Getting Started with OpenCV

Susan Fox

Activity goals

This activity introduces the programming environment we're going to use (PyCharm), and some basic Python tools, and then starts working with images. The goals of this activity are that you:

- Learn to use PyCharm to write Python programs
- Learn a few basic types of Python commands specifically for numbers and strings
- Learn how to access OpenCV from Python
- Learn how to access parts of the Numpy representation of the image
- Learn how color representations work
- Learn how blending of images works

Each of these learning goals is taught through explanations and code demos. Please do everything that the activity asks you to do. There are six "Milestones" in the activity. These are programs that you should write, each in its own file, and save the files to hand in. Make sure to do the tasks in between milestones as well: many times they are meant to lead you toward the next milestone. The final goal is to create a visual echo on a video feed.

Overview

In order to start doing image processing and computer vision, you will need to learn how to write programs that manipulate images. This means learning two tools: Python, a general-purpose programming language, and OpenCV, a computer vision package integrated with Python. I've chosen to drop you straight into using OpenCV, and to teach you the Python you need as you go along. You'll learn just the parts of both Python and OpenCV that you need, though I will provide additional resources for those who need more resources. The "Python Support" document and the *A Byte of Python* online book both have more information about Python that you might find useful.

Take this activity at your own pace. I will provide links to additional reading and practice material if you feel like you need more help with any topic. Different people will take more or less time on each activity, and that's fine.

Try examples and write code on your own computer, but feel free to use the "Updates" forum to ask questions, share ideas and code. Don't hesitate to ask questions of your

instructor as you work through the activities, too! Upload the "Milestone" programs to the associated assignment in Schoology when you finish.

Background

We will start with a bit of information about Python, OpenCV, and PyCharm, the tools we're going to use.

Python

Python is a programming language. That means that it is a set of rules about what makes a "grammatical" program, and what each statement in the program means. Because of the kind of language Python is, we can communicate with the computer interactively using Python's *interactive shell*. The shell is a text interface, where you can type Python statements, and the computer responds to those statements.

Besides interactive "conversations" with the computer written in Python, you can also package up a series of statements as a script, saved in a file. When you run the script file, the computer performs each statement in order. When you run a script, you don't see the result of every line, but you do see places where the program explicitly causes something to print or display.

Related readings (if you want more background):

- Introduction to Python from *A Byte of Python*

OpenCV

OpenCV is a computer vision project that includes many tools for manipulating images, and implements a lot of advanced techniques so that you and I don't have to. It was originally written in C, updated to C++, but in the past several years they have expanded to several other languages. Python is a popular choice, because it is easy to learn and use.

In order to work with images, you will first need to learn how color and grayscale images are represented, and then you will learn the tools OpenCV provides for displaying and modifying images.

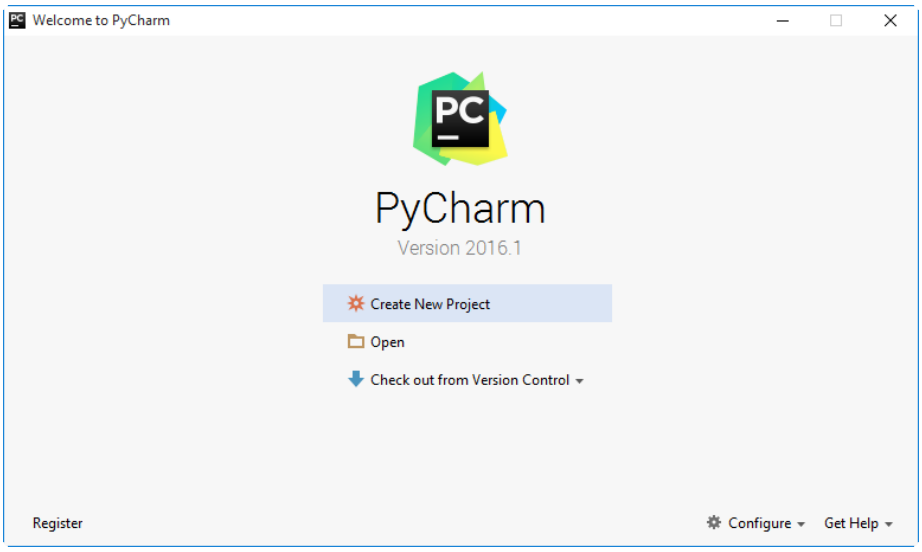
PyCharm Python IDE

There are many different applications we could use when writing Python programs, much as there are many different text editor and word processor applications for writing papers. Some of your readings may refer to a program editor called IDLE, which typically comes with Python

when you download it. Instead of IDLE, we will be using the PyCharm application (you're welcome to use something else, but I can't necessarily help with configuring things). There is a free "community edition" of this program; if you want to you can download both the Python language and this editor for free (see the "installing software" document for details).

PyCharm is an IDE, an Integrated Development Environment. That means that it is an editor for Python programs, but it can also do other things, like run your program run a debugger, and inspect/critique your code. We will not come close to using all its capabilities.

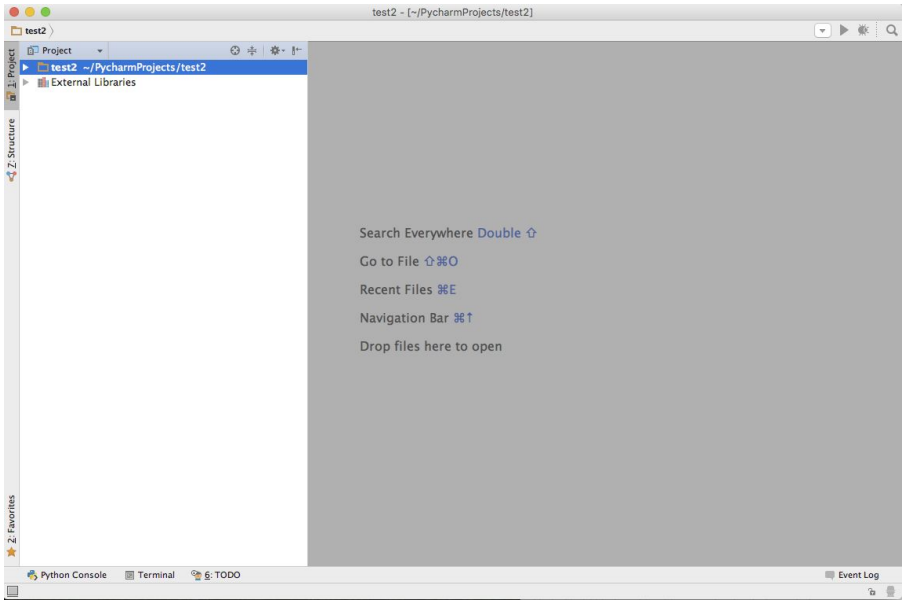
When you start up PyCharm for the first time, you may have to answer some questions, but eventually you should see this starter window:



PyCharm organizes your programs into Projects. Conceptually, a project is a set of files that all have to do with a single ultimate program or task. We will generally suggest that you create a new project for every activity we do, but you could conceivably create one project and put everything you do in that one place.

To start, click on "Create New Project." The next window shows the name of the new project and its location, in the text box at the top. **In addition**, it shows the kind of Python interpreter you want to use for this project (many computers have multiple Python interpreters installed on them). Change the last part of the project path and name, the part that says "untitled," to be "Activity 1." Make sure that the interpreter is **not** a "virtual environment" interpreter, and that it is Python 3 (we'll tell you the details during class, so pay attention or ask for help!).

Once you create the project, the PyCharm window should look something like this:



There are several different panels in the PyCharm window. On the left at the top is the Project panel, which will show you the files in the current project. The gray area to the right is the code editor panel, where you can create and edit Python code files. Think of this as a specialized word processor for writing Python. Along the bottom of the window are several other panels you can open. For now, click on the “Python Console” button. This should give you a python console panel along the bottom portion of the window. This console, or “shell”, provides interactive access to Python; you can talk directly to Python and try out bits of code and see the result immediately. The >>> symbol is a prompt, it is Python's way of telling you that it is ready to accept input from you.

You can adjust the size of every panel of the Python window, by clicking and dragging on the border area between panels. Feel free to adjust the areas to suit yourself. If you accidentally close a panel, often the name of the panel may appear along the edges of the window. If not, ask for help!

Conversing in Python

We’re going to start by having a conversation with the Python interpreter in the Python Console panel. The Python interpreter is the program that translates your Python statements into something the computer can understand, and translates the computer’s response back into something you can understand. We’ll use this conversation to start learning about some of the most basic Python statements, and types of data, as well as basic OpenCV commands.

To get ready for our conversation, do the following:

- Go to our Schoology site, and download the `SampleImages.zip` archive file. Double-click on it to extract a folder called `SampleImages`. Using the Finder (on Mac) or file browser, move this folder into the Activity 1 folder (the folder for the project you created above).
- Also download the `SamplePrograms.zip` archive file. Extract the folder, and put it into your project as well. This has various demo programs we will use over the next several activities.
- Open a Python console, also called a shell, by clicking on the small square in the lower left corner of the PyCharm window, and selecting "Python Console." Make the Python shell area, where you see the >>> symbol, larger by clicking and dragging on the border between areas of the window .

First steps in conversation: numbers and strings

You can use the Python shell as a glorified calculator. Python understands numbers, both integers (no decimal point) and floating-point (real numbers, with a decimal point; computer scientists call them floating-point because inside the computer they are represented in a form of scientific notation). Try typing each of the following examples into the Python console. Ask a question if you don't understand why you got the answers you got: ask for help! Volunteer to help others if you know the answer.

Example	**Meaning**
`3 + 12`	Adds two numbers
`45 - 19`	Subtracts second number from first
`5 * 100`	Multiplies two numbers

`55 / 2`
`55.0 / 2`
`55 / 2.0`
	Divides first number by second
	(note integer versus float results)

`max(1, 5, 3)`	Built-in function returns largest of its inputs
`min(1, 5, 3)`	Built-in function returns smallest of its inputs
`abs(-225)`	Built-in function returns absolute value of its input: |-255|

There are many other math operations available to you, particularly if you load in the `math`, `random`, or `statistics` modules. You will learn more when and if you need it.

Strings are how Python represents characters, words, and text. Python is particularly good at string manipulations, but we won't need to use them much. Try the following examples to see the basics you will need:

Examples	**Meaning**
`'A simple string'` `"A simple string"`	This defines a string containing 15 characters
`len('A simple string')`	Built-in function returns the length of a string
`'bob' + 'cat'`	Concatenates two strings together (makes one string containing the characters from the two arguments)
`print "Size:", 23`	Prints the value of data given to it, separated by spaces

Lists provide a way to collect multiple pieces of data together. Lists are written with square brackets around the data. Tuples are similar to lists, but are written with round parentheses.

Examples	**Meaning**
`[55, 23, 102, 86]`	This defines a list containing four numbers
`(155, 202, 19)`	This defines a tuple containing three numbers
`len(['a', 12, 'v'])`	Built-in function returns the length of a list
`[14, 'x'] + ['g', 'a', 0]`	Concatenates two lists together
`len((155, 202, 19))`	Built-in function returns the list of a tuple

Remembering things: variables and assignment statements

If you want the computer to remember some piece of data, you need to tell it where to store it. In Python, we do that using *variables*, which are really just names for a place where you can store a piece of data. A variable is a single word. It has to start with an alphabetic character (or the underscore symbol) and it is made up of alphabetic characters, numbers, or the underscore. You both create variables and change a variable's value using the assignment statement. Below are some examples of assignment statements. Try each one. Notice that assignment statements don't display any response. To check on a variable's value, you can either type the variable name into the shell, or use a call to the `print` function.

Python expression/statement	**Meaning**

`x = 100`	Assigns the value `100` to variable x
`(2 * x) - 33`	Computes the value $2x - 33$
`z = (x - (5 * 4))`	Assigns the value of the expression (`80`) to variable z
`x = 25`	Changes the value of x to be `25`
`print(x, x - 12, z)`	Prints the values of each expression on a line, separated by spaces (`25 13 80`)
`string = 'Hello there'`	Assigns variable `string` to be the string `Hello there`
`len(string)`	Computes the length of the value in `string` (`11`)
`x * string`	Repeats `Hello there` 25 times
`print(string[1], string[3])`	Prints the characters at position 1 and 3
`lst1 = ['f', 'h', 'y', 'u']`	Assigns variable `lst1` to hold the list
`lst2 = [14, 18 25]`	Assigns variable `lst2` to hold the list
`sum2 = lst2[0] + lst2[2]`	Assigns variable `sum2` to hold the sum of 14 and 25
`print(lst1[1:4])`	Prints a copy of part of `lst1` starting at position 1 and going up to but not including position 4

Related readings (if you want more background):

- Basics from *A Byte of Python*, Sections 7.1 through 7.7
- Operators and Expressions from *A Byte of Python*

Beyond conversation: writing programs

Conversing with the computer through the Python shell is a great way to try out new things, to test out how to do something step by step. But working command by command becomes tedious. In addition, the shell conversation is not permanent: nothing you do there is being saved. In the long run, we want to write programs that are saved in a file to be run over and over, and we want the computer to perform a whole sequence of actions without needing our help. To do this, we create programs, called scripts, and use the editor part of the PyCharm window.

A script is a file that contains plain text. Python scripts have `.py` at the end of their names. We type the script program into the editor window, and then save it to the file. We can then *run* the file, causing the computer to perform the statements in the file as if we had typed them into the shell one after the other.

Try this process out by creating a new Python file. To do this, go to the File menu, and select `New...`. Select `Python File`, and type `script1` as the name of the file. PyCharm will automatically add a .py extension to the filename. In the editor panel to the right, you should get a blank document with the filename you gave it.

Copy the statements from the "Remembering things" section into the editor.

We want to run this program and see what Python does. The first time you run each Python file, you will need to select `Run` from the `Run` menu (or you can right-click on the background of the editor panel). After the first run of a file, the green arrow in the upper-right corner will be enabled, and you can click it to run the file. **Ask for help right away if the program won't run for you.**

Feel free to discuss with your classmates the results you got: how do they differ from the results when you entered the commands one by one into the shell? ***Note:*** *when a script is run it only displays the things you explicitly say to print.* Go back to the script and change all the lines other than assignment statements to be `print` statements, then re-run the script.

Related readings (if you want more background):

- Basics from *A Byte of Python*, Sections 7.12 through 7.14

Starting to work with OpenCV

OpenCV is an add-on *module* in Python. Modules contain constants and functions that have a special purpose. They are not automatically loaded and available when we start up the Python interpreter. We have to explicitly ask for them to be loaded using the `import` command. For instance, the command below will load the OpenCV tools.

```
import cv2
```

You can type the `import` command into the Python shell, or you can put it in a script file. **Note: Always put all `import` statements at the top of your file.**

There are many useful modules provided by Python, or added by third-party folks. Other modules you might explore include `math`, `random`, `statistics`, `sys`, and `os`. A secondary module used by OpenCV is `numpy`, which provides efficient tools for representing and manipulating multidimensional arrays of numbers.

When you import a module, Python makes a special separate *namespace* where the tools from that module reside. In order to access them, you must tell Python to look in that namespace by attaching the name of the module, followed by a period. You'll see examples of this below.

We're going to use the OpenCV module called `cv2`. As you create additional Python files, add them to the Project you already created, which contains the `SampleImages` folder.

OpenCV has excellent online documentation, and you will need to become familiar with using it. The link below leads to the documentation:

https://docs.opencv.org/4.2.0/

Create a new Python file and type the script below into it. Make sure that this script is saved in the same folder as the SampleImages folder you downloaded from our web site.

```
import cv2

img1 = cv2.imread("SampleImages/snowLeo1.jpg")
cv2.imshow("Leopard 1", img1)
img2 = cv2.imread("SampleImages/snowLeo2.jpg")
cv2.imshow("Leopard 2", img2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The script above reads in two images from the `SampleImages` folder, and displays them in separate windows.

- The first line loads OpenCV.
- The next line reads in the image `snowLeo1.jpg`, putting its data in a Numpy *array*, and then assigning the variable `img1` to hold the array.
- The third line of code displays the image in a window called `"Leopard 1"`.
- The next two lines read in and display the `snowLeo2.jpg` image.
- The second-to-last line tells the program to wait indefinitely until the user types a key while one of the image windows is the active window.
- The last line tells the program to close all the OpenCV windows.

Run the program to see what happens. To get the program to finish, click on one of the image windows to make it active, and then hit a key on the keyboard. The windows should disappear and PyCharm should be done with the program.

OpenCV uses the string that is the first input to `imshow` to identify windows: if you use the same string, then the image will appear in the same window. To see this, change the string for the second call to `imshow` to be `"Leopard 1"`. Then rerun the script. You probably will not see the first image, because it is overwritten so quickly by the second one. To slow things down, copy the `waitKey` line and put it right after the first call to `imshow`.

Milestone 1: Displaying images one by one

Create a new program using OpenCV. Your program should read in and display, one after the other, four images of your choice from the `SampleImages` folder. Read the images into four variables, using `imread`. Then display the first image in a window called `"Images"`. When the user types a key (call `waitKey`), go on to the next image of the four, displaying it in the same window. Close the window after the user types a key on the fourth image.

Suppose you wanted to make a slideshow program that would display **every picture** in the `SampleImages` folder. It is very possible to do so, but it would be incredibly tedious if you wrote a separate call to `imread` for every image in the folder. It is a truth in computer science that any time a program starts to be tedious and *repetitive* to create, it's time to find a better way. Repetitive programs are hard to debug and change. Being lazy as a coder, seeking the simplest, shortest solution, often leads to better code.

The next section will show you how to make a slideshow program with less repetitive programming. To do that, you need a few more Python tools, tools that will be very useful going forward.

Tools to simplify the next task

We want to create a slideshow program that will display each image in the `SampleImages` folder, having the user type a key in the window to go to the next image. If we used what we already know, then the program could have more or less the form shown below:

```
import cv2
```

```
nextImg = cv2.imread("SampleImages/antiqueTractors.jpg")
cv2.imshow("Images", nextImg)
cv2.waitKey(0)

nextImg = cv2.imread("SampleImages/beachBahamas.jpg")
cv2.imshow("Images", nextImg)
cv2.waitKey(0)

nextImg = cv2.imread("SampleImages/landscape1MuchSmaller.jpg")
cv2.imshow("Images", nextImg)
cv2.waitKey(0)
.
.
.
cv2.destroyAllWindows()
```

Notice how repetitive this code is. Each chunk of 3 lines is almost identical to the others. I've emphasized this by reusing the same variable name to hold each image. Since we don't need the previous image once we've moved on to the new one, reusing the same variable works fine. The only part that differs is the actual filename.

To simplify this, we would like to have only one copy of the three nearly-identical lines, which we then tell the computer to repeat, changing the filename each time. We need a way to list off the filenames one by one. To do this, we will use a ***list***, a kind of data in Python that holds a sequential list of other data. And we will use a Python ***for loop***, a construct that tells the computer to repeat a set of commands a certain number of times.

A list in Python is written by putting square brackets around a sequence of Python expressions or data, where each piece of data is separated by a comma from the next. The code fragment below defines a variable to hold a list of strings. Each string is a filename from the `SampleImages` folder. Note that you can define a list on a single line, or you can break it up into multiple lines, so long as the extra lines start right where the first piece of data starts. *If your brackets, quotation marks, and commas are all correct, PyCharm will indent correctly for you.*

```
imageNames = [antiqueTractors.jpg',
              'beachBahamas.jpg',
              'chicago.jpg',
              'Coins/coins2.jpg',
              'Coins/dollarCoin.jpg']
```

Once we have a collection of names, we can use the `for` loop to tell Python to repeat the same sequence of commands for each name. The `for` loop in Python is extremely useful, and has a very specific syntax:

```
for var in sequence:
    statements to be repeated (indented)
```

The words shown in bold are fixed; they occur exactly as shown in every `for` loop. The italicized words will be replaced by actual code; they just describe the kind of code you will put there. The word after `for` must be a variable name. This special variable is the *loop variable*. The next word must be `in`, and then we must specify a sequence. For now, think of a sequence as being a list, though there are other sequence data types in Python. At the end of this line we must end with a colon. The commands to be repeated are listed, indented one level, after the `for` line.

Below is a simple example of a `for` loop. The script below loops over the values in the original list, printing out the original number and 100 times the number. **Try this script.**

```
numList = [35, 2, 24, 17, 0, 22]
for num in numList:
    print num, 100 * num
```

For our slideshow, we have a list of filenames. To loop over the names in the list we would have a `for` loop that looks something like this:

```
for name in imageNames:
    ...
```

Related readings (if you want more background):

- Data Structures from *A Byte of Python*, Sections 12.1-12.3
- Control Flow from *A Byte of Python*, Section 9.3

Booleans and if statements: We need to be able to ask yes-no questions in our programs. The Boolean data type helps us to do that.Boolean expressions and functions return `True` or `False`. One place we use Boolean expressions is in an `if` statement. The boolean expression serves as a question, and the program chooses which statements to perform, based on whether the answer to the question is `True` or `False`. If statements have the following general form:

```
if test1:
```

```
    statements done if test1 is True
elif test2:
    statements done if test1 is False and test2 is True
. . .
else:
    statements done if all tests are False
```

The `elif` and `else` parts are optional. The computer performs each test in order. When it finds a test that is `True`, then it performs the statements associated with that test, and then skips the rest of the `if` statement. The `else` must be the last case; if the computer gets to it, then it does the statements associated with `else`.

Related readings (if you want more background):

- Control Flow from *A Byte of Python*, Section 9.1

Milestone 2: Making a basic slideshow

Put these pieces together to make a script program that displays every picture in the `SampleImages` folder, one by one (or at least 10 of them). The user should type a key in the image window to move from one picture to the next. You should:

- Create a new Python file
- Start by importing `cv2`
- Next, define `imageNames` to be a list of file names, as shown above, but include at least 10 file names from `SampleImages`
- Next, write a `for` loop that loops over the names in `imageNames`, like the one shown in the blue box above
- Indented underneath the `for` loop line, include the three steps to display a single image. These three lines will look like the sets of three lines from the first script in the previous section
    - First, a line that reads in the image. Here, the variable `name` holds the current filename we want to read in. We must attach the folder name to it before reading it in. Use use the + (plus) operator to attach the folder name to the filename: `"SampleImages/" + name`)
    - Second, a line to display the image in the `"Images"` window (just like the code from the script)
    - Third, a line to wait for the user to type a key (just like the code from the script
- After the end of the loop body, not indented, add a line that closes all windows

Be sure to test your program all the way through to make sure it works for every image.

Finally, you will make a few changes so that your program will loop over all the images in the `SampleImages` folder.

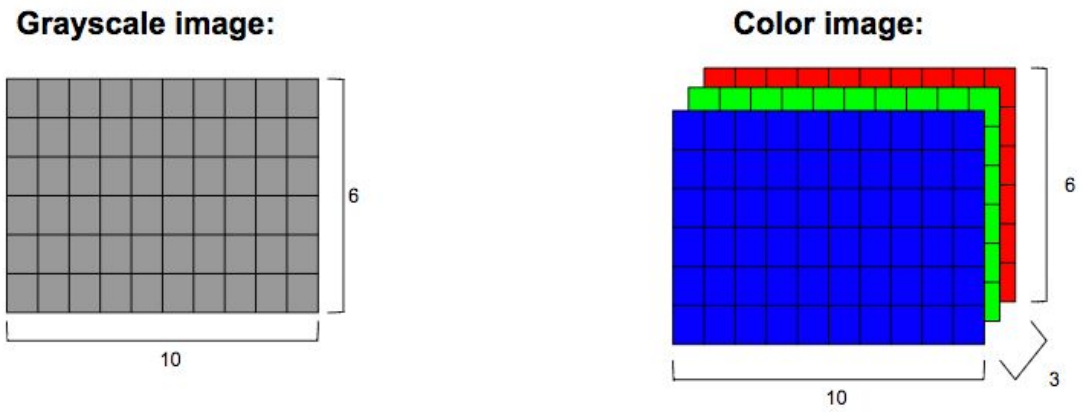
- At the top of your code file add an import statement: `import os`. This module allows you to talk to the computer's operating system.
- Change your definition of imageNames so that is looks like this: `imageNames = os.listdir("SampleImages")`. This will ask the OS to provide a list of all files in the `SampleImages` folder.
- Inside the for loop, before you do anything else, add an if statement to check whether each filename is actually an image. The test should look something like this:
    - `if (name.endswith('jpg') or name.endswith('png')):`
    - Only if this test is true should the program go ahead to read and display the image. Otherwise, it should do nothing and go on to the next filename.

Image Representations in OpenCV

In order to work with images in more depth, you need to understand how a digital image is represented in the computer in general, and particularly in OpenCV. This section will briefly discuss how digital images "work" and ways to represent grayscale and color. When an image is stored in a file, the file format often keeps the image's data in a compressed form, saving space. The exact format and kind of compression used may be determined by the file extension (.jpg, .png, .gif, etc.). When the image is read into a program to be manipulated, however, the image is represented as a *matrix* of brightness (for gray images) or color values. The compression is undone, and we don't have to worry about it for anything we do.

A digital image is a two-dimensional grid of *pixels*. Each pixel has a single color. When the pixels are drawn small enough, we don't notice the individual pixels, and instead see the whole picture. The pictures below show the "Mighty Midway" picture from `SampleImages`, and then a portion of the picture, zoomed in far enough to see the individual pixels. The original picture is 1000 pixels wide by 667 pixels tall.

A grayscale image has just one color per pixel, representing the shade of gray at that location. So the image may be represented as a two-dimensional array, “matrix,” of numbers. A small example is shown in the figure below on the left. Color images, however, have more data per pixel, and generally use 2 or 3 numbers to represent each color. A small example is shown in the figure below on the right. The grayscale image is 10 pixels wide by 6 pixels tall. Each pixel in the matrix would contain a single number, the brightness at that pixel. The color image is 10 pixels wide by 6 pixels tall, but each pixel has three values associated with it to specify its color. This example shows the BGR representation common in OpenCV.



Pixel positions are given as a pair of coordinate values. The origin, (0, 0), is the upper left pixel in the image, and the row and column values increase as it progresses to the right and downward. It is common in image processing to assign x to the horizontal dimension and y to the vertical, so x increases going to the right, and y increases going downward.

Digital color representations

There are several different ways of representing colors in the computer. For the most part, we will use a standard RGB representation of colors. A color, in this representation, is a mixture of some amount of red light, some amount of green light, and some amount of blue light. By mixing these three, we can create many different colors. White is represented as the highest amount of red, green, and blue, and black has no red, green, or blue. Gray shades have equal amounts of red, green, or blue. We call red, green, and blue color *channels* and we represent colors by three numbers that indicate how much of each channel we have.

The website below shows a typical color wheel where you can select colors. It also shows the usual RGB values that generate that color, as well as HSV values. HSV (hue-saturation-value) is one of the main alternative color representations. Try out the color wheel, looking at the values that result.

http://www.colorpicker.com/

Notice that the range of values for each channel is from 0 to 255. This might seem an odd range, but it relates to how data is stored in the computer. Computers represent everything in base two, binary, because ultimately values are represented by either the presence or absence of electricity flowing on a set of wires. The smallest unit inside the computer is a bit, which is one wire, one binary digit that can be 0 or 1. Typically, however, we work with a set of 8 bits, called a byte. If you have 8 binary bits, then there are a total of 256 different patterns of 0s and 1s: 00000000, 00000001, 0000010, … 11111110, 11111111. Those values represent the base-10 positive integers from 0 up to 255. Each color channel is allocated one byte of memory to represent how much of its color is in the image.

In OpenCV, we represent RGB colors as a tuple containing three numbers. A tuple is like a list, but can't be modified. The numbers in the tuple must be integers between 0 and 255. For obscure reasons, the tuple values are written backwards: (blue, green, red). For instance, (0, 0, 0) is black, (255, 0, 0) is bright blue, and (0, 180, 180) is a medium yellow.

The `split` function will separate the channels of a color image into separate one-dimensional arrays, so we can look at them separately. Python lets us capture elements of a tuple into separate variables (see script below). The `merge` function takes a tuple of channel arrays and combines them. **Try the script below to see the color channels.** Notice that each separate channel appears as grayscale when displayed, because it is just a 2d matrix with a single value at each location. There is nothing intrinsically red, green, or blue about the values: they are interpreted as red, green, and blue because of their placement in the color image.

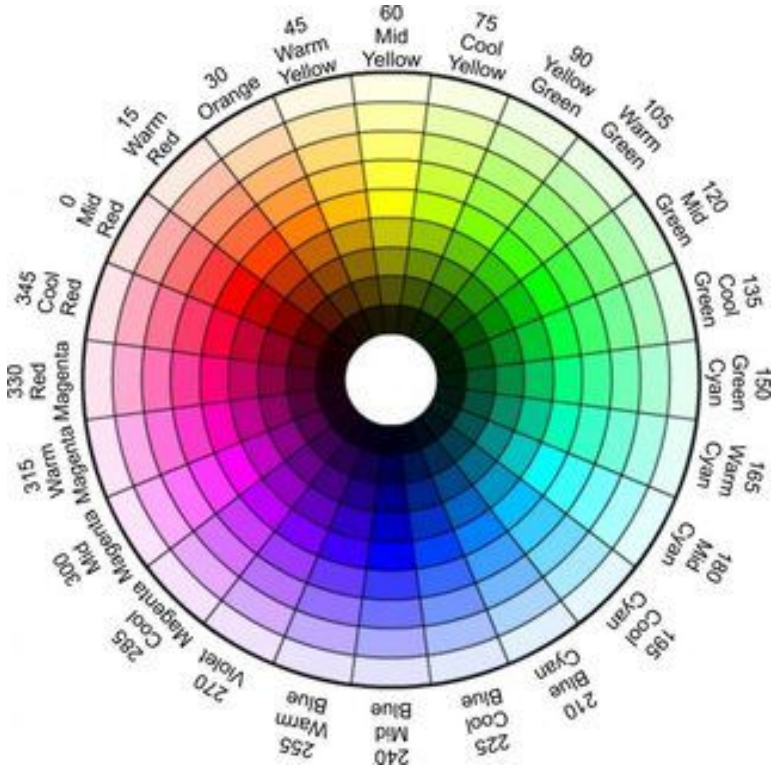
```
import cv2
import numpy as np

image = cv2.imread("SampleImages/antiqueTractors.jpg")
(bc, gc, rc) = cv2.split(image)

# each channel is shown as grayscale, because it only has value per pixel
cv2.imshow("Blue channel", bc)
cv2.imshow("Green channel", gc)
cv2.imshow("Red channel", rc)
cv2.moveWindow("Blue channel", 30, 30)
cv2.moveWindow("Green channel", 330, 60)
cv2.moveWindow("Red channel", 630, 90)
cv2.waitKey(0)

# Put image back together again
imCopy = cv2.merge((bc, gc, rc))
cv2.imshow("Image Copy", imCopy)
cv2.waitKey(0)
```

Other color representations

There are many other color representations that we could use. One useful one is called HSV (there are other similarly-named variants). In HSV, the basic color is defined by single value, the hue. Then the brightness or darkness of the color is given by saturation and value. The color wheel above provides HSV tools as well, experiment with this.

HSV is useful for color-tracking in images, because the hue changes relatively little when the brightness of the light changes, whereas all three channels in an RGB image change when the brightness changes.

The diagram below shows the usual mapping of hues in HSV, in the range from 0 to 360. OpenCV uses the range from 0 to 180, and divides every typical hue value by 2 (to fit them in the range of 0 to 255). For example, yellow shades in the diagram run from 45 to 60. In OpenCV the corresponding hues would be 22 to 30. Notice that red includes values from both ends of the range.



Beyond HSV, the YUV color encoding can also be very useful. In this encoding, the Y channel contains brightness or “luminance” information, and the U and V channels together describe the color. We use this color representation for balancing the brightness of an image, by converting to this format, modifying the Y channel, and then converting back. This YUV Colorspace web page gives some nice visuals showing how this representation works.

Milestone 3: Coding a fun color effect

Define a script that reads in a color image. It should split the channels of the image, like in the script above. Then, make a new tuple where the three channel variables are in a different order, and merge them back into one color image. Display it to see the result.

Next, modify your program so that it generates and displays a *random* reordering of the three channels every time you run the program. The easiest way to do this is:

1. Add a statement at the top of the file to import the `random` module: `import random`
2. Create a list that contains the three channel matrices you get from `cv2.split`
3. Call the `random.shuffle` function and pass it the list. It will modify the list and randomly reorder its elements
4. Make a new image by merging the shuffled channels in the list together
5. Display the new image

Related readings (if you want more background):

- RGB color model on Wikipedia
- Tutorial on binary numbers from Sparkfun

Numpy Basics

Images in OpenCV are represented as arrays from the Numpy module. Numpy (NUMerical PYthon) provides tools for efficient representation of 2d and 3d tables of numbers, and very efficient operations on those arrays of numbers.

Put the script below into a file and run it. Then, in the shell, you'll experiment with the resulting representation of the image. The command `cvtColor` can be used to convert between various representations of an image. In this case, it just makes a grayscale version of the given image. Grayscale images are more simple than color images, because they only have one value per pixel: the value represents a shade of gray, sometimes called brightness or luminance. The script also shows how to create a blank, black image from scratch, using the Numpy command `zeros`, and how to create a blank, white image using `ones`.

```
import cv2
import numpy as np

origImage = cv2.imread("SampleImages/snowLeo4.jpg")
gray = cv2.cvtColor(origImage, cv2.COLOR_BGR2GRAY)
cv2.imshow("Gray image", gray)
```

```
blankImg1 = np.zeros((400, 250), np.uint8)
cv2.imshow("Black background image", blankImg1)

blankImg2 = 255 * np.ones((300, 300), np.uint8)
cv2.imshow("White background image", blankImg2)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

After running the script above, add statements just before the call to `waitKey` that print various features of the four images , as listed in the table below. **Try printing each of the examples below on `origImage`, `gray`, `blankImg1`, and `blankImg2`. Helpful hint:** print a string along with the number that describes what the number represents, and print more than one value on a single line to avoid excessive output.

Examples	**Meaning**
`type(image)`	Evaluates to the type of data the image as a whole is
`image.shape`	Evaluates to a tuple giving the dimensions of the array (height, width) for grayscale images, and (height, width, depth) for color images
`image.size`	Evaluates to the total number of pixels in the image
`image.dtype`	Evaluates to the type of number used in the array, dtype stands for "data type"

The `shape` variable holds the dimensions of the array. The grayscale image and the original have the same rows and columns, but the color image has three channels for representing color. A grayscale image has only one channel. The blank image is a different size, and has only one channel, also. All three images have the same data, `uint8`, which stands for "**u**nsigned **int**eger **8** bits long." This corresponds to values between 0 and 255: $2^8 = 256$. The unsigned part means that the integers are assumed to be zero or greater, so the computer doesn't need to take a bit to represent the sign of the number.

Accessing data in a Numpy array

We can access parts of a Numpy array, including individual numbers, using a square-bracket notation that extends the access and slicing operations on Python data. Suppose we want to know the color values at position (20, 20) in the original image. We can access those values by typing `origImage[20, 20]`, or `origImage[20, 20, :]`. The first option says to return the

array that is at row 20 and column 20. The second uses the colon to indicate that we want all the values from the third dimension (the color channel dimension). If we wanted to access the entire first column, including all three channels: `origImage[:,0,:]`.

Try the script below, which prints selection portions of an image.

```
import cv2
import numpy as np

image = cv2.imread("SampleImages/canyonlands.jpg")
print("Value at row 20, column 20:", image[20,20], image[20, 20, :])
print("Row 5:")
print(image[5, :, :])
print("Column 0:")
print(image[:, 0, :])
print("Small section:")
print(image[20:60, 100:200, :])
cv2.imshow("Image", image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

WARNING: Numpy and OpenCV have different perspectives on array data, which leads them to order the dimensions of the array differently. Numpy existed before OpenCV was ported to Python, and OpenCV existed for a long time before the port as well. Thus we just have to live with their different perspectives.

Numpy views an array in row-major form: made up of rows and columns, and then additional dimensions if they exist. Thus when you specify a location for a Numpy command, you give (row, col) as the coordinates. The small section printed above starts at row 20 and goes to row 60, and includes columns from 100 to 200.

OpenCV views an image in terms of (x, y) coordinates: x is the horizontal dimension, and y is the vertical. Thus when you specify a location for an openCV command, you give (x, y) as the coordinates. This is backwards from what Numpy does.

Be very careful when specifying a pixel's location, or a range of locations, or the width and height of an image, and think about whether you are calling an OpenCV command or a Numpy command. If the first, give (x, y) as the ordering; if the second, give (row, column).

Regions of Interest

A “region of interest” (abbreviated ROI) for an image is a subsection of the image that we want to isolate and work with separately (perhaps it is a region where a specific color or object occurs). We can use existing Numpy tools to access an ROI.

Note: for memory efficiency, when you create an ROI, it is a new “view” on the original image data. It does not copy the data. Many times this is just what we want: we can focus on a small section and any changes we make show up in the original. You cannot use drawing operations on an ROI, however.

Try out the code below to see an illustration of how to make an ROI, and what happens when you manipulate it. This actually makes two ROIs: `faceROI` and `flipFace`. To see that `flipFace` is also an ROI (a view into the original image), add to this script and set its blue channel values to 128. Make sure to re-show all three windows to see the final, updated images.

```
import cv2

catImage = cv2.imread("SampleImages/snowLeo1.jpg")
faceROI = catImage[250:550, 570:860, :]

cv2.imshow("Orig", catImage)
cv2.imshow("Face", faceROI)
cv2.waitKey(0)

# set blue channel of this ROI to zero, notice change shows in original
faceROI[:, :, 1] = 0

cv2.imshow("Orig", catImage)
cv2.imshow("Face", faceROI)
cv2.waitKey(0)

# flip the face upside down by reversing the X direction and keeping the
others the same
flipFace = faceROI[::-1, :, :]
cv2.imshow("Flipped", flipFace)
cv2.waitKey(0)
```

Marking up Images

Now you know enough about colors and images to move forward with marking up images. OpenCV provides a set of tools that allow your program to draw shapes on an image. This can be useful to highlight things the computer vision algorithms are finding. In this section you will explore how colors and images are represented in OpenCV, how to create new images, and how to mark up an image by drawing shapes or writing text on it. You'll also learn how to save an image to a file, so you can store the results.

OpenCV drawing commands

The table below lists the basic drawing commands in OpenCV. You can find more commands and more details in the OpenCV documentation; each function has more optional inputs than are given here.

Examples	**Meaning**
`cv2.line(img, pt1, pt2, col)`	Takes in an image, and two points given as tuples that specify pixels (col, row), and a color, and draws a line on the image between those two points.
`cv2.rectangle(img, pt1, pt2, col)`	Takes in an image, and two points in the image, and a color, and draws a rectangle with one point as upper-left corner and one as lower-right.
`cv2.circle(img, pt, rad, col)`	Takes in an image, a center point, a radius, and a color, and draws a circle at that point.

`cv2.ellipse(img, pt,`
`  axes, angle,`
`  startAng, endAng,`
`  col)`
	Takes in an image and inputs that specify an ellipse or elliptical arc. The point is the center of the ellipse. Axes is also a tuple containing the length of major and minor axes. Angle indicates rotation of ellipse around center. Start and ending angles are how much of the ellipse to draw. **This is the most complex of these functions, the**

	ellipseDemo.py program in SampleCode explains these inputs.

`font = cv2.FONT_HERSHEY_SIMPLEX`
`cv2.putText(img, text, pt,`
`font, fSize, col)`
	Takes in an image, a string, a point for lower-left corner of text, a font and font size, and a color, and draws text as specified. Start font size at 1!

Important notes:

- Points for each function should be Python tuples of the form (x, y).
- Colors should be tuples containing three values (blue, green, red). Each value should be between 0 and 255
- Most commands take an optional line-thickness input. When negative, it causes a filled shape to be drawn.

The script below creates two new images from scratch and then illustrates each of the drawing functions; it saves the resulting pictures to two files. It uses two `numpy` commands: `zeros` and `ones`. The `zeros` function makes an n-dimensional matrix filled entirely with zeros. In this case, we give the dimensions we want (300 rows, 500 columns, 3 channels) and tell it to make the numbers be `uint8`, the special type we need for 0 to 255 values. The `ones` function makes a matrix filled with ones. This call has 500 rows, 300 columns, and 3 channels, all filled with `uint8`. Copy these commands into a script and run it. Note that the `imwrite` function saves an image to a file. The type of the file is given by the extension on the filename.

```
import cv2
import numpy as np

draw1 = np.zeros((300, 500, 3), np.uint8)
draw2 = 255 * np.ones((500, 300, 3), np.uint8)

cv2.line(draw2, (50, 50), (150, 250), (0, 0, 255))
cv2.rectangle(draw1, (10, 100), (100, 10), (0, 180, 0), -1)
cv2.circle(draw2, (30, 30), 30, (220, 0, 0), -1)
cv2.ellipse(draw1, (250, 150), (100, 60), 30, 0, 220, (250, 180, 110), -1)
font = cv2.FONT_HERSHEY_SIMPLEX
cv2.putText(draw1, "Hi, there", (10, 270), font, 1, (255, 255, 255))

cv2.imshow("Black", draw1)
cv2.imshow("White", draw2)

cv2.imwrite("blackPic.jpg", draw1)
```

```
cv2.imwrite("whitePic.jpg", draw2)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

First, read through this script. Then copy it to a new Python file in PyCharm, and run it. Finally, experiment with this script and the drawing commands above. Make sure you can draw lines, filled and unfilled rectangles, filled and unfilled circles, filled and unfilled ellipses and arcs, and text, all in different colors. Change the script so that it draws a smiley face in one window.

Note: the `imshow` function does not automatically update the window; you must call `imshow` after any changes to see the updated image.

Milestone 4: Marking up a leopard

Create a script program that displays `snowLeo2.jpg`. Draw a circle around the leopard's head, draw a filled-in rectangle around its body, and either draw a series of lines down its tail, or draw an elliptical arc underneath its paws. At the top left corner of the image, display the filename (`snowLeo2.jpg`) as text. Use distinct colors for each kind of shape.

How do you know where to place the shapes? The easy way is to guess at a location, see where it is drawn, and then adjust its location and repeat. Remember that (0, 0) is the upper-left corner, the x axis increases across the image to the right, and the y axis increases down the image.

Using the Computer's Camera

I've provided a lot of sample images for you, but you can also use the computer's camera. You can access and display the video stream as a sequence of images, and you can capture and save still frames from the video stream.
The table below lists the main functions you need in order to access frames from the computer's camera. First there is the function `VideoCapture`, which takes one input and creates a VideoCapture object. The one input specifies which camera connected to the computer should be used (zero is the default: the built-in camera if there is one). Three *methods* belonging to the VideoCapture object are shown, as well.

Examples	**Meaning**

`cap = cv2.VideoCapture(0)`	Creates a VideoCapture object connected to specified camera
`cap.isOpened()`	Boolean method returns true if camera connection succeeded
`ret, image = cap.read()`	Method returns two values, a code to tell if the image was read successfully, and an image from the video stream
`cap.release()`	Method disconnects from camera

Start with the script below to get images from the built-in camera. You are going to improve this script and learn some new tools in the process. The `range` function makes the `for` loop repeat for the number of times given. Notice that we need to release the camera at the end. Try changing the number being passed to the range function and see how it changes the time the program runs.

```
import cv2

vidCap = cv2.VideoCapture(0)
for i in range(300):
    ret, img = vidCap.read()
    cv2.imshow("Webcam", img)
    cv2.waitKey(10)

cv2.destroyAllWindows()
vidCap.release()
```

Deciding to quit early: if and break

The first improvement to the script is to allow the user to quit the program early. To do this, you will learn two new Python statements: `if` and `break`, and a new data type, Boolean. Boolean expressions and functions return `True` or `False`. One place we use Boolean expressions is in an `if` statement. The boolean expression serves as a question, and the program chooses which statements to perform, based on whether the answer to the question is `True` or `False`. If statements have the following general form:

```
if test1:
    statements done if test1 is True
```

```
elif test2:
    statements done if test1 is False and test2 is True
. . .
else:
    statements done if all tests are False
```

The `elif` and `else` parts are optional. The computer performs each test in order. When it finds a test that is `True`, then it performs the statements associated with that test, and then skips the rest of the `if` statement. The `else` must be the last case; if the computer gets to it, then it does the statements associated with `else`.

We are going to use an `if` statement to respond to input from the user. We can use the `waitKey` function to get input from the user. The function actually returns the numeric value of whatever key the user typed. Change the last step inside the loop, to be the first statement below, and then add the second statement below just after it.

```
x = cv2.waitKey(10)
print(i, chr(x & 0xFF))
```

The first line gets the number that corresponds to the character the user entered. The next line prints the loop variable (which passes through the loop we are on) and `chr` converts the number back to the character the user typed.

Important notes about `waitKey`: The `waitKey` function is more complicated than we saw at first.

- If the input to `waitKey` is 0, then the program stops and waits for the user to type something. If we did that here then we would see only a still image from the camera, and not a live feed. If the input is a positive integer, then `waitKey` waits that many **milliseconds** and then goes on whether the user types something or not. Waiting 10, 20, or even 50 milliseconds is unlikely to cause a noticeable delay to a video feed. You do want to wait long enough for the camera frame to be displayed in the window!
- Also, `waitKey` returns a value. It is a numeric code for what key the user typed, or -1 if the user did not type anything. We would like to convert that numeric code to the character it really is. But -1 is not a code for any keyboard character, and there are keys on the keyboard that generate strange codes. So the strange expression: `chr(x & 0xFF)`, is a way to handle any numeric code. You can use it without understanding it, but if you want to understand it:
    - `chr` converts an integer between 0 and 255 into its corresponding character in the ASCII table
    - `x & 0xFF` converts x to be an integer between 0 and 255. It treats x as a binary sequence and performs a “bitwise-and” between x and a binary sequence that is

all zeros above 255 and all ones below. The effect is to cut off any value above 255 and also to drop any negatives.

We still need to check for the user typing q, and when they do we need to make the loop quit. To do that, we will change the `if` statement so that it checks for q, and instead of printing, we will put a `break` statement. The `break` statement causes the current loop to stop repeating. Replace the `print` statement above with this:

```
userChar = chr(x & 0xFF)
if userChar == 'q':
    break
```

Indefinite looping

The `for` loop is a nice tool for looping in Python, but it has a downside. When you set up the loop it has a fixed number of repetitions, and it always stops when it reaches the end. Sometimes, we might want a loop to continue until some event happens, like looping until the user says to stop. To do this, we can use a different loop statement, the while loop. A `while` loop has a test, kind of like an `if` statement. It continues to repeat the indented statements, until the test becomes `False`. Here is the general form of the `while` loop:

```
while text:
    statements to be repeated (indented)
```

Change the video program so that it uses a `while` loop instead of a `for` loop, and it loops until the user type the ‘q’ key.

Related readings (if you want more background):

- Control Flow from *A Byte of Python*, Sections 9.1, 9.2, and 9.4

Milestone 5: Extending your video program

If you have been making the updates described in the preceding section, you should now have a program that displays the images from your webcam, continuing indefinitely until the user types q. Make the two extensions below to your video program.

Extension 1: Displaying mirrored video

Try to hold something up in front of the camera while your program is running, and then move the object to the different corners of the video image. Did you find yourself moving the wrong way? We often have trouble with left and right movements on a video feed, because we are more used to seeing a mirror image of ourselves. With a single line, you can create a "view" of the image data that is mirrored in the vertical axis, and you will find it much more intuitive to work with (many video-conferencing tools do exactly this when displaying the webcam view for us). Add the line below just after the `vidCap.read` line, and then change the image in `imshow` so that it displays `img2` instead.

```
img2 = img[:,::-1,:]
```

What is this doing? These are `numpy` commands that say to make a new view of the image where every row stays as is, every column is reversed, and the BGR values stay as is. The next section goes into more detail about using the Numpy representation directly.

Extension 2: Saving snapshots

Suppose that you wanted to be able to save snapshots of the video. Add to the `if` statement that checks the user's input character. Add an `elif` to the `if` statement that checks for another key (the space bar, or s, or whatever you like). When the user hits that, save the current image to a file.

Blending Images

Blending two images together is done by averaging the colors of corresponding pixels: the two red values get averaged to make a new red value, and similarly for the green and blue values. In OpenCV, averaging two images can be done with the `addWeighted` function, if the two images are the same size.

Cropping images to the same size

In the `SampleImages` folder, there are several sets of images that are the same size (`canyonlands.jpg` and `mushrooms.jpg`, or some of the coin images). For images that are not the same size, you can select a section of the larger image to match the size of the smaller

image, using square brackets to select the region you want. The script below illustrates this notation. Because this notation comes from Numpy rather than OpenCV, the first pair of numbers specifies the rows, and the second pair specifies the columns:

```
# This script shows how to crop a part of an image
import cv2
import numpy as np

# Crop first 100 rows and first 200 columns
image = cv2.imread("SampleImages/canyonlands.jpg")
subImg = image[0:100, 0:200]
cv2.imshow("Subimage", subImg)

# Crop two pictures to just the pixel locs they share
im1 = cv2.imread("SampleImages/grandTetons.jpg")
im2 = cv2.imread("SampleImages/mightyMidway.jpg")
(hgt1, wid1, dep1) = im1.shape
(hgt2, wid2, dep2) = im2.shape
newWid = min(wid1, wid2)
newHgt = min(hgt1, hgt2)
im1Crop = im1[0:newHgt, 0:newWid]
im2Crop = im2[0:newHgt, 0:newWid]
cv2.imshow("Im1 Cropped", im1Crop)
cv2.imshow("Im2 Cropped", im2Crop)
```

Blending

The `addWeighted` function takes two images that are the same size and shape, and it averages corresponding pixel/channel values together. The formula below shows how each value in the new image is computed, if v_1 and v_2 are corresponding pixel and channel values.

$$v = av_1 + bv_2 + c$$

The coefficients, a and b, determine how much each picture is weighted. And c is a constant amount added to each value.

Below are some examples illustrating how the `addWeighted` function works.

Examples	**Meaning**
`cv2.addWeighted(img1, 0.5,`	Returns a new image that is an equal blend of the

`img2, 0.5, 0)`	two input images
`cv2.addWeighted(img1, 0.3, img2, 0.6, 50)`	Returns a new image that is an unequal blend of the two inputs image, and that has added 50 to each value (brightening the result)

Milestone 6: Making a visual echo effect

- Start by writing a script that reads in two image and blends them evenly, like the first example above.
- Next, modify this script so that it replaces one of the two images with frames from the video camera, and it blends the frame with the other original image.
- Thirdly, we're going to blend an earlier camera frame with the current one to make an echo effect. For it to be really noticeable, we can't use the immediately preceding frame, we need to go further back. We'll use a Python list to hold the five previous frames, updating the list each time we get a new frame from the camera.
    - Just before the loop that reads video frames, read in a single frame from the camera, and then do the following line, which creates a list five long containing five references to the original frame:

      ```
      prevPics = [frame] * 5
      ```
    - Inside the loop, your program will read in another frame each time through the loop.
    - Inside the loop, after everything else that is done in the loop, insert the following two lines. These first add the new frame to the end of the list, and then remove the first frame of the list, so that the list stays five long and the oldest frame is at the zeroth position:

      ```
      prevPics.append(frame)
      prevPics.pop(0)
      ```
    - Lastly, change the blending line so that, instead of blending the current frame with an image from a file, it blends it with `prevPics[0].`
    - Experiment with changing the weights so that the old frame has a lower weight than the current frame (make sure the weights add up to 1.0 to keep the brightness of the picture constant). What happens if you make the old frame have the higher weight?