
    t.rt(2*angle)
    draw(t, length, n-1)
    t.lt(angle)
    t.bk(length*n)
```

Exercise 5.6. *The Koch curve is a fractal that looks something like Figure 5.2. To draw a Koch curve with length x, all you have to do is*

1. *Draw a Koch curve with length x/3.*

2. *Turn left 60 degrees.*

3. *Draw a Koch curve with length x/3.*

4. *Turn right 120 degrees.*

5. *Draw a Koch curve with length x/3.*

6. *Turn left 60 degrees.*

7. *Draw a Koch curve with length x/3.*

The exception is if x is less than 3: in that case, you can just draw a straight line with length x.

1. *Write a function called* `koch` *that takes a turtle and a length as parameters, and that uses the turtle to draw a Koch curve with the given length.*

2. *Write a function called* `snowflake` *that draws three Koch curves to make the outline of a snowflake.*

   Solution: `http://thinkpython2.com/code/koch.py`.

3. *The Koch curve can be generalized in several ways. See* `http://en.wikipedia.org/wiki/Koch_snowflake` *for examples and implement your favorite.*