Beginning\_Python\_Magnus\_05

CHAPTER 5

Conditionals, Loops, and Some

Other Statements

By now, I’m sure you are getting a bit impatient. All right—all these data types are just dandy, but you can’t

really do much with them, can you?

Let’s crank up the pace a bit. We’ve already encountered a few statement types (print statements,

import statements, and assignments). Let’s first take a look at some more ways of using these before

diving into the world of conditionals and loops. Then we’ll see how list comprehensions work almost like

conditionals and loops, even though they are expressions, and finally we’ll take a look at pass, del, and exec.

More About print and import

As you learn more about Python, you may notice that some aspects of Python that you thought you knew

have hidden features just waiting to pleasantly surprise you. Let’s take a look at a couple of such nice features

in print and import. Though print is really a function, it used to be a statement type of its own, which is

why I’m discussing it here.

■■Tip For many applications, logging (using the logging module) will be more appropriate than using print.

See Chapter 19 for more details.

Printing Multiple Arguments

You’ve seen how print can be used to print an expression, which is either a string or automatically

converted to one. But you can actually print more than one expression, as long as you separate them with

commas:

>>> print('Age:', 42)

Age: 42

As you can see, a space character is inserted between each argument. This behavior can be very useful if you

want to combine text and variable values without using the full power of string formatting.

>>> name = 'Gumby'

>>> salutation = 'Mr.'

>>> greeting = 'Hello,'

>>> print(greeting, salutation, name)

Hello, Mr. Gumby

If the greeting string had no comma, how would you get the comma in the result? You couldn’t just use

print(greeting, ',', salutation, name)

because that would introduce a space before the comma. One solution would be the following:

print(greeting + ',', salutation, name)

which simply adds the comma to the greeting. You can specify a custom separator, if you want:

>>> print("I", "wish", "to", "register", "a", "complaint", sep="\_")

I\_wish\_to\_register\_a\_complaint

You can also specify a custom end string, to replace the default newline. For example, if you supply an empty

string, you can later continue printing on the same line.

print('Hello,', end='')

print('world!')

This program prints out Hello, world!.1

Importing Something as Something Else

Usually, when you import something from a module, you either use

import somemodule

or use

from somemodule import somefunction

or

from somemodule import somefunction, anotherfunction, yetanotherfunction

or

from somemodule import \*

The fourth version should be used only when you are certain that you want to import everything from the

given module. But what if you have two modules, each containing a function called open, for example—what

do you do then? You could simply import the modules using the first form and then use the functions as

follows:

module1.open(...)

module2.open(...)

But there is another option: you can add an as clause to the end and supply the name you want to use, either

for the entire module:

>>> import math as foobar

>>> foobar.sqrt(4)

2.0

or for the given function:

>>> from math import sqrt as foobar

>>> foobar(4)

2.0

For the open functions, you might use the following:

from module1 import open as open1

from module2 import open as open2

■■Note Some modules, such as os.path, are arranged hierarchically (inside each other). For more about

module structure, see the section on packages in Chapter 10.

Assignment Magic

The humble assignment statement also has a few tricks up its sleeve.

Sequence Unpacking

You’ve seen quite a few examples of assignments, both for variables and for parts of data structures (such

as positions and slices in a list, or slots in a dictionary), but there is more. You can perform several different

assignments simultaneously.

>>> x, y, z = 1, 2, 3

>>> print(x, y, z)

1 2 3

Doesn’t sound useful? Well, you can use it to switch the contents of two (or more) variables.

>>> x, y = y, x

>>> print(x, y, z)

2 1 3

Actually, what I’m doing here is called sequence unpacking (or iterable unpacking). I have a sequence (or an

arbitrary iterable object) of values, and I unpack it into a sequence of variables. Let me be more explicit.

>>> values = 1, 2, 3

>>> values

(1, 2, 3)

>>> x, y, z = values

>>> x

1

This is particularly useful when a function or method returns a tuple (or other sequence or iterable object).

Let’s say that you want to retrieve (and remove) an arbitrary key-value pair from a dictionary. You can

then use the popitem method, which does just that, returning the pair as a tuple. Then you can unpack the

returned tuple directly into two variables.

>>> scoundrel = {'name': 'Robin', 'girlfriend': 'Marion'}

>>> key, value = scoundrel.popitem()

>>> key

'girlfriend'

>>> value

'Marion'

This allows functions to return more than one value, packed as a tuple, easily accessible through a single

assignment. The sequence you unpack must have exactly as many items as the targets you list on the left of

the = sign; otherwise, Python raises an exception when the assignment is performed.

>>> x, y, z = 1, 2

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: need more than 2 values to unpack

>>> x, y, z = 1, 2, 3, 4

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: too many values to unpack

Instead of ensuring that the number of values matches exactly, you can gather up any superfluous ones

using the star operator (\*). For example:

>>> a, b, \*rest = [1, 2, 3, 4]

>>> rest

[3, 4]

You can place this starred variable in other positions, too.

>>> name = "Albus Percival Wulfric Brian Dumbledore"

>>> first, \*middle, last = name.split()

>>> middle

['Percival', 'Wulfric', 'Brian']

The right-hand side of the assignment may be any kind of sequence, but the starred variable will always end

up containing a list. This is true even if the number of values matches exactly.

>>> a, \*b, c = "abc"

>>> a, b, c

('a', ['b'], 'c')

The same kind of gathering can also be used in function argument lists (see Chapter 6).

Chained Assignments

Chained assignments are used as a shortcut when you want to bind several variables to the same value. This

may seem a bit like the simultaneous assignments in the previous section, except that here you are dealing

with only one value:

x = y = somefunction()

which is the same as this:

y = somefunction()

x = y

Note that the preceding statements may not be the same as

x = somefunction()

y = somefunction()

For more about this, see the section about the identity operator (is) later in this chapter.

Augmented Assignments

Instead of writing x = x + 1, you can just put the expression operator (in this case +) before the assignment

operator (=) and write x += 1. This is called an augmented assignment, and it works with all the standard

operators, such as \*, /, %, and so on.

>>> x = 2

>>> x += 1

>>> x \*= 2

>>> x

6

It also works with other data types (as long as the binary operator itself works with those data types).

>>> fnord = 'foo'

>>> fnord += 'bar'

>>> fnord \*= 2

>>> fnord

'foobarfoobar'

Augmented assignments can make your code more compact and concise and, in many cases, more readable.

Blocks: The Joy of Indentation

A block isn’t really a type of statement but something you’re going to need when you tackle the next two

sections.

A block is a group of statements that can be executed if a condition is true (conditional statements),

executed several times (loops), and so on. A block is created by indenting a part of your code, that is, putting

spaces in front of it.

■■Note You can use tab characters to indent your blocks as well. Python interprets a tab as moving to the

next tab stop, with one tab stop every eight spaces, but the standard and preferable style is to use spaces only,

not tabs, and specifically four spaces per each level of indentation.

Each line in a block must be indented by the same amount. The following is pseudocode (not real

Python code) that shows how the indenting works:

this is a line

this is another line:

this is another block

continuing the same block

the last line of this block

phew, there we escaped the inner block

In many languages, a special word or character (for example, begin or {) is used to start a block, and another

(such as end or }) is used to end it. In Python, a colon (:) is used to indicate that a block is about to begin,

and then every line in that block is indented (by the same amount). When you go back to the same amount

of indentation as some enclosing block, you know that the current block has ended. (Many programming

editors and IDEs are aware of how this block indenting works and can help you get it right without much

effort.)

Now, let’s take a look at what these blocks can be used for.

Conditions and Conditional Statements

Until now, you’ve written programs in which each statement is executed, one after the other. It’s time to

move beyond that and let your program choose whether or not to execute a block of statements.

So That’s What Those Boolean Values Are For

Now you are finally going to need those truth values (also called Boolean values, after George Boole, who did

a lot of smart stuff on truth values) that you’ve been bumping into repeatedly.

■■Note If you’ve been paying close attention, you noticed the sidebar in Chapter 1, “Sneak Peek: The if

Statement,” which describes the if statement. I haven’t really introduced it formally until now, and as you’ll

see, there is a bit more to it than what I’ve told you so far.

The following values are considered by the interpreter to mean false when evaluated as a Boolean expression

(for example, as the condition of an if statement):

False None 0 "" () [] {}

In other words, the standard values False and None, numeric zero of all types (including float, complex,

and so on), empty sequences (such as empty strings, tuples, and lists), and empty mappings (such as

dictionaries) are all considered to be false. Everything else2 is interpreted as true, including the special value

True.3

Got it? This means that every value in Python can be interpreted as a truth value, which can be a bit

confusing at first, but it can also be extremely useful. And even though you have all these truth values to

choose from, the “standard” truth values are True and False. In some languages (such as C and Python prior

to version 2.3), the standard truth values are 0 (for false) and 1 (for true). In fact, True and False aren’t that

different—they’re just glorified versions of 0 and 1 that look different but act the same.

>>> True

True

>>> False

False

>>> True == 1

True

>>> False == 0

True

>>> True + False + 42

43

So now, if you see a logical expression returning 1 or 0 (probably in an older version of Python), you will

know that what is really meant is True or False.

The Boolean values True and False belong to the type bool, which can be used (just like, for example,

list, str, and tuple) to convert other values.

>>> bool('I think, therefore I am')

True

>>> bool(42)

True

>>> bool('')

False

>>> bool(0)

False

Because any value can be used as a Boolean value, you will most likely rarely (if ever) need such an explicit

conversion (that is, Python will automatically convert the values for you).

■■Note Although [] and "" are both false (that is, bool([]) == bool("") == False), they are not equal

(that is, [] != ""). The same goes for other false objects of different types (for example, the possibly more

obvious example () != False).

Conditional Execution and the if Statement

Truth values can be combined, and we’ll get back to how to do that, but let’s first see what you can use them

for. Try running the following script:

name = input('What is your name? ')

if name.endswith('Gumby'):

print('Hello, Mr. Gumby')

This is the if statement, which lets you do conditional execution. That means that if the condition (the

expression after if but before the colon) evaluates to true (as defined previously), the following block (in this

case, a single print statement) is executed. If the condition is false, then the block is not executed (but you

guessed that, didn’t you?).

■■Note In the sidebar “Sneak Peek: The if Statement” in Chapter 1, the statement was written on a single

line. That is equivalent to using a single-line block, as in the preceding example.

else Clauses

In the example from the previous section, if you enter a name that ends with “Gumby,” the method name.

endswith returns True, making the if statement enter the block, and the greeting is printed. If you want, you

can add an alternative, with the else clause (called a clause because it isn’t really a separate statement, just a

part of the if statement).

name = input('What is your name?')

if name.endswith('Gumby'):

print('Hello, Mr. Gumby')

else:

print('Hello, stranger')

Here, if the first block isn’t executed (because the condition evaluated to false), you enter the second block

instead. This really shows how easy it is to read Python code, doesn’t it? Just read the code aloud (from if),

and it sounds just like a normal (or perhaps not quite normal) sentence.

There is also a close relative of the if statement, called the conditional expression. This is Python’s

version of the ternary operator from C. This is an expression that uses if and else to determine its value:

status = "friend" if name.endswith("Gumby") else "stranger"

The value of the expression is the first value provided (in this case, "friend") whenever the condition

(whatever comes right after if) is true, and the last value (in this case, "stranger") otherwise.

elif Clauses

If you want to check for several conditions, you can use elif, which is short for “else if.” It is a combination of

an if clause and an else clause—an else clause with a condition.

num = int(input('Enter a number: '))

if num > 0:

print('The number is positive')

elif num < 0:

print('The number is negative')

else:

print('The number is zero')

Nesting Blocks

Let’s throw in a few bells and whistles. You can have if statements inside other if statement blocks, as

follows:

name = input('What is your name? ')

if name.endswith('Gumby'):

if name.startswith('Mr.'):

print('Hello, Mr. Gumby')

elif name.startswith('Mrs.'):

print('Hello, Mrs. Gumby')

else:

print('Hello, Gumby')

else:

print('Hello, stranger')

Here, if the name ends with “Gumby,” you check the start of the name as well—in a separate if statement

inside the first block. Note the use of elif here. The last alternative (the else clause) has no condition—if no

other alternative is chosen, you use the last one. If you want, you can leave out either of the else clauses. If

you leave out the inner else clause, names that don’t start with either “Mr.” or “Mrs.” are ignored (assuming

the name was “Gumby”). If you drop the outer else clause, strangers are ignored.

More Complex Conditions

That’s really all there is to know about if statements. Now let’s return to the conditions themselves, because

they are the really interesting part of conditional execution.

Comparison Operators

Perhaps the most basic operators used in conditions are the comparison operators. They are used (surprise,

surprise) to compare things. Table 5-1 summarizes the comparison operators.

Table 5-1. The Python Comparison Operators

ExpressionDescription

x == yx equals y.

x < yx is less than y.

x > yx is greater than y.

x >= yx is greater than or equal to y.

x <= yx is less than or equal to y.

x != yx is not equal to y.

x is yx and y are the same object.

x is not yx and y are different objects.

x in yx is a member of the container (e.g., sequence) y.

x not in yx is not a member of the container (e.g., sequence) y.

COMPARING INCOMPATIBLE TYPES

In theory, you can compare any two objects x and y for relative size (using operators such as < and <=)

and obtain a truth value. However, such a comparison makes sense only if x and y are of the same or

closely related types (such as two integers or an integer and a floating-point number).

Just as it doesn’t make much sense to add an integer to a string, checking whether an integer is less

than a string seems rather pointless. Oddly, in Python versions prior to 3 you were allowed to do this.

Even if you’re using an older Python, you really should stay away from such comparisons, as the result

is totally arbitrary and may change between each execution of your program. In Python 3, comparing

incompatible types in this way is no longer allowed.

Comparisons can be chained in Python, just like assignments—you can put several comparison operators in

a chain, like this: 0 < age < 100.

Some of these operators deserve some special attention and will be described in the following sections.

The Equality Operator

If you want to know if two things are equal, use the equality operator, written as a double equality sign, ==.

>>> "foo" == "foo"

True

>>> "foo" == "bar"

False

Double? Why can’t you just use a single equality sign, as they do in mathematics? I’m sure you’re clever

enough to figure this out for yourself, but let’s try it.

>>> "foo" = "foo"

SyntaxError: can't assign to literal

The single equality sign is the assignment operator, which is used to change things, which is not what you

want to do when you compare things.

is: The Identity Operator

The is operator is interesting. It seems to work just like ==, but it doesn’t.

>>> x = y = [1, 2, 3]

>>> z = [1, 2, 3]

>>> x == y

True

>>> x == z

True

>>> x is y

True

>>> x is z

False

Until the last example, this looks fine, but then you get that strange result: x is not z, even though they are

equal. Why? Because is tests for identity, rather than equality. The variables x and y have been bound to the

same list, while z is simply bound to another list that happens to contain the same values in the same order.

They may be equal, but they aren’t the same object.

Does that seem unreasonable? Consider this example:

>>> x = [1, 2, 3]

>>> y = [2, 4]

>>> x is not y

True

>>> del x[2]

>>> y[1] = 1

>>> y.reverse()

In this example, I start with two different lists, x and y. As you can see, x is not y (just the inverse of x is y),

which you already know. I change the lists around a bit, and though they are now equal, they are still two

separate lists.

>>> x == y

True

>>> x is y

False

Here, it is obvious that the two lists are equal but not identical.

To summarize, use == to see if two objects are equal, and use is to see if they are identical (the same object).

■■Caution Avoid the use of is with basic, immutable values such as numbers and strings. The result is

unpredictable because of the way Python handles these objects internally.

in: The Membership Operator

I have already introduced the in operator (in Chapter 2, in the section “Membership”). It can be used in

conditions, just like all the other comparison operators.

name = input('What is your name?')

if 's' in name:

print('Your name contains the letter "s".')

else:

print('Your name does not contain the letter "s".')

String and Sequence Comparisons

Strings are compared according to their order when sorted alphabetically.

>>> "alpha" < "beta"

True

The ordering is alphabetical, but the alphabet is all of Unicode, ordered by their code points.

>>> "

True

" < "

"

Actually, characters are sorted by their ordinal values. The ordinal value of a letter can be found with the ord

function, whose inverse is chr:

>>> ord(" ")

128585

>>> ord(" ")

128586

>>> chr(128584)

' '

This approach is quite reasonable and consistent, but it might run counter to how you’d sort things yourself

at times. For example, capital letters are may not work the way you want.

>>> "a" < "B"

False

One trick is to ignore the difference between uppercase and lowercase letters and to use the string method

lower. Here’s an example (see Chapter 3):

>>> "a".lower() < "B".lower()

True

>>> 'FnOrD'.lower() == 'Fnord'.lower()

True

Other sequences are compared in the same manner, except that instead of characters, you may have other

types of elements.

>>> [1, 2] < [2, 1]

True

If the sequences contain other sequences as elements, the same rule applies to these sequence elements.

>>> [2, [1, 4]] < [2, [1, 5]]

True

Boolean Operators

Now, you have plenty of things that return truth values. (In fact, given the fact that all values can be

interpreted as truth values, all expressions return them.) But you may want to check for more than one

condition. For example, let’s say you want to write a program that reads a number and checks whether it’s

between 1 and 10 (inclusive). You could do it like this:

number = int(input('Enter a number between 1 and 10: '))

if number <= 10:

if number >= 1:

print('Great!')

else:

print('Wrong!')

else:

print('Wrong!')

This would work, but it’s clumsy. The fact that you have to write print 'Wrong!' in two places should alert

you to this clumsiness. Duplication of effort is not a good thing. So what do you do? It’s so simple.

number = int(input('Enter a number between 1 and 10: '))

if number <= 10 and number >= 1:

print('Great!')

else:

print('Wrong!')

■■Note I could (and quite probably should) have made this example even simpler by using the following

chained comparison: 1 <= number <= 10.

The and operator is a so-called Boolean operator. It takes two truth values, and it returns true if both are

true, and false otherwise. You have two more of these operators, or and not. With just these three, you can

combine truth values in any way you like.

if ((cash > price) or customer\_has\_good\_credit) and not out\_of\_stock:

give\_goods()

SHORT-CIRCUIT LOGIC AND CONDITIONAL EXPRESSIONS

The Boolean operators have one interesting property: they evaluate only what they need to evaluate. For

example, the expression x and y requires both x and y to be true; so if x is false, the expression returns

false immediately, without worrying about y. Actually, if x is false, it returns x; otherwise, it returns y.

(Can you see how this gives the expected meaning?) This behavior is called short-circuit logic (or lazy

evaluation): the Boolean operators are often called logical operators, and as you can see, the second

value is sometimes “short-circuited.” This works with or, too. In the expression x or y, if x is true, it is

returned; otherwise, y is returned. (Can you see how this makes sense?) Note that this means that any

code you have (such as a function call) after a Boolean operator may not be executed at all. You might

see this behavior exploited in code like the following:

name = input('Please enter your name: ') or '<unknown>'

If no name is input, the or expression has the value '<unknown>'. In many cases, you might want

to use a conditional expression rather than such short-circuit tricks, though statements such as the

previous do have their uses.

Assertions

There is a useful relative of the if statement, which works more or less like this (pseudocode):

if not condition:

crash program

Now, why on earth would you want something like that? Simply because it’s better that your program

crashes when an error condition emerges than at a much later time. Basically, you can require that certain

things be true (for example, when checking required properties of parameters to your functions or as an aid

during initial testing and debugging). The keyword used in the statement is assert.

>>> age = 10

>>> assert 0 < age < 100

>>> age = -1

>>> assert 0 < age < 100

Traceback (most recent call last):

File "<stdin>", line 1, in ?

AssertionError

It can be useful to put the assert statement in your program as a checkpoint, if you know something must be

true for your program to work correctly.

A string may be added after the condition, to explain the assertion.

>>> age = -1

>>> assert 0 < age < 100, 'The age must be realistic'

Traceback (most recent call last):

File "<stdin>", line 1, in ?

AssertionError: The age must be realistic

Loops

Now you know how to do something if a condition is true (or false), but how do you do something several

times? For example, you might want to create a program that reminds you to pay the rent every month, but

with the tools we have looked at until now, you would need to write the program like this (pseudocode):

send mail

wait one month send mail

wait one month send mail

wait one month

(... and so on)

But what if you wanted it to continue doing this until you stopped it? Basically, you want something like this

(again, pseudocode):

while we aren't stopped:

send mail

wait one month

Or, let’s take a simpler example. Let’s say that you want to print out all the numbers from 1 to 100. Again, you

could do it the stupid way.

print(1)

print(2)

print(3)

...

print(99)

print(100)

But you didn’t start using Python because you wanted to do stupid things, right?

while Loops

In order to avoid the cumbersome code of the preceding example, it would be useful to be able to do

something like this:

x = 1

while x <= 100:

print(x)

x += 1

Now, how do you do that in Python? You guessed it—you do it just like that. Not that complicated, is it? You

could also use a loop to ensure that the user enters a name, as follows:

name = ''

while not name:

name = input('Please enter your name: ')

print('Hello, {}!'.format(name))

Try running this and then just pressing the Enter key when asked to enter your name. You’ll see that the

question appears again, because name is still an empty string, which evaluates to false.

■■Tip What would happen if you entered just a space character as your name? Try it. It is accepted because

a string with one space character is not empty and therefore not false. This is definitely a flaw in our little

program, but it’s easily corrected: just change while not name to while not name or name.isspace() or,

perhaps, while not name.strip().

for Loops

The while statement is very flexible. It can be used to repeat a block of code while any condition is true.

While this may be very nice in general, sometimes you may want something tailored to your specific needs.

One such need is to perform a block of code for each element of a set (or, actually, sequence or other iterable

object) of values.

■■Note Basically, an iterable object is any object that you can iterate over (that is, use in a for loop). You

learn more about iterables and iterators in Chapter 9, but for now, you can simply think of them as sequences.

You can do this with the for statement:

words = ['this', 'is', 'an', 'ex', 'parrot']

for word in words:

print(word)

or

numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

for number in numbers:

print(number)

Because iterating (another word for looping) over a range of numbers is a common thing to do, Python has a

built-in function to make ranges for you.

>>> range(0, 10)

range(0, 10)

>>> list(range(0, 10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Ranges work like slices. They include the first limit (in this case 0) but not the last (in this case 10). Quite

often, you want the ranges to start at 0, and this is actually assumed if you supply only one limit (which will

then be the last).

>>> range(10)

range(0, 10)

The following program writes out the numbers from 1 to 100:

for number in range(1,101):

print(number)

Notice that this is much more compact than the while loop I used earlier.

■■Tip

If you can use a for loop rather than a while loop, you should probably do so.

Iterating Over Dictionaries

To loop over the keys of a dictionary, you can use a plain for statement, just as you can with sequences.

d = {'x': 1, 'y': 2, 'z': 3}

for key in d:

print(key, 'corresponds to', d[key])

You could have used a dictionary method such as keys to retrieve the keys. If only the values were of interest,

you could have used d.values. You may remember that d.items returns key-value pairs as tuples. One great

thing about for loops is that you can use sequence unpacking in them.

for key, value in d.items():

print(key, 'corresponds to', value)

■■Note As always, the order of dictionary elements is undefined. In other words, when iterating over either

the keys or the values of a dictionary, you can be sure that you’ll process all of them, but you can’t know in

which order. If the order is important, you can store the keys or values in a separate list and, for example, sort it

before iterating over it. If you want your mapping to remember the insertion order of its items, you can use class

OrderedDict from the collections module.

Some Iteration Utilities

Python has several functions that can be useful when iterating over a sequence (or other iterable object).

Some of these are available in the itertools module (mentioned in Chapter 10), but there are some built-in

functions that come in quite handy as well.

Parallel Iteration

Sometimes you want to iterate over two sequences at the same time. Let’s say that you have the following

two lists:

names = ['anne', 'beth', 'george', 'damon']

ages = [12, 45, 32, 102]

If you want to print out names with corresponding ages, you could do the following:

for i in range(len(names)):

print(names[i], 'is', ages[i], 'years old')

Here, i serves as a standard variable name for loop indices (as these things are called). A useful tool for

parallel iteration is the built-in function zip, which “zips” together the sequences, returning a sequence of

tuples. The return value is a special zip object, meant for iteration, but it can be converted using list, to take

a look at its contents.

>>> list(zip(names, ages))

[('anne', 12), ('beth', 45), ('george', 32), ('damon', 102)]

Now we can unpack the tuples in our loop.

for name, age in zip(names, ages):

print(name, 'is', age, 'years old')

The zip function works with as many sequences as you want. It’s important to note what zip does when the

sequences are of different lengths: it stops when the shortest sequence is used up.

>>> list(zip(range(5), range(100000000)))

[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]

Numbered Iteration

In some cases, you want to iterate over a sequence of objects and at the same time have access to the index

of the current object. For example, you might want to replace every string that contains the substring 'xxx'

in a list of strings. There would certainly be many ways of doing this, but let’s say you want to do something

along the following lines:

for string in strings:

if 'xxx' in string:

index = strings.index(string) # Search for the string in the list of strings

strings[index] = '[censored]'

This would work, but it seems unnecessary to search for the given string before replacing it. Also, if you

didn’t replace it, the search might give you the wrong index (that is, the index of some previous occurrence

of the same word). A better version would be the following:

index = 0

for string in strings:

if 'xxx' in string:

strings[index] = '[censored]'

index += 1

This also seems a bit awkward, although acceptable. Another solution is to use the built-in function

enumerate.

for index, string in enumerate(strings):

if 'xxx' in string:

strings[index] = '[censored]'

This function lets you iterate over index-value pairs, where the indices are supplied automatically.

Reversed and Sorted Iteration

Let’s look at another couple of useful functions: reversed and sorted. They’re similar to the list methods

reverse and sort (with sorted taking arguments similar to those taken by sort), but they work on any

sequence or iterable object, and instead of modifying the object in place, they return reversed and sorted

versions.

>>> sorted([4, 3, 6, 8, 3])

[3, 3, 4, 6, 8]

>>> sorted('Hello, world!')

[' ', '!', ',', 'H', 'd', 'e', 'l', 'l', 'l', 'o', 'o', 'r', 'w']

>>> list(reversed('Hello, world!'))

['!', 'd', 'l', 'r', 'o', 'w', ' ', ',', 'o', 'l', 'l', 'e', 'H']

>>> ''.join(reversed('Hello, world!'))

'!dlrow ,olleH'

Note that although sorted returns a list, reversed returns a more mysterious iterable object, just like zip.

You don’t need to worry about what this really means; you can use it in for loops or methods such as join

without any problems. You just can’t index or slice it, or call list methods on it directly. In order to perform

those tasks, just convert the returned object with list.

■■Tip

We can use the trick of lowercasing to get proper alphabetical sorting. For example, you could use

str.lower as the key argument to sort or sorted. For example, sorted("aBc", key=str.lower) returns

['a', 'B', 'c'].

Breaking Out of Loops

Usually, a loop simply executes a block until its condition becomes false, or until it has used up all sequence

elements. But sometimes you may want to interrupt the loop, to start a new iteration (one “round” of

executing the block), or to simply end the loop.

break

To end (break out of ) a loop, you use break. Let’s say you wanted to find the largest square (the result of an

integer multiplied by itself ) below 100. Then you start at 100 and iterate downward to 0. When you’ve found

a square, there’s no need to continue, so you simply break out of the loop.

from math import sqrt

for n in range(99, 0, -1):

root = sqrt(n)

if root == int(root):

print(n)

break

If you run this program, it will print out 81 and stop. Notice that I’ve added a third argument to range—

that’s the step, the difference between every pair of adjacent numbers in the sequence. It can be used to

iterate downward as I did here, with a negative step value, and it can be used to skip numbers.

>>> range(0, 10, 2)

[0, 2, 4, 6, 8]

continue

The continue statement is used less often than break. It causes the current iteration to end and to “jump” to

the beginning of the next. It basically means “skip the rest of the loop body, but don’t end the loop.” This can

be useful if you have a large and complicated loop body and several possible reasons for skipping it. In that

case, you can use continue, as follows:

for x in seq:

if condition1: continue

if condition2: continue

if condition3: continue

do\_something()

do\_something\_else()

do\_another\_thing()

etc()

In many cases, however, simply using an if statement is just as good.

for x in seq:

if not (condition1 or condition2 or condition3):

do\_something()

do\_something\_else()

do\_another\_thing()

etc()

Even though continue can be a useful tool, it is not essential. The break statement, however, is something

you should get used to, because it is used quite often in concert with while True, as explained in the next

section.

The while True/break Idiom

The for and while loops in Python are quite flexible, but every once in a while, you may encounter a

problem that makes you wish you had more functionality. For example, let’s say you want to do something

when a user enters words at a prompt, and you want to end the loop when no word is provided. One way of

doing that would be like this:

word = 'dummy'

while word:

word = input('Please enter a word: ')

# do something with the word:

print('The word was', word)

Here is an example of a session:

Please enter a word: first

The word was first

Please enter a word: second

The word was second

Please enter a word:

This works just as desired. (Presumably, you would do something more useful with the word than print it

out, though.) However, as you can see, this code is a bit ugly. To enter the loop in the first place, you need

to assign a dummy (unused) value to word. Dummy values like this are usually a sign that you aren’t doing

things quite right. Let’s try to get rid of it.

word = input('Please enter a word: ')

while word:

# do something with the word:

print('The word was ', word)

word = input('Please enter a word: ')

Here the dummy is gone, but I have repeated code (which is also a bad thing): I need to use the same

assignment and call to input in two places. How can I avoid that? I can use the while True/break idiom.

while True:

word = input('Please enter a word: ')

if not word: break

# do something with the word:

print('The word was ', word)

The while True part gives you a loop that will never terminate by itself. Instead, you put the condition in an

if statement inside the loop, which calls break when the condition is fulfilled. Thus, you can terminate the

loop anywhere inside the loop instead of only at the beginning (as with a normal while loop). The if/break

line splits the loop naturally in two parts: the first takes care of setting things up (the part that would be

duplicated with a normal while loop), and the other part makes use of the initialization from the first part,

provided that the loop condition is true.

Although you should be wary of using break too often in your code (because it can make your loops

harder to read, especially if you put more than one break in a single loop), this specific technique is so

common that most Python programmers (including yourself ) will probably be able to follow your intentions.

else Clauses in Loops

When you use break statements in loops, it is often because you have “found” something or because

something has “happened.” It’s easy to do something when you break out (like print(n)), but sometimes

you may want to do something if you didn’t break out. But how do you find out? You could use a Boolean

variable, set it to False before the loop, and set it to True when you break out. Then you can use an if

statement afterward to check whether you did break out.

broke\_out = False

for x in seq:

do\_something(x)

if condition(x):

broke\_out = True

break

do\_something\_else(x)

if not broke\_out:

print("I didn't break out!")

A simpler way is to add an else clause to your loop—it is executed only if you didn’t call break. Let’s reuse

the example from the preceding section on break.

from math import sqrt

for n in range(99, 81, -1):

root = sqrt(n)

if root == int(root):

print(n)

break

else:

print("Didn't find it!")

Notice that I changed the lower (exclusive) limit to 81 to test the else clause. If you run the program, it prints

out “Didn’t find it!” because (as you saw in the section on break) the largest square below 100 is 81. You can

use continue, break, and else clauses with both for loops and while loops.

Comprehensions—Slightly Loopy

List comprehension is a way of making lists from other lists (similar to set comprehension, if you know that

term from mathematics). It works in a way similar to for loops and is actually quite simple.

>>> [x \* x for x in range(10)]

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

The list is composed of x\*x for each x in range(10). Pretty straightforward? What if you want to print out

only those squares that are divisible by 3? Then you can use the modulo operator—y % 3 returns zero

when y is divisible by 3. (Note that x\*x is divisible by 3 only if x is divisible by 3.) You put this into your list

comprehension by adding an if part to it.

>>> [x\*x for x in range(10) if x % 3 == 0]

[0, 9, 36, 81]

You can also add more for parts.

>>> [(x, y) for x in range(3) for y in range(3)]

[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]

As a comparison, the following two for loops build the same list:

result = []

for x in range(3):

for y in range(3)

result.append((x, y))

This can be combined with an if clause, just as before.

>>> girls = ['alice', 'bernice', 'clarice']

>>> boys = ['chris', 'arnold', 'bob']

>>> [b+'+'+g for b in boys for g in girls if b[0] == g[0]]

['chris+clarice', 'arnold+alice', 'bob+bernice']

This gives the pairs of boys and girls who have the same initial letter in their first name.

A BETTER SOLUTION

The boy/girl pairing example isn’t particularly efficient because it checks every possible pairing. There

are many ways of solving this problem in Python. The following was suggested by Alex Martelli:

girls = ['alice', 'bernice', 'clarice']

boys = ['chris', 'arnold', 'bob']

letterGirls = {}

for girl in girls:

letterGirls.setdefault(girl[0], []).append(girl)

print([b+'+'+g for b in boys for g in letterGirls[b[0]]])

This program constructs a dictionary, called letterGirls, where each entry has a single letter as

its key and a list of girls’ names as its value. (The setdefault dictionary method is described in the

previous chapter.) After this dictionary has been constructed, the list comprehension loops over all the

boys and looks up all the girls whose name begins with the same letter as the current boy. This way,

the list comprehension doesn’t need to try out every possible combination of boy and girl and check

whether the first letters match.

Using normal parentheses instead of brackets will not give you a “tuple comprehension”—you’ll end up

with a generator. See the sidebar “Loopy Generators” in Chapter 9 for more information. You can, however,

use curly braces to perform dictionary comprehension.

>>> squares = {i:"{} squared is {}".format(i, i\*\*2) for i in range(10)}

>>> squares[8]

'8 squared is 64'

Instead of a single expression in front of the for, as you would have with a list comprehension, you have two

expressions separated by a colon. These will become the keys and their corresponding values.

And Three for the Road

To end the chapter, let’s take a quick look at three more statements: pass, del, and exec.

Nothing Happened!

Sometimes you need to do nothing. This may not be very often, but when it happens, it’s good to know that

you have the pass statement.

>>> pass

>>>

There’s not much going on here.

Now, why on Earth would you want a statement that does nothing? It can be useful as a placeholder while

you are writing code. For example, you may have written an if statement and you want to try it, but you lack

the code for one of your blocks. Consider the following:

if name == 'Ralph Auldus Melish':

print('Welcome!')

elif name == 'Enid':

# Not finished yet ...

elif name == 'Bill Gates':

print('Access Denied')

This code won’t run because an empty block is illegal in Python. To fix this, simply add a pass statement to

the middle block.

if name == 'Ralph Auldus Melish':

print('Welcome!')

elif name == 'Enid':

# Not finished yet ...

pass

elif name == 'Bill Gates':

print('Access Denied')

■■Note An alternative to the combination of a comment and a pass statement is to simply insert a string.

This is especially useful for unfinished functions (see Chapter 6) and classes (see Chapter 7) because they will

then act as docstrings (explained in Chapter 6).

Deleting with del

In general, Python deletes objects that you don’t use anymore (because you no longer refer to them through

any variables or parts of your data structures).

>>> scoundrel = {'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}

>>> robin = scoundrel

>>> scoundrel

{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}

>>> robin

{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}

>>> scoundrel = None

>>> robin

{'age': 42, 'first name': 'Robin', 'last name': 'of Locksley'}

>>> robin = None

At first, robin and scoundrel are both bound to the same dictionary. So when I assign None to scoundrel,

the dictionary is still available through robin. But when I assign None to robin as well, the dictionary

suddenly floats around in the memory of the computer with no name attached to it. There is no way I can

retrieve it or use it, so the Python interpreter (in its infinite wisdom) simply deletes it. (This is called garbage

collection.) Note that I could have used any value other than None as well. The dictionary would be just as

gone.

Another way of doing this is to use the del statement (which we used to delete sequence and dictionary

elements in Chapters 2 and 4, remember?). Not only does this remove a reference to an object, but it also

removes the name itself.

>>> x = 1

>>> del x

>>> x

Traceback (most recent call last):

File "<pyshell#255>", line 1, in ?

x

NameError: name 'x' is not defined

This may seem easy, but it can actually be a bit tricky to understand at times. For instance, in the following

example, x and y refer to the same list:

>>> x = ["Hello", "world"]

>>> y = x

>>> y[1] = "Python"

>>> x

['Hello', 'Python']

You might assume that by deleting x, you would also delete y, but that is not the case.

>>> del x

>>> y

['Hello', 'Python']

Why is this? x and y referred to the same list, but deleting x didn’t affect y at all. The reason for this is that you

delete only the name, not the list itself (the value). In fact, there is no way to delete values in Python—and

you don’t really need to, because the Python interpreter does it by itself whenever you don’t use the value

anymore.

Executing and Evaluating Strings with exec and eval

Sometimes you may want to create Python code “on the fly” and execute it as a statement or evaluate it as

an expression. This may border on dark magic at times—consider yourself warned. Both exec and eval are

functions, but exec used to be a statement type of its own, and eval is closely related to it, so that is why I

discuss them here.

■■Caution In this section, you learn to execute Python code stored in a string. This is a potential security hole

of great dimensions. If you execute a string where parts of the contents have been supplied by a user, you have

little or no control over what code you are executing. This is especially dangerous in network applications, such

as Common Gateway Interface (CGI) scripts, which you will learn about in Chapter 15.

exec

The exec function is used to execute a string.

>>> exec("print('Hello, world!')")

Hello, world!

However, using the exec statement with a single argument is rarely a good thing. In most cases, you want

to supply it with a namespace—a place where it can put its variables. Otherwise, the code will corrupt your

namespace (that is, change your variables). For example, let’s say that the code uses the name sqrt.

>>> from math import sqrt

>>> exec("sqrt = 1")

>>> sqrt(4)

Traceback (most recent call last):

File "<pyshell#18>", line 1, in ?

sqrt(4)

TypeError: object is not callable: 1

Well, why would you do something like that in the first place? The exec function is mainly useful when

you build the code string on the fly. And if the string is built from parts that you get from other places, and

possibly from the user, you can rarely be certain of exactly what it will contain. So to be safe, you give it a

dictionary, which will work as a namespace for it.

■■Note The concept of namespaces, or scopes, is an important one. You will look at it in depth in the next

chapter, but for now, you can think of a namespace as a place where you keep your variables, much like an

invisible dictionary. So when you execute an assignment like x = 1, you store the key x with the value 1 in the

current namespace, which will often be the global namespace (which we have been using, for the most part, up

until now) but doesn’t have to be.

You do this by adding a second argument—some dictionary that will function as the namespace for your

code string.4

>>> from math import sqrt

>>> scope = {}

>>> exec('sqrt = 1', scope)

>>> sqrt(4)

2.0

>>> scope['sqrt']

1

As you can see, the potentially destructive code does not overwrite the sqrt function. The function works

just as it should, and the sqrt variable resulting from the exec’ed assignment is available from the scope.

Note that if you try to print out scope, you see that it contains a lot of stuff because the dictionary called

\_\_builtins\_\_ is automatically added and contains all built-in functions and values.

>>> len(scope)

2

>>> scope.keys()

['sqrt', '\_\_builtins\_\_']

eval

A built-in function that is similar to exec is eval (for “evaluate”). Just as exec executes a series of Python

statements, eval evaluates a Python expression (written in a string) and returns the resulting value. (exec

doesn’t return anything because it is a statement itself.) For example, you can use the following to make a

Python calculator:

>>> eval(input("Enter an arithmetic expression: "))

Enter an arithmetic expression: 6 + 18 \* 2

42

You can supply a namespace with eval, just as with exec, although expressions rarely rebind variables in the

way statements usually do.

■■Caution Even though expressions don’t rebind variables as a rule, they certainly can (for example, by

calling functions that rebind global variables). Therefore, using eval with an untrusted piece of code is no safer

than using exec. There is, currently, no safe way of executing untrusted code in Python. One alternative is to

use an implementation of Python such as Jython (see Chapter 17) and use some native mechanism such as the

Java sandbox.

PRIMING THE SCOPE

When supplying a namespace for exec or eval, you can also put some values in before actually using

the namespace.

>>> scope = {}

>>> scope['x'] = 2

>>> scope['y'] = 3

>>> eval('x \* y', scope)

6

In the same way, a scope from one exec or eval call can be used again in another one.

>>> scope = {}

>>> exec('x = 2', scope)

>>> eval('x \* x', scope)

4

You could build up rather complicated programs this way, but … you probably shouldn’t.

A Quick Summary

In this chapter, you saw several kinds of statements.

Printing: You can use the print statement to print several values by separating

them with commas. If you end the statement with a comma, later print

statements will continue printing on the same line.

Importing: Sometimes you don’t like the name of a function you want to

import—perhaps you’ve already used the name for something else. You can use

the import … as … statement to locally rename a function.

Assignments: You saw that through the wonder of sequence unpacking and

chained assignments, you can assign values to several variables at once, and that

with augmented assignments, you can change a variable in place.

Blocks: Blocks are used as a means of grouping statements through indentation.

They are used in conditionals and loops and, as you see later in the book, in

function and class definitions, among other things.

Conditionals: A conditional statement either executes a block or not, depending

on a condition (Boolean expression). Several conditionals can be strung together

with if/elif/ else. A variation on this theme is the conditional expression, a if

b else c.

Assertions: An assertion simply asserts that something (a Boolean expression)

is true, optionally with a string explaining why it must be so. If the expression

happens to be false, the assertion brings your program to a halt (or actually raises

an exception—more on that in Chapter 8). It’s better to find an error early than to

let it sneak around your program until you don’t know where it originated.

Loops: You either can execute a block for each element in a sequence (such as

a range of numbers) or can continue executing it while a condition is true. To

skip the rest of the block and continue with the next iteration, use the continue

statement; to break out of the loop, use the break statement. Optionally, you may

add an else clause at the end of the loop, which will be executed if you didn’t

execute any break statements inside the loop.

Comprehension: These aren’t really statements—they are expressions that

look a lot like loops, which is why I grouped them with the looping statements.

Through list comprehension, you can build new lists from old ones, applying

functions to the elements, filtering out those you don’t want, and so on.

The technique is quite powerful, but in many cases, using plain loops and

conditionals (which will always get the job done) may be more readable. Similar

expressions can be used to construct dictionaries.

pass, del, exec, and eval: The pass statement does nothing, which can be useful

as a placeholder, for example. The del statement is used to delete variables or

parts of a data structure but cannot be used to delete values. The exec function

is used to execute a string as if it were a Python program. The eval function

evaluates an expression written in a string and returns the result.

New Functions in This Chapter

FunctionDescription

chr(n)Returns a one-character string when passed ordinal n (0 ≤ n < 256)

eval(source[, globals[, locals]])Evaluates a string as an expression and returns the value

exec(source[, globals[, locals]])Evaluates and executes a string as a statement

enumerate(seq)Yields (index, value) pairs suitable for iteration

ord(c)Returns the integer ordinal value of a one-character string

range([start,] stop[, step])Creates a list of integers

reversed(seq)Yields the values of seq in reverse order, suitable for iteration

sorted(seq[, cmp][, key][, reverse]) Returns a list with the values of seq in sorted order

xrange([start,] stop[, step])Creates an xrange object, used for iteration

zip(seq1, seq2,…)Creates a new sequence suitable for parallel iteration

What Now?

Now you’ve cleared the basics. You can implement any algorithm you can dream up; you can read in

parameters and print out the results. In the next couple of chapters, you learn about something that will help

you write larger programs without losing the big picture. That something is called abstraction.