dmtools

Release 0.0.1rc1

Henry Robbins

Nov 12, 2021

CONTENTS

dmtools (Digital Media Tools) is a Python package providing low-level tools for working with digital media programmatically. The netpbm module allows one to read and create Netpbm images. Color space transformations can be done with the colorspace module. Using ffmpeg, the animation module can export .mp4 videos formed from a list of images and the sound module can be used to add sound to these videos as well. Lastly, ASCII art can be produced with the ascii module.

For those experienced with installing and using Python packages, you can find brief installation instructions in the README. The installation instructions found here are aimed at beginner users. First, we will install a programming language called Python. Next, we will install dmtools, a Python package. The last section is optional and a little more intensive. It walks through the installation of a program called FFmpeg which is required if you wish to create videos with dmtools.

CHAPTER

ONE

INSTALLING PYTHON

In order to use dmtools, you will need to install the Python programming language. We preface the Python installation instructions with a breif Q&A. This section is ordered so that each answer naturally leads into the following question so it is best read in order.

1.1 Q&A

"What is Programming Language?"

The purpose of a programming language is to allow us to give instructions to a computer. At first, this may seem foreign. However, every time you interact with a computer, you are giving it instructions to do certain tasks like which website to navigate to, what document to open, etc.. The difference is in the way you are communicating that information. You are most likely familiar with Graphical User Interfaces (GUIs). These are programs which provide graphical ways of giving the computer instructions using the keyboard and mouse to point and click.

"How does a programming lanaguage let us give instructions to a computer?"

Without getting into too much detail, programming langauges are just like human languages. They have syntax which defines the structure of the language and they have semantics which define the meaning of certain structures in the language. Following these rules, we can write up a set of instructions and it off to the computer to execute.

"This sounds complicated. Why would I use this instead of a program with a nice GUI?"

There are two main reasons: humans are lazy and flexibility. Often times, there are tasks on the computer that are extremely repetitive. Unlike GUIs, programming languages don't require the human to be very involved. We only need to give the instructions once and the computer will go on chugging away until we tell it to stop. In terms of flexibility, it may seem that programs like Photoshop and Google Docs have an endless number of tabs, knobs, and dials but their flexibility pales in comparison to programming languages. With a programming language, the limit is quite literally, your imagination.

"What is Python?"

Yes, Python is a programming language. But, there are many different ways to classify programming languages. There are many characteristics of Python but the one we wish to emphasize here is that it is a general-purpose scripting language. Scripting lanaguages "automate the exectution of tasks that would otherwise be performed indiviually by a human operator." It is simple in that files written in the language can be run as scripts where the computer just goes through the file linearly, executing each task as it is given.

1.2 Anaconda

To install Python, we will use Anaconda which provides an extremely popular Python distribution called Anaconda Individual Edition. Navigate to the link and scroll to the bottom to select the Anaconda Installer for your operating system. Choose the Graphical Installer.

To verify you now have Python, open up a terminal (the Terminal Application on macOS) and run `python` to open up a Python prompt (a place where Python instructions can be run). The line beginning with >>> is where you can type Python code and run it. Type `print("Hello World!")` and hit Enter. It should display `Hello World!` as the result of the command! You can then type `quit()` or CTRL+D to exit the prompt.

```
python
Python 3.8.8 (default, Apr 13 2021, 12:59:45)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World!")
Hello World!
>>> quit()
```

You now have Python installed on your computer!

CHAPTER

TWO

INSTALLING DMTOOLS

In this section, we will install the dmtools Python package. But first, what is a Python package? A Python package is essentially pre-bundled Python code that provides some functionality. For example, NumPy is a Python package (one you will get more familiar with in *Working with Images in NumPy*) that allows for easy manipulation of arrays. Python packages are your friend! They allow you to easily use other people's code so you never have to re-invent the wheel and can spend more time being creative.

In installing anaconda, you should now have a program called pip which stands for Pip Installs Packages. It is a Python package manager and it is the tool we will use to install dmtools. Just run the following line.

```
pip install dmtools
```

To the verify the installation worked correctly, open a Python prompt by typing `python` and then type `from dmtools import netpbm`. If you don't get any error messages, the instllation was a success!

```
python
Python 3.8.8 (default, Apr 13 2021, 12:59:45)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from dmtools import netpbm
>>> quit()
```

CHAPTER

THREE

INSTALLING FFMPEG (OPTIONAL)

1. This section is not optional of you wish to create videos with dmtools
2. Currently, these installation instructions focus on macOS users. For installation instructions on other operating systems, see Download FFmpeg.

In order to install FFmpeg, we will first need to install a package manager. A package manager functions similarly to an app store–it provides a way of installing and managing computer programs "in a consistent manner." Homebrew is a package manager for macOS. It is the one we will use to install FFmpeg. To install it, paste the following line in macOS Terminal.

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
↪install.sh)"
```

When running the above line, you will likley be prompted to install Command Line Tools (CLT) for Xcode. This can be installed with

```
xcode-select --install
```

To verify Homebrew was installed properly, run `brew` in Terminal and you should recieive a help page on various Homebrew commands. With Homebrew now installed, you can easily install FFmpeg with

```
brew install ffmpeg
```

This installation may take some time. Once complete, verify it was installed properly by running `ffmpeg` in Terminal. It should return some FFmpeg version information.

Congratulations! You have now installed a package manager and FFmpeg. You will now be able to create videos using dmtools.

CHAPTER

FOUR

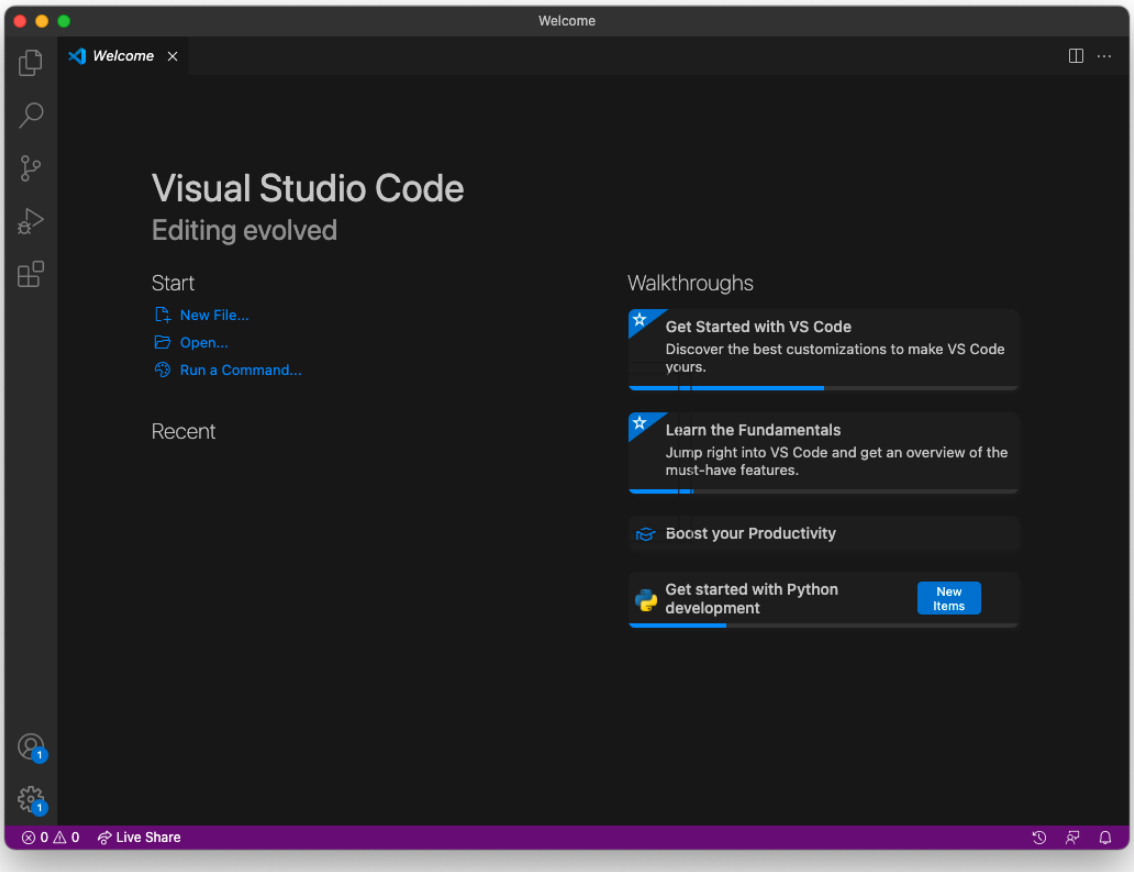
TUTORIALS

This section includes a few tutorials to get you up and running and using dmtools effectivley. The *Introduction to Python* assumes you have installed VS Code. The first tutorial walks you through this installation. The next two tutorials provide a quick introduction to Python and the NumPy python package. If you are familiar with all of these tools, you can skip to the final tutorial which provides an introduction to dmtools.

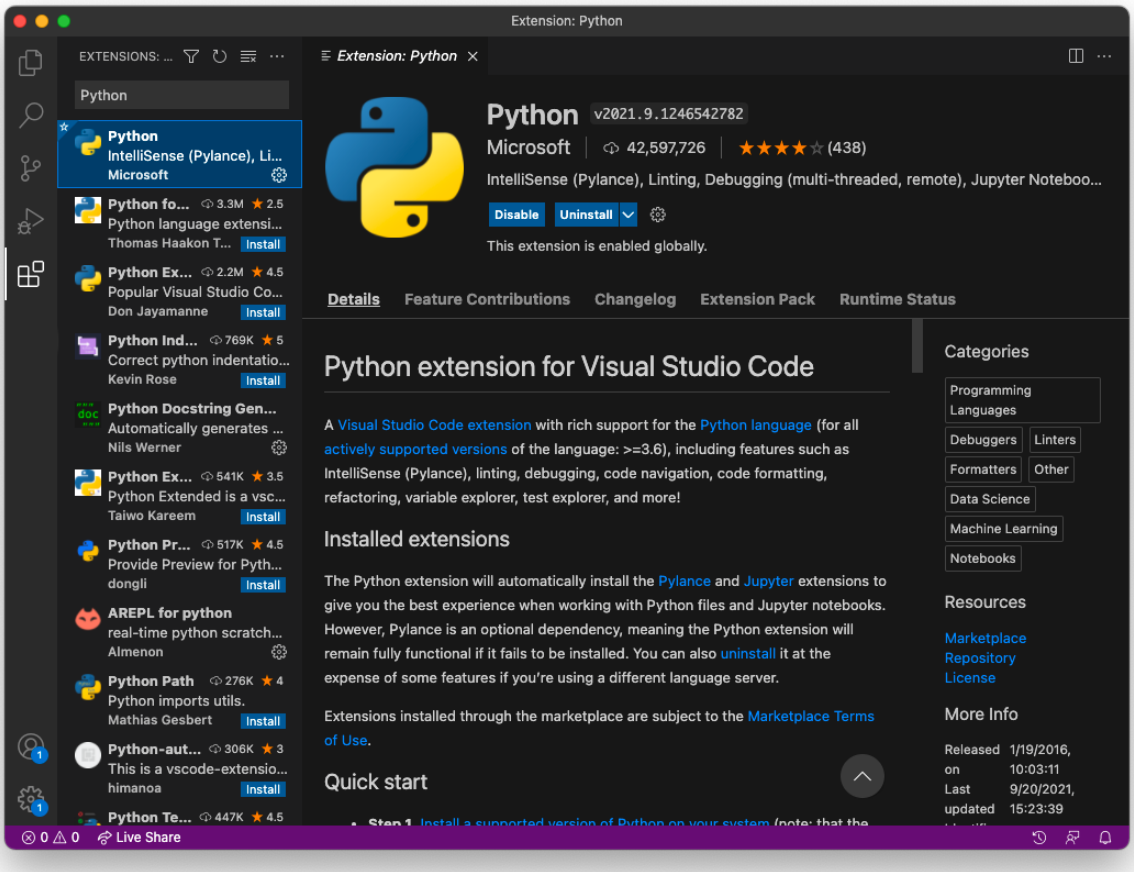
4.1 Installing VS Code

In *Introduction to Python*, we will learn how to create Python scripts. We will write these Python scripts in a text editor. On macOS, TextEdit is the default text editor. However, you may want to use another text editor like VS Code (recommended), Notepad++, or Vim. This tutorial walks through the installation of VS Code in addition to a few nice extensions.

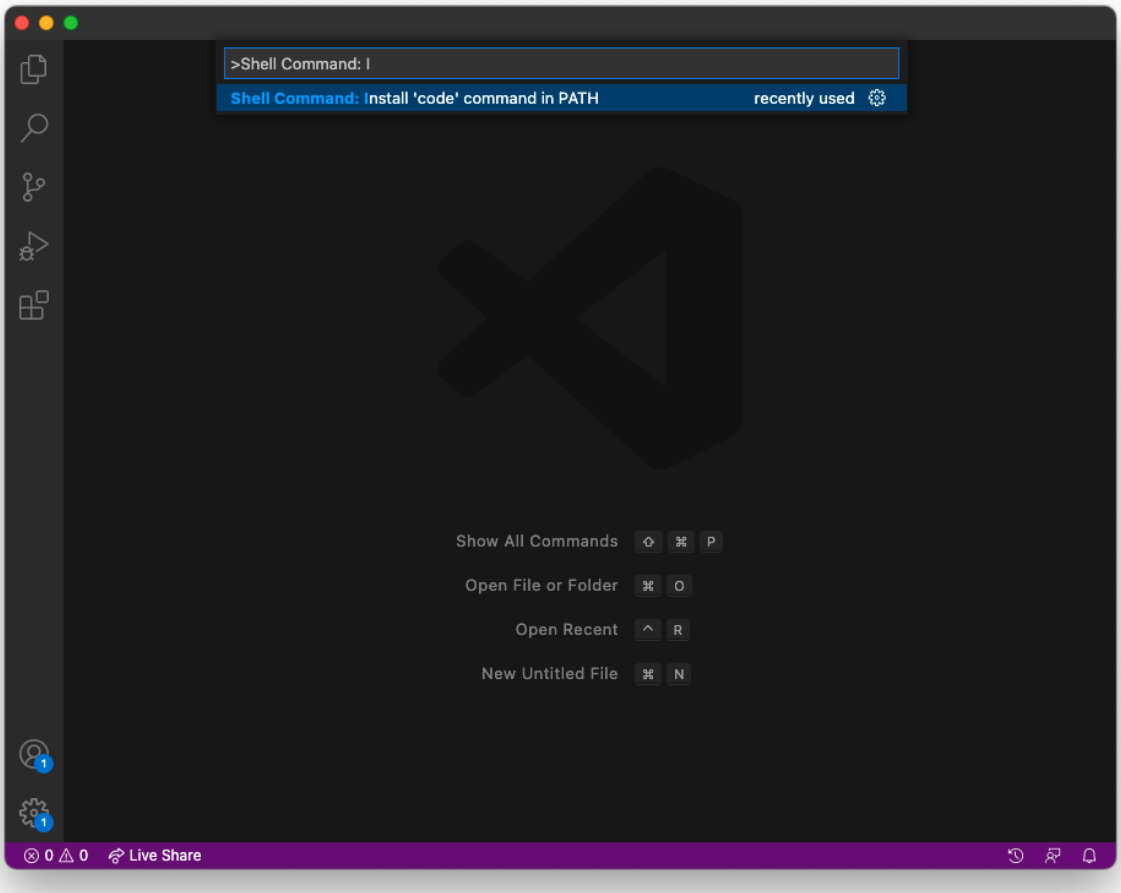
First, install VS Code. Upon opening the application, the window will look something like this.



We will install the Python Extension for VS Code. This provides "rich support for the Python language." On the side bar, you will see a menu item with four boxes. This is the extensions tab. Search for the Python extension and install it.



Lastly, go to `View > Command Palette` and search for `Shell Command: Install 'code' command in PATH`. Run this.



You have now successfully installed VS Code and are ready for the *Introduction to Python*!

4.2 Introduction to Python

This tutorial will walk through a short introduction to Python with emphasis on the neccessary basics for using dmtools and working with images. To get the most out of this tutorial, it is recommeneded to follow along by running all the code snippets yourself.

4.2.1 Python Scripts

In the *Installation* section, you saw how you can open up a Python prompt in a terminal where you can enter Python commands and run them. Anything more than a one line command is going to be bothersome to write in a prompt. Furthermore, it will not be repeatable. The python script addresses these issues. A Python script is a text file with the `.py` extension. We can write these files in any text editor. This tutorial assumes you are using VS Code (see *Installing VS Code*). After creating a Python script, you can run the script from a terminal with `python hello_world.py`. However, there is one catch: you must be in the directory where the `hello_world.py` script is located. Hence, we need to be able to navigate directories while in a terminal.

4.2.2 Navigating Directories in a Terminal

When you open up a terminal, you will see a prompt like the following:

```
(base) Name-Of-Machine:~ Name-Of-User$
```

Let's break down what information is contained in this prompt. The `(base)` let's us know that we are in the base Python envrionment. Don't worry too much about what that means. The `Name-Of-Machine` and `Name-Of-User` tell us (you guessed it) the name of the machine / computer and the user. The `$` is just an indication that it is the end of the prompt and you can type your command. Most pertinent is the `~` which indicates where in the file structure we are located. This is called the current working directory. The `~` indicates we are in the home directory.

We can navigate the file structure in a terminal the same way we would in Finder on macOS. Run the command `ls` by typing `ls` after the prompt and pressing `Enter`. You should see a list of the files and directories in the current working directory.

```
(base) Name-Of-Machine:~ Name-Of-User$ ls
Applications    Library
Desktop         Movies
Documents       Music
Downloads       Public
```

We can go into one of these directories with the `cd` command. For example, the command `cd Desktop` will change our working directory to the `Desktop` directory.

```
(base) Name-Of-Machine:~ Name-Of-User$ cd Desktop
(base) Name-Of-Machine:Desktop Name-Of-User$ ls
DesktopItem1    DesktopItem3
DesktopItem2    DesktopItem4
```

To go back one directory, use `cd ..`. To go back all the way to the home directory, use `cd ~`. Lastly, you can specify a file path to avoid running multiple `cd` commands.

```
(base) Name-Of-Machine:~ Name-Of-User$ cd Music
(base) Name-Of-Machine:Music Name-Of-User$ cd Artist
(base) Name-Of-Machine:Artist Name-Of-User$ cd NewAlbum
(base) Name-Of-Machine:NewAlbum Name-Of-User$ ls
Song1   Song3
Song2   Song4
```

```
(base) Name-Of-Machine:~ Name-Of-User$ cd Music/Artist/NewAlbum
(base) Name-Of-Machine:NewAlbum Name-Of-User$ ls
Song1   Song3
Song2   Song4
```

These few commands are really all you need to know to navigate directories while in a terminal!

4.2.3 Hello World!

In this section, we will create a directory where we will put our Python scripts and create our first script.

First, open up a terminal. Run the command `mkdir scripts` to create a directory called scripts. You can then `cd` into it a run `ls` to see that there is nothing in it yet.

```
(base) Name-Of-Machine:~ Name-Of-User$ ls
Applications    Library
Desktop         Movies
Documents       Music
Downloads       Public
(base) Name-Of-Machine:~ Name-Of-User$ mkdir scripts
(base) Name-Of-Machine:~ Name-Of-User$ ls
Applications    Movies
Desktop         Music
Documents       Public
Downloads       scripts
Library
(base) Name-Of-Machine:~ Name-Of-User$ cd scripts
(base) Name-Of-Machine:scripts Name-Of-User$ ls
(base) Name-Of-Machine:scripts Name-Of-User$
```

Now, let's create our first Python script! Rather than opening VS Code in the traditional way you open applications, we will open it from the terminal. This is because it will automatically put the files we create in the working directory which will prevent us from running into issues when trying to run our Python scripts. Make sure you are still in the `scripts` directory and run `code .` to open VS Code (don't close your terminal because you will need it to run the Python scripts). You will see a file navigation window on the left. Create a new file called `hello_world.py`.

```
# hello_world.py

print("Hello World!")

# Expected Output:
# Hello World!
```

The lines with # at the beginning are just comments in the Python code. You do not need to include them but they can give helpful information! Save the file and try running `python hello_world.py` in the terminal.

```
(base) Name-Of-Machine:scripts Name-Of-User$ python hello_world.py
Hello World!
```

You just created your first Python script! The remainder of this tutorial will walk you through the basics of Python through multiple example Python scripts . Again, it is recommended you follow along by creating and running these scripts. Even better, try modifying them to see if the output changes as you would expect!

4.2.4 Math

We can add, multipy, subtract, and divide numbers quite easily. What if we want to use some more complex math functions like the sine function? A lot of these are provided by a package called NumPy (which we will look at much closer in *Working with Images in NumPy*). To access these functins, we first need to import the package with `import numpy as np`. We can then use `np.sin()` to apply the sine function to some value. The math package also provides some useful functions you may want to use like the floor and ceiling function.

```
# simple_math.py

# addition, subtraction, multiplication, and division
print(1 + 1)  # 2
print(3 - 1)  # 2
print(1 * 2)  # 2
print(4 / 2)  # 2.0

# exponents
print(3**2)        # 9
print(9**(1 / 2))  # 3

# math functions from numpy
import numpy as np
print(np.sin(1))  # 0.8414709848078965
print(np.sin(np.pi / 2))  # 1.0

# math functions from math
import math
print(math.floor(0.5))  # 0
print(math.ceil(0.5))  # 1
```

4.2.5 Variables

It is often helpful to assign a name to a value. This is called a variable. In the script below, we set the variable `x` to be `1` and `y` to be `2`. We can then use these variables just like they were the values we assigned them to.

```
# variables.py

x = 1
y = 2
z = x + y

print(x + y)  # 3
print(y * 2)  # 4
print(z)      # 3
```

4.2.6 Loops

If we want to do the same command multiple times, we can use a loop. A loop has the syntax `for i in range(n)` where `n` is the number of times we will run through this loop. The variable `i` starts at zero and is incremented by one every time we run through the loop. The lines of code that are run in every iteration of the loop make up the loop body. We indent the lines that are in the loop body.

```
# loops.py

for i in range(5):
    print(i)

x = 0
for i in range(5):
    x = x + i
print(x)

# Expected Output:
# 0
# 1
# 2
# 3
# 4
# 10
```

4.2.7 Conditional Statements

What if we want to run a line of code only if a certain condition holds? These are called conditional statements. To compare values, we can use == for equals and `!=` for not equals. Note that `x = 2` assigns variable `x` the value 2 while `x == 2` returns if `x` has value 2 or not. Next, we need the syntax for boolean operators like and, or, and not. The operator `x & y` returns `True` if both `x` and `y` are `True`. The operator `x | y` returns `True` if at least one of `x` or `y` are `True`. Lastly, we have the syntax for the conditional statement which is `if x:` where the body of the conditional statement runs if `x` is `True`. The body of the conditional statement is denoted with indentation like the loop body.

```
# condition.py

x = True
y = False

print(x == True)   # True
print(x == False)  # False
print(x != False)  # True

print(not x)  # False

print(x & y)        # False
print(x & (not y))  # True
print(x | y)        # True
print((not x) | y)  # False

if True:
    print('True!')
```

(continues on next page)

(continued from previous page)

```
if False:
    print('This will not print.')

if x | y:
    print('True!')
```

4.2.8 Lists

Sometimes we have a list of values we care about and not just a single value. We can represent these as a list. For example, `x = [1,2,3]` is a list of three integers. We can then access the value at a certain index with the notation `x[i]` where `i` is the index of the value we want. Python is zero-indexed which means that the first value in a list has index zero. We can add values to lists with `.append()`. We can also add lists together.

```
# lists.py

x = [1, 2, 3]
y = [4, 5]

print(x)     # [1, 2, 3]
print(x[0])  # 1
print(x[2])  # 3

x.append(4)
print(x)     # [1, 2, 3, 4]

print(x + y) # [1, 2, 3, 4, 5]
```

4.2.9 List Comprehension

One of the many nice features in Python is called list comprehension. It allows us to initialize a list. It essentially combines the syntax for a list with the syntax for a loop allowing us to define a list with less code.

```
# list_comprehension.py

# without list comprehension

x = []  # this list is empty
for i in range(4):
    x.append(i)
print(x)  # [0, 1, 2, 3]

# with list comprehension

x = [i for i in range(4)]
print(x)  # [0, 1, 2, 3]

y = [i**2 for i in range(4) if i**2 != 4]
print(y)  # [0, 1, 9]
```

4.2.10 Functions

One of the most important concepts in programming is the function. Functions allow us to avoid writing the same code multiple times. A function has parameters or inputs. Sometimes it has an output and other times, it does not. The notation for a function declaration in Python is `def f(x, y, ...):` where `f` is the name of the function and `x, y, ...` are the function parameters. Like we have seen before, we use indentation in Python to denote the function body. This is the code that will be run every time the function is called. If the function will return a value, we use the keyword `return`. Lastly, when we want to call the function, we use the notation `f(x, y, ...)` where `f` is the name of the function and `x, y, ...` are the arguments or actual values we are assigning the parameters of that function.

```
# functions.py

# this function returns something
def add(x, y):
    return x + y

x = add(2,3)
y = add(7,8)
print(x)       # 5
print(y)       # 15

# this function does not return anything
def append_one(x):
    x.append(1)

x = [4, 3, 2]
y = append_one(x)
print(x)       # [4, 3, 2, 1]
print(y)       # None
```

That concludes the tutorial! If you want to do something in Python but don't know the syntax, Stack Overflow is a great resource. It is also a great resource if you get an error message when trying to run your Python script.

4.3 Working with Images in NumPy

This tutorial will walk through a short introduction to NumPy with emphasis on the tools that can be used for working with images. For more details, see NumPy's excellent documentation.

At their core, images are just two dimensional arrays or matrices of pixels. These pixels might have one value representing their gray value where 0 is black and some upper bound (usually 255) is white or they might have three values representing their red, green, and blue values respectively. Hence, in working with images, it is natural to want a tool that allows us to create and manipulate large arrays of numbers. NumPy is the premier package in Python for doing just that.

This tutorial is structured similarly to the Python tutorial in which each of the follow sections corresponds to a script which introduces some concept. As an important note, using NumPy in a script requires `import numpy as np` at the beginning.

4.3.1 NumPy Array

The NumPy array is very similar to the list. To create a NumPy array, we can pass a list to `np.array()`. We can access values in the array the same way we did in a list. However, the NumPy array does not have the `.append()` method. Additionally, the + operator has a different meaning when applied to two NumPy arrays: if the arrays are of the same size, it adds arrays together element-wise. Two other helpful methods for initializing arrays are `np.zeros()` and `np.ones()` which initializes arrays of all zeros or ones in the shape given.

```
# numpy_array.py
import numpy as np

x = [1, 2, 3]
y = np.array([1, 2, 3])

print(x)     # [1, 2, 3]
print(y)     # [1 2 3]
print(x[0])  # 1
print(y[0])  # 1

print(x + x) # [1, 2, 3, 1, 2, 3]
print(y + y) # [2 4 6]

w = np.zeros(3)
z = np.ones((2, 2))
print(w)  # [0. 0. 0.]
print(z)
# [[1. 1.]
#  [1. 1.]]
```

4.3.2 Array Attributes

The three most common attributes of an array are the dimension `ndim`, size `size`, and shape `shape`. The script below gives all three attributes of two example arrays.

```
# array_attributes.py
import numpy as np

A = np.array([[1, 2, 3],[4, 5, 6]])
print(A)
# [[1 2 3]
#  [4 5 6]]

print(A.ndim)   # 2
print(A.size)   # 6
print(A.shape)  # (2, 3)

B = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
print(B)
# [[[1 2]
#   [3 4]]
#
#  [[5 6]
```

(continues on next page)

(continued from previous page)

```
#   [7 8]]]

print(B.ndim)   # 3
print(B.size)   # 8
print(B.shape)  # (2, 2, 2)
```

4.3.3 Indexing and Slicing

A common operation you will want to do is access part of an array. The notation `x[i:j]` gives the `i` th through `j` th (not including `j`) values in the array `x`. We can use `x[i:]` or `x[:i]` to get all the values after or before the `i` th value respectively. It should be noted that this notation also works with the Python list.

```
# indexing.py
import numpy as np

A = np.array([0, 1, 2, 3, 4])

print(A[2:4])  # [2 3]
print(A[2:])   # [2 3 4]
print(A[:2])   # [0 1]
print(A[-2:])  # [3 4]

B = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(B)
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]

print(B[1:])
# [[4 5 6]
#  [7 8 9]]
print(B[:, 1:])
# [[2 3]
#  [5 6]
#  [8 9]]
print(B[0:2, 0:2])
# [[1 2]
#  [4 5]]

C = np.zeros((3,3))
C[0:2, 0:2] = np.ones((2,2))
print(C)
# [[1. 1. 0.]
#  [1. 1. 0.]
#  [0. 0. 0.]]
```

4.3.4 Conditional Array

Another way to slice an array is with a condition. The syntax for this is `x[condition]`. If we just look at the result of the condition, it returns an array of boolean values where the value is `True` if the corresponding element satisfied the condition and `False` otherwise. Passing this boolean array to the slicing notation indicates which values to keep.

```
# condition_array.py
import numpy as np

A = np.array([1, 2, 3, 4, 5])

print(A > 2)     # [False False True True True]
print(A[A > 2])  # [3 4 5]

B = np.array([[1, 2], [3, 4]])

print(B < 4)
# [[ True  True]
#  [ True False]]
print(B[B < 4])  # [1 2 3]
```

4.3.5 Array Math

The addition, subtraction, multiplication, and division operations for values correspond to the element-wise operations for arrays. Element-wise meaning that the operation is applied to corresponding elements in the two arrays. We can also apply more advanced mathematical functions to an array using the NumPy implmentation.

```
# array_math.py
import numpy as np

A = np.array([1, 2, 3])
B = np.array([4, 5, 6])

print(A + B)  # [5 7 9]
print(B - A)  # [3 3 3]
print(A * B)  # [ 4 10 18]
print(B / A)  # [4.  2.5 2. ]

print(np.power(A,2))  # [1 4 9]
print(np.sin(A))      # [0.84147098 0.90929743 0.14112001]
```

4.3.6 Miscellaneous Operations

There are some additional array operations that may be useful. `.max()` and `.min()` can be used to get the minimum or maximum element in an array respectively. An array can be transposed (axes swapped) with `.T`. Lastly, `.vstack()` and `.hstack()` can be used to vertically or horizontally stack a pair of arrays.

```
# misc_operations.py
import numpy as np

A = np.array([[1, 2], [3, 4]])
```

(continues on next page)

(continued from previous page)

```
B = np.array([[5, 6], [7, 8]])

print(A.min())  # 1
print(A.max())  # 4
print(A.T)
# [[1 3]
#  [2 4]]
print(np.hstack((A,B)))
# [[1 2 5 6]
#  [3 4 7 8]]
print(np.vstack((A,B)))
# [[1 2]
#  [3 4]
#  [5 6]
#  [7 8]]
```

That concludes the tutorial! There is an endless amount to learn about the NumPy Python package. Feel free to explore the documentation further to learn more neat capabilities.

4.4 Introduction to dmtools

This tutorial will walk through a short introduction to dmtools. The tutorial is divided into sections which each focus on a certain module of dmtools. Note that this documentation is under development.

4.4.1 io (input / output)

The first step in manuipulating images programtically with dmtools is loading in an image. This is done using *dmtools.io.read_png()*. Similarly, after manuipulating the image, you can export it to a PNG file with *dmtools.io.write_png()*. Here is a short example script with no manipulations. Note that this example script assumes that the script `io_ex.py` and `checks_10.png` are in the same directory.

```
# io_ex.py
import dmtools

image = dmtools.read_png('checks_10.png')
dmtools.write_png(image, 'checks_10_clone.png')
```

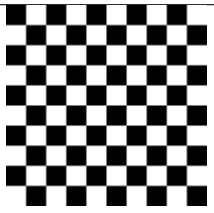


Fig. 1: checks_10.png

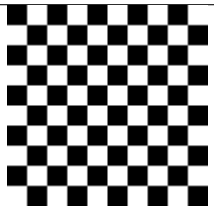


Fig. 2: checks_10_clone.png

4.4.2 transform

The transform module contains many functions for manipulating images. The full API reference can be found here: *dmtools.transform*. In this section, we will highlight some of the functionality.

Currently, the transform module is mainly focused on image manipulations related to rescaling an image. Frequently, the first step in image rescaling is blurring the image. This provides a good removal of "noise" from the image. The *dmtools.transform.blur()* functions does just that. It takes a parameter called `sigma` which indicates how much to blur the image. Usually, `sigma=0.5` is a good default. The example script below reads `red_blue_square.png` and then blurs the image with two different values of `sigma`. You can see the resulting images below where the larger `sigma` results in a blurrier image.

```
# simple_blur.py
import dmtools
from dmtools import transform

image = dmtools.read_png('red_blue_square.png')
blurred_image = transform.blur(image, sigma=5)
dmtools.write_png(blurred_image, 'red_blue_square_blur_5.png')

blurred_image = transform.blur(image, sigma=10)
dmtools.write_png(blurred_image, 'red_blue_square_blur_10.png')
```

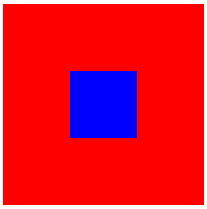


Fig. 3: red_blue_square.png

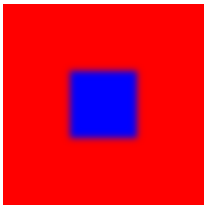


Fig. 4: red_blue_square_blur_5.png

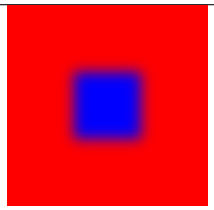


Fig. 5: red_blue_square_blur_10.png

After blurring, we can actually rescale the image. This step is also called "resampling." This is done with *dmtools.transform.rescale()*. This takes a parameter `k` which specifies by what factor to scale the image. Hence, `k=2` would double the width and height of the image.

When scaling down an image, we have more than one source pixel for each new pixel and we must decide how to assign a color to that new pixel. Similarly, when scaling up an image, we have many pixels in the new image for which there are no corresponding source pixels. Again, we must decide how to assign these pixels a color based on their proximity to the source pixels. A filter is a combination of a weighting function and support which determine how we choose.

In the dmtools rescale implementation, there are multiple built-in filters. A comprehensive list of them is given in the documentation: *dmtools.transform.rescale()*. Depending on the use case, one or more of the filters may be applicable. The exciting feature of this implementation is the ability to provide one's own weighting function and support to define custom filters. The weighting function (blue) and supports (red) of some common filters are given below. The weighting function tells us how much to weight the color of a source pixel as a function of its distance to the new pixel. The support defines the "neighborhood" of pixels. In most cases, that is the furthest a source pixel can be while still contributing some weight.

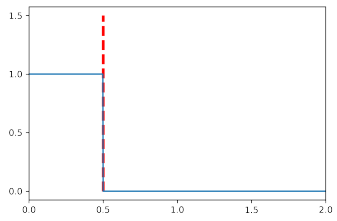


Fig. 6: Box Filter

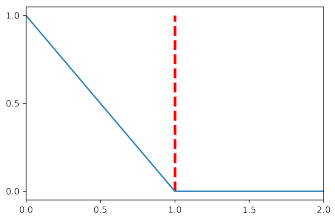


Fig. 7: Triangle Filter

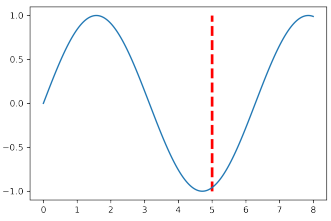


Fig. 8: Custom Filter

In the example script below, we load the 10x10 checkerboard image and scale it up using three different filters: "point" or "nearest neighbor", "triangle", and a custom filter. You can ignore `transform.clip` and `transform.normalize` for now. The resulting images are also shown.

```
# rescale_ex.py
import dmtools
from dmtools import transform
import numpy as np

image = dmtools.read_png('checks_10.png')
scaled_image = transform.rescale(image, k=10, filter='point')
scaled_image = transform.clip(scaled_image).astype(np.uint8)
dmtools.write_png(scaled_image, 'checks_10_point.png')

scaled_image = transform.rescale(image, k=10, filter='triangle')
scaled_image = transform.clip(scaled_image).astype(np.uint8)
dmtools.write_png(scaled_image, 'checks_10_triangle.png')

def f(x):
    return np.sin(x)

# use a custom weighting function and support
scaled_image = transform.rescale(image, k=10, weighting_function=f, support=5)
scaled_image = transform.normalize(scaled_image).astype(np.uint8)
dmtools.write_png(scaled_image, 'checks_10_custom.png')
```

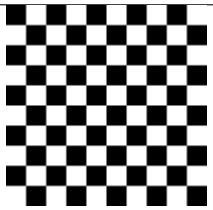


Fig. 9: checks_10_point.png

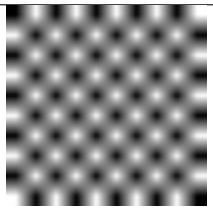


Fig. 10: checks_10_triangle.png

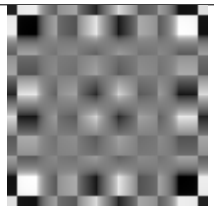


Fig. 11: checks_10_custom.png

You can see how "point" is the best filter for maintaining the pixels of the original image. Here, the "triangle" filter causes the image to be blurred since it is takes the average of surrounding white and black pixels causing the gray space between them. Any reasonable filter will mostly decrease the weight as the distance gets further. Furthermore, they will not have significant negative weights. For that reason, the custom filter used here does all sorts of strange things to the image.

After rescaling the image, we would like to write it to a PNG file. However, the rescaling step results in pixels having

non-integer values which can also be outside of the [0, 255] range (especially when using strange filters). The transform module provides three different functions to adjust values back into the [0, 255] range: *dmtools.transform.clip()*, *dmtools.transform.normalize()*, *dmtools.transform.wraparound()*. It is recommended to use `.astype(np.uint8)` to round. The example script below shows how the choice of which of these you use affects the resulting image.

```
# clamping_ex.py
import dmtools
from dmtools import transform
import numpy as np

def f(x):
    return np.sin(x)

image = dmtools.read_png('checks_10.png')
scaled_image = transform.rescale(image, k=10, weighting_function=f, support=7)

clip_image = transform.clip(scaled_image).astype(np.uint8)
dmtools.write_png(clip_image, 'checks_10_clip.png')

normalize_image = transform.normalize(scaled_image).astype(np.uint8)
dmtools.write_png(normalize_image, 'checks_10_normalize.png')

wraparound_image = transform.wraparound(scaled_image).astype(np.uint8)
dmtools.write_png(wraparound_image, 'checks_10_wraparound.png')
```

Fig. 12: checks_10_clip.png

Fig. 13: checks_10_normalize.png

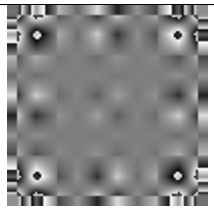


Fig. 14: checks_10_wraparound.png

In `checks_10_wraparound.png`, we can see harsh contrast between gradients where a gradient progressively gets darker until it switches white. This is arising from dark values above 255 (black) wrapping around to 0 (white) and vice versa. In `checks_10_clip.png`, these are the darkest and whitest areas in the image since values above or below just get clipped to 0 and 255 respectively. Lastly, `checks_10_normalize.png` normalizes the minimum and largest value to 0 and 255 causing this image to loose contrast in the center when compared to the clipping algorithm.

4.4.3 adjustments

The adjustments module currently contains an equivalent to a curves tool. The full API reference can be found here: *dmtools.adjustments*. In this section, we will give a more detailed explanation of how the curve tool can be used.

A curves tool is a comprehensive tool for changing the colors of an image. It can be used to achieve a variety of effects. It works by specifying a function for remapping the tones of an image. This function can be applied to the image as a whole or to an individual channel (examples of both are given below). As dmtools works with images normalized to [0,1], the curve function should be a function with a domain and range of [0,1].

Let us walk through the example script below. Aside from the identity function (the straight line from (0,0) to (1,1) in which every tone maps to itself), all of the functions it uses are given below.

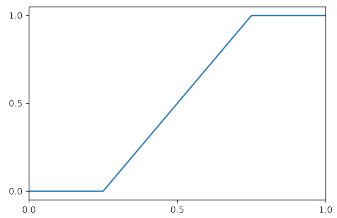


Fig. 15: Clip to [0.25, 0.75]

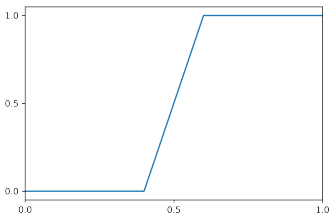


Fig. 16: Clip to [0.40, 0.60]

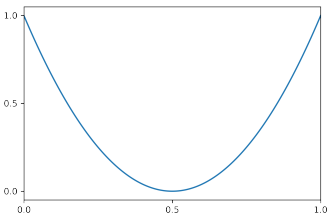


Fig. 17: Parabola

In this script, we apply a variety of different curves to the *pallette.png* image. Some curves are applied to all channels of an image (when no channel is given) and some are applied to individual channels. As we are working in the RGB (Red, Green, Blue) colorspace, the red channel is channel 0 and the blue channel is channel 2.

```
# curve.py
import dmtools
from dmtools import adjustments
import numpy as np

image = dmtools.read('pallette.png')

# apply identity to all channels
tmp = adjustments.apply_curve(image, lambda x: x)
dmtools.write_png(tmp, 'pallette_identity.png')

# apply clip from 0.25 to 0.75 to all channels
tmp = adjustments.apply_curve(image, lambda x: np.clip(2*(x-0.25), 0, 1))
dmtools.write_png(tmp, 'pallette_clip_25_75.png')

# apply clip from 0.4 to 0.6 to all channels
tmp = adjustments.apply_curve(image, lambda x: np.clip(5*(x-0.4), 0, 1))
dmtools.write_png(tmp, 'pallette_clip_40_60.png')

# apply clip from 0.4 to 0.6 to red channels
tmp = adjustments.apply_curve(image, lambda x: np.clip(5*(x-0.4), 0, 1), 0)
dmtools.write_png(tmp, 'pallette_clip_40_60_red.png')

# apply clip from 0.4 to 0.6 to blue channels
tmp = adjustments.apply_curve(image, lambda x: np.clip(5*(x-0.4), 0, 1), 2)
dmtools.write_png(tmp, 'pallette_clip_40_60_blue.png')

# apply parabola to all channels
tmp = adjustments.apply_curve(image, lambda x: 4*np.power(x - 0.5, 2))
dmtools.write_png(tmp, 'pallette_parabola.png')
```

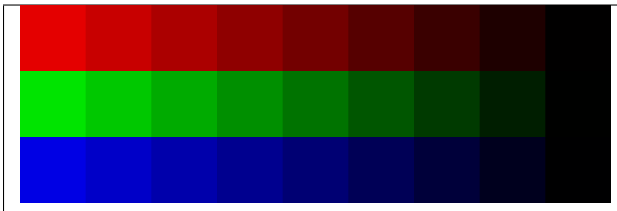


Fig. 18: pallette.png

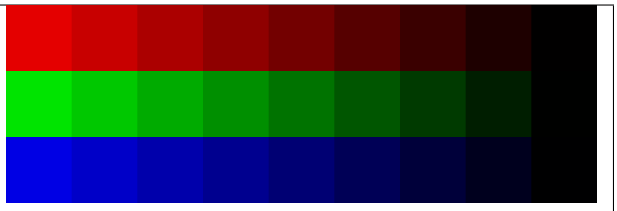


Fig. 19: pallette_identity.png

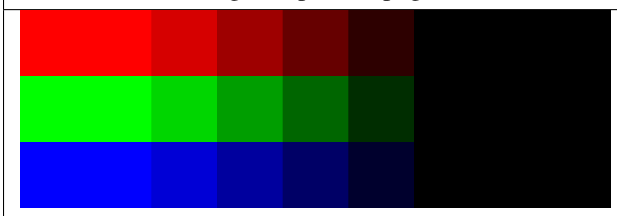


Fig. 20: pallette_clip_25_75.png

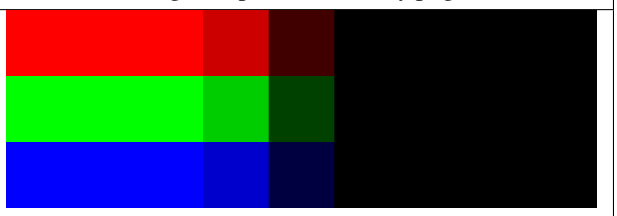


Fig. 21: pallette_clip_40_60.png

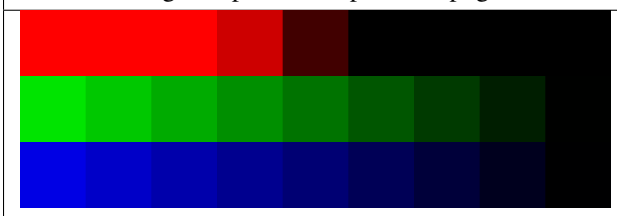


Fig. 22: pallette_clip_40_60_red.png

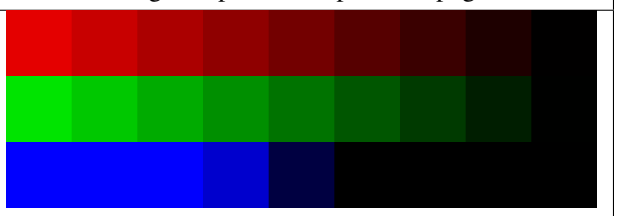


Fig. 23: pallette_clip_40_60_blue.png

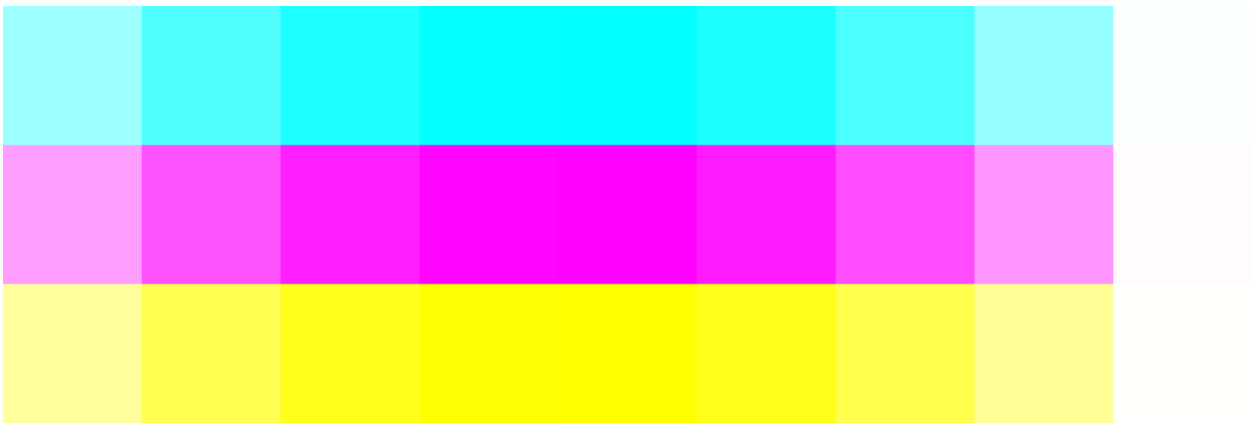


Fig. 24: pallette_parabola.png

All of the images generated by the script are shown above. You should make a few important observations. The identity function does not alter the image. The clip functions reduce contrast at either end of the tonal range but increase it in the center of the range. The clip to [0.40, 0.60] has a more pronounced effect that the clip to [0.25, 0.75]. Lastly, when the curve is applied to a single filter, the colors of other channels are unaffected.

CHAPTER

FIVE

DOCUMENTATION

5.1 dmtools package

5.1.1 dmtools.io module

dmtools.io.**read**(*path: str*) → numpy.ndarray
Read an image file into a NumPy array.

Parameters **path** (*str*) – String file path with extention in {png, pbm, pgm, ppm}.

Returns NumPy array representing the image.

Return type np.ndarray

dmtools.io.**read_netpbm**(*path: str*) → numpy.ndarray
Read a Netpbm file (pbm, pgm, ppm) into a NumPy array.

Netpbm is a package of graphics programs and a programming library. These programs work with a set of graphics formats called the "netpbm" formats. Each format is identified by a "magic number" which is denoted as `P` followed by the number identifier. This class works with the following formats.

- pbm: Pixels are black or white (`P1` and `P4`).
- pgm: Pixels are shades of gray (`P2` and `P5`).
- ppm: Pixels are in full color (`P3` and `P6`).

Each of the formats has two "magic numbers" associated with it. The lower number corresponds to the ASCII (plain) format while the higher number corresponds to the binary (raw) format. This class can handle reading both the plain and raw formats though it can only export Netpbm images in the plain formats (`P1`, `P2`, and `P3`).

The plain formats for all three of pbm, pgm, and ppm are quite similar. Here is an example pgm format.

```
P2
5 3
4
1 1 0 1 0
2 0 3 0 1
2 2 3 1 0
```

The first row of the file contains the "magic number". In this example, the file is a grayscale pgm image. The second row gives the file dimensions (width by height) separated by whitespace. The third row gives the maximum gray/color value. In this case, it is the maximum gray value since this is a grayscale pgm image. Essentially, this number encodes how many different gradients there are in the image. Lastly, the remaining lines of the file encode the actual pixels of the image. In a pbm image, the third line is not needed since pixels have binary (black

or white) values. In a ppm full-color image, each pixels has three values represeting it–the values of the red, green, and blue channels.

This descriptions serves as a brief overview of the Netpbm formats with the relevant knowledge for using this class. For more information about Netpbm, see the Netpbm Home Page.

Parameters **path** (*str*) – String file path.

Returns NumPy array representing image.

Return type image (np.ndarray)

dmtools.io.**read_png**(*path: str*) → numpy.ndarray
Read a png file into a NumPy array.

Parameters **path** (*str*) – String file path.

Returns NumPy array representing the image.

Return type np.ndarray

dmtools.io.**write_netpbm**(*image: numpy.ndarray*, *k: int*, *path: str*, *comment: List[str] = []*)
Write object to a Netpbm file (pbm, pgm, ppm).

Uses the ASCII (plain) magic numbers.

Parameters

- **image** (*np.ndarray*) – NumPy array representing image.
- **k** (*int*) – Maximum color/gray value.
- **path** (*str*) – String file path.
- **comment** (*str*) – List of comment lines to include in the file.

dmtools.io.**write_png**(*image: numpy.ndarray*, *path: str*)
Write NumPy array to a png file.

The NumPy array should have values in the range [0, 1]. Otherwise, this function has undefined behavior.

Parameters

- **image** (*np.ndarray*) – NumPy array representing image.
- **path** (*str*) – String file path.

5.1.2 dmtools.transform module

dmtools.transform.**blur**(*image: numpy.ndarray*, *sigma: float*, *radius: float = 0*) → numpy.ndarray
Blur the image.

This image blur implentation is largley based off of the ImageMagick impmenetation. It uses a Gaussian Filter with parameter `sigma` and a support of `radius` to blur the image.

Parameters

- **image** (*np.ndarray*) – Image to be blurred.
- **sigma** (*float*) – "Neighborhood" of the blur. A larger value is blurrier.
- **radius** (*float*) – Limit of the blur. Defaults to 4 x sigma.

Returns Blurred image.

Return type np.ndarray

dmtools.transform.**clip**(*image: numpy.ndarray*) → numpy.ndarray
Clip gray/color values that are out of bounds.

Every value less than 0 is mapped to 0 and every value more than 1 is mapped to 1. Values in [0,1] are untouched.

Parameters **image** (*np.ndarray*) – Image to clip.

Returns Clipped image.

Return type np.ndarray

dmtools.transform.**normalize**(*image: numpy.ndarray*) → numpy.ndarray
Normalize the image to bring all gray/color values into bounds.

Normalize the range of values in the image to [0,1]. If applied to a three channel image, normalizes each channel by the same amount.

Parameters **image** (*np.ndarray*) – Image to normalize.

Returns Normalized image.

Return type np.ndarray

dmtools.transform.**rescale**(*image: numpy.ndarray*, *k: int*, *filter: str = 'point'*, *weighting_function: Optional[Callable] = None*, *support: Optional[Callable] = None*, ***kwargs*) → numpy.ndarray
Rescale the image by the given scaling factor.

This image rescale implentation is largley based off of the ImageMagick impmenetation. The following filters are built-in:

- Point Filter ("point"): Nearest-neighbor heuristic.
- Box Filter ("box"): Average of neighboring pixels.
- Triangle Filter ("triangle"): Linear decrease in pixel weight.
- Catmull-Rom Filter ("catrom"): Produces a sharper edge.
- Gaussian Filter ("gaussian"): Blurs image. Useful as low pass filter.

Additionally, advanced users can specify a custom filter by providing a weighting function and a support.

Parameters

- **image** (*np.ndarray*) – Image to rescale.
- **k** (*int*) – Scaling factor.
- **filter** (*str*) – {point, box, triangle, catrom, gaussian}.
- **weighting_function** (*Callable*) – Weighting function to use.
- **support** (*float*) – Support of the provided weighting function.

Returns Rescaled image.

Return type np.ndarray

dmtools.transform.**wraparound**(*image: numpy.ndarray*) → numpy.ndarray
Wraparound gray/color values that are out of bounds.

Each value x is mapped to x mod 1 such that values outside of [0,1] wraparound until they fall in the desired range.

Parameters **image** (*np.ndarray*) – Image to wraparound

Returns Wraparound image.

Return type np.ndarray

5.1.3 dmtools.adjustments module

dmtools.adjustments.**apply_curve**(*image: numpy.ndarray*, *f: Callable*, *c: int = - 1*) → numpy.ndarray
Apply a curve f to an image or channel of an image.

Parameters

- **image** (*np.ndarray*) – Image on which to apply curve.
- **f** (*Callable*) – Curve to apply. f: [0,1] -> [0,1].
- **c** (*int*) – Channel to apply curve to. Apply to all channels if -1.

Returns Image with curve applied.

Return type np.ndarray

5.1.4 dmtools.colorspace module

dmtools.colorspace.**Lab_to_RGB**(*image: numpy.ndarray*, *illuminant: str = 'D65'*) → numpy.ndarray
Convert an image in Lab space to CIE RGB space.

For details about the implemented conversion, see CIE 1931 color space and CIELAB color space.

Parameters

- **image** (*np.ndarray*) – Image in Lab space.
- **illuminant** (*str*) – Standard illuminant {D65, D50}

Returns Image in CIE RGB space.

Return type np.ndarray

dmtools.colorspace.**Lab_to_XYZ**(*image: numpy.ndarray*, *illuminant: str = 'D65'*) → numpy.ndarray
Convert an image in Lab space to CIE XYZ space.

For details about the implemented conversion, see CIELAB color space.

Parameters

- **image** (*np.ndarray*) – Image in Lab space.
- **illuminant** (*str*) – Standard illuminant {D65, D50}

Returns Image in CIE XYZ space.

Return type np.ndarray

dmtools.colorspace.**RGB_to_Lab**(*image: numpy.ndarray*, *illuminant: str = 'D65'*) → numpy.ndarray
Convert an image in CIE RGB space to Lab space.

For details about the implemented conversion, see CIE 1931 color space and CIELAB color space.

Parameters

- **image** (*np.ndarray*) – Image in CIE RGB space.
- **illuminant** (*str*) – Standard illuminant {D65, D50}

Returns Image in Lab space.

Return type np.ndarray

dmtools.colorspace.**RGB_to_XYZ**(*image: numpy.ndarray*) → numpy.ndarray
Convert an image in CIE RGB space to XYZ space.

For details about the implemented conversion, see CIE 1931 color space.

Parameters **image** (*np.ndarray*) – Image in CIE RGB space.

Returns Image in CIE XYZ space.

Return type np.ndarray

dmtools.colorspace.**RGB_to_YUV**(*image: numpy.ndarray*) → numpy.ndarray
Convert an image in CIE RGB space to YUV space.

For details about the implemented conversion, see YUV.

Parameters **image** (*np.ndarray*) – Image in CIE RGB space.

Returns Image in YUV space.

Return type np.ndarray

dmtools.colorspace.**RGB_to_gray**(*image: numpy.ndarray*) → numpy.ndarray
Convert an image in CIE RGB space to grayscale.

For details about the implemented conversion, see FAQs about Color.

Parameters **image** (*np.ndarray*) – Image in CIE RGB space.

Returns Image in grayscale.

Return type np.ndarray

dmtools.colorspace.**XYZ_to_Lab**(*image: numpy.ndarray*, *illuminant: str = 'D65'*) → numpy.ndarray
Convert an image in CIE XYZ space to Lab space.

For details about the implemented conversion, see CIELAB color space.

Parameters

- **image** (*np.ndarray*) – Image in CIE XYZ space.
- **illuminant** (*str*) – Standard illuminant {D65, D50}

Returns Image in Lab space.

Return type np.ndarray

dmtools.colorspace.**XYZ_to_RGB**(*image: numpy.ndarray*) → numpy.ndarray
Convert an image in CIE XYZ space to RGB space.

For details about the implemented conversion, see CIE 1931 color space.

Parameters **image** (*np.ndarray*) – Image in CIE XYZ space.

Returns Image in CIE RGB space.

Return type np.ndarray

dmtools.colorspace.**YUV_to_RGB**(*image: numpy.ndarray*) → numpy.ndarray
Convert an image in YUV space to CIE RGB space.

For details about the implemented conversion, see YUV.

Parameters **image** (*np.ndarray*) – Image in YUV space.

Returns Image in CIE RGB space.

Return type np.ndarray

dmtools.colorspace.**denormalize**(*image: numpy.ndarray*, *color_space: str*) → numpy.ndarray
Denormalize the image in the given color space.

Parameters

- **image** (*np.ndarray*) – Normalized image in the given color space.
- **color_space** (*str*) – Color space {RGB, Lab, YUV}.

Returns Denormalized image in the given color space.

Return type np.ndarray

dmtools.colorspace.**gray_to_RGB**(*image: numpy.ndarray*) → numpy.ndarray
Convert an image in grayscale to CIE RGB space.

Parameters **image** (*np.ndarray*) – Image in grayscale.

Returns Image in CIE RGB space.

Return type np.ndarray

dmtools.colorspace.**normalize**(*image: numpy.ndarray*, *color_space: str*) → numpy.ndarray
Normalize the image in the given color space.

Parameters

- **image** (*np.ndarray*) – Image in the given color space.
- **color_space** (*str*) – Color space {RGB, Lab, YUV}.

Returns Normalized image with values in [0,1].

Return type np.ndarray

5.1.5 dmtools.animation module

dmtools.animation.**clip**(*path: str*, *start: int = 0*, *end: int = - 1*) → List[numpy.ndarray]
Return a list of images in the given directory.

Images are ordered according to their name. Hence, the following naming convention is recommend.

name0000.png, name0001.png, ...

Parameters

- **path** (*str*) – String directory path.
- **start** (*int, optional*) – Starting frame. Defaults to 0.
- **end** (*int, optional*) – Ending frame. Defaults to -1.

Returns List of NumPy arrays representing images.

Return type List[np.ndarray]

dmtools.animation.**to_mp4**(*frames: List[numpy.ndarray]*, *path: str*, *fps: int*, *s: int = 1*, *audio: Optional[*dmtools.sound.WAV*] = None*)
Write an animation as a .mp4 file using ffmpeg through imageio.mp4

Parameters

- **frames** (*List[np.ndarray]*) – List of frames in the animation.
- **audio** (*sound.WAV*) – Audio for the animation (None if no audio).
- **path** (*str*) – String file path.

- **fps** (*int*) – Frames per second.
- **s** (*int, optional*) – Multiplier for scaling. Defaults to 1.

5.1.6 dmtools.ascii module

class dmtools.ascii.**Ascii**(*M: numpy.ndarray*)
Bases: object

An object representing an ASCII image.

For more information about ASCII, see ASCII

to_png(*path: str*)
Write object to a png file.

Parameters **path** (*str*) – String file path.

to_txt(*path: str*)
Write object to a txt file.

Parameters **path** (*str*) – String file path.

dmtools.ascii.**image_to_ascii**(*image: numpy.ndarray*) → *dmtools.ascii.Ascii*
Return an ASCII representation of the given image.

This function uses a particular style of ASCII art in which "symbols with various intensities [are used for] creating gradients or contrasts."

Parameters **image** (*np.ndarray*) – An image.

Returns ASCII representation of image.

Return type *Ascii*

5.1.7 dmtools.sound module

class dmtools.sound.**WAV**(*r: numpy.ndarray*, *l: numpy.ndarray*, *sample_rate: int = 44100*)
Bases: object

An object representing a WAV audio file.

For more information about the audio file format, see WAV

to_wav(*path*)
Write object to a WAV audio file (wav)

Parameters **path** (*str*) – String file path.

dmtools.sound.**wave**(*f: float*, *a: float*, *t: float*) → numpy.ndarray
Generate the samples of a sound wave.

Parameters

- **f** (*float*) – Frequency of the sound wave.
- **a** (*float*) – Amplitude of the sound wave.
- **t** (*float*) – Duration (seconds) of the sound wave.

Returns NumPy array with sample points of wave.

Return type np.ndarray

dmtools.sound.**wave_sequence**(*frequencies: numpy.ndarray*, *t*) → *dmtools.sound.WAV*

Return a Wav sound which iterates through the given frequencies.

Parameters

- **frequencies** (*np.ndarray*) – frequencies to iterate through.
- **t** (*[type]*) – duration of iteration.

Returns Wav file.

Return type *WAV*

5.1.8 dmtools.arrange module

dmtools.arrange.**border**(*image: numpy.ndarray*, *b: int*, *color: int = 'white'*) → numpy.ndarray

Add a border of width b to the image.

Parameters

- **image** (*Netpbm*) – Netpbm image to add a border to
- **b** (*int*) – width of the border/margin.
- **color** (*int*) – color of border {'white', 'black'} (defaults to white).

Returns Image with border added.

Return type np.ndarray

dmtools.arrange.**image_grid**(*images: List[numpy.ndarray]*, *w: int*, *h: int*, *b: int*, *color: int = 'white'*) → numpy.ndarray

Create a w * h grid of images with a border of width b.

Parameters

- **images** (*List[np.ndarray]*) – images (of same dimension) for grid.
- **w** (*int*) – number of images in each row of the grid.
- **h** (*int*) – number of images in each column of the grid.
- **b** (*int*) – width of the border/margin.
- **color** (*int*) – color of border {'white', 'black'} (defaults to white).

Returns grid layout of the images.

Return type np.ndarray

PYTHON MODULE INDEX

d

INDEX

Y