8. More About Iteration

8.1. Iteration Revisited

8.2. The for loop revisited

8.3. The while Statement

8.4. Randomly Walking Turtles

8.5. The 3n + 1 Sequence

8.6. Newton’s Method

8.7. The Accumulator Pattern Revisited

8.8. Other uses of while

8.8.1. Sentinel Values

8.8.2. Validating Input

8.9. Algorithms Revisited

8.10. Simple Tables

8.11. 2-Dimensional Iteration: Image Processing

8.11.1. The RGB Color Model

8.11.2. Image Objects

8.11.3. Image Processing and Nested Iteration

8.12. Image Processing on Your Own

8.13. Glossary

8.14. Exercises

SavvyPython

View:

Textbook

Social

Search

User

Scratch Activecode

Help

This Chapter

8.1. Iteration Revisited

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.

Repeated execution of a sequence of statements is called iteration. Because iteration is so common, Python provides several language features to make it easier. We’ve already seen the for statement in a previous chapter. This is a very common form of iteration in Python. In this chapter we are going to look at the while statement — another way to have your program do iteration.

8.2. The for loop revisited

Recall that the for loop processes each item in a list. Each item in turn is (re-)assigned to the loop variable, and the body of the loop is executed. We saw this example in an earlier chapter.

1 for f in ["Joe", "Amy", "Brad", "Angelina", "Zuki", "Thandi", "Paris"]:

2 print("Hi", f, "Please come to my party on Saturday")

3

We have also seen iteration paired with the update idea to form the accumulator pattern. For example, to compute the sum of the first n integers, we could create a for loop using the range to produce the numbers 1 through n. Using the accumulator pattern, we can start with a running total variable initialized to 0 and on each iteration, add the current value of the loop variable. A function to compute this sum is shown below.

1 def sumTo(aBound):

2 theSum = 0

3 for aNumber in range(1, aBound + 1):

4 theSum = theSum + aNumber

5

​6 return theSum

7

​8 print(sumTo(4))

9

​

10 print(sumTo(1000))

11

​

To review, the variable theSum is called the accumulator. It is initialized to zero before we start the loop. The loop variable, aNumber will take on the values produced by the range(1, aBound + 1) function call. Note that this produces all the integers from 1 up to the value of aBound. If we had not added 1 to aBound, the range would have stopped one value short since range does not include the upper bound.

The assignment statement, theSum = theSum + aNumber, updates theSum each time through the loop. This accumulates the running total. Finally, we return the value of the accumulator.

8.3. The while Statement

There is another Python statement that can also be used to build an iteration. It is called the while statement. The while statement provides a much more general mechanism for iterating. Similar to the if statement, it uses a boolean expression to control the flow of execution. The body of while will be repeated as long as the controlling boolean expression evaluates to True.

We can use the while loop to create any type of iteration we wish, including anything that we have previously done with a for loop. For example, the program in the previous section could be rewritten using while. Instead of relying on the range function to produce the numbers for our summation, we will need to produce them ourselves. To to this, we will create a variable called aNumber and initialize it to 1, the first number in the summation. Every iteration will add aNumber to the running total until all the values have been used. In order to control the iteration, we must create a boolean expression that evaluates to True as long as we want to keep adding values to our running total. In this case, as long as aNumber is less than or equal to the bound, we should keep going.

Here is a new version of the summation program that uses a while statement.

1 def sumTo(aBound):

2 """ Return the sum of 1+2+3 ... n """

3

​4 theSum = 0

5 aNumber = 1

6 while aNumber <= aBound:

7 theSum = theSum + aNumber

8 aNumber = aNumber + 1

9 return theSum

10

​11 print(sumTo(4))

12

​13 print(sumTo(1000))

14

​

You can almost read the while statement as if it were in natural language. It means, while aNumber is less than or equal to aBound, continue executing the body of the loop. Within the body, each time, update theSum using the accumulator pattern and increment aNumber. After the body of the loop, we go back up to the condition of the while and reevaluate it. When aNumber becomes greater than aBound, the condition fails and flow of control continues to the return statement.

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

Evaluate the condition, yielding False or True.

If the condition is False, exit the while statement and continue execution at the next statement.

If the condition is True, execute each of the statements in the body and then go back to step 1.

The body consists of all of the statements below the header with the same indentation.

This type of flow is called a loop because the third step loops back around to the top. Notice that if the condition is False the first time through the loop, the statements inside the loop are never executed.

Warning

Though Python’s while is very close to the English “while”, there is an important difference: In English “while X, do Y”, we usually assume that immediately after X becomes false, we stop with Y. In Python there is not an immediate stop: After the initial test, any following tests come only after the execution of the whole body, even if the condition becomes false in the middle of the loop body.

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. This is called an infinite loop. An endless source of amusement for computer scientists is the observation that the directions written on the back of the shampoo bottle (lather, rinse, repeat) create an infinite loop.

In the case shown above, we can prove that the loop terminates because we know that the value of aBound is finite, and we can see that the value of aNumber increments each time through the loop, so eventually it will have to exceed aBound. In other cases, it is not so easy to tell.

Introduction of the while statement causes us to think about the types of iteration we have seen. The for statement will always iterate through a sequence of values like the list of names for the party or the list of numbers created by range. Since we know that it will iterate once for each value in the collection, it is often said that a for loop creates a definite iteration because we definitely know how many times we are going to iterate. On the other hand, the while statement is dependent on a condition that needs to evaluate to False in order for the loop to terminate. Since we do not necessarily know when this will happen, it creates what we call indefinite iteration. Indefinite iteration simply means that we don’t know how many times we will repeat but eventually the condition controlling the iteration will fail and the iteration will stop. (Unless we have an infinite loop which is of course a problem.)

What you will notice here is that the while loop is more work for you — the programmer — than the equivalent for loop. When using a while loop you have to control the loop variable yourself. You give it an initial value, test for completion, and then make sure you change something in the body so that the loop terminates.

So why have two kinds of loop if for looks easier? The next section, Randomly Walking Turtles, shows an indefinite iteration where we need the extra power that we get from the while loop.

Check your understanding

iter-3-5: True or False: You can rewrite any for-loop as a while-loop.

A. True

B. False

iter-3-6: The following code contains an infinite loop. Which is the best explanation for why the loop does not terminate?

n = 10

answer = 1

while n > 0:

answer = answer + n

n = n + 1

print(answer)

A. n starts at 10 and is incremented by 1 each time through the loop, so it will always be positive

B. answer starts at 1 and is incremented by n each time, so it will always be positive

C. You cannot compare n to 0 in while loop. You must compare it to another variable.

D. In the while loop body, we must set n to False, and this code does not do that.

iter-3-7: What is printed by this code?

n = 1

x = 2

while n < 5:

n = n + 1

x = x + 1

n = n + 2

x = x + n

print(n, x)

A. 4 7

B. 5 7

C. 7 15

8.3. The while Statement

There is another Python statement that can also be used to build an iteration. It is called the while statement. The while statement provides a much more general mechanism for iterating. Similar to the if statement, it uses a boolean expression to control the flow of execution. The body of while will be repeated as long as the controlling boolean expression evaluates to True.

The following figure shows the flow of control.

We can use the while loop to create any type of iteration we wish, including anything that we have previously done with a for loop. For example, the program in the previous section could be rewritten using while. Instead of relying on the range function to produce the numbers for our summation, we will need to produce them ourselves. To to this, we will create a variable called aNumber and initialize it to 1, the first number in the summation. Every iteration will add aNumber to the running total until all the values have been used. In order to control the iteration, we must create a boolean expression that evaluates to True as long as we want to keep adding values to our running total. In this case, as long as aNumber is less than or equal to the bound, we should keep going.

Here is a new version of the summation program that uses a while statement.

1 def sumTo(aBound):

2 """ Return the sum of 1+2+3 ... n """

3

​4 theSum = 0

5 aNumber = 1

6 while aNumber <= aBound:

7 theSum = theSum + aNumber

8 aNumber = aNumber + 1

9 r eturn theSum

10

​11 print(sumTo(4))

12

​13 print(sumTo(1000))

14

​

You can almost read the while statement as if it were in natural language. It means, while aNumber is less than or equal to aBound, continue executing the body of the loop. Within the body, each time, update theSum using the accumulator pattern and increment aNumber. After the body of the loop, we go back up to the condition of the while and reevaluate it. When aNumber becomes greater than aBound, the condition fails and flow of control continues to the return statement.

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

Evaluate the condition, yielding False or True.

If the condition is False, exit the while statement and continue execution at the next statement.

If the condition is True, execute each of the statements in the body and then go back to step 1.

The body consists of all of the statements below the header with the same indentation.

This type of flow is called a loop because the third step loops back around to the top. Notice that if the condition is False the first time through the loop, the statements inside the loop are never executed.

Warning

Though Python’s while is very close to the English “while”, there is an important difference: In English “while X, do Y”, we usually assume that immediately after X becomes false, we stop with Y. In Python there is not an immediate stop: After the initial test, any following tests come only after the execution of the whole body, even if the condition becomes false in the middle of the loop body.

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. This is called an infinite loop. An endless source of amusement for computer scientists is the observation that the directions written on the back of the shampoo bottle (lather, rinse, repeat) create an infinite loop.

In the case shown above, we can prove that the loop terminates because we know that the value of aBound is finite, and we can see that the value of aNumber increments each time through the loop, so eventually it will have to exceed aBound. In other cases, it is not so easy to tell.

Introduction of the while statement causes us to think about the types of iteration we have seen. The for statement will always iterate through a sequence of values like the list of names for the party or the list of numbers created by range. Since we know that it will iterate once for each value in the collection, it is often said that a for loop creates a definite iteration because we definitely know how many times we are going to iterate. On the other hand, the while statement is dependent on a condition that needs to evaluate to False in order for the loop to terminate. Since we do not necessarily know when this will happen, it creates what we call indefinite iteration. Indefinite iteration simply means that we don’t know how many times we will repeat but eventually the condition controlling the iteration will fail and the iteration will stop. (Unless we have an infinite loop which is of course a problem.)

What you will notice here is that the while loop is more work for you — the programmer — than the equivalent for loop. When using a while loop you have to control the loop variable yourself. You give it an initial value, test for completion, and then make sure you change something in the body so that the loop terminates.

So why have two kinds of loop if for looks easier? The next section, Randomly Walking Turtles, shows an indefinite iteration where we need the extra power that we get from the while loop.

Check your understanding

iter-3-5: True or False: You can rewrite any for-loop as a while-loop.

A. True

B. False

iter-3-6: The following code contains an infinite loop. Which is the best explanation for why the loop does not terminate?

n = 10

answer = 1

while n > 0:

answer = answer + n

n = n + 1

print(answer)

A. n starts at 10 and is incremented by 1 each time through the loop, so it will always be positive

B. answer starts at 1 and is incremented by n each time, so it will always be positive

C. You cannot compare n to 0 in while loop. You must compare it to another variable.

D. In the while loop body, we must set n to False, and this code does not do that.

iter-3-7: What is printed by this code?

n = 1

x = 2

while n < 5:

n = n + 1

x = x + 1

n = n + 2

x = x + n

print(n, x)

A. 4 7

B. 5 7

C. 7 15

8.5. The 3n + 1 Sequence

As another example of indefinite iteration, let’s look at a sequence that has fascinated mathematicians for many years. The rule for creating the sequence is to start from some positive integer, call it n, and to generate the next term of the sequence from n, either by halving n, whenever n is even, or else by multiplying it by three and adding 1 when it is odd. The sequence terminates when n reaches 1.

This Python function captures that algorithm. Try running this program several times supplying different values for n.

1 def seq3np1(n):

2 """ Print the 3n+1 sequence from n, terminating when it reaches 1."""

3 while n != 1:

4 print(n)

5 if n % 2 == 0: # n is even

6 n = n // 2

7 else: # n is odd

8n = n \* 3 + 1

9 print(n) # the last print is 1

10

​11 seq3np1(3)

12

​

The condition for this loop is n != 1. The loop will continue running until n == 1 (which will make the condition false).

Each time through the loop, the program prints the value of n and then checks whether it is even or odd using the remainder operator. If it is even, the value of n is divided by 2 using integer division. If it is odd, the value is replaced by n \* 3 + 1. Try some other examples.

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.

You might like to have some fun and see if you can find a small starting number that needs more than a hundred steps before it terminates.

Experimenting with the 3n+1 Sequence In this guided lab exercise we will try to learn more about this sequence.

Particular values aside, the interesting question is whether we can prove that this sequence terminates for all positive values of n. So far, no one has been able to prove it or disprove it!

Think carefully about what would be needed for a proof or disproof of the hypothesis “All positive integers will eventually converge to 1”. With fast computers we have been able to test every integer up to very large values, and so far, they all eventually end up at 1. But this doesn’t mean that there might not be some as-yet untested number which does not reduce to 1.

You’ll notice that if you don’t stop when you reach one, the sequence gets into its own loop: 1, 4, 2, 1, 4, 2, 1, 4, and so on. One possibility is that there might be other cycles that we just haven’t found.

Choosing between for and while

Use a for loop if you know the maximum number of times that you’ll need to execute the body. For example, if you’re traversing a list of elements, or can formulate a suitable call to range, then choose the for loop.

So any problem like “iterate this weather model run for 1000 cycles”, or “search this list of words”, “check all integers up to 10000 to see which are prime” suggest that a for loop is best.

By contrast, if you are required to repeat some computation until some condition is met, as we did in this 3n + 1 problem, you’ll need a while loop.

As we noted before, the first case is called definite iteration — we have some definite bounds for what is needed. The latter case is called indefinite iteration — we are not sure how many iterations we’ll need — we cannot even establish an upper bound!

Check your understanding

iter-5-2: Consider the code that prints the 3n+1 sequence in ActiveCode box 6. Will the while loop in this code always terminate for any positive integer value of n?

A. Yes.

B. No.

C. No one knows.

8.6. Newton’s Method

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 n. If you start with almost any approximation, you can compute a better approximation with the following formula:

better = 1/2 \* (approx + n/approx)

Execute this algorithm a few times using your calculator. Can you see why each iteration brings your estimate a little closer? One of the amazing properties of this particular algorithm is how quickly it converges to an accurate answer.

The following implementation of Newton’s method requires two parameters. The first is the value whose square root will be approximated. The second is the number of times to iterate the calculation yielding a better result.

1 def newtonSqrt(n, howmany):

2 approx = 0.5 \* n

3 f or i in range(howmany):

4 b etterapprox = 0.5 \* (approx + n/approx)

5 approx = betterapprox

6 return betterapprox

7

​8 print(newtonSqrt(100, 10))

9 print(newtonSqrt(4, 10))

10 print(newtonSqrt(1, 10))

11

Modify the program …

All three of the calls to newtonSqrt in the previous example produce the correct square root for the first parameter. However, were 10 iterations required to get the correct answer? Experiment with different values for the number of repetitions (the 10 on lines 8, 9, and 10). For each of these calls, find the smallest value for the number of repetitions that will produce the correct result.

Repeating more than the required number of times is a waste of computing resources. So definite iteration is not a good solution to this problem.

In general, Newton’s algorithm will eventually reach a point where the new approximation is no better than the previous. At that point, we could simply stop. In other words, by repeatedly applying this formula until the better approximation gets close enough to the previous one, we can write a function for computing the square root that uses the number of iterations necessary and no more.

This implementation, shown in codelens, uses a while condition to execute until the approximation is no longer changing. Each time through the loop we compute a “better” approximation using the formula described earlier. As long as the “better” is different, we try again.

The while statement shown above uses comparison of two floating point numbers in the condition. Since floating point numbers are themselves approximation of real numbers in mathematics, it is often better to compare for a result that is within some small threshold of the value you are looking for.

8.7. The Accumulator Pattern Revisited

Newton’s method to calculate square roots is an example of an algorithm that repeats as long as it can improve the result. It’s just a variation of our accumulator pattern. Many algorithms work this way and so require the use of indefinite iteration.

Here is another accumulator pattern program. It adds up the reciprocals of powers of two.

You may have studied this sequence in a math class and learned that the sum approaches but never reaches 2.0. That is true in theory. However, when we implement this summation in a program, we see something different.

1 def sumTo():

2 """ Return the sum of reciprocals of powers of 2 """

3

4 theSum = 0

5 aNumber = 0

6 while theSum < 2.0:

7 theSum = theSum + 1/2\*\*aNumber

8 aNumber = aNumber + 1

9

​10 return theSum

11

​12 print(sumTo())

13

​

Modify the program …

If the sum never reaches 2.0, the loop would never terminate. But the loop does stop! How many repetitions did it make before it stopped?

On line 9 (not indented), print the value of aNumber and you will see.

But why did it reach 2.0? Are those math teachers wrong?

SavvyPython

View:

Textbook

Social

Search

User

Scratch Activecode

Help

This Chapter

8.8. Other uses of while

8.8.1. Sentinel Values

Indefinite loops are much more common in the real world than definite loops.

If you are selling tickets to an event, you don’t know in advance how many tickets you will sell. You keep selling tickets as long as people come to the door and there’s room in the hall.

When the baggage crew unloads a plane, they don’t know in advance how many suitcases there are. They just keep unloading while there are bags left in the cargo hold. (Why your suitcase is always the last one is an entirely different problem.)

When you go through the checkout line at the grocery, the clerks don’t know in advance how many items there are. They just keep ringing up items as long as there are more on the conveyor belt.

Let’s implement the last of these in Python, by asking the user for prices and keeping a running total and count of items. When the last item is entered, the program gives the grand total, number of items, and average price. We’ll need these variables:

total - this will start at zero

count - the number of items, which also starts at zero

moreItems - a boolean that tells us whether more items are waiting; this starts as True

The pseudocode (code written half in English, half in Python) for the body of the loop looks something like this:

while moreItems

ask for price

add price to total

add one to count

This pseudocode has no option to set moreItems to False, so it would run forever. In a grocery store, there’s a little plastic bar that you put after your last item to separate your groceries from those of the person behind you; that’s how the clerk knows you have no more items. We don’t have a “little plastic bar” data type in Python, so we’ll do the next best thing: we will use a price of zero to mean “this is my last item.” In this program, zero is a sentinel value, a value used to signal the end of the loop. Here’s the code:

1

def checkout():

2

total = 0

3

count = 0

4

moreItems = True

5

while moreItems:

6

price = float(input('Enter price of item (0 when done): '))

7

if price != 0:

8

count = count + 1

9

total = total + price

10

print('Subtotal: $', total)

11

else:

12

moreItems = False

13

average = total / count

14

print('Total items:', count)

15

print('Total $', total)

16

print('Average price per item: $', average)

17

​

18

checkout()

19

​

Activity: 8.8.1.1 ActiveCode (ch07\_sentinel)

There are still a few problems with this program.

If you enter a negative number, it will be added to the total and count. Modify the code so that negative numbers give an error message instead (but don’t end the loop) Hint: elif is your friend.

If you enter zero the first time you are asked for a price, the loop will end, and the program will try to divide by zero. Use an if/else statement outside the loop to avoid the division by zero and tell the user that you can’t compute an average without data.

This program doesn’t display the amounts to two decimal places. In the next chapter you will see the String Format Method that will do the trick.

8.8.2. Validating Input

You can also use a while loop when you want to validate input; when you want to make sure the user has entered valid input for a prompt. Let’s say you want a function that asks a yes-or-no question. In this case, you want to make sure that the person using your program enters either a Y for yes or N for no (in either upper or lower case). Here is a program that uses a while loop to keep asking until it receives a valid answer. As a preview of coming attractions, it uses the upper() method which is described in String Methods to convert a string to upper case. When you run the following code, try typing something other than Y or N to see how the code reacts:

1 def get\_yes\_or\_no(message):

2 valid\_input = False

3 while not valid\_input:

4 answer = input(message)

5 answer = answer.upper() # convert to upper case

6 if answer == 'Y' or answer == 'N':

7 valid\_input = True

8 else:

9 print('Please enter Y for yes or N for no.')

10 return answer

11

​12 response = get\_yes\_or\_no('Do you like lima beans? Y)es or N)o: ')

13 if response == 'Y':

14 print('Great! They are very healthy.')

15 else:

16 print('Too bad. If cooked right, they are quite tasty.')

17

8.9. Algorithms Revisited

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.

On the other hand, understanding that hard problems can be solved by step-by-step algorithmic processes is one of the major simplifying breakthroughs that has had enormous benefits. So while the execution of the algorithm may be boring and may require no intelligence, algorithmic or computational thinking is having a vast impact. It is the process of designing algorithms that 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 a step-by-step mechanical algorithm.

8.10. Simple Tables

One of the things loops are good for is generating tabular data. Before computers were readily available, people had to calculate logarithms, sines and cosines, and other mathematical functions by hand. To make that easier, mathematics books contained long tables listing the values of these functions. Creating the tables was slow and boring, and they tended to be full of errors.

When computers appeared on the scene, one of the initial reactions was, “This is great! We can use the computers to generate the tables, so there will be no errors.” That turned out to be true (mostly) but shortsighted. Soon thereafter, computers and calculators were so pervasive that the tables became obsolete.

Well, almost. For some operations, computers use tables of values to get an approximate answer and then perform computations to improve the approximation. In some cases, there have been errors in the underlying tables, most famously in the table the Intel Pentium processor chip used to perform floating-point division.

Although a power of 2 table is not as useful as it once was, it still makes a good example of iteration. The following program outputs a sequence of values in the left column and 2 raised to the power of that value in the right column:

1 print("n", '\t', "2\*\*n") #table column headings

2 print("---", '\t', "-----")

3

​4 for x in range(13): # generate values for columns

5 print(x, '\t', 2 \*\* x)

6

​

The string '\t' represents a tab character. The backslash character in '\t' indicates the beginning of an escape sequence. Escape sequences are used to represent invisible characters like tabs and newlines. The sequence \n represents a newline.

An escape sequence can appear anywhere in a string. In this example, the tab escape sequence is the only thing in the string. How do you think you represent a backslash in a string?

As characters and strings are displayed on the screen, an invisible marker called the cursor keeps track of where the next character will go. After a print function is executed, the cursor normally goes to the beginning of the next line.

The tab character shifts the cursor to the right until it reaches one of the tab stops. Tabs are useful for making columns of text line up, as in the output of the previous program. Because of the tab characters between the columns, the position of the second column does not depend on the number of digits in the first column.

Check your understanding

iter-8-2: What is the difference between a tab ('\t') and a sequence of spaces?

A. A tab will line up items in a second column, regardless of how many characters were in the first column, while spaces will not.

B. There is no difference

C. A tab is wider than a sequence of spaces

D. You must use tabs for creating tables. You cannot use spaces.

8.11. 2-Dimensional Iteration: Image Processing

Two dimensional tables have both rows and columns. You have probably seen many tables like this if you have used a spreadsheet program. Another object that is organized in rows and columns is a digital image. In this section we will explore how iteration allows us to manipulate these images.

A digital image is a finite collection of small, discrete picture elements called pixels. These pixels are organized in a two-dimensional grid. Each pixel represents the smallest amount of picture information that is available. Sometimes these pixels appear as small “dots”.

Each image (grid of pixels) has its own width and its own height. The width is the number of columns and the height is the number of rows. We can name the pixels in the grid by using the column number and row number. However, it is very important to remember that computer scientists like to start counting with 0! This means that if there are 20 rows, they will be named 0,1,2, and so on through 19. This will be very useful later when we iterate using range.

8.11.1. The RGB Color Model

Each pixel of the image will represent a single color. The specific color depends on a formula that mixes various amounts of three basic colors: red, green, and blue. This technique for creating color is known as the RGB Color Model. The amount of each color, sometimes called the intensity of the color, allows us to have very fine control over the resulting color.

The minimum intensity value for a basic color is 0. For example if the red intensity is 0, then there is no red in the pixel. The maximum intensity is 255. This means that there are actually 256 different amounts of intensity for each basic color. Since there are three basic colors, that means that you can create 2563 distinct colors using the RGB Color Model.

Here are the red, green and blue intensities for some common colors. Note that “Black” is represented by a pixel having no basic color. On the other hand, “White” has maximum values for all three basic color components.

Color

Red

Green

Blue

Red

255

0

0

Green

0

255

0

Blue

0

0

255

White

255

255

255

Black

0

0

0

Yellow

255

255

0

Magenta

255

0

255

In order to manipulate an image, we need to be able to access individual pixels. This capability is provided by a module called image, provided in ActiveCode. See Image Processing on Your Own for ways to deal with images in standard Python. The image module defines two classes: Image and Pixel.

Each Pixel object has three attributes: the red intensity, the green intensity, and the blue intensity. A pixel provides three methods that allow us to ask for the intensity values. They are called getRed, getGreen, and getBlue. In addition, we can ask a pixel to change an intensity value using its setRed, setGreen, and setBlue methods.

Method Name

Example

Explanation

Pixel(r,g,b)

Pixel(20,100,50)

Create a new pixel with 20 red, 100 green, and 50 blue.

getRed()

r = p.getRed()

Return the red component intensity.

getGreen()

r = p.getGreen()

Return the green component intensity.

getBlue()

r = p.getBlue()

Return the blue component intensity.

setRed()

p.setRed(100)

Set the red component intensity to 100.

setGreen()

p.setGreen(45)

Set the green component intensity to 45.

setBlue()

p.setBlue(156)

Set the blue component intensity to 156.

In the example below, we first create a pixel with 45 units of red, 76 units of green, and 200 units of blue. We then print the current amount of red, change the amount of red, and finally, set the amount of blue to be the same as the current amount of green.

1 import image

2

​3 p = image.Pixel(45, 76, 200)

4 print(p.getRed())

5 p.setRed(66)

6 print(p.getRed())

7 p.setBlue(p.getGreen())

8 print(p.getGreen(), p.getBlue())

9

Check your understanding

iter-9-2: If you have a pixel whose RGB value is (50, 0, 0), what color will this pixel appear to be?

A. Dark red

B. Light red

C. Dark green

D. Light green

8.11.2. Image Objects

To access the pixels in a real image, we need to first create an Image object. Image objects can be created in two ways. First, an Image object can be made from the files that store digital images. The image object has an attribute corresponding to the width, the height, and the collection of pixels in the image.

It is also possible to create an Image object that is “empty”. An EmptyImage has a width and a height. However, the pixel collection consists of only “White” pixels.

We can ask an image object to return its size using the getWidth and getHeight methods. We can also get a pixel from a particular location in the image using getPixel and change the pixel at a particular location using setPixel.

The Image class is shown below. Note that the first two entries show how to create image objects. The parameters are different depending on whether you are using an image file or creating an empty image.

Method Name

Example

Explanation

Image(filename)

img = image.Image(“cy.png”)

Create an Image object from the file cy.png.

EmptyImage()

img = image.EmptyImage(100,200)

Create an Image object that has all “White” pixels

getWidth()

w = img.getWidth()

Return the width of the image in pixels.

getHeight()

h = img.getHeight()

Return the height of the image in pixels.

getPixel(col,row)

p = img.getPixel(35,86)

Return the pixel at column 35, row 86.

setPixel(col,row,p)

img.setPixel(100,50,mp)

Set the pixel at column 100, row 50 to be mp.

Consider the image shown below. Assume that the image is stored in a file called “luther.jpg”. Line 2 opens the file and uses the contents to create an image object that is referred to by img. Once we have an image object, we can use the methods described above to access information about the image or to get a specific pixel and check on its basic color intensities.

1 import image

2 img = image.Image("luther.jpg")

3

​4 print(img.getWidth())

5 print(img.getHeight())

6

​7 p = img.getPixel(45, 55)

8 print(p.getRed(), p.getGreen(), p.getBlue())

9

8.11.3. Image Processing and Nested Iteration

Image processing refers to the ability to manipulate the individual pixels in a digital image. In order to process all of the pixels, we need to be able to systematically visit all of the rows and columns in the image. The best way to do this is to use nested iteration.

Nested iteration simply means that we will place one iteration construct inside of another. We will call these two iterations the outer iteration and the inner iteration. To see how this works, consider the iteration below.

for i in range(5):

print(i)

We have seen this enough times to know that the value of i will be 0, then 1, then 2, and so on up to 4. The print will be performed once for each pass. However, the body of the loop can contain any statements including another iteration (another for statement). For example,

for i in range(5):

for j in range(3):

print(i, j)

The for i iteration is the outer iteration and the for j iteration is the inner iteration. Each pass through the outer iteration will result in the complete processing of the inner iteration from beginning to end. This means that the output from this nested iteration will show that for each value of i, all values of j will occur.

Here is the same example in activecode. Try it. Note that the value of i stays the same while the value of j changes. The inner iteration, in effect, is moving faster than the outer iteration.

1 for i in range(5):

2 for j in range(3):

3 print(i, j)

4

​

Another way to see this in more detail is to examine the behavior with codelens. Step through the iterations to see the flow of control as it occurs with the nested iteration. Again, for every value of i, all of the values of j will occur. You can see that the inner iteration completes before going on to the next pass of the outer iteration.

1 for i in range(5):

2 for j in range(3):

3 print(i, j)

Step 1 of 41

Our goal with image processing is to visit each pixel. We will use an iteration to process each row. Within that iteration, we will use a nested iteration to process each column. The result is a nested iteration, similar to the one seen above, where the outer for loop processes the rows, from 0 up to but not including the height of the image. The inner for loop will process each column of a row, again from 0 up to but not including the width of the image.

The resulting code will look like the following. We are now free to do anything we wish to each pixel in the image.

for row in range(img.getHeight()):

for col in range(img.getWidth()):

# do something with the pixel at position (col,row)

One of the easiest image processing algorithms will create what is known as a negative image. A negative image simply means that each pixel will be the opposite of what it was originally. But what does opposite mean?

In the RGB color model, we can consider the opposite of the red component as the difference between the original red and 255. For example, if the original red component was 50, then the opposite, or negative red value would be 255-50 or 205. In other words, pixels with a lot of red will have negatives with little red and pixels with little red will have negatives with a lot. We do the same for the blue and green as well.

The program below implements this algorithm using the previous image (luther.jpg). Run it to see the resulting negative image. Note that there is a lot of processing taking place and this may take a few seconds to complete. In addition, here are two other images that you can use (cy.png

1 import image

2

​3 img = image.Image("luther.jpg")

4 win = image.ImageWin(img.getWidth(), img.getHeight())

5 img.draw(win)

6 img.setDelay(1,15) # setDelay(0) turns off animation

7

​8 for row in range(img.getHeight()):

9 for col in range(img.getWidth()):

10 p = img.getPixel(col, row)

11

​12 newred = 255 - p.getRed()

13 newgreen = 255 - p.getGreen()

14 newblue = 255 - p.getBlue()

15

​16 newpixel = image.Pixel(newred, newgreen, newblue)

17

​18 img.setPixel(col, row, newpixel)

19

​20 img.draw(win)

21 win.exitonclick()

22

​

Let’s take a closer look at the code. After importing the image module, we create an image object called img that represents a typical digital photo. we will update each pixel in this image from top to bottom, left to right, which you should be able to observe. You can change the values in setDelay to make the program progress faster or slower.

Lines 8 and 9 create the nested iteration that we discussed earlier. This allows us to process each pixel in the image. Line 10 gets an individual pixel.

Lines 12-14 create the negative intensity values by extracting the original intensity from the pixel and subtracting it from 255. Once we have the newred, newgreen, and newblue values, we can create a new pixel (Line 15).

Finally, we need to replace the old pixel with the new pixel in our image. It is important to put the new pixel into the same location as the original pixel that it came from in the digital photo.

Try to change the program above so that the outer loop iterates over the columns and the inner loop iterates over the rows. We still create a negative image, but you can see that the pixels update in a very different order.

Other pixel manipulation

There are a number of different image processing algorithms that follow the same pattern as shown above. Namely, take the original pixel, extract the red, green, and blue intensities, and then create a new pixel from them. The new pixel is inserted into an empty image at the same location as the original.

For example, you can create a gray scale pixel by averaging the red, green and blue intensities and then using that value for all intensities.

From the gray scale you can create black white by setting a threshold and selecting to either insert a white pixel or a black pixel into the empty image.

You can also do some complex arithmetic and create interesting effects, such as Sepia Tone

You have just passed a very important point in your study of Python programming. Even though there is much more that we will do, you have learned all of the basic building blocks that are necessary to solve many interesting problems. From an algorithm point of view, you can now implement selection and iteration. You can also solve problems by breaking them down into smaller parts, writing functions for those parts, and then calling the functions to complete the implementation. What remains is to focus on ways that we can better represent our problems in terms of the data that we manipulate. We will now turn our attention to studying the main data collections provided by Python.

Check your understanding

for i in range(3):

for j in range(2):

print(i, j)

a.

0 0

0 1

1 0

1 1

2 0

2 1

b.

0 0

1 0

2 0

0 1

1 1

2 1

c.

0 0

0 1

0 2

1 0

1 1

1 2

d.

0 1

0 1

0 1

A. Output a

B. Output b

C. Output c

D. Output d

8.12. Image Processing on Your Own

If you want to try some image processing on your own, outside of the textbook you can do so using the cImage module. The easiest way to get this is to run the command pip install cImage from the command line. You can also download cImage.py from The github page . If you get the file from github, put cImage.py in the same folder as your program you can then do the following to be fully compatible with the code in this book.

import cImage as image

img = image.Image("myfile.gif")

Note

One important caveat about using cImage.py is that it will only work with GIF files unless you also install the Python Image Library. Don’t worry if you pip install cImage it will automatically take care of this for you. Otherwise, the easiest version to install is called Pillow. If you have the pip command installed on your computer this is really easy to install, with pip install pillow otherwise you will need to follow the instructions on the Python Package Index page. With Pillow installed you will be able to use almost any kind of image that you download.

8.13. Glossary

algorithm

A step-by-step process for solving a category of problems.

body

The indented statements after a heading ending in a colon, for instance after a for-loop heading.

counter

A variable used to count something, usually initialized to zero and incremented in the body of a loop.

cursor

An invisible marker that keeps track of where the next character will be printed.

definite iteration

A loop where we have an upper bound on the number of times the body will be executed. Definite iteration is usually best coded as a for loop.

escape sequence

An escape character, \, followed by one or more printable characters used to designate a nonprintable character.

generalize

To replace something unnecessarily specific (like a constant value) with something appropriately general (like a variable or parameter). Generalization makes code more versatile, more likely to be reused, and sometimes even easier to write.

infinite loop

A loop in which the terminating condition is never satisfied.

indefinite iteration

A loop where we just need to keep going until some condition is met. A while statement is used for this case.

iteration

Repeated execution of a set of programming statements.

loop

A statement or group of statements that execute repeatedly until a terminating condition is satisfied.

loop variable

A variable used as part of the terminating condition of a loop.

nested loop

A loop inside the body of another loop.

newline

A special character that causes the cursor to move to the beginning of the next line.

reassignment

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

tab

A special character that causes the cursor to move to the next tab stop on the current line.

SavvyPython

View:

Textbook

Social

Search

User

Scratch Activecode

Help

This Chapter

8.14. Exercises

Add a print statement to Newton’s sqrt function that prints out better each time it is calculated. Call your modified function with 25 as an argument and record the results.

Write a function print\_triangular\_numbers(n) that prints out the first n triangular numbers. A call to print\_triangular\_numbers(5) would produce the following output:

1 1

2 3

3 6

4 10

5 15

(hint: use a web search to find out what a triangular number is.)

Write a function, is\_prime, that takes a single integer argument and returns True when the argument is a prime number and False otherwise.

Modify the walking turtle program so that rather than a 90 degree left or right turn the angle of the turn is determined randomly at each step.

Modify the turtle walk program so that you have two turtles each with a random starting location. Keep the turtles moving until one of them leaves the screen.

Modify the previous turtle walk program so that the turtle turns around when it hits the wall or when one turtle collides with another turtle (when the positions of the two turtles are closer than some small number).

Write a function to remove all the red from an image.

Write a function to convert the image to grayscale.

Write a function to convert an image to black and white.

Sepia Tone images are those brownish colored images that may remind you of times past. The formula for creating a sepia tone is as follows:

newR = (R × 0.393 + G × 0.769 + B × 0.189)

newG = (R × 0.349 + G × 0.686 + B × 0.168)

newB = (R × 0.272 + G × 0.534 + B × 0.131)

Write a function to convert an image to sepia tone. Hint: Remember that rgb values must be integers between 0 and 255.

Write a function to uniformly enlarge an image by a factor of 2 (double the size).

After you have scaled an image too much it looks blocky. One way of reducing the blockiness of the image is to replace each pixel with the average values of the pixels around it. This has the effect of smoothing out the changes in color. Write a function that takes an image as a parameter and smooths the image. Your function should return a new image that is the same as the old but smoothed.

Write a general pixel mapper function that will take an image and a pixel mapping function as parameters. The pixel mapping function should perform a manipulation on a single pixel and return a new pixel.

When you scan in images using a scanner they may have lots of noise due to dust particles on the image itself or the scanner itself, or the images may even be damaged. One way of eliminating this noise is to replace each pixel by the median value of the pixels surrounding it.

Research the Sobel edge detection algorithm and implement it.