

Lab 1 - Python Basics

ARIZONA STATE UNIVERSITY
SCHOOL OF ELECTRICAL, COMPUTER,
AND ENERGY ENGINEERING,
EEE598: Deep Learning Media Processing & Understanding

1 Objectives

- Setup Ubuntu Linux using VirtualBox Virtual Environment
- Setup Python and basic Python packages
- Learn Python and Numpy fundamentals
- Implement the `ImageClass` class to practice using Python and Numpy

2 Setup Ubuntu Linux environment using VirtualBox

For this course, we will be using Ubuntu Linux. Although it is possible to use Windows or macOS to train deep models, library support can be poor, especially for deep learning. Additionally, most deep learning development and research is done on Linux machines, even if the product is eventually to be deployed on other operating systems. Ubuntu Linux is a popular distribution of Linux which can run the latest versions of popular machine learning libraries.

We will be using a Virtual Machine (VM) to install and work in a Linux environment. The VM will run as an application in the “host” OS on your computer, so we do not need to dual-boot or partition the hard drive to run Ubuntu. Additionally by using a VM, we ensure that everyone has an identical development environment to work with Python. Another advantage of the VM is that it is easy to restart with a “fresh” Ubuntu installation if needed.

Extra Information: Other ways to run Python

It is possible to run Python and most Python packages in Windows or macOS. However, we will not support any problems that may occur when installing Python on these operating systems. We recommend installing Ubuntu Linux because later in the course we will be using Ubuntu Linux instances on the Amazon Web Services (AWS) Cloud to run deep learning models using GPUs. It is necessary that you become familiar with the Linux operating system.

It is also possible to dual boot Ubuntu Linux with another operating system on your personal PC. We will not support any problems with dual-boot installation in this course. If you do try to dual boot, it is recommended that all of the data is backed up before installation.

We will be using Oracle VirtualBox to run the virtual machine. This is free software that is available for Windows, macOS, and Linux. First we need to download the VirtualBox package and the Ubuntu installation disk ISO file. For Ubuntu we will be using version 16.04 LTS, which is the latest stable release. For VirtualBox we will be using the latest version. Please download the two files at the following links:

1. <https://www.virtualbox.org/wiki/Downloads>
2. <https://www.ubuntu.com/download/desktop>

Install VirtualBox and remember the location of the Ubuntu installation ISO file that you just downloaded.


Important: Disk Space

Your computer will need approximately 20GB of space. We are installing Ubuntu, several large Python packages, and in the future we will need space for datasets. Do not try to make the VirtualBox hard drive size smaller than 20GB, as in the future you may run out of space.

VirtualBox Setup Open VirtualBox and create a new virtual machine by pressing the **New** button. Name your virtual machine “Ubuntu”, and VirtualBox will automatically infer that you want to install 64-bit Ubuntu Linux (Figure 1).

VirtualBox will ask how much memory to allocate to the VM. If possible, we recommend to give the Virtual Machine at least 4096MB. However, if your total computer memory is small you should leave enough memory for the host OS to run. Ubuntu will run with less than 4096MB, but performance may be sluggish. The memory allocation can be changed later in the VirtualBox settings.

Next VirtualBox will ask about the virtual disk. Select **Create a virtual hard disk now**. Create a VDI (**V**irtual **B**ox **D**isk **I**mage) that is **Dynamically allocated**. Dynamic allocation means that the virtual disk will automatically grow and shrink as the data in the virtual machine changes. However, the disk cannot easily extend beyond the maximum allocation. So set the maximum allocation to 20GB. This space will be enough for the operating system, the Python libraries, and some data for experiments.

Now that the VM is setup, we need to configure it to launch from the Ubuntu ISO file that we just downloaded so we can install Ubuntu. To do this click on **Settings** and go to **Storage**. Under **Controller:** IDE select **Empty**. On the right side, click on the disk icon  and select **Choose Virtual Optical Disk File...** select the Ubuntu ISO file that we downloaded (Figure 2). Click OK to finish this setup.

Installing Ubuntu Now we can start the VM and install Ubuntu. To do this, click the green **Start** button. Once the installation disk boots click **Install Ubuntu**. Follow the instructions for a basic install. There is no need to install third-party software or download updates while installing.

During installation choose **Erase Disk and install Ubuntu**. This will not erase the local disk of your computer. This will only erase the virtual disk image used by the virtual machine (which is currently empty). Later you will be asked to setup a username and password. Remember these because we will need it later to install software.

When the installation is complete you will be asked to restart the machine. After restarting, Ubuntu will ask you to remove the installation media. VirtualBox should automatically unmount the Ubuntu install ISO file after installation, so just press enter. Ubuntu should boot to the login screen.

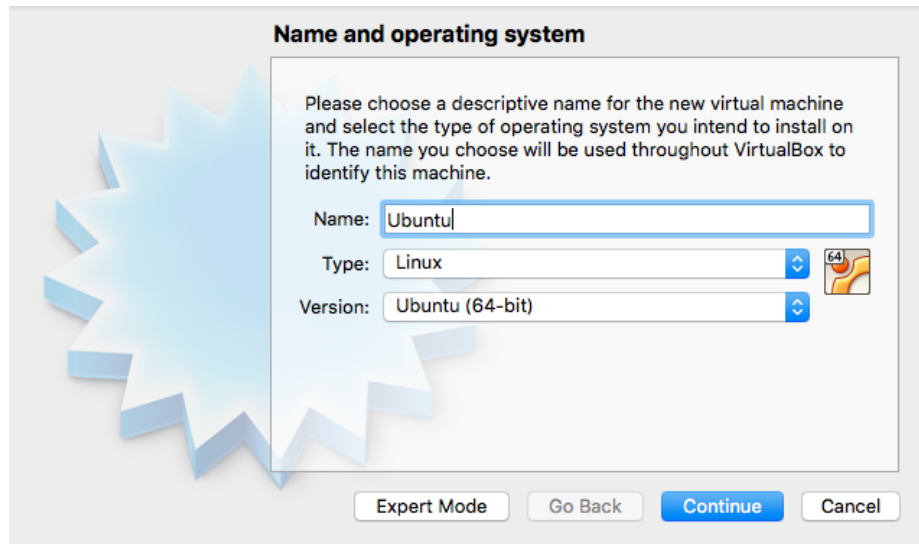


Figure 1: Starting a new virtual machine.

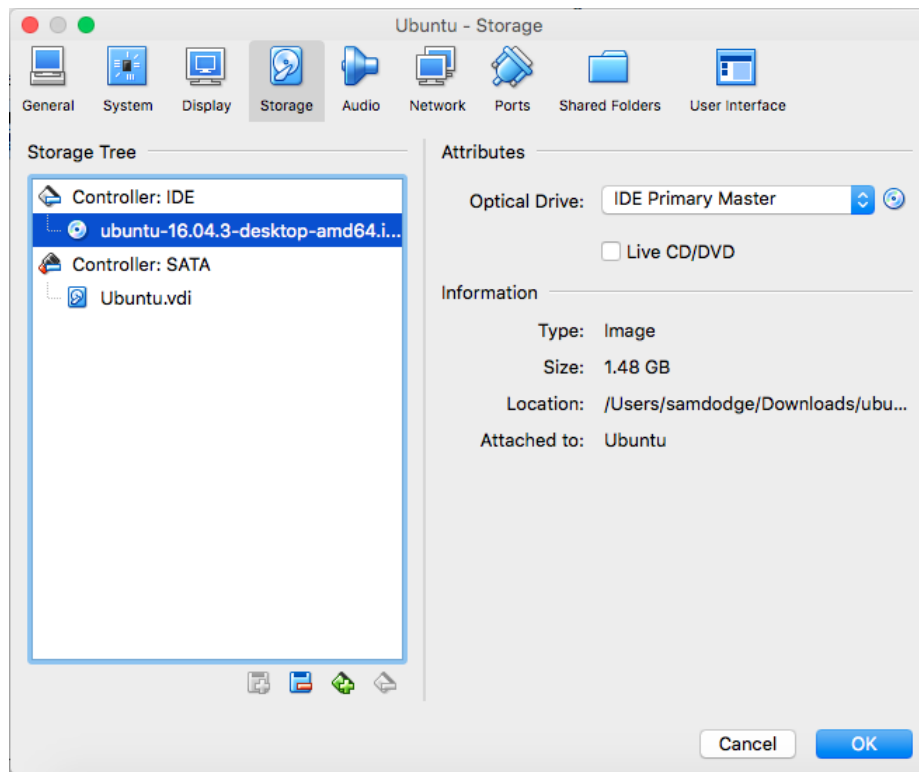


Figure 2: Selecting startup disk.

Table 1: Useful Linux commands


Command	Description	Example
ls	List files in current directory	
cd	Change directory	cd new_directory
cp	Copy files	cp old_file.py new_file.py
rm	Remove files (warning no undo!)	rm file.py
mv	Move files	mv old_name.py new_name.py
mkdir	make new directory	mkdir new_dir
ssh	Open SSH connection to server	ssh username@server
sudo	Execute command as super user	sudo command
pip	Install python packages	pip install package
python	Run python script	python script.py
apt-get	Package management for Ubuntu packages	apt-get install package
man	List command description and arguments	man command

Extra Information: Using a shared folder

Right now the file systems of our host and our VM are completely separate. We can keep it this way and work on all of our projects locally with the VM. Ubuntu comes with Firefox pre-installed, so we can download and upload files from this VM. Additionally, Dropbox is available for Linux, which offers an easy way to share files between the host OS and Ubuntu.

Another way is to setup a “shared” folder between the host OS and the VM. The shared folder can be seen and edited by both operating systems at the same time. To enable this functionality, we need to install **VirtualBox Guest Additions**.

To install the guest additions, from the VirtualBox menu bar select **Devices->Insert Guest Additions CD...** In the Ubuntu VM, you will be prompted to run a program from the inserted CD image. Click run and wait for the software to finish installing. After installation reboot the VM.

After the additions are installed we can setup the shared folder. In VirtualBox, go to **Machine->Settings->Shared Folders** and click the  icon to add a new shared folder. Select the folder path you want to share, and select a name that you will remember. Be sure that **Read Only** is not selected, and select **Make Permanent**. Now we can return to the VM and mount the shared folder somewhere in the VM file system. Open a Terminal window (**ctrl+alt+t**) and type the following to mount the folder. Replace **FOLDER_NAME** with the name you defined in VirtualBox. Your shared folder will now appear in **~/share** in the Ubuntu file system.

```
mkdir ~/share
sudo mount -t vboxsf -o uid=1000,gid=1000 FOLDER_NAME ~/share/
```

3 Installing Python Packages

In Ubuntu, open a new terminal shell by typing **ctrl+alt+t**. The terminal shell allows us to access the underlying Linux programs. We can launch programs, navigate the file system, edit files, etc. Table 1 shows a list of useful commands while working in the terminal.

To manage Python packages we use the **pip** (Python Packaging Index) program. The Python

Packaging Index contains many useful Python packages and allows us to install, remove, and upgrade those Python packages. To install `pip` we use the Ubuntu package manager program `apt-get`. Install `pip` using the following command.

```
sudo apt-get install python-pip
```

`sudo` runs the `apt-get` command in super user mode. This is analogous to `Run as administrator...` in Windows operating systems. `sudo` is often needed when installing programs because the programs will be installed system-wide.

With `pip` installed, we can use the command `pip install` to install needed Python packages. Run the following commands to install all of the packages we need for this lab. This may take a moment to complete. We also install `python-tk` using `apt-get`, which is necessary for plotting results to the screen.

```
sudo pip install numpy scipy matplotlib imageio
sudo apt-get install python-tk
```

The Numpy package includes all of the basic matrix operations we will need. Think of Numpy as MATLAB-like operations for Python. Scipy extends on Numpy with added functionality. Matplotlib is a MATLAB style plotting library. Finally, Imageio is used to load and save images. In later labs we will again use `pip` to install deep learning libraries.

4 Python and NumPy fundamentals

In this course we will be using the Python programming language. Python has become a popular language because it is open source, easy to learn, and has support for many popular libraries. There are two popular branches of Python: Python 2.X and Python 3.X. We will be using Python 2.X because of better compatibility with some older packages.

The main strength of Python for machine learning and data science is the abundance of dependable libraries. There are packages for scientific computing (`Numpy` and `Scipy`), machine learning (`Scikit-learn`), image processing (`Scikit-image`), computer vision (`OpenCV`), data visualization (`Matplotlib`) and deep learning (`Pytorch`, `Tensorflow`, `MxNet`, etc.). Most of these libraries can be installed with a single `pip` command, which makes setup very easy.

4.1 Task 1: Read Python Tutorial

Please read the following Python tutorial:

<http://cs231n.github.io/python-numpy-tutorial/>

All of this information will be useful, so you can leave this tutorial open for reference for the rest of the lab. It is also useful to consult the Numpy documentation [1], Scipy documentation [2], and PyPlot documentation [3].

Extra Information: Coming from MATLAB

If you are familiar with MATLAB, the following link is a cheat sheet that shows equivalent functions in Python/NumPy:

<http://mathesaurus.sourceforge.net/matlab-numpy.html>

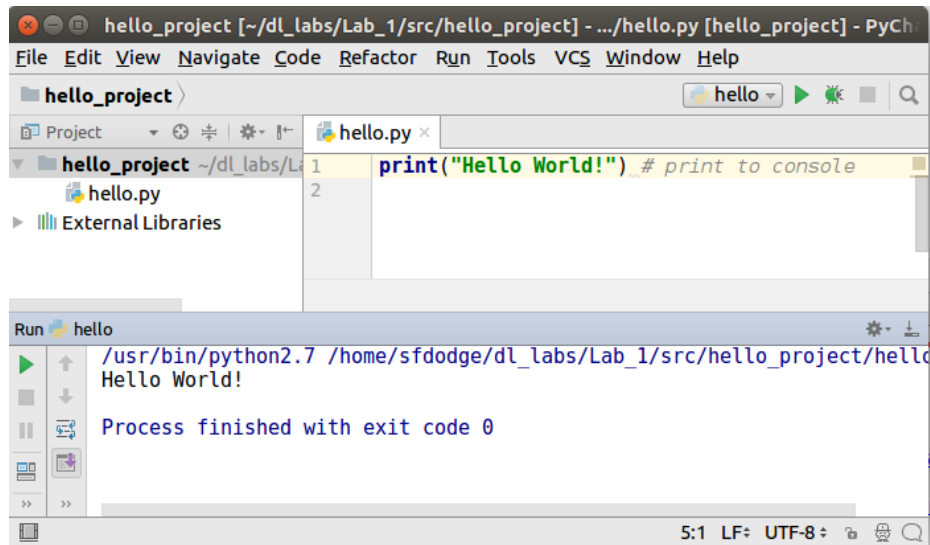


Figure 3: PyCharm layout

4.2 Task 2: Running code and Debugging in PyCharm

In this class we use the PyCharm IDE for editing and debugging code. PyCharm provides useful features such as auto-completion, versioning integration, visual debugging, etc.

Installing PyCharm We will use the `snap` command to install PyCharm. The `snap` command works similarly to `apt-get`:

```
sudo snap install pycharm-community --classic
```

We are installing the `pycharm-community` package instead of the `pycharm-professional` package. The `pycharm-professional` package is non-free software that has a limited evaluation period. However the free `pycharm-community` package has everything needed for this course.

After installation, PyCharm can be started by typing `pycharm-community` in a terminal window, or by using Ubuntu’s search button and searching for “PyCharm”.

Using PyCharm First lets open PyCharm. When you start PyCharm select **Do not import settings**. Next click the button **Skip Remaining and Set Defaults**. We have provided a file with settings for the built in Python interpreter. On the splash screen select **Configure->Import Settings**. Navigate to the included `pycharm.settings.jar` file. Hit ok to import settings, and ok to restart PyCharm.

Now in the PyCharm launch screen, select **Open** and navigate to the `hello_project` folder. We see that the default PyCharm layout has our files in the left pane, the code in the center, and at the bottom there is a pane for debugging our code (Figure 3). In the left pane our project contains a single file called `hello.py`.

Running scripts in PyCharm In PyCharm open the script `hello.py` by double clicking on its name in the left panel. Right click on the `hello.py` file in the left pane, and select **run** (Figure 4). After running, in the bottom panel the program will output: “Hello World!”.

We can also run python scripts in the terminal. Open a new terminal (`ctrl + alt + t`) and navigate to the folder that contains `hello.py`. We can navigate the file system using the change

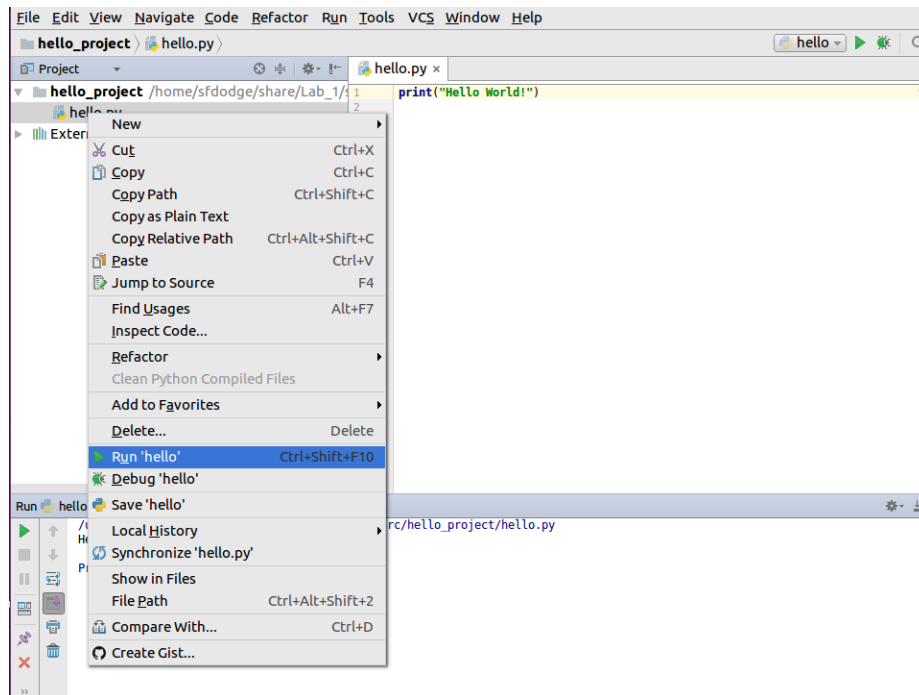


Figure 4: Running Python scripts in PyCharm

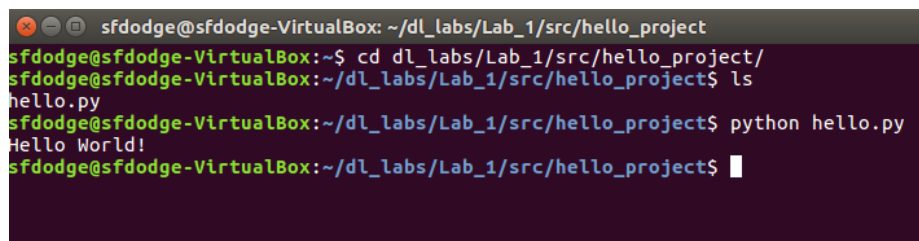


Figure 5: Running Python scripts in the terminal

directory command (`cd directory_name`). In the `hello_project` folder, type `python hello.py` to see the output of the script (Figure 5).

Debugging in PyCharm Now add the following code under the print statement:

```
foo = bar / 2
```

Obviously the code shouldn't run since `bar` has not been defined anywhere. But lets see what happens when we run the code in PyCharm. We see an error message (Figure 6 which tells us which line the error happens and a description of the error. In this case we get: `NameError: name 'bar' is not defined`.

We can fix this error by modifying the code to be:

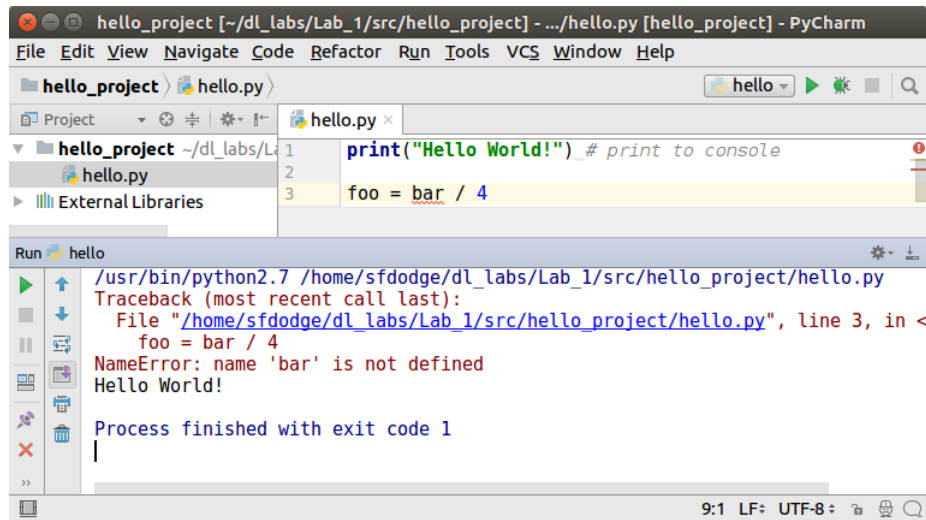


Figure 6: Error message

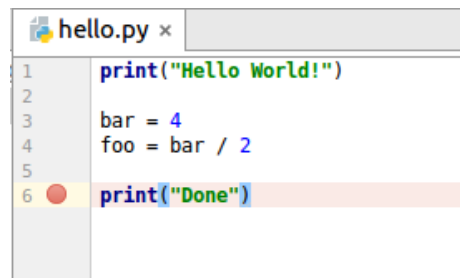


Figure 7: Setting a breakpoint

```
print("Hello World!")
bar = 4
foo = bar / 2
print("Done")
```

Now the code should run with no errors. Since we are not printing `foo` or `bar` to the screen, we don't know if the variables are actually set properly. We could print the variables to the screen using the `print` function, but this becomes tedious with larger scripts with many variables. Instead, we can set a breakpoint in the code and view the values of the variables. Click to the left of the `print("Done")` statement we just added to add a breakpoint (Figure 7).

Now we will run the code in debug mode so that PyCharm will stop at the breakpoint. To run the script in debug mode right click the "hello.py" name in the left pane and select **Debug**. Now the program should halt at the breakpoint, and we can inspect the local variables in the bottom pane (Figure 8).

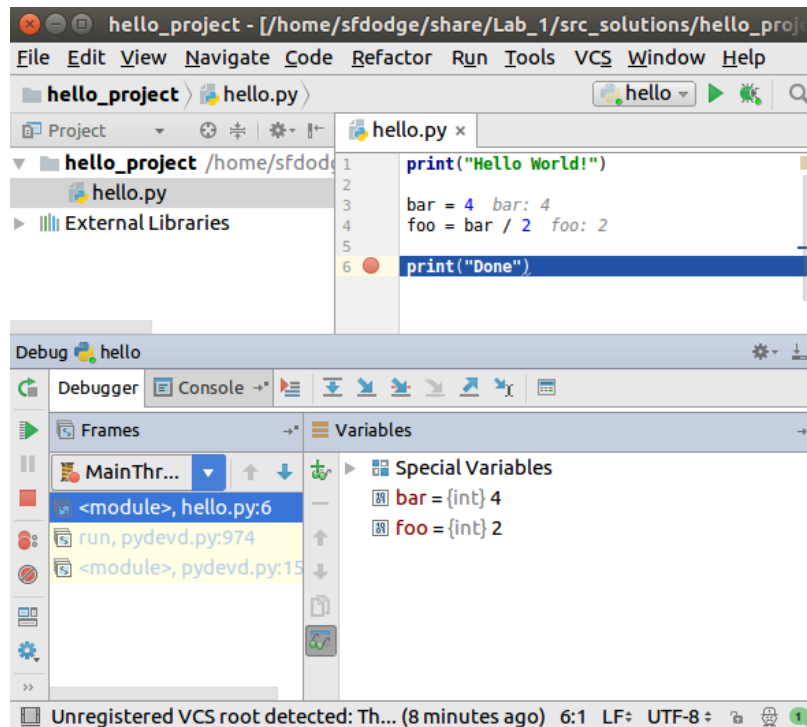


Figure 8: Running in Debug Mode

Extra Information: Other Python tools

PyCharm is not the only tool for debugging/editing Python scripts. Other popular Python editors include Atom, Sublime Text, Vim, Emacs, and Spyder. Debugging can be done directly in the terminal with the `pdb` command. `pdb` behaves very similarly to the popular `gdb` tool for debugging C/C++ code.

4.3 Task 3: Introduction to Unit Testing in Python

During this course we will make use of automated unit testing. In unit testing we write functions that test parts of our code. If all of these test functions pass, we have some “proof” that our code works as intended. Automated unit testing is not completely infallible, as we could have a bug in our testing code. However, in general automated unit testing will lead to more correct code.

Without automated testing, we would typically test the function or code we are currently implementing by manually trying some inputs and observing the outputs. After we are convinced our function is working correctly we would move on to some other function, and assume that the first function we implemented still works. However, it is possible that we unintentionally break the first function while implementing some new functionality. In this case we might not realize this until much later, when it will be difficult to determine what is the cause of the failing first function. However if we have automated testing, we can run all the tests after implementing new functionality. If any tests fail, we will know which functions are failing, and for which situations they are failing for.

Unit testing means that we test small components (units) of our code independently. When testing the entire functionality of a system it may be difficult to pinpoint exactly where the problem is occurring. But if we test small units, we can exactly determine where the problem lies. As a

side benefit, unit testing encourages us to construct our code in well contained functions and classes instead of large monolithic blocks of code. This makes our code easier to read, maintain and reuse.

For this lab it may seem tedious to use unit testing, because all the functions implemented are very simple. However in later labs we will be implementing more complicated functions. For example, we will be implementing the gradient calculation of parameters in a deep neural network. It is possible to derive or implement an incorrect gradient calculation, but we can write a test that calculates the gradient numerically and compare it with our function. If the numeric gradient and our calculated gradient match, then we can be confident that our function has no mistakes.

Note that it is also sometimes useful to test manually. A simple Python script can be written for this purpose. Manual testing can be used in conjunction with automated testing. However if you find yourself repeatedly testing the same thing manually, it might be useful to write an automated test for it.

We will use a simple project to introduce automated unit testing in Python. In PyCharm open the folder `test_project`. In this folder we have two files `src.py` and `test.py`. The `src.py` file includes the source code and `test.py` includes testing functions that are used to test the correctness of our code.

`src.py` contains the following function that adds two numbers and returns the result:

```
def add_two_numbers(a, b):  
    return a + b
```

`test.py` includes a test to insure that if the function is passed 2 and 3 as inputs, the return value will be 5. For testing we will use the built-in Python package `unittest`. We use the `import` statement to use this package in our Python code. PyCharm is integrated with the `unittest` package, so we can visually inspect the test results.

To create a test case we create a class that is a subclass of `unittest.TestCase`. We can create different test case classes for different parts of our code. This is useful if we don't want to run all of the tests every time. Within the test case class we define several test functions to test different parts of our code. Note that there can be an optional `SetUp` function in the test case class that will be called before each test is run. This can be used to set parameters or initial conditions for the test.

Our first test case is defined as follows:

```
def test_integer_add(self):  
    self.assertEqual(add_two_numbers(2, 3), 5)
```

This test will “pass” if the output is correct. The `assertEqual` will pass if the arguments are equal, and fail if the arguments are not equal. Note that there are many types of assert statements found in the `unittest` package, such as `assertTrue` and `assertFalse`.

In PyCharm we can run the tests by right-clicking the `test.py` name in the left pane and clicking `Run Unittests...`. The tests will run and we can see that the test passed in the bottom pane (Figure 9).

Modify the `add_two_numbers` in `src.py` to return an incorrect value (e.g. return $a - b$), and run the test again. You should see that the test indeed fails. In the console we can see why the test failed: `AssertionError: -1 != 5`. This says that the output (-1) is different than the expected output (5). After this, revert the `add_two_numbers` function to the original definition.

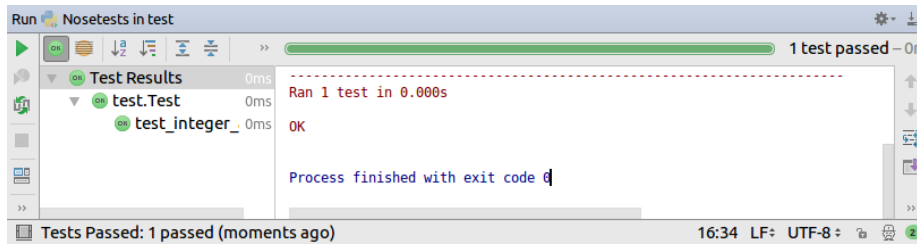


Figure 9: Running a passing test in PyCharm.

Deliverable: Screenshot of failing test

Take a screenshot of the PyCharm window showing the failing test. The screenshot should include the error message that shows the expected output compared with the output. In Ubuntu screenshots can be taken by using `shift + print screen`, or the screenshot can be taken from the host operating system. Name this screenshot `failing_test.png` and save it in the `test_project` folder.

Next we will test if our function can add a floating point number to an integer number. What does Python do here? Does it know how to cast the integer to a float? Lets add the following test to `test.py` and run the tests again.

```
def test_integer_plus_float(self):
    self.assertEqual(add_two_numbers(1.1, 2), 3.1)
```

After running the updated `test.py`, we see that the new test passes (in addition to the first test). This shows that Python is automatically converting the integer to a floating point. We could further test things like overflow, negative numbers, imaginary numbers, etc. We leave this to the interested reader.

Hint: Python indentation

Python uses indentation instead of brackets to separate code blocks. When we add the `test_integer_plus_float` function it needs to be part of the `Test` class. In order for the function to be a member of the class, the indentation must be the same as with the other functions in the class. Use tabs to make sure of this. Additionally, the body of the function must also be indented to be executed properly when the function is called.

Deliverable: `src.py` and `test.py`

You must submit your `src.py` and `test.py` file with the modifications that we made in this Section. Specifically, the `test.py` should contain two test functions.

5 Application: An Image Container Class

In this section we will write a Python class with functions for image manipulations called `ImageClass`. This class will be able to do things like load an image, show the image, plot the histogram, crop the image, etc.

The folder `image_project` has the starting code for this task. Inside this folder there are two files `test.py` and `image_class.py`. In `test.py` we have written unit tests to test the implementation of

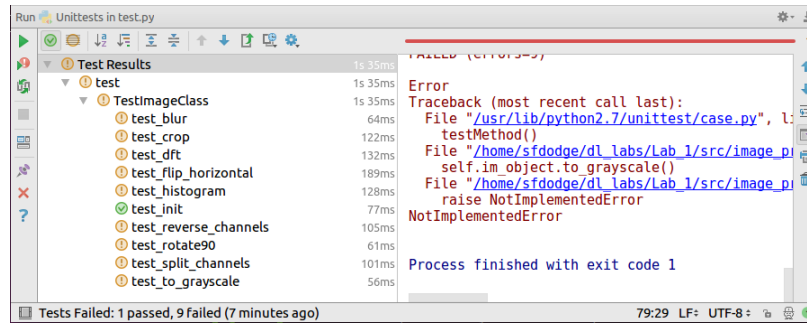


Figure 10: Failing tests for ImageClass

our new `ImageClass`. If we run the unit tests in `test.py` we see that most of our tests have failed (Figure 10).

The functions you need to implement currently have one line: `raise NotImplementedError`. This line forces the test to fail and serves as a reminder that the test is failing because you have not implemented the function. You will need to replace this line with your own code that implements the desired functions. After implementation your code should pass all of the tests.

We have decided to make this a class rather than a collection of functions for two reasons. First it is useful to learn how to program a class, because object oriented programming is very common in Python. Secondly, by using a class we can encapsulate our data without having to pass it back and forth between functions. This lets us use this class to compute several functions on an image very easily. For example the following code uses our class (after implementation) to load an image, flip it horizontally, crop it, blur the image, and save it to a file.

```
from image import ImageClass
im = ImageClass('sparky.png')
im.flip_horizontal()
im.crop(100, 100, 50, 50)
im.blur()
im.save('output.png')
```

In `image_class.py`, we have provided the skeleton of the class. We have provided implementations of the `show`, `save` and `__init__` methods. The rest of the functions need to be implemented. Figure 11 shows the expected outputs for a sample image for the functions you will implement.

At the beginning of the file we will load the required libraries. Note the `as` keyword which can replace a package name with a shorthand name. In this case we use this to replace `numpy` with `np` as a shorthand, and `matplotlib.pyplot` with `plt`. This means that we can use `np.x()` to call a function from Numpy, and `plt.x()` to call a function from PyPlot.

```
import numpy as np # for matrix computations
import imageio # for loading the image
import scipy.signal # for convolution
import matplotlib.pyplot as plt # for plotting functions
```

For this class we will be using many functions from the Numpy package. This package can handle many matrix operations. Images can be thought of as matrices, so the Numpy package can be used for many image processing tasks. Many of the Numpy functions are analogous to similar MATLAB operations, but note that there are some differences (such as matrix multiplication). Table 2 lists

Table 2: Useful Numpy functions

Command	Description
<code>a[0]</code>	Take the 0th index of the array <code>a</code>
<code>a[-2]</code>	Take the second from the last index of the array <code>a</code>
<code>a[::-1]</code>	Reverse the order of array <code>a</code>
<code>A[3,5]</code>	Take the element at the 3rd row and 5th column of matrix <code>A</code>
<code>A[3,:]</code>	Take the 3rd row of matrix <code>A</code>
<code>np.dot(A, B)</code>	Dot product of <code>A</code> and <code>B</code>
<code>np.concatenate((A, B), axis=0)</code>	Concatenate matrices on an existing axis
<code>np.stack((A, B), axis=0)</code>	Concatenate matrices on a new axis
<code>A = A[:, np.newaxis]</code>	Add a new axis to array <code>A</code>
<code>A * B</code>	Elementwise multiplication of matrices <code>A</code> and <code>B</code>
<code>np.zeros((w, h, c))</code>	Create a matrix of zeros with size $w \times h \times c$
<code>np.ones((w, h, c))</code>	Create a matrix of ones with size $w \times h \times c$
<code>A.dtype</code>	Variable storing the current type of matrix <code>A</code>
<code>A.astype('uint8')</code>	Convert the matrix <code>A</code> to have type <code>uint8</code>

some common Numpy commands. You can also refer to the Python tutorial from earlier for help [4], or the official Numpy documentation [1].

Extra Information: Python Doc-strings

In Python programs it is common practice to comment the beginning of a function with a “doc-string” that describes the function. The doc-string is enclosed by `"""` which denotes multi-line comments in Python. We can easily view the doc-string of any function in PyCharm by pressing `ctrl + q` when the text cursor is over that function.

ImageClass.__init__ We have provided the initialization function for the class that takes a single argument `path` that specifies the path of the image file to be load. This function loads the image into a variable called `self.im` using the `imageio.imread` function. The `self.im` variable is part of the `self` object which is like a pointer to the current object. This makes this variable a member of the class. In general it is best to initialize member variables to default values in the initialization function.

```
def __init__(self, path):
    """ Loads the image specified by path """
    self.im = imageio.imread(path)
```

ImageClass.show() We have provided a function that uses `pyplot` to show the image to the screen using the `plt.imshow` and `plt.show` functions. These functions will be very useful in future labs. For this lab, the `show` function can be used to visually check the output of your code while debugging.

```
def show(self):
    """ Shows the image using Matplotlib's pyplot """
    plt.imshow(self.im)
    plt.show()
```

ImageClass.save(path) We have also provided a function that will save the current image to the destination specified by the variable `path`. This function uses the `imageio.imwrite` function.

```
def save(self, path):
    """ Saves the image to path """
    imageio.imwrite(path, self.im)
```

ImageClass.crop(self, r, c, h, w) You need to implement the remaining functions in the `ImageClass` class. The `crop` function crops the image from row `r` and column `c` with height `h` and width `w`. This function should overwrite the value in `self.im` with the new cropped image. We can crop the image by using Numpy slicing:

<https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.indexing.html>

ImageClass.flip_horizontal() This function will flip the image horizontally and store the flipped image back in `self.im`. There is a built-in Numpy function that does this (`np.fliplr`), but we can also do this with indexing.

ImageClass.transpose() This function should transpose the image and store the image back in `self.im`. We want to only transpose the rows and columns. The `np.transpose` function could be used to do this.

ImageClass.reverse_channels() This function should reverse the order of the color channels. Usually the channels in an RGB image are ordered (R, G, B). This function should change this so that the image channels are ordered (B, G, R). Again the resulting image should overwrite the original image in `self.im`. If you plot an image with these reversed channels the colors should be different (Figure 11).

ImageClass.split_channels() This function will split the image by channels and return a list of three matrices: the red channel, the green channel, and the blue channel. This can be accomplished with Numpy indexing. To return multiple variables in Python we can return a list (e.g. `return red, green, blue`).

ImageClass.to_grayscale() To create the grayscale image we combine the channels to create a single channel image and overwrite the `self.im` variable. We can use the previously defined function `split_channels()` to get the individual channels. If we are inside a class function we can call this function using `self.split_channels()`. We use the following formula to combine the channels:

$$I_{gray} = 0.30I_{red} + 0.59I_{green} + 0.11I_{blue} \quad (1)$$

where I_{red} , I_{green} , I_{blue} represent the original color channels, and I_{gray} is the desired grayscale output. The weights are determined from the Rec. 601 standard used in PAL and NTSC systems for television [5].

For this function we have to be careful of the type of the data. If the original image is of size $w \times h \times c$, the grayscale output should have dimensions $w \times h$. The output data type should be the same as the input data type: `uint8` (unsigned 8-bit integer). However, we want to multiply the channels of the image with floating point values. We could first cast the data to floating point using `channel.astype('float64')`. Then we can weight and add the channels. In fact Numpy will automatically cast the variable to float64 if we multiply a matrix with a floating point constant. However, Numpy does not automatically cast back to `uint8`. We can manually cast back to unsigned 8-bit integer using the command `im.astype('uint8')`. Usually before casting to `uint8` we need to make sure that the data lies between 0 and 255, but in this case equation 1 does not change the range of the data so rescaling is not needed.

Extra Information:

Note that after implementing `to_grayscale` there may be bugs in our code. For example if we call `to_grayscale` and then call `split_channels`, our code will likely fail because there are no longer three channels in the image. A better implementation could check the number of channels and raise an exception or return only the single channel. For now we won't worry about this in our implementation.

ImageClass.blur() This function will blur the image using 2D convolution and store the resulting image in `self.im`. We will use a 7×7 averaging filter:

$$\mathbf{Y} = \frac{1}{49} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} ** \mathbf{X} \quad (2)$$

where `**` is the 2D convolution operation. Recall that for convolution we flip the filter, slide the filter over the image, and compute the sum of the multiplication of the corresponding filter and image pixels. For this filter, the convolution will replace each pixel with the average of the 7×7 pixel region around the pixel.

You will need to define the filter in your code as a Numpy matrix using the `numpy.ones` method. For the actual convolution we will be using the function: `scipy.signal.convolve2d`. We want to perform 'valid' convolution, that is convolution with no padding or symmetric extensions at the edges. The documentation for this function can be found here:

<https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve2d.html>

The `scipy.signal.convolve2d` function works on a single channel image. For this function we want to blur the color image. To do this we can blur each channel separately. The channels can be added back together using the `np.stack` command. Note that the convolution function may change the data type of our result, so again we need to convert the image back to `uint8`.

ImageClass.plot_histogram() We will use pyplot again to plot the histogram. In this case we have a 1-D array that we want to plot. For this function we will need to first convert the image to grayscale. The histogram can then be computed using the `np.histogram` function. We want to create a histogram with 256 bins, covering the full range from 0 to 255. In addition to plotting the histogram, this function should also **return the computed histogram**. The histogram should be an array of 256 values with each element of the array corresponding to the count of pixels with grayscale value equal to the index of the element. Plotting can be performed with Pyplot's `plt.plot` function.

ImageClass.compute_dft() Next we will compute the DFT of the image. It is possible to directly compute the DFT using the Numpy function `numpy.fft.fft2`. However, we want to practice using the lower level Numpy functions to calculate the DFT ourselves.

The DFT can be computed using a DFT matrix [6]. The DFT matrix is defined as:

$$\mathbf{W} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \omega^6 & \dots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \omega^9 & \dots & \omega^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \omega^{3(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{bmatrix} \quad (3)$$

where N is the size of the matrix we want to compute the DFT for, and $\omega = e^{-2\pi i/N}$. Using this matrix we can compute the DFT of our image as follows:

$$\mathbf{Y} = \mathbf{W}\mathbf{X}\mathbf{W} \quad (4)$$

where \mathbf{X} is the original image and \mathbf{Y} is the output of the 2D DFT.

We will implement this using Numpy's `np.dot` function. We have written a function `_DFT_matrix()` that computes the matrix \mathbf{W} given the size of the image. For this function we will also need to convert the image to grayscale before computing the DFT.

ImageClass.plot_save_dft(self, dft, fname) This function plots the magnitude of the 2D DFT matrix (such as the one computed using the `compute_dft()` function) and saves the plot as an image in a file whose name and type are specified by `fname`. This function is provided, and you need only to run it to plot a 2D DFT and save the plot. You need to set `fname` to the desired filename.

```
def plot_save_dft(self, dft, fname="dft_plot.png"):
    """
    Plot and save the 2D dft of the image

    Parameters
    -----
    dft      : complex
               2D array contain the dft of an image.
    fname    : string
               The file name to save the 2D dft.
    """
    dft = np.log10(np.abs(dft) + 1)
    dft = np.roll(dft, dft.shape[0]/2, axis=0)
    dft = np.roll(dft, dft.shape[1]/2, axis=1)
    dft = dft - dft.min()
    dft = dft / dft.max()
    plt.imshow(dft, cmap='gray')
    scipy.misc.imsave(fname, dft)
    plt.show()
```


Hint:

All of these functions can be implemented in less than 10 lines each. Some functions can be implemented with a single Numpy function call. If one of your functions is becoming very long, there is likely a simple Numpy function that you are missing.

Important:

Do not import any other packages. You should be able to implement the entire `ImageClass` using only `numpy`, `matplotlib`, `imageio` and `scipy` packages.

Deliverable: `image_class.py`

You must implement the following functions in the `image_class.py` file:

Function	Description
<code>crop(r,c,w,h)</code>	Crop the image
<code>flip_horizontal()</code>	Horizontally flip the image
<code>transpose()</code>	Transpose the image
<code>reverse_channels()</code>	Reverse the input channels (RGB becomes BGR)
<code>split_channels()</code>	Split the image by channel and return channels
<code>to_grayscale()</code>	Convert the color image to grayscale
<code>blur()</code>	Blur the image with a 5×5 averaging filter
<code>plot_histogram()</code>	Plot the histogram of the image
<code>compute_dft()</code>	Compute the DFT using the DFT matrix and dot product

Deliverable: `image_blur_dft.py`

Use the functions in `image_class.py` and write a script to perform the following tasks in the order specified below:

- 1) Load the image `sparky.png`.
- 2) Compute the 2D DFT of the input image; generate a plot of the DFT magnitude and save the plot in a file called `orig_img_dft.png`.
- 3) Blur the image using a 7×7 averaging filter.
- 4) Compute the 2D DFT of the blurred image; generate a plot of the DFT magnitude and save the plot in a file called `blur_img_dft.png`.

We have provided an empty script called `image_blur_dft.py`; please write your code in this script. Run the script and compare the generated DFT plots. What do you observe? Provide your observations as instructed in the commented section at the beginning of the script. Submit the completed `image_blur_dft.py` file along with the two saved images corresponding to the two generated DFT plots.

Important:

Do not use functions from other packages. You should be able to implement these tasks using functions from `ImageClass`. There will not be a test unit for this script.

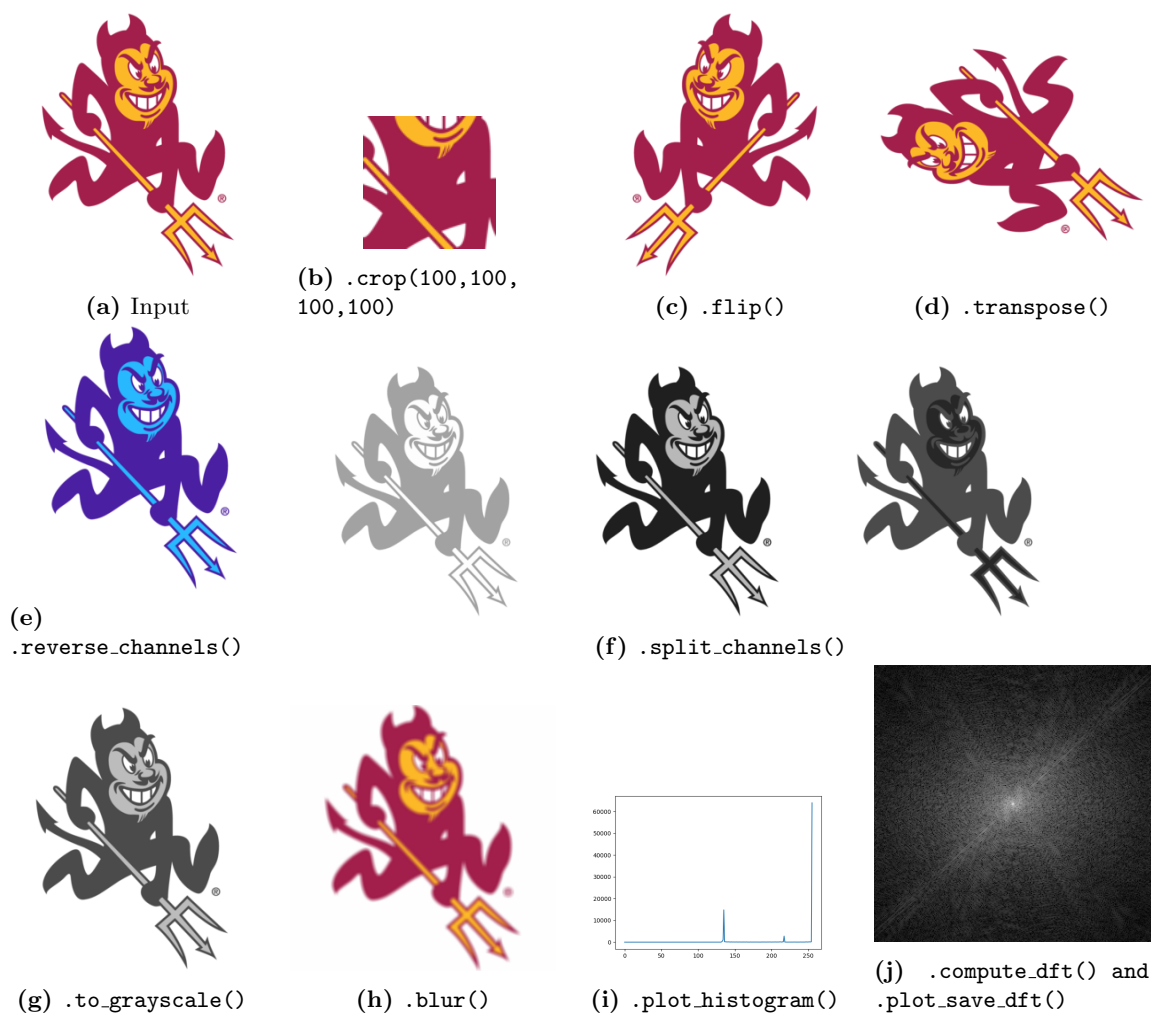


Figure 11: Expected outputs of functions implemented in `ImageClass`

6 Submission Instructions

We will test your code using unit tests similar to (but not identical to) the unit tests that we used in this lab. Thus you must be sure that your code works for all cases, not just the particular cases in the given unit tests. Do not change any of the names of the classes or functions. Also do not change the folder structure of the code as this is assumed by the testing code used for grading. You will be graded based on how many tests your code passes.

You must use the same folder structure as in the provided `src` folder. Replace the name of the `src` folder with `FIRSTNAME_LASTNAME_LAB1` and zip the folder for submission.

The following are the grading guidelines we will use for this lab:

References

- [1] “Numpy reference.” <https://docs.scipy.org/doc/numpy-1.13.0/reference/>, 2017.

Table 3: Grading rubric

Points	Description
10	Correct file names and folder structure
10	<code>hello_project</code> folder with modified <code>hello.py</code>
10	<code>test_project</code> folder with added unit test
10	Screenshot of the failing test
45	<code>image_project</code> folder with code that passes all tests
15	Two DFT figures and script <code>image_blur_dft.py</code>
Total 100	

- [2] “Scipy reference.” <https://docs.scipy.org/doc/scipy-1.0.0/reference/>, 2017.
- [3] “Matplotlib pyplot reference.” https://matplotlib.org/api/pyplot_api.html, 2017.
- [4] J. Johnson, “Python numpy tutorial.” <http://cs231n.github.io/python-numpy-tutorial/>, 2017.
- [5] “Luma coding in video systems.” https://en.wikipedia.org/wiki/Grayscale#Luma_coding_in_video_systems, 2017.
- [6] Wikipedia, “DFT matrix — Wikipedia, the free encyclopedia.” <http://en.wikipedia.org/w/index.php?title=DFT%20matrix&oldid=811427639>, 2017.