Pictures and Loops

In the previous section we saw how to change the red, green, and blue values of one pixel. In some applications (such as turning an image into a grayscale image) we may want to change the color of every pixel in the image, or in other applications (such as removing red eye), we may want to change the color of pixels in a small region of the image. If we have 640 x 480 = 307,200 pixels, do you want to type in 307,200 individual statements? Don't worry, we're not going to do this! We are going to use a loop statement to do this. A loop is a way to tell the computer to do the same thing (or almost the same thing) over and over again. We want to do the same thing, but to a different pixel each time.

A `for` loop will execute a block of commands that you specify for each item in a list that you provide. As part of the loop statement, you will specify a variable that will get the value of a different item from your list each time the group of commands executes. The list is an ordered collection of data – it could be numbers, strings, pixels, or many other different kinds of collections.

A `for` loop in Python looks like the following:

```
for ______________ in  ______________:
    # do some cool stuff here
    # do some more cool stuff here
```

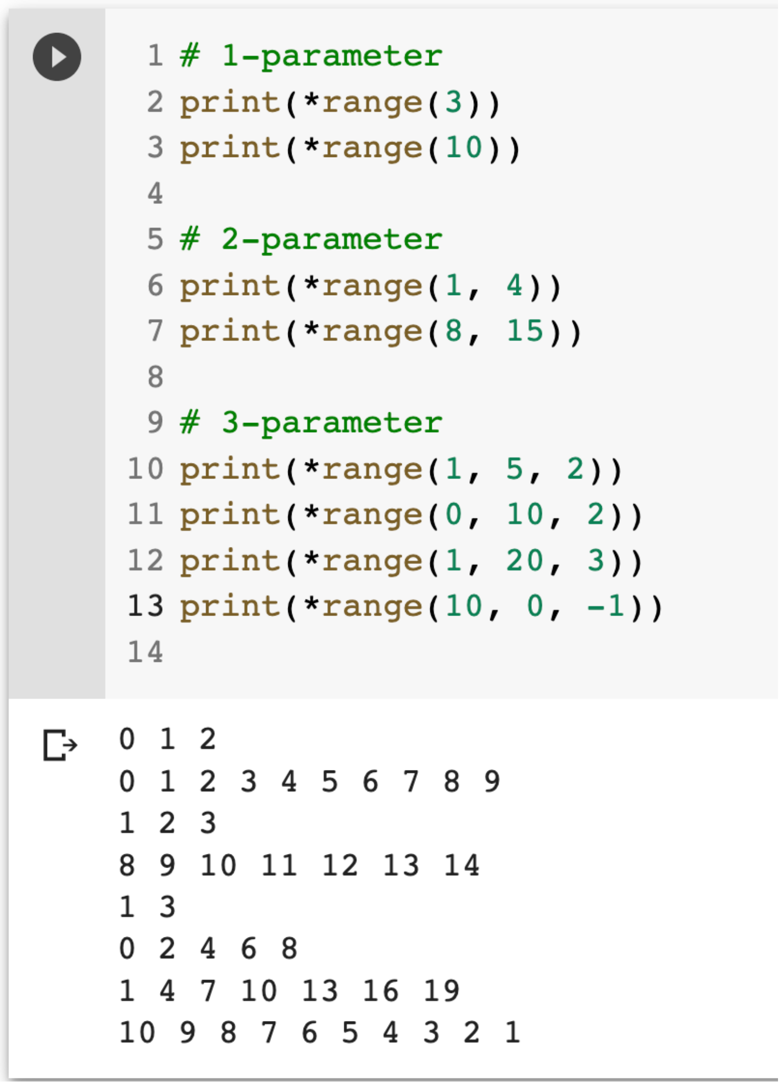
The very first part of this statement is the keyword `for`. The first blank (after the keyword `for`) will be filled in with your choice of a variable name, typically a name that represents the elements in your list. This variable is followed by the keyword `in`. The second blank (after the keyword `in`) will be filled in with the list for which you want to repeat the code with. At the end of the line you *must* have a colon, to indicate that what comes next is the block of code you want to repeat. (This is a Python rule!) The lines of code that are to be repeated should all be indented and indented the same amount.

The `range` function in Python is a commonly used function to get a list of numbers. When working with images, we will use this function to get the range of possible values for the coordinates of the pixels we want to modify.

There are three ways to use the `range` function in Python – with one, two, or three parameters. Let's look at examples to show how these different versions of `range` work.

Example 1: The `range` function

Examples of `range` function use



```
# 1-parameter
print(*range(3))
print(*range(10))

# 2-parameter
print(*range(1, 4))
print(*range(8, 15))

# 3-parameter
print(*range(1, 5, 2))
print(*range(0, 10, 2))
print(*range(1, 20, 3))
print(*range(10, 0, -1))

```

```
0 1 2
0 1 2 3 4 5 6 7 8 9
1 2 3
8 9 10 11 12 13 14
1 3
0 2 4 6 8
1 4 7 10 13 16 19
10 9 8 7 6 5 4 3 2 1
```

The 1-parameter `range` function returns a list of integers that starts at 0 and goes up to the value that was input but does not include it. The 2-parameter `range` function returns a list of integers that starts at the first number indicated and goes up to, but does not include, the second number. The 3-parameter `range` function works similarly to the 2-parameter `range` function, except that the third parameter specifies what the step is (*i.e.*, what to count by). Notice that in the last example of the 3-parameter `range` function, the range starts at 10 and goes *down* to 0. The step is negative to make the count go backwards.

(Note: Since the range function gives us a list of values, we put the * in front of it when printing so that it will print all of the values in the list.)

The following function to calculate the sum of the first *n* integers uses the range function to give a list of numbers to be used in a loop:

Example 2: Function to sum first n integers

```
def sumIntegers(n):
   sum = 0
   for j in range(1, n+1):
      sum = sum + j
   return sum
```

How does this work? The variable *sum* will hold the accumulated total as we go through the loop; it starts out with the value 0. The first time we go through the loop, *j* takes on the value 1. It then gets added to the value in *sum* (currently 0), and then the result gets stored back in *sum*. So *sum* now has the value 1. The next time through the loop, *j* takes on the next value in the list, which is 2. This value gets added to what is in *sum* (currently 1), and gets stored back in *sum*. So *sum* now has the value 3. This continues until the last iteration of the loop, when *x* takes on the value of *n*. This gets added to the value in *sum*, and stored back in *sum*. The variable *sum* then contains the value of 1+2+3+…+*n*.

Now how can we use this `range` function to do anything with pictures?

In a previous activity, we learned how to draw shapes on an image. If we wanted to repeatedly draw a shape to make a pattern, we can do this with a loop. In the next example, the function will draw horizontal lines across an image, from top to bottom.

Example 3: Drawing lines

```
def drawLines(picture):
  # Duplicate the original picture
  newPic = picture.copy()
  d = ImageDraw.Draw(newPic)

  # Draw the lines
  for y in range(0, newPic.height, 10):
    d.line([0, y, newPic.width -1, y], 'blue')

  # Return the new picture
  return newPic
```

How does this work? We begin by copying our original picture so that we work with a copy, not the original. Then we create a loop that will let the *y*- coordinate vary from the top of the picture to the bottom, jumping by 10. The first line gets drawn at row 0. The second line get drawn at row 10, the third at row 20, and so on.

Another way we can make use of the `range` function is to use it to specify a range of pixel coordinates for which we would like to change the pixel colors. The basic idea is that we will use it in our loop statements to indicate which coordinates of pixels we would like to modify. Consider the following example that will change the color of the first 20 pixels in the 5th row of the picture to black.

Example 4: Change some pixels to black

```
def makeSomeBlack(picture):
  # Duplicate the original picture
  newPic = picture.copy()

  # Change the pixels
  for x in range(20):
    newPic.putpixel((x, 5), (0, 0, 0))

  # Return the new picture
  return newPic
```

So how does this work? We begin by copying our original picture so that we work with a copy, not the original. Then we loop through the first 20 integers (0 through 19). Each time through the loop, we get one pixel, with coordinates (x, 5) and change the color of that pixel to black. The first time through the loop, we get and change pixel (0,5). The second time through the loop, we get and change pixel (1, 5). The next time through the loop, we get and change pixel (2, 5). We continue this until the last time through the loop, when we get and change pixel (19, 5).

Suppose we wanted to change the entire row to black. We only need to modify the parameter in the `range` function of the example so that it ends at the last pixel in the row (instead of the 20th pixel in the row). If our picture is 640 x 480, the last pixel in row 5 is (639, 5). We will use the `width` attribute to specify what this should be.

Example 5: Change pixels in row 5 to black

```
def changeRow5(picture):
  # Duplicate the original picture
  newPic = picture.copy()

  # Change the pixels
  for x in range(newPic.width):
    newPic.putpixel((x, 5), (0, 0, 0))

  # Return the new picture
  return newPic
```

Notice that the only difference between this example and Example 4 is the parameter in the `range` function.

Let's extend this example so that we now change the color of *every* pixel in the picture to black. We could write functions similar to `changeRow5` such as `changeRow0`, `changeRow1`, `changeRow2`, etc. This is really too much work! Instead of doing this, we will actually use a loop *inside* of a loop (nested loops). If we want to change all of the pixels in all of the rows and all of the columns, we will use one loop to color each row like we did with `changeRow5`, and then we will nest that loop inside of a loop that lets us go through each of the rows. The code would look like the following:

Example 5: Color all pixels using nested loops

```
def colorAllBlack(picture):
  # Duplicate the original picture
  newPic = picture.copy()

  for y in range(newPic.height):
    # Change the pixels in row y
    for x in range(newPic.width):
      newPic.putpixel((x,y),(0, 0, 0))

  # Return the new picture
  return newPic
```

How does this work? For *each* value of *y* in the list [0, …, h-1] (where *h* is the height of the picture), we go through the entire loop for *x*. So *y* starts at 0. Then *x* starts at 0 and we modify pixel (0, 0). Then *y* stays at 0 and *x* becomes 1. We then modify pixel (1, 0). We modify all of the pixels (2, 0), (3, 0), …, (w-1, 0) before *y* changes value. After we modify pixel (w-1, 0), *y* becomes 1, and we modify pixel (0, 1). We then modify all the pixels (1, 1), (2, 1), (3, 1), …(w-1, 1). Then *y* will become 2 and we will modify pixels (0, 2), (1, 2), (2, 2), (3, 2), …etc. We continue this until *y* becomes h-1 and we modify the pixels in the bottom row of the picture: (0, h-1), (1, h-1), (2, h-1), …, (w-1, h-1). The value of *y* tells us which row we are working with, and the values of *x* let us move right across the picture.

We should now practice these ideas of using the range function to get coordinates of pixels to modify by working through the next mini-lab.

Activity: Functions and Loops