pdf995

Create PDF files quickly and easily!

The pdf995 suite of products - Pdf995, PdfEdit995, and Signature995 - is a complete solution for your document publishing needs. It provides ease of use, flexibility in format, and industry-standard security- and all at no cost to you.

Pdf995 makes it easy and affordable to create professional-quality documents in the popular PDF file format. Its easy-to-use interface helps you to create PDF files by simply selecting the "print" command from any application, creating documents which can be viewed on any computer with a PDF viewer. Pdf995 supports network file saving, fast user switching on XP, Citrix/Terminal Server, custom page sizes and large format printing. Pdf995 is a printer driver that works with any Postscript to PDF converter. The pdf995 printer driver and a free Converter are available for easy download.

PdfEdit995 offers a wealth of additional functionality, such as: combining documents into a single PDF; automatic link insertion; hierarchical bookmark insertion; PDF conversion to HTML or DOC (text only); integration with Word toolbar with automatic table of contents and link generation; autoattach to email; stationery and stamping.

Signature995 offers state-of-the-art security and encryption to protect your documents and add digital signatures.

The Pdf995 Suite offers the following features, all at no cost:

Automatic insertion of embedded links
Hierarchical Bookmarks
Support for Digital Signatures
Support for Triple DES encryption
Append and Delete PDF Pages
Batch Print from Microsoft Office
Asian and Cyrillic fonts
Integration with Microsoft Word toolbar
PDF Stationery
Combining multiple PDF's into a single PDF
Three auto-name options to bypass Save As dialog
Imposition of Draft/Confidential stamps
Support for large format architectural printing
Convert PDF to JPEG, TIFF, BMP, PCX formats
Convert PDF to HTML and Word DOC conversion
Convert PDF to text
Automatic Table of Contents generation
Support for XP Fast User Switching and multiple user sessions
Standard PDF Encryption (restricted printing, modifying, copying text and images)
Support for Optimized PDF
Support for custom page sizes

Option to attach PDFs to email after creation
Automatic text summarization of PDF documents
Easy integration with document management and Workflow systems
n-Up printing
Automatic page numbering
Simple Programmers Interface
Option to automatically display PDFs after creation
Custom resizing of PDF output
Configurable Font embedding
Support for Citrix/Terminal Server
Support for Windows 2003 Server
Easy PS to PDF processing
Specify PDF document properties
Control PDF opening mode
Can be configured to add functionality to Acrobat Distiller
Free: Creates PDFs without annoying watermarks
Free: Fully functional, not a trial and does not expire
Over 5 million satisfied customers
Over 1000 Enterprise Customers worldwide

Please visit us at www.pdf995.com to learn more.

This document illustrates several features of the Pdf995 Suite of Products.

Introduction

The **Virtual Reality Modeling Language** (VRML) is a language for describing multi-participant interactive simulations -- virtual worlds networked via the global Internet and hyperlinked with the World Wide Web. All aspects of virtual world display, interaction and internetworking can be specified using VRML. It is the intention of its designers that VRML become the standard language for interactive simulation within the World Wide Web.

The first version of VRML allows for the creation of virtual worlds with limited interactive behavior. These worlds can contain objects which have hyperlinks to other worlds, HTML documents or other valid MIME types. When the user selects an object with a hyperlink, the appropriate MIME viewer is launched. When the user selects a link to a VRML document from within a correctly configured WWW browser, a VRML viewer is launched. Thus VRML viewers are the perfect companion applications to standard WWW browsers for navigating and visualizing the Web. Future versions of VRML will allow for richer behaviors, including animations, motion physics and real-time multi-user interaction.

This document specifies the features and syntax of Version 1.0 of VRML.

VRML Mission Statement

The history of the development of the Internet has had three distinct phases; first, the development of the TCP/IP infrastructure which allowed documents and data to be stored in a proximally independent way; that is, Internet provided a layer of abstraction between data sets and the hosts which manipulated them. While this abstraction was useful, it was also confusing; without any clear sense of "what went where", access to Internet was restricted to the class of sysops/net surfers who could maintain internal cognitive maps of the data space.

Next, Tim Berners-Lee's work at CERN, where he developed the hypermedia system known as **World Wide Web**, added another layer of abstraction to the existing structure. This abstraction provided an "addressing" scheme, a unique identifier (the Universal Resource Locator), which could tell anyone "where to go and how to get there" for any piece of data within the Web. While useful, it lacked dimensionality; there's no *there* there within the web, and the only type of navigation permissible (other than surfing) is by direct reference. In other words, I can only tell you how to get to the VRML Forum home page by saying, "http://www.wired.com/", which is not human-centered data. In

fact, I need to make an effort to remember it at all. So, while the World Wide Web provides a retrieval mechanism to complement the existing storage mechanism, it leaves a lot to be desired, particularly for human beings.

Finally, we move to "perceptualized" Internetworks, where the data has been sensualized, that is, rendered sensually. If something is represented sensually, it is possible to make sense of it. VRML is an attempt (how successful, only time and effort will tell) to place humans at the center of the Internet, ordering its universe to our whims. In order to do that, the most important single element is a standard that defines the particularities of perception. Virtual Reality Modeling Language is that standard, designed to be a *universal description language for multi-participant simulations*.

These three phases, storage, retrieval, and perceptualization are analogous to the human process of consciousness, as expressed in terms of semantics and cognitive science. Events occur and are recorded (memory); inferences are drawn from memory (associations), and from sets of related events, maps of the universe are created (cognitive perception). What is important to remember is that the map is **not** the territory, and we should avoid becoming trapped in any single representation or world-view. Although we need to *design to avoid disorientation*, we should always push the envelope in the kinds of experience we can bring into manifestation!

This document is the living proof of the success of a process that was committed to being open and flexible, responsive to the needs of a growing Web community. Rather than re-invent the wheel, we have adapted an existing specification (Open Inventor) as the basis from which our own work can grow, saving years of design work and perhaps many mistakes. Now our real work can begin; that of rendering our noospheric space.

History

VRML was conceived in the spring of 1994 at the first annual World Wide Web Conference in Geneva, Switzerland. Tim Berners-Lee and Dave Raggett organized a Birds-of-a-Feather (BOF) session to discuss Virtual Reality interfaces to the World Wide Web. Several BOF attendees described projects already underway to build three dimensional graphical visualization tools which interoperate with the Web. Attendees agreed on the need for these tools to have a common language for specifying 3D scene description and WWW hyperlinks -- an analog of HTML for virtual reality. The term Virtual Reality Markup Language (VRML) was coined, and the group resolved to begin specification work after the conference. The word 'Markup' was later changed to 'Modeling' to reflect the graphical nature of VRML.

Shortly after the Geneva BOF session, the www-vrml mailing list was created to discuss the development of a specification for the first version of VRML. The response to the list invitation was overwhelming: within a week, there were over a thousand members. After an initial settling-in period, list moderator Mark Pesce of Labyrinth Group announced his intention to have a draft version of the specification ready by the WWW Fall 1994 conference, a mere five months away. There was general agreement on the list that, while this schedule was aggressive, it was achievable provided that the requirements for the first version were not too ambitious and that VRML could be adapted from an existing solution. The list quickly agreed upon a set of requirements for the first version, and began a search for technologies which could be adapted to fit the needs of VRML.

The search for existing technologies turned up a several worthwhile candidates. After much deliberation the list came to a consensus: the Open Inventor ASCII File Format from Silicon Graphics, Inc. The Inventor File Format supports complete descriptions of 3D scenes with polygonally rendered objects, lighting, materials, ambient properties and realism effects. A subset of the Inventor File Format, with extensions to support networking, forms the basis of VRML. Gavin Bell of Silicon Graphics has adapted the Inventor File Format for VRML, with design input from the mailing list. SGI has publicly stated that the file format is available for use in the open market, and have contributed a file format parser into the public domain to bootstrap VRML viewer development.

VRML
VIRTUAL REALITY MODELING LANGUAGE

## A Graphical Representation of Inverse VRML Uptake

Change the number in red below to adjust for download rate and/or bandwidth.

| 1 | The number 1 represents an engineer with an "average" cube *

CF	Min	fsw	Air	EANx 32%	EANx 36%
80.0	149.12	0			
61.4	114.43	10			
49.8	92.846	20			
41.9	78.102	30	180		
36.2	67.402	40	120		
31.8	59.275	50	80.0	147.0	192.0
28.4	52.9	60	57.0	92.0	123.0
25.6	47.774	70	40.0	65.0	79.0
23.4	43.543	80	30.0	49.0	59.0
21.5	40.001	90	24.0	37.0	45.0
19.9	37	100	19.0	30.0	35.0
18.5	34.409	110	16.0	25.0	29.0
17.3	32.154	120	13.0	20.0	n/a
16.2	30.178	130	10.0	17.0	n/a
15.1	28.202	140	8.0	n/a	n/a

17

MANIPULATING IMAGES
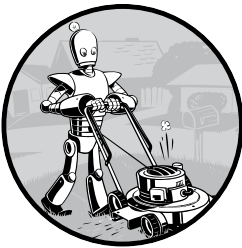
If you have a digital camera or even if you just upload photos from your phone to Facebook, you probably cross paths with digital image files all the time. You may know how to use basic graphics software, such as Microsoft Paint or Paintbrush, or even more advanced applications such as Adobe Photoshop. But if you need to edit a massive number of images, editing them by hand can be a lengthy, boring job.

Enter Python. Pillow is a third-party Python module for interacting with image files. The module has several functions that make it easy to crop, resize, and edit the content of an image. With the power to manipulate images the same way you would with software such as Microsoft Paint or Adobe Photoshop, Python can automatically edit hundreds or thousands of images with ease.

Computer Image Fundamentals

In order to manipulate an image, you need to understand the basics of how computers deal with colors and coordinates in images and how you can work with colors and coordinates in Pillow. But before you continue, install the `pillow` module. See Appendix A for help installing third-party modules.

Colors and RGBA Values

Computer programs often represent a color in an image as an *RGBA value*. An RGBA value is a group of numbers that specify the amount of red, green, blue, and *alpha* (or transparency) in a color. Each of these component values is an integer from `0` (none at all) to `255` (the maximum). These RGBA values are assigned to individual *pixels*; a pixel is the smallest dot of a single color the computer screen can show (as you can imagine, there are millions of pixels on a screen). A pixel's RGB setting tells it precisely what shade of color it should display. Images also have an alpha value to create RGBA values. If an image is displayed on the screen over a background image or desktop wallpaper, the alpha value determines how much of the background you can "see through" the image's pixel.

In Pillow, RGBA values are represented by a tuple of four integer values. For example, the color red is represented by (255, 0, 0, 255). This color has the maximum amount of red, no green or blue, and the maximum alpha value, meaning it is fully opaque. Green is represented by (0, 255, 0, 255), and blue is (0, 0, 255, 255). White, the combination of all colors, is (255, 255, 255, 255), while black, which has no color at all, is (0, 0, 0, 255).

If a color has an alpha value of `0`, it is invisible, and it doesn't really matter what the RGB values are. After all, invisible red looks the same as invisible black.

Pillow uses the standard color names that HTML uses. Table 17-1 lists a selection of standard color names and their values.

Table 17-1: Standard Color Names and Their RGBA Values

Name	RGBA value	Name	RGBA value
White	(255, 255, 255, 255)	Red	(255, 0, 0, 255)
Green	(0, 128, 0, 255)	Blue	(0, 0, 255, 255)
Gray	(128, 128, 128, 255)	Yellow	(255, 255, 0, 255)
Black	(0, 0, 0, 255)	Purple	(128, 0, 128, 255)

Pillow offers the `ImageColor.getcolor()` function so you don't have to memorize RGBA values for the colors you want to use. This function takes a color name string as its first argument, and the string `'RGBA'` as its second argument, and it returns an RGBA tuple.

To see how this function works, enter the following into the interactive shell:

```
❶ >>> from PIL import ImageColor
❷ >>> ImageColor.getcolor('red', 'RGBA')
(255, 0, 0, 255)
❸ >>> ImageColor.getcolor('RED', 'RGBA')
(255, 0, 0, 255)
>>> ImageColor.getcolor('Black', 'RGBA')
(0, 0, 0, 255)
>>> ImageColor.getcolor('chocolate', 'RGBA')
(210, 105, 30, 255)
>>> ImageColor.getcolor('CornflowerBlue', 'RGBA')
(100, 149, 237, 255)
```

First, you need to import the `ImageColor` module from PIL ❶ (not from Pillow; you'll see why in a moment). The color name string you pass to `ImageColor.getcolor()` is case insensitive, so passing `'red'` ❷ and passing `'RED'` ❸ give you the same RGBA tuple. You can also pass more unusual color names, like `'chocolate'` and `'Cornflower Blue'`.

Pillow supports a huge number of color names, from `'aliceblue'` to `'whitesmoke'`. You can find the full list of more than 100 standard color names in the resources at *http://nostarch.com/automatestuff/*.

Coordinates and Box Tuples

Image pixels are addressed with x- and y-coordinates, which respectively specify a pixel's horizontal and vertical location in an image. The *origin* is the pixel at the top-left corner of the image and is specified with the notation (0, 0). The first zero represents the x-coordinate, which starts at zero at the origin and increases going from left to right. The second zero represents the y-coordinate, which starts at zero at the origin and increases going

down the image. This bears repeating: y-coordinates increase going downward, which is the opposite of how you may remember y-coordinates being used in math class. Figure 17-1 demonstrates how this coordinate system works.

Many of Pillow's functions and methods take a *box tuple* argument. This means Pillow is expecting a tuple of four integer coordinates that represent a rectangular region in an image. The four integers are, in order, as follows:

- *Left*: The x-coordinate of the leftmost edge of the box.
- *Top*: The y-coordinate of the top edge of the box.
- *Right*: The x-coordinate of one pixel to the right of the rightmost edge of the box. This integer must be greater than the left integer.
- *Bottom*: The y-coordinate of one pixel lower than the bottom edge of the box. This integer must be greater than the top integer.

Note that the box includes the left and top coordinates and goes up to but does not include the right and bottom coordinates. For example, the box tuple (3, 1, 9, 6) represents all the pixels in the black box in Figure 17-2.
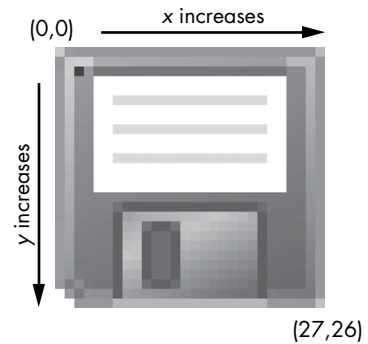

Figure 17-1: The x- and y-coordinates of a 27×26 image of some sort of ancient data storage device
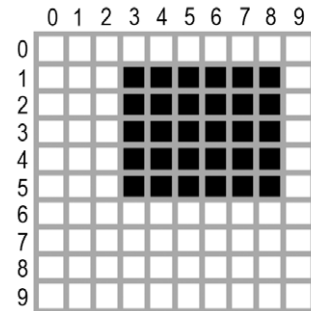

Figure 17-2: The area represented by the box tuple (3, 1, 9, 6)

Manipulating Images with Pillow

Now that you know how colors and coordinates work in Pillow, let's use Pillow to manipulate an image. Figure 17-3 is the image that will be used for all the interactive shell examples in this chapter. You can download it from *http://nostarch.com/automatestuff/*.

Once you have the image file *Zophie.png* in your current working directory, you'll be ready to load the image of Zophie into Python, like so:

```
>>> from PIL import Image
>>> catIm = Image.open('zophie.png')
```

Automate the Boring Stuff with Python: Practical Programming for Total Beginners
© 2015 Al Sweigart

Figure 17-3: My cat Zophie. The camera adds 10 pounds (which is a lot for a cat).

To load the image, you import the `Image` module from Pillow and call `Image.open()`, passing it the image's filename. You can then store the loaded image in a variable like `CatIm`. The module name of Pillow is `PIL` to make it backward compatible with an older module called Python Imaging Library, which is why you must run `from PIL import Image` instead of `from Pillow import Image`. Because of the way Pillow's creators set up the `pillow` module, you must use the `from PIL import Image` form of import statement, rather than simply `import PIL`.

If the image file isn't in the current working directory, change the working directory to the folder that contains the image file by calling the `os.chdir()` function.

```
>>> import os
>>> os.chdir('C:\\folder_with_image_file')
```

The `Image.open()` function returns a value of the `Image` object data type, which is how Pillow represents an image as a Python value. You can load an `Image` object from an image file (of any format) by passing the `Image.open()` function a string of the filename. Any changes you make to the `Image` object can be saved to an image file (also of any format) with the `save()` method. All the rotations, resizing, cropping, drawing, and other image manipulations will be done through method calls on this `Image` object.

To shorten the examples in this chapter, I'll assume you've imported Pillow's `Image` module and that you have the Zophie image stored in a variable named *catIm*. Be sure that the *zophie.png* file is in the current working directory so that the `Image.open()` function can find it. Otherwise, you will also have to specify the full absolute path in the string argument to `Image.open()`.

Working with the Image Data Type

An `Image` object has several useful attributes that give you basic information about the image file it was loaded from: its width and height, the filename, and the graphics format (such as JPEG, GIF, or PNG).

For example, enter the following into the interactive shell:

```
>>> from PIL import Image
>>> catIm = Image.open('zophie.png')
>>> catIm.size
❶ (816, 1088)
❷ >>> width, height = catIm.size
❸ >>> width
816
❹ >>> height
1088
>>> catIm.filename
'zophie.png'
>>> catIm.format
'PNG'
>>> catIm.format_description
'Portable network graphics'
❺ >>> catIm.save('zophie.jpg')
```

After making an `Image` object from *Zophie.png* and storing the `Image` object in `catIm`, we can see that the object's `size` attribute contains a tuple of the image's width and height in pixels ❶. We can assign the values in the tuple to `width` and `height` variables ❷ in order to access with width ❸ and height ❹ individually. The `filename` attribute describes the original file's name. The `format` and `format_description` attributes are strings that describe the image format of the original file (with `format_description` being a bit more verbose).

Finally, calling the save() method and passing it `'zophie.jpg'` saves a new image with the filename *zophie.jpg* to your hard drive ❺. Pillow sees that the file extension is *.jpg* and automatically saves the image using the JPEG image format. Now you should have two images, *zophie.png* and *zophie.jpg*, on your hard drive. While these files are based on the same image, they are not identical because of their different formats.

Pillow also provides the Image.new() function, which returns an `Image` object—much like Image.open(), except the image represented by Image.new()'s object will be blank. The arguments to Image.new() are as follows:

- The string `'RGBA'`, which sets the color mode to RGBA. (There are other modes that this book doesn't go into.)
- The size, as a two-integer tuple of the new image's width and height.

- The background color that the image should start with, as a four-integer tuple of an RGBA value. You can use the return value of the `ImageColor.getcolor()` function for this argument. Alternatively, `Image.new()` also supports just passing the string of the standard color name.

For example, enter the following into the interactive shell:

```
>>> from PIL import Image
❶ >>> im = Image.new('RGBA', (100, 200), 'purple')
>>> im.save('purpleImage.png')
❷ >>> im2 = Image.new('RGBA', (20, 20))
>>> im2.save('transparentImage.png')
```

Here we create an `Image` object for an image that's 100 pixels wide and 200 pixels tall, with a purple background ❶. This image is then saved to the file *purpleImage.png*. We call `Image.new()` again to create another `Image` object, this time passing (20, 20) for the dimensions and nothing for the background color ❷. Invisible black, (`0, 0, 0, 0`), is the default color used if no color argument is specified, so the second image has a transparent background; we save this 20×20 transparent square in *transparentImage.png*.

Cropping Images

Cropping an image means selecting a rectangular region inside an image and removing everything outside the rectangle. The `crop()` method on `Image` objects takes a box tuple and returns an `Image` object representing the cropped image. The cropping does not happen in place—that is, the original `Image` object is left untouched, and the `crop()` method returns a new `Image` object. Remember that a boxed tuple—in this case, the cropped section—includes the left column and top row of pixels but only goes up to and does *not* include the right column and bottom row of pixels.

Enter the following into the interactive shell:

```
>>> croppedIm = catIm.crop((335, 345, 565, 560))
>>> croppedIm.save('cropped.png')
```

This makes a new `Image` object for the cropped image, stores the object in `croppedIm`, and then calls `save()` on `croppedIm` to save the cropped image in *cropped.png*. The new file *cropped.png* will be created from the original image, like in Figure 17-4.
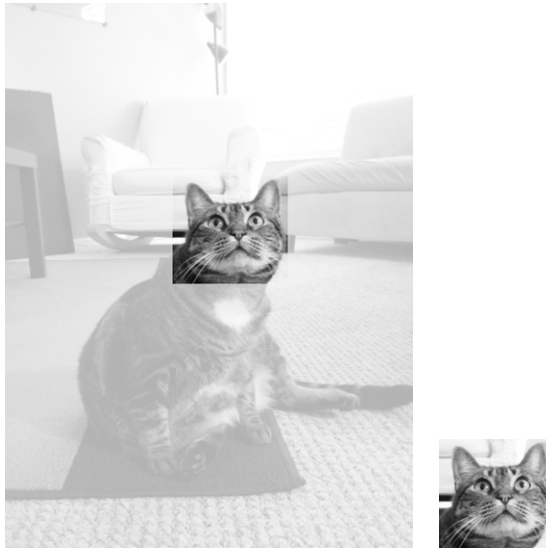
Figure 17-4: The new image will be just the cropped section of the original image.

Copying and Pasting Images onto Other Images

The copy() method will return a new Image object with the same image as the Image object it was called on. This is useful if you need to make changes to an image but also want to keep an untouched version of the original. For example, enter the following into the interactive shell:

```
>>> catIm = Image.open('zophie.png')
>>> catCopyIm = catIm.copy()
```

The catIm and catCopyIm variables contain two separate Image objects, which both have the same image on them. Now that you have an Image object stored in catCopyIm, you can modify catCopyIm as you like and save it to a new filename, leaving *zophie.png* untouched. For example, let's try modifying catCopyIm with the paste() method.

The paste() method is called on an Image object and pastes another image on top of it. Let's continue the shell example by pasting a smaller image onto catCopyIm.

```
>>> faceIm = catIm.crop((335, 345, 565, 560))
>>> faceIm.size
(230, 215)
>>> catCopyIm.paste(faceIm, (0, 0))
>>> catCopyIm.paste(faceIm, (400, 500))
>>> catCopyIm.save('pasted.png')
```

First we pass crop() a box tuple for the rectangular area in *zophie.png* that contains Zophie's face. This creates an Image object representing a 230×215 crop, which we store in faceIm. Now we can paste faceIm onto catCopyIm. The paste() method takes two arguments: a "source" Image object and a tuple of the x- and y-coordinates where you want to paste the top-left corner of the source Image object onto the main Image object. Here we call paste() twice on catCopyIm, passing (0, 0) the first time and (400, 500) the second time. This pastes faceIm onto catCopyIm twice: once with the top-left corner of faceIm at (0, 0) on catCopyIm, and once with the top-left corner of faceIm at (400, 500). Finally, we save the modified catCopyIm to *pasted.png*. The *pasted.png* image looks like Figure 17-5.
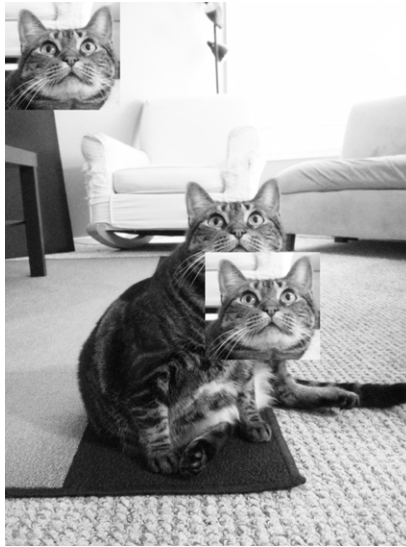

Figure 17-5: Zophie the cat, with her face pasted twice

NOTE *Despite their names, the* copy() *and* paste() *methods in Pillow do not use your computer's clipboard.*

Note that the paste() method modifies its Image object *in place*; it does not return an Image object with the pasted image. If you want to call paste() but also keep an untouched version of the original image around, you'll need to first copy the image and then call paste() on that copy.

Say you want to tile Zophie's head across the entire image, as in Figure 17-6. You can achieve this effect with just a couple for loops. Continue the interactive shell example by entering the following:

```
>>> catImWidth, catImHeight = catIm.size
>>> faceImWidth, faceImHeight = faceIm.size
❶ >>> catCopyTwo = catIm.copy()
```

```
❷ >>> for left in range(0, catImWidth, faceImWidth):
❸         for top in range(0, catImHeight, faceImHeight):
              print(left, top)
              catCopyTwo.paste(faceIm, (left, top))
0 0
0 215
0 430
0 645
0 860
0 1075
230 0
230 215
--snip--
690 860
690 1075
>>> catCopyTwo.save('tiled.png')
```

Here we store the width of height of catIm in catImWidth and catImHeight. At ❶ we make a copy of catIm and store it in catCopyTwo. Now that we have a copy that we can paste onto, we start looping to paste faceIm onto catCopyTwo. The outer for loop's left variable starts at 0 and increases by faceImWidth(230) ❷. The inner for loop's top variable start at 0 and increases by faceImHeight(215) ❸. These nested for loops produce values for left and top to paste a grid of faceIm images over the catCopyTwo Image object, as in Figure 17-6. To see our nested loops working, we print left and top. After the pasting is complete, we save the modified catCopyTwo to *tiled.png*.

Figure 17-6: Nested for loops used with paste() to duplicate the cat's face (a dupli-cat, if you will).

Resizing an Image

The resize() method is called on an Image object and returns a new Image
object of the specified width and height. It accepts a two-integer tuple argu-
ment, representing the new width and height of the returned image. Enter
the following into the interactive shell:

```
❶ >>> width, height = catIm.size
❷ >>> quartersizedIm = catIm.resize((int(width / 2), int(height / 2)))
   >>> quartersizedIm.save('quartersized.png')
❸ >>> svelteIm = catIm.resize((width, height + 300))
   >>> svelteIm.save('svelte.png')
```

Here we assign the two values in the catIm.size tuple to the variables
width and height ❶. Using width and height instead of catIm.size[0] and
catIm.size[1] makes the rest of the code more readable.

The first resize() call passes int(width / 2) for the new width and
int(height / 2) for the new height ❷, so the Image object returned from
resize() will be half the length and width of the original image, or one-
quarter of the original image size overall. The resize() method accepts
only integers in its tuple argument, which is why you needed to wrap both
divisions by 2 in an int() call.

This resizing keeps the same proportions for the width and height. But
the new width and height passed to resize() do not have to be proportional
to the original image. The svelteIm variable contains an Image object that
has the original width but a height that is 300 pixels taller ❸, giving Zophie
a more slender look.

Note that the resize() method does not edit the Image object in place
but instead returns a new Image object.

Rotating and Flipping Images

Images can be rotated with the `rotate()` method, which returns a new `Image` object of the rotated image and leaves the original `Image` object unchanged. The argument to `rotate()` is a single integer or float representing the number of degrees to rotate the image counterclockwise. Enter the following into the interactive shell:

```
>>> catIm.rotate(90).save('rotated90.png')
>>> catIm.rotate(180).save('rotated180.png')
>>> catIm.rotate(270).save('rotated270.png')
```

Note how you can *chain* method calls by calling `save()` directly on the `Image` object returned from `rotate()`. The first `rotate()` and `save()` call makes a new `Image` object representing the image rotated counterclockwise by 90 degrees and saves the rotated image to *rotated90.png*. The second and third calls do the same, but with 180 degress and 270 degress. The results look like Figure 17-7.

Figure 17-7: The original image (left) and the image rotated counterclockwise by 90, 180, and 270 degrees

Notice that the width and height of the image change when the image is rotated 90 or 270 degrees. If you rotate an image by some other amount, the original dimensions of the image are maintained. On Windows, a black background is used to fill in any gaps made by the rotation, like in Figure 17-8. On OS X, transparent pixels are used for the gaps instead.

The `rotate()` method has an optional expand keyword argument that can be set to `True` to enlarge the dimensions of the image to fit the entire rotated new image. For example, enter the following into the interactive shell:

```
>>> catIm.rotate(6).save('rotated6.png')
>>> catIm.rotate(6, expand=True).save('rotated6_expanded.png')
```

The first call rotates the image 6 degrees and saves it to *rotate6.png* (see the image on the left of Figure 17-8). The second call rotates the image 6 degrees with expand set to `True` and saves it to *rotate6_expanded.png* (see the image on the right of Figure 17-8).

Figure 17-8: The image rotated 6 degrees normally (left) and with expand=True (right)

You can also get a "mirror flip" of an image with the transpose() method. You must pass either Image.FLIP_LEFT_RIGHT or Image.FLIP_TOP_BOTTOM to the transpose() method. Enter the following into the interactive shell:

```
>>> catIm.transpose(Image.FLIP_LEFT_RIGHT).save('horizontal_flip.png')
>>> catIm.transpose(Image.FLIP_TOP_BOTTOM).save('vertical_flip.png')
```

Like rotate(), transpose() creates a new Image object. Here was pass Image.FLIP_LEFT_RIGHT to flip the image horizontally and then save the result to *horizontal_flip.png*. To flip the image vertically, we pass Image.FLIP_TOP_BOTTOM and save to *vertical_flip.png*. The results look like Figure 17-9.
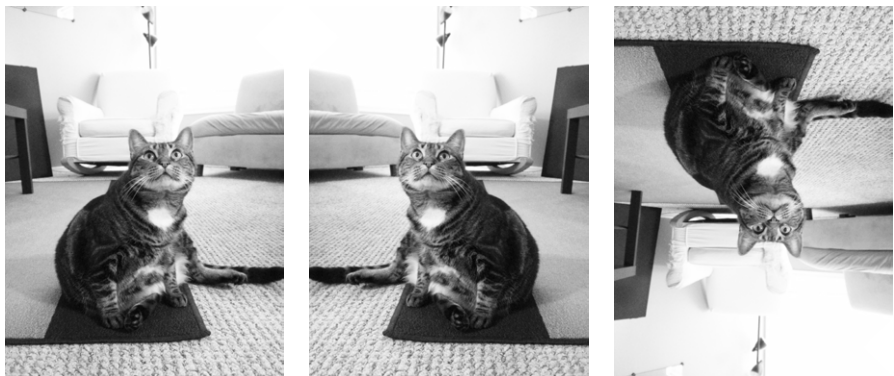

Figure 17-9: The original image (left), horizontal flip (center), and vertical flip (right)

Changing Individual Pixels

The color of an individual pixel can be retrieved or set with the getpixel() and putpixel() methods. These methods both take a tuple representing the x- and y-coordinates of the pixel. The putpixel() method also takes an additional tuple argument for the color of the pixel. This color argument is a four-integer RGBA tuple or a three-integer RGB tuple. Enter the following into the interactive shell:

```
❶ >>> im = Image.new('RGBA', (100, 100))
❷ >>> im.getpixel((0, 0))
   (0, 0, 0, 0)
❸ >>> for x in range(100):
           for y in range(50):
❹              im.putpixel((x, y), (210, 210, 210))

   >>> from PIL import ImageColor
❺ >>> for x in range(100):
           for y in range(50, 100):
❻              im.putpixel((x, y), ImageColor.getcolor('darkgray', 'RGBA'))
   >>> im.getpixel((0, 0))
   (210, 210, 210, 255)
   >>> im.getpixel((0, 50))
   (169, 169, 169, 255)
   >>> im.save('putPixel.png')
```

At ❶ we make a new image that is a 100×100 transparent square. Calling getpixel() on some coordinates in this image returns (0, 0, 0, 0) because the image is transparent ❷. To color pixels in this image, we can use nested for loops to go through all the pixels in the top half of the image ❸ and color each pixel using putpixel() ❹. Here we pass putpixel() the RGB tuple (210, 210, 210), a light gray.

Say we want to color the bottom half of the image dark gray but don't know the RGB tuple for dark gray. The putpixel() method doesn't accept a standard color name like 'darkgray', so you have to use ImageColor.getcolor() to get a color tuple from 'darkgray'. Loop through the pixels in the bottom half of the image ❺ and pass putpixel() the return value of ImageColor.getcolor() ❻, and you should now have an image that is light gray in its top half and dark gray in the bottom half, as shown in Figure 17-10. You can call getpixel() on some coordinates to confirm that the color at any given pixel is what you expect. Finally, save the image to *putPixel.png*.

Figure 17-10: The putPixel.png *image*

Of course, drawing one pixel at a time onto an image isn't very convenient. If you need to draw shapes, use the ImageDraw functions explained later in this chapter.

Automate the Boring Stuff with Python: Practical Programming for Total Beginners
© 2015 Al Sweigart

Project: Adding a Logo

Say you have the boring job of resizing thousands of images and adding a small logo watermark to the corner of each. Doing this with a basic graphics program such as Paintbrush or Paint would take forever. A fancier graphics application such as Photoshop can do batch processing, but that software costs hundreds of dollars. Let's write a script to do it instead.

Say that Figure 17-11 is the logo you want to add to the bottom-right corner of each image: a black cat icon with a white border, with the rest of the image transparent.

Figure 17-11: The logo to be added to the image.

At a high level, here's what the program should do:

- Load the logo image.
- Loop over all *.png* and *.jpg* files in the working directory.
- Check whether the image is wider or taller than 300 pixels.
- If so, reduce the width or height (whichever is larger) to 300 pixels and scale down the other dimension proportionally.
- Paste the logo image into the corner.
- Save the altered images to another folder.

  This means the code will need to do the following:

- Open the *catlogo.png* file as an `Image` object.
- Loop over the strings returned from `os.listdir('.')`.
- Get the width and height of the image from the `size` attribute.
- Calculate the new width and height of the resized image.
- Call the `resize()` method to resize the image.
- Call the `paste()` method to paste the logo.
- Call the `save()` method to save the changes, using the original filename.

Step 1: Open the Logo Image

For this project, open a new file editor window, enter the following code, and save it as *resizeAndAddLogo.py*:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.
```
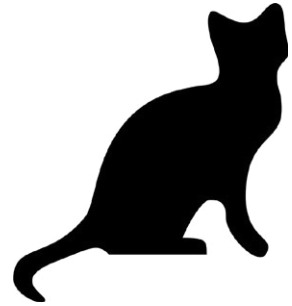
```
       import os
       from PIL import Image

❶ SQUARE_FIT_SIZE = 300
❷ LOGO_FILENAME = 'catlogo.png'

❸ logoIm = Image.open(LOGO_FILENAME)
❹ logoWidth, logoHeight = logoIm.size

   # TODO: Loop over all files in the working directory.

   # TODO: Check if image needs to be resized.

   # TODO: Calculate the new width and height to resize to.

   # TODO: Resize the image.

   # TODO: Add the logo.

   # TODO: Save changes.
```

By setting up the SQUARE_FIT_SIZE ❶ and LOGO_FILENAME ❷ constants at the start of the program, we've made it easy to change the program later. Say the logo that you're adding isn't the cat icon, or say you're reducing the output images' largest dimension to something other than 300 pixels. With these constants at the start of the program, you can just open the code, change those values once, and you're done. (Or you can make it so that the values for these constants are taken from the command line arguments.) Without these constants, you'd instead have to search the code for all instances of 300 and 'catlogo.png' and replace them with the values for your new project. In short, using constants makes your program more generalized.

The logo Image object is returned from Image.open() ❸. For readability, logoWidth and logoHeight are assigned to the values from logoIm.size ❹.

The rest of the program is a skeleton of TODO comments for now.

Step 2: Loop Over All Files and Open Images

Now you need to find every *.png* file and *.jpg* file in the current working directory. Note that you don't want to add the logo image to the logo image itself, so the program should skip any image with a filename that's the same as LOGO_FILENAME. Add the following to your code:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.

import os
from PIL import Image

--snip--
```

```
    os.makedirs('withLogo', exist_ok=True)
    # Loop over all files in the working directory.
❶ for filename in os.listdir('.'):
❷     if not (filename.endswith('.png') or filename.endswith('.jpg')) \
          or filename == LOGO_FILENAME:
❸         continue    # skip non-image files and the logo file itself

❹     im = Image.open(filename)
       width, height = im.size

--snip--
```

First, the os.makedirs() call creates a *withLogo* folder to store the fin-
ished images with logos, instead of overwriting the original image files.
The exist_ok=True keyword argument will keep os.makedirs() from raising
an exception if *withLogo* already exists. While looping through all the files
in the working directory with os.listdir('.') ❶, the long if statement ❷
checks whether each filename doesn't end with *.png* or *.jpg*. If so—or if the
file is the logo image itself—then the loop should skip it and use continue ❸
to go to the next file. If filename *does* end with '.png' or '.jpg' (and isn't the
logo file), you can open it as an Image object ❹ and set width and height.

Step 3: Resize the Images

The program should resize the image only if the width or height is larger
than SQUARE_FIT_SIZE (300 pixels, in this case), so put all of the resizing code
inside an if statement that checks the width and height variables. Add the
following code to your program:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.

import os
from PIL import Image

--snip--

    # Check if image needs to be resized.
    if width > SQUARE_FIT_SIZE and height > SQUARE_FIT_SIZE:
        # Calculate the new width and height to resize to.
        if width > height:
❶           height = int((SQUARE_FIT_SIZE / width) * height)
            width = SQUARE_FIT_SIZE
        else:
❷           width = int((SQUARE_FIT_SIZE / height) * width)
            height = SQUARE_FIT_SIZE

        # Resize the image.
        print('Resizing %s...' % (filename))
❸       im = im.resize((width, height))

--snip--
```

If the image does need to be resized, you need to find out whether it is a wide or tall image. If `width` is greater than `height`, then the height should be reduced by the same proportion that the width would be reduced ❶. This proportion is the `SQUARE_FIT_SIZE` value divided by the current width. The new `height` value is this proportion multiplied by the current `height` value. Since the division operator returns a float value and `resize()` requires the dimensions to be integers, remember to convert the result to an integer with the `int()` function. Finally, the new `width` value will simply be set to `SQUARE_FIT_SIZE`.

If the `height` is greater than or equal to the `width` (both cases are handled in the `else` clause), then the same calculation is done, except with the `height` and `width` variables swapped ❷.

Once `width` and `height` contain the new image dimensions, pass them to the `resize()` method and store the returned `Image` object in `im` ❸.

Step 4: Add the Logo and Save the Changes

Whether or not the image was resized, the logo should still be pasted to the bottom-right corner. Where exactly the logo should be pasted depends on both the size of the image and the size of the logo. Figure 17-12 shows how to calculate the pasting position. The left coordinate for where to paste the logo will be the image width minus the logo width; the top coordinate for where to paste the logo will be the image height minus the logo height.
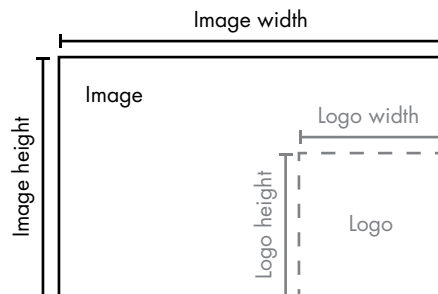

Figure 17-12: The left and top coordinates for placing the logo in the bottom-right corner should be the image width/height minus the logo width/height.

After your code pastes the logo into the image, it should save the modified `Image` object. Add the following to your program:

```
#! python3
# resizeAndAddLogo.py - Resizes all images in current working directory to fit
# in a 300x300 square, and adds catlogo.png to the lower-right corner.

import os
from PIL import Image
```

```
--snip--

    # Check if image needs to be resized.
    --snip--

    # Add the logo.
❶    print('Adding logo to %s...' % (filename))
❷    im.paste(logoIm, (width - logoWidth, height - logoHeight), logoIm)

    # Save changes.
❸    im.save(os.path.join('withLogo', filename))
```

The new code prints a message telling the user that the logo is being added ❶, pastes `logoIm` onto `im` at the calculated coordinates ❷, and saves the changes to a filename in the *withLogo* directory ❸. When you run this program with the *zophie.png* file as the only image in the working directory, the output will look like this:

```
Resizing zophie.png...
Adding logo to zophie.png...
```

The image *zophie.png* will be changed to a 225×300-pixel image that looks like Figure 17-13. Remember that the paste() method will not paste the transparency pixels if you do not pass the `logoIm` for the third argument as well. This program can automatically resize and "logo-ify" hundreds of images in just a couple minutes.

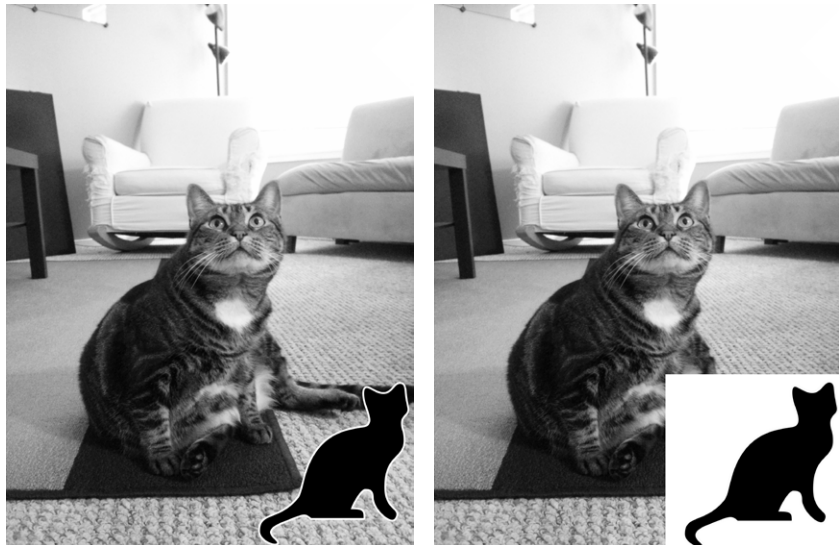
Figure 17-13: The image zophie.png *resized and the logo added (left). If you forget the third argument, the transparent pixels in the logo will be copied as solid white pixels (right).*

Ideas for Similar Programs

Being able to composite images or modify image sizes in a batch can be useful in many applications. You could write similar programs to do the following:

- Add text or a website URL to images.
- Add timestamps to images.
- Copy or move images into different folders based on their sizes.
- Add a mostly transparent watermark to an image to prevent others from copying it.

Drawing on Images

If you need to draw lines, rectangles, circles, or other simple shapes on an image, use Pillow's `ImageDraw` module. Enter the following into the interactive shell:

```
>>> from PIL import Image, ImageDraw
>>> im = Image.new('RGBA', (200, 200), 'white')
>>> draw = ImageDraw.Draw(im)
```

First, we import `Image` and `ImageDraw`. Then we create a new image, in this case, a 200×200 white image, and store the `Image` object in `im`. We pass the `Image` object to the `ImageDraw.Draw()` function to receive an `ImageDraw` object. This object has several methods for drawing shapes and text onto an `Image` object. Store the `ImageDraw` object in a variable like `draw` so you can use it easily in the following example.

Drawing Shapes

The following ImageDraw methods draw various kinds of shapes on the image. The `fill` and `outline` parameters for these methods are optional and will default to white if left unspecified.

Points

The `point(xy, fill)` method draws individual pixels. The *xy* argument represents a list of the points you want to draw. The list can be a list of x- and y-coordinate tuples, such as [(x, y), (x, y), ...], or a list of x- and y-coordinates without tuples, such as [x1, y1, x2, y2, ...]. The *fill* argument is the color of the points and is either an RGBA tuple or a string of a color name, such as 'red'. The *fill* argument is optional.

Lines

The `line(xy, fill, width)` method draws a line or series of lines. *xy* is either a list of tuples, such as [(x, y), (x, y), ...], or a list of integers, such as [x1, y1, x2, y2, ...]. Each point is one of the connecting points on the

lines you're drawing. The optional *fill* argument is the color of the lines, as an RGBA tuple or color name. The optional *width* argument is the width of the lines and defaults to 1 if left unspecified.

Rectangles

The rectangle(*xy, fill, outline*) method draws a rectangle. The *xy* argument is a box tuple of the form (*left, top, right, bottom*). The *left* and *top* values specify the x- and y-coordinates of the upper-left corner of the rectangle, while *right* and *bottom* specify the lower-right corner. The optional *fill* argument is the color that will fill the inside of the rectangle. The optional *outline* argument is the color of the rectangle's outline.

Ellipses

The ellipse(*xy, fill, outline*) method draws an ellipse. If the width and height of the ellipse are identical, this method will draw a circle. The *xy* argument is a box tuple (*left*, *top*, *right*, *bottom*) that represents a box that precisely contains the ellipse. The optional *fill* argument is the color of the inside of the ellipse, and the optional *outline* argument is the color of the ellipse's outline.

Polygons

The polygon(*xy, fill, outline*) method draws an arbitrary polygon. The *xy* argument is a list of tuples, such as [(x, y), (x, y), ...], or integers, such as [x1, y1, x2, y2, ...], representing the connecting points of the polygon's sides. The last pair of coordinates will be automatically connected to the first pair. The optional *fill* argument is the color of the inside of the polygon, and the optional *outline* argument is the color of the polygon's outline.

Drawing Example

Enter the following into the interactive shell:

```
>>> from PIL import Image, ImageDraw
>>> im = Image.new('RGBA', (200, 200), 'white')
>>> draw = ImageDraw.Draw(im)
❶ >>> draw.line([(0, 0), (199, 0), (199, 199), (0, 199), (0, 0)], fill='black')
❷ >>> draw.rectangle((20, 30, 60, 60), fill='blue')
❸ >>> draw.ellipse((120, 30, 160, 60), fill='red')
❹ >>> draw.polygon(((57, 87), (79, 62), (94, 85), (120, 90), (103, 113)),
   fill='brown')
❺ >>> for i in range(100, 200, 10):
        draw.line([(i, 0), (200, i - 100)], fill='green')

>>> im.save('drawing.png')
```

After making an Image object for a 200×200 white image, passing it to ImageDraw.Draw() to get an ImageDraw object, and storing the ImageDraw object in draw, you can call drawing methods on draw. Here we make a thin, black

outline at the edges of the image ❶, a blue rectangle with its top-left corner at (20, 30) and bottom-right corner at (60, 60) ❷, a red ellipse defined by a box from (120, 30) to (160, 60) ❸, a brown polygon with five points ❹, and a pattern of green lines drawn with a for loop ❺. The resulting *drawing.png* file will look like Figure 17-14.
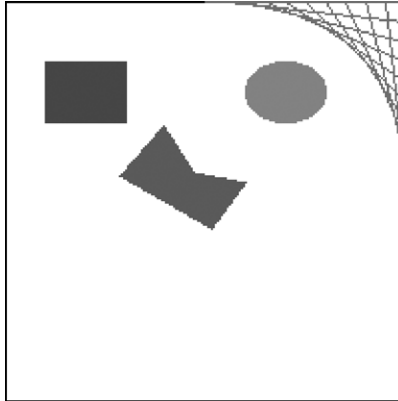

Figure 17-14: The resulting drawing.png *image*

There are several other shape-drawing methods for ImageDraw objects. The full documentation is available at *http://pillow.readthedocs.org/en/latest/reference/ImageDraw.html*.

Drawing Text

The ImageDraw object also has a text() method for drawing text onto an image. The text() method takes four arguments: *xy*, *text*, *fill*, and *font*.

- The *xy* argument is a two-integer tuple specifying the upper-left corner of the text box.
- The *text* argument is the string of text you want to write.
- The optional *fill* argument is the color of the text.
- The optional *font* argument is an ImageFont object, used to set the type-face and size of the text. This is described in more detail in the next section.

Since it's often hard to know in advance what size a block of text will be in a given font, the ImageDraw module also offers a textsize() method. Its first argument is the string of text you want to measure, and its second argument is an optional ImageFont object. The textsize() method will then return a two-integer tuple of the width and height that the text in the given

font would be if it were written onto the image. You can use this width and height to help you calculate exactly where you want to put the text on your image.

The first three arguments for text() are straightforward. Before we use text() to draw text onto an image, let's look at the optional fourth argument, the ImageFont object.

Both text() and textsize() take an optional ImageFont object as their final arguments. To create one of these objects, first run the following:

```
>>> from PIL import ImageFont
```

Now that you've imported Pillow's ImageFont module, you can call the ImageFont.truetype() function, which takes two arguments. The first argument is a string for the font's *TrueType file*—this is the actual font file that lives on your hard drive. A TrueType file has the *.ttf* file extension and can usually be found in the following folders:

- On Windows: *C:\Windows\Fonts*
- On OS X: */Library/Fonts* and */System/Library/Fonts*
- On Linux: */usr/share/fonts/truetype*

You don't actually need to enter these paths as part of the TrueType file string because Python knows to automatically search for fonts in these directories. But Python will display an error if it is unable to find the font you specified.

The second argument to ImageFont.truetype() is an integer for the font size in *points* (rather than, say, pixels). Keep in mind that Pillow creates PNG images that are 72 pixels per inch by default, and a point is 1/72 of an inch.

Enter the following into the interactive shell, replacing FONT_FOLDER with the actual folder name your operating system uses:

```
>>> from PIL import Image, ImageDraw, ImageFont
>>> import os
❶ >>> im = Image.new('RGBA', (200, 200), 'white')
❷ >>> draw = ImageDraw.Draw(im)
❸ >>> draw.text((20, 150), 'Hello', fill='purple')
>>> fontsFolder = 'FONT_FOLDER' # e.g. '/Library/Fonts'
❹ >>> arialFont = ImageFont.truetype(os.path.join(fontsFolder, 'arial.ttf'), 32)
❺ >>> draw.text((100, 150), 'Howdy', fill='gray', font=arialFont)
>>> im.save('text.png')
```

After importing Image, ImageDraw, ImageFont, and os, we make an Image object for a new 200×200 white image ❶ and make an ImageDraw object from the Image object ❷. We use text() to draw *Hello* at (20, 150) in purple ❸. We didn't pass the optional fourth argument in this text() call, so the typeface and size of this text aren't customized.

To set a typeface and size, we first store the folder name (like */Library/Fonts*) in fontsFolder. Then we call ImageFont.truetype(), passing it the *.ttf* file for the font we want, followed by an integer font size ❹. Store the Font object you get from ImageFont.truetype() in a variable like arialFont, and then pass the variable to text() in the final keyword argument. The text() call at ❺ draws *Howdy* at (100, 150) in gray in 32-point Arial.
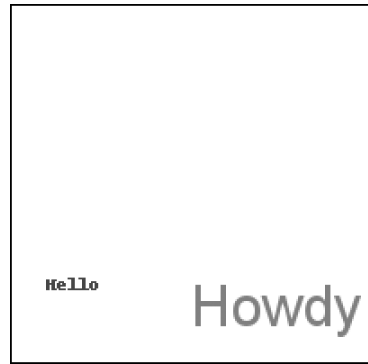
The resulting *text.png* file will look like Figure 17-15.

Figure 17-15: The resulting text.png *image*

Summary

Images consist of a collection of pixels, and each pixel has an RGBA value for its color and its addressable by x- and y-coordinates. Two common image formats are JPEG and PNG. The pillow module can handle both of these image formats and others.

When an image is loaded into an Image object, its width and height dimensions are stored as a two-integer tuple in the size attribute. Objects of the Image data type also have methods for common image manipulations: crop(), copy(), paste(), resize(), rotate(), and transpose(). To save the Image object to an image file, call the save() method.

If you want your program to draw shapes onto an image, use ImageDraw methods to draw points, lines, rectangles, ellipses, and polygons. The module also provides methods for drawing text in a typeface and font size of your choosing.

Although advanced (and expensive) applications such as Photoshop provide automatic batch processing features, you can use Python scripts to do many of the same modifications for free. In the previous chapters, you wrote Python programs to deal with plaintext files, spreadsheets, PDFs, and other formats. With the pillow module, you've extended your programming powers to processing images as well!

Practice Questions

1. What is an RGBA value?
2. How can you get the RGBA value of 'CornflowerBlue' from the Pillow module?
3. What is a box tuple?
4. What function returns an Image object for, say, an image file named *zophie.png*?

5. How can you find out the width and height of an `Image` object's image?

6. What method would you call to get `Image` object for a 100×100 image, excluding the lower left quarter of it?

7. After making changes to an `Image` object, how could you save it as an image file?

8. What module contains Pillow's shape-drawing code?

9. `Image` objects do not have drawing methods. What kind of object does? How do you get this kind of object?

Practice Projects

For practice, write programs that do the following.

Extending and Fixing the Chapter Project Programs

The *resizeAndAddLogo.py* program in this chapter works with PNG and JPEG files, but Pillow supports many more formats than just these two. Extend *resizeAndAddLogo.py* to process GIF and BMP images as well.

Another small issue is that the program modifies PNG and JPEG files only if their file extensions are set in lowercase. For example, it will process *zophie.png* but not *zophie.PNG*. Change the code so that the file extension check is case insensitive.

Finally, the logo added to the bottom-right corner is meant to be just a small mark, but if the image is about the same size as the logo itself, the result will look like Figure 17-16. Modify *resizeAndAddLogo.py* so that the image must be at least twice the width and height of the logo image before the logo is pasted. Otherwise, it should skip adding the logo.
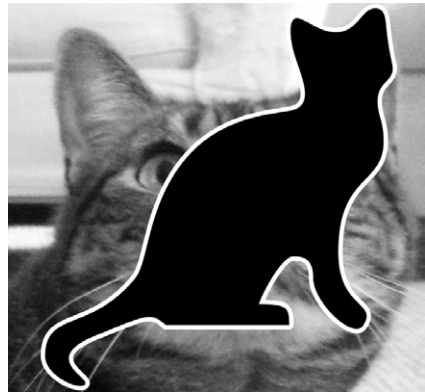

Figure 17-16: When the image isn't much larger than the logo, the results look ugly.

Identifying Photo Folders on the Hard Drive

I have a bad habit of transferring files from my digital camera to temporary folders somewhere on the hard drive and then forgetting about these folders. It would be nice to write a program that could scan the entire hard drive and find these leftover "photo folders."

Write a program that goes through every folder on your hard drive and finds potential photo folders. Of course, first you'll have to define what you consider a "photo folder" to be; let's say that it's any folder where more than half of the files are photos. And how do you define what files are photos?

First, a photo file must have the file extension *.png* or *.jpg*. Also, photos are large images; a photo file's width and height must both be larger than 500 pixels. This is a safe bet, since most digital camera photos are several thousand pixels in width and height.

As a hint, here's a rough skeleton of what this program might look like:

```
#! python3
# Import modules and write comments to describe this program.

for foldername, subfolders, filenames in os.walk('C:\\'):
    numPhotoFiles = 0
    numNonPhotoFiles = 0
    for filename in filenames:
        # Check if file extension isn't .png or .jpg.
        if TODO:
            numNonPhotoFiles += 1
            continue    # skip to next filename

        # Open image file using Pillow.

        # Check if width & height are larger than 500.
        if TODO:
            # Image is large enough to be considered a photo.
            numPhotoFiles += 1
        else:
            # Image is too small to be a photo.
            numNonPhotoFiles += 1

    # If more than half of files were photos,
    # print the absolute path of the folder.
    if TODO:
        print(TODO)
```

When the program runs, it should print the absolute path of any photo folders to the screen.

Custom Seating Cards

Chapter 13 included a practice project to create custom invitations from a list of guests in a plaintext file. As an additional project, use the `pillow` module to create images for custom seating cards for your guests. For each of the guests listed in the *guests.txt* file from the resources at *http://nostarch.com/automatestuff/*, generate an image file with the guest name and some flowery decoration. A public domain flower image is available in the resources at *http://nostarch.com/automatestuff/*.

To ensure that each seating card is the same size, add a black rectangle on the edges of the invitation image so that when the image is printed out, there will be a guideline for cutting. The PNG files that Pillow produces are set to 72 pixels per inch, so a 4×5-inch card would require a 288×360-pixel image.