

→ **$h(x)$** → **02668838e0c58f3f**

1. Input Image

2. Hashing Function

3. Image Fingerprint

Fingerprinting Images for Near-Duplicate Detection

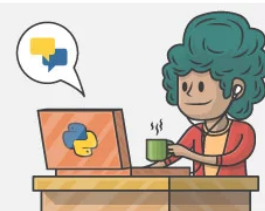
by [Real Python](#) 6 Comments [data-science](#) [machine-learning](#)

Table of Contents

- [What are we going to do?](#)
- [What is image fingerprinting/hashing?](#)
- [Why can't we use md5, sha-1, etc.?](#)
- [Where can image fingerprinting be used?](#)
- [What libraries will we need?](#)
- [Step 1: Fingerprinting a Dataset](#)
- [Step 2: Searching a Dataset](#)
- [Results](#)
- [Improving our Algorithm](#)
- [Summary](#)

A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You

pythonistacafe.com



This is a guest post by Adrian Rosebrock from [PyImageSearch.com](https://pyimagesearch.com), a blog all about computer vision, image processing, and building image search engines.

Updates:

1. 12/22/2014 - Removed references to line numbers in the code.
2. 05/22/2015 - Added dependency versions.

About five years ago, I was doing development work for a dating website. They were an early stage startup, but were starting to see some initial traction. Unlike other dating websites, this company marketed themselves with a squeaky clean reputation. This wasn't a site where you would go for hookups — this is where you went to find an honest *relationship*.

Fueled by millions in venture capital (this was prior to the recession in the United States), their online advertisements regarding true love and finding a soulmate converted like gangbusters. They were covered by Forbes. And they were even featured in a short spotlight on national television. These early successes lead to the coveted exponential growth in startups — their user ship was *doubling each month*. Things were looking extremely good for them.

But they had a serious problem — **a porn problem**.

A small subset of the dating website’s users were uploading pornographic images and setting them as their profile pictures. This behavior ruined the experience for many of their customers — leading to them canceling their memberships.

Now, perhaps for some dating websites even be considered “normal” or “expect

However, this behavior could be neither

Remember, this was a startup that had r that plagued other dating websites. In sl uphold.

Desperately, the dating website scrambl *literally nothing but stare at an administ* were uploaded to the social network.

They were literally throwing tens of thousands of dollars (not to mention, countless man-hours) at the problem, simply trying to moderate and contain the outbreak rather than stop it at its source.

The outbreak reached critical levels in July of 2009. For the first time in eight months, user ship failed to double (and had even started to decline). Worse yet, investors were threatening to pull their funding if the company did not solve the problem.

Indeed, the tides of filth were beating against the ivory tower, threatening to topple it into the sea.

As the knees of the dating giant started to buckle, I proposed a more robust and long-term solution: **What if we used image fingerprinting to combat the outbreak?**

You see, every image has a fingerprint. And just like a fingerprint can identify a person, it can also identify an image.

This lead to the implementation of a three stage algorithm:

1. Fingerprint our set of inappropriate images and store the image fingerprints in a database.
2. When a user uploaded a new profile picture, we compared it to our database of image fingerprints. If the fingerprint of the upload matched the fingerprint of any inappropriate images, we prevented the user from setting the image as their profile picture.
3. As image moderators flagged new pornographic images, they were also fingerprinted and stored in our database, creating an ever-evolving database that could be used to prevent invalid uploads.

Our process, while not perfect, worked. Slowly but surely, the outbreak slowed. It never fully stopped — but by using this algorithm we managed to reduce the number of inappropriate uploads **by over 80%**.

We also managed to keep the investors happy. They continued to fund us — until the recession hit. Then we were all out of a job.

Looking back, I can’t help but laugh. My job didn’t last. The company didn’t last. Even a few of the investors were left kicking rocks.

But one thing *did* survive. The image fingerprinting algorithm. Years later, I want to share the basics of this algorithm with you, in hopes that you can use it in your own projects.

But the biggest question is, how do we create this image fingerprint?

Read on to find out.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh  **Python Trick** 
code snippet every couple of days:

[Send Python Tricks »](#)

What are we going to do?

We are going to utilize image fingerprinting to perform near-duplicate image detection. This technique is commonly called “perceptual image hashing” or simply “image hashing”.

What is image fingerprinting/hashing?

Image hashing is the process of examining the contents of an image and then constructing a value that uniquely identifies an image based on these contents.

For example, take a look at the image at [this link](#) and compute an “image hash” based on that are “similar” as well”. Using image hashing is substantially easier.

In particular, we’ll be using the “difference hash” algorithm. Simply put, the dHash algorithm looks at the differences, a hash value is created.

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

[Send Python Tricks »](#)

Why can’t we use md5?

Unfortunately, we cannot use cryptographic hashing algorithms in our implementation. Due to the nature of cryptographic hashing algorithms, very tiny changes in the input file will result in a substantially different hash. In the case of image fingerprinting, we actually want our *similar inputs* to have *similar output hashes* as well.

Where can image fingerprinting be used?

Just like my example above, you could use image fingerprinting to maintain a database of inappropriate images — and alert users when they try to upload such an image.

You could build a reverse image search engine such as [TinEye](#) that keeps track of images and the associated webpages that they appear on.

You could even use image fingerprinting to help manage your own personal photo collection. Imagine you have a hard drive filled with partial backups of your photo library, but need a way to prune through the partial backups and keep only unique copies of an image — image fingerprinting can help with that as well.

Simply put, you can use image fingerprinting/hashing in nearly any setting where you are concerned with detecting near-duplicate copies of an image.

What libraries will we need?

In order to build our image fingerprinting solution, we’ll be utilizing three main Python packages:

- [PIL/Pillow](#) to facilitate reading and loading images.
- [ImageHash](#), which contains our implementation of dHash.
- And [NumPy/SciPy](#), which are required by ImageHash.

You can install all the required pre-requisites by executing the following command:

```
Shell
$ pip install pillow==2.6.1 imagehash==0.3
```

Step 1: Fingerprinting a Dataset

The first step is to fingerprint our image dataset.

And before you ask, no, we are not going to use pornographic images like my days working at the dating website. Instead, I have created an artificial dataset that we can use.

[Improve Your Python](#) 

Among computer vision researchers, the [CALTECH-101](#) dataset is legendary. It contains over 7,500 images from 101 different categories, including people, motorcycles, and airplanes.

From these ~7,500 images, I have randomly selected 17 of them.

Then, from these 17 randomly selected images, I have created N new images by randomly resizing them by +/- a few percentage points. Our goal here is to find these near-duplicate images — kind of like finding a needle in a haystack.

Want to create a similar dataset to work with? Download the [CALTECH-101](#) dataset, grab 17 or so images, then run the `gather.py` script found in the [repo](#).

Again, these images are identical in every dimension, we cannot rely on simple `md5` hashes. Instead, we also have similar hash fingerprints.

So let's get started by writing the code to work:

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh  **Python Trick** 
code snippet every couple of days:

[Send Python Tricks »](#)

Python

```
# import the necessary packages
from PIL import Image
import imagehash
import argparse
import shelve
import glob

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required = True,
    help = "path to input dataset of images")
ap.add_argument("-s", "--shelve", required = True,
    help = "output shelve database")
args = vars(ap.parse_args())

# open the shelve database
db = shelve.open(args["shelve"], writeback = True)
```

The first thing we'll do is import the packages we'll need. We'll use the `Image` class from `PIL` or `Pillow` to load our images off disk. Then the `imagehash` library can be utilized to construct the perceptual hash.

From there, `argparse` is used to parse command line arguments, `shelve` is used as a simple key-value database (Python dictionary) residing on disk, and `glob` is utilized to easily gather the paths to our images.

We then parse our command line arguments. The first, `--dataset` is the path to our input directory of images. The second, `--shelve` is the output path to our `shelve` database.

Next, we open our `shelve` database for writing. This `db` will store our image hashes. More on that next:

Python

```
# loop over the image dataset
for imagePath in glob.glob(args["dataset"] + "/*.jpg"):
    # load the image and compute the difference hash
    image = Image.open(imagePath)
    h = str(imagehash.dhash(image))

    # extract the filename from the path and update the database
    # using the hash as the key and the filename append to the
    # list of values
    filename = imagePath[imagePath.rfind("/") + 1:]
    db[h] = db.get(h, []) + [filename]

# close the shelf database
db.close()
```

Improve Your Python



Here is where the bulk of the work happens. We start looping over our dataset of images on, load it from disk, and then create the image fingerprint.

Now, we reach the two most important lines of code in this entire tutorial:

Python

```
filename = imagePath[imagePath.rfind("/") + 1:]
db[h] = db.get(h, []) + [filename]
```

Like I mentioned earlier in this post, images with the same fingerprint are considered to be identical.

Thus, if our goal is to find near-identical value.

And that's exactly what those lines do.

The former extracts the filename of the image hash.

To extract image fingerprints from our d

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh  **Python Trick** 
code snippet every couple of days:

[Send Python Tricks »](#)

Shell

```
$ python index.py --dataset images --shelve db.shelve
```

The script will run for a few seconds and once it is done, you'll have a file named `db.shelve` that contains the key-value pairs of image fingerprints and filenames.

This same basic algorithm is what I utilized years ago when I was working for the dating startup. We took our dataset of inappropriate images, constructed an image fingerprint for each image, and then stored them in our database. When a new image arrived, I simply computed the hash of the image and checked to database to see if the upload had already been flagged for invalid content.

In the next step I'll show you how to perform the actual search to determine if an image already exists in the database with the same hash value.

Step 2: Searching a Dataset

Now that we have built a database of image fingerprints, it's time to *search* our dataset.

Open up a new file, name it `search.py`, and we'll get coding:

Python

```
# import the necessary packages
from PIL import Image
import imagehash
import argparse
import shelve

# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required = True,
    help = "path to dataset of images")
ap.add_argument("-s", "--shelve", required = True,
    help = "output shelve database")
ap.add_argument("-q", "--query", required = True,
    help = "path to the query image")
args = vars(ap.parse_args())
```

Once again we'll import our relevant packages on. We then parse our command line arguments on. We'll need three switches, `-dataset`, which is the path to our original dataset of images, `-shelve`, the path to where our `shelve` database of key-value pairs resides, and `-query`, the path to our query/uploaded image. Our goal will be to take the query image and determine if it already exists in our database.

[Improve Your Python](#)

Now, let’s write the code to perform the actual search:

Python

```
# open the shelve database
db = shelve.open(args["shelve"])



# load the query image, compute the difference image hash, and
# and grab the images from the database that have the same hash
# value
query = Image.open(args["query"])
h = str(imagehash.dhash(query))
filenames = db[h]
print "Found %d images" % (len(filenames))

# loop over the images
for filename in filenames:
    image = Image.open(args["dataset"] + filename)
    image.show()

# close the shelve database
db.close()
```

```
1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python

...with a fresh  **Python Trick** 

code snippet every couple of days:

Email Address

Send Python Tricks »

We first open our database, and then we loop over these images and display them to our screen. If there are any images with the same hash value, we loop over these images and display them to our screen. Using this code we will be able to determine if an image already exists in our database using nothing but the fingerprint value.

Results

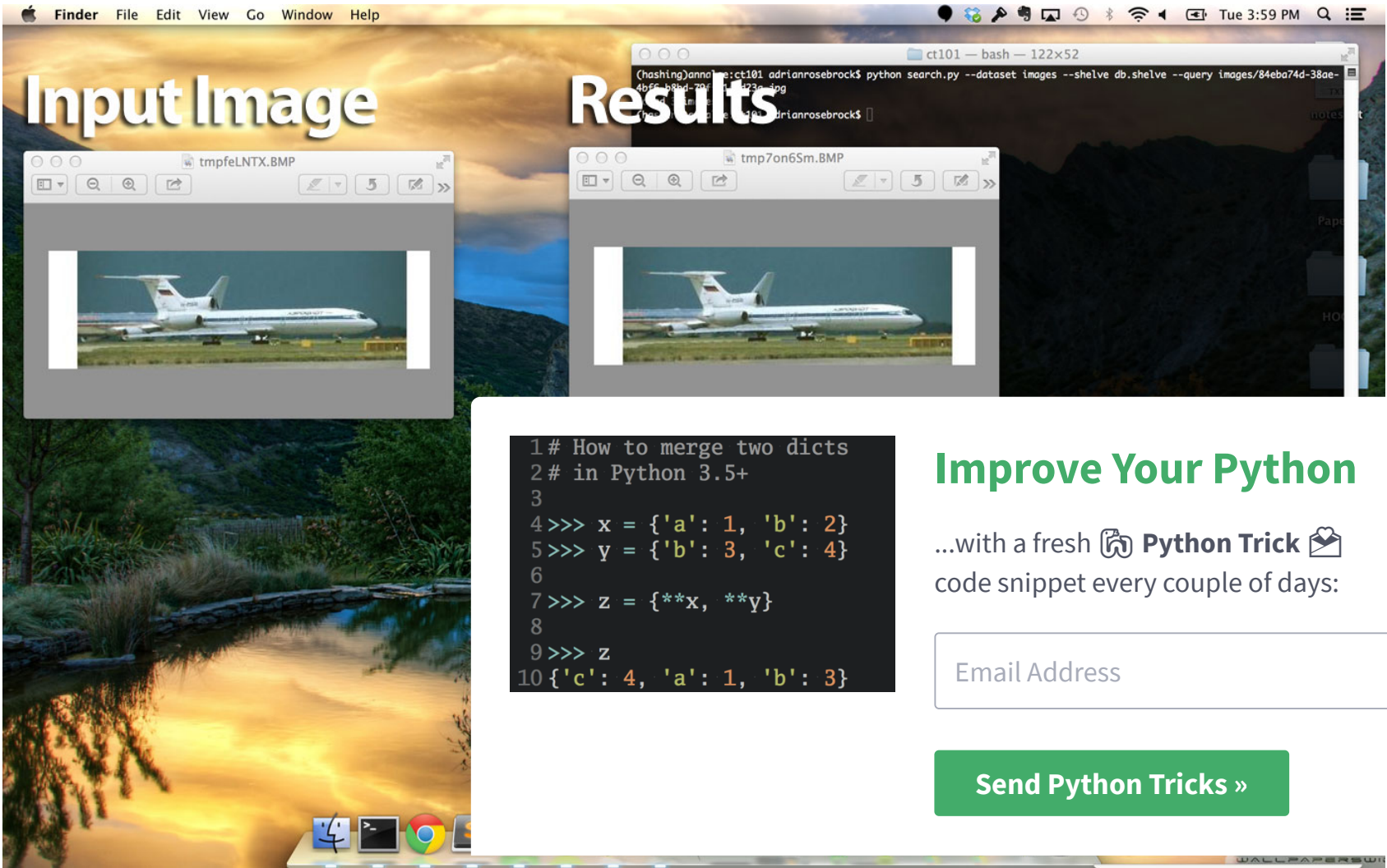
As I mentioned earlier in this post, I have taken the original ~7,500 images from CALTECH-101 dataset, randomly selected 17 of them, and then generated *N* new images by randomly resizing them by a few percentage points. The dimensions of these images are only different by a handful of pixels — but because of this we cannot rely on the md5 hash of the file (this point is elaborated on further in the “Improving our Algorithm” section). Instead, we need to utilize image hashing to find the near-duplicate images.

Open up your terminal and execute the following command:

Shell

```
$ python search.py --dataset images --shelve db.shelve --query images/84eba74d-38ae-4bf6-b8bd-79ffa1dad23a.jpg
```

If all goes well you should see the following results:



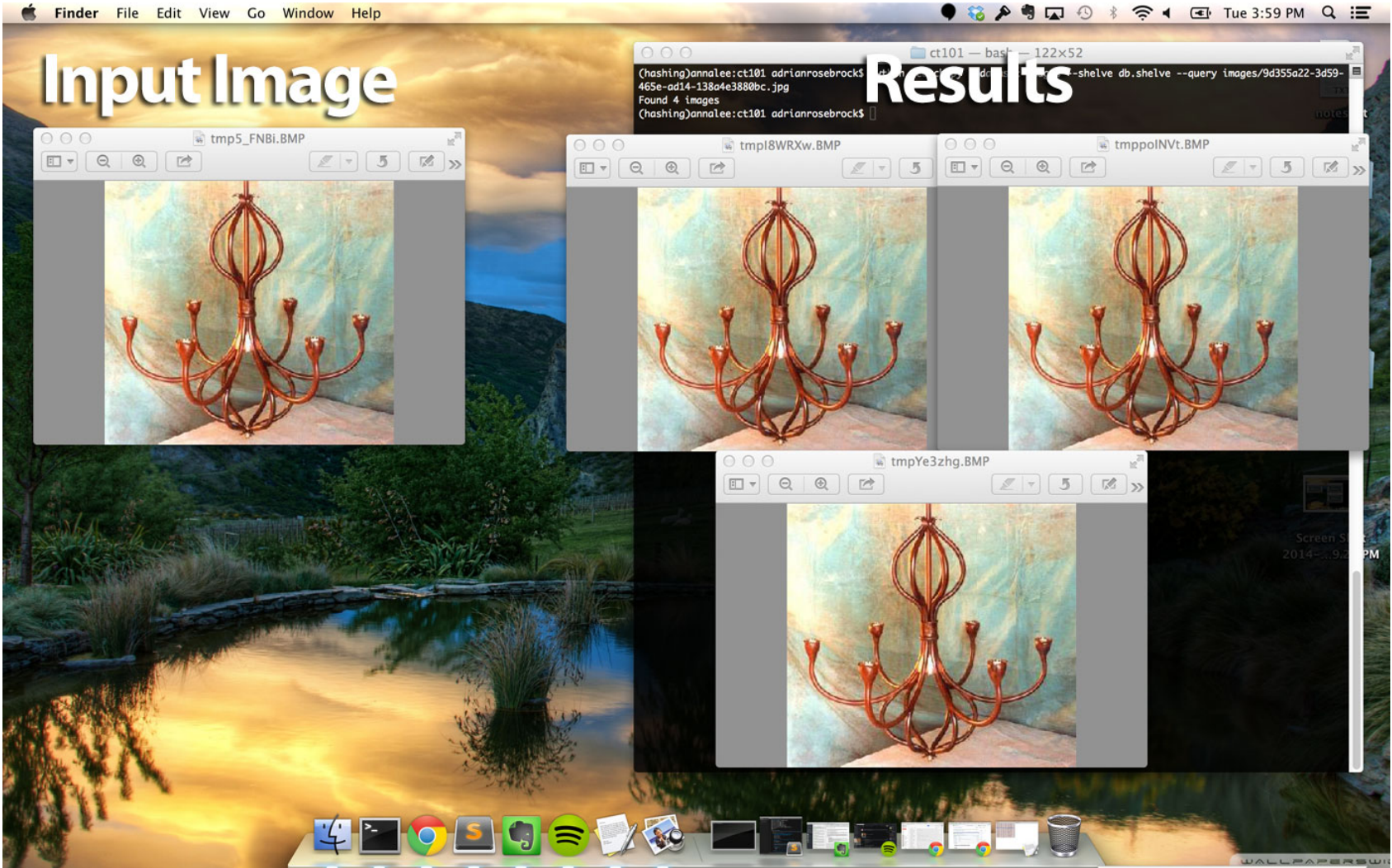
On the **left** we have our input image. We take this image, compute its image fingerprint, and then lookup the fingerprint in our database to see if any other images have the same fingerprint.

Sure enough – there are two other images in our dataset that have the exact same fingerprint, as shown on the **right**. While it’s not entirely obvious from the screenshot, these images, while having the exact same visual content, are not entirely identical! All three images have different widths and heights.

Let’s try another input image:

```
Shell
$ python search.py --dataset images --shelve db.shelve --query images/9d355a22-3d59-465e-ad14-138a4e3880bc.jpg
```

And here are the results:



Once again we have the input image on the **left**. Our image fingerprinting algorithm was able to find three identical images with the same fingerprint, as displayed on the **right**

Improve Your Python

One last example:


Shell

\$ python search.py --dataset images --shelve db.shelve --query images/5134e0c2-34d3-40b6-9473-98de8be16c67.jpg

Finder File Edit View Go Window Help

Input Image

tmppswiM5.BMP



Results

ct101 — bash — 122x52

(hashing)annolee:ct101 adrianrosebrock\$ python search.py --dataset images --shelve db.shelve --query images/5134e0c2-34d3-40b6-9473-98de8be16c67.jpg
Found 4 images
(hashing)annolee:ct101 adrianrosebrock\$

1# How to merge two dicts
2# in Python 3.5+
3
4>>> x = {'a': 1, 'b': 2}
5>>> y = {'b': 3, 'c': 4}
6
7>>> z = {**x, **y}
8
9>>> z
10{'c': 4, 'a': 1, 'b': 3}

Improve Your Python

...with a fresh Python Trick code snippet every couple of days:

Email Address

Send Python Tricks »

Screen Shot 2014-09-03 at 4:00 PM

This time our input image is a motorcycle on the *left*. We took this motorcycle image, computed its image fingerprint, and then looked it up in our database of fingerprints. As we can see on the *right*, we were able to determine that there are three other images in the database with the same fingerprint.

Improving our Algorithm

There are many ways to improve our algorithm — but the most crucial way is to take into consideration hashes that are *similar*, but not *identical*.

For example, images in this post were only resized (either up or down in scale) by a few percentage points. If an image were resized by a larger factor, or if the aspect ratio was changed, the hashes would not be identical.

However, the images would still be *similar*.

In order to find similar but not identical images, we would need to explore the [Hamming distance](#). The Hamming distance can be used to calculate the number of bits in a hash that are *different*. Therefore, two images that have only a 1-bit difference in hashes are substantially more similar than images that have a 10-bit difference.

However, we then run in to a second problem — the scalability of our algorithm.

Consider this: We are given an input image and are instructed to find all similar images in our database. We then have to compute the Hamming distance between our input image and *each and every single image in the database*.

As the size of our database grows, so will the time it takes to compare all hashes. Eventually, our database of hashes will reach such a size where this linear comparison is not practical.

The solution, while outside the scope of this post, is to utilize [K-d trees](#) and [VP trees](#) to reduce the complexity of the search problem from linear to sub-linear.

Summary

In this blog post we learned how to construct and utilize image hashes to perform near-duplicate image detection. These image hashes are constructed using the visual content

And just like a fingerprint can identify a person, an image hash can uniquely identify an image as well.

Using our knowledge of image fingerprinting, we then built a system to find and recognize images with similar content using nothing but the image hash.

We then demonstrated how this image hash can be used to rapidly find images with near-duplicate content.

Oh and be sure to grab the code from the [repo](#).

Learn computer vision in a single weekend: If you enjoyed this post and want to learn more about computer vision, image processing, and building image search engines, just head over to my blog at [PyImageSearch.com](#)

Cheers!



Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe at any time. [Team.](#)

Email Address

Send Me Python Tricks »

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```



Improve Your Python

...with a fresh  **Python Trick**  code snippet every couple of days:

Email Address

Send Python Tricks »

```
1 # How to merge two dicts
2 # in Python 3.5+
3
4 >>> x = {'a': 1, 'b': 2}
5 >>> y = {'b': 3, 'c': 4}
6
7 >>> z = {**x, **y}
8
9 >>> z
10 {'c': 4, 'a': 1, 'b': 3}
```

Improve Your Python with  Python Tricks 

Get a short & sweet Python code snippet delivered to your inbox every couple of days:

» [Click here to see examples](#)

What Do You Think?

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won’t make the cut here.

Improve Your Python

https://realpython.com/fingerprinting-images-for-near-duplicate-detection/#what-are-we-going-to-do

9/12