

Chapter 5

Conditionals and recursion

5.1 Modulus operator

The **modulus operator** works on integers and yields the remainder when the first operand is divided by the second. In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

```
>>> quotient = 7 / 3
>>> print quotient
2
>>> remainder = 7 % 3
>>> print remainder
1
```

So 7 divided by 3 is 2 with 1 left over.

The modulus operator turns out to be surprisingly useful. 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.

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)
<type 'bool'>
>>> type(False)
<type '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* 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.”

```
>>> 17 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, then the indented statement gets executed. 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          # 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 gets executed. 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 a message to that effect. If the condition is false, the second set of statements is executed. Since the condition must be true or false, exactly one of the alternatives will be executed. 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 be executed. 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 executes, and the statement ends. Even if more than one condition is true, only the first true branch executes.

5.7 Nested conditionals

One conditional can also be nested within another. We could have written the trichotomy example 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. In general, 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 is executed 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.'

```

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 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)
```

If $n \leq 0$ the return statement exits the function. The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed.

The rest of the function is similar to countdown: if n is greater than 0, 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 3.10, 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 new function frame, which contains 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.

Exercise 5.1. Draw a stack diagram for `print_n` called with $s = \text{'Hello'}$ and $n=2$.

Exercise 5.2. 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.

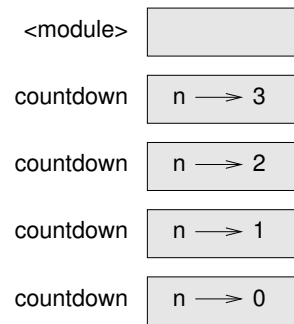


Figure 5.1: Stack diagram.

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!

5.11 Keyboard input

The programs we have written so far are a bit rude in the sense that they accept no input from the user. They just do the same thing every time.

Python 2 provides a built-in function called `raw_input` that gets input from the keyboard. In Python 3, it is called `input`. When this function is called, the program stops and waits for the user to type something. When the user presses Return or Enter, the program resumes and `raw_input` returns what the user typed as a string.

```
>>> text = raw_input()
What are you waiting for?
>>> print 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 input. `raw_input` can take a prompt as an argument:

```
>>> name = raw_input('What...is your name?\n')
What...is your name?
Arthur, King of the Britons!
>>> print 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 = raw_input(prompt)
What...is the airspeed velocity of an unladen swallow?
17
>>> int(speed)
17
```

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

```
>>> speed = raw_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

The traceback Python displays when an error occurs contains a lot of information, but it can be overwhelming, especially when there are many frames on the stack. 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
```

But when you run it in Python 2, you get an error message.

```
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, because dividing two integers does floor division. The solution is to represent signal power and noise power with floating-point values.

In general, error messages tell you where the problem was discovered, but that is often not where it was caused.

In Python 3, this example does not cause an error; the division operator performs floating-point division even with integer operands.

5.13 Glossary

modulus operator: An operator, denoted with a percent sign (%), that works on integers and yields 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 is executed.

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.

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 5.3. *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 that checks to see if Fermat's theorem holds. If n is greater than 2 and it turns out to be true that

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

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 5.4. *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, it is clear that 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.

The following exercises use TurtleWorld from Chapter 4:

Exercise 5.5. *Read the following function and see if you can figure out what it does. Then run it (see the examples in Chapter 4).*

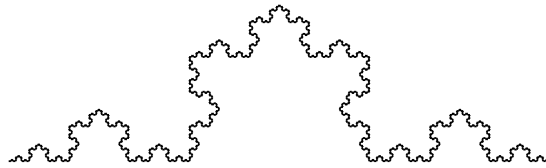


Figure 5.2: A Koch curve.

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

Chapter 6

Fruitful functions

6.1 Return values

Some of the built-in functions we have used, such as the math functions, produce results. Calling the function generates a value, which we usually assign to a variable or use as part of an expression.

```
e = math.exp(1.0)
height = radius * math.sin(radians)
```

All of the functions we have written so far are void; they print something or move turtles around, but their return value is `None`.

In this chapter, we are (finally) going to write fruitful functions. The first example is `area`, which returns the area of a circle with the given radius:

```
def area(radius):
    temp = math.pi * radius**2
    return temp
```

We have seen the `return` statement before, but in a fruitful function the `return` statement includes an expression. This statement means: “Return immediately from this function and use the following expression as a return value.” The expression can be arbitrarily complicated, so we could have written this function more concisely:

```
def area(radius):
    return math.pi * radius**2
```

On the other hand, **temporary variables** like `temp` often make debugging easier.

Sometimes it is useful to have multiple return statements, one in each branch of a conditional:

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

Since these return statements are in an alternative conditional, only one will be executed.

As soon as a return statement executes, the function terminates without executing any subsequent statements. Code that appears after a return statement, or any other place the flow of execution can never reach, is called **dead code**.

In a fruitful function, it is a good idea to ensure that every possible path through the program hits a return statement. For example:

```
def absolute_value(x):
    if x < 0:
        return -x
    if x > 0:
        return x
```

This function is incorrect because if x happens to be 0, neither condition is true, and the function ends without hitting a return statement. If the flow of execution gets to the end of a function, the return value is `None`, which is not the absolute value of 0.

```
>>> print absolute_value(0)
None
```

By the way, Python provides a built-in function called `abs` that computes absolute values.

Exercise 6.1. Write a compare function that returns 1 if $x > y$, 0 if $x == y$, and -1 if $x < y$.

6.2 Incremental development

As you write larger functions, you might find yourself spending more time debugging.

To deal with increasingly complex programs, you might want to try a process called **incremental development**. The goal of incremental development is to avoid long debugging sessions by adding and testing only a small amount of code at a time.

As an example, suppose you want to find the distance between two points, given by the coordinates (x_1, y_1) and (x_2, y_2) . By the Pythagorean theorem, the distance is:

$$\text{distance} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

The first step is to consider what a distance function should look like in Python. In other words, what are the inputs (parameters) and what is the output (return value)?

In this case, the inputs are two points, which you can represent using four numbers. The return value is the distance, which is a floating-point value.

Already you can write an outline of the function:

```
def distance(x1, y1, x2, y2):
    return 0.0
```

Obviously, this version doesn't compute distances; it always returns zero. But it is syntactically correct, and it runs, which means that you can test it before you make it more complicated.

To test the new function, call it with sample arguments:

```
>>> distance(1, 2, 4, 6)
0.0
```

I chose these values so that the horizontal distance is 3 and the vertical distance is 4; that way, the result is 5 (the hypotenuse of a 3-4-5 triangle). When testing a function, it is useful to know the right answer.

At this point we have confirmed that the function is syntactically correct, and we can start adding code to the body. A reasonable next step is to find the differences $x_2 - x_1$ and $y_2 - y_1$. The next version stores those values in temporary variables and prints them.

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    print 'dx is', dx
    print 'dy is', dy
    return 0.0
```

If the function is working, it should display 'dx is 3' and 'dy is 4'. If so, we know that the function is getting the right arguments and performing the first computation correctly. If not, there are only a few lines to check.

Next we compute the sum of squares of dx and dy:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    print 'dsquared is: ', dsquared
    return 0.0
```

Again, you would run the program at this stage and check the output (which should be 25). Finally, you can use `math.sqrt` to compute and return the result:

```
def distance(x1, y1, x2, y2):
    dx = x2 - x1
    dy = y2 - y1
    dsquared = dx**2 + dy**2
    result = math.sqrt(dsquared)
    return result
```

If that works correctly, you are done. Otherwise, you might want to print the value of `result` before the `return` statement.

The final version of the function doesn't display anything when it runs; it only returns a value. The `print` statements we wrote are useful for debugging, but once you get the function working, you should remove them. Code like that is called **scaffolding** because it is helpful for building the program but is not part of the final product.

When you start out, you should add only a line or two of code at a time. As you gain more experience, you might find yourself writing and debugging bigger chunks. Either way, incremental development can save you a lot of debugging time.

The key aspects of the process are:

1. Start with a working program and make small incremental changes. At any point, if there is an error, you should have a good idea where it is.

2. Use temporary variables to hold intermediate values so you can display and check them.
3. Once the program is working, you might want to remove some of the scaffolding or consolidate multiple statements into compound expressions, but only if it does not make the program difficult to read.

Exercise 6.2. Use incremental development to write a function called `hypotenuse` that returns the length of the hypotenuse of a right triangle given the lengths of the two legs as arguments. Record each stage of the development process as you go.

6.3 Composition

As you should expect by now, you can call one function from within another. This ability is called **composition**.

As an example, we'll write a function that takes two points, the center of the circle and a point on the perimeter, and computes the area of the circle.

Assume that the center point is stored in the variables `xc` and `yc`, and the perimeter point is in `xp` and `yp`. The first step is to find the radius of the circle, which is the distance between the two points. We just wrote a function, `distance`, that does that:

```
radius = distance(xc, yc, xp, yp)
```

The next step is to find the area of a circle with that radius; we just wrote that, too:

```
result = area(radius)
```

Encapsulating these steps in a function, we get:

```
def circle_area(xc, yc, xp, yp):
    radius = distance(xc, yc, xp, yp)
    result = area(radius)
    return result
```

The temporary variables `radius` and `result` are useful for development and debugging, but once the program is working, we can make it more concise by composing the function calls:

```
def circle_area(xc, yc, xp, yp):
    return area(distance(xc, yc, xp, yp))
```

6.4 Boolean functions

Functions can return booleans, which is often convenient for hiding complicated tests inside functions. For example:

```
def is_divisible(x, y):
    if x % y == 0:
        return True
    else:
        return False
```

It is common to give boolean functions names that sound like yes/no questions; `is_divisible` returns either `True` or `False` to indicate whether `x` is divisible by `y`.

Here is an example:

```
>>> is_divisible(6, 4)
False
>>> is_divisible(6, 3)
True
```

The result of the `==` operator is a boolean, so we can write the function more concisely by returning it directly:

```
def is_divisible(x, y):
    return x % y == 0
```

Boolean functions are often used in conditional statements:

```
if is_divisible(x, y):
    print 'x is divisible by y'
```

It might be tempting to write something like:

```
if is_divisible(x, y) == True:
    print 'x is divisible by y'
```

But the extra comparison is unnecessary.

Exercise 6.3. Write a function `is_between(x, y, z)` that returns `True` if $x \leq y \leq z$ or `False` otherwise.

6.5 More recursion

We have only covered a small subset of Python, but you might be interested to know that this subset is a *complete* programming language, which means that anything that can be computed can be expressed in this language. Any program ever written could be rewritten using only the language features you have learned so far (actually, you would need a few commands to control devices like the keyboard, mouse, disks, etc., but that's all).

Proving that claim is a nontrivial exercise first accomplished by Alan Turing, one of the first computer scientists (some would argue that he was a mathematician, but a lot of early computer scientists started as mathematicians). Accordingly, it is known as the Turing Thesis. For a more complete (and accurate) discussion of the Turing Thesis, I recommend Michael Sipser's book *Introduction to the Theory of Computation*.

To give you an idea of what you can do with the tools you have learned so far, we'll evaluate a few recursively defined mathematical functions. A recursive definition is similar to a circular definition, in the sense that the definition contains a reference to the thing being defined. A truly circular definition is not very useful:

vorpal: An adjective used to describe something that is vorpal.

If you saw that definition in the dictionary, you might be annoyed. On the other hand, if you looked up the definition of the factorial function, denoted with the symbol $!$, you might get something like this:

$$\begin{aligned} 0! &= 1 \\ n! &= n(n-1)! \end{aligned}$$

This definition says that the factorial of 0 is 1, and the factorial of any other value, n , is n multiplied by the factorial of $n - 1$.

So $3!$ is 3 times $2!$, which is 2 times $1!$, which is 1 times $0!$. Putting it all together, $3!$ equals 3 times 2 times 1 times 1, which is 6.

If you can write a recursive definition of something, you can usually write a Python program to evaluate it. The first step is to decide what the parameters should be. In this case it should be clear that `factorial` takes an integer:

```
def factorial(n):
```

If the argument happens to be 0, all we have to do is return 1:

```
def factorial(n):
    if n == 0:
        return 1
```

Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of $n - 1$ and then multiply it by n :

```
def factorial(n):
    if n == 0:
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        return result
```

The flow of execution for this program is similar to the flow of `countdown` in Section 5.8. If we call `factorial` with the value 3:

Since 3 is not 0, we take the second branch and calculate the factorial of $n-1$...

Since 2 is not 0, we take the second branch and calculate the factorial of $n-1$...

Since 1 is not 0, we take the second branch and calculate the factorial of $n-1$...

Since 0 is 0, we take the first branch and return 1 without making any more recursive calls.

The return value (1) is multiplied by n , which is 1, and the result is returned.

The return value (1) is multiplied by n , which is 2, and the result is returned.

The return value (2) is multiplied by n , which is 3, and the result, 6, becomes the return value of the function call that started the whole process.

Figure 6.1 shows what the stack diagram looks like for this sequence of function calls.

The return values are shown being passed back up the stack. In each frame, the return value is the value of `result`, which is the product of `n` and `recurse`.

In the last frame, the local variables `recurse` and `result` do not exist, because the branch that creates them does not execute.

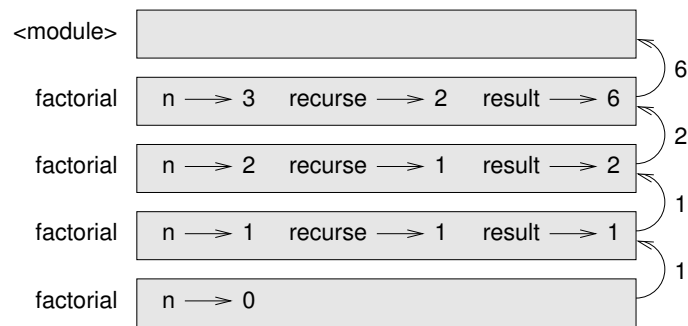


Figure 6.1: Stack diagram.

6.6 Leap of faith

Following the flow of execution is one way to read programs, but it can quickly become labyrinthine. An alternative is what I call the “leap of faith.” When you come to a function call, instead of following the flow of execution, you *assume* that the function works correctly and returns the right result.

In fact, you are already practicing this leap of faith when you use built-in functions. When you call `math.cos` or `math.exp`, you don’t examine the bodies of those functions. You just assume that they work because the people who wrote the built-in functions were good programmers.

The same is true when you call one of your own functions. For example, in Section 6.4, we wrote a function called `is_divisible` that determines whether one number is divisible by another. Once we have convinced ourselves that this function is correct—by examining the code and testing—we can use the function without looking at the body again.

The same is true of recursive programs. When you get to the recursive call, instead of following the flow of execution, you should assume that the recursive call works (yields the correct result) and then ask yourself, “Assuming that I can find the factorial of $n - 1$, can I compute the factorial of n ?” In this case, it is clear that you can, by multiplying by n .

Of course, it’s a bit strange to assume that the function works correctly when you haven’t finished writing it, but that’s why it’s called a leap of faith!

6.7 One more example

After `factorial`, the most common example of a recursively defined mathematical function is `fibonacci`, which has the following definition (see http://en.wikipedia.org/wiki/Fibonacci_number):

$$\begin{aligned}\text{fibonacci}(0) &= 0 \\ \text{fibonacci}(1) &= 1 \\ \text{fibonacci}(n) &= \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)\end{aligned}$$

Translated into Python, it looks like this:

```
def fibonacci (n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

If you try to follow the flow of execution here, even for fairly small values of n , your head explodes. But according to the leap of faith, if you assume that the two recursive calls work correctly, then it is clear that you get the right result by adding them together.

6.8 Checking types

What happens if we call `factorial` and give it 1.5 as an argument?

```
>>> factorial(1.5)
RuntimeError: Maximum recursion depth exceeded
```

It looks like an infinite recursion. But how can that be? There is a base case—when `n == 0`. But if `n` is not an integer, we can *miss* the base case and recurse forever.

In the first recursive call, the value of `n` is 0.5. In the next, it is -0.5. From there, it gets smaller (more negative), but it will never be 0.

We have two choices. We can try to generalize the `factorial` function to work with floating-point numbers, or we can make `factorial` check the type of its argument. The first option is called the gamma function and it's a little beyond the scope of this book. So we'll go for the second.

We can use the built-in function `isinstance` to verify the type of the argument. While we're at it, we can also make sure the argument is positive:

```
def factorial (n):
    if not isinstance(n, int):
        print 'Factorial is only defined for integers.'
        return None
    elif n < 0:
        print 'Factorial is not defined for negative integers.'
        return None
    elif n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

The first base case handles nonintegers; the second catches negative integers. In both cases, the program prints an error message and returns `None` to indicate that something went wrong:

```
>>> factorial('fred')
Factorial is only defined for integers.
None
>>> factorial(-2)
Factorial is not defined for negative integers.
None
```

If we get past both checks, then we know that n is positive or zero, so we can prove that the recursion terminates.

This program demonstrates a pattern sometimes called a **guardian**. The first two conditionals act as guardians, protecting the code that follows from values that might cause an error. The guardians make it possible to prove the correctness of the code.

In Section 11.3 we will see a more flexible alternative to printing an error message: raising an exception.

6.9 Debugging

Breaking a large program into smaller functions creates natural checkpoints for debugging. If a function is not working, there are three possibilities to consider:

- There is something wrong with the arguments the function is getting; a precondition is violated.
- There is something wrong with the function; a postcondition is violated.
- There is something wrong with the return value or the way it is being used.

To rule out the first possibility, you can add a `print` statement at the beginning of the function and display the values of the parameters (and maybe their types). Or you can write code that checks the preconditions explicitly.

If the parameters look good, add a `print` statement before each `return` statement that displays the return value. If possible, check the result by hand. Consider calling the function with values that make it easy to check the result (as in Section 6.2).

If the function seems to be working, look at the function call to make sure the return value is being used correctly (or used at all!).

Adding `print` statements at the beginning and end of a function can help make the flow of execution more visible. For example, here is a version of `factorial` with `print` statements:

```
def factorial(n):
    space = ' ' * (4 * n)
    print space, 'factorial', n
    if n == 0:
        print space, 'returning 1'
        return 1
    else:
        recurse = factorial(n-1)
        result = n * recurse
        print space, 'returning', result
        return result
```

`space` is a string of space characters that controls the indentation of the output. Here is the result of `factorial(5)`:

```

                factorial 5
            factorial 4
        factorial 3
    factorial 2
factorial 1
factorial 0
returning 1
    returning 1
        returning 2
            returning 6
                returning 24
                    returning 120

```

If you are confused about the flow of execution, this kind of output can be helpful. It takes some time to develop effective scaffolding, but a little bit of scaffolding can save a lot of debugging.

6.10 Glossary

temporary variable: A variable used to store an intermediate value in a complex calculation.

dead code: Part of a program that can never be executed, often because it appears after a return statement.

None: A special value returned by functions that have no return statement or a return statement without an argument.

incremental development: A program development plan intended to avoid debugging by adding and testing only a small amount of code at a time.

scaffolding: Code that is used during program development but is not part of the final version.

guardian: A programming pattern that uses a conditional statement to check for and handle circumstances that might cause an error.

6.11 Exercises

Exercise 6.4. Draw a stack diagram for the following program. What does the program print?

Solution: http://thinkpython.com/code/stack_diagram.py.

```

def b(z):
    prod = a(z, z)
    print z, prod
    return prod

```

```

def a(x, y):
    x = x + 1
    return x * y

```

```
def c(x, y, z):
    total = x + y + z
    square = b(total)**2
    return square
```

```
x = 1
y = x + 1
print c(x, y+3, x+y)
```

Exercise 6.5. The Ackermann function, $A(m, n)$, is defined:

$$A(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ A(m - 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m - 1, A(m, n - 1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

See http://en.wikipedia.org/wiki/Ackermann_function. Write a function named `ack` that evaluates Ackermann's function. Use your function to evaluate `ack(3, 4)`, which should be 125. What happens for larger values of `m` and `n`? Solution: <http://thinkpython.com/code/ackermann.py>.

Exercise 6.6. A palindrome is a word that is spelled the same backward and forward, like “noon” and “redivider”. Recursively, a word is a palindrome if the first and last letters are the same and the middle is a palindrome.

The following are functions that take a string argument and return the first, last, and middle letters:

```
def first(word):
    return word[0]

def last(word):
    return word[-1]

def middle(word):
    return word[1:-1]
```

We'll see how they work in Chapter 8.

1. Type these functions into a file named `palindrome.py` and test them out. What happens if you call `middle` with a string with two letters? One letter? What about the empty string, which is written `''` and contains no letters?
2. Write a function called `is_palindrome` that takes a string argument and returns `True` if it is a palindrome and `False` otherwise. Remember that you can use the built-in function `len` to check the length of a string.

Solution: http://thinkpython.com/code/palindrome_soln.py.

Exercise 6.7. A number, a , is a power of b if it is divisible by b and a/b is a power of b . Write a function called `is_power` that takes parameters `a` and `b` and returns `True` if `a` is a power of `b`. Note: you will have to think about the base case.

Exercise 6.8. The greatest common divisor (GCD) of a and b is the largest number that divides both of them with no remainder.

One way to find the GCD of two numbers is based on the observation that if r is the remainder when a is divided by b , then $\text{gcd}(a, b) = \text{gcd}(b, r)$. As a base case, we can use $\text{gcd}(a, 0) = a$.

Write a function called `gcd` that takes parameters `a` and `b` and returns their greatest common divisor.

*Credit: This exercise is based on an example from Abelson and Sussman's *Structure and Interpretation of Computer Programs*.*

Chapter 7

Iteration

7.1 Multiple assignment

As you may have discovered, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

```
bruce = 5
print bruce,
bruce = 7
print bruce
```

The output of this program is 5 7, because the first time `bruce` is printed, its value is 5, and the second time, its value is 7. The comma at the end of the first `print` statement suppresses the newline, which is why both outputs appear on the same line.

Figure 7.1 shows what **multiple assignment** looks like in a state diagram.

With multiple assignment it is especially important to distinguish between an assignment operation and a statement of equality. Because Python uses the equal sign (`=`) for assignment, it is tempting to interpret a statement like `a = b` as a statement of equality. It is not!

First, equality is a symmetric relation and assignment is not. For example, in mathematics, if $a = 7$ then $7 = a$. But in Python, the statement `a = 7` is legal and `7 = a` is not.

Furthermore, in mathematics, a statement of equality is either true or false, for all time. If $a = b$ now, then a will always equal b . In Python, an assignment statement can make two variables equal, but they don't have to stay that way:

```
a = 5
b = a    # a and b are now equal
a = 3    # a and b are no longer equal
```

The third line changes the value of `a` but does not change the value of `b`, so they are no longer equal.

Although multiple assignment is frequently helpful, you should use it with caution. If the values of variables change frequently, it can make the code difficult to read and debug.

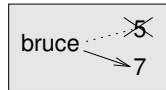


Figure 7.1: State diagram.

7.2 Updating variables

One of the most common forms of multiple assignment is an **update**, where the new value of the variable depends on the old.

```
x = x+1
```

This means “get the current value of *x*, add one, and then update *x* with the new value.”

If you try to update a variable that doesn’t exist, you get an error, because Python evaluates the right side before it assigns a value to *x*:

```
>>> x = x+1
NameError: name 'x' is not defined
```

Before you can update a variable, you have to **initialize** it, usually with a simple assignment:

```
>>> x = 0
>>> x = x+1
```

Updating a variable by adding 1 is called an **increment**; subtracting 1 is called a **decrement**.

7.3 The while statement

Computers are often used to automate repetitive tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

We have seen two programs, `countdown` and `print_n`, that use recursion to perform repetition, which is also called **iteration**. Because iteration is so common, Python provides several language features to make it easier. One is the `for` statement we saw in Section 4.2. We’ll get back to that later.

Another is the `while` statement. Here is a version of `countdown` that uses a `while` statement:

```
def countdown(n):
    while n > 0:
        print n
        n = n-1
    print 'Blastoff!'
```

You can almost read the `while` statement as if it were English. It means, “While *n* is greater than 0, display the value of *n* and then reduce the value of *n* by 1. When you get to 0, display the word `Blastoff!`”

More formally, here is the flow of execution for a `while` statement:

1. Evaluate the condition, yielding `True` or `False`.

2. If the condition is false, exit the `while` statement and continue execution at the next statement.
3. If the condition is true, execute the body and then go back to step 1.

This type of flow is called a **loop** because the third step loops back around to the top.

The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, “Lather, rinse, repeat,” are an infinite loop.

In the case of `countdown`, we can prove that the loop terminates because we know that the value of `n` is finite, and we can see that the value of `n` gets smaller each time through the loop, so eventually we have to get to 0. In other cases, it is not so easy to tell:

```
def sequence(n):
    while n != 1:
        print n,
        if n%2 == 0:          # n is even
            n = n/2
        else:                 # n is odd
            n = n*3+1
```

The condition for this loop is `n != 1`, so the loop will continue until `n` is 1, which makes the condition false.

Each time through the loop, the program outputs the value of `n` and then checks whether it is even or odd. If it is even, `n` is divided by 2. If it is odd, the value of `n` is replaced with `n*3+1`. For example, if the argument passed to `sequence` is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.

Since `n` sometimes increases and sometimes decreases, there is no obvious proof that `n` will ever reach 1, or that the program terminates. For some particular values of `n`, we can prove termination. For example, if the starting value is a power of two, then the value of `n` will be even each time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.

The hard question is whether we can prove that this program terminates for *all positive values* of `n`. So far, no one has been able to prove it *or* disprove it! (See http://en.wikipedia.org/wiki/Collatz_conjecture.)

Exercise 7.1. Rewrite the function `print_n` from Section 5.8 using iteration instead of recursion.

7.4 break

Sometimes you don’t know it’s time to end a loop until you get half way through the body. In that case you can use the `break` statement to jump out of the loop.

For example, suppose you want to take input from the user until they type `done`. You could write:

```
while True:
    line = raw_input('> ')
    if line == 'done':
        break
    print line

print 'Done!'
```

The loop condition is `True`, which is always true, so the loop runs until it hits the `break` statement.

Each time through, it prompts the user with an angle bracket. If the user types `done`, the `break` statement exits the loop. Otherwise the program echoes whatever the user types and goes back to the top of the loop. Here's a sample run:

```
> not done
not done
> done
Done!
```

This way of writing `while` loops is common because you can check the condition anywhere in the loop (not just at the top) and you can express the stop condition affirmatively ("stop when this happens") rather than negatively ("keep going until that happens.").

7.5 Square roots

Loops are often used in programs that compute numerical results by starting with an approximate answer and iteratively improving it.

For example, one way of computing square roots is Newton's method. Suppose that you want to know the square root of a . If you start with almost any estimate, x , you can compute a better estimate with the following formula:

$$y = \frac{x + a/x}{2}$$

For example, if a is 4 and x is 3:

```
>>> a = 4.0
>>> x = 3.0
>>> y = (x + a/x) / 2
>>> print y
2.16666666667
```

Which is closer to the correct answer ($\sqrt{4} = 2$). If we repeat the process with the new estimate, it gets even closer:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00641025641
```

After a few more updates, the estimate is almost exact:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00001024003
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.00000000003
```

In general we don't know ahead of time how many steps it takes to get to the right answer, but we know when we get there because the estimate stops changing:

```
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.0
>>> x = y
>>> y = (x + a/x) / 2
>>> print y
2.0
```

When `y == x`, we can stop. Here is a loop that starts with an initial estimate, `x`, and improves it until it stops changing:

```
while True:
    print x
    y = (x + a/x) / 2
    if y == x:
        break
    x = y
```

For most values of `a` this works fine, but in general it is dangerous to test float equality. Floating-point values are only approximately right: most rational numbers, like $1/3$, and irrational numbers, like $\sqrt{2}$, can't be represented exactly with a `float`.

Rather than checking whether `x` and `y` are exactly equal, it is safer to use the built-in function `abs` to compute the absolute value, or magnitude, of the difference between them:

```
if abs(y-x) < epsilon:
    break
```

Where `epsilon` has a value like `0.0000001` that determines how close is close enough.

Exercise 7.2. *Encapsulate this loop in a function called `square_root` that takes `a` as a parameter, chooses a reasonable value of `x`, and returns an estimate of the square root of `a`.*

7.6 Algorithms

Newton's method is an example of an **algorithm**: it is a mechanical process for solving a category of problems (in this case, computing square roots).

It is not easy to define an algorithm. It might help to start with something that is not an algorithm. When you learned to multiply single-digit numbers, you probably memorized the multiplication table. In effect, you memorized 100 specific solutions. That kind of knowledge is not algorithmic.

But if you were “lazy,” you probably cheated by learning a few tricks. For example, to find the product of n and 9, you can write $n - 1$ as the first digit and $10 - n$ as the second digit. This trick is a general solution for multiplying any single-digit number by 9. That’s an algorithm!

Similarly, the techniques you learned for addition with carrying, subtraction with borrowing, and long division are all algorithms. One of the characteristics of algorithms is that they do not require any intelligence to carry out. They are mechanical processes in which each step follows from the last according to a simple set of rules.

In my opinion, it is embarrassing that humans spend so much time in school learning to execute algorithms that, quite literally, require no intelligence.

On the other hand, the process of designing algorithms is interesting, intellectually challenging, and a central part of what we call programming.

Some of the things that people do naturally, without difficulty or conscious thought, are the hardest to express algorithmically. Understanding natural language is a good example. We all do it, but so far no one has been able to explain *how* we do it, at least not in the form of an algorithm.

7.7 Debugging

As you start writing bigger programs, you might find yourself spending more time debugging. More code means more chances to make an error and more place for bugs to hide.

One way to cut your debugging time is “debugging by bisection.” For example, if there are 100 lines in your program and you check them one at a time, it would take 100 steps.

Instead, try to break the problem in half. Look at the middle of the program, or near it, for an intermediate value you can check. Add a `print` statement (or something else that has a verifiable effect) and run the program.

If the mid-point check is incorrect, there must be a problem in the first half of the program. If it is correct, the problem is in the second half.

Every time you perform a check like this, you halve the number of lines you have to search. After six steps (which is fewer than 100), you would be down to one or two lines of code, at least in theory.

In practice it is not always clear what the “middle of the program” is and not always possible to check it. It doesn’t make sense to count lines and find the exact midpoint. Instead, think about places in the program where there might be errors and places where it is easy to put a check. Then choose a spot where you think the chances are about the same that the bug is before or after the check.

7.8 Glossary

multiple assignment: Making more than one assignment to the same variable during the execution of a program.

update: An assignment where the new value of the variable depends on the old.

initialization: An assignment that gives an initial value to a variable that will be updated.

increment: An update that increases the value of a variable (often by one).

decrement: An update that decreases the value of a variable.

iteration: Repeated execution of a set of statements using either a recursive function call or a loop.

infinite loop: A loop in which the terminating condition is never satisfied.

7.9 Exercises

Exercise 7.3. To test the square root algorithm in this chapter, you could compare it with `math.sqrt`. Write a function named `test_square_root` that prints a table like this:

1.0	1.0	1.0	0.0
2.0	1.41421356237	1.41421356237	2.22044604925e-16
3.0	1.73205080757	1.73205080757	0.0
4.0	2.0	2.0	0.0
5.0	2.2360679775	2.2360679775	0.0
6.0	2.44948974278	2.44948974278	0.0
7.0	2.64575131106	2.64575131106	0.0
8.0	2.82842712475	2.82842712475	4.4408920985e-16
9.0	3.0	3.0	0.0

The first column is a number, a ; the second column is the square root of a computed with the function from Section 7.5; the third column is the square root computed by `math.sqrt`; the fourth column is the absolute value of the difference between the two estimates.

Exercise 7.4. The built-in function `eval` takes a string and evaluates it using the Python interpreter. For example:

```
>>> eval('1 + 2 * 3')
7
>>> import math
>>> eval('math.sqrt(5)')
2.2360679774997898
>>> eval('type(math.pi)')
<type 'float'>
```

Write a function called `eval_loop` that iteratively prompts the user, takes the resulting input and evaluates it using `eval`, and prints the result.

It should continue until the user enters 'done', and then return the value of the last expression it evaluated.

Exercise 7.5. The mathematician Srinivasa Ramanujan found an infinite series that can be used to generate a numerical approximation of $1/\pi$:

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

Write a function called `estimate_pi` that uses this formula to compute and return an estimate of π . It should use a `while` loop to compute terms of the summation until the last term is smaller than `1e-15` (which is Python notation for 10^{-15}). You can check the result by comparing it to `math.pi`.

Solution: <http://thinkpython.com/code/pi.py>.