Mastering OpenCV 4 with Python

A practical guide covering topics from image processing, augmented reality to deep learning with OpenCV 4 and Python 3.7

Packt
www.packt.com

Alberto Fernández Villán

Mastering OpenCV 4 with Python

A practical guide covering topics from image processing, augmented reality to deep learning with OpenCV 4 and Python 3.7

Alberto Fernández Villán

Packt

BIRMINGHAM - MUMBAI

Mastering OpenCV 4 with Python

Commissioning Editor: Richa Tripathi
Acquisition Editor: Alok Dhuri
Content Development Editor: Manjusha Mantri
Technical Editor: Riddesh Dawne
Copy Editor: Safis Editing
Project Coordinator: Prajakta Naik
Proofreader: Safis Editing
Indexer: Rekha Nair
Graphics: Jisha Chirayil
Production Coordinator: Shraddha Falebhai

First published: March 2019

Production reference: 1280319

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78934-491-2

www.packtpub.com

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Alberto Fernández Villán is a software engineer with more than 12 years of experience in developing innovative solutions. In the last couple of years, he has been working in various projects related to monitoring systems for industrial plants, applying both Internet of Things (IoT) and big data technologies. He has a Ph.D. in computer vision (2017), a deep learning certification (2018), and several publications in connection with computer vision and machine learning in journals such as *Machine Vision and Applications*, *IEEE Transactions on Industrial Informatics*, *Sensors*, *IEEE Transactions on Industry Applications*, *IEEE Latin America Transactions*, and more. As of 2013, he is a registered and active user (*albertofernandez*) on the Q&A OpenCV forum.

About the reviewers

Wilson Choo is a computer vision engineer working on validating computer vision and deep learning algorithms on many different hardware configurations. His strongest skills include algorithm benchmarking, integration, app development, and test automation.

He is also a machine learning and computer vision enthusiast. He often researches trending CVDL algorithms and applies them to solve modern-day problems. Besides that, Wilson likes to participate in hackathons, where he showcases his ideas and competes with other developers. His favorite programming languages are Python and C++.

Vincent Kok is a maker and a software platform application engineer in the transportation industry. He graduated from USM with a MSc in embedded system engineering. Vincent actively involves himself with the developer community, as well as attending Maker Faire events held around the world, such as in Shenzhen in 2014, and in Singapore and Tokyo in 2015. Designing electronics hardware kits and giving soldering/Arduino classes for beginners are some of his favorite ways to spend his free time. Currently, his focus is in computer vision technology, software test automation, deep learning, and constantly keeping himself up to date with the latest technology.

Rubén Usamentiaga is a tenured associate professor in the department of computer science and engineering at the University of Oviedo. He received his M.S. and Ph.D. degrees in computer science from the University of Oviedo in 1999 and 2005, respectively. He has participated in 4 European projects, 3 projects of the National R&D Plan, 2 projects of the Regional Plan of the Principado of Asturias, and 14 contracts with companies. He is the author of more than 60 publications in JCR journals (25 of Q1) and more than 50 publications in international conferences. In addition, he has completed a 6-month research stay at the *Aeronautical Technology Center* and a 3-month research stay at the University of Laval in Quebec.

Arun Ponnusamy, works as a computer vision research engineer at an AI start-up in India. He is a lifelong learner, passionate about image processing, computer vision, and machine learning. He is an engineering graduate from PSG College of Technology, Coimbatore. He started his career at MulticoreWare Inc., where he spent most of his time on image processing, OpenCV, software optimization, and GPU computing.

Arun loves to understand computer vision concepts clearly and explain them in an intuitive way in his blog and in meetups. He has created an open source Python library for computer vision, named cvlib, which is aimed at simplicity and user friendliness. He is currently working on object detection, action recognition, and generative networks.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

In a nutshell, this book is about computer vision using OpenCV, which is a computer vision (and also machine learning) library, and the Python programming language. You may be wondering why OpenCV and Python? That is really a good question, which we address in the first chapter of this book. To summarize, OpenCV is the best open source computer vision library (BSD license—it is free for both academic and commercial use), offering more than 2,500 optimized algorithms, including state-of-the-art computer vision algorithms, and it also has machine learning and deep learning support. OpenCV is written in optimized C/C++, but it provides Python wrappers. Therefore, this library can be used in your Python programs. In this sense, Python is considered the ideal language for scientific computing because it stimulates rapid prototyping and has a lot of prebuilt libraries for every aspect of your computer vision projects.

As introduced in the previous paragraph, there are many prebuilt libraries you can use in your projects. Indeed, in this book, we use lots of them, showing you that it's really easy to install and use new libraries. Libraries such as Matplotlib, scikit-image, SciPy, dlib, face-recognition, Pillow, cvlib, Keras, TensorFlow, and Flask will be used in this book to show you the potential of the Python ecosystem. If this is the first time that you're reading about these libraries, don't worry, because we introduce *hello world* examples for almost all of these libraries.

This book is a complete resource for creating advanced applications with Python and OpenCV using various techniques, such as facial recognition, target tracking, augmented reality, object detection, and classification, among others. In addition, this book explores the potential of machine learning and deep learning techniques in computer vision applications using the Python ecosystem.

It's time to dive deeper into the content of this book. We are going to introduce you to what this book covers, including a short paragraph talking about each chapter of the book. So, let's get started!

Who this book is for

This book is great for students, researchers, and developers with basic Python programming knowledge who are new to computer vision and who would like to dive deeper into this world. It's assumed that readers have some previous experience with Python. A basic understanding of image data (for example, pixels and color channels) would also be helpful, but is not necessary, because these concepts are covered in the book. Finally, standard mathematical skills are required.

What this book covers

`Chapter 1`, *Setting Up OpenCV*, shows how to install everything you need to start programming with Python and OpenCV. You'll also be introduced to general terminology and concepts to contextualize what you will learn, establishing and setting the bases in relation to the main concepts of computer vision using OpenCV.

`Chapter 2`, *Image Basics in OpenCV*, demonstrates how to start writing your first scripts, in order to introduce you to the OpenCV library.

`Chapter 3`, *Handling Files and Images*, shows you how to cope with files and images, which are necessary for building your computer vision applications.

`Chapter 4`, *Constructing Basic Shapes in OpenCV*, covers how to draw shapes—from basic ones to some that are more advanced—using the OpenCV library.

`Chapter 5`, *Image Processing Techniques*, introduces most of the common image processing techniques you will need for your computer vision projects.

`Chapter 6`, *Constructing and Building Histograms*, shows how to both create and understand histograms, which are a powerful tool for understanding image content.

`Chapter 7`, *Thresholding Techniques*, introduces the main thresholding techniques you will need for your computer vision applications as a key process of image segmentation.

`Chapter 8`, *Contour Detection, Filtering, and Drawing*, shows how to deal with contours, which are used for shape analysis and for both object detection and recognition.

`Chapter 9`, *Augmented Reality*, teaches you how to build your first augmented reality application.

`Chapter 10`, *Machine Learning with OpenCV*, introduces you to the world of machine learning. You will see how machine learning can be used in your computer vision projects.

`Chapter 11`, *Face Detection, Tracking, and Recognition*, demonstrates how to create face processing projects using state-of-the-art algorithms, in connection with face detection, tracking, and recognition.

`Chapter 12`, *Introduction to Deep Learning*, introduces you to the world of deep learning with OpenCV and also some deep learning Python libraries (TensorFlow and Keras).

`Chapter 13`, *Mobile and Web Computer Vision with Python and OpenCV*, shows how to create computer vision and deep learning web applications using Flask.

To get the most out of this book

With the aim of making the most of this book, you have to take into account two simple but key considerations:

1. Some basic knowledge of Python programming is assumed as all the scripts and examples in this book are in Python.
2. The NumPy and OpenCV-Python packages are highly interconnected (you will learn why in this book). In spite of NumPy examples being fully explained, the learning curve can be softened if some NumPy knowledge is acquired before starting this book.

Download the example code files

You can download the example code files for this book from your account at `www.packt.com`. If you purchased this book elsewhere, you can visit `www.packt.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-OpenCV-4-with-Python`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/9781789344912_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The code for `build_sample_image()` is provided next."

A block of code is set as follows:

```
channels = cv2.split(img)
   eq_channels = []
   for ch in channels:
       eq_channels.append(cv2.equalizeHist(ch))
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
Hu moments (original): '[ 1.92801772e-01 1.01173781e-02 5.70258405e-05
1.96536742e-06 2.46949980e-12 -1.88337981e-07 2.06595472e-11]'
 Hu moments (rotation): '[ 1.92801772e-01 1.01173781e-02 5.70258405e-05
1.96536742e-06 2.46949980e-12 -1.88337981e-07 2.06595472e-11]'
 Hu moments (reflection): '[ 1.92801772e-01 1.01173781e-02 5.70258405e-05
1.96536742e-06 2.46949980e-12 -1.88337981e-07 -2.06595472e-11]'
```

Any command-line input or output is written as follows:

```
$ mkdir opencv-project
$ cd opencv-project
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packt.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

1
Section 1: Introduction to OpenCV 4 and Python

In this first section of the book, you will be introduced to the OpenCV library. You will learn how to install everything you need to start programming with Python and OpenCV. Also, you will familiarize yourself with the general terminology and concepts to contextualize what you will learn, establishing the foundations you will need in order to grasp the main concepts of this book. Additionally, you will start writing your first scripts in order to get to grips with the OpenCV library, and you will also learn how to work with files and images, which are necessary for building your computer vision applications. Finally, you will see how to draw basic and advanced shapes using the OpenCV library.

The following chapters will be covered in this section:

- `Chapter 1`, *Setting Up OpenCV*
- `Chapter 2`, *Image Basics in OpenCV*
- `Chapter 3`, *Handling Files and Images*
- `Chapter 4`, *Constructing Basic Shapes in OpenCV*

1
Setting Up OpenCV

Mastering OpenCV 4 with Python will give you the knowledge to build projects involving **Open Source Computer Vision Library** (**OpenCV**) and Python. These two *technologies* (the first one is a programming language, while the second one is a computer vision and machine learning library) will be introduced. Also, you will learn why the combination of OpenCV and Python has the potential to build every kind of computer application. Finally, an introduction about the main concepts related to the content of this book will be provided.

In this chapter, you will be given step-by-step instructions to install everything you need to start programming with Python and OpenCV. This first chapter is quite long, but do not worry, because it is divided into easily assimilated sections, starting with general terminology and concepts, which assumes that the reader is new to this information. At the end of this chapter, you will be able to build your first project involving Python and OpenCV.

The following topics will be covered in this chapter:

- A theoretical introduction to the OpenCV library
- Installing Python OpenCV and other packages
- Running samples, documentation, help, and updates
- Python and OpenCV project structure
- First Python and OpenCV project

Technical requirements

This chapter and subsequent chapters are focused on Python (a programming language) and OpenCV (a computer vision library) concepts in connection with computer vision, machine learning, and deep learning techniques (among others). Therefore, Python (`https://www.python.org/`) and OpenCV (`https://opencv.org/`) should be installed on your computer. Moreover, some Python packages related to scientific computing and data science should also be installed (for example, NumPy (`http://www.numpy.org/`) or Matplotlib (`https://matplotlib.org/`)).

Additionally, it is recommended that you install an **integrated development environment** (**IDE**) software package because it facilitates computer programmers with software development. In this sense, a Python-specific IDE is recommended. The *de facto* Python IDE is PyCharm, which can be downloaded from `https://www.jetbrains.com/pycharm/`.

Finally, in order to facilitate GitHub activities (for example, cloning a repository), you should install a Git client. In this sense, GitHub provides desktop clients that include the most common repository actions. For an introduction to Git commands, check out `https://education.github.com/git-cheat-sheet-education.pdf`, where commonly used Git command-line instructions are summarized. Additionally, instructions for installing a Git client on your operating system are included.

The GitHub repository for this book, which contains all the supporting project files necessary to work through the book from the first chapter to the last, can be accessed at `https://github.com/PacktPublishing/Mastering-OpenCV-4-with-Python`.

Finally, it should be noted that the README file of the GitHub repository for Mastering OpenCV with Python includes the following, which is also attached here for the sake of completeness:

- Code testing specifications
- Hardware specifications
- Related books and products

Code testing specifications

Mastering OpenCV 4 with Python requires some installed packages, which you can see here:

- `Chapter 1`, *Setting Up OpenCV*: `opencv-contrib-python`
- `Chapter 2`, *Image Basics in OpenCV*: `opencv-contrib-python` and `matplotlib`

- `Chapter 3`, *Handling Files and Images*: `opencv-contrib-python` and `matplotlib`
- `Chapter 4`, *Constructing Basic Shapes in OpenCV*: `opencv-contrib-python` and `matplotlib`
- `Chapter 5`, *Image Processing Techniques*: `opencv-contrib-python` and `matplotlib`
- `Chapter 6`, *Constructing and Building Histograms*: `opencv-contrib-python` and `matplotlib`
- `Chapter 7`, *Thresholding Techniques*: `opencv-contrib-python`, `matplotlib`, `scikit-image`, and `scipy`
- `Chapter 8`, *Contours Detection, Filtering, and Drawing*: `opencv-contrib-python` and `matplotlib`
- `Chapter 9`, *Augmented Reality*: `opencv-contrib-python` and `matplotlib`
- `Chapter 10`, *Machine Learning with OpenCV*: `opencv-contrib-python` and `matplotlib`
- `Chapter 11`, *Face Detection, Tracking, and Recognition*: `opencv-contrib-python`, `matplotlib`, `dlib`, `face-recognition`, `cvlib`, `requests`, `progressbar`, `keras`, and `tensorflow`
- `Chapter 12`, *Introduction to Deep Learning*: `opencv-contrib-python`, `matplotlib`, `tensorflow`, and `keras`
- `Chapter 13`, *Mobile and Web Computer Vision with Python and OpenCV*: `opencv-contrib-python`, `matplotlib`, `flask`, `tensorflow`, `keras`, `requests`, and `pillow`

Make sure that the version numbers of your installed packages are equal to, or greater than, versions specified here to ensure that the code examples run correctly.

If you want to install the exact versions this book was tested on, include the version when installing from `pip`, which is indicated as follows.

Run the following command to install the both main and contrib modules:

- Install `opencv-contrib-python`:

```
pip install opencv-contrib-python==4.0.0.21
```

It should be noted that OpenCV requires `numpy`. `numpy-1.16.1` has been installed when installing `opencv-contrib-python==4.0.0.21`.

Run the following command to install Matplotlib library:

- Install `matplotlib`:

```
pip install matplotlib==3.0.2
```

It should be noted that `matplotlib` requires `kiwisolver`, `pyparsing`, `six`, `cycler`, and `python-dateutil`.

`cycler-0.10.0`, `kiwisolver-1.0.1`, `pyparsing-2.3.1`, `python-dateutil-2.8.0`, and `six-1.12.0` have been installed when installing `matplotlib==3.0.2`.

Run the following command to install library which contains collections of algorithm for image processing:

- Install `scikit-image`:

```
pip install scikit-image==0.14.2
```

It should be noted that `scikit-image` requires `cloudpickle`, `decorator`, `networkx`, `numpy`, `toolz`, `dask`, `pillow`, `PyWavelets`, and `six`.

`PyWavelets-1.0.1`, `cloudpickle-0.8.0`, `dask-1.1.1`, `decorator-4.3.2`, `networkx-2.2`, `numpy-1.16.1`, `pillow-5.4.1`, `six-1.12.0`, and `toolz-0.9.0` have been installed when installing `scikit-image==0.14.2`.

If you need SciPy, you can install it with the following command:

- Install `scipy`:

```
pip install scipy==1.2.1
```

It should be noted that `scipy` requires `numpy`.

`numpy-1.16.1` has been installed when installing `scipy==1.2.1`.

Run the following command to install `dlib` library:

- Install `dlib`:

```
pip install dlib==19.8.1
```

To install the face recognition library, run the following command:

- Install `face-recognition`:

```
pip install face-recognition==1.2.3
```

It should be noted that `face-recognition` requires `dlib`, `Click`, `numpy`, `face-recognition-models`, and `pillow`.

`dlib-19.8.1`, `Click-7.0`, `face-recognition-models-0.3.0`, and `pillow-5.4.1` have been installed when installing `face-recognition==1.2.3`.

Run the following command to install open source computer vision library:

- Install `cvlib`:

```
pip install cvlib==0.1.8
```

To install requests library run the following command:

- Install `requests`:

```
pip install requests==2.21.0
```

It should be noted that `requests` requires `urllib3`, `chardet`, `certifi`, and `idna`.

`urllib3-1.24.1`, `chardet-3.0.4`, `certifi-2018.11.29`, and `idna-2.8` have been installed when installing `requests==2.21.0`.

Run the following command to install text progress bar library:

- Install `progressbar`:

```
pip install progressbar==2.5
```

Run the following command to install Keras library for deep learning:

- Install `keras`:

```
pip install keras==2.2.4
```

It should be noted that `keras` requires `numpy`, `six`, `h5py`, `keras-applications`, `scipy`, `keras-preprocessing`, and `pyyaml`.

`h5py-2.9.0`, `keras-applications-1.0.7`, `keras-preprocessing-1.0.9`, `numpy-1.16.1 pyyaml-3.13`, and `scipy-1.2.1 six-1.12.0` have been installed when installing `keras==2.2.4`.

Run the following command to install TensorFlow library:

- Install `tensorflow`:

```
pip install tensorflow==1.12.0
```

It should be noted that TensorFlow requires `termcolor`, `numpy`, `wheel`, `gast`, `six`, `setuptools`, `protobuf`, `markdown`, `grpcio`, `werkzeug`, `tensorboard`, `absl-py`, `h5py`, `keras-applications`, `keras-preprocessing`, and `astor`.

`termcolor-1.1.0`, `numpy-1.16.1`, `wheel-0.33.1`, `gast-0.2.2`, `six-1.12.0`, `setuptools-40.8.0`, `protobuf-3.6.1`, `markdown-3.0.1`, `grpcio-1.18.0`, `werkzeug-0.14.1`, `tensorboard-1.12.2`, `absl-py-0.7.0`, `h5py-2.9.0`, `keras-applications-1.0.7`, `keras-preprocessing-1.0.9`, and `astor-0.7.1` have been installed when installing `tensorflow==1.12.0`.

Run the following command to install Flask library:

- Install `flask`:

```
pip install flask==1.0.2
```

It should be noted that `flask` requires `Werkzeug`, `click`, `itsdangerous`, and `MarkupSafe Jinja2`.

`Jinja2-2.10`, `MarkupSafe-1.1.1`, `Werkzeug-0.14.1`, `click-7.0`, and `itsdangerous-1.1.0` have been installed when installing `flask==1.0.2`.

Hardware specifications

The hardware specifications are as follows:

- 32-bit or 64-bit architecture
- 2+ GHz CPU
- 4 GB RAM
- At least 10 GB of hard disk space available

Understanding Python

Python is an interpreted high-level and general-purpose programming language with a dynamic type system and automatic memory management. The official home of the Python programming language is `https://www.python.org/`. The popularity of Python has risen steadily over the past decade. This is because Python is a very important programming language in some of today's most exciting and challenging technologies. **Artificial intelligence** (**AI**), machine learning, neural networks, deep learning, **Internet of Things** (**IoT**), and robotics (among others) rely on Python.

Here are some advantages of Python:

- Python is considered a perfect language for scientific computing, mainly for four reasons:
    - It is very easy to understand.
    - It has support (via packages) for scientific computing.
    - It removes many of the complexities other programming languages have.
    - It has a simple and consistent syntax.
- Python stimulates rapid prototyping because it helps in easy writing and execution of code. Indeed, Python can implement the same logic with as little as one-fifth of the code as compared to other programming languages.
- Python has a lot of prebuilt libraries (NumPy, SciPy, scikit-learn) for every need of your AI project. Python benefits from a rich ecosystem of libraries for scientific computing.
- It is an independent platform, which allows developers to save time in testing on different platforms.
- Python offers some tools, such as Jupyter Notebook, that can be used to share scripts in an easy and comfortable way. This is perfect in scientific computing because it stimulates collaboration in an interactive computational environment.

Introducing OpenCV

OpenCV is a C++ programming library, with real-time capabilities. As it is written in optimized C/C++, the library can profit from multi-core processing. A theoretical introduction about the OpenCV library is carried out in the next section.

In connection with the OpenCV library, here are some reasons for its popularity:

- Open source computer vision library
- OpenCV (BSD license—`https://en.wikipedia.org/wiki/BSD_licenses`) is free
- Specific library for image processing
- It has more than 2,500 optimized algorithms, including state-of-the-art computer vision algorithms
- Machine learning and deep learning support
- The library is optimized for performance
- There is a big community of developers using and supporting OpenCV
- It has C++, Python, Java, and MATLAB interfaces
- The library supports Windows, Linux, Android, and macOS
- Fast and regular updates (official releases now occur every six months)

Contextualizing the reader

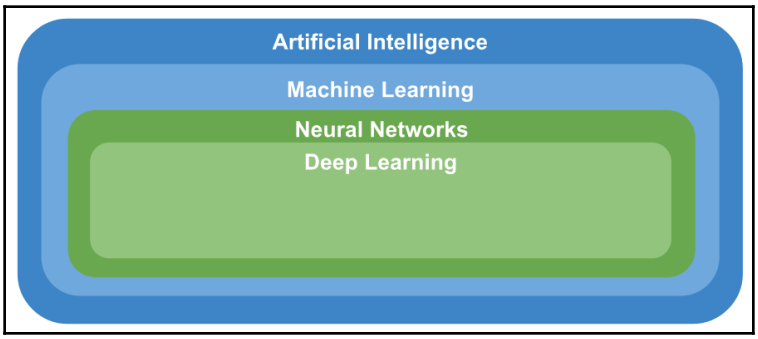
In order to contextualize the reader, it is necessary to establish and set the bases in relation to the main concepts concerning the theme of this book. The last few years have seen considerable interest in AI and machine learning, specifically in the area of deep learning. These terms are used interchangeably and very often confused with each other. For the sake of completeness and clarification, these terms are briefly described next.

AI refers to a set of technologies that enable machines – computers or robotic systems – to process information in the same way humans would.

The term AI is commonly used as an umbrella for a machine technology in order to provide intelligence covering a wide range of methods and algorithms. **Machine Learning** is the process of programming computers to learn from historical data to make predictions on new data. Machine learning is a sub-discipline of AI and refers to statistical techniques that machines use on the basis of learned interrelationships. On the basis of data gathered or collected, algorithms are independently *learned* by computers. These algorithms and methods include support vector machine, decision tree, random forest, logistic regression, Bayesian networks, and neural networks.

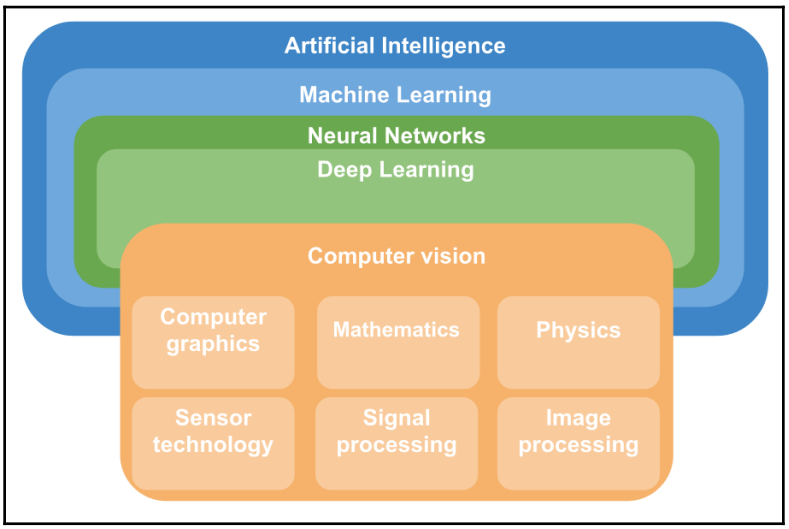
Neural Networks are computer models for machine learning that are based on the structure and functioning of the biological brain. An artificial neuron processes a plurality of input signals and, in turn, when the sum of the input signals exceeds a certain threshold value, signals to further adjacent neurons will be sent. **Deep Learning** is a subset of machine learning that operates on large volumes of unstructured data, such as human speech, text, and images. A deep learning model is an artificial neural network that comprises multiple layers of mathematical computation on data, where results from one layer are fed as input into the next layer in order to classify the input data and/or make a prediction.

Therefore, these concepts are interdependent in a hierarchical way, AI being the broadest term and deep learning the most specific. This structure can be seen in the next diagram:



Computer vision is an interdisciplinary field of **Artificial Intelligence** that aims to give computers and other devices with computing capabilities a high-level understanding from both digital images and videos, including functionality for acquiring, processing, and analyzing digital images. This is why computer vision is, partly, another sub-area of **Artificial Intelligence**, heavily relying on machine learning and deep learning algorithms to build computer vision applications. Additionally, **Computer vision** is composed of several technologies working together—**Computer graphics**, **Image processing**, **Signal processing**, **Sensor technology**, **Mathematics**, or even **Physics**.

Therefore, the previous diagram can be completed to introduce the computer vision discipline:

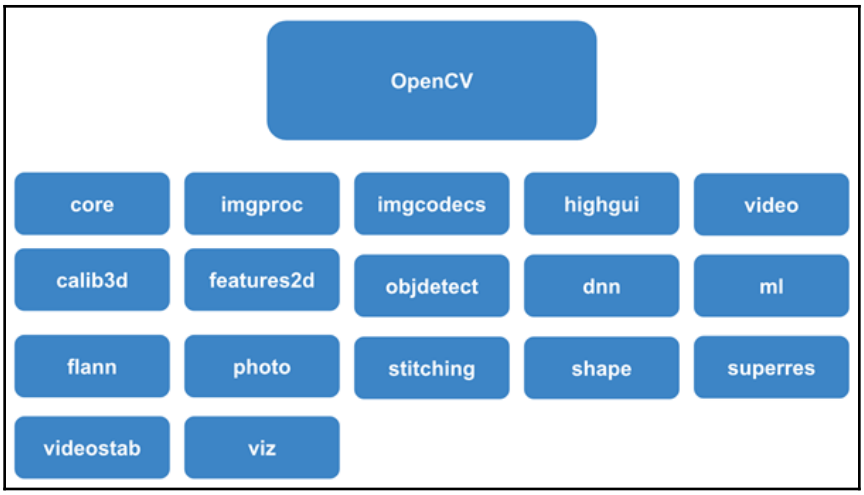


A theoretical introduction to the OpenCV library

OpenCV is a programming library with real-time computer vision capabilities and it is free for both academic and commercial use (BSD license). In this section, an introduction about the OpenCV library will be given, including its main modules and other useful information in connection with the library.

OpenCV modules

OpenCV (since version 2) is divided into several modules, where each module can be understood, in general, as being dedicated to one group of computer vision problems. This division can be seen in the next diagram, where the main modules are shown:



OpenCV modules are shortly described here:

- **core**: Core functionality. Core functionality is a module defining basic data structures and also basic functions used by all other modules in the library.
- **imgproc**: Image processing. An image-processing module that includes image filtering, geometrical image transformations, color space conversion, and histograms.
- **imgcodecs**: Image codecs. Image file reading and writing.
- **videoio**: Video I/O. An interface to video capturing and video codecs.
- **highgui**: High-level GUI. An interface to UI capabilities. It provides an interface to easily do the following:
    - Create and manipulate windows that can display/show images
    - Add trackbars to the windows, keyboard commands, and handle mouse events
- **video**: Video analysis. A video-analysis module including background subtraction, motion estimation, and object-tracking algorithms.
- **calib3d**: Camera calibration and 3D reconstruction. Camera calibration and 3D reconstruction covering basic multiple-view geometry algorithms, stereo correspondence algorithms, object pose estimation, both single and stereo camera calibration, and also 3D reconstruction.

- **features2d**: 2D features framework. This module includes feature detectors, descriptors, and descriptor matchers.
- **objdetect**: Object detection. Detection of objects and instances of predefined classes (for example, faces, eyes, people, and cars).
- **dnn**: **Deep neural network** (**DNN**) module. This module contains the following:
    - API for new layers creation
    - Set of built useful layers
    - API to construct and modify neural networks from layers
    - Functionality for loading serialized networks models from different deep learning frameworks
- **ml**: Machine learning. The **Machine Learning Library** (**MLL**) is a set of classes and methods that can be used for classification, regression, and clustering purposes.
- **flann**: Clustering and search in multi-dimensional spaces. **Fast Library for Approximate Nearest Neighbors** (**FLANN**) is a collection of algorithms that are highly suited for fast nearest-neighbor searches.
- **photo**: Computational photography. This module provides some functions for computational photography.
- **stitching**: Images stitching. This module implements a stitching pipeline that performs automatic panoramic image stitching.
- **shape**: Shape distance and matching. Shape distance and matching module that can be used for shape matching, retrieval, or comparison.
- **superres**: Super-resolution. This module contains a set of classes and methods that can be used for resolution enhancement.
- **videostab**: Video stabilization. This module contains a set of classes and methods for video stabilization.
- **viz**: 3D visualizer. This module is used to display widgets that provide several methods to interact with scenes and widgets.

OpenCV users

Regardless of whether you are a professional software developer or a novice programmer, the OpenCV library will be interesting for graduate students, researchers, and computer programmers in image-processing and computer vision areas. The library has gained popularity among scientists and academics because many state-of-the-art computer vision algorithms are provided by this library.

Additionally, it is often used as a teaching tool for both computer vision and machine learning. It should be taken into account that OpenCV is robust enough to support real-world applications. That is why OpenCV can be used for non-commercial and commercial products. For example, it is used by companies such as Google, Microsoft, Intel, IBM, Sony, and Honda. Research institutes in leading universities, such as MIT, CMU, or Stanford, provide support for the library. OpenCV has been adopted all around the world. It has more than 14 million downloads and more than 47,000 people in its community.

OpenCV applications

OpenCV is being used for a very wide range of applications:

- 2D and 3D feature toolkits
- Street view image stitching
- Egomotion estimation
- Facial-recognition system
- Gesture recognition
- Human-computer interaction
- Mobile robotics
- Motion understanding
- Object identification
- Automated inspection and surveillance
- Segmentation and recognition
- Stereopsis stereo vision – depth perception from two cameras
- Medical image analysis
- Structure from motion
- Motion tracking
- Augmented reality
- Video/image search and retrieval
- Robot and driverless car navigation and control
- Driver drowsiness and distraction detection

Why citing OpenCV in your research work

If you are using OpenCV in your research, it is recommended you cite the OpenCV library. This way, other researchers can better understand your proposed algorithms and reproduce your results for better credibility. Additionally, OpenCV will increase repercussion, resulting in a better computer vision library. The BibTex entry for citing OpenCV is shown in the following code:

```
@article{opencv_library,
 author = {Bradski, G.},
 citeulike-article-id = {2236121},
 journal = {Dr. Dobb's Journal of Software Tools},
 keywords = {bibtex-import},
 posted-at = {2008-01-15 19:21:54},
 priority = {4},
 title = {{The OpenCV Library}},
 year = {2000}
}
```

Installing OpenCV, Python, and other packages

OpenCV, Python, and AI-related packages can be installed on most operating systems. We will see how to install these packages by means of different approaches.

Make sure you check out the different installation options before choosing the one that best suits your needs.

Additionally, at the end of this chapter, an introduction to Jupyter Notebook is given due to the popularity of these documents, which can be run to perform data analysis.

Installing Python, OpenCV, and other packages globally

In this section, you will see how to install Python, OpenCV, and any other package globally. Specific instructions are given for both Linux and Windows operating systems.

Installing Python

We are going to see how to install Python globally on both the Linux and Windows operating systems.

Installing Python on Linux

On Debian derivatives such as Ubuntu, use APT to install Python. Afterwards, it is recommended to upgrade the pip version. pip (`https://pip.pypa.io/en/stable/`) is the PyPA (`https://packaging.python.org/guides/tool-recommendations/`) recommended tool for installing Python packages:

```
$ sudo apt-get install python3.7

$ sudo pip install --upgrade pip
```

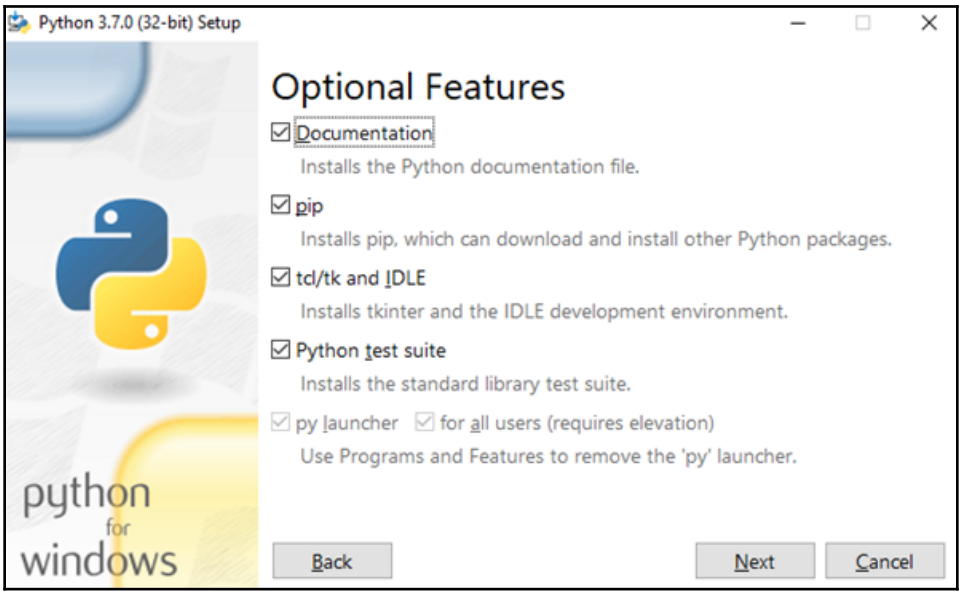
To verify that Python has been installed correctly, open a Command Prompt or shell and run the following command:

```
$ python3 --version
 Python 3.7.0
```

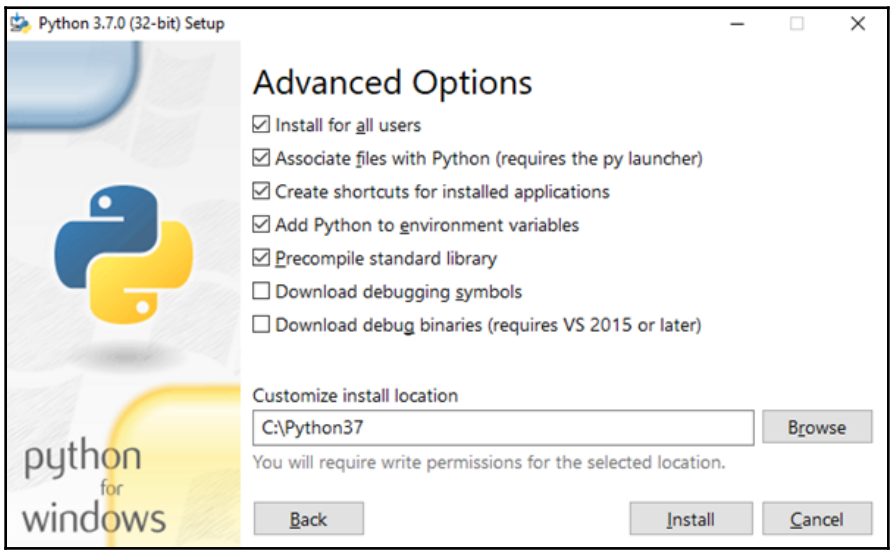
Installing Python on Windows

Go to `https://www.python.org/downloads/`. The default Python Windows installer is 32 bits. Start the installer. Select **Customize installation**:

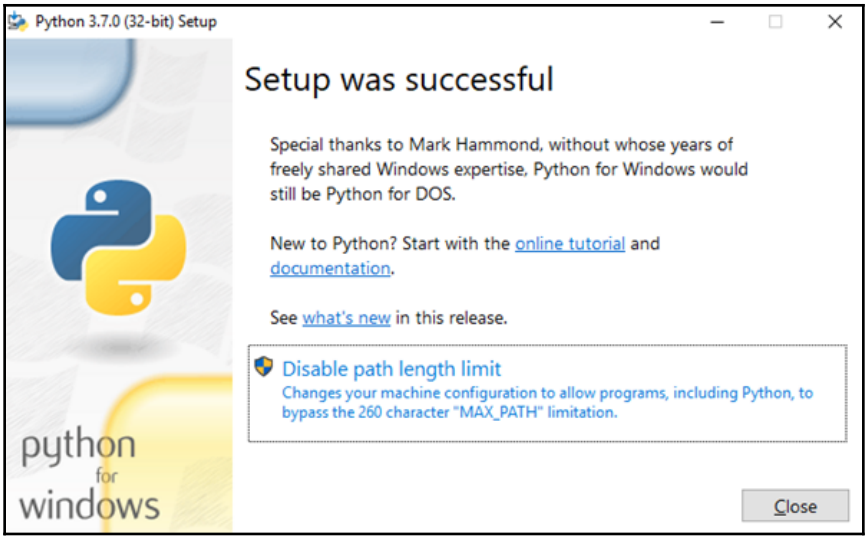
On the next screen, all the optional features should be checked:



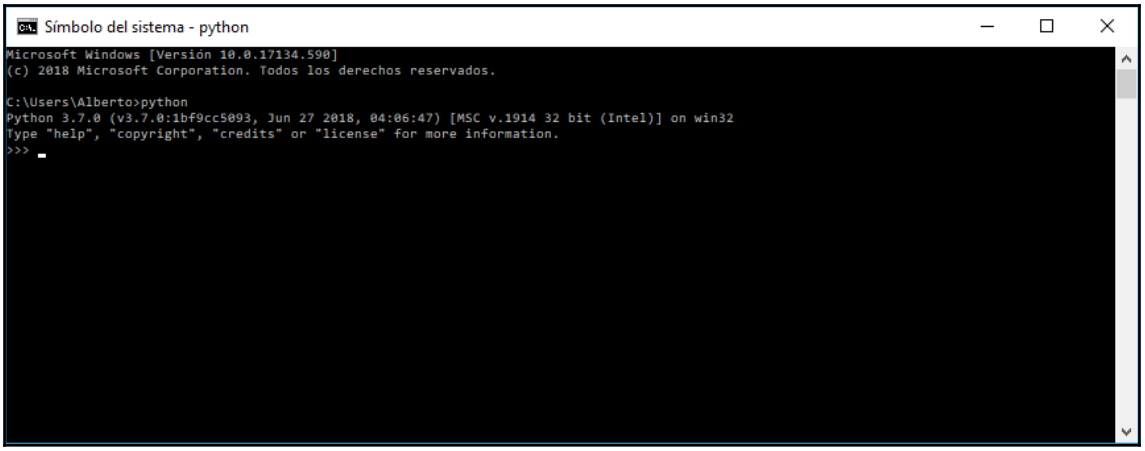
Finally, on the next screen, make sure to check **Add Python to environment variables** and **Precompile standard library**. Optionally, you can customize the location of the installation, for example, `C:\Python37`:



Press the **Install** button and, in a few minutes, the installation should be ready. On the last page of the installer, you should also press **Disable path length limit**:



To check whether Python has been installed properly, press and hold the *Shift* key and right-click with your mouse somewhere on your desktop. Select **Open command window here**. Alternatively, on Windows 10, use the lower-left search box to search for `cmd`. Now, write `python` in the command window and press the *Enter* key. You should see something like this:



You should also upgrade pip:

```
$ python –m pip install ––upgrade pip
```

Installing OpenCV

Now, we are going to install OpenCV on both the Linux and Windows operating systems. First, we are going to see how to install OpenCV on Linux, and then how to install OpenCV on Windows.

Installing OpenCV on Linux

Ensure you have installed NumPy. To install NumPy, enter the following:

```
$ pip3 install numpy
```

Then install OpenCV:

```
$ pip3 install opencv–contrib–python
```

Additionally, we can install Matplotlib, which is a Python plotting library that produces quality figures:

```
$ pip3 install matplotlib
```

Installing OpenCV on Windows

Ensure you have installed NumPy. To install NumPy, enter the following:

```
$ pip install numpy
```

Then install OpenCV:

```
$ pip install opencv–contrib–python
```

Additionally, we can install Matplotlib:

```
$ pip install matplotlib
```

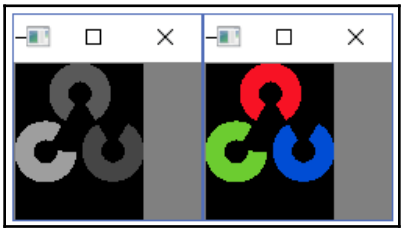
Testing the installation

One way to test the installation is to execute an OpenCV Python script. In order to do it, you should have two files, `logo.png` and `test_opencv_installation.py`, in a specific folder:

Open a cmd and go to the path where these two files are. Next, we can check the installation by typing the following:

```
python test_opencv_installation.py
```

You should see both the OpenCV RGB logo and the OpenCV grayscale logo:



In that case, the installation has been successful.

Installing Python, OpenCV, and other packages with virtualenv

`virtualenv` (`https://pypi.org/project/virtualenv/`) is a very popular tool that creates isolated Python environments for Python libraries. `virtualenv` allows multiple Python projects that have different (and sometimes conflicting) requirements. In a technical way, `virtualenv` works by installing some files under a directory (for example, `env/`).

Additionally, `virtualenv` modifies the PATH environment variable to prefix it with a custom bin directory (for example, `env/bin/`). Additionally, an exact copy of the Python or Python3 binary is placed in this directory. Once this virtual environment is activated, you can install packages in the virtual environment using pip. `virtualenv` is also recommended by the PyPA (`https://packaging.python.org/guides/tool-recommendations/`). Therefore, we will see how to install OpenCV or any other packages using virtual environments.

Usually, pip and `virtualenv` are the only two packages you need to install globally. This is because, once you have installed both packages, you can do all your work inside a virtual environment. In fact, `virtualenv` is really all you need, because this package provides a copy of pip, which gets copied into every new environment you create.

Now, we will see how to install, activate, use, and deactivate virtual environments. Specific commands are given now for both Linux and Windows operating systems. We are not going to add a specific section for each of the operating systems, because the process is very similar in each one. Let's start installing `virtualenv`:

```
$ pip install virtualenv
```

Inside this directory (`env`), some files and folders are created with all you need to run your python applications. For example, the new python executable will be located at `/env/scripts/python.exe`. The next step is to create a new virtual environment. First, change the directory into the root of the project directory. The second step is to use the `virtualenv` command-line tool to create the environment:

```
$ virtualenv env
```

Here, `env` is the name of the directory you want to create your virtual environment inside. It is a common convention to call the directory you want to create your virtual environment inside `env`, and to put it inside your project directory. This way, if you keep your code at `~/code/myproject/`, the environment will be at `~/code/myproject/env/`.

The next step is to activate the `env` environment that you have just created using the command-line tool to execute the `activate` script, which is in the following location:

- `~/code/myprojectname/env/bin/activate` (Linux)
- `~/code/myprojectname/env/Scripts/activate` (Windows)

For example, under Windows, you should type the following:

```
$ ~/code/myprojectname/env/Scripts/activate
 (env) $
```

Now you can install the required packages only for this activated environment. For example, if you want to install Django, which is a free and open source web framework, written in Python, you should type this:

```
(env)$ pip install Django
```

Remember that this package will only be installed for the `myprojectname` project.

You can also deactivate the environment by executing the following:

```
$ deactivate
$
```

You should see that you have returned to your normal prompt, indicating that you are no longer in any `virtualenv`. Finally, if you want to delete your environment, just type the following:

```
$ rmvirtualenv test
```

Python IDEs to create virtual environments with virtualenv

In the next section, we are going to create virtual environments with PyCharm, which is a Python IDE. But before doing that, we are going to discuss IDEs. An IDE is a software application that facilitates computer programmers with software development. IDEs present a single program where all the development is done. In connection with Python IDEs, two approaches can be found:

- General editors and IDEs with Python support
- Python-specific editors and IDEs

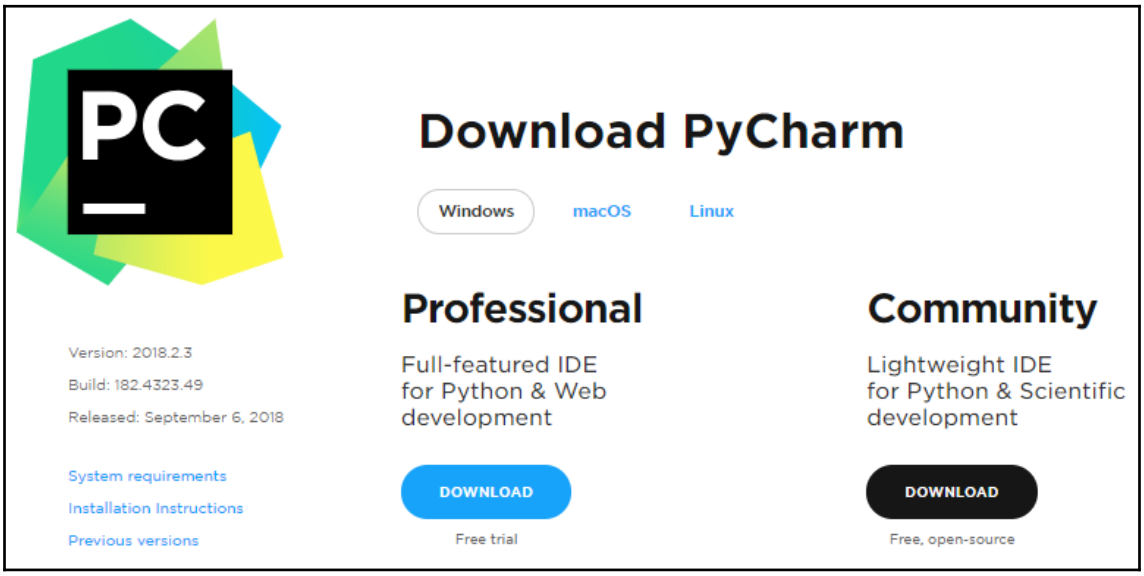
In the first category (general IDEs), some examples should be highlighted:

- Eclipse + PyDev
- Visual Studio + Python Tools for Visual Studio
- Atom + Python extension

In the second category, here are some Python-specific IDEs:

- **PyCharm**: One of the best full-featured, dedicated IDEs for Python. PyCharm installs quickly and easily on Windows, macOS, and Linux platforms. It is the *de facto* Python IDE environment.
- **Spyder**: Spyder, which comes with the Anaconda package manager distribution, is an open source Python IDE that is highly suited for data science workflows.
- **Thonny**: Thonny is intended to be an IDE for beginners. It is available for all major platforms (Windows, macOS, Linux), with installation instructions on the site.

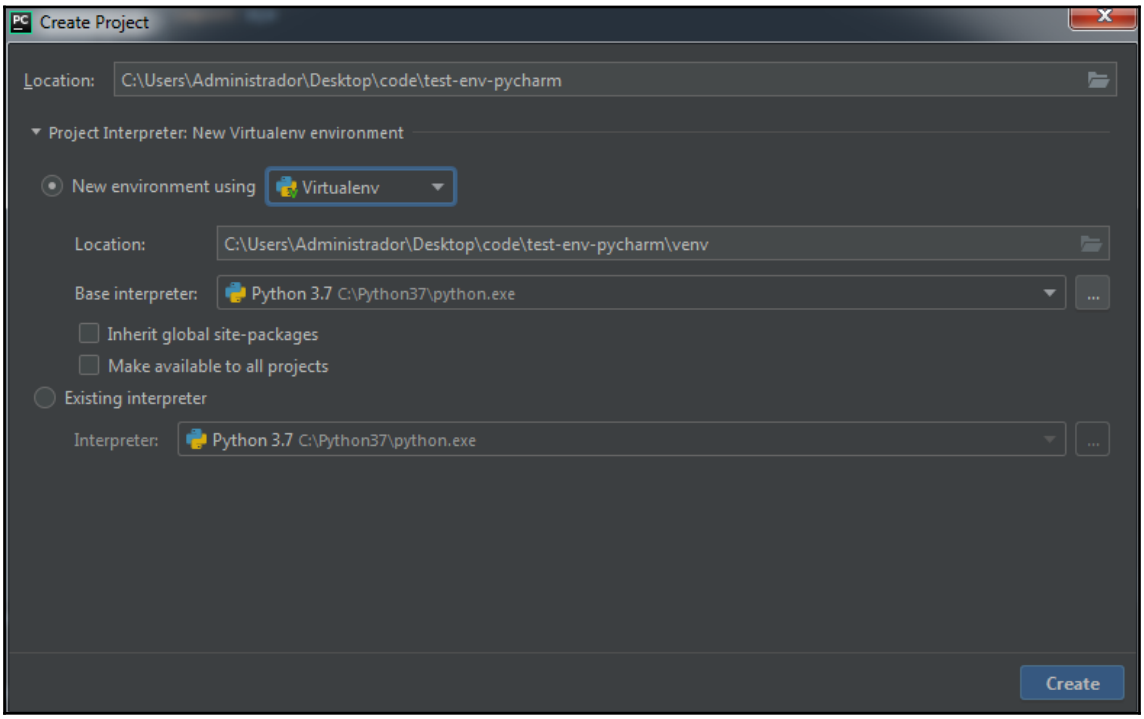
In this case, we are going to install PyCharm (the *de facto* Python IDE environment) Community Edition. Afterwards, we are going to see how to create virtual environments using this IDE. PyCharm can be downloaded from `https://www.jetbrains.com/pycharm/`. PyCharm can be installed on Windows, macOS, and Linux:



After the installation of PyCharm, we are ready to use it. Using PyCharm, we can create virtual environments in a very simple and intuitive way.

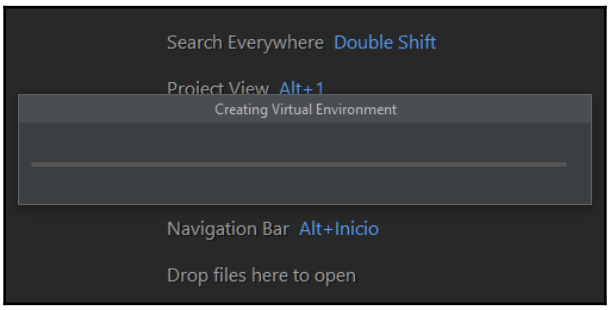
PyCharm makes it possible to use the `virtualenv` tool to create a project-specific isolated virtual environment. Additionally, the `virtualenv` tool comes bundled with PyCharm, so the user does not need to install it.

After opening Pycharm, you can click **Create New Project**. If you want to create a new environment, you should click on **Project Interpreter: New Virtualenv environment**. Then click on **New environment using Virtualenv**. This can be seen in the next screenshot:

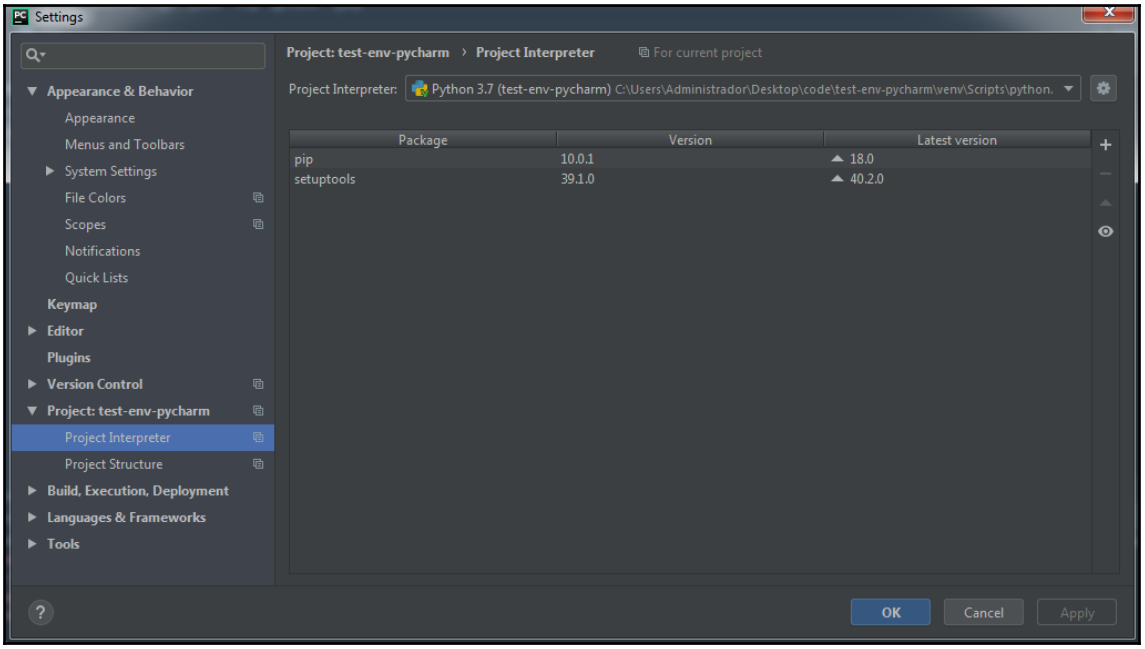


You should note that the virtual environment is named (by default in PyCharm) `venv` and located under the project folder. In this case, the project is named `test-env-pycharm` and the virtual environment, `venv`, is located at `test-env-pycharm/venv`. Additionally, you can see that the `venv` name can be changed according to your preferences.

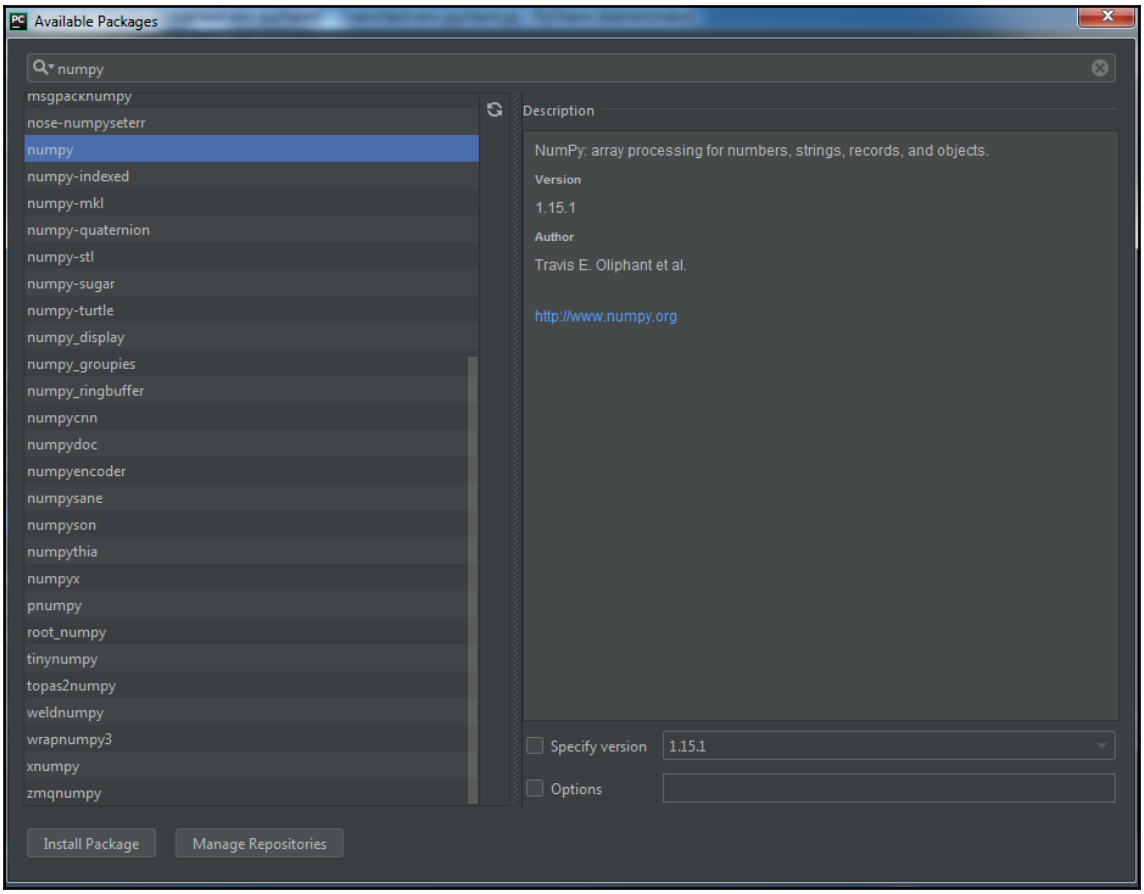
When you click on the **Create** button, PyCharm loads the project and creates the virtual environment. You should see something like this:



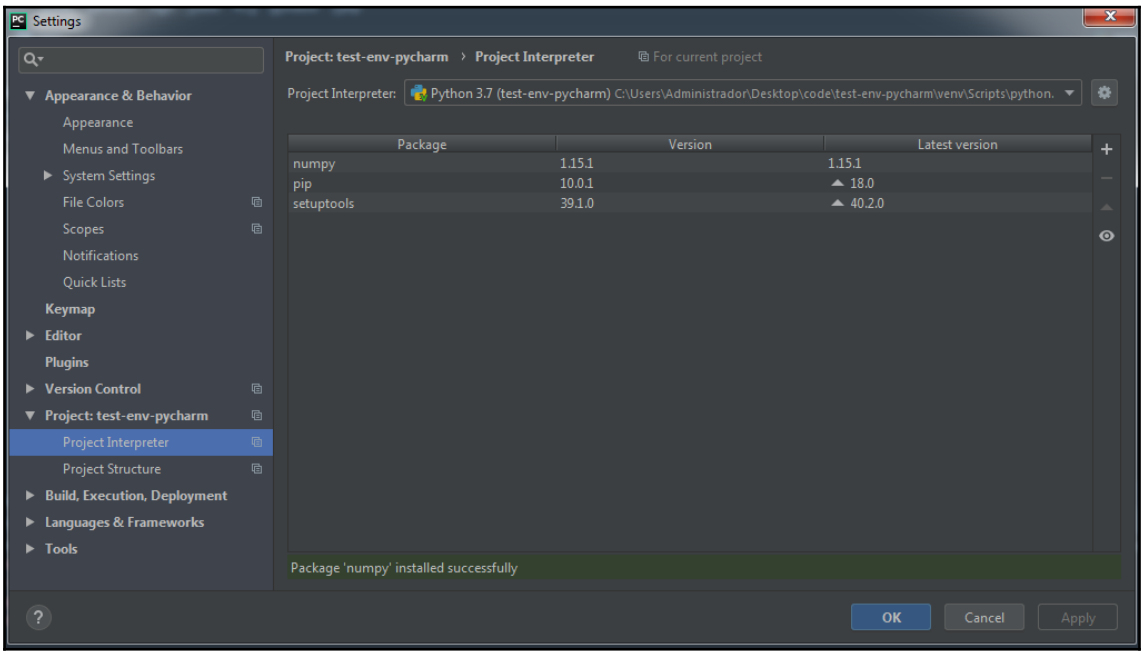
After the project is created, you are ready to install a package with just a few clicks. Click on **File**, then click on **Settings...** (*Ctrl* + *Alt* + *S*). A new window will appear, showing something like this:



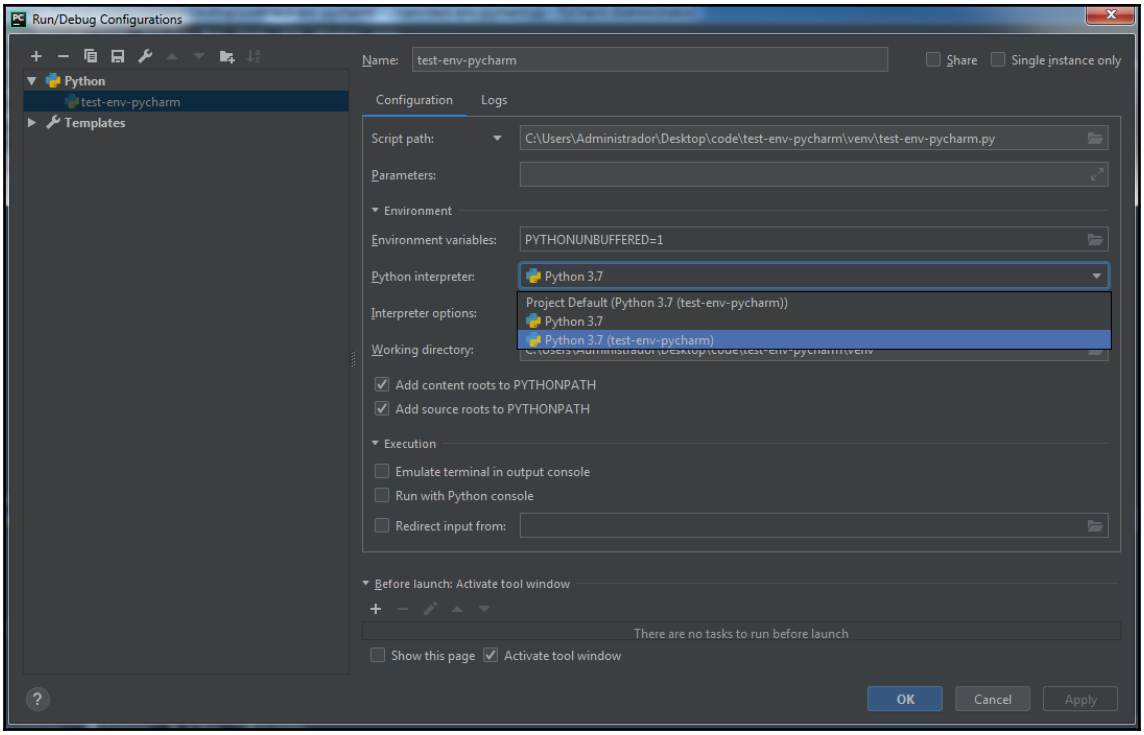
Now, click on **Project:** and select **Project Interpreter**. On the right-hand side of this screen, the installed packages are shown in connection with the selected **Project Interpreter**. You can change it on top of this screen. After selecting the appropriate interpreter (and, hence, the environment for your project), you can install a new package. To do so, you can search in the upper-left input box. In the next screenshot, you can see an example of searching for the `numpy` package:



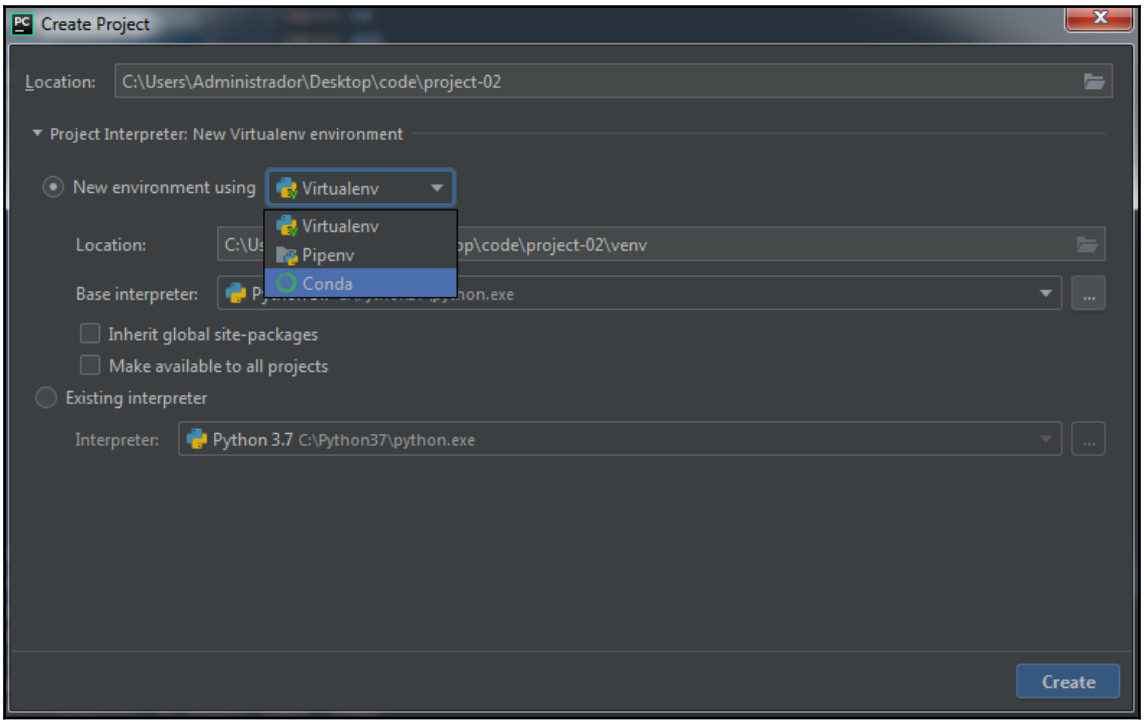
You can install the package (latest version by default) by clicking on **Install Package**. You can also specify a concrete version, as can be seen in the previous screenshot:



After the installation of this package, we can see that we now have three installed packages on our virtual environment. Additionally, it is very easy to change between environments. You should go to **Run/Debug Configurations** and click on **Python interpreter** to change between environments. This feature can be seen in the next screenshot:



Finally, you may have noticed that, in the first step, of creating a virtual environment with PyCharm, options other than `virtualenv` are possible. PyCharm gives you the ability to create virtual environments using **Virtualenv**, **Pipenv**, and **Conda**:



We previously introduced **Virtualenv** and how to work with this tool for creating isolated Python environments for Python libraries.

Pyenv (`https://github.com/pyenv/pyenv`) is used to isolate Python versions. For example, you may want to test your code against Python 2.6, 2.7, 3.3, 3.4, and 3.5, so you will need a way to switch between them.

Conda (`https://conda.io/docs/`) is an open source package management and environment management system (provides virtual environment capabilities) that runs on Windows, macOS, and Linux. Conda is included in all versions of Anaconda and Miniconda.

Since working with Anaconda/Miniconda and Conda may be of interest to readers, a quick introduction is given in the next subsection, but it is not necessary to run the code examples included in this book.

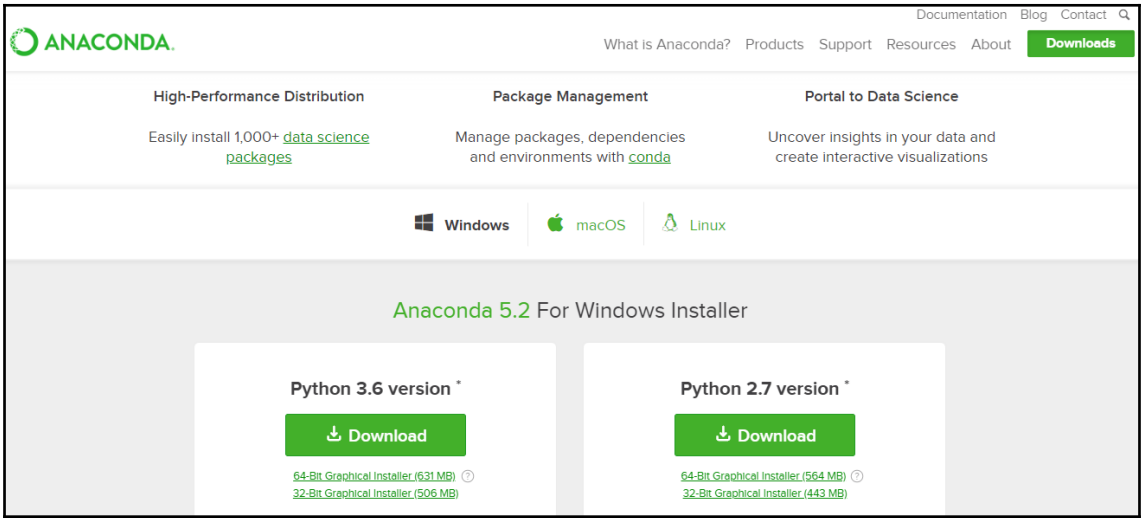
Anaconda/Miniconda distributions and conda package–and environment-management system

Conda (`https://conda.io/docs/`) is an open source package-management and environment-management system (provides virtual environment capabilities) that runs on many operating systems (for example, Windows, macOS, and Linux). Conda installs, runs, and updates packages and their dependencies. Conda can create, save, load, and switch between environments.

As conda is included in all versions of Anaconda and Miniconda, you should have already installed Anaconda or Miniconda.

Anaconda is a downloadable, free, open source, high-performance Python and R distribution. Anaconda comes with conda, conda build, Python, and more than 100 open source scientific packages and their dependencies. Using the conda install command, you can easily install popular open source packages for data science from the Anaconda repository. Miniconda is a small version of Anaconda, which includes only conda, Python, the packages they depend on, and a small number of other useful packages.

Installing Anaconda or Miniconda is easy. For the sake of simplicity, we are focusing on Anaconda. To install Anaconda, check the Acadonda installer for your operating system (`https://www.anaconda.com/download/`). Anaconda 5.2 can be installed in both Python 3.6 and Python 2.7 versions on Windows, macOS, and Linux:



After you have finished installing, in order to test the installation, in Terminal or Anaconda Prompt, run the following command:

```
$ conda list
```

For a successful installation, a list of installed packages appears. As mentioned, Anaconda (and Miniconda) comes with conda, which is a simple package manager similar to `apt-get` on Linux. In this way, we can install new packages in Terminal using the following command:

```
$ conda install packagename
```

Here, `packagename` is the actual name of the package we want to install. Existing packages can be updated using the following command:

```
$ conda update packagename
```

We can also search for packages using the following command:

```
$ anaconda search –t conda packagename
```

This will bring up a whole list of packages available through individual users.
A package called `packagename` from a user called username can then be installed as follows:

```
$ conda install –c username packagename
```

Additionally, conda can be used to create and manage virtual environments. For example, creating a `test` environment and installing NumPy version 1.7 is as simple as typing the next command:

```
$ conda create --name test numpy=1.7
```

In a similar fashion as working with `virtualenv`, environments can be activated and deactivated. To do this on macOS and Linux, just run the following:

```
$ source activate test
 $ python
 ...
 $ source deactivate
```

On Windows, run the following:

```
$ activate test
 $ python
 ...
 $ deactivate
```

See the conda cheat sheet PDF (1 MB) for a single-page summary of the most important information about using conda: `https://conda.io/docs/_downloads/conda-cheatsheet.pdf`.

Finally, it should be pointed out that we can work with conda under the PyCharm IDE, in a similar way as `virtualenv` to create and manage virtual environments, because PyCharm can work with both tools.

Packages for scientific computing, data science, machine learning, deep learning, and computer vision

So far, we have seen how to install Python, OpenCV, and a few other packages (`numpy` and `matplotlib`) from scratch, or using Anaconda distribution, which includes many popular data-science packages. In this way, some knowledge about the main packages for scientific computing, data science, machine learning, and computer vision is a key point because they offer powerful computational tools. Throughout this book, many Python packages will be used. Not all of the cited packages in this section will, but a comprehensive list is provided for the sake of completeness in order to show the potential of Python in topics related to the content of this book:

- **NumPy** (`http://www.numpy.org/`) provides support for large, multi-dimensional arrays. NumPy is a key library in computer vision because images can be represented as multi-dimensional arrays. Representing images as NumPy arrays has many advantages.
- **OpenCV** (`https://opencv.org/`) is an open source computer vision library.
- **Scikit-image** (`https://scikit-image.org/`) is a collection of algorithms for image processing. Images manipulated by scikit-image are simply NumPy arrays.
- The **Python Imaging Library** (**PIL**) (`http://www.pythonware.com/products/pil/`) is an image-processing library that provides powerful image-processing and graphics capabilities.
- **Pillow** (`https://pillow.readthedocs.io/`) is the friendly PIL fork by Alex Clark and contributors. The PIL adds image-processing capabilities to your Python interpreter.
- **SimpleCV** (`http://simplecv.org/`) is a framework for computer vision that provides key functionalities to deal with image processing.
- **Mahotas** (`https://mahotas.readthedocs.io/`) is a set of functions for image processing and computer vision in Python. It was originally designed for bioimage informatics. However, it is useful in other areas as well. It is completely based on numpy arrays as its datatype.
- **Ilastik** (`http://ilastik.org/`) is a user-friendly and simple tool for interactive image segmentation, classification, and analysis.
- **Scikit-learn** (`http://scikit-learn.org/`) is a machine learning library that features various classification, regression, and clustering algorithms.

- **SciPy** (`https://www.scipy.org/`) is a library for scientific and technical computing.
- **NLTK** (`https://www.nltk.org/`) is a suite of libraries and programs to work with human-language data.
- **spaCy** (`https://spacy.io/`) is an open-source software library for advanced natural language processing in Python.
- **LibROSA** (`https://librosa.github.io/librosa/`) is a library for both music and audio processing.
- **Pandas** (`https://pandas.pydata.org/`) is a library (built on top of NumPy) that provides high-level data computation tools and easy-to-use data structures.
- **Matplotlib** (`https://matplotlib.org/`) is a plotting library that produces publication-quality figures in a variety of formats.
- **Seaborn** (`https://seaborn.pydata.org/`) is a graphics library that is built on top of Matplotlib.
- **Orange** (`https://orange.biolab.si/`) is an open source machine learning and data-visualization toolkit for novices and experts.
- **PyBrain** (`http://pybrain.org/`) is a machine learning library that provides easy-to-use state-of-the-art algorithms for machine learning.
- **Milk** (`http://luispedro.org/software/milk/`) is a machine learning toolkit focused on supervised classification with several classifiers.
- **TensorFlow** (`https://www.tensorflow.org/`) is an open source machine learning and deep learning library.
- **PyTorch** (`https://pytorch.org/`) is an open source machine learning and deep learning library.
- **Theano** (`http://deeplearning.net/software/theano/`) is a library for fast mathematical expressions, evaluation, and computation, which has been compiled to run on both CPU and GPU architectures (a key point for deep learning).
- **Keras** (`https://keras.io/`) is a high-level deep learning library that can run on top of TensorFlow, CNTK, Theano, or Microsoft Cognitive Toolkit.
- **Django** (`https://www.djangoproject.com/`) is a Python-based free and open source web framework that encourages rapid development and clean, pragmatic design.
- **Flask** (`http://flask.pocoo.org/`) is a micro web framework written in Python based on Werkzeug and Jinja 2.

All these packages can be organized based on their main purpose:

- **To work with images**: NumPy, OpenCV, scikit-image, PIL Pillow, SimpleCV, Mahotas, ilastik
- **To work in text**: NLTK, spaCy, NumPy, scikit-learn, PyTorch
- **To work in audio**: LibROSA
- **To solve machine learning problem**: pandas, scikit-learn, Orange, PyBrain, Milk
- **To see data clearly**: Matplotlib, Seaborn, scikit-learn, Orange
- **To use deep learning**: TensorFlow, Pytorch, Theano, Keras
- **To do scientific computing**: SciPy
- **To integrate web applications**: Django, Flask

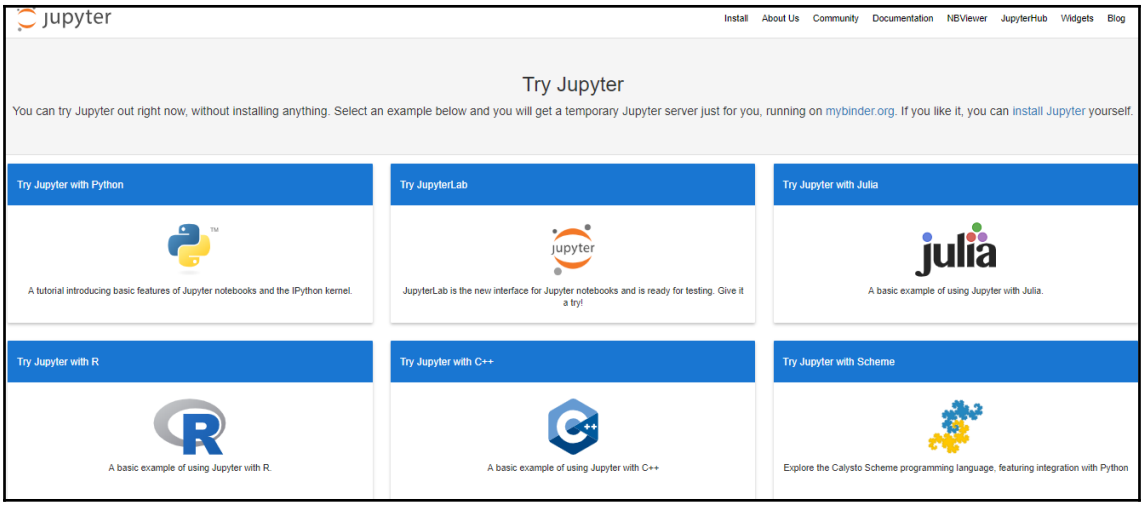
Additional Python libraries and packages for AI and machine learning can be found at `https://python.libhunt.com/packages/artificial-intelligence`.

Jupyter Notebook

The Jupyter Notebook an open source web application that allows you to edit and run documents via a web browser. These documents, which are called Notebook documents (or notebooks), contain code (more than 40 programming languages, including Python, are supported) and rich text elements (paragraphs, equations, figures). The Jupyter Notebook can be executed on a local computer or can be installed on a remote server. You can start with notebooks, trying them online or installing the Jupyter Notebook.

Trying Jupiter Notebook online

First, go to `https://jupyter.org/try`. You will see something like this:



To try Jupyter with Python online, click on the Python option, or paste this URL into your web browser: `https://mybinder.org/v2/gh/ipython/ipython-in-depth/master?filepath=binder/Index.ipynb`. Once the page has loaded, you can start coding/loading notebooks.

Installing the Jupyter Notebook

To install Jupyter, you can follow the main steps at `http://jupyter.org/install.html`. Installing Jupyter Notebook can also be done using Anaconda or using Python's package manager, pip.

Installing Jupyter using Anaconda

It is strongly recommended you install Python and Jupyter using the Anaconda Distribution, which includes Python, the Jupyter Notebook, and other commonly used packages for scientific computing and data science. To install Jupyter using Anaconda, download Anaconda (`https://www.anaconda.com/distribution/`) and install it. This way, you have installed Jupyter Notebook. To run the notebook, run the following command in Command Prompt (Windows) or Terminal (macOS/Linux):

```
$ jupyter notebook
```

Installing Jupyter with pip

You can also install Jupyter using Python's package manager, pip, by running the following commands:

```
$ python -m pip install --upgrade pip
$ python -m pip install jupyter
```

At this point, you can start the notebook server by running the following command:

```
$ jupyter notebook
```

The previous command will show you some key information in connection with the notebook server, including the URL of the web application (which by default is `http://localhost:8888`). It will then open your default web browser to this URL. To start a specific notebook, the following command should be used:

```
$ jupyter notebook notebook.ipynb
```

This was a quick introduction to notebooks. In the next chapters, we are going to create some notebooks, so you will have the opportunity to play with them and gain full knowledge of this useful tool.

The OpenCV and Python project structure

The **project structure** is the way you organize all the files inside a folder in a way that the project best accomplishes the objectives. We are going to start with a `.py` script (`sampleproject.py`) that should be with other files in order to complete the information about this script – dependencies, license, how to install it, or how to test it. A common approach for structuring this basic project is as follows:

```
sampleproject/
│
├── .gitignore
├── sampleproject.py
├── LICENSE
├── README.rst
├── requirements.txt
├── setup.py
└── tests.py
```

`sampleproject.py`—if your project is only a single Python source file, then put it into the directory and name it something related to your project.

The `README` (`.rst` or `.md` extension) is used to register the main properties of the project, which should cover, at least, the following:

- What your project does
- How to install it
- Example usage
- How to set up the dev environment
- How to ship a change
- Change log
- License and author info

A template you can use can be downloaded from this GitHub repository: `https://github.com/dbader/readme-template`. For further information, please see `https://dbader.org/blog/write-a-great-readme-for-your-github-project`.

The `LICENSE.md` document contains the applicable license. This is arguably the most important part of your repository, aside from the source code itself. The full license text and copyright claims should exist in this file. It is always a good idea to have one if you are distributing code. Typically, the **GNU General Public License** (**GPL**) (`http://www.gnu.org/licenses/gpl.html`) or the MIT license (`https://opensource.org/licenses/MIT`) are used in open source projects. You can check out `http://choosealicense.com/` if you are not sure which license should be applied to your project.

A `requirements.txt` pip requirements file (`https://pip.pypa.io/en/stable/user_guide/#requirements-files`) should be placed at the root of the repository, which is used to specify the dependencies required to contribute to the project. The `requirements.txt` file can be generated using the following:

```
$ pip freeze > requirements.txt
```

To install these requirements, you can use the following command:

```
$ pip install -r requirements.txt
```

The `setup.py` file allows you to create packages you can redistribute. This script is meant to install your package on the end user's system, not to prepare the development environment as `pip install -r < requirements.txt` does. It is a key file because it defines information of your package (such as versioning, package requirements, and the project description).

The `tests.py` script contains the tests.

The `.gitignore` file tells Git what kind of files to ignore, such as IDE clutter or local configuration files. You can find sample `.gitignore` files for Python projects at `https://github.com/github/gitignore`.

Our first Python and OpenCV project

Based on the minimal project structure that was shown in the previous section, we are going to create our first Python and OpenCV project. This project has the following structure:

```
helloopencv/
|
├── images/
|
├── .gitignore
├── helloopencv.py
├── LICENSE
├── README.rst
├── requirements.txt
├── setup.py
└── helloopencvtests.py
```

`README.rst` (`.rst` extension) follows a basic structure, which was shown in the previous section. Python and **ReStructuredText** (**RST**) are deeply linked—RST is the format of docutils and sphinx (the *de facto* standard for documenting python code). RST is used both to document objects via docstrings, and to write additional documentation. If you go to the official Python documentation (`https://docs.python.org/3/library/stdtypes.html`), you can view the RST source of each page (`https://raw.githubusercontent.com/python/cpython/3.6/Doc/library/stdtypes.rst`). Using RST for the `README.rst` makes it directly compatible with the entire documentation setup. In fact, `README.rst` is often the cover page of a project's documentation.

There are some RST editors you can use to help you write the `README.rst`. You can also use some online editors. For example, the online Sphinx editor is a good choice (`https://livesphinx.herokuapp.com/`).

The `.gitignore` file specifies intentionally untracked files that Git should ignore (`https://git-scm.com/docs/gitignore`). `.gitignore` tells `git` which files (or patterns) Git should ignore. It is usually used to avoid committing transient files from your working directory that are not useful to other collaborators, such as compilation products and temporary files that IDEs create. Open `https://github.com/github/gitignore/blob/master/Python.gitignore` to see a `.gitignore` file that you can include in your Python projects.

`setup.py` (see the previous section for a deeper description), it is a Python file that is usually shipped with libraries or programs, also written in Python. Its purpose is to correctly install the software. A very complete example of this file can be seen at `https://github.com/pypa/sampleproject/blob/master/setup.py`, which is full of comments to help you understand how to adapt it to your needs. This file is proposed by the **Python Packaging Authority** (**PyPa**) (`https://www.pypa.io/en/latest/`). One key point is in connection with the *packages* option, as we can read in the aforementioned `setup.py` file.

You can just specify package directories manually here if your project is simple. Or you can use `find_packages()`. Alternatively, if you just want to distribute a single Python file, use the `py_modules` argument instead as follows, which will expect a file called `my_module.py` to exist, `py_modules=["my_module"]`.

Therefore, in our case, `py_modules =["helloopencv"]` is used.

Additionally, `setup.py` allows you to easily install Python packages. Often it is enough to write the following:

```
$ python setup.py install
```

Therefore, if we want to install this simple package, we can write the previous command, `python setup.py install`, inside the `helloopencv` folder. For example, in Windows, run the following command:

```
C:\...\helloopencv>python setup.py install
```

You should see something like this:

```
running install
...
...
Installed c:\python37\lib\site-packages\helloopencv-0.1-py3.7.egg
Processing dependencies for helloopencv==0.1
...
...
Finished processing dependencies for helloopencv==0.1
```

When finished, `helloopencv` is installed in our system (like any other Python package). You can also install `helloopencv` with pip, `pip install .`, inside the `helloopencv` folder. For example, in Windows, run the following command:

```
C:\...\helloopencv>pip install .
```

You should see something like this:

```
Processing c:\...\helloopencv
...
...
Successfully installed helloopencv-0.1
```

This indicates that `helloopencv` has been installed successfully. To use this package, we can write a Python file and import the `helloopencv` package. Alternatively, we can perform a quick use of this package by importing it directly from the Python interpreter. Following this second approach, you can open Command Prompt, import the package, and make use of it. First, open Command Prompt, then type `python` to run the interpreter:

```
C:\...\helloopencv>python
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Once the interpreter is loaded, we can import the package:

```
>>> import helloopencv
helloopencv.py is being imported into another module
>>>
```

The `helloopencv.py is being imported into another module` output is a message from the `helloopencv` package (specifically from the `helloopencv.py` file) indicating that this file has been imported. Therefore, this message indicates that the module has been successfully imported. Once imported, we can make use of it. For example, we can call the `show_message` method:

```
>>> helloopencv.show_message()
'this function returns a message'
>>>
```

We can see that the result of calling this method is a message that is printed on the screen. This method is a simple way to know that everything has been installed correctly because it involves installing, importing, and making use of a function from the package. Furthermore, we can call a more useful method contained in the `helloopencv` package. You can, for example, call the `load_image` method to load an image from disk and afterwards, you can display it using the `show_image` method:

```
>>> image = helloopencv.load_image("C:/.../images/logo.png")
>>> helloopencv.show_image(image)
```

Here, the parameter of the `load_image` function is the path of an image from your computer. In this case, the `logo.png` image is loaded. After calling the `show_image` method, an image should be displayed. To close the window, a key must be pressed. Then you should be able to write in the interpreter again. To see all the methods that are available in the `helloopencv` package, you can open the `helloopencv.py` file with your favorite editor or IDE and have a look at it. In this Python file, you can see some methods that conform to our first Python project:

- `show_message()`: This function prints the `this function returns a message` message.
- `load_image()`: This function loads an image from its path.
- `show_image()`: This function shows an image once it has been loaded.
- `convert_to_grayscale()`: This function converts an image to grayscale once it has been loaded.
- `write_image_to_disk()`: This function saves an image on disk.

All of these methods perform simple and basic operations. Most of them make use of the OpenCV library, which is imported at the beginning of this file (`import cv2`). Do not worry about the Python code contained in this file, because only basic operations and calls to the OpenCV library are performed.

You can execute the `helloopencv.py` script without installing the package. To execute this file, you should run the `python helloopencv.py` command once Command Prompt is opened:

```
C:\...\helloopencv>python helloopencv.py
helloopencv.py is being run directly
```

After the execution of this file, the `helloopencv.py is being run directly` message will appear, which means that the file is executed directly and not imported from another module or package (or the Python interpreter). You can also see that an image is loaded and displayed. You can press any key to continue the execution. Once again, the grayscale version of the logo is displayed and any key should be pressed again to end the execution. The execution ends after the grayscale image is saved on the disk.

Lastly, the `helloopencvtests.py` file can be used for unit testing. Testing applications has become a standard skill for any competent developer. The Python community embraces testing, and the Python standard library has good built-in tools to support testing (`https://docs.python.org/3/library/unittest.html`).

In the Python ecosystem, there are a lot of testing tools. The two most common packages used for testing are `nose` (`https://pypi.org/project/nose/`) and `pytest` (`https://pypi.org/project/pytest/`). In this first Python project, we are going to use `pytest` for unit testing.

To execute the test, run the `py.test -s -v helloopencvtests.py` command once Command Prompt is opened:

```
C:\...\helloopencv>py.test -s -v helloopencvtests.py
============================= test session starts
============================
 platform win32 -- Python 3.7.0, pytest-3.8.0, py-1.6.0, pluggy-0.7.1 --
c:\python37\python.exe
 cachedir: .pytest_cache
 collected 4 items
 helloopencvtests.py::test_show_message testing show_message
 PASSED
 helloopencvtests.py::test_load_image testing load_image
 PASSED
 helloopencvtests.py::test_write_image_to_disk testing
 write_image_to_disk
 PASSED
 helloopencvtests.py::test_convert_to_grayscale testing
test_convert_to_grayscale
 PASSED
========================== 4 passed in 0.57 seconds
===========================
```

After the execution of the tests, you can see that four tests were executed. The `PASSED` message means that the tests were executed successfully. This is a quick introduction to unit testing in Python. Nevertheless, the full `pytest` documentation can be found at `https://docs.pytest.org/en/latest/contents.html#toc`.

Summary

In this first chapter, we covered the main steps to set up OpenCV and Python to build your computer vision projects. At the beginning of this chapter, we quickly looked at the main concepts in this book – Artificial Intelligence, machine learning, neural networks, and deep learning. Then we explored the OpenCV library, including the history of the library and its main modules. As OpenCV and other packages can be installed in many operating systems and in different ways, we covered the main approaches.

Specifically, we saw how to install Python, OpenCV, and other packages globally or in a virtual environment. In connection with the installation of the packages, we introduced Anaconda/Miniconda and Conda, because we can also create and manage virtual environments. Additionally, Anaconda/Miniconda comes with many open source scientific packages, including SciPy and NumPy.

We explored the main packages for scientific computing, data science, machine learning, and computer vision, because they offer powerful computational tools. Then we discussed the Python-specific IDEs, including PyCharm (the *de facto* Python IDE environment). PyCharm (and other IDEs) can help us create virtual environments in a very intuitive way. We also looked at Jupyter Notebooks, because it can be a good tool for the readers of this book. In the next chapters, more Jupyter Notebooks will be created to give you a better understanding of this useful tool. Finally, we explored an OpenCV and Python project structure, covering the main files that should be included. Then we built our first Python and OpenCV sample project, where we saw the commands to build, run, and test this project.

In the next chapter, you will start to write your first scripts as you get better acquainted with the OpenCV library. You will see some basic concepts necessary to start coding your computer vision projects (for example, understanding main image concepts, the coordinate system in OpenCV, and accessing and manipulating pixels in OpenCV).

Questions

1. What is a virtual environment?
2. What is the connection between pip, virtualenv, pipenv, Anaconda, and conda?
3. What is the Jupyter Notebook?
4. What are the main packages to work with computer vision in Python?
5. What does `pip install -r requirements.txt` do?
6. What is an IDE and why should you use one during the development of your projects?
7. Under what license is OpenCV published?

Further reading

The following references will help you dive deeper into concepts presented in this chapter:

- Python Machine Learning:
    - *Python Machine Learning*, by Sebastian Raschka: `https://www.packtpub.com/big-data-and-business-intelligence/python-machine-learning`
- Python Deep Learning:
    - *Python Deep Learning Projects*, by Rahul Kumar, Matthew Lamons: `https://www.packtpub.com/big-data-and-business-intelligence/python-deep-learning-projects`

Check out these references (mainly books) for more information on concepts that will be presented in future chapters of the book. Keep this list handy; it will be really helpful:

- *OpenCV Computer Vision with Python* (`https://www.packtpub.com/application-development/opencv-computer-vision-python`)
- *OpenCV: Computer Vision Projects with Python* (`https://www.packtpub.com/application-development/opencv-computer-vision-projects-python`)
- *Augmented Reality for Developers* (`https://www.packtpub.com/web-development/augmented-reality-developers`)
- *Deep Learning with Python and OpenCV* (`https://www.packtpub.com/big-data-and-business-intelligence/deep-learning-python-and-opencv`)
- *Deep Learning with Keras* (`https://www.packtpub.com/big-data-and-business-intelligence/deep-learning-keras`)
- *Getting Started with TensorFlow* (`https://www.packtpub.com/big-data-and-business-intelligence/getting-started-tensorflow`)
- *Mastering Flask Web Development - Second Edition* (`https://www.packtpub.com/web-development/mastering-flask-web-development-second-edition`)

2
Image Basics in OpenCV

Images are a key component in a computer vision project because they provide, in many cases, the input to work with. Therefore, understanding main image concepts is the basic knowledge you need to start coding your computer vision projects. Also, some of the OpenCV library peculiarities, such as the coordinate system or the BGR order (rather than RGB), will be introduced.

In this chapter, you will learn how to start writing your first scripts, which will introduce you to the OpenCV library. At the end of this chapter, you will have enough knowledge to start programming your first computer vision project in OpenCV and Python.

In this chapter, we will cover the following topics:

- A theoretical introduction to image basics
- Concepts of pixel, colors, channels, images, and color spaces
- The coordinate system in OpenCV
- Accessing and manipulating pixels in OpenCV in different color spaces (getting and setting)
- BGR order in OpenCV (rather than RGB)

Technical requirements

The technical requirements for this chapter are listed as follows:

- Python and OpenCV
- A Python-specific IDE
- The NumPy and Matplotlib packages
- Jupyter Notebook
- Git client

Further details about how to install these requirements can be found in `Chapter 1`, *Setting Up OpenCV*. The GitHub repository for Mastering OpenCV with Python, which contains all the supporting project files that are necessary to work through this book from the first chapter to the last, can be accessed at `https://github.com/PacktPublishing/Mastering-OpenCV-4-with-Python`.

A theoretical introduction to image basics

The main purpose of this section is to provide a theoretical introduction to image basics – these will be explained in detail in the next section. First, a quick introduction will be given to underline the importance of some of the difficulties you will encounter when developing the image-processing set in a computer vision project, and then some simple formulation will be introduced in connection with images.

Main problems in image processing

The first concept to introduce is related to images, which can be seen as a **two-dimensional** (**2D**) view of a 3D world. A digital image is a numeric representation, normally binary, of a 2D image as a finite set of digital values, which are called **pixels** (the concept of a pixel will be explained in detail in the *Concepts of pixels, colors, channels, images, and color spaces* section). Therefore, the goal of computer vision is to transform this 2D data into the following:

- A new representation (for example, a new image)
- A decision (for example, perform a concrete task)
- A new result (for example, correct classification of the image)
- Some useful information extraction (for example, object detection)

Computer vision may tackle common problems (or difficulties) when dealing with image-processing techniques:

- Ambiguous images because they are affected by perspective, which can produce changes in the visual appearance of the image. For example, the same object viewed from different perspectives can result in different images.
- Images commonly affected by many factors, such as illumination, weather, reflections, and movements.

- Objects in the image may also be occluded by other objects, making it difficult to detect or classify the occluded ones. Depending on the level of the occlusion, the required task (for example, classification of an image into some predefined categories) can be really challenging.

To put all of these difficulties together, imagine that you want to develop a face-detection system. This system should be robust enough to deal with changes in illumination or weather conditions. Additionally, the system should tackle the movements of the head, and could even deal with the fact that the user can be farther from or closer to the camera. It should be able to detect the head of the user with some degree of rotation in every axis (yaw, roll, and pitch). For example, many face-detection algorithms show good performance when the head is near frontal. However, they fail to detect a face if it's not frontal (for example, a face in profile). Moreover, you may want to detect the face even if the user is wearing glasses or sunglasses, which produces an occlusion in the eye region. When developing a computer vision project, you must take all of these factors into consideration. A good approximation is to have many test images to validate your algorithm by incorporating some difficulties. You can also classify your test images in connection with the main difficulty they have to easily detect the weak points of your algorithm.

Image-processing steps

Image processing includes the following three steps:

1. Get the image to work with. This process usually involves some functions so that you can read the image from different sources (camera, video stream, disk, online resources).
2. Process the image by applying image-processing techniques to achieve the required functionality (for example, detecting a cat in an image).
3. Show the result of the processing step (for example, drawing a bounding box in the image and then saving it to disk).

Furthermore, step two can be broken down into three processing levels:

- Low-level process
- Mid-level process
- High-level process

The **low-level process** usually takes an image as the input and then outputs another image. Example procedures that can be applied in this step include the following:

- Noise removal
- Image sharpening
- Illumination normalization
- Perspective correction

In connection with the face-detection example, the output image can be an illumination normalization image to deal with changes caused by sun reflections.

The **mid-level process** takes the preprocessed image to output some kind of representation of the image. Consider this as a collection of numbers (for example, a vector containing 100 numbers), which summarizes the main information of the image to be used for further processing. In connection with the face-detection example, the output could be a rectangle defined by a point *(x,y)*, the width and the height containing the detected face.

The **high-level process** takes this vector of numbers (usually called **attributes**) and outputs the final result. For example, the input could be the detected face and the output could be the following:

- Face recognition
- Emotion recognition
- Drowsiness and distraction detection
- Remote heart rate measurement from the face

Images formulation

An image can be described as a 2D function, $f(x,y)$, where (x,y) are the spatial coordinates and the value of f at any point, (x,y), is proportional to the brightness or gray levels of the image. Additionally, when both (x,y) and brightness values of f are all finite discrete quantities, the image is called a **digital image**. Therefore, $f(x,y)$ takes the following values:

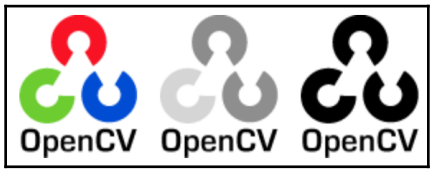
- $x \in [0, h-1]$, where h is the height of the image
- $y \in [0, w-1]$, where w is the width of the image
- $f(x,y) \in [0, L-1]$, where $L = 256$ (for an 8-bit image)

A color image can be represented in the same way, but we need to define three functions to represent the red, green, and blue values, respectively. Each of these three individual functions follows the same formulation as the *f(x,y)* function that was defined for grayscale images. We will denote these three functions subindex R, G and B for the three formulations (for the color images) as *fR(x,y)*, *fG(x,y)*, and *fB(x,y)*.

A black and white image follows the same approximation in the way that only one function is required to represent the image. However, one key point is that *f(x,y)* can only take two values. Usually, these values are `0` (black) and `255` (white).

These three types of images are commonly used in computer vision, so remember their formulation.

The following screenshot shows the three types of images (a color image, a grayscale image, and a black and white image):



Remember that the digital image can be seen as an approximation of the real scene because *f(x,y)* values are finite discrete quantities. Additionally, both grayscale and black and white images have only one sample per point (only one function is needed) and color images have three samples per point (three functions are needed—corresponding to the red, green, and blue components of the image).

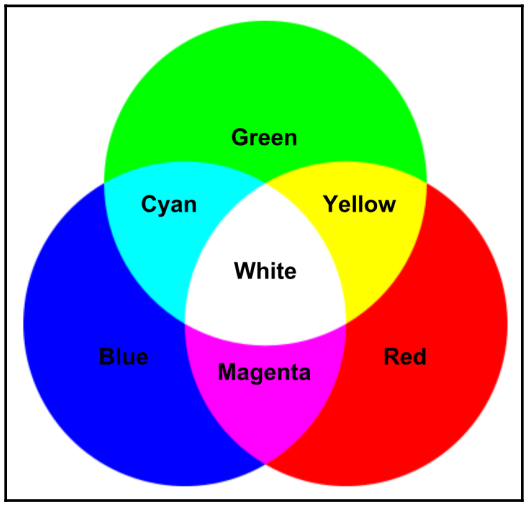
Concepts of pixels, colors, channels, images, and color spaces

There are several different color models, but the most common one is the **Red, Green, Blue** (**RGB**) model, which will be used to explain some key concepts concerning digital images.

In `Chapter 5`, *Image Processing Techniques*, the main color models (also known as **color spaces**) will be fully explained.

The RGB model is an additive color model in which the primary colors *(R, G, B)* are mixed together to reproduce a broad range of colors. As we previously stated, in the RGB model, the primary colors are red, green, and blue.

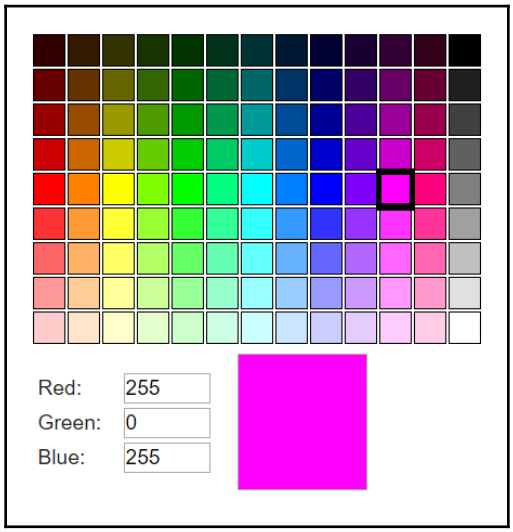
Each primary color, *(R*, G, B*)*, is usually called a channel, which is commonly represented as an integer value in the [0, 255] range. Therefore, each channel produces a total of 256 discrete values, which corresponds to the total number of bits that are used to represent the color channel value ($2^8=256$). Additionally, since there are three different channels, this is called a **24-bit color depth**:



In the previous diagram, you can see the additive color property of the RGB color space:

- Adding red to green gives yellow
- Adding red to blue produces magenta
- Adding green to blue generates cyan
- Adding all three primary colors together produces white

As we previously stated and in connection with the RGB color model, a specific color is represented by red, green, and blue values, expressing the pixel value as an RGB triplet, *(r, g, b)*. A typical RGB color selector in a piece of graphics software is shown as follows. As you can imagine, each slider ranges from `0` to `255`:



You can also see that adding pure red to pure blue produces a perfect magenta color. You can play with the RGB color chart at `https://www.rapidtables.com/web/color/RGB_Color.html`.

An image with a resolution of *800 × 1,200* is a grid with 800 columns and 1,200 rows, containing *800 × 1,200 = 960,000* pixels. It should be noted that knowing how many pixels are in an image does not indicate its physical dimensions (one pixel does not equal one millimeter). Instead, how *large* a pixel is (and hence, how large an image will be) will depend on the **pixels per inch** (**PPI**) that have been set for that image. A general rule of thumb is to have a PPI in the range of `[200 - 400]`.

The basic equation for calculating PPI is as follows:
PPI = width(pixels)/width of image (inches)
PPI = height(pixels)/height of image (inches)
So, for example, if you want to print a *4 × 6* inch image, and your image is *800 × 1,200*, the PPI will be 200.

We will now look into file extensions.

File extensions

Although the images we are going to manipulate in OpenCV can be seen as rectangular arrays of RGB triplets (in the case of RGB images), they are not necessarily created, stored, or transmitted in that format. In this sense, some file formats, such as GIF, PNG, bitmaps, or JPEG, use different forms of compression (lossless or lossy) to represent images more efficiently.

In this way, and for the sake of completeness, a brief introduction to these image files is given here, with a special focus on the file formats supported by OpenCV. The following file formats (with the associated file extensions) are supported by OpenCV:

- **Windows bitmaps**: `*.bmp` and `*.dib`
- **JPEG files**: `*.jpeg`, `*.jpg`, and `*.jpe`
- **JPEG 2000 files**: `*.jp2`
- **Portable Network Graphics**: `*.png`
- **Portable image format**: `*.pbm`, `*.pgm`, and `*.ppm`
- **TIFF files**: `*.tiff` and `*.tif`

The **Bitmap image file** (**BMP**) or **device independent bitmap** (**DIB**) file format is a raster image file format that's used to store bitmap digital images. The BMP file format can deal with 2D digital images in various color depths and, optionally, with data compression, alpha channels, or color profiles.

Joint Photographic Experts Group (**JPEG**) is a raster image file format that's used to store images that have been compressed to store a lot of information in a small file.

JPEG 2000 is an image compression standard and coding system that uses wavelet-based compression techniques offering a high level of scalability and accessibility. In this way, JPEG 2000 compresses images with fewer artefacts than a regular JPEG.

Portable Network Graphics (PNG) is a compressed raster graphics file format, which was introduced in 1994 as an improved replacement for **Graphics Interchange Format** (**GIF**).

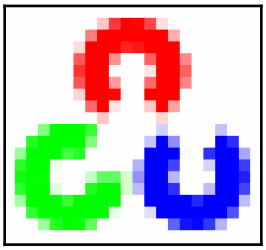
The **portable pixmap format** (**PPM**), the **portable bitmap format** (**PBM**), and the **portable graymap format** (**PGM**) specify rules for exchanging graphics files. Several applications refer to these file formats collectively as the **portable anymap format** (**PNM**). These files are a convenient and simple method of saving image data. Additionally, they are easy to read. In this sense, the PPM, PBM, and PGM formats are all designed to be as simple as possible.

Tagged Image File Format (**TIFF**) is an adaptable file format for handling images and data within a single file.

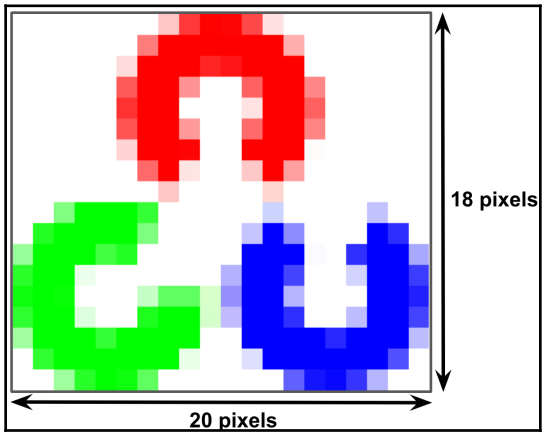
The lossless and lossy types of compression algorithms are applied to the image, resulting in images that are smaller than the uncompressed image. On the one hand, in lossless compression algorithms, the resulting image is equivalent to the original, meaning that after reversing the compression process, the resulting image is equivalent (equal) to the original. On the other hand, in lossy compression algorithms, the resulting image is not equivalent to the original, meaning that some details in the image are lost. In this sense, in many lossy compression algorithms, the level of compression can be adjusted.

The coordinate system in OpenCV

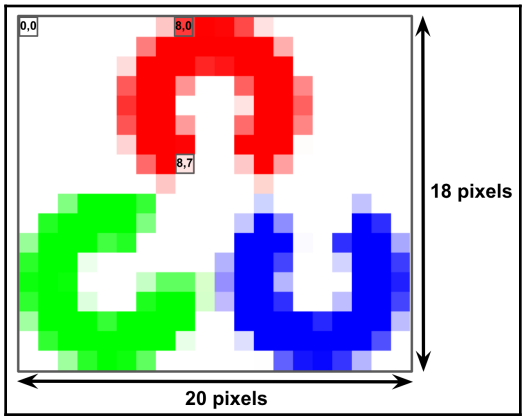
To show you the coordinate system in OpenCV and how to access individual pixels, we are going to show you a low-resolution image of the OpenCV logo:



This logo has a dimension of *20* × *18* pixels, that is, this image has 360 pixels. So, we can add the pixel count in every axis, as shown in the following image:



Now, we are going to look at the indexing of the pixels in the form *(x,y)*. Notice that pixels are zero-indexed, meaning that the upper left corner is at **(0, 0)**, not **(1, 1)**. Take a look at the following image, which indexes three individual pixels. As you can see, the upper left corner of the image is the coordinates of the origin. Moreover, *y* coordinates get larger as they go down:



The information for an individual pixel can be extracted from an image in the same way as an individual element of an array is referenced in Python. In the next section, we are going to see how we can do this.

Accessing and manipulating pixels in OpenCV

In this section, you will learn how to access and read pixel values with OpenCV and how to modify them. Additionally, you will learn how to access the image properties. If you want to work with many pixels at a time, you need to create **Region of Image** (**ROI**). In this section, you will learn how to do this. Finally, you will learn how to split and merge images.

Remember that in Python, images are represented as NumPy arrays. Therefore, most of the operations that are included in these examples are related to NumPy, so a good understanding about the NumPy package is required to both understand the code included in these examples and to write optimized code with OpenCV.

Accessing and manipulating pixels in OpenCV with BGR images

Now, we are going to see how we can work with BGR images in OpenCV. OpenCV loads the color images so that the blue channel is the first, the green channel is the second, and the red channel is the third. Please see the *Accessing and manipulating pixels in OpenCV with grayscale images* section to fully understand this concept.

First, read the image to work with using the `cv2.imread()` function. The image should be in the working directory, or a full path to the image should be provided. In this case, we are going to read the `logo.png` image and store it in the `img` variable:

```
# The function cv2.imread() is used to read an image from the the working
directory
# Alternatively, you should provide a full path of the image:
# Load OpenCV logo image (in this case from the working directoy):
img = cv2.imread('logo.png')
```

After the image has been loaded in `img`, we will gain access to some properties of the image. The first property we are going to extract from the loaded image is `shape`, which will tell us the number of rows, columns, and channels (if the image is in color). We will store this information in the `dimensions` variable for future use:

```
# To get the dimensions of the image use img.shape
# img.shape returns a tuple of number of rows, columns and channels (if a
colour image)
# If image is grayscale, img.shape returns a tuple of number of rows and
columns.
# So,it can be used to check if loaded image is grayscale or color image.
# Get the shape of the image:
dimensions = img.shape
```

Another property is the size of the image (`img.size` is equal to the multiplication of *height* × *width* × *channels*):

```
# Total number of elements is obtained by img.size:
total_number_of_elements= img.size
```

The property image datatype is obtained by `img.dtype`. In this case, the image datatype is `uint8` (unsigned char), because values are in the `[0 - 255]` range:

```
# Image datatype is obtained by img.dtype.
# img.dtype is very important because a large number of errors is caused by
invalid datatype.
# Get the image datatype:
image_dtype = img.dtype
```

To display an image, we will use the `cv2.imshow()` function to show an image in a window. The window automatically fits to the image size. The first argument to this function is the window name and the second one is the image to be displayed. In this case, since the loaded image has been stored in the `img` variable, we will use this variable as the second argument:

```
# The function cv2.imshow() is used to display an image in a window
# The first argument of this function is the window name
# The second argument of this function is the image to be shown.
# Each created window should have different window names.
# Show original image:
cv2.imshow("original image", img)
```

After the image is displayed, the `cv2.waitKey()` function, which is a keyboard binding function, will wait for a specified number of milliseconds for any keyboard event. The argument is the time in milliseconds. If any key is pressed at that time, the program will continue. If the number of milliseconds is `0` (`cv2.waitKey(0)`), it will wait indefinitely for a keystroke. Therefore, this function will allow us to see the displayed window waiting for a keystroke:

```
# The function cv2.waitKey(), which is a keyboard binding function, waits
for any keyboard event.
# This function waits the value indicated by the argument (in
milliseconds).
# If any keyboard event is produced in this period of time, the program
continues its execution
# If the value of the argument is 0, the program waits indefinitely until a
keyboard event is produced:
cv2.waitKey(0)
```

To access (read) a pixel value, we need to provide the row and column of the desired pixel to the `img` variable, which contains the loaded image. For example, to get the value of the pixel (`x=40`, `y=6`), we would use the following code:

```
# A pixel value can be accessed by row and column coordinates.
# In case of BGR image, it returns an array of (Blue, Green, Red) values.
# Get the value of the pixel (x=40, y=6):
(b, g, r) = img[6, 40]
```

We have loaded the three pixel values in three variables, `(b,g,r)`. You can see here that OpenCV uses the BGR format for color images. Additionally, we can access one channel at a time. In this case, we will use row, column, and the index of the desired channel for indexing. For example, to get only the blue value of the pixel (`x=40`, `y=6`), we would use the following code:

```
# We can only  access one channel at a time.
# In this case, we will use row, column and the index of the desired
channel for indexing.
# Get only blue value of the pixel (x=40, y=6):
b = img[6, 40, 0]
```

The pixel values can be also modified in the same way. Remember that it is the `(b, g, r)` format. For example, to set the pixel (`x=40`, `y=6`) to red, perform the following:

```
# The pixel values can be also modified in the same way - (b, g, r) format:
img[6, 40] = (0, 0, 255)
```

Sometimes, you will have to deal with a certain region rather than one pixel. In this case, the ranges of the values should be provided instead of the individual values. For example, to get to the top-left corner of the image, enter the following:

```
# In this case, we get the top left corner of the image:
top_left_corner = img[0:50, 0:50]
```

The `top_left_corner` variable is another image (smaller than `img`), but we can play with it in the same way.

Accessing and manipulating pixels in OpenCV with grayscale images

Grayscale images have only one channel. Therefore, some differences are introduced when working with these images. We are going to highlight these differences here.

Again, we will use the `cv2.imread()` function to read an image. In this case, the second argument is needed because we want to load the image in grayscale. The second argument is a flag specifying the way the image should be read. The value that's needed for loading an image in grayscale is `cv2.IMREAD_GRAYSCALE`:

```
# The function cv2.imshow() is used to display an image in a window
# The first argument of this function is the window name
# The second argument of this function is the image to be shown.
# In this case, the second argument is needed because we want to load the
image in grayscale.
# Second argument is a flag specifying the way the image should be read.
# Value needed for loading an image in grayscale: 'cv2.IMREAD_GRAYSCALE'.
# load OpenCV logo image:
gray_img = cv2.imread('logo.png', cv2.IMREAD_GRAYSCALE)
```

In this case, we store the image in the `gray_img` variable. If we get the dimensions of the image (using `gray_img.shape`), we will get only two values that is, rows and columns. In grayscale images, the channel information is not provided:

```
# To get the dimensions of the image use img.shape
# If color image, img.shape returns returns a tuple of number of rows,
columns and channels
# If grayscale, returns a tuple of number of rows and columns.
# So, it can be used to check if the loaded image is grayscale or color
image.
# Get the shape of the image (in this case only two components!):
dimensions = gray_img.shape
```

`img.shape` will return the dimensions of the image in a tuple, like this—`(99, 82)`.

A pixel value can be accessed by row and column coordinates. In grayscale images, only one value is obtained (usually called the **intensity** of the pixel). For example, if we want to get the intensity of the pixel (`x=40`, `y=6`), we would use the following code:

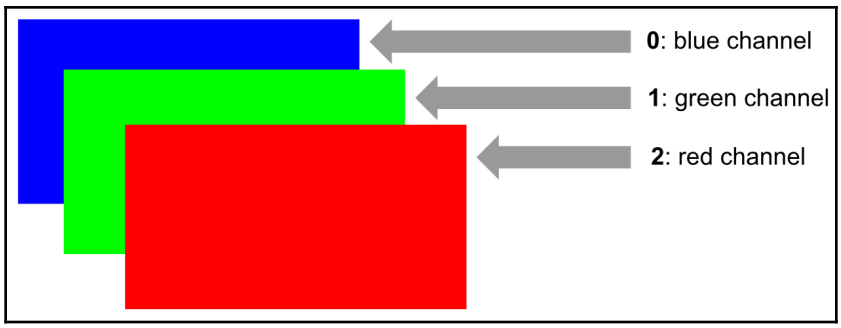
```
# You can access a pixel value by row and column coordinates.
# For BGR image, it returns an array of (Blue, Green, Red) values.
# Get the value of the pixel (x=40, y=6):
i = gray_img[6, 40]
```

The pixel values of the image can be also modified in the same way. For example, if we want to change the value of the pixel (`x=40`, `y=6`) to black (intensity equals to `0`), we would use the following code:

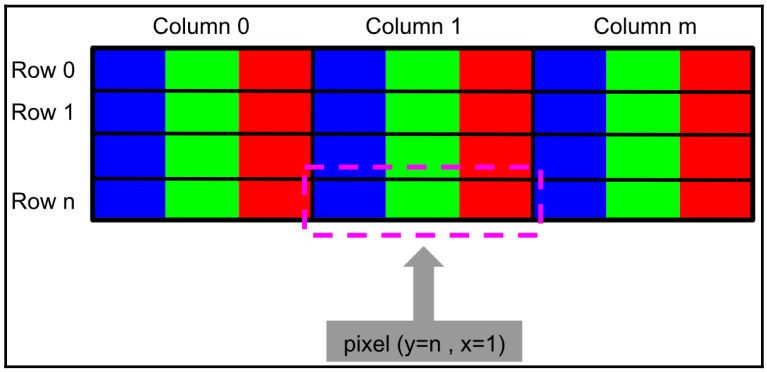
```
# You can modify the pixel values of the image in the same way.
# Set the pixel to black:
gray_img[6, 40] = 0
```

BGR order in OpenCV

We already mentioned that OpenCV uses the BGR color format instead of the RGB one. This can be seen in the following diagram, where you can see the order of the three channels:



The pixel structure of a BGR image can be seen in the following diagram. In particular, we have detailed how to access **pixel (y=n, x=1)** for clarification purposes:



Initial developers at OpenCV chose the BGR color format (instead of the RGB one) because at the time, the BGR color format was very popular among software providers and camera manufacturers. For example, in Windows, when specifying a color value using COLORREF, they used the BGR format, `0x00bbggrr` (`https://docs.microsoft.com/es-es/windows/desktop/gdi/colorref`). In summary, BGR was chosen for historical reasons.

Additionally, other Python packages use the RGB color format. Therefore, we need to know how to convert an image from one format into the other. For example, Matplotlib uses the RGB color format. Matplotlib (`https://matplotlib.org/`) is the most popular 2D Python plotting library and offers you a wide variety of plotting methods. You can interact with the plotted images (for example, zoom into images and save them). Matplotlib can be used both in Python scripts or in the Jupyter Notebook. You can check out the Matplotlib documentation for further details (`https://matplotlib.org/contents.html`).

Therefore, a good choice for your projects is to show the images using the Matplotlib package instead of the functionality offered by OpenCV. Now we are going to see now how we can deal with the different color formats in the two libraries.

First of all, we load the image using the `cv2.imread()` function:

```
# Load image using cv2.imread:
img_OpenCV = cv2.imread('logo.png')
```

The image is stored in the `img_OpenCV` variable because the `cv2.imread()` function loads the image in BGR order. Then, we split the loaded image into its three channels, `(b, g, r)`, using the `cv2.split()` function. The parameter of this function is the image we want to split:

```
# Split the loaded image into its three channels (b, g, r):
b, g, r = cv2.split(img_OpenCV)
```

The next step is to merge the channels again (in order to build a new image based on the information provided by the channels) but in a different order. We change the order of the `b` and `r` channels in order to follow the RGB format, that is, the one we need for Matplotlib:

```
# Merge again the three channels but in the RGB format:
img_matplotlib = cv2.merge([r, g, b])
```

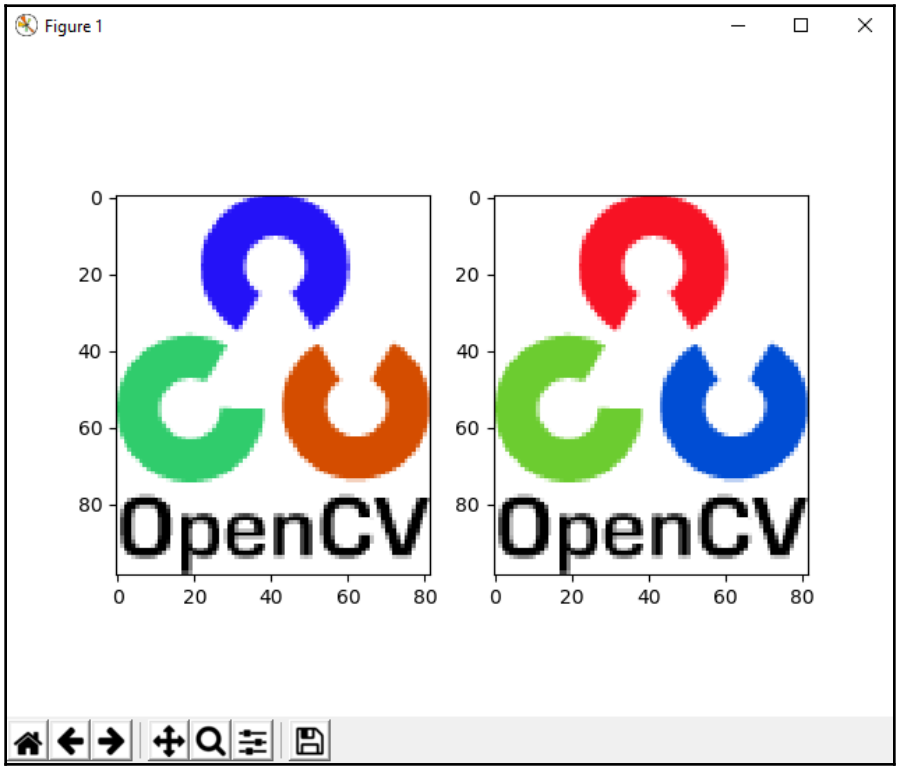
At this point, we have two images (`img_OpenCV` and `img_matplotlib`), which are going to be plotted with both OpenCV and Matplotlib so that we can see the results. First of all, we will show these two images with Matplotlib.

To show the two images with Matplotlib in the same window, we will use `subplot`, which places multiple images within the same window. You have three parameters to use within `subplot`, for example `subplot(m,n,p)`. In this case, `subplot` handles plots in a m *x n* grid, where *m* establishes the number of rows, *n* establishes the number of columns, and *p* establishes where you want to place your plot in the grid. To show the images with Matplotlib, we will use `imshow`.

In this case, as we are showing two images horizontally, `m = 1` and `n = 2`. We will be using `p = 1` for the first subfigure (`img_OpenCV`) and `p = 2` for the second subfigure (`img_matplotlib`):

```
# Show both images (img_OpenCV and img_matplotlib) using matplotlib
# This will show the image in wrong color:
plt.subplot(121)
plt.imshow(img_OpenCV)
# This will show the image in true color:
plt.subplot(122)
plt.imshow(img_matplotlib)
plt.show()
```

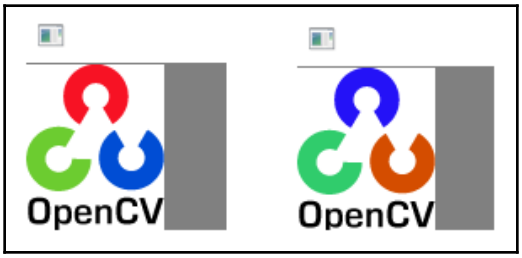
Therefore, the output you will get should be very similar to the output that's shown in the following diagram:



As you can see, the first subfigure shows the image in the wrong color (BGR order), while the second subfigure shows the image in true color (RGB order). In the same way, we will show the two images using `cv2.imshow()`:

```
# Show both images (img_OpenCV and img_matplotlib) using cv2.imshow()
# This will show the image in true color:
cv2.imshow('bgr image', img_OpenCV)
# This will show the image in wrong color:
cv2.imshow('rgb image', img_matplotlib)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The following screenshot shows what you will get from executing the previous code:

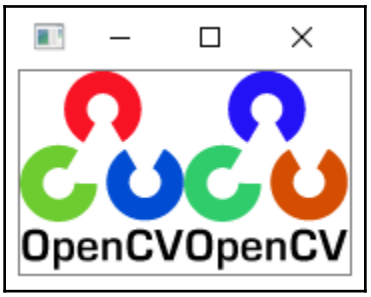


As expected, the screenshot shows the image in true color, while the second figure shows the image in the wrong color.

Additionally, if we want to show the two images in the same window, we can build a *full* image that contains the two images, concatenating them horizontally. To do so, we will use NumPy's `concatenate()` method. The parameters of this method are the two images to concatenate and the axis. In this case, `axis = 1` (to stack them horizontally):

```
# To stack horizontally (img_OpenCV to the left of img_matplotlib):
img_concats = np.concatenate((img_OpenCV, img_matplotlib), axis=1)
# Now, we show the concatenated image:
cv2.imshow('bgr image and rgb image', img_concats)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Check out the following screenshot to see the concatenated image:



One consideration to take into account is that `cv2.split()` is a time-consuming operation. Depending on your needs, consider using NumPy indexing. For example, if you want to get one channel of the image, instead of using `cv2.split()` to get the desired channel, you can use NumPy indexing. See the following example to get the channels using NumPy indexing:

```
# Using numpy capabilities to get the channels and to build the RGB image
# Get the three channels (instead of using cv2.split):
B = img_OpenCV[:, :, 0]
G = img_OpenCV[:, :, 1]
R = img_OpenCV[:, :, 2]
```

Another consideration is that you can use NumPy for converting the image from BGR into RGB in a single instruction:

```
# Transform the image BGR to RGB using Numpy capabilities:
img_matplotlib = img_OpenCV[:, :, ::-1]
```

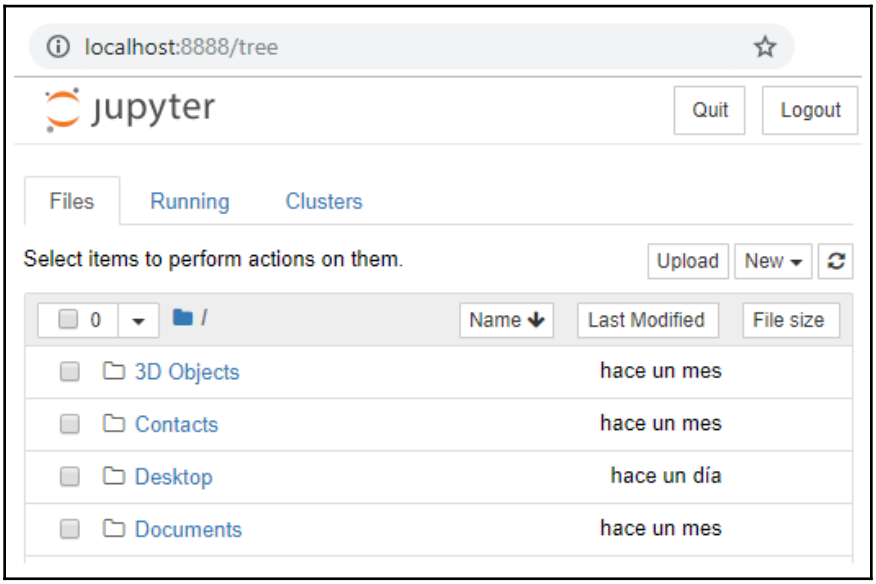
To summarize everything in this chapter, we have created two Jupiter Notebooks. In these notebooks, you can play with all of the concepts that have been introduced so far:

- `Getting-And-Setting-BGR.ipynb`
- `Getting-And-Setting-GrayScale.ipynb`

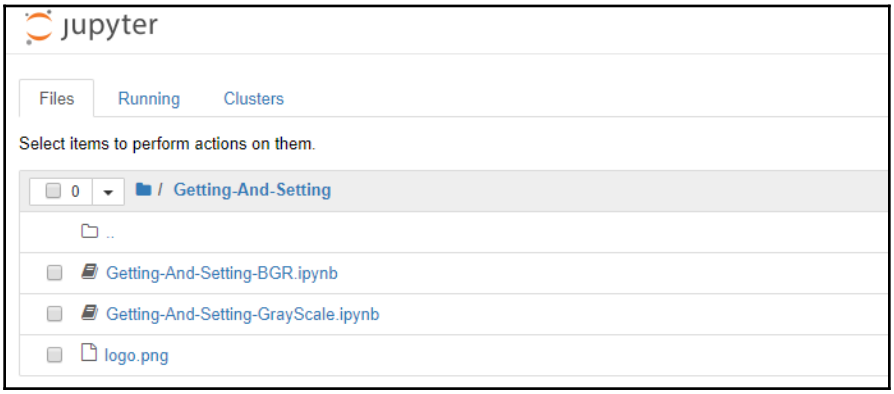
Taking advantage of the benefits notebook (and all of the information included in this chapter), no additional information is needed to play with them. So, go ahead and try it for yourself. Remember (see `Chapter 1`, *Setting Up OpenCV*) that to run the notebook, you need to run the following command at the Terminal (Mac/Linux) or Command Prompt (Windows):

```
$ jupyter notebook
```

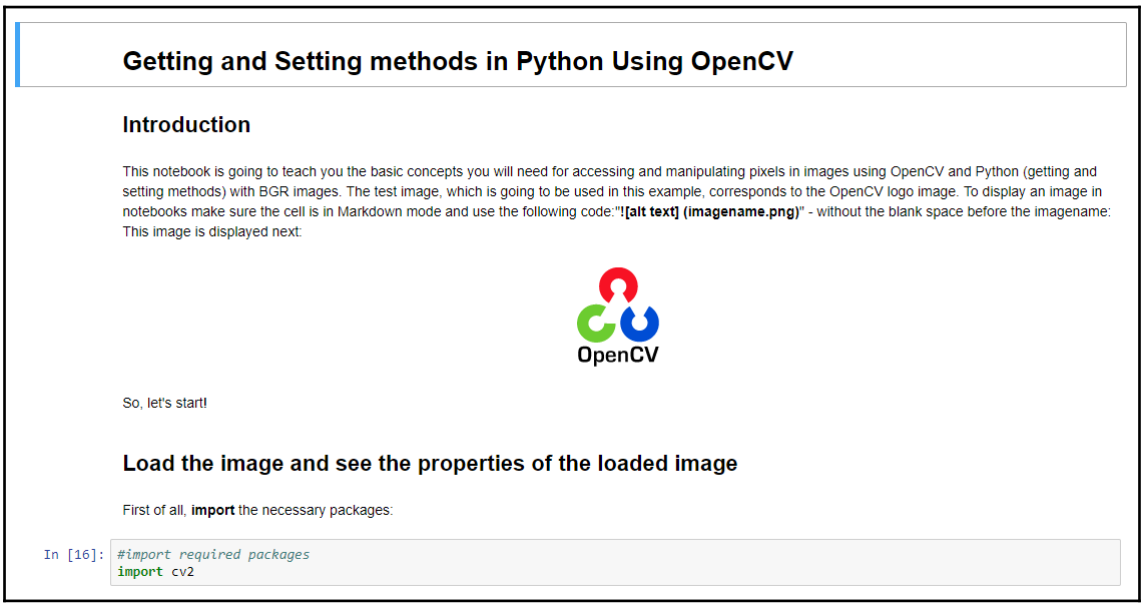
This command will print information in connection with the notebook server, including the URL of the web application (by default, this URL is `http://localhost:8888`). Additionally, this command will also open your web browser pointing to this URL:



At this point, you are able to upload the `Getting-And-Setting-BGR.ipynb` and `Getting-And-Setting-GrayScale.ipynb` files by clicking on the **Upload** button (see the previous screenshot). These files use the `logo.png` image. Therefore, you should upload this image in the same way. After loading these three files, you should see these files loaded:



At this point, you can open these notebooks by clicking on them. You should see the content of the notebook, which is shown in the following screenshot:



Finally, you can start executing the loaded notebook document. You can execute the notebook step by step (one cell a time) by pressing *Shift* + *Enter*. Additionally, you can execute the whole notebook in a single step by clicking on the **Cell** | **Run All** menu. Moreover, you can also restart the kernel (the computational engine) by clicking on the **Kernel** | **Restart** menu.

For more information on editing a notebook, check out `https://github.com/jupyter/notebook/blob/master/docs/source/examples/Notebook/Notebook%20Basics.ipynb`, which is also a notebook!

Summary

In this chapter, we looked at the key concepts related to images. Images constitute rich information that's necessary to build your computer vision projects. OpenCV uses the BGR color format instead of RGB, but some Python packages (for example, Matplotlib) use the latter format. Therefore, we have covered how to convert the image from one color format into the other.

Additionally, we have summarized the main functions and options to work with images:

- To access image properties
- Some OpenCV functions, such as `cv2.imread()`, `cv2.split()`, `cv2.merge()`, `cv2.imshow()`, `cv2.waitKey()`, and `cv2.destroyAllWindows()`
- How to get and set image pixels in both BGR and grayscale images

Finally, we included two notebooks, which let you play with all these concepts. Remember that once you have loaded the notebook, you can run it step by step by pressing *Shift* + *Enter* or run the notebook in a single step by clicking on the **Cell** | **Run All** menu.

In the next chapter, you will learn how to cope with files and images, which are necessary for building your computer vision applications.

Questions

1. What are the main image-processing steps?
2. What are the three processing levels?
3. What is the difference between a grayscale image and a black and white image?
4. What is a pixel?
5. What is image resolution?
6. What OpenCV functions do you use to perform the following actions?
    - Load (read) an image
    - Show an image
    - Wait for a keystroke
    - Split the channels
    - Merge the channels
7. What command do you use to run the Jupyter Notebook?
8. What color will you get with the following triplets?
    - `B = 0, G = 255, R = 255`
    - `B = 255, G = 255, R = 0`
    - `B = 255, G = 0, R = 255`
    - `B = 255, G = 255, R = 255`
9. Suppose that you have loaded an image in `img`. How do you check whether `img` is color or grayscale?

Further reading

The following references will help you dive deeper into the concepts that were presented in this chapter:

- For more information about Git, have a look at this book:

  Mastering Git, by Jakub Narębski (`https://www.packtpub.com/application-development/mastering-git`)

- For more on Jupyter Notebook:

  Jupyter Notebook for All – Part I, by Dan Toomey (`https://www.packtpub.com/big-data-and-business-intelligence/jupyter-notebook-for-all-part-1-video`)

3
Handling Files and Images

In any kind of project, coping with files and images is a key aspect. In this sense, many projects should work with files as forms of data input. Additionally, the project can generate some data after any kind of processing has been done, which can be outputted in the form of files or images. In computer vision, this information flow (input-processing-output) takes special relevance due to the inherent characteristics of these types of projects (for example, images to be processed and models that are generated by machine learning algorithms).

In this chapter, we are going to see how we can handle both files and images. You will learn how to cope with files and images, which are necessary for building computer vision applications.

More specifically, we will cover the following topics:

- A theoretical introduction to handling files and images
- Reading/writing images
- Reading camera frames and video files
- Writing a video file
- Playing with video capture properties

Technical requirements

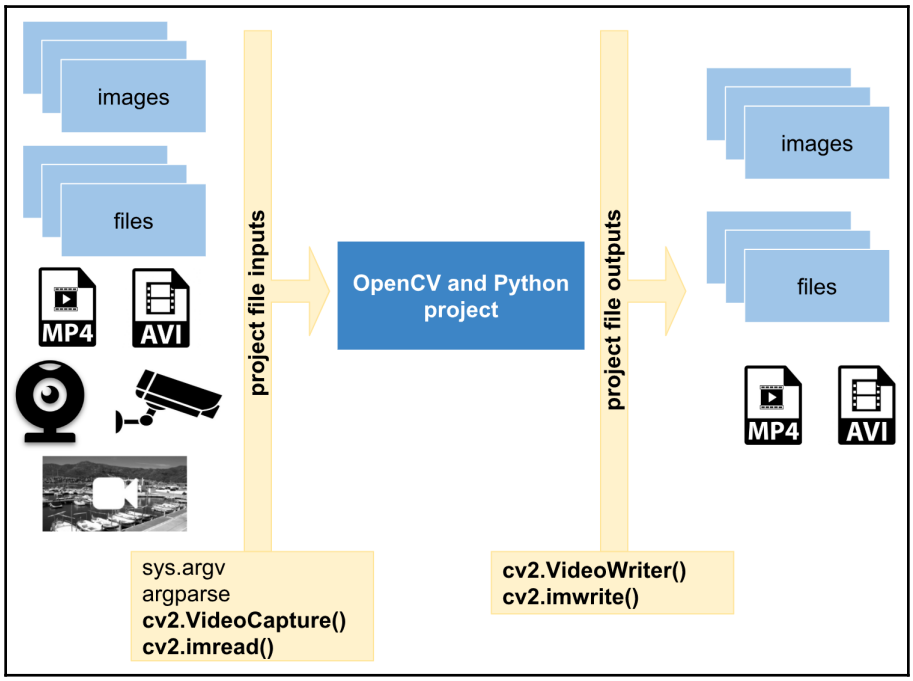
The technical requirements for this chapter are listed as follows:

- Python and OpenCV
- A Python-specific IDE
- The NumPy and Matplotlib Python packages
- Git client

The GitHub repository for Mastering OpenCV with Python can be accessed at `https://github.com/PacktPublishing/Mastering-OpenCV-4-with-Python`.

An introduction to handling files and images

Before going deeper in handling files and images, we are going to give you an overview of what we will look at in this chapter. This overview is summarized in the following diagram:



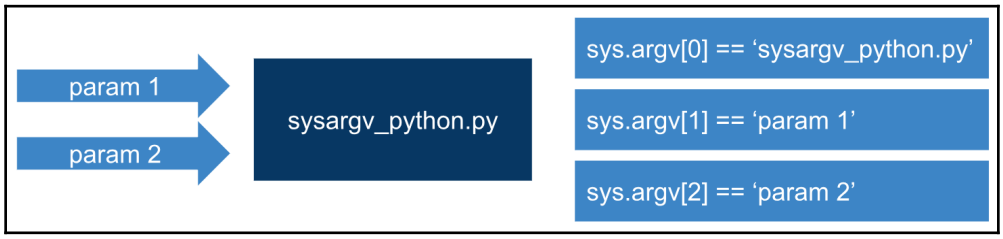
In the preceding diagram, you can see that a computer vision project (for example, an **OpenCV and Python project**) should deal with some input files (for example, **files** and **images**). Additionally, after some processing, the project can output some files (for example, **images** and **files**). So, in this chapter, we are going to see how to cope with these requirements and how to implement this flow (input-processing-output) properly.

A primary and necessary step to execute a program is to properly cope with command-line arguments, which are parameters that are given to a program or script containing some kind of *parameterized* information. For example, if you write a script to add two numbers, a common approach is to have two arguments, which are the two numbers that are necessary to perform the addition. In computer vision projects, **images** and different types of files are usually passed to the script as command-line arguments.

Command-line arguments are a common and simple way to parameterize the execution of programs.

sys.argv

To handle command-line arguments, Python uses `sys.argv`. In this sense, when a program is executed, Python takes all the values from the command line and sets them in the `sys.argv` list. The first element of the list is the full path to the script (or the script name—it is operating system dependent), which is always `sys.argv[0]`. The second element of the list is the first argument to the script, which is `sys.argv[1]`, and so on. This can be seen in the following diagram, where the `sysargv_python.py` script is executed with two arguments:



To see how `sys.argv` works, we are going to use the `sysargv_python.py` script:

```
# Import the required packages
import sys

# We will print some information in connection with sys.argv to see how it
works:
print("The name of the script being processed is:
'{}'".format(sys.argv[0]))
print("The number of arguments of the script is:
'{}'".format(len(sys.argv)))
print("The arguments of the script are: '{}'".format(str(sys.argv)))
```

If we execute this script without any parameter, we will see the following information:

```
The name of the script being processed is: 'sysargv_python.py'
The number of arguments of the script is: '1'
The arguments of the script are: '['sysargv_python.py']'
```

Additionally, if we execute this script with one parameter (for example, `sysargv_python.py OpenCV`), we will get the following information:

```
The name of the script being processed is: 'sysargv_python.py'
The number of arguments of the script is: '2'
The arguments of the script are: '['sysargv_python.py', 'OpenCV']'
```

As you can see, the first element, `sysargv_python.py` (`sys.argv[0]`), of the list is the script name. The second element, `OpenCV`, of the list (`sys.argv[1]`) is the first argument to our script.

`argv[0]` is the script name, which is operating system dependent if it is a full pathname or not. See `https://docs.python.org/3/library/sys.html` for more information..

Argparse – command-line option and argument parsing

It should be taken into account that we should not handle `sys.argv` directly, mainly when our programs take complex parameters or multiple filenames. Alternatively, we should use Python's `argparse` library, which handles command-line arguments in a systematic way, making it accessible to write user-friendly command-line programs. In other words, Python has a module called `argparse` (`https://docs.python.org/3/library/argparse.html`) in the standard library for parsing command-line arguments. First, the program determines what arguments it requires. Then, `argparse` will work out how to parse these arguments to `sys.argv`. Also, `argparse` produces help and usage messages, and issues errors when invalid arguments are provided.

The minimum example to introduce this module is given here, `argparse_minimal.py`, which is shown as follows:

```
# Import the required packages
import argparse

# We first create the ArgumentParser object
# The created object 'parser' will have the necessary information
# to parse the command-line arguments into data types.
parser = argparse.ArgumentParser()

# The information about program arguments is stored in 'parser' and used
when parse_args() is called.
# ArgumentParser parses arguments through the parse_args() method:
parser.parse_args()
```

Running this script with no parameters results in nothing being displayed to `stdout`. However, if we include the `--help` (or `-h`) option, we will get the usage message of the script:

```
usage: argparse_minimal.py [-h]
optional arguments:
-h, --help show this help message and exit
```

Specifying any other parameters results in an error, for example:

```
argparse_minimal.py 6
usage: argparse_minimal.py [-h]
argparse_minimal.py: error: unrecognized arguments: 6
```

Therefore, we must call this script with the `-h` argument. In this way, the usage message information will be shown. No other possibilities are allowed as no arguments are defined. In this way, the second example to introduce `argparse` is to add a parameter, which can be seen in the `argparse_positional_arguments.py` example:

```
# Import the required packages
import argparse

# We first create the ArgumentParser object
# The created object 'parser' will have the necessary information
# to parse the command-line arguments into data types.
parser = argparse.ArgumentParser()

# We add a positional argument using add_argument() including a help
parser.add_argument("first_argument", help="this is the string text in
connection with first_argument")
```

```
# The information about program arguments is stored in 'parser'
# Then, it is used when the parser calls parse_args().
# ArgumentParser parses arguments through the parse_args() method:
args = parser.parse_args()

# We get and print the first argument of this script:
print(args.first_argument)
```

We added the `add_argument()` method. This method is used to specify what command-line options the program will accept. In this case, the `first_argument` argument is required. Additionally, the `argparse` module stores all the parameters, matching its name with the name of each added parameter—in this case, `first_argument`. Therefore, to obtain our parameter, we perform `args.first_argument`.

If this script is executed as `argparse_positional_arguments.py 5`, the output will be `5`. However, if the script is executed without arguments as `argparse_positional_arguments.py`, the output will be as follows:

```
usage: argparse_positional_arguments.py [-h] first_argument
argparse_positional_arguments.py: error: the following arguments are
required: first_argument
```

Finally, if we execute the script with the `-h` option, the output will be as follows:

```
usage: argparse_positional_arguments.py [-h] first_argument
positional arguments:
 first_argument this is the string text in connection with first_argument
optional arguments:
 -h, --help show this help message and exit
```

By default, `argparse` treats the options we give it as strings. Therefore, if the parameter is not a string, the `type` option should be established. We will see the `argparse_sum_two_numbers.py` script adding two arguments and, hence, these two arguments are of the `int` type:

```
# Import the required packages
import argparse

# We first create the ArgumentParser object
# The created object 'parser' will have the necessary information
# to parse the command-line arguments into data types.
parser = argparse.ArgumentParser()

# We add 'first_number' argument using add_argument() including a help. The
type of this argument is int
parser.add_argument("first_number", help="first number to be added",
```

```
type=int)

# We add 'second_number' argument using add_argument() including a help The
type of this argument is int
parser.add_argument("second_number", help="second number to be added",
type=int)

# The information about program arguments is stored in 'parser'
# Then, it is used when the parser calls parse_args().
# ArgumentParser parses arguments through the parse_args() method:
args = parser.parse_args()
print("args: '{}'".format(args))

print("the sum is: '{}'".format(args.first_number + args.second_number))

# Additionally, the arguments can be stored in a dictionary calling vars()
function:
args_dict = vars(parser.parse_args())

# We print this dictionary:
print("args_dict dictionary: '{}'".format(args_dict))

# For example, to get the first argument using this dictionary:
print("first argument from the dictionary:
'{}'".format(args_dict["first_number"]))
```

If the script is executed without arguments, the output will be as follows:

```
argparse_sum_two_numbers.py
usage: argparse_sum_two_numbers.py [-h] first_number second_number
argparse_sum_two_numbers.py: error: the following arguments are required:
first_number, second_number
```

Additionally, if we execute the script with the `-h` option, the output will be as follows:

```
argparse_sum_two_numbers.py --help
usage: argparse_sum_two_numbers.py [-h] first_number second_number

positional arguments:
  first_number first number to be added
  second_number second number to be added

optional arguments:
  -h, --help show this help message and exit
```

It should be taken into account that in the previous example, we introduced the possibility of storing arguments in a dictionary by calling the `vars()` function:

```
# Additionally, the arguments can be stored in a dictionary calling vars()
function:
args_dict = vars(parser.parse_args())

# We print this dictionary:
print("args_dict dictionary: '{}'".format(args_dict))

# For example, to get the first argument using this dictionary:
print("first argument from the dictionary:
'{}'".format(args_dict["first_number"]))
```

For example, if this script is executed as `argparse_sum_two_numbers.py 5 10`, the output will be as follows:

```
args: 'Namespace(first_number=5, second_number=10)'
the sum is: '15'
args_dict dictionary: '{'first_number': 5, 'second_number': 10}'
first argument from the dictionary: '5'
```

This was a quick introduction to both `sys.argv` and `argparse`. An advanced introduction to `argparse` can be seen at `https://docs.python.org/3/howto/argparse.html`. Additionally, its docs are quite detailed and meticulous and have covered plenty of examples (`https://docs.python.org/3/library/argparse.html`). At this point, you can now learn how to read and write images using `argparse` in your OpenCV and Python programs, which will be shown in the *Reading and writing images* section.

Reading and writing images

In computer vision projects, images are commonly used as command-line arguments in our scripts. In the following sections, we are going to see how we can read and write images.

Reading images in OpenCV

The following example, `argparse_load_image.py`, shows you how to load an image:

```
# Import the required packages
import argparse
import cv2

# We first create the ArgumentParser object
```

```
# The created object 'parser' will have the necessary information
# to parse the command-line arguments into data types.
parser = argparse.ArgumentParser()

# We add 'path_image' argument using add_argument() including a help. The
type of this argument is string (by default)
parser.add_argument("path_image", help="path to input image to be
displayed")

# The information about program arguments is stored in 'parser'
# Then, it is used when the parser calls parse_args().
# ArgumentParser parses arguments through the parse_args() method:
args = parser.parse_args()

# We can now load the input image from disk:
image = cv2.imread(args.path_image)

# Parse the argument and store it in a dictionary:
args = vars(parser.parse_args())

# Now, we can also load the input image from disk using args:
image2 = cv2.imread(args["path_image"])

# Show the loaded image:
cv2.imshow("loaded image", image)
cv2.imshow("loaded image2", image2)

# Wait until a key is pressed:
cv2.waitKey(0)

# Destroy all windows:
cv2.destroyAllWindows()
```

In this example, the required argument is `path_image`, which contains the path of the image we want to load. The path of the image is a string. Therefore, no type should be included in the positional argument because it is a string by default. Both `args.path_image` and `args["path_image"]` will contain the path of the image (two different ways of getting the value from the parameter), so we will use them as the parameter of the `cv2.imread()` function.

Reading and writing images in OpenCV

A common approach is to load an image, perform some kind of processing, and finally output this processed image (see `Chapter 2`, *Image Basics in OpenCV*, for a deeper explanation of these three steps). In this sense, the processed image can be saved to disk. In the following example, these three steps (load, processing, and save) are introduced. In this case, the processing step is very simple (convert the image into grayscale). This can be seen in the following example, `argparse_load_processing_save_image.py`:

```
# Import the required packages
import argparse
import cv2

# We first create the ArgumentParser object
# The created object 'parser' will have the necessary information
# to parse the command-line arguments into data types.
parser = argparse.ArgumentParser()

# Add 'path_image_input' argument using add_argument() including a help.
The type is string (by default):
parser.add_argument("path_image_input", help="path to input image to be
displayed")

# Add 'path_image_output' argument using add_argument() including a help.
The type is string (by default):
parser.add_argument("path_image_output", help="path of the processed image
to be saved")

# Parse the argument and store it in a dictionary:
args = vars(parser.parse_args())

# We can load the input image from disk:
image_input = cv2.imread(args["path_image_input"])

# Show the loaded image:
cv2.imshow("loaded image", image_input)

# Process the input image (convert it to grayscale):
gray_image = cv2.cvtColor(image_input, cv2.COLOR_BGR2GRAY)

# Show the processed image:
cv2.imshow("gray image", gray_image)

# Save the processed image to disk:
cv2.imwrite(args["path_image_output"], gray_image)

# Wait until a key is pressed:
```

```
cv2.waitKey(0)

# Destroy all windows:
cv2.destroyAllWindows()
```

In this previous example, there are two required arguments. The first one is `path_image_input`, which contains the path of the image we want to load. The path of the image is a string. Therefore, no type should be included in the positional argument because it is a string by default. The second one is `path_image_output`, which contains the path of the resulting image we want to save. In this example, the processing step consists of converting the loaded image into grayscale:

```
# Process the input image (convert it to grayscale)
gray_image = cv2.cvtColor(image_input, cv2.COLOR_BGR2GRAY)
```

It should be noted that the second argument, `cv2.COLOR_BGR2GRAY`, assumes that the loaded image is a BGR color image. If you have loaded an RGB color image and you want to convert it into grayscale, you should use `cv2.COLOR_RGB2GRAY`.

This is a very simple processing step, but it is included for the sake of simplicity. In future chapters, more elaborate processing algorithms will be shown.

Reading camera frames and video files

In some projects, you have to capture camera frames (for example, capture frames with the webcam of your laptop). In OpenCV, we have `cv2.VideoCapture`, which is a class for video capturing from different sources, such as image sequences, video files, and cameras. In this section, we are going to see some examples to introduce us to this class for capturing camera frames.

Reading camera frames

This first example, `read_camera.py`, shows you how to read frames from a camera that's connected to your computer. The required argument is `index_camera`, which indicates the index of the camera to read. If you have connected a webcam to your computer, it has an index of `0`. Additionally, if you have a second camera, you can select it by passing `1`. As you can see, the type of this parameter is `int`.

The first step to work with `cv2.VideoCapture` is to create an object to work with. In this case, the object is `capture`, and we call the constructor like this:

```
 # We create a VideoCapture object to read from the camera (pass 0):
capture = cv2.VideoCapture(args.index_camera)
```

If `index_camera` is `0` (your first connected camera), it is equivalent to `cv2.VideoCapture(0)`. To check whether the connection has been established correctly, we have the `capture.isOpened()` method, which returns `False` if the connection could not be established. In the same way, if the capture was initialized correctly, this method returns `True`.

To capture footage frame by frame from the camera, we call the `capture.read()` method, which returns the frame from the camera. This frame has the same structure as an image in OpenCV, so we can work with it in the same way. For example, to convert the frame into grayscale, do the following:

```
gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

Additionally, `capture.read()` returns a bool. This bool indicates whether the frame has been correctly read from the capture object.

Accessing some properties of the capture object

Finally, you can access some properties of the capture object using `capture.get(property_identifier)`. In this case, we get some properties, such as frame **width**, frame **height**, and **frames per second** (**fps**). If we call a property that is not supported, the returned value will be `0`:

```
# Import the required packages
import cv2
import argparse

# We first create the ArgumentParser object
# The created object 'parser' will have the necessary information
# to parse the command-line arguments into data types.
parser = argparse.ArgumentParser()

# We add 'index_camera' argument using add_argument() including a help.
parser.add_argument("index_camera", help="index of the camera to read
from", type=int)
args = parser.parse_args()

# We create a VideoCapture object to read from the camera (pass 0):
```

```
capture = cv2.VideoCapture(args.index_camera)

# Get some properties of VideoCapture (frame width, frame height and frames
per second (fps)):
frame_width = capture.get(cv2.CAP_PROP_FRAME_WIDTH)
frame_height = capture.get(cv2.CAP_PROP_FRAME_HEIGHT)
fps = capture.get(cv2.CAP_PROP_FPS)

# Print these values:
print("CV_CAP_PROP_FRAME_WIDTH: '{}'".format(frame_width))
print("CV_CAP_PROP_FRAME_HEIGHT : '{}'".format(frame_height))
print("CAP_PROP_FPS : '{}'".format(fps))

# Check if camera opened successfully
if capture.isOpened()is False:
    print("Error opening the camera")

# Read until video is completed
while capture.isOpened():
    # Capture frame-by-frame from the camera
    ret, frame = capture.read()

    if ret is True:
        # Display the captured frame:
        cv2.imshow('Input frame from the camera', frame)

        # Convert the frame captured from the camera to grayscale:
        gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Display the grayscale frame:
        cv2.imshow('Grayscale input camera', gray_frame)

        # Press q on keyboard to exit the program
        if cv2.waitKey(20) & 0xFF == ord('q'):
            break
    # Break the loop
    else:
        break

# Release everything:
capture.release()
cv2.destroyAllWindows()
```

Saving camera frames

This previous example can be easily modified to add a useful functionality. Imagine that you want to save some frames to disk when something interesting happens. In the following example, `read_camera_capture.py`, we are going to add this functionality. When the *C* key is pressed on the keyboard, we save the current frame to disk. We save both the BGR and the grayscale frames. The code that performs this functionality is shown here:

```
# Press c on keyboard to save current frame
if cv2.waitKey(20) & 0xFF == ord('c'):
    frame_name = "camera_frame_{}.png".format(frame_index)
    gray_frame_name = "grayscale_camera_frame_{}.png".format(frame_index)
    cv2.imwrite(frame_name, frame)
    cv2.imwrite(gray_frame_name, gray_frame)
    frame_index += 1
```

`ord('c')` returns the value representing the `c` character using eight bits. Additionally, the `cv2.waitKey()` value is bitwise AND using the `&` operator with `0xFF` to get only its last eight bits. Therefore, we can perform a comparison between these two 8-bit values. When the *C* key is pressed, we build the names for both frames. Then, we save the two images to disk. Finally, `frame_index` is incremented so that it's ready for the next frame to be saved. Check out `read_camera_capture.py` to see the full code of this script.

Reading a video file

`cv2.VideoCapture` also allows us to read a video file. Therefore, to read a video file, the path to the video file should be provided when creating the `cv2.VideoCapture` object:

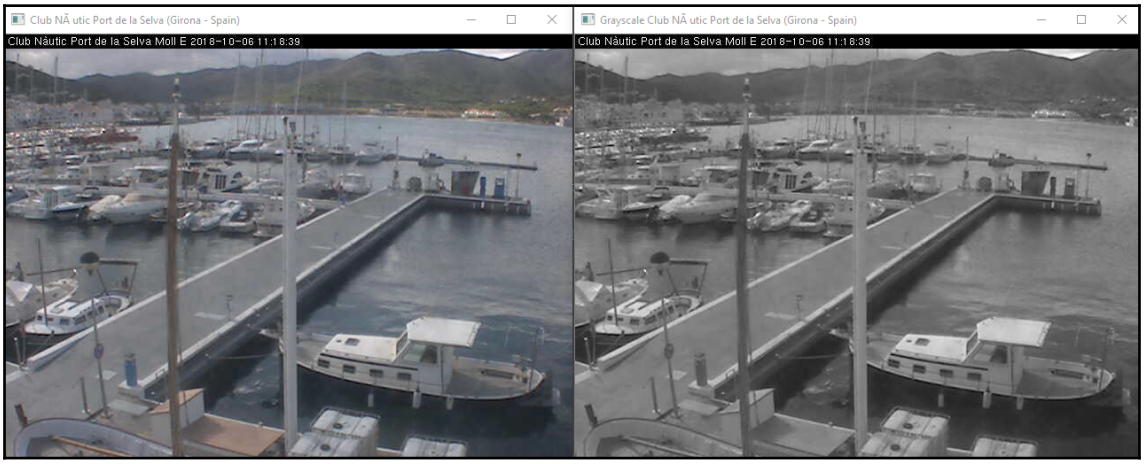
```
# We first create the ArgumentParser object
# The created object 'parser' will have the necessary information
# to parse the command-line arguments into data types.
parser = argparse.ArgumentParser()

# We add 'video_path' argument using add_argument() including a help.
parser.add_argument("video_path", help="path to the video file")
args = parser.parse_args()

# Create a VideoCapture object. In this case, the argument is the video
file name:
capture = cv2.VideoCapture(args.video_path)
```

Check out `read_video_file.py` to see the full example of how to read and display a video file using `cv2.VideoCapture`.

Reading from an IP camera

To finish with `cv2.VideoCapture`, we are going to see how we can read from an IP camera. Reading from an IP camera in OpenCV is very similar to reading from a file. In this sense, only the parameter to the constructor of `cv2.VideoCapture` should be changed. The good thing about this is that you do not need an IP camera in your local network to try this functionality. There are many public IP cameras you can try to connect. For example, we are going to connect to an IP public camera, which is placed at *Club Nàutic Port de la Selva – Costa Brava – Cap de Creus (Girona, Spain)*. The web page of this port is hosted at `https://www.cnps.cat/`. You can navigate to the webcam sections (`https://www.cnps.cat/webcams`/) to find some webcams to connect with.

Therefore, the only thing you have to modify is the parameter that's given to `cv2.VideCapture`. In this case, it's `http://217.126.89.102:8010/axis-cgi/mjpg/video.cgi`. If you execute this example (`read_ip_camera.py`), you should see something similar to the following screenshot, where both the BGR and the grayscale images that were obtained from the IP camera are shown:



Writing a video file

In this section, we are going to see how we can write to video files using `cv2.VideoWriter`. However some concepts (for example, fps, codecs, and video file formats) should be introduced first.

Calculating frames per second

In the *Reading camera frame and video files* section, we saw how we can get some properties from the `cv2.VideoCapture` object. fps is an important metric in computer vision projects. This metric indicates how many frames are processed per second. It is safe to say that a higher number of fps is better. However, the number of frames your algorithm should process every second will depend on the specific problem you have to solve. For example, if your algorithm should track and detect people walking down the street, 15 fps is probably enough. But if your goal is to detect and track cars going fast on a highway, 20-25 fps are probably necessary.

Therefore, it is important to know how to calculate the fps metric in your computer vision projects. In the following example, `read_camera_fps.py`, we are going to modify `read_camera.py` to output the number of fps. The key points are shown in the following code:

```
# Read until the video is completed, or 'q' is pressed
while capture.isOpened():
    # Capture frame-by-frame from the camera
    ret, frame = capture.read()

    if ret is True:
        # Calculate time before processing the frame:
        processing_start = time.time()

        # All the processing should be included here
        # ...
        # ...
        # End of processing

        # Calculate time after processing the frame
        processing_end = time.time()

        # Calculate the difference
        processing_time_frame = processing_end - processing_start

        # FPS = 1 / time_per_frame
        # Show the number of frames per second
        print("fps: {}".format(1.0 / processing_time_frame))

    # Break the loop
    else:
        break
```

First, we take the time before the processing is done:

```
processing_start = time.time()
```

Then, we take the time after all the processing is done:

```
processing_end = time.time()
```

Following that, we calculate the difference:

```
processing_time_frame = processing_end - processing_start
```

Finally, we calculate and print the number of fps:

```
print("fps: {}".format(1.0 / processing_time_frame))
```

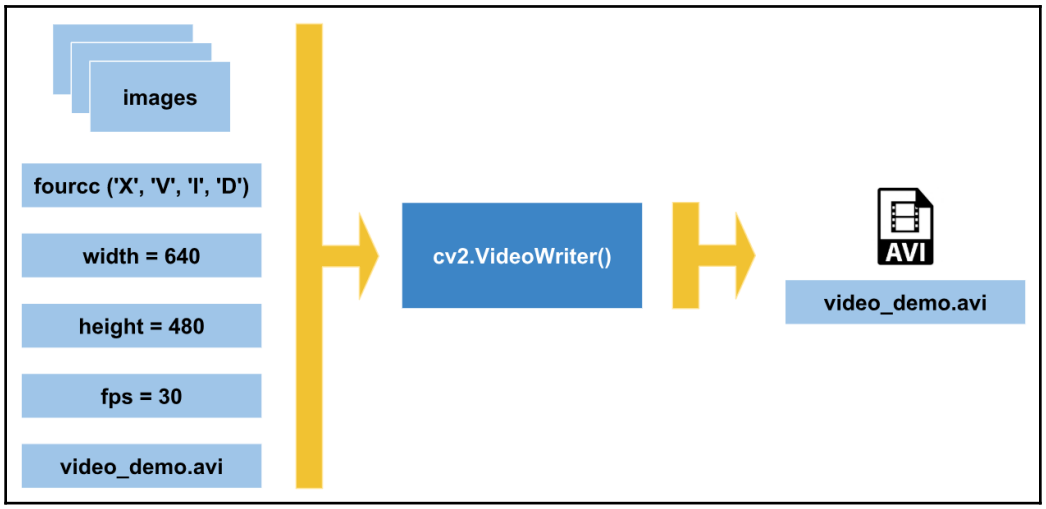
Considerations for writing a video file

A video code is a piece of software that's used to both compress and decompress a digital video. Therefore, a codec can be used to convert an uncompressed video into a compressed one, or it can be used to convert a compressed video to an uncompressed one. The compressed video format usually follows a standard specification called **video compression specification** or **video coding format**. In this sense, OpenCV provides FOURCC, which is a 4-byte code that's used to specify the video codec. FOURCC stands for *four character code*. The list of all available codes can be seen at `http://www.fourcc.org/codecs.php`. It should be taken into account that the supported codecs are platform-dependent. This means that if you want to work with a specific codec, this codec should already be installed on your system. Typical codecs are DIVX, XVID, X264, and MJPG.

Additionally, a video file format is a type of file format that's used to store digital video data. Typical video file formats are AVI (`*.avi`), MP4 (`*.mp4`), QuickTime (`*.mov`), and Windows Media Video (`*.wmv`).

Finally, it should be taken into account that the right combination between video file formats (for example, `*.avi`) and FOURCC (for example, DIVX) is not straightforward. You will probably need to experiment and play with these values. Therefore, when creating a video file in OpenCV, you will have to take all of these factors into consideration.

The following diagram tries to summarize them:



This diagram summarizes the main considerations you should take into account when creating a video file using `cv2.VideoWriter()` in OpenCV. In this diagram, the `video_demo.avi` video has been created. In this case, the FOURCC value is `XVID` and the video file format is `AVI` (`*.avi`). Finally, both the fps and the dimensions of every frame of the video should be established.

Additionally, the following example, `write_video_file.py`, writes a video file and it can also be helpful to play with these concepts. Some key points of this example are commented here. In this example, the required argument is the video file name (for example, `video_demo.avi`):

```
# We first create the ArgumentParser object
# The created object 'parser' will have the necessary information
# to parse the command-line arguments into data types.
parser = argparse.ArgumentParser()

# We add 'output_video_path' argument using add_argument() including a
help.
parser.add_argument("output_video_path", help="path to the video file to
write")
args = parser.parse_args()
```

We are going to take frames from the first camera that's connected to our computer. Therefore, we create the object accordingly:

```
# Create a VideoCapture object and pass 0 as argument to read from the
camera
capture = cv2.VideoCapture(0)
```

Next, we will get some properties from the capture object (frame width, frame height, and fps). We are going to use them to create our video file:

```
# Get some properties of VideoCapture (frame width, frame height and frames
per second (fps)):
frame_width = capture.get(cv2.CAP_PROP_FRAME_WIDTH)
frame_height = capture.get(cv2.CAP_PROP_FRAME_HEIGHT)
fps = capture.get(cv2.CAP_PROP_FPS)
```

Now, we specify the video codec using FOURCC, a four-byte code. Remember, it is platform-dependent. In this case, we define the codec as `XVID`:

```
# FourCC is a 4-byte code used to specify the video codec and it is
platform dependent!
# In this case, define the codec XVID
fourcc = cv2.VideoWriter_fourcc('X', 'V', 'I', 'D')
```

The following line also works:

```
# FourCC is a 4-byte code used to specify the video codec and it is
platform dependent!
# In this case, define the codec XVID
fourcc = cv2.VideoWriter_fourcc(*'XVID')
```

Then, we create the `cv2.VideoWriter` object, `out_gray`. We use the same properties as the input camera. The last argument is `False` so that we can write the video in grayscale. If we want to create the video in color, this last argument should be `True`:

```
# Create VideoWriter object. We use the same properties as the input
camera.
# Last argument is False to write the video in grayscale. True otherwise
(write the video in color)
out_gray = cv2.VideoWriter(args.output_video_path, fourcc, int(fps),
(int(frame_width), int(frame_height)), False)
```

We get frame by frame output from the capture object by using `capture.read()`. Each frame is converted into grayscale and written to the video file. We can show the frame, but this is not necessary to write the video. If *q* is pressed, the program ends:

```
# Read until video is completed or 'q' is pressed
while capture.isOpened():
    # Read the frame from the camera
    ret, frame = capture.read()
    if ret is True:

        # Convert the frame to grayscale
        gray_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Write the grayscale frame to the video
        out_gray.write(gray_frame)

        # We show the frame (this is not necessary to write the video)
        # But we show it until 'q' is pressed
        cv2.imshow('gray', gray_frame)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
    else:
        break
```

Finally, we release everything (the `cv2.VideoCapture` and `cv2.VideWriter` objects, and we destroy the created windows):

```
# Release everything:
capture.release()
out_gray.release()
cv2.destroyAllWindows()
```

The full code for this example can be seen in the `write_video_file.py` file.

Playing with video capture properties

In some of the previous examples, we saw how to get some properties from the `cv2.VideoCapture` object. In this section, we are going to see how we can get all of the properties and understand how they work. Finally, we are going to use these properties to load a video file and output it backwards (showing the last frame of the video first and so on).

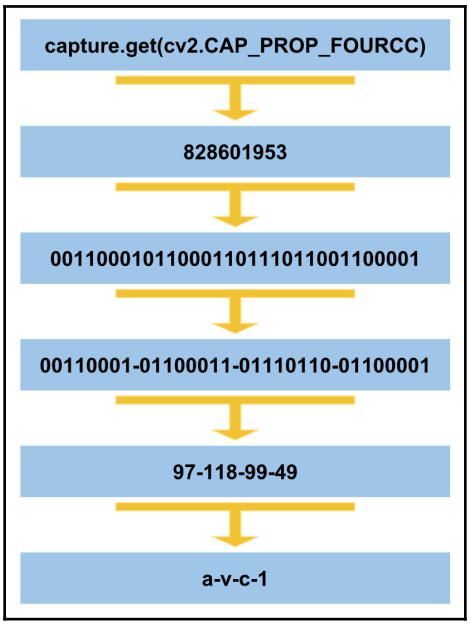
Getting all the properties from the video capture object

First, we create the `read_video_file_all_properties.py` script to show all the properties. Some of these properties only work when we're working with cameras (not with video files). In these cases, a `0` value is returned. Additionally, we have created the `decode_fourcc()` function, which converts the value that's returned by `capture.get(cv2.CAP_PROP_FOURCC)` as a string value that contains the int representation of the codec. In this sense, this value should be converted into a four-byte char representation to output the codec properly. Therefore, the `decode_fourcc()` function copes with this.

The code of this function is given as follows:

```
def decode_fourcc(fourcc):
    """Decodes the fourcc value to get the four chars identifying it

    """
    fourcc_int = int(fourcc)

    # We print the int value of fourcc
    print("int value of fourcc: '{}'".format(fourcc_int))

    # We can also perform this in one line:
    # return "".join([chr((fourcc_int >> 8 * i) & 0xFF) for i in range(4)])

    fourcc_decode = ""
    for i in range(4):
        int_value = fourcc_int >> 8 * i & 0xFF
        print("int_value: '{}'".format(int_value))
        fourcc_decode += chr(int_value)
    return fourcc_decode
```

To explain how it works, the following diagram summarizes the main steps:



As you can see, the first step is to obtain the int representation of the value that's returned by `capture.get(cv2.CAP_PROP_FOURCC)`, which is a string. Then, we iterate four times to get every eight bits and convert these eight bits into `int`. Finally, these `int` values are converted into `char` using the `chr()` function. It should be noted that we can perform this function in only one line of code, as follows:

```
return "".join([chr((fourcc_int >> 8 * i) & 0xFF) for i in range(4)])
```

The `CAP_PROP_POS_FRAMES` property gives you the current frame of the video file and the `CAP_PROP_POS_MSEC` property gives you the timestamp of the current frame. We can also get the number of fps with the `CAP_PROP_FPS` property.
The `CAP_PROP_FRAME_COUNT` property gives you the total number of frames of the video file.

To get and print all the properties, use the following code:

```
# Get and print these values:
print("CV_CAP_PROP_FRAME_WIDTH:
'{}'".format(capture.get(cv2.CAP_PROP_FRAME_WIDTH)))
print("CV_CAP_PROP_FRAME_HEIGHT :
'{}'".format(capture.get(cv2.CAP_PROP_FRAME_HEIGHT)))
print("CAP_PROP_FPS : '{}'".format(capture.get(cv2.CAP_PROP_FPS)))
```

```
print("CAP_PROP_POS_MSEC :
'{}'".format(capture.get(cv2.CAP_PROP_POS_MSEC)))
print("CAP_PROP_POS_FRAMES :
'{}'".format(capture.get(cv2.CAP_PROP_POS_FRAMES)))
print("CAP_PROP_FOURCC  :
'{}'".format(decode_fourcc(capture.get(cv2.CAP_PROP_FOURCC))))
print("CAP_PROP_FRAME_COUNT  :
'{}'".format(capture.get(cv2.CAP_PROP_FRAME_COUNT)))
print("CAP_PROP_MODE : '{}'".format(capture.get(cv2.CAP_PROP_MODE)))
print("CAP_PROP_BRIGHTNESS :
'{}'".format(capture.get(cv2.CAP_PROP_BRIGHTNESS)))
print("CAP_PROP_CONTRAST :
'{}'".format(capture.get(cv2.CAP_PROP_CONTRAST)))
print("CAP_PROP_SATURATION :
'{}'".format(capture.get(cv2.CAP_PROP_SATURATION)))
print("CAP_PROP_HUE : '{}'".format(capture.get(cv2.CAP_PROP_HUE)))
print("CAP_PROP_GAIN  : '{}'".format(capture.get(cv2.CAP_PROP_GAIN)))
print("CAP_PROP_EXPOSURE :
'{}'".format(capture.get(cv2.CAP_PROP_EXPOSURE)))
print("CAP_PROP_CONVERT_RGB :
'{}'".format(capture.get(cv2.CAP_PROP_CONVERT_RGB)))
print("CAP_PROP_RECTIFICATION :
'{}'".format(capture.get(cv2.CAP_PROP_RECTIFICATION)))
print("CAP_PROP_ISO_SPEED :
'{}'".format(capture.get(cv2.CAP_PROP_ISO_SPEED)))
print("CAP_PROP_BUFFERSIZE :
'{}'".format(capture.get(cv2.CAP_PROP_BUFFERSIZE)))
```

You can view the full code of this script in the `read_video_file_all_properties.py` file.

Using the properties – playing a video backwards

To see how we can use the aforementioned properties, we are going to understand the `read_video_file_backwards.py` script, which uses some of these properties to load a video and output it backwards, showing the last frame of the video first and so on. We are going to use the following properties:

- `cv2.CAP_PROP_FRAME_COUNT`: This property provides the total number of frames
- `cv2.CAP_PROP_POS_FRAMES`: This property provides the current frame

The first step is to get the index of the last frame:

```
# We get the index of the last frame of the video file
frame_index = capture.get(cv2.CAP_PROP_FRAME_COUNT) - 1
```

Therefore, we set the current frame to read to this position:

```
# We set the current frame position
capture.set(cv2.CAP_PROP_POS_FRAMES, frame_index)
```

This way, we can read this frame as usual:

```
# Capture frame-by-frame from the video file
ret, frame = capture.read()
```

Finally, we decrement the index in order to read the next frame from the video file:

```
# Decrement the index to read next frame
frame_index = frame_index - 1
```

The full code is provided in the `read_video_file_backwards.py` script. This script can be easily modified to save the resulting video playing backwards (not only showing it). This script is proposed in the *Question* section.

Summary

In this chapter, we saw that working with images and files is a key element of computer vision projects. A common approach in this kind of project is to load some images first, perform some processing, and then output the processed images. In this chapter, we reviewed this flow. Additionally, in connection with video streams, both `cv2.VideoCapture` and `cv2.VideoWriter` were covered. We also looked at the `cv2.VideoWriter` class for video writing. Two key aspects were reviewed when writing video files—video codecs (for example, DIVX) and video file formats (for example, AVI). To work with video codecs, OpenCV provides FOURCC, a four-byte code. Typical codecs are DIVX, XVID, X264, and MJPG, while typical video file formats are AVI (`*.avi`), MP4 (`*.mp4`), QuickTime (`*.mov`), and Windows Media Video (`*.wmv`).

We also reviewed the concept of fps and how to calculate it in our programs. Additionally, we looked at how to get all the properties of the `cv2.VideoCapture` object and how to use them to load a video and output it backwards, showing the last frame of the video first. Finally, we saw how to cope with command-line arguments. Python uses `sys.argv` to handle command-line arguments. When our programs take complex parameters or multiple filenames, we should use Python's argparse library.

In the next chapter, we are going to learn how to draw basic and more advanced shapes using the OpenCV library. OpenCV provides functions to draw lines, circles, rectangles, ellipses, text, and polylines. In connection with computer vision projects, it is a common approach to draw basic shapes in the image in order to do the following:

- Show some intermediate results of your algorithm (for example, bounding box of the detected objects)
- Show the final results of your algorithm (for example, the class of the detected objects, such as cars, cats, or dogs)
- Show some debugging information (for example, execution time)

Therefore, the next chapter can be of great help in connection with your computer vision algorithms.

Questions

1. What is `sys.argv[1]`?
2. Write a piece of code to add a `first_number` argument of the `int` type and include the help first number to be added using `parser.add_argument()`.
3. Write a piece of code to save the imagine `img` to disk with the name `image.png`.
4. Create the `capture` object using `cv2.VideoCapture()` to read from the first camera that's connected to your computer.
5. Create the object capture using `cv2.VideoCapture()` to read from the first camera connected to your computer and print the `CAP_PROP_FRAME_WIDTH` property.
6. Read an image and save it to disk with the same name but ending in `_copy.png` (for example, `logo_copy.png`).
7. Create a script (`read_video_file_backwards_save_video.py`) that loads a video file and creates another played backwards (containing the last frame of the video first and so on).

Further reading

The following references will help you dive deeper into argparse, which is a key point in your computer vision projects:

- *Parsing the command line with argparse* (`https://www.packtpub.com/mapt/book/application_development/9781783280971/16/ch16lvl1sec147/parsing-the-command-line-with-argparse`)
- *Using argparse to get command-line input* (`https://www.packtpub.com/mapt/book/application_development/9781786469250/5/ch05lvl1sec60/using-argparse-to-get-command-line-input`)