



[Buy this book at Amazon.com](#)

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:

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

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

Are you using one of our books in a class?

We'd like to know about it. Please consider filling out this short survey.

amazon



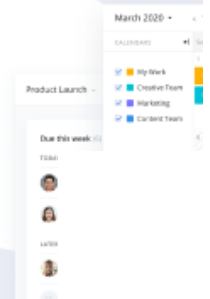
Python for Software...

\$44.80

Shop now

wrike

Try more than just a calendar



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 the if statement 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 .

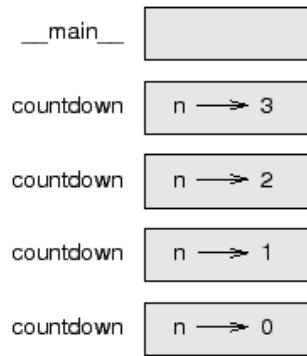
For simple examples like this, it is probably easier to use a `for` loop. But we will see examples later that are hard to write with a `for` loop and easy to write with recursion, so it is good to start early.

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.

This figure 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.

Draw a stack diagram for `print_n` called with `s = 'Hello'` and `n=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.

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 provides a built-in function called `raw_input` that gets input from the keyboard¹. 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.

```
>>> input = raw_input()
What are you waiting for?
>>> print input
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()
```

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
        ^
SyntaxError: invalid syntax
```

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, you get an error message²:

```
Traceback (most recent call last):
  File "snr.py", line 5, in ?
```

```
decibels = 10 * math.log10(ratio)
OverflowError: math range 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.

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 1

Fermat's Last Theorem says that there are no 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 2

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 can³.”

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 3

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

```
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 4

The Koch curve is a fractal that looks something like this:



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 only 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. You can see my solution at thinkpython.com/code/koch.py.
3. The Koch curve can be generalized in several ways. See wikipedia.org/wiki/Koch_snowflake for examples and implement your favorite.

1

In Python 3.0, this function is named `input`.

2

In Python 3.0, you no longer get an error message; the division operator performs floating-point division even with integer operands.

3

If the sum of two lengths equals the third, they form what is called a “degenerate” triangle.

