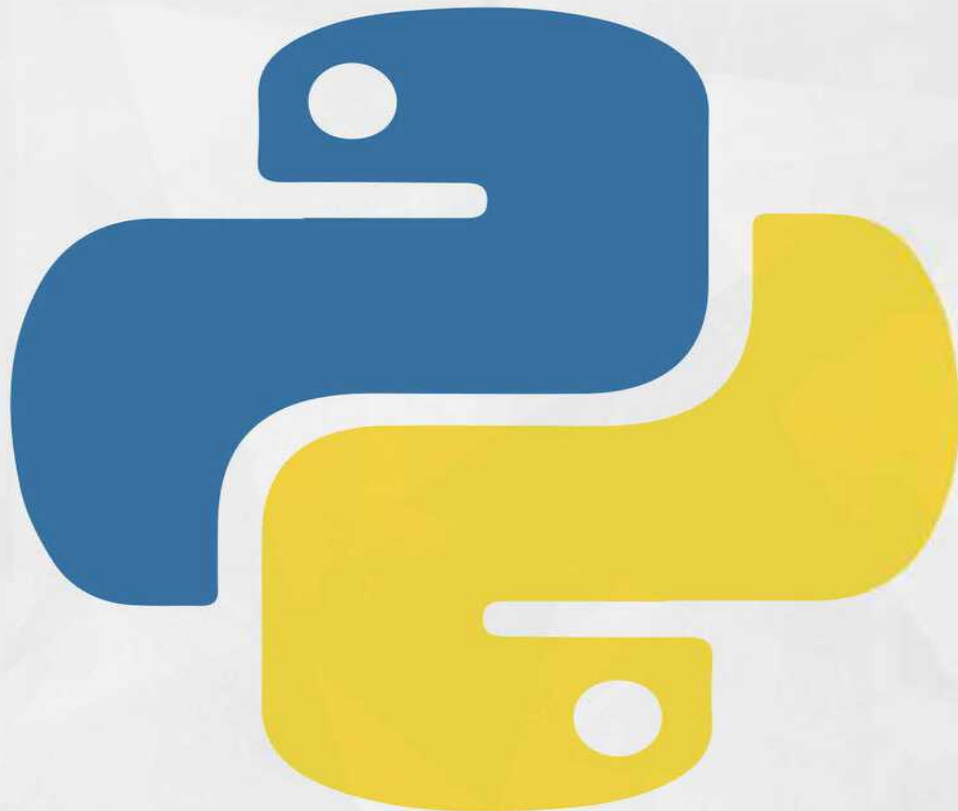


# PROGRAMMING FOR BEGINNERS

**3 MANUSCRIPTS**

**THE COMPLETE GUIDE TO LEARNING  
PYTHON CRASH COURSE,  
PYTHON MACHINE LEARNING AND  
PYTHON DATA SCIENCE IN A WEEK**

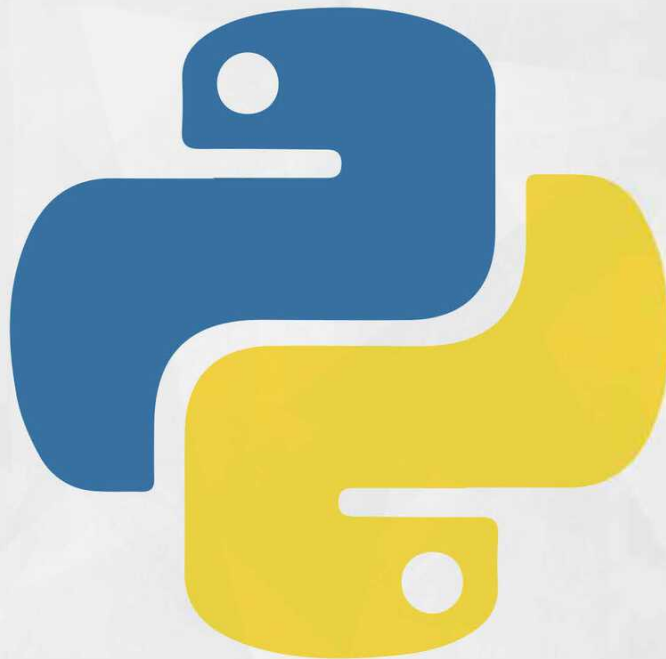


**ANDREW LEE**

# PROGRAMMING FOR BEGINNERS

3 MANUSCRIPTS

THE COMPLETE GUIDE TO LEARNING  
PYTHON CRASH COURSE,  
PYTHON MACHINE LEARNING AND  
PYTHON DATA SCIENCE IN A WEEK



ANDREW LEE

# ***Programming for Beginners: 3 Manuscripts:***

The Complete Guide to Learning Python Crash Course, Python  
Machine Learning and Python Data Science in a Week.

Andrew Lee

# **Copyright**

Copyright © 2021 Andrew Lee. All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, without the prior written permission of the publisher.

# TABLE OF CONTENT

[Copyright](#)

[Introduction](#)

[Understanding The Python Coding language](#)

[How Do Loops Work?](#)

[Writing An Exception In Your Code](#)

[Working With User-Defined Functions](#)

[Inheritances In The Python Code](#)

[Working With The Python Generators](#)

[The Classes And The Objects In Python](#)

[What Are The Operators, And How To Use Them?](#)

[The Variables in the Python Language](#)

[Troubleshooting A Python Program](#)

[Conclusion](#)

[Python Machine Learning for Beginners](#)

[The Different Types of Machine Learning](#)

[Python Ecosystem for Machine Learning](#)

[Getting Familiar with Python and SciPy](#)

[Understand Your Data With Descriptive Statistics](#)

[Understand Your Data With Visualization](#)

[Preparing Your Data For Machine Learning](#)

[Real-World Applications of Machine Learning](#)

[Using Tree-Based Algorithms for Advertising Click-Through Prediction](#)

[Conclusion](#)

[Python Data Science](#)

[Descriptive Statistics](#)

[Data Analysis and Libraries](#)

[NumPy Arrays and Vectorized Computation](#)

[Data Analysis with Pandas](#)

[Data Visualization](#)

[Data Mining](#)

[Classifying with Scikit-learn Estimators](#)

[Giving Computers the Ability to Learn from Data](#)

[Training Machine Learning Algorithms](#)

[Conclusion](#)

***Python***  
***Crash Course***  
***for Beginners***

Andrew Lee

# Introduction

It is easy to learn and much more powerful than other languages because of its dynamic nature and simple syntax, which allows small lines of code.

Such advantages of Python make it different from other languages and that's why Python is preferred for development in a plethora of companies. Python can be used to process anything that can be saved on a computer like numbers, text, data, images, statistics etc. Its easy-to-use feature will keep programmers engaged and excited as they

begin to learn Python. It has become famous, with its characteristics like easy indentation, naming conventions, modularity etc. Python is widely used in the daily operations of Google, NASA, New York Stock Exchange and even on our favorite video sharing website, YouTube.

In industries, machine learning using Python has become popular. This is because it has standard libraries which are used for scientific and numerical calculations. Also, it can be used on Linux, Windows, Mac OS and UNIX. It's not used just by the industry big shots, however Python is extensively used even in business, government and non-government organizations too. To be able to delve deeper into programming, one needs to have a basic understanding of some topics so that they can achieve mastery from the area.

Python Crash Course For Beginners contains proven steps and strategies into learn Python Programming quickly and easily. It provides all essential programming concepts and information you need to start developing your own Python program. This book provides a comprehensive walk-through of Python programming in a clear, straightforward manner that even a beginner will appreciate.

You can use this book as a manual to help you explore, harness, and gain appreciation of the capabilities and features of Python.



Let's get started!

# Understanding The Python Coding language



Reading and writing code at Python is much like reading and writing regular English statements. Because they are not written in machine-readable language, Python programs need to be processed before machines can conduct them. This means that every time a program is conduct, its interpreter runs through the code and translates it into machine-readable byte code.

Users to manage and control data structures or objects to create and conduct programs.

Languages die and become obsolete when they fail to live up to expectations, and are afterward replaced and superseded by languages that are

more powerful. It came with several new features and enhancements, along with a amount of deprecated features. But, the new features of Python 3.0 made it more contemporary and popular. Many developers even switched into Version 3.0 of the programming language to avail themselves of these awesome features. Python 3.0 replaced print statement together with.

That the built-in `print()` function, while allowing programmers to use custom separators between lines. In case the operands are not organized in a natural and meaningful order, the ordering comparison operators can now raise a `TypeError` exception.

Version 3 of that the programming language further uses text and data instead of Unicode and 8-bit strings.

The instrument highlights incompatibility and areas of concern via comments and warnings. The comments help programmers to make changes to the code, and upgrade their existing applications to that the latest version of

that the programming language. Latest Versions of all Python Version 3.7.3 or even 2.7.16 of Python. Python 2.7 enables developers to access improved numeric handling and enhancements for that the standard library.

This version further makes it easier for developers to migrate to Python 3. On the other hand, Python 3.7 comes with several new features and library modules, security improvements, and CPython implementation improvements. However, a number of features are deprecated in both Python API and programming language.

Developers can still use Python 3.7 to avail themselves of support in the longer run. Version 4 of Python Python 4.0 is expected to be available in 2023, after the release of Python 3.9. It will come with features that will help programmers switch from version 3 to 4 seamlessly. Also, as they gain experience, the expert Python developers can take advantage of a number of backwards compatible features to modernize their existing applications without putting any extra time and effort. But, the developers still have to wait many years to get a clear picture of Python 4.0. However, they must monitor that the latest releases into easily migrate to Version 4.0 of that the popular coding language.

Version 2.x and Version 3.0 of Python are completely different from each other. So each programmer must understand the features of these distinct versions, and compare their functionality based on

specific needs of the project. Also, he or she needs to check the version of Python that each framework supports. However, each developer must take advantage of the latest version of Python to access new features and long-term support.

## Features of Python

A question that comes up is why machine learning using Python is preferred over other languages? This is because Python has some features over other programming languages. Here are some basic features of Python that make it better than other languages: ·

Python is a High-level language. It means the context of Python is user-friendly rather than machine language.

The interactive nature of Python makes it simple and attractive for users. Back in interactive mode, users are able to check the output for each statement.

As an Object Oriented Programming language, it allows reuse and recycling of programs.

The syntax of Python is extensible through many libraries. One of the highlights of Python is that it is a highly extensible language. This means that various functional element are not built into the core of this platform. Rather, you can use third party applications and extend the platform's functionality. Additionally, you can also integrate Python code into an existing program and create an interface for programming. This is called Embedding and Extending. As mentioned above, the syntax of Python is simple. Complicated syntax is rejected and the platform embraces codes that are less cluttered and sparse. However, this does not in any way influence the performance or functionality of programs. Also, unlike other popular programming languages such as Perl, Python does not offer unnecessary clutter by giving the programmer multiple ways of achieving the same purpose. Python's philosophy is to provide one powerful way of obtaining that the desired result. This philosophy is that the main driving

force supporting the simplicity of Python. So, if you want to become adept in this language, you need to change your mindset and think in a simple and straightforward manner. This approach towards programming works best with Python.

## **Benefits of Using the Python Coding language**

One of the most robust and dynamic

programming languages being used today is Python. It stresses code readability, and because of its syntax as nicely as implementation, programmers are able to write less complex codes if compared into Java and C++. Memory management in Python is done automatically, and several standard libraries are available for that the programmer here. After completing a certification course at Python training, a programmer can frequently gain entry into various top IT companies. Python programming supports numerous styles such as functional programming, imperative and object-oriented styles. Here are the top five reasons why a computer programmer must learn the Python language: ·

Ease of learning Python has been created with the newcomer in mind. Completion of basic tasks requires less code in Python, compared to other languages. The codes are usually 3-5 times shorter than Java, and 5-10 times smaller than C++. Python codes are easily readable and, with a small bit of knowledge, new developers can learn a lot by just looking at the code. Python is relatively easy to learn. Many find Python a good first language for learning programming because it uses simple syntax and shorter codes.

**Readability** Python programs use clear, simple, and concise instructions that are easy to read, even by those who have no substantial programming background. Programs written at Python are, therefore, easier to maintain, debug, or enhance. In order to aid simplicity, Python coding and syntax uses English words rather than punctuations or symbols. This enhances readability as nicely. Some examples of statements written in Python include "if", "for",

"while", "try", "class", "def", "with", "yield", "import" and many others. Most of the commands used are self-explanatory.

Highly preferred for net development Python consists of an array of frameworks that are useful in designing a website. Among these frameworks, Django is the most popular a single for Python development. Due to these frameworks, web designing with Python has immense flexibility. The number of websites online today is close to 1 billion, and with the ever-increasing scope for more, it is natural that Python programming will continue to be an important skill for web developers.

Considered ideal for start-ups Time and budget are vital constraints for any new product or service in a company, and more so if it is a startup. One can create a product that differentiates itself from the rest in any language. However, for quick development, less code and a smaller cost, Python is the ideal language here. Python can easily scale up any complex application and can also be handled by a small team. Not only do you save resources, however you also get to develop applications in the right direction with Python.

Unlimited availability of resources and testing framework Several resources for Python are available today, and these are also constantly being updated. As a result, it is very rare that a Python developer gets stuck. The vast standard library provides built in functionalities. Its built in testing framework enables speedy workflow and less debugging time.

Fat paycheques Today, top IT companies such as Google, Yahoo, IBM, and Nokia make use of Python. Among all programming languages, it has had amazing expansion over the last few years.

Higher productivity Codes used in Python are considerably. Shorter, simpler, and less verbose than other high end programming languages, such as Java and C++. Back in addition, it has well-designed built-in features and a standard library, as well as access to third party modules and source libraries. These features make programming at Python much more efficient.

Runs across different platforms Python works on Windows, Linux/UNIX, Mac OS X, other operating systems, and smallform devices. It also runs on microcontrollers used in appliances, toys, remote controls, embedded devices, and other similar devices. It is clear that Python is a vital language for web-based programmers.

## **Applications of Python**

There are a great deal of advantages of Python that make it different from others. Its applications have made it an in-demand language for software development, web development, graphic design, and other uses.

Its standard libraries support internet protocols such as HTML, JSON, XML, IMAP, FTP and many more. Libraries are able to support many operations like Data Scraping, NLP and additional applications of machine learning. Due into such advantages and uses, students are preferring into use that the Python programming tutorial rather than other languages. Also, there are many online video training courses available; the user or any interested candidate can buy them from any place. No need to worry about location, it can be learned from their home.

## **Python Frameworks and Their Real World Usage**

Python is extremely popular among the web Developers due into its unmatched quality and performance, along with a few highly efficient Python frameworks. Let's discuss that the top Python frameworks of the industry. Django framework This is probably the most prominent Python Web framework of all the industry. Django is extremely powerful and was developed by Jason Sole and Jason Mc Laugilin. It was implemented to get the first time in a job portal into ensure its efficiency. Later on, it was released for everyone and received an overwhelmingly positive response in the industry. This is the largest Python-based net framework at the industry and hence comes using extraordinary power and features for Python developers. There is a huge support community for this framework who are working 24x7 to provide support to others. It has the most powerful admin interface for top degree control of any large scale web application. Practically

speaking, this framework is extremely useful in developing online forums, portals and other social networking sites. It is also extremely popular in developing quick web solutions with maximum efficiency and minimum effort. Flask is a micro-framework developed in Python and has a strange background. It was the result of an April Fool surprise from an intelligent Python developer of India. Mr. Pradeep Gowda challenged his colleague to develop a single-file micro-framework in Python. He succeeded in performing just that and gifted it to his friend as an April Fool surprise. Well, this is not as powerful as that of Django however it can be used for moderate-sized Python web application development projects. It is most suitable for beginners who want to learn Python and start coding in a short time span. It creates an insatiable thirst among Python developers to experiment with the framework. It is often used in small web development that has a low budget and limited period frame. Hence, it has received good term of mouth from that the industry since its inception. Pyramid is yet another awesome Python development framework that has extraordinary flexibility to serve a wide range of web applications. It is also powered by GitHub, so that there is no more fear of all losing support. It is the combination of pylons 1.0 and repose.bfg. It is growing at a much faster pace in the community due to its flexible nature. Practically, it is helpful for developing enterprise standard API Python projects and creating Python based CMS or KMS.

## **Getting Python on Your System**

Whether you use a Mac, Windows, or Linux OS (operating system), you can find and install Python on your computer. The following sections give you instructions for each OS.

### **How to install Python on Mac OSX**

To find and start Python on Mac OSX computers, follow these steps:

- Press Cmd+spacebar to open Spotlight.
- Type the word terminal.



Or, from the Finder, select Finder → Go → Utilities → Terminal.

The Terminal window opens.

- In the terminal, type python.

The Python interpreter that's built in to Mac OSX opens.

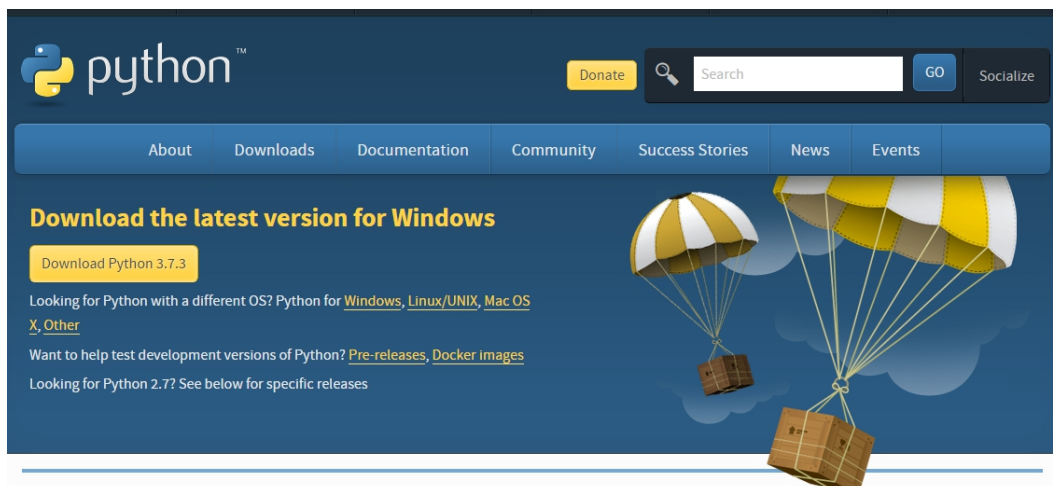
## How to install Python on Windows

Unfortunately, Python doesn't come on Windows. If you're running Windows, then you need to download and install Python by following the instructions here. Installing Python on Windows isn't difficult. If you can download a file from a website, you have the skills to install Python.

Fortunately, the Python Foundation (the peeps who guide the development of Python) makes installable files available from its website [www.python.org/downloads/](http://www.python.org/downloads/).

To install Python on a Windows machine, follow these steps:

- Visit [www.python.org/downloads/](http://www.python.org/downloads/).
- Click the button that says Download Python 3.7.3. Or, if it's there, click a more recent version number that starts with 3.7.



Clicking this button automatically downloads and saves the exe file for you.

When the download's complete, click the icon for download tool.

- Click the file called python-3.7.3.exe (or the more recent version, if you downloaded one).

Python 3.7.3 installs on your computer.

<p>{</p><div>

<a name="\_Toc62891891">Installing the Interpreter on Python</a>

|}<p></p><p>On Your desktop, click on the Start button, {then|subsequently} click on Run. </p>

<p>In {the Run window, enter the following text in the text box, then click on the OK

button. |} </p> {<p>C:\Python23\python.exe

|} </p>

<p>You Will {then|subsequently|afterward} see Python running in an MSDOS window (DOS shell) </p>

<p>You Will see a few lines of information followed by the Python prompt &gt;&gt;{&gt;. |p &gt; } A prompt is simply {the|that the} location where instructions are typed in (at the shell

command line). </p>

<p>After {typing in a Python statement at the prompt, you must press the Enter key (also

known as the Return key) on your keyboard. |} </p>

<p>At

The &gt;&gt;{&gt;|p} prompt, {try|attempt} typing in the following simple Python statement: </p>

<p>print

{"Hello World!" |} {(then press the Enter key). |} </p>

<p>Python

{responds by printing the text "Hello World!" } {(without the q uotes).

} </p>

<p>If

{you've got this far, you know that Python has been installed correctly! } </p>

<p>Notice

{that after Python interprets and executes your command, the prompt returns ready

for the next command (Python statement). } {(When running in this manner the Python

interpreter is said to be in the interactive mode, and will continue in this

mode indefinitely until you issue the command to q uit.

} </p>

<p>To

{exit from the Python interpreter you must hold down the Ctrl keyboard key and

press the Z key at the same time (Ctrl-Z). } {(Different operating systems may re q uire

a different key sequence, e.g. } {(Ctrl-C). } </p>

<a name="\_Toc62891892">Working with the Python Interpreter</a>

}<p>Before

{we begin, let's narrow down what we mean by "a Python interpreter".

The "interpreter" can be used in a variety of different ways

when discussing Python. Sometimes interpreter refers to the Python REPL, the

interactive prompt you get by typing python at the command {line|lineup}. {Sometimes people

use "the Python interpreter" more or less interchangeably with "Python"

to talk about executing Python code from start to finish. In this section, "interpreter" has a more narrow meaning: it's the last step in the

process of executing a Python program.

[Features of Python Interpreter](#)

Python {interpreter offers some pretty cool features:

3/4

Interactive editing

3/4

History substitution

3/4

{Code completion on systems with support

for readline

In

{the first Python prompt, try pressing the following keys:

Ctrl+P

This

{tells you if your interpreter supports command-line editing. |} {A beep indicates

that it does support command-line editing. |} {Otherwise, it will either perform a no-operation or echo ^p to indicate it isn't available. |}  
</p>

<p>Before

{the interpreter takes over, Python performs four other steps: lexing, parsing,

compiling, and interpreting. |} </p>

<p>3/4

{Lexing- The lexer breaks the line of code into tokens. |} </p>

<p>3/4

{Parsing- The parser uses these tokens

to generate a structure, here, an Abstract Syntax Tree, to depict the relationship between these tokens. |} </p>

<p>3/4

{Compiling- The compiler turns this AST into code object(s). |} </p>

<p>3/4

{Interpreting- The interpreter executes each code object. |} </p>

<p>Together,

{these steps transform the programmer's source code from lines of text into

structured code objects containing instructions that the interpreter can

understand. } {The interpreter's job is to take these code objects and follow the

instructions. } </p>

<p>You

May be surprised {to|into} hear that compiling is a step in executing Python code at

all. Python is {often|frequently} called an"interpreted" language like Ruby or Perl,

as opposed to a"compiled" language like C or Rust. {However, this

terminology isn't as precise as it may seem. } {Most interpreted languages,

including Python, do involve a compilation step. } {The reason Python is called

"interpreted" is that the compilation step does relatively less work

(and the interpreter does relatively more) than in a compiled language. } </p>

<p>Since

{Python has shown its enormous applications and use cases. } {It is mostly used in

Machine Learning and Artificial intelligence companies as a basic programming

language. } {Students who want to start their career in AI and machine learning

should have a basic understanding of Python. } {Further, it is an easy programming

language to learn as a beginner. } {It can be learned quickly

because user can think like a programmer due to its readable and understandable

syntax. } {With Python we can develop anything by computer programs, only need is

to spend time to understand Python and its standard libraries. } {PyCharm is its

IDE which makes interface so easy and comfortable while learning. } {With the help

of debugging feature of PyCharm we can easily analyse the output of each line

and the error can be detected easily. } {</p>

<br>

<a name="\_Toc62891894">The Python Code Basics</a>

}<p>Python

{code basics refers to the set of rules that defines how human users and the system

should write and interpret a Python program. } {If you want to write and run your

program in Python, you must familiarize yourself with its basics. } {For delving

deeper into the programming part, one needs to have a basic understanding of

some topics so that they can reach a mastery over programming. } {Some of the topics

re q uired for this are inclusive of the

following. } {</p>

<a name="\_Toc62891895">Types of Data</a>

Python

Python supports numerous kinds of data types, these describe the operations that can

be performed on the variables as well as the storage method. The different

types of data types are inclusive of sets, numbers, Booleans, date, time,

dictionary, strings as well as lists.

The Keywords

Python

Keywords are reserved words in Python that should not be used as variable,

constant, function name, or identifier in your code. Take note of these

keywords if you don't want to encounter errors when you execute your program:

break

continue

del else

exec



</p>

<p>for </p><p>global

</p>

<p>import

</p>

<p>is </p><p>not </p><p>pass

</p>

<p>raise

</p>

<p>try </p><p>with

</p>

<p>assert

</p>

<p>class

</p>

<p>def </p><p>elif

</p>

<p>except

</p>

<p>finally

</p>

<p>from

</p>

<p>if </p><p>in </p><p>lambda

</p>

<p>or </p><p>print

</p>

<p>return

</p>

<p>while

</p>

<p>yield

{</p>

</div>

<br>

<a name="\_Toc62891897">The Importance Of The Identifiers</a>

|}<p>A Python

Identifier is a name given to a function, class, variable, module, or {other

|alternative }objects that you'll be using in {your|your own} Python program. {Any entity you'll be using

in Python should be appropriately named or identified as they will form part of

your program. |} </p>

<p>Here

{are Python naming conventions that you should be aware of: </p>

|}<p>An

{identifier can be a combination of uppercase letters, lowercase letters,

underscores, and digits (0-9). } {Hence, the following are valid identifiers: myClass,

my\_variable, var\_1, and print\_hello\_world. } </p>

<p>Special

{characters such as %, @, and \$ are not allowed within identifiers. } {An

identifier should not begin with a number. } Hence, 2variable is not valid, {but

|however }variable2 is acceptable. </p>

<p>Python

{is a case-sensitive language and this behavior extends to identifiers. } Thus,

Labor and labor are {two|2} distinct identifiers in Python. </p>

<p>You

{cannot use Python keywords as identifiers. } </p>

<p>Class

Identifiers begin with an uppercase letter, but the rest of {the|that the} identifiers

begin in lowercase. </p>

<p>You can

{use underscores to separate multiple words in your identifier. } </p>

<p>You

{should always choose identifiers that will make sense to you even after a long

gap. |} Hence, while it is easy to set your variable to `c = {2|two}`, you might find it

more helpful for future reference if you use a longer {but|however} more relevant

variable name such as `count = {2|two}`. {</p>

<a name="\_Toc62891898">Using Quotations</a>

|}<p>Python

{allows the use of q uotation marks to indicate string

literals. |} You can use single, double, or triple q uotes

{but|however} you must start and end the string with {the|that the} same type. {You would use the

triple q uotes when your string runs across

several lines. |} {</p>

<a name="\_Toc62891899">The Statements</a>

|}<p>Statements

{are instructions that a Python interpreter can execute. |} {When you assign a value

to a variable, say `my_variable = "dog"`, you're making an assignment statement. |} {An

assignment statement may also be as short as `c = 3`. |} {There are other kinds of statements

in Python, like conditional statements, while statements, for statements, etc. |} {</p>

<h3><a name="\_Toc62891900">Conditional statements</a>

</h3>

Conditional

statements are the type of statements that assist in the carrying out of a set

of standards. All of these sets of standards are based upon a certain

condition. There are about three conditional statements, these are inclusive of

Else, If and Elif (We'll discuss this in detail in the next chapter).

</p>

<h3><a name="\_Toc62891901">Multi-line statements</a>

</h3>

A statement

may span over several lines. To break a long statement over multiple lines, you

can wrap the expression inside parentheses, braces, and brackets. This is the

preferred style for handling multi-line expressions. Another way to wrap

multiple lines is by using a backslash (\) at the end of every line to indicate

line continuation. </p>

<a name="\_Toc62891902">The Comments</a>

<p>When

writing a program, you'll find it helpful to put some notes within your code to

describe what it does. A comment is very handy when you have to review or

revisit your program. It will also help another programmer who might need to go

over the source code. You can write comments within your program by starting

the line with a hash (#) symbol. A hash symbol tells the Python interpreter to

ignore the comment when running your code. </p>

<p>For

multi-line comments, you can use a hash symbol at the beginning of each line. Alternatively,

you can also wrap multi-line comment with triple quotes. </p>

<a name="\_Toc62891903">Working with The Control Flow</a>

<p>A

Program's control flow is the {order|dictate} in which {the|that the} program's code executes. The

control flow of a Python program is regulated by conditional statements, loops,

and function calls. This section covers the if statement and for and while

loops. Raising and handling exceptions also affects control flow. </p>

<h3><a name="\_Toc62891904">The if Statement</a>

</h3>

<p>Often,

you need to execute some statements only if some condition holds, or choose statements

to execute depending on several mutually exclusive conditions. The Python

compound statement `if`, which uses `if`, `elif`, and `else` clauses, lets you conditionally execute blocks of statements. Here's the syntax for {the|that the} `if`

statement:

`if` *expression*:

*expression*:

`elif`

*expression*:

`statement(s)` `...` `else:`

*statement(s)* The `elif` and `else` clauses

are optional. {Note|Notice} that unlike some languages, Python does not have a `switch` statement,

so you must use `if`, `elif`, and `else` for all conditional processing.

Here's

a typical `if` statement:

`if` `x`

< 0: `print "x is negative"`

`elif`

`x % 2: print "x is positive and {odd|strange|peculiar}"`

`else:`

`Print "x is {even|also} and non-negative"`

<p>When

There are multiple statements in a clause (i.e., the clause controls a block of

statements), the statements are placed on separate logical lines after the line

containing the clause's keyword (known as the header {line|lineup} of the clause) and

indented rightward from the header line. The block terminates {when|if} the

indentation returns to that of the clause header (or further left from there).

When there is just a single simple statement, as here, it can follow the: on

the same logical line as the header, but it can also be placed on a separate

logical line, immediately after the header {line|lineup} and indented rightward from it.

Many Python practitioners consider the separate-line style more readable:</p>

<p>if x

&lt; 0:</p>

<p>print "x is negative"</p><p>elif

x % 2:</p>

<p>print "x is positive and odd"</p><p>else:</p><p>print "x is even and non-negative"</p><p>You

can use any Python expression as the condition in an if or elif clause. When



you use an expression this way, you are using it in a Boolean context. In a

Boolean context, any value is taken as either possibly true or false. As we discussed

earlier, any non-zero number or non-empty string, tuple, list, or dictionary

evaluates as true. Zero (of any numeric type), None, and empty strings, tuples,

lists, and dictionaries evaluate as false. When you want to test a value x in a

Boolean context, use the following coding style:

```
if x:
```

 This

is the clearest and most Pythonic form. Don't use:

```
if x
```

```
is True:
```

```
if x
```

```
= {} True:
```

```
if
```

```
bool(x):
```

There

is a crucial difference between saying that an expression "returns True"

(meaning the expression returns the value 1 intended as a Boolean result) and

saying that an expression "evaluates as true" (meaning the expression returns any

result that is true in a Boolean context). When testing an expression, you care

about the latter condition, not the former. </p>

<p>If

The expression for {the|that the} if clause evaluates as true, the statements following

the {if|should} clause execute, and the entire if statement ends. Otherwise, the expressions

for any elif clauses are evaluated in order. The statements following the first

elif clause whose condition is {true|accurate|authentic}, if any, are executed, and {the|that the} entire if

statement ends. Otherwise, if an else clause exists, the statements following

it are executed. </p>

<h3><a name="\_Toc62891905">The while Statement</a>

</h3>

<p>The

While statement {in|at} Python supports repeated execution of a statement or block

of statements that is controlled by a conditional expression. Here's the syntax

for the while statement:</p>

<p>while

<i>expression</i>:</p>

<p><i>statement(s)</i></p><p>A while statement

can also include an else clause and break and continue statements, as we'll discuss shortly. </p>

<p>Here's

a typical while statement:</p>

<p>count

= 0</p>

<p>while

x > 0:</p>

<p>{X|X} = x // {2            |two} # truncating division</p><p>count += 1</p><p>print

"The approximate log2 is", count</p>

<p>First,

expression, which is known as the loop condition, is evaluated. {If|In case} the

condition is false, the while statement ends. {If|When} the loop condition is

satisfied, the statement or statements that comprise {the|that the} loop {body|figure} are executed.

When the loop body finishes executing, the loop condition is evaluated again,

to see if another iteration should be performed. This process continues until

the loop condition is false, at which point the while statement ends.</p>

<p>The

Loop body should contain code that eventually makes {the|that the} loop condition false,

or the loop will never end unless an exception is raised or the loop body

executes a break statement. A loop that is in a function's body also ends if a

return statement executes in the loop body, as the whole function ends in this

case. </p>

<h3><a name="\_Toc62891906">The for Statement</a>

</h3>

<p>The

for statement in Python supports repeated execution of a statement or block of

statements that is controlled by an iterable expression. Here's the syntax for

{the|your} for statement:</p>

<p>{For|To get} <i>target</i>

{In|At} <i>iterable</i>:</p>

<p><i>statement(s)</i></p><p>{Note

|Notice }

That the {in keyword|key word} is part of {the|this} syntax of {the|this} for statement

and is functionally unrelated {to|into} the in operator used {for|to get} membership

testing. A forstatement can also include an else clause and break and continuestatements,

as {we'll|we will} discuss shortly. </p>

<p>Here's

a typical for statement:</p>

<p>for

letter in "ciao":</p>

<p>Print"{give|provide} me a", letter,

"..."</p>

<p><i>iterable</i> may

be any Python expression suitable as an argument to built-in function iter,

which returns an iterator object (explained in detail in the next section).

target is normally an identifier that names the control variable of the loop;

the for statement successively rebinds this variable to each item of the

iterator, in order. The statement or statements that comprise the loopbody execute

once for each item in iterable (unless the loop ends because an exception is

raised or a break or return statement is executed). </p>

<p>A

target with multiple identifiers is also allowed, as with an unpacking assignment.

In this case, the iterator's items must then be sequences,

each with the same {length|span}, equal to the number of identifiers in {the

{that the }target. For example, when {d|d e} is a dictionary, this is a typical way to loop on

the items in {d|p}:

<p>for

{Key|Crucial|Primary}, value {in|at} d.items( ):

<p>If not {key|essential} or {not|maybe not} value: del d[key] #  
keep {only|just} {true|real} keys and values</p><p>The

items method returns a list of key/value pairs, so we can use a for loop with

two identifiers in the target to unpack each item into key and value.  
</p>

<p>If

the iterator has a mutable underlying object, that object must not be altered

while a for loop is in progress on it. For example, the previous example cannot

use iteritems instead of items. Iteritems returns an iterator whose underlying

object is d{,|e,|e} so therefore the loop body cannot mutate {d|d e} (by deld[key]). Items

returns a list, though, so {d|d e} is not the underlying object of the iterator and

the loop body can mutate d. </p>

<p>The control

variable may be rebound in the loop body, but is rebound again to the next item

in the iterator at the next iteration of the loop. The loop body does not

execute at all if {the|that the} iterator yields no items. {In|Back in} this case, the control

variable is not bound or rebound in any way by the for statement. {If|In case|Should} the

iterator yields at least one item, however, when the loop statement terminates,

the control variable remains bound to the last value to which {the|that the} loop statement

has bound it. The following code is thus correct, as long as some seq is

not empty:

```
<p>for
```

```
{X in X} some seq:
```

```
<p>process(x)</p><p>print
```

```
"Last item processed was", x</p>
```

### [Iterators](#)

An

iterator is any object *i* such that you can call *i*.next( ) without any arguments.

*i*.Next( ) returns the {next|second} {item|thing} of iterator *i*, or, when iterator *i* has no more

items, raises a StopIteration exception. When you {write|compose} a class, you can allow

instances of the class to be iterators by defining such a method next. Most

Iterators are built by implicit or explicit calls to the built-in function `iter`. Calling

a generator also returns an iterator, as we'll discuss later in this chapter. The `for` statement implicitly calls `iter` to get an iterator. The following statement:

```
for x in c:
```

```
    statement(s)
```

is equivalent to:

```
_temporary_iterator = iter(c)
```

```
while True:
```

```
    try: x = _temporary_iterator.next( )
```

```
    except StopIteration: break
```

```
    statement(s)
```

Thus, if `iter( c )` returns an iterator `i` such that `i.next( )` never raises `StopIteration` (an infinite iterator), the loop `for x in c:` never terminates (unless the statements in the loop body contain

break or {return|reunite} statements or propagate exceptions). `iter( c )`, in turn, calls

special method `c.__iter__( )` to obtain and {return|reunite} an iterator on `c`.

Iterators were first introduced in Python 2.2. In earlier versions, `for x in S:` required

`S` to be a sequence that was indexable with

progressively larger indices `{0, 1, ..., n-1}`, and raised an `IndexError` when indexed

with a too-large index. Thanks to iterators, the `for` statement can now be used



on a container that is not a sequence, such as a dictionary, as long as

the container is iterable (i.e., it defines an `__iter__` special method so that

function `iter` can accept {the|that the} container as the argument and return an iterator

on the container). Built-in functions that used to require

a sequence argument {now|today} also accept any

iterable. {</p><h3><a name="\_Toc62891908">range and xrange</a></h3>|}<p>Looping

Over a sequence of integers is a common task, so Python provides {built-in|built in} functions `range` and `xrange` {to

into }generate and {return|yield} integer sequences. The simplest, most idiomatic way {to|into} loop <i>n</i> times

in Python is:</p><p>for

{|}| in xrange(<i>n</i>):</p><p><i>statement(s)</i>|}</p><p>range(

{x } returns a list whose items are consecutive integers from 0(included) up to

x (excluded). } Range( x,y ) returns a list whose items are consecutive integers

{from|in} x (included) up to y (excluded). {The result is the empty list if x is

greater than or equal to y. } Range( {x|y},{y|y, y }step ) returns a

list of integers {from|in} x (included) up to y (excluded), such that {the|that the} difference

{between|involving} each two adjacent items {in|at} the list is step. {If|Should} step is less than 0, range

counts down from x to y. range returns the empty list when x is greater than or

equal to y and step is greater than 0, or when x is less than or equal

to y and step is less than 0. {If|In case} step equals

0, range raises an exception. </p><p>While

Range returns a normal list object, usable for all purposes, xrange returns a

special-purpose object, specifically intended to be used in iterations {like|such as} the

{for|to get} statement shown previously. Xrange consumes less memory than range {for|to get} this specific

use. Leaving aside memory consumption, you can use range {wherever|where} you could use

xrange. {</p><h3><a name="\_Toc62891909">List comprehensions</a>

</h3>|}<p>A

Common use of a for loop is to inspect each item in a sequence

and build a new list by appending {the|that the} results of an expression computed on some

or all of the items inspected. The expression {form|type}, called a list comprehension, lets {you|that you} code this common idiom concisely and directly. Since a

list comprehension is an expression (rather than a block of statements), you

can use it directly {wherever|where} you need an expression (e.g., as an actual argument

in a function call, in a {return|yield} statement, or as a subexpression for some other

expression). </p><p>A

List comprehension has {the|that the} following syntax:</p><p>[<i>expression</i> {for|to get} <i>target</i> {in|at} <i>iterable</i>

</p>{<p><i>lc-clauses</i> }]</p>{<p><i>target</i> and <i>iterable</i> are

}

The same as {in|from|at} a regular {for|to get} statement. You must enclose {the|that the} <i>expression</i> {in

|at } parentheses {if|in case} it indicates a tuple. </p>{<p><i>lc-clauses</i> is

}

A series of {zero or }more clauses, each with {one|a few} of {the following|these} forms:</p><p>{For|To get} <i>target</i>

{In|At} <i>iterable</i></p><p>{If|In case|When} <i>expression</i></p><p><i>Target</i> and <i>iterable</i> {in

|at }

Each {for|to get} clause of a list comprehension have {the|that the} same syntax as those

{in|at} a regular for statement, and {the|that the} <i>expression</i> {in|at} each if clause

of a list comprehension has {the|that the} same syntax as the <i>expression</i> {in

|at }a regular ifstatement. </p><p>A list

Comprehension is equivalent {to|into} a {for|to get} loop that builds {the|that the} same list

by repeated calls to the resulting list's append method. {For example

(assigning the list comprehension result to a variable for clarity):  
result1

result1 is

the same as the for loop:  
result2

result2 is

the same as the for loop:  
result2.append(x+1)  
Here's

A list comprehension that uses an if clause:  
result3

result3 is the same as a for loop that contains an if statement:  
result4

result4 is the same as a for loop that contains an if statement:  
result4

result4 is

the same as the for loop:  
result4.append(x+1)  
And

here's a list comprehension that uses a for clause:  
result5

result5 is the same as a for loop with another for loop nested inside:  
result6

result6 is the same as a for loop with another for loop nested inside:  
result6

result6 is

the same as the for loop:  
result6.append(x+y)  
As

these examples show, the order of for and if in a list comprehension is the same as in the equivalent loop, but in the list comprehension the nesting

stays implicit. 

### The break Statement

The

{break statement is allowed only inside a loop body. } {When break executes, the

loop terminates. } {If a loop is nested inside other loops, break terminates only

the innermost nested loop. } {In practical use, a break statement is usually

inside some clause of an if statement in the loop body so that it executes conditionally.

} </p><p>One common

{use of break is in the implementation of a loop that decides if it should keep

looping only in the middle of each loop iteration:</p>}<p>while

```
{True:          # this loop can never terminate naturally</p>}<p>x = get_next( )</p><p>y = preprocess(x)</p><p>if not keep_looping(x, y): break</p><p>process(x, y)</p><h3><a name="_Toc62891911">The continue Statement</a></h3>}<p>The continue
```

{statement is allowed only inside a loop body. } {When continue executes, the current

iteration of the loop body terminates, and execution continues with the next

iteration of the loop. } {In practical use, a continue statement is usually inside

some clause of an if statement in the loop body so that it executes conditionally.

} </p><p>The

{continue statement can be used in place of deeply nested if statements within a

loop. } {For example:</p>|}<p>for

{x in some\_container:</p>|}<p>if not seems\_ok(x): continue|}</p>  
{<p>lowbound, highbound = bounds\_to\_test( )|}</p>{<p>if  
x<lowbound or="" x="">=highbound: continue|}</lowbound></p>  
{<p>if final\_check(x):|}</p>{<p>do\_processing(x)|}</p><p>This

{equivalent code does conditional processing without continue:</p>|}  
<p>for

{x in some\_container:</p>|}<p>if seems\_ok(x):|}</p>{<p>lowbound,  
highbound = bounds\_to\_test( )|}</p>{<p>if  
lowbound<=x<highbound:|}<p="">{</highbound:|}<></p><p>if  
final\_check(x):|}</p>{<p>do\_processing(x)|}</p><p>Both

{versions function identically, so which one you use is a matter of  
personal preference. } {</p><h3><a name="\_Toc62891912">The  
else Clause on Loop Statements</a></h3>|}<p>Both

{the while and for statements may optionally have a trailing else  
clause. } {The

statement or statements after the else execute when the loop  
terminates naturally

(at the end of the for iterator or when the while loop condition  
becomes

false), but not when the loop terminates prematurely (via break,  
return, or an

exception). } {When a loop contains one or more break statements,  
you often need

to check whether the loop terminates naturally or prematurely. }  
{You can use an

else clause on the loop for this purpose:</p>|}<p>for

{x in some\_container: } { if is\_ok(x): break # item x is satisfactory, terminate

}

loop else: Print "Warning: {no more} satisfactory {item|thing} was

{Found|Located} in container" x = None {<h3><a name="\_Toc62891913">The pass Statement</a></h3>|} The

{body of a Python compound statement cannot be empty--it must contain at least

one statement. } {The pass statement, which performs no action, can be used

as a placeholder when a statement is syntactically required but you have

nothing specific to do. } Here's an example of using pass {in|at} a conditional

statement as a part of somewhat convoluted logic, with mutually exclusive conditions

{being|becoming} tested: {If|In case} condition1(x): process1(x)|} elif

{X>X }23 or condition2(x) and x<5: Pass # nothing {to|more to|else to} be

{Done|Completed} in this case elif

{condition3(x): } { process3(x)|} else: process\_default(x)|} {<h3><a name="\_Toc62891914">The {try|attempt} Statement</a></h3>|} Python

Supports exception handling {with|together with|using} {the|all the} {try|attempt} statement, which includes try, except, finally,

and else clauses. A program can explicitly raise an exception {with

using together with the raise statement. When an exception is raised, normal control flow

of the program stops and Python looks for a suitable exception handler.

[The Variables](#)

A Python

variable is a reserved memory location to store values. In other words, a

variable in a python program gives data to the computer for processing. Every

value in Python has a datatype. Different data types in Python are Numbers, List,

Tuple, Strings, Dictionary, etc. Variables can be declared by any name or even

alphabets like a, aa, abc, etc.

[Indentation](#)

While

most programming languages such as Java, C, and C++ use braces to denote blocks

of code, Python programs are structured through indentation. In Python, blocks

of codes are defined by indentation not as a matter of style or preference but

however as a rigid language requirement. This principle makes Python



codes more readable and understandable. }

{block of code can be easily identified when you look at a Python program as

they start on the same distance to the right. } {If|In case|When} it has to be more deeply nestled,

you can simply indent another block further to the right. {For example, here is a

segment of a program defining car\_rental\_cost: </p>|}{<p>def car\_rental\_cost(days):

|}

Cost = 35 \* days if days {&gt;|p }= 8: </p><p>cost

--= 70 elif days {&gt;|p }= 3: </p><p>cost

--= 20 </p><p>return

cost </p><p>You

have to make sure that the indent space is consistent within a block. {When|If} you

use IDLE and {other|additional} IDEs to input your codes, Python intuitively provides

indentation on the subsequent line when you {enter|input} a statement

that requires indentation. Indentation, by

convention, is equivalent to 4 spaces to the right. </p><p>

<a name="\_Toc62891917">Classes and Objects</a>

</p><p>Python is an object oriented programming language. </p>

<p>Almost everything {in|at} Python is an object, with its properties

and methods. </p><p>Objects are an encapsulation of variables and functions into

a single entity. </p><p>A Class is like an object constructor, or a "blueprint" {for|to get} creating objects. Objects get {their|their own} variables and functions

{from|out of} classes. Classes are essentially a template to create your objects. </p><p>

<a name="\_Toc62891918">The Operators</a>

</p><p>With the help of operators, one can push around the values of the operands. Python comprises of a list of operators, they are inclusive of

{- |}Logical, Arithmetic, Identity, Membership, Bitwise, Assignment, as {well|nicely} as Comparison.

</p><p>

<br>

</p><p>

<a name="\_Toc62891919">Working With Conditional Statement</a>

</p><p>Conditional

statements are common among programming languages and they are used to perform

actions or calculations based on whether a condition is evaluated as true or

false. In programming languages, most of the {time|timing} we have to control the flow

of execution of your program, you want to execute some set of statements {only

just }if the given condition is satisfied, and a different set of statements when it's

not satisfied. Which we also call it as control statements or decision making

statements. </p><p>Conditional

statements are also known as decision-making statements. We use these statements

when we want to execute a block of code when the given condition is true or

false. </p><p>In Python

we can achieve decision making by using the below statements:  
</p><p>3/4

If statements </p><p>3/4

If-else statements </p><p>3/4

If-then-else statements </p><p>3/4

Elif statements </p><p>3/4

Nested if and if-else statements </p><p>3/4

Elif ladder </p><p>In

this chapter, we will discuss all the statements in detail with some real-time

examples. </p><p>

<a name="\_Toc62891920">If statements</a>

</p><p>If

statement is one of the most commonly used conditional statement in most of the

programming languages. It decides whether certain statements need to be

executed or not. If statement checks for a given condition, if the condition is

true, then the set of code present inside the if block will be executed. </p><p>The

If condition evaluates a Boolean expression and executes the block of code only

when the Boolean expression becomes TRUE. </p><p><b>Syntax</b>: </p><p>If (Boolean

expression): Block of code #Set of

statements to execute if the condition is true </p><p>Here,

the condition will be evaluated to a Boolean expression (true or false). If the

condition is true, then the statement or program present inside the if block will

be executed and if the condition is false, then the statements or program

present inside the if block will not be executed. </p><p>If

you observe the above flow-chart, first the controller will come to an if condition

and evaluate the condition if it is true, then the statements will be executed,

otherwise the code present outside the block will be executed. </p><p>Let's

see some examples on if statements. **Example: 1**

```
Num
= 5
```

```
if (Num
```

```
< 10):
    print("Num is smaller than 10")
    print("This
```

```
statements will always be executed.")
Output: Num is
```

```
smaller than 10.
This
```

```
statements will always be executed.
In
```

the above example, we declared a variable called 'Num' with the value as 5 and

in the if statement we are checking if the number is lesser than 10 or not if

the condition is true then a set of statements inside the if block will be executed.

**Example: 2**

```
a =
7
b =
0
if (a
```

```
> b):
    print("a is greater than b")
Output:
a is
```

```
greater than b
In
```

The above example, we are checking the relationship between a and b using the

greater than (>) operator in the if condition. In case a is greater than b then 'a is greater than b' will be printed.

**Example: 3**

```
passing_Score
= 60
my_Score
```

```
= 67
if (my_Score
```

```
>= passing_Score):  
print("You are passed in the exam")  
print("Congratulations!!!")  
Output:  
You
```

are passed in the exam  
Congratulations!!!

Here,

print("Congratulations!!!")      Statement will always be executed  
{even|also} though if

the given condition is true or false.      The

problem with the above code is that the statement  
'print("Congratulations!!!")'

Will always be executed even if the condition is evaluated to  
{true|authentic} or false.

{But|However,} in real time, if you pass in the exam or if you fail in  
{the|that the} exam then the

system will say Congratulations!!! .      In

Order {to|into} avoid this python provides one conditional statement  
called if-else.     

[If-else statements](#)

The

statement itself tells that if a given condition is true then execute the  
statements present inside if block and if the condition is false then  
execute

the else block.      Else

block will execute only when the condition becomes false, this is the  
block

where you will perform some actions when the condition is not true.

Statement evaluates the Boolean expression and executes the block of code

present inside the if block if the condition becomes TRUE and executes a block

of code present in the else block if the condition becomes FALSE.

**Syntax:** if(Boolean

expression): Block

{Of all} code #Set of statements to execute if condition is true

else:

Block

Of code #Set of statements to execute if condition is false

Here, the condition will be evaluated to a Boolean expression (true or false). If the

condition is true then the statements or program present inside the if block

will be executed and if the condition is false then the statements or program

present inside else block will be executed.

**Example: 1**

num = 5

if(num > 10): print("number is greater than 10")

else: print("number is less than 10")

print("This statement will always be executed")

executed.   
In the above example, we have declared a variable

Called 'num' with the value as 5 and in the if statement we are checking if the

number is greater than 5 or not.   
If the number is greater than 5 then, the block of

code inside the if block will be executed and if the condition fails then the

block of code present inside the else block will be executed.   
**Example:**

```
{2|Two }  
a =
```

```
7  
{B|B} =
```

```
0  
if(a
```

```
{&gt;|[] b):  
print("a is greater than b")  
else:  
print("b is greater than a")
```

```
<b>Output:  
a is  
greater than b  
In
```

The above code {if|when} a is greater than b {then|afterward} the statements present inside the {if

|should }block {will|would} {be|probably be} executed and {the|that the} statements present inside {the|that the} else block will {be

|probably be |likely be |soon be }skipped.   
**Example:**

```
3  
a = 7  
b = 0  
{If|In  
case} (a < b):  
print("a is smaller than b")
```

```
<b>else:  
print("b is smaller than  
a")  
Output:  
b is
```

```
smaller than a  
In
```

The above code, a is smaller than b, hence statements present inside the else



block will be executed and statements present inside the if block will {be|probably be|likely be|soon be} skipped.

</p><p>Now

{Let|Permit}'s take a real-time example. </p><p>

<a name="\_Toc62891922">If-Then-Else Statements</a>

</p><p>If-then-else

statements or conditional expressions are essential features of programming

languages and they make programs more useful to users. The if-then-else

statement {in|at} Python has the {following|after} basic structure:

</p><p>if condition1:</p><p>block1\_statement</p><p>elif

condition2:</p><p>block2\_statement</p><p>else:</p>

</div>block3\_statement

This structure will be evaluated as:

If condition1 is True, Python will execute block1\_statement. If condition1 is False, condition2 will be executed. If condition2 is evaluated as True, block2\_statement will be executed. If condition2 turns out to be False, Python will execute block3\_statement.

This structure will be evaluated as:

If condition1 is True, Python will execute block1\_statement. If condition1 is False, condition2 will be executed. If condition2 is evaluated as True, block2\_statement will be executed. If condition2 turns out to be False, Python will execute block3\_statement.

To illustrate, here is an if-then-else statement built within the function 'your\_choice':

```
def your_choice(answer):
```

```
if answer > 18:  
    print("You are underaged.")  
elif answer <= 65 and answer >18:  
    print("Welcome to the Adult's Club!")  
else:  
    print("You are too young for Adult's Club.")
```

```
print(your_choice(6))
```

```
print(your_choice(3))
```

```
print(your_choice(1))
```

```
print(your_choice(0))
```

You will get this output on the Python Shell:

You are underaged.

None

Welcome to the Adult's Club!

None

You are too young for Adult's Club.

None

You are too young for Adult's Club.

None

Conditional constructs may branch out to multiple 'elif' branches but can only have one 'else' branch at the end. Using the same code block, another elif statement may be inserted to provide for privileged member of the Adult's club: 23 year-old teens.

```
def your_choice(answer):
```

```
if answer < 18:
```

```
print("You are underaged.")
elif answer == 18 and answer < 23:
    print("Welcome to the Adult's Club!")
elif answer == 23:
    print("Welcome! You are a star member of the Adult's Club!")
else:
    print("You are too young for Adult's Club.")
print(your_choice(6))
print(your_choice(3))
print(your_choice(1))
print(your_choice(0))
print(your_choice(2))
```

You are overaged.

None

Welcome to the Adult's Club!

None

You are too young for Adult's Club.

None

You are too young for Adult's Club.

None

Welcome! You are a star member of the Adult's Club!

None

## **Elif Statements**

In python, we have one more conditional statement called elif statements. Elif statement is used to check multiple conditions only if the given if condition false. It's similar to an if-else statement and the only difference is that in else we will not check the condition but in elif we will do check the condition.

Elif statements are similar to if-else statements but elif statements evaluate multiple conditions.

### **Syntax:**

if (condition):

    #Set of statement to execute if condition is true

elif (condition):

    #Set of statements to be executed when if condition is false and elif condition is true

else:

    #Set of statement to be executed when both if and elif conditions are false

### **Example 1:**

```
num = 10
```

```
if (num == 0):
```

```
    print("Number is Zero")
```

```
elif (num > 5):
```

```
    print("Number is greater than 5")
```

```
else:
```

```
    print("Number is smaller than 5")
```

Output:

Number is greater than 5

In the above example we have declared a variable called 'num' with the value as 10, and in the if statement we are checking the condition if the condition becomes true. Then the block of code present inside the if condition will be executed.

If the condition becomes false then it will check the elif condition if the condition becomes true, then a block of code present inside the elif statement will be executed.

If it is false then a block of code present inside the else statement will be executed.

### **Example 2:**

```
num = -7
```

```
if (num > 0):
```

```
    print("Number is positive")
```

```
elif (num < 0):
```

```
    print("Number is negative")
```

```
else:
```

```
    print("Number is Zero")
```

### **Output:**

Number is negative

In the above example, first we are assigning value 7 to a variable called num. The controller will come to if statement and evaluate the Boolean expression  $\text{num} > 0$  but the number is not greater than zero hence if block will be skipped.

As if condition is evaluated to false the controller will come to elif statement and evaluate the Boolean expression  $\text{num} < 0$ , hence in our case number is less than zero hence 'Number is negative' is printed.

In case both if and elif condition is evaluated to false then a set of statements present inside the else block will be executed.

## **Nested if-else statements**

Nested if-else statements mean that an if statement or if-else statement is present inside another if or if-else block. Python provides this feature as well, this in turn will help us to check multiple conditions in a given program.

An if statement present inside another if statement which is present inside another if statements and so on.

### **Nested if Syntax:**

```
if(condition):
```

```
    #Statements to execute if condition is true
```

```
    if(condition):
```

```
        #Statements to execute if condition is true
```

```
    #end of nested if
```

```
#end of if
```

The above syntax clearly says that the if block will contain another if block in it and so on. If block can contain 'n' number of if block inside it.

### **Example: 1**

```
num = 5
```

```
if(num > 0):
```

```
    print("number is positive")
```

```
if(num < 10):
```

```
    print("number is less than 10")
```

### **Output:**

number is positive

number is less than 10

In the above example, we have declared a variable called 'num' with the value as 5.

First, it will check the first if statement if the condition is true, then the block of code present inside the first if statement will be executed then it will check the second if statement if the first if statement is true and so on.

### **Example 2:**

```
num = 7
```

```
if (num != 0):
```

```
    if (num > 0):
```

```
        print("Number is greater than Zero")
```

### **Output:**

Number is greater than Zero

Here, the controller will check if the given number is not equal to Zero or not, if the number is not equal to zero then it enters the first if block and then in the second if block it will check if the number is greater than Zero or not, if it's true then the control enters the nested if block and executes the statements and leaves the block and terminates the program.

### **Example 3:**

```
if ('python' in ['Java', 'python', 'C#']):
```

```
    print("Python is present in the list")
```

```
    if ('C#' in ['Java', 'python', 'C#']):
```

```
        print("Java is present in the list")
```

```
    if ('C#' in ['Java', 'python', 'C#']) :
```

```
print("C# is present in the list")
```

**Output:**

Python is present in the list

Java is present in the list

C# is present in the list

**Nested if-else Syntax:**

```
if(condition):
```

```
    #Statements to execute if condition is true
```

```
    if(condition):
```

```
        #Statements to execute if condition is true
```

```
    else:
```

```
        #Statements to execute if condition is false
```

```
    else:
```

```
        #Statements to execute if condition is false
```

Here we have included if-else block inside an if block, you can also include if-else block inside else block.

**Elif Ladder**

We have seen about the elif statements but what is this elif ladder. As the name itself suggests a program which contains ladder of elif statements or elif statements which are structured in the form of a ladder.

This statement is used to test multiple expressions.

**Syntax:**

```
if (condition):
```

```
    #Set of statement to execute if condition is true
```



elif (condition):

    #Set of statements to be executed when if condition is false and elif condition is true

elif (condition):

    #Set of statements to be executed when both if and first elif condition is false and second elif condition is true

elif (condition):

    #Set of statements to be executed when if, first elif and second elif conditions are false and third elif statement is true

else:

    #Set of statement to be executed when all if and elif conditions are false

### **Example: 1**

```
my_marks = 89
```

```
    if (my_marks < 35):
```

```
        print("Sorry!!!, You are failed in the exam")
```

```
    elif(my_marks < 60): print("Passed in Second class")
```

```
elif(my_marks > 60 and my_marks < 85):
```

```
    print("Passed in First class")
```

```
else:
```

```
    print("Passed in First class with distinction")
```

### **Output:**

Passed in First class with distinction

The above example describes the elif ladder. Firstly the control enters the if statement and evaluates the condition if the condition is true then the set of statements present inside the if block will be

executed else it will be skipped and the controller will come to the first elif block and evaluate the condition.

The similar process will continue for all the remaining elif statements and in case all if and elif conditions are evaluated to false then the else block will be executed.

## **If-Else In One Line**

In python, we can write if statements, if-else statements and elif statements in one line without worrying about the indentation.

If statement in one line

We know we can write if statements as shown below

### **Syntax:**

if (condition):

    #Set of statements to execute if condition is true

In python, it is permissible to write the above block in one line, which is similar to the above block.

### **Syntax:**

if (condition): #set of statements to execute if condition in true

There can be multiple statements as well, you just need to separate it by a semicolon (;)

### **Syntax:**

if (condition): statement 1; statement 2; statement 3;...;statement n

If the condition is true, then execute statement 1, statement 2 and so on up to statement n.

In case if the condition is false then none of the statements will be executed.

### **Example 1:**

```
num = 7
```

```
if (num > 0): print("Number is greater than Zero")
```

Output:

Number is greater than Zero

### **Example :2**

```
if ('y' in 'Python'): print('1'); print('2'); print('3')
```

Output:

1

2

3

## **If-Else Statements In One Line**

Syntax:

```
if (condition):
```

```
    #Set of statement to execute if condition is true
```

```
else:
```

```
    #Set of statement to execute if condition is false
```

The above if-else block can also be written as shown below.

### **Syntax:**

```
if (condition): #Set of statement to execute if condition is true
```

```
else: #Set of statement to execute if condition is false
```

There can be multiple statements as well, you just need to separate it by a semicolon (;)

Syntax:

```
if (condition): statement 1; statement 2; statement 3;...;statement n
else: statement 1; statement 2; statement 3;...;statement n
```

### **Example: 1**

```
num = 7
```

```
    if (num < 0): print("Number is greater than Zero")
    else: print("Number is smaller than Zero")
```

### **Output:**

Number is smaller than Zero

### **Example 2:**

```
if ('a' in 'fruits'): print("Apple"); print("Orange")
else: print("Mango"); print("Grapes")
```

### **Output:**

Mango  
Grapes

## **Elif Statements In One Line**

### **Syntax:**

```
if (condition):
```

```
    #Set of statement to execute if condition is true
```

```
elif (condition1):
```

```
    #Set of statement to execute if condition1 is true
```

```
else:
```

```
    #Set of statement to execute if condition and condition1 is
false
```

The above elif block can also be written as below.

## **Syntax:**

if (condition): #Set of statement to execute if condition is true

elif (condition1): #Set of statement to execute if condition1 is true

else: #Set of statement to execute if condition and condition1 is false

There can be multiple statements as well, you just need to separate it by a semicolon (;)

Syntax:

if (condition): statement 1; statement 2; statement 3;...;statement n

elif (condition): statement 1; statement 2; statement 3;...;statement n

else: statement 1; statement 2; statement 3;...;statement n

## **Example 1:**

```
num = 7
```

```
if (num < 0): print("Number is smaller than Zero") elif (num > 0):  
print("Number is greater than Zero")
```

```
else: print("Number is Zero")
```

Output:

Number is greater than Zero

## **Example 2:**

```
if ('a' in 'fruits'): print("Apple"); print("Orange")
```

```
elif ('u' in 'fruits'): print("Mango"); print("Grapes")
```

```
else: print("No fruits available")
```

Output:

Mango

Grapes

## Multiple Conditions In If Statements

It's not that you can only write one condition inside an if statement, we can also evaluate multiple conditions in if statement like below.

### Example 1:

```
num1 = 10
```

```
num2 = 20
```

```
num3 = 30
```

```
if (num1 == 10 and num2 == 20 and num3 == 30):
```

```
    print("All the conditions are true")
```

### Output:

All the conditions are true

Here, in if statement we are checking multiple conditions using AND operator, which means if all the conditions are true only when the statements inside an if block will be executed.

We can also specify the OR operators as well.

### Example: 2

```
fruitName = "Apple"
```

```
if (fruitName == "Mango" or fruitName == "Apple" or fruitName ==  
    "Grapes"):
```

```
    print("It's a fruit")
```

### Output:

It's a fruit

Here, in an, if statement out of three conditions, only one condition is true as that's the rule of OR operator. If any one condition is true then the condition will become true and the statement present inside the if block will be executed.

Let's consider a real-time scenario to find the number of days present in a month and we know that during a leap year the number of days will change. We will see this in a programmatic way using if, elif and else statements.

Example: 1

```
currentYear = int(input("Enter the year: "))
month = int(input("Enter the month: "))

if ((currentYear % 4) == 0 and (currentYear % 100) != 0 or
(currentYear % 400) == 0):
    print ("Leap Year")

    if (month == 1 or month == 3 or month == 5 or month == 7
or month == 8 or month == 10 or month == 12):
        print ("There are 31 days in this month")

    elif (month == 4 or month == 6 or month == 9 or month ==
11):
        print("There are 30 days in this month")

    elif (month == 2):
        print("There are 29 days in this month")

    else:
        print("Invalid month")

elif ((currentYear % 4) != 0 or (currentYear % 100) != 0 or
(currentYear % 400) != 0):
    print ("Non Leap Year")

    if (month == 1 or month == 3 or month == 5 or month == 7
or month == 8 or month == 10 or month == 12) :
        print ("There are 31 days in this month")
```

```
11): elif (month == 4 or month == 6 or month == 9 or month ==
      print("There are 30 days in this month")
      elif (month == 2):
          print("There are 28 days in this month")
      else:
          print("Invalid month")
      else:
          print("Invalid Year")
```

### **Output: 1**

Enter the year: 2020

Enter the month: 4

There are 30 days in this month

### **Output: 2**

Enter the year: 2020

Enter the month: 1

There are 31 days in this month

### **Output: 3**

Enter the year: 2019

Enter the month: 2

There are 28 days in this month

Output: 4

Enter the year: 2020



Enter the month: 2

There are 29 days in this month

Wrapping up, we learned about the Conditional Statements in Python. These are the statements which alter the control flow of execution in our program. We have different types of conditional statements like if, if-else, elif, nested if and nested if-else statements which control the execution of our program.

If the statement evaluates a Boolean expression to true or false, if the condition is true then the statement inside the if block will be executed in case if the condition is false then the statement present inside the else block will be executed only if you have written the else block.

We have one more statement called elif statement where the else statement is combined with an if statement, which executes depending on the previous if or elif statements.

## How Do Loops Work?

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times. Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. For the purpose of going over the small parts of coding again loops are used. Python programming language provides following types of loops to handle looping requirements. Python provides three ways for executing the loops; these are for loops, while loops and lastly nested loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

Python programming language provides following types of loops to handle looping requirements.

- While loop: Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
- For loop: Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
- Nested loops: You can use one or more loop inside any another while, for or do..while loop.

## Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

Let us go through the loop control statements briefly

- Break statement: Terminates the loop statement and transfers execution to the statement immediately following the

loop.

- Continue statement: Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
- Pass statement: The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute.

## Python break statement

It terminates the current loop and resumes execution at the next statement, just like the traditional break statement in C.

The most common use for break is when some external condition is triggered requiring a hasty exit from a loop. The break statement can be used in both while and for loops.

If you are using nested loops, the break statement stops the execution of the innermost loop and start executing the next line of code after the block.

### Syntax

The syntax for a break statement in Python is as follows –

```
break
```

### Example

```
#!/usr/bin/python
```

```
for letter in 'Python':    # First Example
```

```
    if letter == 'h':
```

```
        break
```

```
    print 'Current Letter :', letter
```

```
var = 10                # Second Example
```

```
while var > 0:
```

```
print 'Current variable value :', var
var = var -1
if var == 5:
    break
print "Good bye!"
```

When the above code is executed, it produces the following result –

```
Current Letter : P
Current Letter : y
Current Letter : t
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Good bye!
```

## **Python continue statement**

It returns the control to the beginning of the while loop.. The continue statement rejects all the remaining statements in the current iteration of the loop and moves the control back to the top of the loop.

The continue statement can be used in both while and forloops.

### **Syntax**

```
continue
```

### **Example**

```
#!/usr/bin/python
```

```
for letter in 'Python':    # First Example
```

```
    if letter == 'h':
```

```
        continue
```

```
    print 'Current Letter :', letter
```

```
var = 10                    # Second Example
```

```
while var > 0:
```

```
    var = var -1
```

```
    if var == 5:
```

```
        continue
```

```
    print 'Current variable value :', var
```

```
print "Good bye!"
```

When the above code is executed, it produces the following result –

Current Letter : P

Current Letter : y

Current Letter : t

Current Letter : o

Current Letter : n

Current variable value : 9

Current variable value : 8

Current variable value : 7

Current variable value : 6

Current variable value : 4

Current variable value : 3

Current variable value : 2

Current variable value : 1

Current variable value : 0

Good bye!

## **Python pass statement**

It is used when a statement is required syntactically but you do not want any command or code to execute.

The pass statement is a null operation; nothing happens when it executes. The pass is also useful in places where your code will eventually go, but has not been written yet (e.g., in stubs for example) –

### **Syntax**

```
pass
```

### **Example**

```
#!/usr/bin/python
```

```
for letter in 'Python':
```

```
    if letter == 'h':
```

```
        pass
```

```
        print 'This is pass block'
```

```
    print 'Current Letter :', letter
```

```
print "Good bye!"
```

When the above code is executed, it produces following result –

Current Letter : P

Current Letter : y

Current Letter : t

This is pass block

Current Letter : h

Current Letter : o

Current Letter : n

Good bye!

## Understanding The For Loop

For loops are used for sequential traversal. For example: traversing a list or string or array etc. In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is “for in” loop which is similar to for each loop in other languages. Let us learn how to use for in loop for sequential traversals.

Syntax:

```
for iterator_var in sequence:
```

```
    statements(s)
```

It can be used to iterate over iterators and a range.

```
# Python program to illustrate
```

```
# Iterating over a list
```

```
print("List Iteration")
```

```
l = ["Techs", "for", " Techs"]
```

```
for i in l:
```

```
    print(i)
```

```
# Iterating over a tuple (immutable)
```

```
print("\nTuple Iteration")
```

```
t = ("Techs ", "for", " Techs ")
```

```
for i in t:
```

```
    print(i)
```

```
# Iterating over a String
```

```
print("\nString Iteration")
```

```
s = " Techs"
```

```
for i in s :
```

```
    print(i)
```

```
# Iterating over dictionary
```

```
print("\nDictionary Iteration")
```

```
d = dict()
```

```
d['xyz'] = 123
```

```
d['abc'] = 345
```

```
for i in d :
```

```
    print("%s %d" %(i, d[i]))
```

Output:

List Iteration

Techs

for

techs

Tuple Iteration

techs



for

techs

String Iteration

T

e

c

h

s

Dictionary Iteration

xyz 123

abc 345

## Using else statement with for loops

We can also combine else statement with for loop like in while loop. But as there is no condition in for loop based on which the execution will terminate so the else block will be executed immediately after for block finishes execution.

Below example explains how to do this:

```
# Python program to illustrate
```

```
# combining else with for
```

```
list = ["Techs", "for", "techs"]
```

```
for index in range(len(list)):
```

```
    print list[index]
```

```
else:
```

```
    print "Inside Else Block"
```

Output:

techs

for

techs

Inside Else Block

## Working With While Loop

In python, while loop is used to execute a block of statements repeatedly until a given a condition is satisfied. And when the condition becomes false, the line immediately after the loop in program is executed.

Syntax :

while expression:

    statement(s)

All the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

Example:

```
# Python program to illustrate
```

```
# while loop
```

```
count = 0
```

```
while (count < 3):
```

```
    count = count + 1
```

```
    print("Hello Tech")
```

Output:

Hello Tech

Hello Tech

Hello Tech

## Using else statement with while loops

As discussed above, while loop executes the block until a condition is satisfied. When the condition becomes false, the statement immediately after the loop is executed.

The else clause is only executed when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won't be executed.

If else like this:

if condition:

    # execute these statements

else:

    # execute these statements

**and while loop like this are similar**

while condition:

    # execute these statements

else:

    # execute these statements

#Python program to illustrate

# combining else with while

count = 0

while (count < 3):

    count = count + 1

```
print("Hello Tech ")
```

else:

```
print("In Else Block")
```

Output:

Hello Tech

Hello Tech

Hello Tech

In Else Block

## Single statement while block

Just like the if block, if the while block consists of a single statement the we can declare the entire loop in a single line as shown below:

```
# Python program to illustrate
```

```
# Single statement while block
```

```
count = 0
```

```
while (count == 0): print("Hello Tech ")
```

### Note

It is suggested **not to use** this type of loops as it is a never ending infinite loop where the condition is always true and you have to forcefully terminate the compiler.

## The Nested Loop

Python programming language allows to use one loop inside another loop. Following section shows few examples to illustrate the concept.

Syntax:

```
for iterator_var in sequence:
```

```
    for iterator_var in sequence:
```

```
statements(s)
```

```
statements(s)
```

The syntax for a nested while loop statement in Python programming language is as follows:

while expression:

while expression:

```
statement(s)
```

```
statement(s)
```

A final note on loop nesting is that we can put any type of loop inside of any other type of loop. For example a for loop can be inside a while loop or vice versa.

# Python program to illustrate

# nested for loops in Python

```
from __future__ import print_function
```

```
for i in range(1, 5):
```

```
    for j in range(i):
```

```
        print(i, end=' ')
```

```
    print()
```

Output:

1

2 2

3 3 3

4 4 4 4

## Writing An Exception In Your Code

A Python program terminates as soon as it encounters an error. In Python, an error can be a syntax error or an exception. In this chapter, you will see what an exception is and how it differs from a syntax error. After that, you will learn about raising exceptions and making assertions. Then, you'll finish with a demonstration of the try and except block.

## Exceptions Versus Syntax Errors

Syntax errors occur when the parser detects an incorrect statement. Observe the following example:

```
>>> print( 0 / 0 )
```

```
File "<stdin>", line 1
```

```
    print( 0 / 0 )
```

```
          ^
```

SyntaxError: invalid syntax

The arrow indicates where the parser ran into the **syntax error** . In this example, there was one bracket too many. Remove it and run your code again:

```
>>> print( 0 / 0 )
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
```

ZeroDivisionError: integer division or modulo by zero

This time, you ran into an **exception error** . This type of error occurs whenever syntactically correct Python code results in an error. The last line of the message indicated what type of exception error you ran into.

Instead of showing the message exception error, Python details what type of exception error was encountered. In this case, it was a `ZeroDivisionError`. Python comes with [various built-in exceptions](#) as well as the possibility to create self-defined exceptions.

## Can I Raise My Own Exceptions in the Code?

We can use `raise` to throw an exception if a condition occurs. The statement can be complemented with a custom exception.

If you want to throw an error when a certain condition occurs using `raise`, you could go about it like this:

```
x = 10
if x > 5:
    raise Exception('x should not exceed 5. The value of x was:
{}'.format(x))
```

When you run this code, the output will be the following:

Traceback (most recent call last):

File "<input>", line 4, in <module>

Exception: x should not exceed 5. The value of x was: 10

The program comes to a halt and displays our exception to screen, offering clues about what went wrong.

## The AssertionError Exception

Instead of waiting for a program to crash midway, you can also start by making an assertion in Python. We assert that a certain condition is met. If this condition turns out to be `True`, then that is excellent! The program can continue. If the condition turns out to be `False`, you can have the program throw an `AssertionError` exception.

Have a look at the following example, where it is asserted that the code will be executed on a Linux system:

```
import sys
```

```
assert ('linux' in sys.platform), "This code runs on Linux only."
```

If you run this code on a Linux machine, the assertion passes. If you were to run this code on a Windows machine, the outcome of the assertion would be False and the result would be the following:

Traceback (most recent call last):

```
File "<input>", line 2, in <module>
```

AssertionError: This code runs on Linux only.

In this example, throwing an AssertionError exception is the last thing that the program will do. The program will come to halt and will not continue. What if that is not what you want?

## The Try And Except Block: Handling Exceptions

The try and except block in Python is used to catch and handle exceptions. Python executes code following the try statement as a “normal” part of the program. The code that follows the except statement is the program’s response to any exceptions in the preceding try clause.

When syntactically correct code runs into an error, Python will throw an exception error. This exception error will crash the program if it is unhandled. The except clause determines how your program responds to exceptions.

The following function can help you understand the try and except block:

```
def linux_interaction():
```

```
    assert ('linux' in sys.platform), "Function can only run on Linux systems."
```

```
    print('Doing something.')
```

The linux\_interaction() can only run on a Linux system. The assert in this function will throw an AssertionError exception if you call it on an operating system other than Linux.



You can give the function a try using the following code:

```
try:
```

```
    linux_interaction()
```

```
except:
```

```
    pass
```

The way you handled the error here is by handing out a pass. If you were to run this code on a Windows machine, you would get the following output:

You got nothing. The good thing here is that the program did not crash. But it would be nice to see if some type of exception occurred whenever you ran your code. To this end, you can change the pass into something that would generate an informative message, like so:

```
try:
```

```
    linux_interaction()
```

```
except:
```

```
    print('Linux function was not executed')
```

Execute this code on a Windows machine:

```
Linux function was not executed
```

When an exception occurs in a program running this function, the program will continue as well as inform you about the fact that the function call was not successful.

What you did not get to see was the type of error that was thrown as a result of the function call. In order to see exactly what went wrong, you would need to catch the error that the function threw.

The following code is an example where you capture the AssertionError and output that message to screen:

```
try :
```

```
linux_interaction()
except AssertionError as error:
    print(error)
    print('The linux_interaction() function was not executed')
```

Running this function on a Windows machine outputs the following:

Function can only run on Linux systems.

The linux\_interaction() function was not executed

The first message is the AssertionError, informing you that the function can only be executed on a Linux machine. The second message tells you which function was not executed.

In the previous example, you called a function that you wrote yourself. When you executed the function, you caught the AssertionError exception and printed it to screen.

Here's another example where you open a file and use a built-in exception:

```
try:
    with open('file.log') as file:
        read_data = file.read()
except:
    print('Could not open file.log')
```

If *file.log* does not exist, this block of code will output the following:

Could not open file.log

This is an informative message, and our program will still continue to run. In the Python docs, you can see that there are a lot of built-in exceptions that you can use here. One exception described on that page is the following:

## Exception FileNotFoundError

Raised when a file or directory is requested but doesn't exist. Corresponds to errno ENOENT.

To catch this type of exception and print it to screen, you could use the following code:

try:

```
    with open('file.log') as file:
```

```
        read_data = file.read()
```

```
except FileNotFoundError as fnf_error:
```

```
    print(fnf_error)
```

In this case, if file.log does not exist, the output will be the following:

```
[Errno 2] No such file or directory: 'file.log'
```

You can have more than one function call in your try clause and anticipate catching various exceptions. A thing to note here is that the code in the try clause will stop as soon as an exception is encountered.

**Warning** : Catching Exception hides all errors...even those which are completely unexpected. This is why you should avoid bare except clauses in your Python programs. Instead, you'll want to refer to specific exception classes you want to catch and handle.

Look at the following code. Here, you first call the linux\_interaction() function and then try to open a file:

try:

```
    linux_interaction()
```

```
    with open('file.log') as file:
```

```
        read_data = file.read()
```

```
except FileNotFoundError as fnf_error :
```

```
print(fnf_error)
```

```
except AssertionError as error:
```

```
    print(error)
```

```
    print('Linux linux_interaction() function was not executed')
```

If the file does not exist, running this code on a Windows machine will output the following:

Function can only run on Linux systems.

Linux linux\_interaction() function was not executed

Inside the try clause, you ran into an exception immediately and did not get to the part where you attempt to open file.log. Now look at what happens when you run the code on a Linux machine:

[Errno 2] No such file or directory: 'file.log'

Here are the key takeaways:

- A try clause is executed up until the point where the first exception is encountered.
- Inside the except clause, or the exception handler, you determine how the program responds to the exception.
- You can anticipate multiple exceptions and differentiate how the program should respond to them.

- Avoid using bare except clauses.

## The Else Clause

In Python, using the else statement, you can instruct a program to execute a certain block of code only in the absence of exceptions.

Look at the following example:

```
try:
```

```
    linux_interaction()
```

```
except AssertionError as error:
```

```
    print(error)
```

```
else:
```

```
    print('Executing the else clause.')
```

If you were to run this code on a Linux system, the output would be the following:

Doing something.

Executing the else clause.

Because the program did not run into any exceptions, the else clause was executed.

You can also try to run code inside the else clause and catch possible exceptions there as well:

```
try:
```

```
    linux_interaction()
```

```
except AssertionError as error:
```

```
    print(error)
```

```
else:
```

```
    try:
```

```
        with open('file.log') as file:
```

```
            read_data = file.read()
```

```
except FileNotFoundError as fnf_error:
```

```
    print(fnf_error)
```

If you were to execute this code on a Linux machine, you would get the following result:

Doing something.

[Errno 2] No such file or directory: 'file.log'

From the output, you can see that the `linux_interaction()` function ran. Because no exceptions were encountered, an attempt to open `file.log` was made. That file did not exist, and instead of opening the file, you caught the `FileNotFoundError` exception.

## **Cleaning Up After Using finally**

Imagine that you always had to implement some sort of action to clean up after executing your code. Python enables you to do so using the `finally` clause.

Have a look at the following example:

```
try:
```

```
    linux_interaction()
```

```
except AssertionError as error:
```

```
    print(error)
```

```
else:
```

```
    try:
```

```
        with open('file.log') as file:
```

```
            read_data = file.read()
```

```
    except FileNotFoundError as fnf_error:
```

```
print(fnf_error)
```

finally:

```
print('Cleaning up, irrespective of any exceptions.')
```

In the previous code, everything in the finally clause will be executed. It does not matter if you encounter an exception somewhere in the try or else clauses. Running the previous code on a Windows machine would output the following:

Function can only run on Linux systems.

Cleaning up, irrespective of any exceptions.

## Working With User-Defined Functions

In all programming and scripting language, a function is a block of program statements which can be used repetitively in a program. The purpose of functions is to break down the code into blocks that are useful. Thus, allowing one to authorize the code, as well as transform into a form that is readable, so that it can be reused from time to time. Thus, a lot of time is saved in this process.

Functions that we define ourselves to do certain specific task are referred as user-defined functions. The way in which we define and call functions in Python are already discussed. Functions that readily come with Python are called built-in functions. If we use functions written by others in the form of library, it can be termed as library functions. All the other functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.

The following is the syntax of defining a function.

Syntax:

```
def function_name(parameters):
```

```
    "function docstring"
```

```
    statement1
```

```
    statement2
```

```
    ...
```

```
    ...
```

```
    return [expr]
```

The keyword `def` is followed by a suitable identifier as the name of the function and parentheses. One or more parameters may be optionally mentioned inside parentheses. The `:` symbol after parentheses starts an indented block.



The first statement in this block is an explanatory string which tells something about the functionality. It is called a docstring and it is optional. It is somewhat similar to a comment. Subsequent statements that perform a certain task form the body of the function.

The last statement in the function block includes the return keyword. It sends an execution control back to calling the environment. If an expression is added in front of return, its value is also returned.

Given below is the definition of the SayHello() function. When this function is called, it will print a greeting message.

In Python concept of function is same as in other languages. There are some built-in functions which are part of Python. Besides that, we can define functions according to our need.

Example: Function

```
def SayHello():
```

```
    """First line is docstring. When called, a greeting message will be displayed"""
```

```
    print ("Hello! Welcome to Python tutorial")
```

```
    return
```

To call a defined function, just use its name as a statement anywhere in the code. For example, the above function can be called as SayHello() and it will show the following output.

```
>>> SayHello()
```

```
Hello! Welcome to Python tutorial
```

## **Importance of User-Defined Functions**

In general, developers can write user-defined functions or it can be borrowed as a third-party library. This also means your own user-defined functions can also be a third-party library for other users. User-defined functions have certain advantages depending when and how they are used. Let 's have a look at the following points.

- User-defined functions are reusable code blocks; they only need to be written once, then they can be used multiple times. They can even be used in other applications, too.
- These functions are very useful, from writing common utilities to specific business logic. These functions can also be modified per requirement.
- The code is usually well organized, easy to maintain, and developer-friendly. Which means it can support the modular design approach.
- As user-defined functions can be written independently, the tasks of a project can be distributed for rapid application development.
- A well-defined and thoughtfully written user-defined function can ease the application development process.

Now that we have a basic understanding of the advantages, let's have a look at different function arguments in Python.

## **The Arguments of Functions Available**

- In Python, a user-defined function's declaration begins with the keyword `def` and followed by the function name.
- The function may take arguments(s) as input within the opening and closing parentheses, just after the function name followed by a colon.
- After defining the function name and arguments(s) a block of program statement(s) start at the next line and these statement(s) must be indented.

In Python, user-defined functions can take four different types of arguments. The argument types and their meanings, however, are pre-defined and can't be changed. But a developer can, instead, follow these pre-defined rules to make their own custom functions. The following are the four types of arguments and their rules.

### **Default arguments**

Python has a different way of representing syntax and default values for function arguments. Default values indicate that the function argument will take that value if no argument value is passed during function call. The default value is assigned by using assignment (=) operator. Below is a typical syntax for default argument. Here, msg parameter has a default value Hello!.

— **Function definition**

```
def defaultArg ( name, msg = "Hello!"):
```

— **Function call**

```
defaultArg ( name)
```

## **Required arguments**

Required arguments are the mandatory arguments of a function. These argument values must be passed in correct number and order during function call. Below is a typical syntax for a required argument function.

— **Function definition**

```
def requiredArg (str,num):
```

— **Function call**

```
requiredArg ("Hello",12)
```

## **Keyword arguments:**

Keyword arguments are relevant for Python function calls. The keywords are mentioned during the function call along with their corresponding values. These keywords are mapped with the function arguments so the function can easily identify the corresponding values even if the order is not maintained during the function call. The following is the syntax for keyword arguments.

— **Function definition**

```
def keywordArg ( name, role ):
```

### — **Function call**

**keywordArg** ( name = "Tom", role = "Manager")

or

**keywordArg** ( role = "Manager", name = "Tom")

## **Variable number of arguments:**

This is very useful when we do not know the exact number of arguments that will be passed to a function. Or we can have a design where any number of arguments can be passed based on the requirement. Below is the syntax for this type of function call.

### — **Function definition**

**def varlengthArgs** (\*varargs):

### — **Function call**

varlengthArgs(30,40,50,60)

Now that we have an idea about the different argument types in Python. Let's check the steps to write a user-defined function.

## **Writing Out Your Own User-Defined Functions**

These are the basic steps in writing user-defined functions in Python. For additional functionalities, we need to incorporate more steps as needed.

- Step 1: Declare the function with the keyword **def** followed by the function name.
- Step 2: Write the arguments inside the opening and closing parentheses of the function, and end the declaration with a colon.
- Step 3: Add the program statements to be executed.
- Step 4: End the function with/without return statement.

The example below is a typical syntax for defining functions:

```
def userDefFunction (arg1, arg2, arg3 ...):  
    program statement1  
    program statement3  
    program statement3  
    ....  
return ;
```

## Function with Parameters

It is possible to define a function to receive one or more parameters (also called arguments) and use them for processing inside the function block. Parameters/arguments may be given suitable formal names. The SayHello() function is now defined to receive a string parameter called name. Inside the function, print() statement is modified to display the greeting message addressed to the received parameter.

Example: Parameterized Function

```
def SayHello(name):  
    print ("Hello {}".format(name))  
    return
```

You can call the above function as shown below.

```
>>> SayHello("Tech")
```

```
Hello Tech!
```

The names of the arguments used in the definition of the function are called formal arguments/parameters. Objects actually used while calling the function are called actual arguments/parameters.

In the following example, the result() function is defined with three arguments as marks. It calculates the percentage and displays

pass/fail result. The function is called by providing different values on every call.

```
def result(m1,m2,m3) :  
    ttl=m1+m2+m3  
    percent=ttl/3  
    if percent>=50:  
        print ("Result: Pass")  
    else:  
        print ("Result: Fail")  
    return
```

```
p=int(input("Enter your marks in physics: "))  
c=int(input("Enter your marks in chemistry: "))  
m=int(input("Enter your marks in maths: "))  
result(p,c,m)
```

Run the above script in IDLE with two different sets of inputs is shown below:

Result:

Enter your marks in physics: 50

Enter your marks in chemistry: 60

Enter your marks in maths: 70

Result: Pass

Enter your marks in physics: 30

Enter your marks in chemistry: 40

Enter your marks in maths: 50

Result: Fail

### Parameter with Default Value

While defining a function, its parameters may be assigned default values. This default value gets substituted if an appropriate actual argument is passed when the function is called. However, if the actual argument is not provided, the default value will be used inside the function.

The following SayHello() function is defined with the name parameter having the default value 'Guest'. It will be replaced only if some actual argument is passed.

Example: Parameter with Default Value

```
def SayHello(name='Guest'):
    print ("Hello " + name)
    return
```

You can call the above function with or without passing a value, as shown below.

```
>>> SayHello()
```

Hello Guest

```
>>> SayHello(Tech')
```

Hello Tech

## Function with Keyword Arguments

In order to call a function with arguments, the same number of actual arguments must be provided. Consider the following function.

```
def AboutMe(name, age):
    print ("Hi! My name is {} and I am {} years old".format(name,age))
```

return

The above function is defined to receive two positional arguments. If we try to call it with only one value passed, the Python interpreter flashes `TypeError` with the following message:

```
>>> AboutMe("Tech")
```

```
TypeError: AboutMe() missing 1 required positional argument: 'age'
```

If a function receives the same number of arguments and in the same order as defined, it behaves correctly.

```
>>> AboutMe("Tech", 23)
```

```
Hi! My name is Mohan and I am 23 years old
```

Python provides a useful mechanism of using the name of the formal argument as a keyword in function call. The function will process the arguments even if they are not in the order provided in the definition, because, while calling, values are explicitly assigned to them. The following calls to the `AboutMe()` function is perfectly valid.

```
>>> AboutMe(age=23, name="Tech")
```

```
Hi! My name is Tech and I am 23 years old
```

## Function with Return Value

Most of the times, we need the result of the function to be used in further processes. Hence, when a function returns, it should also return a value.

A user-defined function can also be made to return a value to the calling environment by putting an expression in front of the return statement. In this case, the returned value has to be assigned to some variable.

Consider the following example function with the return value.

Example: Function with Return Value

```
def sum(a, b):
```



```
    return a + b
```

The above function can be called and provided the value, as shown below.

```
>>> total=sum(10, 20)
```

```
>>> total 30
```

```
>>> total=sum(5, sum(10, 20))
```

```
>>> total 35
```

## Passing Arguments by Reference

In Python, arguments are always passed by reference. The following snippet will confirm this:

Example: Passing Argument by Reference

```
def myfunction(arg):
```

```
    print ('value received:',arg,'id:',id(arg))
```

```
    return
```

```
x=10
```

```
print ('value passed:',x, 'id:',id(x))
```

```
myfunction(x)
```

The built-in `id()` function returns a unique integer corresponding to the identity of an object. In the above code, `id()` of `x` before and after passing to a function shows a similar value.

```
>>> x=10
```

```
>>> id(x)
```

```
1678339376
```

```
>>> myfunction(x)
```

```
value received: 10 id: 1678339376
```

If we change the above number variable inside a function then it will create a different variable as number, which is immutable. However, if we modify a mutable list object inside the function, and display its contents after the function is completed, the changes are reflected outside the function as well.

Example: Passing List by Reference

```
def myfunction (list):  
    list.append(40)  
    print ("Modified list inside a function: ", list)  
    return
```

The following result confirms that arguments are passed by reference to a Python function.

```
>>> mylist=[10,20,30]
```

```
>>> myfunction(mylist)
```

```
Modified list inside a function: [10, 20, 30, 40]
```

```
>>> mylist
```

```
[10, 20, 30, 40]
```

## **Inheritances In The Python Code**

Inheritance is the capability of one class to derive or inherit the properties from some another class. The benefits of inheritance are:

1. It represents real-world relationships well.
2. It provides reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

Every object-oriented programming language would not be worthy to look at or use, if it weren't to support inheritance. Inheritance was invented in 1969 for Simula. Python not only supports inheritance but multiple inheritance as well. Generally speaking, inheritance is the mechanism of deriving new classes from existing ones. By doing this we get a hierarchy of classes. In most class-based object-oriented languages, an object created through inheritance (a "child object") acquires all, - though there are exceptions in some programming languages, - of the properties and behaviors of the parent object.

Inheritance allows programmers to create classes that are built upon existing classes, and this makes it possible that a class created through inheritance inherits the attributes and methods of the parent class. This means that inheritance supports code reusability. The methods or generally speaking the software inherited by a subclass is considered to be reused in the subclass. The relationships of objects or classes through inheritance give rise to a directed graph.

The class from which a class inherits is called the parent or superclass. A class which inherits from a superclass is called a subclass, also called heir class or child class. Superclasses are sometimes called ancestors as well. There exists a hierarchy relationship between classes. It's similar to relationships or

categorizations that we know from real life. Think about vehicles, for example. Bikes, cars, buses and trucks are vehicles. Pick-ups, vans, sports cars, convertibles and estate cars are all cars and by being cars they are vehicles as well. We could implement a vehicle class in Python, which might have methods like accelerate and brake. Cars, Buses and Trucks and Bikes can be implemented as subclasses which will inherit these methods from vehicle.

Below is a simple example of inheritance in Python

```
# A Python program to demonstrate inheritance
# Base or Super class. Note object in bracket.
# (Generally, object is made ancestor of all classes)
# In Python 3.x "class Person" is
# equivalent to "class Person(object)"
class Person(object):

    # Constructor
    def __init__(self, name):
        self.name = name

    # To get name
    def getName(self):
        return self.name

    # To check if this person is employee
    def isEmployee(self) :
        return False
```

# Inherited or Sub class (Note Person in bracket)

```
class Employee(Person):
```

```
    # Here we return true
```

```
    def isEmployee(self):
```

```
        return True
```

# Driver code

```
emp = Person("Tech1") # An Object of Person
```

```
print(emp.getName(), emp.isEmployee())
```

```
emp = Employee("Tech2") # An Object of Employee
```

```
print(emp.getName(), emp.isEmployee())
```

**Output:**

Tech1 False

Tech2 True

## Different Forms Of Inheritance

Inheritance is an important aspect of the object-oriented paradigm. Inheritance provides code reusability to the program because we can use an existing class to create a new class instead of creating it from scratch.

In inheritance, the child class acquires the properties and can access all the data members and functions defined in the parent class. A child class can also provide its specific implementation to the functions of the parent class. In this section of the tutorial, we will discuss inheritance in detail.

In python, a derived class can inherit base class by just mentioning the base in the bracket after the derived class name. Consider the

following syntax to inherit a base class into the derived class.

- **Single inheritance** : When a child class inherits from only one parent class, it is called as single inheritance. We saw an example above.
- **Multiple inheritance** : When a child class inherits from multiple parent classes, it is called as multiple inheritance.

Python provides us the flexibility to inherit multiple base classes in the child class.

The syntax to perform multiple inheritance is given below.

Syntax

```
class Base1:
```

```
    <class -suite>
```

```
class Base2:
```

```
    <class -suite>
```

```
.
```

```
class BaseN:
```

```
    <class -suite>
```

```
class Derived(Base1, Base2, ..... BaseN):
```

```
    <class -suite>
```

Example

```
class Calculation1:
```

```
    def Summation(self,a,b):
```

```
        return a+b;
```

```
class Calculation2:
```

```

    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print (d.Summation(10,20))
print (d.Multiplication(10,20))
print (d.Divide(10,20))

```

### Output:

```

30
200
0.5

```

Unlike Java and like C++, Python supports multiple inheritance. We specify all parent classes as comma separated list in bracket.

- **Multilevel inheritance:** When we have child and grand child relationship.

Multi-Level inheritance is possible in python like other object-oriented languages. Multi-level inheritance is archived when a derived class inherits another derived class. There is no limit on the number of levels up to which, the multi-level inheritance is archived in python.

The syntax of multi-level inheritance is given below.

Syntax

```

class class1:

```

```
<class -suite>
class class2(class1):
    <class suite>
class class3(class2):
    <class suite>
```

Example

```
class Animal:
    def speak(self):
        print ("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print ("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print ("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```



## Output:

dog barking

Animal Speaking

Eating bread...

- **Hierarchical inheritance** More than one derived classes are created from a single base.
- **Hybrid inheritance** : This form combines more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.

## Can I Override My Base Class?

We can provide some specific implementation of the parent class method in our child class. When the parent class method is defined in the child class with some specific implementation, then the concept is called method overriding. We may need to perform method overriding in the scenario where the different definition of a parent class method is needed in the child class.

Consider the following example to perform method overriding in python.

Example

```
class Animal:
```

```
    def speak(self):
```

```
        print ("speaking")
```

```
class Dog(Animal):
```

```
    def speak(self):
```

```
        print ("Barking")
```

```
d = Dog()
```

```
d.speak()
```

## Output:

Barking

In the above example, notice that `__init__()` method was defined in both classes, Triangle as well Polygon. When this happens, the method in the derived class overrides that in the base class. This is to say, `__init__()` in Triangle gets preference over the same in Polygon.

Generally when overriding a base method, we tend to extend the definition rather than simply replace it. The same is being done by calling the method in base class from the one in derived class (calling `Polygon.__init__()` from `__init__()` in Triangle).

A better option would be to use the built-in function `super()`. So, `super().__init__(3)` is equivalent to `Polygon.__init__(self,3)` and is preferred. You can learn more about the `super()` function in Python.

Two built-in functions `isinstance()` and `issubclass()` are used to check inheritances. Function `isinstance()` returns True if the object is an instance of the class or other classes derived from it. Each and every class in Python inherits from the base class object.

```
>>> isinstance(t, Triangle)
```

```
True
```

```
>>> isinstance(t, Polygon)
```

```
True
```

```
>>> isinstance(t, int)
```

```
False
```

```
>>> isinstance(t, object)
```

```
True
```

Similarly, `issubclass()` is used to check for class inheritance.

```
>>> issubclass(Polygon, Triangle)
```

False

```
>>> issubclass(Triangle,Polygon)
```

True

```
>>> issubclass(bool,int)
```

True

## **The issubclass(sub,sup) method**

The `issubclass(sub, sup)` method is used to check the relationships between the specified classes. It returns true if the first class is the subclass of the second class, and false otherwise.

Consider the following example.

Example

```
class Calculation1:
```

```
    def Summation(self,a,b):
```

```
        return a+b;
```

```
class Calculation2:
```

```
    def Multiplication(self,a,b):
```

```
        return a*b;
```

```
class Derived(Calculation1,Calculation2):
```

```
    def Divide(self,a,b):
```

```
        return a/b;
```

```
d = Derived()
```

```
print (issubclass(Derived,Calculation2))
```

```
print (issubclass(Calculation1,Calculation2))
```

**Output:**

True

False

## **The isinstance (obj, class) method**

The isinstance() method is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

Consider the following example.

Example

```
class Calculation1:
```

```
    def Summation(self,a,b):
```

```
        return a+b;
```

```
class Calculation2:
```

```
    def Multiplication(self,a,b):
```

```
        return a*b;
```

```
class Derived(Calculation1,Calculation2):
```

```
    def Divide(self,a,b):
```

```
        return a/b;
```

```
d = Derived()
```

```
print (isinstance(d,Derived))
```

**Output:**

True

## **Overloading**

In Python you can define a method in such a way that there are multiple ways to call it. Given a single method or function, we can

specify the number of parameters ourself. Depending on the function definition, it can be called with zero, one, two or more parameters.

This is known as method overloading. Not all programming languages support method overloading, but Python does.

For example, we create a class with one method sayHello(). The first parameter of this method is set to None, this gives us the option to call it with or without a parameter.

An object is created based on the class, and we call its method using zero and one parameter.

```
class Human:
```

```
    def sayHello(self, name=None):
```

```
        if name is not None:
```

```
            print('Hello ' + name)
```

```
        else:
```

```
            print('Hello ')
```

```
# Create instance
```

```
obj = Human()
```

```
# Call the method
```

```
obj.sayHello()
```

```
# Call the method with a parameter
```

```
obj.sayHello('Tech')
```

Output:

Hello

Hello Tech

To clarify method overloading, we can now call the method `sayHello()` in two ways:

```
obj.sayHello()
```

```
obj.sayHello('Tech')
```

We created a method that can be called with fewer arguments than it is defined to allow.

We are not limited to two variables, your method could have more variables which are optional.

## **A Final Note About Inheritances**

Inheritances allows us to define a class that inherits all the methods and properties from another class.

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

## Working With The Python Generators

Generators are functions that can be paused and resumed on the fly, returning an object that can be iterated over. Unlike lists, they are lazy and thus produce items one at a time and only when asked. So they are much more memory efficient when dealing with large datasets.

### How Does a Generator Work?

There is a lot of overhead in building an iterator in Python; we have to implement a class with `__iter__()` and `__next__()` method, keep track of internal states, raise `StopIteration` when there was no values to be returned etc.

This is both lengthy and counter intuitive. Generator comes into rescue in such situations.

Python generators are a simple way of creating iterators. All the overhead we mentioned above are automatically handled by generators in Python.

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

### Why Use Generators In Python

There are several reasons which make generators an attractive implementation to go for.

#### — Easy to Implement

Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of 2's using iterator class.

#### — Memory Efficient

A normal function to return a sequence will create the entire sequence in memory before returning the result. This is an overkill if the number of items in the sequence is very large.

Generator implementation of such sequence is memory friendly and is preferred since it only produces one item at a time.

### **— Represent Infinite Stream**

Generators are excellent medium to represent an infinite stream of data. Infinite streams cannot be stored in memory and since generators produce only one item at a time, it can represent infinite stream of data.

### **— Pipelining Generators**

Generators can be used to pipeline a series of operations. This is best illustrated using an example.

Suppose we have a log file from a famous fast food chain. The log file has a column (4th column) that keeps track of the number of pizza sold every hour and we want to sum it to find the total pizzas sold in 5 years.

Assume everything is in string and numbers that are not available are marked as 'N/A'. A generator implementation of this could be as follows.

## **How To Create A Generator In Python?**

It is fairly simple to create a generator in Python. It is as easy as defining a normal function with yield statement instead of a return statement.

If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.

The difference is that, while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

## **Differences Between Generator Function And A Normal Function**



Here is how a generator function differs from a normal function.

- Generator function contains one or more yield statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like `__iter__()` and `__next__()` are implemented automatically. So we can iterate through the items using `next()`.
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, `StopIteration` is raised automatically on further calls.

## Python Generator Expression

Simple generators can be easily created on the fly using generator expressions. It makes building generators easy.

Same as lambda function creates an anonymous function, generator expression creates an anonymous generator function.

The syntax for generator expression is similar to that of a list comprehension in Python. But the square brackets are replaced with round parentheses.

The major difference between a list comprehension and a generator expression is that while list comprehension produces the entire list, generator expression produces one item at a time.

They are kind of lazy, producing items only when asked for. For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

## The Differences Between “Yield” and “Return”?

The yield statement suspends function's execution and sends a value back to caller, but retains enough state to enable function to resume where it is left off. When resumed, the function continues

execution immediately after the last yield run. This allows its code to produce a series of values over time, rather than computing them at once and sending them back like a list.

Return sends a specified value back to its caller whereas Yield can produce a sequence of values. We should use yield when we want to iterate over a sequence, but don't want to store the entire sequence in memory.

Yield are used in Python generators. A generator function is defined like a normal function, but whenever it needs to generate a value, it does so with the yield keyword rather than return. If the body of a def contains yield, the function automatically becomes a generator function.

## **Are There Specific Times When I Should Work with a Generator?**

Generators give you lazy evaluation. You should use them when you need to iterate; either explicitly with 'for' or implicitly by passing it to any function or construct that iterates. You can think of generators as returning multiple items, as if they return a list, but instead of returning them all at once they return them one-by-one, and the generator function is paused until the next item is requested.

You should use generators when you want to calculate large sets of results (in particular calculations involving loops themselves) where you don't know if you are going to need all results, or where you don't want to allocate the memory for all results at the same time. Or for situations where the generator uses another generator, or consumes some other resource, and it's more convenient if that happened as late as possible.

Another time to use generator is when you want to replace callbacks with iteration. In some situations you want a function to do a lot of work and occasionally report back to the caller. Traditionally you'd use a callback function for this. You pass this callback to the work-function and it would periodically call this callback. The generator approach is that the work-function (now a generator) knows nothing

about the callback, and merely yields whenever it wants to report something. The caller, instead of writing a separate callback and passing that to the work-function, does all the reporting work in a little 'for' loop around the generator.

Generators allow us to ask for values as and when we need them, making our applications more memory efficient and perfect for infinite streams of data. They can also be used to refactor out the processing from loops resulting in cleaner, decoupled code.

## What Are The Regular Expressions?

A regular expression is a special sequence of characters that helps you match or find other strings or sets of strings, using a specialized syntax held in a pattern. Regular expressions are widely used in UNIX world.

The module `re` provides full support for Perl-like regular expressions in Python. The `re` module raises the exception `re.error` if an error occurs while compiling or using a regular expression.

Regular expressions are used to identify whether a pattern exists in a given sequence of characters (string) or not. They help in manipulating textual data, which is often a prerequisite for data science projects that involve text mining. You must have come across some application of regular expressions: they are used at the server side to validate the format of email addresses or password during registration, used for parsing text data files to find, replace or delete certain string, etc.

In Python, regular expressions are supported by the `re` module. That means that if you want to start using them in your Python scripts, you have to import this module with the help of `import`:

We would cover two important functions, which would be used to handle regular expressions. But a small thing first: There are various characters, which would have special meaning when they are used in regular expression. To avoid any confusion while dealing with regular expressions, we would use Raw Strings as `r'expression'`.

## The Basic Patterns Present

You can easily tackle many basic patterns in Python using the ordinary characters. Ordinary characters are the simplest regular expressions. They match themselves exactly and do not have a special meaning in their regular expression syntax.

Examples are `'A'`, `'a'`, `'X'`, `'5'`.

Ordinary characters can be used to perform simple exact matches:

```
pattern = r"Cookie"
sequence = "Cookie"
if re.match(pattern, sequence):
    print("Match!")
else: print("Not a match!")
```

The `match()` function returns a match object if the text matches the pattern. Otherwise it returns `None`. The `re` module also contains several other functions and you will learn some of them later on in the tutorial.

For now, though, let's focus on ordinary characters! Do you notice the `r` at the start of the pattern `Cookie`?

This is called a raw string literal. It changes how the string literal is interpreted. Such literals are stored as they appear.

For example, `\` is just a backslash when prefixed with a `r` rather than being interpreted as an escape sequence. You will see what this means with special characters. Sometimes, the syntax involves backslash-escaped characters and to prevent these characters from being interpreted as escape sequences, you use the raw `r` prefix. You don't actually need it for this example, however it is a good practice to use it for consistency.

## Repetitions

It becomes quite tedious if you are looking to find long patterns in a sequence. Fortunately, the `re` module handles repetitions using the following special characters:

- `+` Checks for one or more characters to its left.
- `*` Checks for zero or more characters to its left.
- `?` Checks for exactly zero or one character to its left.

But what if you want to check for exact number of sequence repetition?

For example, checking the validity of a phone number in an application. `re` module handles this very gracefully as well using the following regular expressions:

`{x}` - Repeat exactly x number of times.

`{x,}` - Repeat at least x times or more.

`{x, y}` - Repeat at least x times but no more than y times.

## Groups and Grouping Using Regular Expressions

Suppose that, when you're validating email addresses and want to check the user name and host separately.

This is when the group feature of regular expression comes in handy. It allows you to pick up parts of the matching text.

Parts of a regular expression pattern bounded by parenthesis() are called groups. The parenthesis does not change what the expression matches, but rather forms groups within the matched sequence. You have been using the `group()` function all along in this tutorial's examples. The plain `match.group()` without any argument is still the whole matched text as usual.

## The Match Function

This function attempts to match RE pattern to string with optional flags.

Here is the syntax for this function –

```
re.match(pattern, string, flags=0)
```

Here is the description of the parameters –

- Pattern: This is the regular expression to be matched.
- string: This is the string, which would be searched to match the pattern at the beginning of string.

- Flags: You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The `re.match` function returns a match object on success, `None` on failure. We use `group(num)` or `groups()` function of match object to get matched expression.

- `group(num=0)`: This method returns entire match (or specific subgroup `num`)
- `groups()`: This method returns all matching subgroups in a tuple (empty if there weren't any)

### Example

```
#!/usr/bin/python
```

```
import re
```

```
line = "Cats are smarter than dogs"
```

```
matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)
```

```
if matchObj:
```

```
    print "matchObj.group() : ", matchObj.group()
```

```
    print "matchObj.group(1) : ", matchObj.group(1)
```

```
    print "matchObj.group(2) : ", matchObj.group(2)
```

```
else:
```

```
    print "No match!!"
```

When the above code is executed, it produces following result –

```
matchObj.group() : Cats are smarter than dogs
```

```
matchObj.group(1) : Cats
```

```
matchObj.group(2) : smarter
```

## The *search* Function

This function searches for first occurrence of RE pattern within string with optional flags.

Here is the syntax for this function –

```
re.search(pattern, string, flags=0)
```

Here is the description of the parameters –

- Pattern: This is the regular expression to be matched.
- String: This is the string, which would be searched to match the pattern anywhere in the string.
- flags: You can specify different flags using bitwise OR (|). These are modifiers, which are listed in the table below.

The `re.search` function returns a match object on success, none on failure. We use `group(num)` or `groups()` function of match object to get matched expression.

- `group(num=0)`: This method returns entire match (or specific subgroup num)
- `groups()`: This method returns all matching subgroups in a tuple (empty if there weren't any)

### Example

```
#!/usr/bin/python
```

```
import re
```

```
line = "Cats are smarter than dogs";
```

```
searchObj = re.search( r'(.*) are (.*?) .*', line, re.M|re.I)
```

```
if searchObj:
```

```
    print "searchObj.group() : ", searchObj.group( )
```



```
print "searchObj.group(1) : ", searchObj.group(1)
print "searchObj.group(2) : ", searchObj.group(2)
else:
    print "Nothing found!!"
```

When the above code is executed, it produces following result –

```
searchObj.group() : Cats are smarter than dogs
searchObj.group(1) : Cats
searchObj.group(2) : smarter
```

## Matching Versus Searching

Python offers two different primitive operations based on regular expressions: match checks for a match only at the beginning of the string, while search checks for a match anywhere in the string (this is what Perl does by default).

### Example

```
import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
    print "match --> matchObj.group() : ", matchObj.group()
else:
    print "No match!!"

searchObj = re.search( r'dogs', line, re.M|re.I)
if searchObj:
```

```
    print "search --> searchObj.group() : ", searchObj.group()
else:
    print "Nothing found!!"
```

When the above code is executed, it produces the following result –  
No match!!

```
search --> matchObj.group() : dogs
```

### Search and Replace

One of the most important re methods that use regular expressions is sub.

### Syntax

```
re.sub(pattern, repl, string, max=0)
```

This method replaces all occurrences of the RE pattern in string with repl, substituting all occurrences unless max provided. This method returns modified string.

### Example

```
import re
phone = "2004-959-559 # This is Phone Number"
```

```
# Delete Python-style comments
```

```
num = re.sub(r'#.*$', "", phone)
```

```
print "Phone Num : ", num
```

```
# Remove anything other than digits
```

```
num = re.sub(r'\D', "", phone)
```

```
print "Phone Num : ", num
```

When the above code is executed, it produces the following result –

Phone Num : 2004-959-559

Phone Num : 2004959559

## Regular Expression Modifiers: Option Flags

Regular expression literals may include an optional modifier to control various aspects of matching. The modifiers are specified as an optional flag. You can provide multiple modifiers using exclusive OR (|), as shown previously and may be represented by one of these

–

re.I

Performs case-insensitive matching.

re.L

Interprets words according to the current locale. This interpretation affects the alphabetic group (\w and \W), as well as word boundary behavior(\b and \B).

re.M

Makes \$ match the end of a line (not just the end of the string) and makes ^ match the start of any line (not just the start of the string).

re.S

Makes a period (dot) match any character, including a newline.

re.U

Interprets letters according to the Unicode character set. This flag affects the behavior of \w, \W, \b, \B.

re.X

Permits "cuter" regular expression syntax. It ignores whitespace (except inside a set [] or when escaped by a backslash) and treats unescaped # as a comment marker.

## Regular Expression Patterns

Except for control characters, (+ ? . \* ^ \$ ( ) [ ] { } | \), all characters match themselves. You can escape a control character by preceding it with a backslash.

Following table lists the regular expression syntax that is available in Python –

`^`

Matches beginning of line.

`$`

Matches end of line.

`.`

Matches any single character except newline. Using `m` option allows it to match newline as well.

`[...]`

Matches any single character in brackets.

`[^...]`

Matches any single character not in brackets

`re*`

Matches 0 or more occurrences of preceding expression.

`re+`

Matches 1 or more occurrence of preceding expression.

`re?`

Matches 0 or 1 occurrence of preceding expression.

`re{ n}`

Matches exactly `n` number of occurrences of preceding expression.

`re{ n,}`

Matches n or more occurrences of preceding expression.

`re{ n, m }`

Matches at least n and at most m occurrences of preceding expression.

`a| b`

Matches either a or b.

`(re)`

Groups regular expressions and remembers matched text.

`(?imx)`

Temporarily toggles on i, m, or x options within a regular expression. If in parentheses, only that area is affected.

`(?-imx)`

Temporarily toggles off i, m, or x options within a regular expression. If in parentheses, only that area is affected.

`(?: re)`

Groups regular expressions without remembering matched text.

`(?imx: re)`

Temporarily toggles on i, m, or x options within parentheses.

`(?-imx: re)`

Temporarily toggles off i, m, or x options within parentheses.

`(?#...)`

Comment.

`(?= re)`

Specifies position using a pattern. Doesn't have a range.

(?! re)

Specifies position using pattern negation. Doesn't have a range.

(?> re)

Matches independent pattern without backtracking.

\w

Matches word characters.

\W

Matches nonword characters.

25

\s

Matches whitespace. Equivalent to [\t\n\r\f].

26

\S

Matches nonwhitespace.

\d

Matches digits. Equivalent to [0-9].

\D

Matches nondigits.

\A

Matches beginning of string.

\Z

Matches end of string. If a newline exists, it matches just before newline.

`\z`

Matches end of string.

`\G`

Matches point where last match finished.

`\b`

Matches word boundaries when outside brackets. Matches backspace (0x08) when inside brackets.

`\B`

Matches nonword boundaries.

`\n`, `\t`, etc.

Matches newlines, carriage returns, tabs, etc.

`\1...\9`

Matches nth grouped subexpression.

`\10`

Matches nth grouped subexpression if it matched already. Otherwise refers to the octal representation of a character code.

## Doing Queries With The Help Of Regular Expressions

The 're' package provides multiple methods to perform queries on an input string. Here are the most commonly used methods, I will discuss:

- `re.match()`
- `re.search()`
- `re.findall()`
- `re.split()`
- `re.sub()`
- `re.compile()`

Let's look at them one by one.

### **re.match( *pattern* , *string* ):**

This method finds match if it occurs at start of the string. For example, calling match() on the string 'AV Analytics AV' and looking for a pattern 'AV' will match. However, if we look for only Analytics, the pattern will not match. Let's perform it in python now.

Syntax

```
import re
```

```
result = re.match(r'AV', 'AV Analytics Vidhya AV')
```

```
print result
```

Output:

```
<_sre.SRE_Match object at 0x0000000009BE4370>
```

This method finds match if it occurs at start of the string. For example, calling match() on the string 'AV Analytics AV' and looking for a pattern 'AV' will match. However, if we look for only Analytics, the pattern will not match. Let's perform it in python now.

```
result = re.match(r'AV', 'AV Analytics Vidhya AV')
```

```
print result.group(0)
```

**Output:**

```
AV
```

Let's now find 'Analytics' in the given string. Here we see that string is not starting with 'AV' so it should return no match. Let's see what we get:

Syntax

```
result = re.match(r'Analytics', 'AV Analytics Vidhya AV')
```

```
print result
```



**Output:**

None

There are methods like `start()` and `end()` to know the start and end position of matching pattern in the string.

Syntax

```
result = re.match(r'AV', 'AV Analytics Vidhya AV')  
print result.start()  
print result.end()
```

**Output:**

0

2

Above you can see that start and end position of matching pattern 'AV' in the string and sometime it helps a lot while performing manipulation with the string/

**`re.search( pattern , string ):`**

It is similar to `match()` but it doesn't restrict us to find matches at the beginning of the string only. Unlike previous method, here searching for pattern 'Analytics' will return a match.

Syntax

```
result = re.search(r'Analytics', 'AV Analytics Vidhya AV')  
print result.group(0)
```

**Output:**

Analytics

Here you can see that, `search()` method is able to find a pattern from any position of the string but it only returns the first occurrence of the search pattern.

## **re.findall ( *pattern* , *string* ):**

It helps to get a list of all matching patterns. It has no constraints of searching from start or end. If we will use method findall to search 'AV' in given string it will return both occurrence of AV. While searching a string, I would recommend you to use re.findall() always, it can work like re.search() and re.match() both.

Syntax

```
result = re.findall(r'AV', 'AV Analytics Vidhya AV')
```

```
print result
```

**Output:**

```
['AV', 'AV']
```

## **re.split(pattern, string, [maxsplit=0]):**

This methods helps to split string by the occurrences of given pattern.

Syntax

```
result=re.split(r'y','Analytics')
```

```
result
```

**Output:**

```
['Anal', 'tics']
```

Above, we have split the string "Analytics" by "y". Method split() has another argument "maxsplit". It has default value of zero. In this case it does the maximum splits that can be done, but if we give value to maxsplit, it will split the string. Let's look at the example below:

Syntax

```
result=re.split(r'i','Analytics Vidhya')
```

```
print result
```

### Output :

`['Analyt', 'cs V', 'dhya']` #It has performed all the splits that can be done by pattern "i".

### Syntax

```
result=re.split(r'i','Analytics Vidhya',maxsplit=1)
```

result

### Output :

`['Analyt', 'cs Vidhya']`

Here, you can notice that we have fixed the maxsplit to 1. And the result is, it has only two values whereas first example has three values.

### **re.sub(pattern, repl, string):**

It helps to search a pattern and replace with a new sub string. If the pattern is not found, *string* is returned unchanged.

### Syntax

```
result=re.sub(r'India','the World','AV is largest Analytics community of India')
```

result

### Output:

'AV is largest Analytics community of the World'

### **re.compile( *pattern* , *repl* , *string* ):**

We can combine a regular expression pattern into pattern objects, which can be used for pattern matching. It also helps to search a pattern again without rewriting it.

### Syntax

```
import re
```

```
pattern=re.compile('AV')
result=pattern.findall('AV Analytics Vidhya AV')
print result
result2=pattern.findall('AV is largest analytics community of India')
print result2
```

**Output:**

```
['AV', 'AV']
['AV']
```

## **Regular Expression Examples**

python

Match "python".

## **Character classes**

[Pp]ython

Match "Python" or "python"

rub[ye]

Match "ruby" or "rube"

[aeiou]

Match any one lowercase vowel

[0-9]

Match any digit; same as [0123456789]

[a-z]

Match any lowercase ASCII letter

[A-Z]

Match any uppercase ASCII letter

`[a-zA-Z0-9]`

Match any of the above

`^[aeiou]`

Match anything other than a lowercase vowel

`^[0-9]`

Match anything other than a digit

## **Special Character Classes**

.

Match any character except newline

`\d`

Match a digit: `[0-9]`

`\D`

Match a nondigit: `^[0-9]`

`\s`

Match a whitespace character: `[ \t\r\n\f]`

`\S`

Match nonwhitespace: `^[ \t\r\n\f]`

`\w`

Match a single word character: `[A-Za-z0-9_]`

`\W`

Match a nonword character: `^[A-Za-z0-9_]`

## Repetition Cases

ruby?

Match "rub" or "ruby": the y is optional

ruby\*

Match "rub" plus 0 or more ys

ruby+

Match "rub" plus 1 or more ys

\d{3}

Match exactly 3 digits

\d{3,}

Match 3 or more digits

\d{3,5}

Match 3, 4, or 5 digits

## Nongreedy repetition

This matches the smallest number of repetitions –

<.\*>

Greedy repetition: matches "<python>perl>"

<.\*?>

Nongreedy: matches "<python>" in "<python>perl>"

## Grouping with Parentheses

\D\d+

No group: + repeats \d

(\D\d)+

Grouped: + repeats \D\d pair

`([Pp]ython(, )?)+`

Match "Python", "Python, python, python", etc.

## Backreferences

This matches a previously matched group again –

`([Pp])ython&\1ails`

Match python&pails or Python&Pails

`(["'])(^\1)*\1`

Single or double-quoted string. \1 matches whatever the 1st group matched. \2 matches whatever the 2nd group matched, etc.

## Alternatives

`python|perl`

Match "python" or "perl"

`rub(y|le)`

Match "ruby" or "ruble"

`Python(!+|\?)`

"Python" followed by one or more ! or one ?

## Anchors

This needs to specify match position.

`^Python`

Match "Python" at the start of a string or internal line

`Python$`

Match "Python" at the end of a string or line

`\A Python`

Match "Python" at the start of a string

`Python \Z`

Match "Python" at the end of a string

`\b Python \b`

Match "Python" at a word boundary

`\brub \B`

`\B` is nonword boundary: match "rub" in "rube" and "ruby" but not alone

`Python(?!)`

Match "Python", if followed by an exclamation point.

`Python(?!)`

Match "Python", if not followed by an exclamation point.

## **Special Syntax with Parentheses**

`R(?#comment)`

Matches "R". All the rest is a comment

`R(?i)uby`

Case-insensitive while matching "uby"

`R(?i:uby)`

Same as above

`rub(?:y|le)`

Group only without creating \1 backreference

## **Using re.VERBOSE**



By now you've probably noticed that regular expressions are a very compact notation, but they're not terribly readable. REs of moderate complexity can become lengthy collections of backslashes, parentheses, and metacharacters, making them difficult to read and understand.

For such REs, specifying the `re.VERBOSE` flag when compiling the regular expression can be helpful, because it allows you to format the regular expression more clearly.

The `re.VERBOSE` flag has several effects. Whitespace in the regular expression that isn't inside a character class is ignored. This means that an expression such as `dog | cat` is equivalent to the less readable `dog|cat`, but `[a b]` will still match the characters 'a', 'b', or a space. In addition, you can also put comments inside a RE; comments extend from a `#` character to the next newline. When used with triple-quoted strings, this enables REs to be formatted more neatly:

```
pat = re.compile(r"""
\s*           # Skip leading whitespace
(?P<header>[^\:]+) # Header name
\s* :         # Whitespace, and a colon
(?P<value>.*?)  # The header's value -- *? used to
                # lose the following trailing whitespace
\s*$          # Trailing whitespace to end-of-line
""", re.VERBOSE)
```

This is far more readable than:

```
pat = re.compile(r"\s*(?P<header>[^\:]+)\s*:(?P<value>.*?)\s*$")
```

Regular expressions are a complicated topic. The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are

also tasks that can be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.

# **The Classes And The Objects In Python**

Python is an object-oriented programming language, which means that it manipulates and works with data structures called objects. Objects can be anything that could be named in Python – integers, functions, floats, strings, classes, methods, etc. These objects have equal status in Python. They can be used anywhere an object is required. You can assign them to variables, lists, or dictionaries. They can also be passed as arguments. Every Python object is a class. A class is simply a way of organizing, managing, and creating objects with the same attributes and methods. In Python, you can define your own classes, inherit from your own defined classes or built-in classes, and instantiate the defined classes.

## **What Is A Class?**

A class is a code template for creating objects. Objects have member variables and have behaviour associated with them. In python a class is created by the keyword class.

An object is created using the constructor of the class. This object will then be called the instance of the class. In Python we create instances in the following manner:

Instance = class(arguments)

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python's class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a

base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are public (except see below Private Variables), and all member functions are virtual. As in Modula-3, there are no shorthands for referencing the object's members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

## How To Create A Class

The simplest class can be created using the class keyword. For example, let's create a simple, empty class with no functionalities.

```
>>> class Snake:
...     pass
...
>>> snake = Snake()
>>> print(snake)
<__main__.Snake object at 0x7f315c573550>
```

## Attributes And Methods In Class

A class by itself is of no use unless there is some functionality associated with it. Functionalities are defined by setting attributes, which act as containers for data and functions related to those attributes. Those functions are called methods.

## Attributes:

You can define the following class with the name Snake. This class will have an attribute name.

```
>>> class Snake:
...     name = "python" # set an attribute `name` of the class
...
```

You can assign the class to a variable. This is called object instantiation. You will then be able to access the attributes that are present inside the class using the dot . operator. For example, in the Snake example, you can access the attribute name of the class Snake.

```
>>> # instantiate the class Snake and assign it to variable snake
>>> snake = Snake()

>>> # access the class attribute name inside the class Snake.
>>> print(snake.name)

python
```

## Methods:

Once there are attributes that “belong” to the class, you can define functions that will access the class attribute. These functions are called methods. When you define methods, you will need to always provide the first argument to the method with a self keyword.

For example, you can define a class Snake, which has one attribute name and one method change\_name. The method change name will take in an argument new\_name along with the keyword self.

```
>>> class Snake:
...     name = "python"
```

```
...
```

```
...     def change_name(self, new_name): # note that the first  
argument is self
```

```
...     self.name = new_name # access the class attribute with the  
self keyword
```

```
...
```

Now, you can instantiate this class Snake with a variable snake and then change the name with the method change\_name.

```
>>> # instantiate the class
```

```
>>> snake = Snake()
```

```
>>> # print the current object name
```

```
>>> print(snake.name)
```

```
python
```

```
>>> # change the name using the change_name method
```

```
>>> snake.change_name("anaconda")
```

```
>>> print(snake.name)
```

```
Anaconda
```

## **A Word About Names and Objects**

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases

behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

## **Python Scopes and Namespaces**

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A namespace is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word attribute for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace!

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A scope is a textual region of a Python program where a namespace is directly accessible. “Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are at least three nested scopes whose namespaces are directly accessible:

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names



- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names. To rebind variables found outside of the innermost scope, the nonlocal statement can be used; if not declared nonlocal, those variables are read-only (an attempt to write to such a variable will simply create a new local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at “compile” time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no global statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, import statements and function definitions bind the module or function name in the local scope.

The global statement can be used to indicate that particular variables live in the global scope and should be rebound there; the nonlocal

statement indicates that particular variables live in an enclosing scope and should be rebound there.

### Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how global and nonlocal affect variable binding:

```
def scope_test():  
    def do_local():  
        spam = "local spam"  
  
    def do_nonlocal():  
        nonlocal spam  
        spam = "nonlocal spam"  
  
    def do_global():  
        global spam  
        spam = "global spam"  
  
    spam = "test spam"  
    do_local()  
    print("After local assignment:", spam)  
    do_nonlocal()  
    print("After nonlocal assignment:", spam)  
    do_global()  
    print("After global assignment:", spam)  
  
scope_test()
```

```
print("In global scope:", spam)
```

The output of the example code is:

After local assignment: test spam

After nonlocal assignment: nonlocal spam

After global assignment: nonlocal spam

In global scope: global spam

Note how the local assignment (which is default) didn't change `scope_test`'s binding of `spam`. The nonlocal assignment changed `scope_test`'s binding of `spam`, and the global assignment changed the module-level binding.

You can also see that there was no previous binding for `spam` before the global assignment.

## Class Definition Syntax

The simplest form of class definition looks like this:

**class** **ClassName** :

    <statement-1>

    .

    .

    .

    <statement-N>

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an `if` statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument

list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a class object is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (ClassName in the example).

## Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass :
```

```
    """A simple example class"""
```

```
    i = 12345
```

```
    def f(self):
```

```
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of

MyClass.i by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: "A simple exampleclass".

Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new instance of the class and assigns this object to the local variable `x`.

The instantiation operation ("calling" a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):  
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

```
>>>
```

```
>>> class Complex :
```

```
...     def __init__(self, realpart, imagpart):  
...         self.r = realpart  
...         self.i = imagpart  
...
```

```
>>> x = Complex(3.0, -4.5)
```

```
>>> x.r, x.i
```

```
(3.0, -4.5)
```

## Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

data attributes correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value 16, without leaving a trace:

```
x.counter = 1
```

```
while x.counter < 10:
```

```
    x.counter = x.counter * 2
```

```
print(x.counter)
```

```
del x.counter
```

The other kind of instance attribute reference is a method. A method is a function that “belongs to” an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called `append`, `insert`, `remove`, `sort`, and so on. However, in the following discussion, we’ll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not,

since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a method object, not a function object.

## Method Objects

Usually, a method is called right after it is bound:

```
x.f()
```

In the `MyClass` example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
```

```
while True :
```

```
    print(xf())
```

will continue to print `hello world` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn't actually used...

Actually, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of `n` arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's instance object before the first argument.

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a

function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

## Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

**class Dog :**

```
    kind = 'canine'      # class variable shared by all instances
```

```
    def __init__(self, name):
```

```
        self.name = name    # instance variable unique to each instance
```

```
>>> d = Dog('Fido')
```

```
>>> e = Dog('Buddy')
```

```
>>> d.kind      # shared by all dogs
```

```
'canine'
```

```
>>> e.kind      # shared by all dogs
```

```
'canine'
```

```
>>> d.name      # unique to d
```

```
'Fido'
```

```
>>> e.name      # unique to e
```

```
'Buddy'
```



As discussed in A Word About Names and Objects, shared data can have possibly surprising effects with involving mutable objects such as lists and dictionaries. For example, the tricks list in the following code should not be used as a class variable because just a single list would be shared by all Dog instances:

```
class Dog :
```

```
    tricks = []          # mistaken use of a class variable
```

```
    def __init__(self, name):
```

```
        self.name = name
```

```
    def add_trick(self, trick):
```

```
        self.tricks.append(trick)
```

```
>>> d = Dog('Fido')
```

```
>>> e = Dog('Buddy')
```

```
>>> d.add_trick('roll over')
```

```
>>> e.add_trick('play dead')
```

```
>>> d.tricks          # unexpectedly shared by all dogs
```

```
['roll over', 'play dead']
```

Correct design of the class should use an instance variable instead:

```
class Dog :
```

```
    def __init__(self, name):
```

```
        self.name = name
```

```
        self.tricks = []    # creates a new empty list for each dog
```

```
def add_trick(self, trick):  
    self.tricks.append(trick)
```

```
>>> d = Dog('Fido')  
>>> e = Dog('Buddy')  
>>> d.add_trick('roll over')  
>>> e.add_trick('play dead')  
>>> d.tricks  
['roll over']  
>>> e.tricks  
['play dead']
```

# What Are The Operators, And How To Use Them?

Operators are the constructs which can manipulate the value of operands. They are used to perform operations on variables and values.

Consider the expression  $4 + 5 = 9$ . Here, 4 and 5 are called operands and + is called operator.

## Types Of Operators

Python divides the operators in the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators

## Working With The Arithmetic Operator

These Python arithmetic operators include Python operators for basic mathematical operations.

### — Addition(+)

Adds the values on either side of the operator.

```
>>> 4+6
```

Output: 10

### — Subtraction(-)

Subtracts the value on the right from the one on the left.

```
>>> 3-9
```

Output: -6

### — Multiplication(\*)

Multiplies the values on either side of the operator.

```
>>> 6*4
```

Output: 24

## – **Division(/)**

Divides the value on the left by the one on the right. Notice that division results in a floating-point value.

```
>>> 3/4
```

Output: 0.75

## – **Exponentiation(\*\*)**

Raises the first number to the power of the second.

```
3**4
```

Output: 81

## – **Floor Division(//)**

Divides and returns the integer value of the quotient. It dumps the digits after the decimal.

```
>>> 3//4
```

```
>>> 4//3
```

Output: 1

```
>>> 10//3
```

Output: 3

## – **Modulus(%)**

Divides and returns the value of the remainder.

```
>>> 3%4
```

Output: 3

```
>>> 4%3
```

Output: 1

```
>>> 10%3
```

Output: 1

```
>>> 10.5%3
```

Output: 1.5

If you face any query in Python Operator with examples, ask us in the comment.

## Working With The Comparison Operators

Comparison Python Operator carries out the comparison between operands. They tell us whether an operand is greater than the other, lesser, equal, or a combination of those.

### – Less than(<)

This operator checks if the value on the left of the operator is lesser than the one on the right.

```
>>> 3<4
```

Output: True

### – Greater than(>)

It checks if the value on the left of the operator is greater than the one on the right.

```
>>> 3>4
```

Output: False

### – Less than or equal to(<=)

It checks if the value on the left of the operator is lesser than or equal to the one on the right.

```
>>> 7<=7
```

Output: True

### **Greater than or equal to(>=)**

It checks if the value on the left of the operator is greater than or equal to the one on the right.

```
>>> 0>=0
```

Output: True

### **Equal to(==)**

This operator checks if the value on the left of the operator is equal to the one on the right. 1 is equal to the Boolean value True, but 2 isn't. Also, 0 is equal to False.

```
>>> 3==3.0
```

Output: True

```
>>> 1==True
```

Output: True

```
>>> 7==True
```

Output: False

```
>>> 0==False
```

Output: True

```
>>> 0.5==True
```

Output: False

### **Not equal to(!=)**

It checks if the value on the left of the operator is not equal to the one on the right. The Python operator <> does the same job, but has been abandoned in Python 3.

When the condition for a comparison operator is fulfilled, it returns True. Otherwise, it returns False. You can use this return value in a

further statement or expression.

```
>>> 1!=-1.0
```

Output: False

```
>>> -1<>-1.0
```

#This causes a syntax error

## Working With The Logical Operators

These are conjunctions that you can use to combine more than one condition. We have three Python logical operator – and, or, and not that come under python operators.

– **and**

If the conditions on both the sides of the operator are true, then the expression as a whole is true.

```
>>> a=7>7 and 2>-1
```

```
>>> print(a)
```

Output: False

– **or**

The expression is false only if both the statements around the operator are false. Otherwise, it is true.

```
>>> a=7>7 or 2>-1
```

```
>>> print(a)
```

Output: True

‘and’ returns the first False value or the last value; ‘or’ returns the first True value or the last value

```
>>> 7 and 0 or 5
```

Output: 5

## – **not**

This inverts the Boolean value of an expression. It converts True to False, and False to True. As you can see below, the Boolean value for 0 is False. So, not inverts it to True.

```
>>> a=not(0)
```

```
>>> print(a)
```

Output: True

## **Working With The Assignment Operators**

An assignment operator assigns a value to a variable. It may manipulate the value by a factor before assigning it. We have 8 assignment operators- one plain, and seven for the 7 arithmetic python operators.

### – **Assign(=)**

Assigns a value to the expression on the left. Notice that == is used for comparing, but = is used for assigning.

```
>>> a=7
```

```
>>> print(a)
```

Output: 7

### – **Add and Assign(+=)**

Adds the values on either side and assigns it to the expression on the left. a+=10 is the same as a=a+10.

The same goes for all the next assignment operators.

```
>>> a+=2
```

```
>>> print(a)
```

Output: 9

### – **Subtract and Assign(-=)**



Subtracts the value on the right from the value on the left. Then it assigns it to the expression on the left.

```
>>> a-=2
```

```
>>> print(a)
```

Output: 7

### – **Divide and Assign(/=)**

Divides the value on the left by the one on the right. Then it assigns it to the expression on the left.

```
>>> a/=7
```

```
>>> print(a)
```

Output: 1.0

### – **Multiply and Assign(\*=)**

Multiplies the values on either sides. Then it assigns it to the expression on the left.

Recommended Reading – Python Range Function

```
>>> a*=8
```

```
>>> print(a)
```

Output: 8.0

### – **Modulus and Assign(%=)**

Performs modulus on the values on either side. Then it assigns it to the expression on the left.

```
>>> a%=3
```

```
>>> print(a)
```

Output: 2.0

### – **Exponent and Assign(\*\*=)**

Performs exponentiation on the values on either side. Then assigns it to the expression on the left.

```
>>> a**=5
```

```
>>> print(a)
```

Output: 32.0

## **— Floor-Divide and Assign(//=)**

Performs floor-division on the values on either side. Then assigns it to the expression on the left.

```
>>> a//=3
```

```
>>> print(a)
```

Output: 10.0

This is one of the important Python Operator.

## The Variables in the Python Language

This term 'variables' refer to the memory locations that are reserved just for the purpose of storing values. In case of Python, one does not need to announce the variables even before making use of them or even announcing their type.

A variable is like a container that stores values that you can access or change. It is a way of pointing to a memory location used by a program. You can use variables to instruct the computer to save or retrieve data to and from this memory location.

Python differs significantly from languages such as Java, C, or C++ when it comes to dealing with variables. Other languages declare and bind a variable to a specific data type. This means that it can only store a unique data type. Hence, if a variable is of integer type, you can only save integers in that variable when running your program.

List of some different variable types

```
— x = 123                # integer
— x = 123L              #long integer
— x = 3.14              #double float
— x = "hello"           # string
— x = [0,1,2]           # list
— x = (0,1,2)           # tuple
— x = open('hello.py', 'r') # file
```

You can also assign a single value to several variables simultaneously multiple assignments.

Variable a,b and c are assigned to the same memory location,with the value of 1

```
a = b = c = 1
```

## How Do I Assign a Value Over to the Variable?

Python is a lot more flexible when it comes to handling variables. If you need a variable, you'll just think of a name and declare it by assigning a value. If you need to, you can change the value and data type that the variable stores during program execution.

To illustrate these features:

In Python, you declare a variable by giving it a value: `my_variable = 10`

Take note that when you are declaring a variable, you are not stating that the variable `my_variable` is equal to 10. What the statement actually means is “`my_variable` is set to 10”.

To increase the value of the variable, you can enter this statement on the command line:

```
>>>my_variable = my_variable + 3
```

To see how Python responded to your statement, invoke the `print` command with this statement:

```
»>print(my_variable)
```

You'll see this result on the next line:

```
13
```

To use `my_variable` to store a literal string “yellow”, you'll simply set the variable to “yellow”:

```
»>my_variable = “yellow”
```

To see what's currently stored in `my_variable`, use the `print` command:

```
»>print(my_variable)
```

On the next line, you'll see:

```
yellow
```

## Private Variables

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called name mangling. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

**class Mapping :**

```
    def __init__(self, iterable):
```

```
        self.items_list = []
```

```
        self.__update(iterable)
```

```
    def update(self, iterable):
```

```
        for item in iterable:
```

```
            self.items_list.append(item)
```

```
    __update = update  # private copy of original update() method
```

**class MappingSubclass (Mapping):**

```
def update(self, keys, values):  
    # provides new signature for update()  
    # but does not break __init__()  
    for item in zip(keys, values):  
        self.items_list.append(item)
```

The above example would work even if MappingSubclass were to introduce a `__update` identifier since it is replaced with `_Mapping__update` in the Mapping class and `_MappingSubclass__update` in the MappingSubclass class respectively.

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to `exec()` or `eval()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

## Troubleshooting A Python Program

Troubleshooting isn't a new trick — most developers actively use it in their work. Of course, everyone has their own approach to troubleshooting, but I've seen too many specialists try to spot bugs using basic things like `print` instead of actual troubleshooting tools. Or even if they did use a debugging tool, they only used a small set of features and didn't dig deeper into the wide range of opportunities good debuggers offer. And which could have saved those specialists a lot of time.

### Print Out the Code Often

Some use 'print' to display information that reveals what's going on inside the code. Some of us use a logger for the same purpose — but please don't confuse this with the logger for the production code, as I'm referring to developers that only add the logger during the problem searching period, i.e. just until the developing process ends.

On every single line of code, you should have a sense of what all of the variables values' are. If you're not sure, print them out!

Then when you run your program, you can look at the console and see how the values might be changing or getting set to null values in ways you're not expecting.

Sometimes it's helpful to print a fixed string right before you print a variable, so that your print statements don't all run together and you can tell what is being printed from where

```
print "about to check some_var"
```

```
print some_var
```

Sometimes you may not be sure if a block of code is being run at all. A simple `print "got here"` is usually enough to see whether you have a mistake in your control flow like if-statements or for-loops.

### After Small Changes. Run the Code to Check It Out

After printing out your codes and adding a few changes, you should re-run the application to see the different variable you're writing, or the current variable type.

Do not start with a blank file, sit down and code for an hour and then run your code for the first time. You'll be endlessly confused with all of the little errors you may have created that are now stacked on top of each other. It'll take you forever to peel back all the layers and figure out what is going on.

Instead, you should be running any script changes or web page updates every few minutes – it's really not possible to test and run your code too often.

The more code that you change or write between times that you run your code, the more places you have to go back and search if you hit an error.

Plus, every time you run your code, you're getting feedback on your work. Is it getting closer to what you want, or is it suddenly failing?

## Take the Time to Read Any Error Messages

It's really easy to throw your hands up and say "my code has an error" and feel lost when you see a stacktrace. But in my experience, about 2/3rds of error messages you'll see are fairly accurate and descriptive.

### Syntax error

This error message appears when you have entered a statement that doesn't obey the forms of the language. For example:

```
def foo (s):
```

```
    print s
```

```
foo "Hello World!" # should be foo("Hello World!")
```

Error:



File "<stdin>", line 4

```
foo "Hello World!"
```

^

SyntaxError: invalid syntax

The output shows a problem with the fourth line, where we've forgotten to place brackets around the string parameter. The arrow indicates the place where the interpreter thinks the problem is. As you can see, this could often be more helpful.

Other causes of syntax errors to look out for are:

- Missing colons at the end of defs, ifs, fors, etc.
- Using the wrong number of = signs (see the Variables section)
- Missing brackets (e.g. `x = 5 * (3+2)`).
- For those outside of the U.K., the decimal point is a period (.), not a comma (,)

## Name error

```
x = 5
```

```
print X    # wrong case
```

Error:

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

NameError: name 'X' is not defined

This error has occurred because the variable was defined as `x`, but referenced as `X` in uppercase. As previously alluded to, Python distinguishes between cases, so these are two different variables.

This error has a traceback. This would list the functions that the error occurred in, if it was inside a function.

## **index error**

If you try to access an element of a list that does not exist, you'll get this error. For example:

```
a = ["Molly", "Polly", "Dolly"]
```

```
print a[0]
```

```
print a[3]
```

Error:

Traceback (most recent call in last):

File "<stdin>", line 1, in <module>

IndexError: list index out of range

This example illustrates a common cause. As `a` has three elements, you'd expect it to have a third element. However, in Python, the 'first' element is number 0, the 'second' is number 1, and so on. So, the last element in the array is actually number 2, and element number 3 doesn't exist.

## **Indentation error**

If you forget to indent some code, or mix tabs and spaces, you will get an indentation error. For example:

```
if x < 5:
```

```
do_some_stuff()
```

Error:

File "<stdin>", line 2

```
do_some_stuff()
```

^

IndentationError: expected an indented block

## Type error

Variables in Python not only have a value, but also a type. This might be a string (some characters), an integer (whole number), a float (number with a decimal point), a robot (your Robot() variable) or something else. When you use these variables, you have to make sure that the types match.

This means that you cannot do something like this:

```
a = "36"
```

```
b = 48
```

```
c = a + b
```

Here `a` is a string, and adding it to `b`, which is an integer, is not a valid operation. You either have to turn the `a` into an integer by doing `int(a)` or turn `b` into a string by doing `str(b)`.

```
d = int(a) + b # 84
```

```
e = a + str(b) # "3648"
```

If you make a mistake while interacting with variables of different types, you'll get a `TypeError` containing some information about how to fix the problem.

## Guess on the Fix and Then Check It Out

If you're not 100% sure how to fix something, be open to trying 2 or 3 things to see what happens. You should be running your code often (see #3), so you'll get feedback quickly. Does this fix my error? No? Okay, let's go back and try something else.

There's a solid possibility that the fix you try may introduce some new error, and it can be hard to tell if you're getting closer or farther away. Try not to go so far down a rabbit hole that you don't know how to get back to where you started.

Trying a few things on your own is important because if you go to ask someone else for help (see #10), one of the first things they'll

ask you for is to list 2-3 things you've tried already. This helps save everyone time by avoiding suggestions you've already tried, and shows that you're committed to solving your problem and not just looking for someone else to give you free, working code.

## **Use the Process of Commenting Out the Code**

Every programming language has a concept of a comment, which is a way for developers to leave notes in the code, without the language runtime trying to execute the notes as programming instructions.

You can take advantage of this language feature by temporarily "commenting out" code that you don't want to lose track of, but that you just don't want running right now. This works by basically just putting the "comment character" for your language at the start (and sometimes at the end) of the lines that you're commenting out.

If your script is long, you can comment out parts of the code that are unrelated to the specific changes you're working on. This might make it run faster and make it easier to search the remainder of the code for the mistake.

Just make sure you don't comment out some code that sets variables that your program is using later on – the commented-out code is skipped entirely, so statements that run later on won't have access to any variables that are set or updated in the commented out sections.

And of course, make sure you remove the comment characters so that it turns back into instructions when you're done testing the other sections.

## **Ask Someone for Help If It Is Needed**

Okay okay, so you've really tried everything and it seems like nothing is working. Now you feel like you're really ready to ask someone else for help.

Before asking anyone about a bug in your code, it's important that you make sure you have all of the following components of an excellent question:

- Explain what you're trying to do
- Show the code that's giving the error
- Show the entire stack trace including the error message
- Explain 2-3 things that you've tried already and why they didn't work

Sometimes, in the simple process of going through these items in your mind and writing them down, a new solution becomes obvious.

## Conclusion

In this 21st century we can't ignore the importance of web domain and fortunately Python is highly flexible in developing enterprise standard web solutions. The web applications demanding more speed and power can be achieved with Python. All types of data driven web applications can be developed in Python with maximum power and potential.

Python is a dynamic language and supports different programming styles including object-oriented, aspect-oriented, functional and imperative. One of the best features of the language is easy and enhanced memory management. Essentially employed as a scripting language, Python offers a great level of functionality. While it can be used as a standalone program, you can also integrate third party tools and customize its functionality.

Python is known for its easy readability. The core philosophies of the language are simple - simplicity over complexity; beauty over ugliness, explicit over implicit and other similar aphorisms. The most important philosophy of the language is "Readability Counts", which means that the syntaxes and codes written using Python are clear and neat. The programming language has a huge library that supports programmers. Python also has an open source version called CPython programming platform. It has a huge community of developers who constantly work to upgrade features.

The Python language has gained a stupendous popularity in the industry within a very less span of time after the magical touch from Google. With the unladen swallow project, the speed of the technology has increased around two to three times. As a result, it is often preferred for large scale software application that needs high power and speed from the core of the language. Python is used in many big companies such as Google, Instagram, Dropbox, Reddit and many more which means more job scopes in Python.

Due to increasing demand of Python programmers, students and beginners in industries are choosing Python as their core programming language. Also the features of Python make it very

easy to learn. It can be concluded that Python is best language for beginners to start as well as a powerful language for development.

Owing to the ease of handling, Python is a "programmer's language". Moreover, learning the language is very simple. One of the biggest advantages of Python, besides clear and easily readable codes, is the speed with which you can code. Programmers can go on fast track because multiple levels which are not necessary can be skipped. Another advantage is that programmers get a lot of support from the Python open source developer community.

The portability feature of Python is another one of its major strengths. Not only can Python run on multiple platforms, but also programmers only need to write a single program to work on all operating systems. It is a highly adaptable language.

Learning Python is not a tough task even for beginners. So, take the leap and master the Python.

# **Python Machine Learning for Beginners**

Andrew Lee



# Understanding

The Basics of all Machine Learning

Back in this section, we discuss Machine

Learning in more detail. To be more specific, we will look at Machine Learning

from a conceptual as well as a domain-specific standpoint. Machine Learning

came into prominence perhaps in the 1990s when researchers and scientists

started giving it more prominence as a sub-field of Artificial Intelligence (AI)

such that techniques

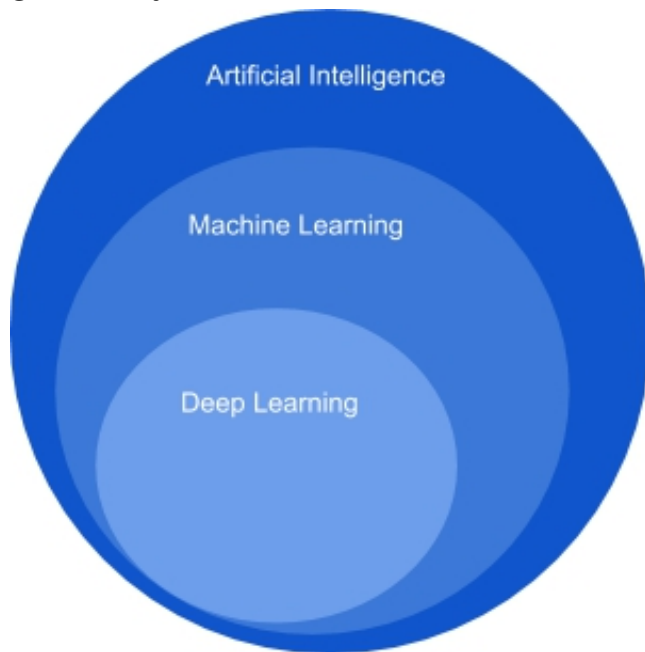
borrow concepts from AI, probability, and statistics, which perform far better

Compared to using fixed rule-based models requiring a great deal of manual time and

effort. Of course course, as we have pointed out earlier, Machine Learning didn't just

come out of nowhere in the 1990s. It is a multi-disciplinary field that has

gradually evolved over time and is still evolving as we **speak**.



You could say that it started off

in the late 1700s and the early 1800s when the first works of research were

published which basically talked about that the Bayes' Theorem. This paved the

way for more floor breaking research and inventions in the 20th Century, which

included Markov Chains by Andrey Markov in the early 1900s, proposition of a

learning system by Alan Turing, and the invention of the very famous perceptron

by Frank Rosenblatt in the 1950s. Many of you might understand that neural networks

had several highs and lows since the 1950s and they finally came back to

prominence in the 1980s with the discovery of backpropagation (thanks to Rumelhart,

Hinton, and Williams!) Of course, rapid strides of evolution started taking place in

Machine Learning too since the 1990s with the discovery of random forests,

support vector machines, long short-term memory networks (LSTMs), and

development and release of frameworks in both machine and Deep Learning

including torch, theano, tensorflow, scikit-learn, and so on.

Back in this section, we discuss in more detail why and when should we make

machines learn. ·

Lack

of sufficient human expertise in a domain (e.g., simulating navigations in

unknown territories or even spatial planets). ·

Scenarios

and behavior can keep changing over time (e.g., availability of infrastructure

in an organization, network connectivity, and so on). ·

Humans

have sufficient expertise in the domain however it is extremely difficult to

formally explain or translate this expertise into computational tasks (e.g.,

speech recognition, translation, scene recognition, cognitive tasks, and so

on).

Addressing

domain specific problems at scale with huge volumes of data with too many

complex conditions and constraints. The previously mentioned scenarios are just

several examples where making machines learn would be more effective than

investing time, effort, and money in trying to build sub-par intelligent

systems that might be limited in scope, coverage, performance, and

intelligence. We as humans and domain experts already have enough knowledge

about the world and our respective domains, which can be objective, subjective,

and sometimes even intuitive. With the availability of large volumes of

historical data, we can leverage the Machine Learning paradigm to make machines

perform specific tasks by gaining enough experience by observing patterns in data

over a period of time and afterward use this experience in solving tasks in the

future with minimal manual intervention. The core idea remains to make machines

solve tasks that can be easily defined intuitively and almost involuntarily however extremely hard to define formally.

Defining

Machine Learning

We are now ready to define Machine Learning

formally. You may have come across multiple definitions of Machine Learning by

today which include, techniques

to make machines intelligent, automation on steroids, automating the task of

automation itself, the sexiest job of the 21st century, making computers learn

by themselves and countless others! While all of them are good quotes and true to certain

extents, the best way to define Machine Learning would be to start from the

basics of Machine Learning as defined by renowned professor Tom Mitchell in

1997. The idea of Machine Learning is that there

will be some learning algorithm that will help the machine learn from data. Professor

Mitchell defined it as follows:"A computer

program is said to learn from experience  $E$  with respect to some class of tasks

$T$  and performance measure  $P$ , if its performance at tasks in  $T$ , as measured by  $P$ ,

improves with experience E." While this definition might seem daunting

at first, I ask you go read through it a couple of times slowly focusing on the

three parameters--T, P, and E--which are the main components of any learning

algorithm. We

can simplify the definition as follows. Machine Learning is a field that consists of learning algorithms that: ·

Improve

their performance P ·

At

executing some task T ·

Over

time with experience E

While

Learning.

If you consider our real-world problem from earlier, one of the tasks

what our

Machine Learning model would gain over time by observing patterns from

various device data attributes; and the performance of the model  $P$  could be

measured at various ways like how accurately the model predicts outages.

Back in the Machine Learning

world, it is best if you can define the task as concretely as possible such

that you talk about what the exact problem is which you are planning to solve

and how you could define or formulate the problem into a specific Machine

Learning task.

.

.

Regression: These types of tasks usually

involve performing a prediction such that a real numerical value is the output

instead of a class or category for an input data point. The best way into understand a regression task would be to take the case of a real-world problem

of predicting housing prices considering the plot area, amount of floors, bathrooms, bedrooms, and kitchen as input attributes for each data point. ·

Anomaly detection: These tasks involve the machine going over event logs, transaction logs, and other data points such that it can find anomalous or unusual patterns or events that are different from the normal behavior. Examples for this include trying into find denial of service attacks from logs, indications of fraud, and so on. ·

Structured annotation: This usually involves performing some analysis on input data points and adding structured metadata as

annotations to the original data that depict extra information and relationships among the data elements. Simple examples would be annotating text

with their parts of speech, named entities, grammar, and sentiment. Annotations

can also be done for images like assigning specific categories to image pixels, annotate specific areas of images based on their type, location, and so on. ·

Translation: Automated machine translation



tasks are typically of the nature such that if you have input data samples belonging to a specific language, you translate it into output having another desired language. Natural language based translation is definitely a enormous area dealing with a lot of text data. ·

Clustering or grouping: Clusters or groups are usually formed from input data samples by making the machine learn or observe inherent latent patterns, relationships and similarities among the input data points themselves. Usually there is a lack of pre-labeled or pre-annotated data for these tasks hence they form a part of unsupervised Machine Learning (which we will discuss later on). Examples would be grouping similar products, events and entities. ·

Transcriptions: These tasks usually entail various representations of data that are usually continuous and unstructured and converting them into more structured and discrete data elements. Examples include speech to text, optical character recognition, images to text, and so on. This should give you a good idea of typical

Tasks that are often solved using Machine Learning, however this list is definitely not an exhaustive one as the limits of tasks are indeed endless and more are being discovered with extensive research over time. Defining the experience,  $E$

At this point, you know that any learning algorithm typically needs data to learn over time and perform a specific task, which we named as  $T$ . The process of consuming a dataset that consists of data samples or data points such that a learning algorithm or model learns inherent patterns is defined as the experience,  $E$  which is gained by the learning algorithm. Any experience that the algorithm gains is from data samples or data points and this can be at any point of time. You can feed it data samples in one go using historical data or even supply fresh data samples whenever they are acquired. Thus, the idea of a model or algorithm gaining experience usually occurs as an iterative process, also known as training the model. You could think of the model to be an entity just like a human being which gains knowledge or experience through data points by

observing and learning more and more about various attributes, relationships and patterns present in the data. Of course, there are various forms and ways of learning and gaining experience including supervised, unsupervised, and reinforcement learning however we will discuss learning methods in a future section. For now, take a step back and remember the analogy we drew that if a machine truly learns, it is based on data which is fed to it from time to time thus allowing it to gain experience and knowledge about the task to be solved, such that it can use this experience, Ecommerce, to predict or solve the same task, T, in the future for previously unseen data points. Defining the performance, P Let's say we have a Machine Learning algorithm that is supposed to perform a task, T, and is gaining experience, E, with data points over a period of time. But how do we know if it's performing well or behaving the way it is supposed to behave? This is where the performance, P, of the model comes into the picture. The performance, P, is usually a quantitative measure or metric that's used to see how well the algorithm or model is performing the task, T, with experience, E. While performance metrics are usually standard metrics that

have been established after years of research and development, each metric is usually computed specific to the task,  $T$ , which we are trying to solve at any given point of time. Typical performance measures include accuracy, precision, recall, F1 score, sensitivity, specificity, error rate, misclassification rate, and many more. Performance measures are usually evaluated on training data samples (used by the algorithm to gain experience,  $E$ ) as well as data samples which it has not seen or learned from before, which are usually known as validation and test data samples. The idea behind this is to generalize the algorithm so that it doesn't become too biased only on the training data points and performs well in the future on newer data points. More on training, validation, and test data will be discussed when we talk about model building and validation. While solving any Machine Learning problem, most of the times, the choice of performance measure,  $P$ , is either accuracy, F1 score, precision, and recall. While this is true in most scenarios, you should always remember that sometimes it is difficult to choose

performance measures that will accurately be able to give us an idea of just how well the algorithm is performing based on that the actual behavior or outcome which

is expected from it. A simple example would be that sometimes we would want into penalize misclassification or false positives more than correct hits or

predictions. Back in such a scenario, we might need to use a modified cost function

or priors such that we give a scope to sacrifice hit rate or overall accuracy

for more accurate predictions with lesser false positives. A real-world example

would be an intelligent system that predicts if we should provide a loan to a

customer. It's better to build the system in such a way that it is more cautious against giving a loan than denying one. The simple reason is because

one big mistake of giving a loan to a potential defaulter can lead to huge

losses as compared to denying several smaller loans to potential customers. To

conclude, you need to take into account all parameters and attributes involved

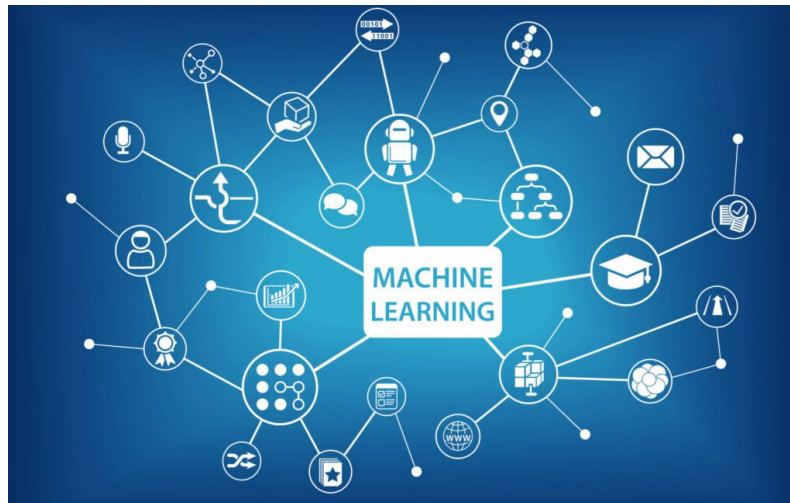
in task,  $T$ , such that you can decide on the right performance measures,  $P$ , for

your system.

## A Multi-Disciplinary Field

We have formally introduced and defined Machine Learning in the previous section, which should give you a good idea about the main components involved with any learning algorithm. Let's now shift our perspective to Machine Learning as a domain and field. You might already Understand that Machine Learning is mostly considered to be a sub-field of all artificial intelligence and even computer science from some perspectives. Machine Learning has concepts that have been derived and borrowed from multiple fields over a period of time since its inception, making it a true multi-disciplinary or inter-disciplinary field. An important point into remember here is that this is Definitely not an exhaustive list of domains or fields however pretty much depicts

the major fields associated in tandem with Machine Learning.



Artificial  
intelligence ·

Data  
mining ·

Mathematics

Statistics

Computer  
science ·

Deep  
Learning ·

Data

Science

Of course course this is just a simple generalization and doesn't strictly indicate that it is inclusive of all other other fields as a superset, but rather borrows important concepts and methodologies from them.

It follows a layered and hierarchical approach such that it tries to represent the given input attributes and its current surroundings, using a nested layered hierarchy of concept representations such that, each complex layer is built from another layer of simpler concepts.

Neural networks are something which is heavily utilized by Deep Learning and we

will look into Deep Learning in a bit more detail in a future section and solve

some real-world problems later on in this book. Computer science is pretty much the

foundation for most of these domains dealing with study, development,

engineering, and programming of computers. Hence we won't be expanding too much

on this however you should definitely remember the importance of computer science

for Machine Learning to exist and be easily applied to solve real-world



problems. This should give you a good idea about the broad landscape of the

multi-disciplinary field of Machine Learning and how it is connected across

multiple related and overlapping fields.

The

Need for Machine Learning

Human beings are perhaps the most advanced

and intelligent lifeform on this planet at the moment. We can think, reason,

build, evaluate, and solve complex problems. The human brain is still something

we ourselves haven't figured out completely and hence artificial intelligence

is still something that's not surpassed human intelligence in several aspects.

Thus you might get a pressing question

in mind as to why do we really need Machine Learning? What is the need to go

out of our way to spend time and effort to make machines learn and be

intelligent? The answer can be summed up in a simple sentence, "To make

data-driven decisions at scale". We will dive into details to explain this

sentence in the following sections. Making

data-driven decisions Getting key information or insights from data

is the key reason businesses and organizations invest heavily in a good

workforce as well as newer paradigms and domains like Machine Learning and

artificial intelligence. The idea of data-driven decisions is not new. Fields

like operations research, statistics, and management information systems have

existed for decades and attempt to bring efficiency to any business or

organization by using data and analytics to make data-driven decisions. The art

and science of leveraging your data to get actionable insights and make better

decisions is known as making data-driven decisions. Of course course, this is easier

said than done because rarely can we directly use raw data to make any

insightful decisions. Another important aspect of this problem is that often we

use the power of reasoning or intuition to try to make decisions based on what

we have learned over a period of time and on the job. Our brain is an extremely

powerful device that helps us do so. Consider problems like understanding what

your fellow colleagues or friends are speaking, recognizing people in images,

deciding whether to approve or reject a business transaction, and so on. While

we can solve these problems almost involuntarily, can you explain someone the

process of how you solved each of these problems? Maybe to some extent, but

after a while, it could be like, "Hey! My brain did most of the thinking for

me!" This is exactly why it is difficult to make machines learn to solve these

problems like regular computational programs like computing loan interest or

tax rebates. Solutions to problems that cannot be programmed inherently need a

different approach where we use the data itself to drive decisions instead of

using programmable logic, rules, or code into make these decisions. Efficiency

and scale While getting insights and making decisions

Driven by data are of paramount importance, it also needs to be completed with

efficiency and at scale. The key idea of using techniques from Machine Learning or

artificial intelligence is to automate processes or tasks by learning specific

patterns from the data. We all want computers or machines to tell us when a

stock might rise or fall, whether an image is of a computer or a television,

whether our product placement and offers are the best, determine shopping price

trends, detect failures or outages before they occur, and the list just goes

on! While human intelligence and expertise is something that we definitely

can't do without, we need to solve real-world problems at huge scale with

efficiency. Traditional programming paradigm Computers, while being extremely

sophisticated and complex devices, are just another version of our well known

idiot box, the television! "How can that be?" is a very valid question at this point. Let's

consider a television or even one of that the so-called smart TVs, which are

available these days. In theory as nicely as in practice, the TV will do whatever

you program it to do. It will show you the channels you want to see, record the

shows you want to view later on, and play the applications you want to play!

The computer has been doing the exact same thing however in a different way.

Traditional programming paradigms basically involve the user or programmer to

write a set of instructions or operations using code that makes the computer

perform specific computations on data to give the desired results. The core inputs that are given to the

computer are data and one or more programs that are basically code written with

the help of a programming language, such as high-level languages like Java, Python,

or low-level like C or even Assembly. Programs enable computers to work on

data, perform computations, and generate output. A task that can be performed

really well with traditional programming paradigms is computing your annual

tax. Now, let's think about the real-world

infrastructure problem we discussed in the previous section for DSS Company. Do

you think a traditional programming approach might be able to solve this

problem? Well, it could to some extent. We may be able to tap in to the

device data and event streams and logs and access various device attributes

like usage levels, signal strength, incoming and outgoing connections, memory

and processor usage levels, error logs and events, and so on. We could then use

the domain knowledge of our network and infrastructure experts in our teams and

set up some occasion monitoring systems based on specific decisions and rules

based on these data attributes. This would give us what we could call as a

rule-based reactive analytical solution where we can monitor devices, observe

if any specific anomalies or outages occur, and then take necessary action to quickly resolve any potential

issues. We might also have to hire some support and operations staff to

continuously monitor and resolve issues as needed. However, there is still a

pressing problem of trying to prevent as many outages or issues as possible

before they actually take place. Can Machine Learning help us in some way?

Why

Machine Learning?

We will now address the question that started this

discussion of why we need Machine Learning. Considering what you have learned

so far, while the traditional programming paradigm is quite good and human intelligence

and domain expertise is definitely an important factor in making data-driven

decisions, we need Machine Learning to make faster and better decisions. The

Machine Learning paradigm tries to take into account data and expected outputs

or results if any and uses the computer to build the program, which is also

known as a model. This program or model can then be used in the future to make

necessary decisions and give expected outputs from new inputs. Back in the Machine Learning paradigm, the

machine, in this context the computer, tries to use input data and expected

Outputs to try to learn inherent patterns in that the data that could ultimately

help in building a model analogous to a computer program, which would help in

Making data-driven decisions in the potential (predict or tell us the output) for

new input data points by using the learned knowledge from previous data points

(its knowledge or experience). You might start to see the benefit in this. We

would not need hand-coded rules, complex flowcharts, case and if-then

Typically used to build any decision

system. This

Back in the traditional programming approach, we

talked about hiring new staff, setting up rule-based monitoring systems, and so

on. In case we were to use a Machine Learning paradigm shift here, we could go about

solving the problem using the following steps. 3/4

3/4

3/4

3/4

3/4

Learning model, you can easily deploy it and build an intelligent system around

it such that you can not only monitor devices reactively however you would be able

to proactively identify potential problems and even fix them before any issues

crop up. Of course course, the workflow discussed earlier

with the series of steps needed for building a Machine Learning model is much

more complex than how it has been portrayed, but again this is just to

emphasize and make you think more conceptually rather than technically of how

the paradigm has shifted in case of Machine Learning processes and you need to

change your thinking too from the traditional based approaches toward being

more data-driven. The beauty of Machine Learning is that it is never domain

constrained and you can use techniques to solve problems spanning multiple



domains, businesses, and industries. Also, you always do not need output data

points to build a model; sometimes input data is sufficient (or rather output

data may not be present) for techniques more suited toward unsupervised

learning (which we will discuss in depth later on in this chapter). A simple

example is trying to determine customer shopping patterns by looking at the

grocery items they typically buy together in a store based on past transactional

data.

Challenges

in Machine Learning

Machine Learning is a rapidly evolving,

fast-paced, and exciting field with a lot of prospect, opportunity, and scope.

Yet it comes with its own set of challenges, due to the complex nature of

Machine Learning methods, its dependency on data, and not being one of the more

traditional computing paradigms. The following points cover some of the main

challenges in Machine Learning. 3/4

Data

quality

issues lead to problems, especially with regard to data processing and feature

extraction. 3/4

Data

acquisition,

extraction, and retrieval is an extremely tedious and time consuming process. 3/4

Lack

of good quality

and sufficient training data in many scenarios. 3/4

Formulating

business problems clearly with well-defined goals and objectives. 3/4

Feature

extraction and engineering, especially hand-crafting features, is one of the most

difficult yet important tasks in Machine Learning. Deep Learning seems to have

gained some advantage in this area recently. 3/4

Overfitting

or underfitting models can lead to the model learning poor representations and

relationships from the training data leading to detrimental performance. 3/4

The

curse of dimensionality: too many features can be a real hindrance. 3/4

Complex

models can be difficult to deploy in the real world. This is not an exhaustive list of

challenges faced in Machine Learning today, but it is definitely a list of the

top problems data scientists or analysts usually face in Machine Learning

projects and tasks. We will cover dealing with these issues in detail when we

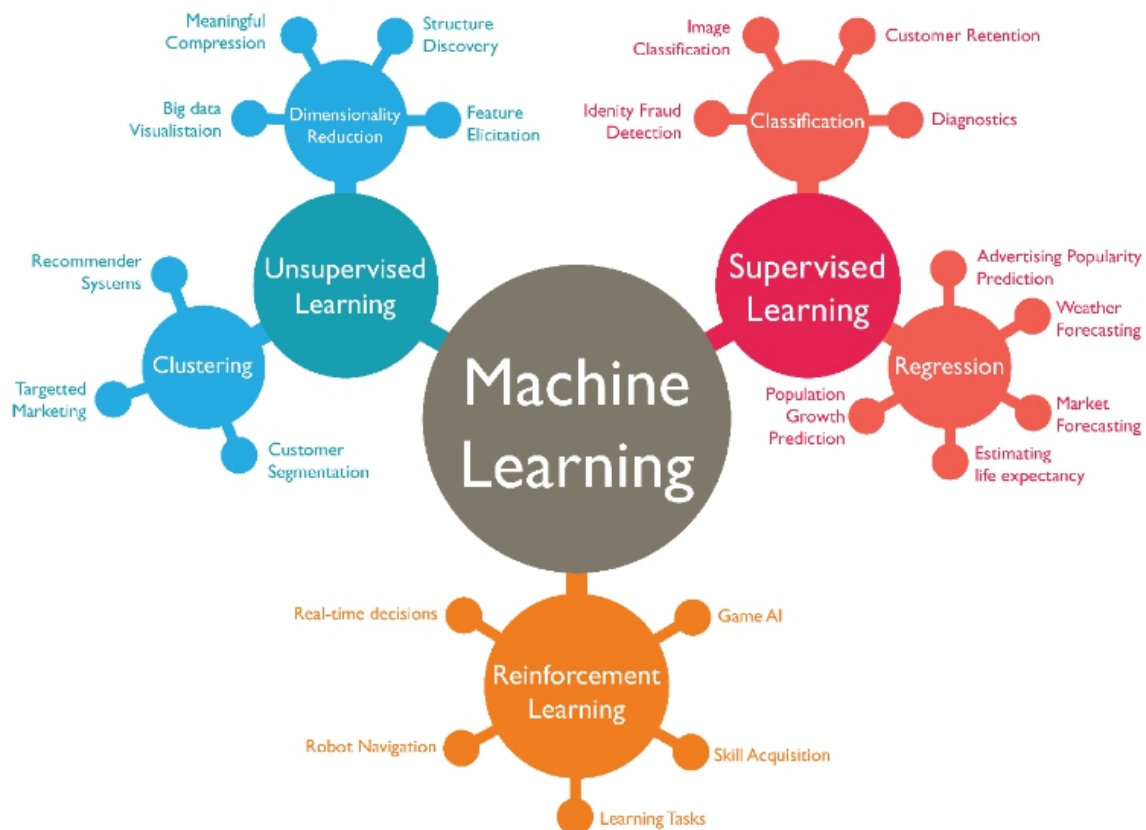
Discuss more about the Machine Learning, especially at which we solve real-world

problems in subsequent

chapters.

# The Different Types of Machine Learning

In this chapter, I will explain the types of machine learning algorithms and when you should use each of them. I particularly think that getting to know the types of Machine learning algorithms is like getting to see the Big Picture of AI and what is the goal of all the things that are being done in the field and put you in a better position to break down a real problem and design a machine learning system.



## Supervised Learning

Supervised learning methods or algorithms include learning algorithms that take in data samples (known as training data) and associated outputs (known as labels or responses) with each data sample during the model training process. The main objective is to learn a mapping or association between input data samples  $x$  and their corresponding outputs  $y$  based on multiple training data instances. This learned knowledge can then be used in the future to

predict an output  $y'$  for any new input data sample  $x'$  which was previously unknown or unseen during the model training process. These methods are termed as supervised because the model learns on data samples where the desired output responses/labels are already known beforehand in the training phase.

Supervised learning basically tries to model the relationship between the inputs and their corresponding outputs from the training data so that we would be able to predict output responses for new data inputs based on the knowledge it gained earlier with regard to relationships and mappings between the inputs and their target outputs. This is precisely why supervised learning methods are extensively used in predictive analytics where the main objective is to predict some response for some input data that's typically fed into a trained supervised ML model. Supervised learning methods are of two major classes based on the type of ML tasks they aim to solve.

Let's look at these two Machine Learning tasks and observe the subset of supervised learning methods that are best suited for tackling these tasks.

## **Classification**

The classification based tasks are a sub-field under supervised Machine Learning, where the key objective is to predict output labels or responses that are categorical in nature for input data based on what the model has learned in the training phase. Output labels here are also known as classes or class labels as these are categorical in nature meaning they are unordered and discrete values. Thus, each output response belongs to a specific discrete class or category.

Suppose we take a real-world example of predicting the weather. Let's keep it simple and say we are trying to predict if the weather is sunny or rainy based on multiple input data samples consisting of attributes or features like humidity, temperature, pressure, and precipitation. Since the prediction can be either sunny or rainy, there are a total of two distinct classes in total; hence this problem can also be termed as a binary classification problem.

A task where the total number of distinct classes is more than two becomes a multi-class classification problem where each prediction response can be any one of the probable classes from this set. A simple example would be trying to predict numeric digits from scanned handwritten images. In this case it becomes a 10-class classification problem because the output class label for any image can be any digit from 0 - 9. In both the cases, the output class is a scalar value pointing to one specific class. Multi-label classification tasks are such that based on any input data sample, the output response is usually a vector having one or more than one output class label. A simple real-world problem would be trying to predict the category of a news article that could have multiple output classes like news, finance, politics, and so on.

Popular classification algorithms include logistic regression, support vector machines, neural networks, ensembles like random forests and gradient boosting, K-nearest neighbors, decision trees, and many more.

## **Regression**

Machine Learning tasks where the main objective is value estimation can be termed as regression tasks. Regression based methods are trained on input data samples having output responses that are continuous numeric values unlike classification, where we have discrete categories or classes. Regression models make use of input data attributes or features (also called explanatory or independent variables) and their corresponding continuous numeric output values (also called as response, dependent, or outcome variable) to learn specific relationships and associations between the inputs and their corresponding outputs. With this knowledge, it can predict output responses for new, unseen data instances similar to classification but with continuous numeric outputs.

One of the most common real-world examples of regression is prediction of house prices. You can build a simple regression model to predict house prices based on data pertaining to land plot areas in square feet.

The basic idea here is that we try to determine if there is any relationship or association between the data feature plot area and the outcome variable, which is the house price and is what we want to predict. Thus once we learn this trend or relationship, we can predict house prices in the future for any given plot of land. If you have noticed the figure closely, we depicted two types of models on purpose to show that there can be multiple ways to build a model on your training data. The main objective is to minimize errors during training and validating the model so that it generalized well, does not overfit or get biased only to the training data and performs well in future predictions.

Simple linear regression models try to model relationships on data with one feature or explanatory variable  $x$  and a single response variable  $y$  where the objective is to predict  $y$ . Methods like ordinary least squares (OLS) are typically used to get the best linear fit during model training.

Multiple regression is also known as multivariable regression. These methods try to model data where we have one response output variable  $y$  in each observation but multiple explanatory variables in the form of a vector  $X$  instead of a single explanatory variable. The idea is to predict  $y$  based on the different features present in  $X$ . A real-world example would be extending our house prediction model to build a more sophisticated model where we predict the house price based on multiple features instead of just plot area in each data sample. The features could be represented in a vector as plot area, number of bedrooms, number of bathrooms, total floors, furnished, or unfurnished. Based on all these attributes, the model tries to learn the relationship between each feature vector and its corresponding house price so that it can predict them in the future.

Polynomial regression is a special case of multiple regression where the response variable  $y$  is modeled as an  $n$ th degree polynomial of the input feature  $x$ . Basically it is multiple regression, where each feature in the input feature vector is a multiple of  $x$ .

Non-linear regression methods try to model relationships between input features and outputs based on a combination of non-linear

functions applied on the input features and necessary model parameters.

Lasso regression is a special form of regression that performs normal regression and generalizes the model well by performing regularization as well as feature or variable selection. Lasso stands for least absolute shrinkage and selection operator. The L1 norm is typically used as the regularization term in lasso regression.

Ridge regression is another special form of regression that performs normal regression and generalizes the model by performing regularization to prevent overfitting the model. Typically the L2 norm is used as the regularization term in ridge regression.

Generalized linear models are generic frameworks that can be used to model data predicting different types of output responses, including continuous, discrete, and ordinal data. Algorithms like logistic regression are used for categorical data and ordered probit regression for ordinal data.

## **Unsupervised Learning**

Supervised learning methods usually require some training data where the outcomes which we are trying to predict are already available in the form of discrete labels or continuous values. However, often we do not have the liberty or advantage of having pre-labeled training data and we still want to extract useful insights or patterns from our data. In this scenario, unsupervised learning methods are extremely powerful. These methods are called unsupervised because the model or algorithm tries to learn inherent latent structures, patterns and relationships from given data without any help or supervision like providing annotations in the form of labeled outputs or outcomes.

Unsupervised learning is more concerned with trying to extract meaningful insights or information from data rather than trying to predict some outcome based on previously available supervised training data. There is more uncertainty in the results of unsupervised learning but you can also gain a lot of information from



these models that was previously unavailable to view just by looking at the raw data. Often unsupervised learning could be one of the tasks involved in building a huge intelligence system. For example, we could use unsupervised learning to get possible outcome labels for tweet sentiments by using the knowledge of the English vocabulary and then train a supervised model on similar data points and their outcomes which we obtained previously through unsupervised learning. There is no hard and fast rule with regard to using just one specific technique. You can always combine multiple methods as long as they are relevant in solving the problem. Unsupervised learning methods can be categorized under the following broad areas of ML tasks relevant to unsupervised learning. We explore these tasks briefly in the following sections to get a good feel of how unsupervised learning methods are used in the real world.

## **Clustering**

Clustering methods are Machine Learning methods that try to find patterns of similarity and relationships among data samples in our dataset and then cluster these samples into various groups, such that each group or cluster of data samples has some similarity, based on the inherent attributes or features. These methods are completely unsupervised because they try to cluster data by looking at the data features without any prior training, supervision, or knowledge about data attributes, associations, and relationships.

Consider a real-world problem of running multiple servers in a data center and trying to analyze logs for typical issues or errors. Our main task is to determine the various kinds of log messages that usually occur frequently each week. In simple words, we want to group log messages into various clusters based on some inherent characteristics. A simple approach would be to extract features from the log messages, which would be in textual format and apply clustering on the same and group similar log messages together based on similarity in content. Basically we have raw log messages to start with. Our clustering system would employ feature extraction to extract features from text like word occurrences, phrase

occurrences, and so on. Finally, a clustering algorithm like K-means or hierarchical clustering would be employed to group or cluster messages based on similarity of their inherent features.

It is quite clear that our systems have three distinct clusters of log messages where the first cluster depicts disk issues, the second cluster is about memory issues, and the third cluster is about processor issues. Top feature words that helped in distinguishing the clusters and grouping similar data samples (logs) together are also depicted in the figure. Of course, sometimes some features might be present across multiple data samples hence there can be slight overlap of clusters too since this is unsupervised learning. However, the main objective is always to create clusters such that elements of each cluster are near each other and far apart from elements of other clusters.

There are various types of clustering methods that can be classified under the following major approaches.

- Centroid based methods such as K-means and K-medoids
- Hierarchical clustering methods such as agglomerative and divisive (Ward's, affinity propagation)
- Distribution based clustering methods such as Gaussian mixture models
- Density based methods such as dbscan and optics.

Besides this, we have several methods that recently came into the clustering landscape, like birch and clarans.

## **Dimensionality reduction**

Once we start extracting attributes or features from raw data samples, sometimes our feature space gets bloated up with a humongous number of features. This poses multiple challenges including analyzing and visualizing data with thousands or millions of features, which makes the feature space extremely complex posing problems with regard to training models, memory, and space constraints. In fact this is referred to as the “curse of dimensionality”. Unsupervised methods can also be used in these scenarios, where

we reduce the number of features or attributes for each data sample. These methods reduce the number of feature variables by extracting or selecting a set of principal or representative features. There are multiple popular algorithms available for dimensionality reduction like Principal Component Analysis (PCA), nearest neighbors, and discriminant analysis.

It is quite clear that each data sample originally had three features or dimensions, namely  $D(x_1, x_2, x_3)$  and after applying PCA, we reduce each data sample from our dataset into two dimensions, namely  $D'(z_1, z_2)$ . Dimensionality reduction techniques can be classified in two major approaches as follows.

- **Feature Selection methods** : Specific features are selected for each data sample from the original list of features and other features are discarded. No new features are generated in this process.
- **Feature Extraction methods** : We engineer or extract new features from the original list of features in the data. Thus the reduced subset of features will contain newly generated features that were not part of the original feature set. PCA falls under this category.

## **Anomaly detection**

The process of anomaly detection is also termed as outlier detection, where we are interested in finding out occurrences of rare events or observations that typically do not occur normally based on historical data samples. Sometimes anomalies occur infrequently and are thus rare events, and in other instances, anomalies might not be rare but might occur in very short bursts over time, thus have specific patterns. Unsupervised learning methods can be used for anomaly detection such that we train the algorithm on the training dataset having normal, non-anomalous data samples. Once it learns the necessary data representations, patterns, and relations among attributes in normal samples, for any new data sample, it would be able to identify it as anomalous or a normal data point by using its learned knowledge

Anomaly detection based methods are extremely popular in real-world scenarios like detection of security attacks or breaches, credit card fraud, manufacturing anomalies, network issues, and many more.

## **Association rule-mining**

Typically association rule-mining is a data mining method used to examine and analyze large transactional datasets to find patterns and rules of interest. These patterns represent interesting relationships and associations, among various items across transactions. Association rule-mining is also often termed as market basket analysis, which is used to analyze customer shopping patterns. Association rules help in detecting and predicting transactional patterns based on the knowledge it gains from training transactions. Using this technique, we can answer questions like what items do people tend to buy together, thereby indicating frequent item sets. We can also associate or correlate products and items, i.e., insights like people who buy beer also tend to buy chicken wings at a pub.

Based on different customer transactions over a period of time, we have obtained the items that are closely associated and customers tend to buy them together. Some of these frequent item sets are depicted like {meat, eggs}, {milk, eggs} and so on. The criterion of determining good quality association rules or frequent item sets is usually done using metrics like support, confidence, and lift.

This is an unsupervised method, because we have no idea what the frequent item sets are or which items are more strongly associated with which items beforehand. Only after applying algorithms like the apriori algorithm or FP-growth, can we detect and predict products or items associated closely with each other and find conditional probabilistic dependencies.

## **Reinforcement Learning**

The reinforcement learning methods are a bit different from conventional supervised or unsupervised methods. In this context,

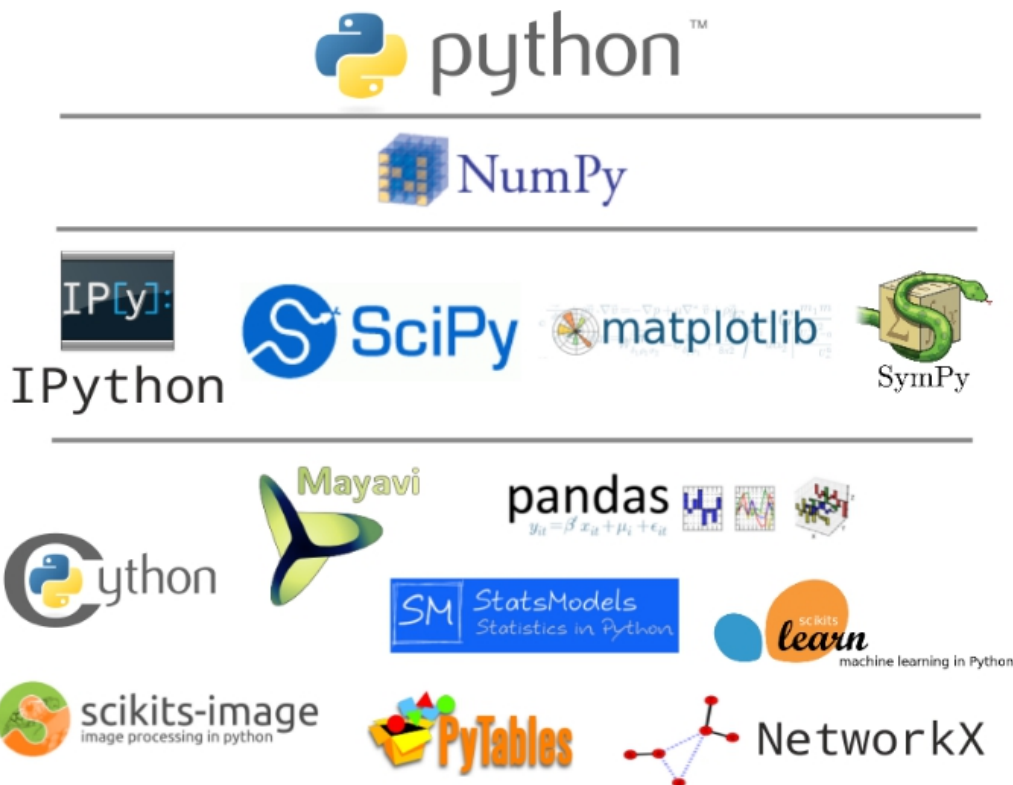
we have an agent that we want to train over a period of time to interact with a specific environment and improve its performance over a period of time with regard to the type of actions it performs on the environment. Typically the agent starts with a set of strategies or policies for interacting with the environment. On observing the environment, it takes a particular action based on a rule or policy and by observing the current state of the environment. Based on the action, the agent gets a reward, which could be beneficial or detrimental in the form of a penalty. It updates its current policies and strategies if needed and this iterative process continues till it learns enough about its environment to get the desired rewards. The main steps of a reinforcement learning method are mentioned as follows:

- Prepare agent with set of initial policies and strategy
- Observe environment and current state
- Select optimal policy and perform action
- Get corresponding reward (or penalty)
- Update policies if needed
- Repeat Steps 2 - 5 iteratively until agent learns the most optimal policies

Consider a real-world problem of trying to make a robot or a machine learn to play chess. In this case the agent would be the robot and the environment and states would be the chessboard and the positions of the chess pieces.

# Python Ecosystem for Machine Learning

The Python ecosystem is growing and may become the dominant platform for machine learning. The primary rationale for adopting Python for machine learning is because it is a general purpose programming language that you can use both for R&D and in production. In this chapter you will discover the Python ecosystem for machine learning, SciPy and the functionality it provides with NumPy, Matplotlib and Pandas, scikit-learn that provides all of the machine learning algorithms, how to setup your Python ecosystem for machine learning and what versions to use.



Python is a general purpose interpreted programming language. It is easy to learn and use primarily because the language focuses on readability. The philosophy of Python is captured in the Zen of Python which includes phrases like:

— Beautiful is better than ugly.

- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.

It is a popular language in general, consistently appearing in the top 10 programming languages in surveys on StackOverflow . It's a dynamic language and very suited to interactive development and quick prototyping with the power to support the development of large applications. It is also widely used for machine learning and data science because of the excellent library support and because it is a general purpose programming language (unlike R or Matlab).

This is a simple and very important consideration. It means that you can perform your research and development (figuring out what models to use) in the same programming language that you use for your production systems. Greatly simplifying the transition from development to production.

## SciPy

SciPy is an ecosystem of Python libraries for mathematics, science and engineering. It is an add-on to Python that you will need for machine learning. The SciPy ecosystem is comprised of the following core modules relevant to machine learning:

- **NumPy** : A foundation for SciPy that allows you to efficiently work with data in arrays.
- **Matplotlib** : Allows you to create 2D charts and plots from data.
- **Pandas** : Tools and data structures to organize and analyze your data.

To be effective at machine learning in Python you must install and become familiar with SciPy. Specifically:

- You will prepare your data as NumPy arrays for modeling in machine learning algorithms.

- You will use Matplotlib (and wrappers of Matplotlib in other frameworks) to create plots and charts of your data.
- You will use Pandas to load explore and better understand your data.

## **scikit-learn**

The scikit-learn library is how you can develop and practice machine learning in Python. It is built upon and requires the SciPy ecosystem. The name scikit suggests that it is a SciPy plug-in or toolkit. The focus of the library is machine learning algorithms for classification, regression, clustering and more. It also provides tools for related tasks such as evaluating models, tuning parameters and pre-processing data.

Like Python and SciPy, scikit-learn is open source and is usable commercially under the BSD license. This means that you can learn about machine learning, develop models and put them into operations all with the same ecosystem and code. A powerful reason to use scikit-learn.

## **Python Ecosystem Installation**

There are multiple ways to install the Python ecosystem for machine learning. In this section we cover how to install the Python ecosystem for machine learning.

Today, the debate rages on over which version of Python to use. Version 2.7, released in 2010, is perhaps the most widely used of all Python versions. Version 2.7, however, is not the most recent. In 2008, Python 3.0 — often stylized as Python “3.x” to represent all incremental updates to 3.0 — was released. As of this writing, the most recent version is Python 3.7.2 and there are some fundamental differences between Python 2.x and Python 3.x.

You may not need to install Python at all. Some computer operating systems have Python version 3.7 installed “out of the box” (meaning, pre-installed). You can check whether Python by opening the command line terminal on your computer. Once the terminal is open,



enter the following command and press “Enter” (or “Return”) on your keyboard:

Python

If the Python interpreter responds, a version number for Python will also appear. If the version is 3.7 (or 2.7.x), then you’re in luck! Otherwise, you’ll have to install version 3.7.

If you have a version other than 3.7, you should uninstall it and install 3.7.

## Installing Python

To install Python, follow these steps:

- Navigate to the Python downloads page: [Python downloads](#) .
- Click on the link/button to download Python 3.7.x.
- Follow the installation instructions (leave all defaults as-is).
- Open your terminal again and type the command `cd`. Next, type the command `python`. The Python interpreter should respond with the version number. If you’re on a Windows machine, you will likely have to navigate to the folder where Python is installed (for example, `Python3.7`, which is the default) for the `python` command to function.

Congrats! You should have Python 3.7 installed now. Let’s run some Python code!

## Installing SciPy

There are many ways to install SciPy. For example two popular ways are to use package management on your platform (e.g. `yum` on RedHat or `macports` on OS X) or use a Python package management tool like `pip`. The SciPy documentation is excellent and covers howto instructions for many different platforms on the page [Installing the SciPy Stack](#) . When installing SciPy, ensure that you install the following packages as a minimum:

- numpy
- pandas
- scipy
- matplotlib

## **Install with Anaconda**

Anaconda is a python edition which is used in scientific area, so if you install anaconda, all above packages will be installed automatically. Anaconda is a scientific Python distribution which contains a lot of not often used scientific python libraries. If you want to do data analyze, scientific computing, you can install anaconda and use it to implement what you want.

The anaconda installation is very simple and straight forward. The installation process on all the three platform is very similar. But before that you should [download anaconda for your OS platform](#) first.

After that follow below steps to install it on different OS platform.

### **— Windows**

- Double click the installation file and follow the wizard steps to install.
- When the installation complete, the installer will add anaconda bin directory in the PATH system environment variable.
- Open a dos window and run python command, if you see something like below, it means anaconda has been installed successfully in your Windows.

```
C:> python
```

```
Python 3.7.1 (default, Dec 14 2018, 19:28:38)
```

```
[GCC 7.3.0] :: Anaconda, Inc. on windows
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

If you have installed other python version before, and the output do not contain Anaconda, you need to check PATHsystem environment

variable, to check whether anaconda installation path is included at the beginning of the PATH variable value or not.

## — MacOS

- Install anaconda on MacOS is very similar with Windows.
- The different is that the downloaded file is a .pkg file.
- Double click the .pkg file to install it.
- After installation, the bin folder ( anaconda executable file saved folder ) in the anaconda installation directory will be added to /Users/\$USER/.bash\_profile automatically.
- To make sure the PATH value change take effect, run source .bash\_profile command in a terminal.
- To verify the anaconda installation is success, open a terminal and run python or ipython command, then you should see anaconda in the output text.
- If it shows error message, you should run env command in terminal to check the PATH env variable value, to see whether it's value contains the anaconda bin folder or not.

After installation, you can run command conda in a terminal to list above packages to make sure it has been installed correctly.

```
~$ conda list pandas
```

```
# packages in environment at /home/zhaosong/anaconda3:
```

```
#
```

```
# Name Version Build Channel
```

```
pandas 0.23.4 py37h04863e7_0
```

To list all installed anconda packages, just run \$ conda list.

Run \$ conda -h to list conda command help information.

If you want to remove / uninstall a package, run \$ conda remove <package name>

## Install by PIP Command

First make sure pip has been installed on your OS. PIP is a python script that can manage python packages. It can process search, install, update and uninstall operation to python packages. To find all useful python packages, you can go to <https://pypi.org/> . In this example pip is saved in C:\Python37\Scripts directory.

Open a dos window and run pip --version command then you can see below output in console that means pip has been installed correctly.

```
C:\Users>pip --version
```

```
pip 10.0.1 from c:\python37\lib\site-packages\pip (python 3.7)
```

— Run **pip install** command to install related packages.

```
pip install numpy
```

```
pip install pandas
```

```
pip install scipy
```

```
pip install matplotlib
```

Run **pip uninstall** command to uninstall related packages.

```
pip uninstall numpy
```

```
pip uninstall pandas
```

```
pip uninstall scipy
```

```
pip uninstall matplotlib
```

- Run **pip show** command to display package install information.

```
~$ pip show pandas
```

Name: pandas

Version: 0.23.4

Summary: Powerful data structures for data analysis, time series, and statistics

Home-page: <http://pandas.pydata.org>

Author: None

Author-email: None

License: BSD

Location: /home/zhaosong/anaconda3/lib/python3.7/site-packages

Requires: python-dateutil, pytz, numpy

Required-by: seaborn, odo

Once SciPy is installed, you can confirm that the installation was successful. If the installation is completed successfully, you will receive the following output

Collecting scipy

Downloading scipy-0.18.1-cp27-cp27mu-manylinux1\_x86\_64.whl  
(40.3MB)

100% |#####| 40.3MB 20kB/s

Installing collected packages: scipy

Successfully installed scipy-0.18.1

The examples in this book assume you have these version of the SciPy libraries or newer. If you have an error, you may need to consult the documentation for your platform.

## Installing scikit-learn

I would suggest that you use the same method to install scikit-learn as you used to install SciPy.

scikit-learn is a Python module for machine learning built on top of SciPy and distributed under the 3-Clause BSD license.

The project was started in 2007 by David Cournapeau as a Google Summer of Code project, and since then many volunteers have contributed.

It is currently maintained by a team of volunteers.

Website: <http://scikit-learn.org>

There are instructions for installing scikit-learn, but they are limited to using the Python pip and conda package managers.

scikit-learn requires:

- Python ( $\geq 3.5$ )
- NumPy ( $\geq 1.11.0$ )
- SciPy ( $\geq 0.17.0$ )
- joblib ( $\geq 0.11$ )

**Scikit-learn 0.20 was the last version to support Python2.7.**

Scikit-learn 0.21 and later require Python 3.5 or newer.

For running the examples Matplotlib  $\geq 1.5.1$  is required. A few examples require scikit-image  $\geq 0.12.3$ , a few examples require pandas  $\geq 0.18.0$ .

Like SciPy, you can confirm that scikit-learn was installed successfully. Start your Python interactive environment and type and run the following code.

```
# scikit-learn import sklearn
```

```
print(' sklearn: {}'.format(sklearn.__version__))
```

It will print the version of the scikit-learn library installed. On my workstation at the time of writing I see the following output:

```
sklearn: 0.18
```

The examples in this book assume you have this version of scikit-learn or newer.

If you are not confident at installing software on your machine, Anaconda is an easier option for you, which you can download and install for free. Anaconda is a packaged compilation of Python along with a whole suite of a variety of libraries, including core libraries which are widely used in Data Science. Developed by Anaconda, formerly known as Continuum Analytics, it is often the go-to setup for data scientists. Travis Oliphant, primary contributor to both the numpy and scipy libraries, is Anaconda's president and one of the co-founders.

The Anaconda distribution is BSD licensed and hence it allows us to use it for commercial and redistribution purposes. A major advantage of this distribution is that we don't require an elaborate setup and it works well on all flavors of operating systems and platforms, especially Windows, which can often cause problems with installing specific Python packages. Thus, we can get started with our Data Science journey with just one download and install.

The Anaconda distribution is widely used across industry Data Science environments and it also comes with a wonderful IDE, Spyder (Scientific Python Development Environment), besides other useful utilities like jupyter notebooks, the IPython console, and the excellent package management tool, conda. Recently they have also talked extensively about Jupyterlab, the next generation UI for Project Jupyter.

We recommend using the Anaconda distribution and also checking out <https://www.anaconda.com/what-is-anaconda/> to learn more about Anaconda. It supports the three main platforms of Microsoft Windows, Mac OS X and Linux. It includes Python, SciPy and scikit-

learn. Everything you need to learn, practice and use machine learning with the Python Environment.



## Getting Familiar with Python and SciPy

You do not need to be a Python developer to get started using the Python ecosystem for machine learning. As a developer who already knows how to program in one or more programming languages, you are able to pick up a new language like Python very quickly. You just need to know a few properties of the language to transfer what you already know to the new language.



The aim of this chapter is to show you the following:

- How to navigate Python language syntax.
- Enough NumPy, Matplotlib and Pandas to read and write machine learning Python scripts.
- A foundation from which to build a deeper understanding of machine learning tasks in Python.

If you already know a little Python, this chapter will be a friendly reminder for you. Let's get started.

## Python Crash Course

When getting started in Python you need to know a few key details about the language syntax to be able to read and understand Python code. This includes: assignment, flow control, data structures, functions.

We will cover each of these topics in turn with small standalone examples that you can type and run. Remember, whitespace has meaning in Python.

### Assignment

As a programmer, assignment and types should not be surprising to you.

#### — Strings

```
# Strings data = 'hello world'
print(data[0])
print(len(data))
print(data)
```

Notice how you can access characters in the string using array syntax. Running the example prints:

h

11

hello world

#### — Numbers

```
# Numbers
value = 123.1
```

```
print(value)
```

```
value = 10
```

```
print(value)
```

Running the example prints:

```
123.1
```

```
10
```

## ``` — ``` **Boolean**

```
# Boolean
```

```
a = True
```

```
b = False
```

```
print(a, b)
```

Running the example prints:

```
(True, False)
```

## ``` — ``` **Multiple Assignment**

```
# Multiple Assignment
```

```
a, b, c = 1, 2, 3
```

```
print(a, b, c)
```

This can also be very handy for unpacking data in simple data structures. Running the example prints:

```
(1, 2, 3)
```

## ``` — ``` **No Value**

```
# No value
```

```
a = None
```

```
print(a)
```

Running the example prints:

None

## Flow control

There are three main types of flow control that you need to learn: If-Then-Else conditions, For-Loops and While-Loops.

### – If-Then-Else Conditional

```
value = 99 if value == 99: print 'That is fast'
```

```
elif value > 200:
```

```
print 'That is too fast'
```

```
else:
```

```
print 'That is safe'
```

Notice the colon (:) at the end of the condition and the meaningful tab intend for the code block under the condition. Running the example prints:

If-Then-Else conditional

### – For-Loop

```
# For-Loop
```

```
for i in range(10):
```

```
    print i
```

Running the example prints:

0

1

2

3

4

5

6

7

8

9

## **— While-Loop**

# While-Loop

i = 0 while i < 10:

print i i += 1

Running the example prints:

0

1

2

3

4

5

6

7

8

## Data structures

There are three data structures in Python that you will find the most used and useful. They are tuples, lists and dictionaries.

### – Tuple

Tuples are read-only collections of items.

```
a = (1, 2, 3)
```

```
print a
```

Running the example prints:

```
(1, 2, 3)
```

### – List

Lists use the square bracket notation and can be index using array notation.

```
mylist = [1, 2, 3]
```

```
print("Zeroth Value: %d" %
```

```
mylist[0]
```

```
mylist.append(4)
```

```
print("List Length: %d" % len(mylist))
```

```
for value in mylist:
```

```
    print value
```

Notice that we are using some simple printf-like functionality to combine strings and variables when printing.

Running the example prints:

Zeroth Value: 1

List Length: 4

1

2

3

4

## – Dictionary

Dictionaries are mappings of names to values, like key-value pairs. Note the use of the curly bracket and colon notations when defining the dictionary.

```
mydict = {'a': 1, 'b': 2, 'c': 3}
print("A value: %d" %
mydict['a'] mydict['a'] = 11
print("A value: %d" % mydict['a']
print("Keys: %s" % mydict.keys()
print("Values: %s" % mydict.values()
for key in mydict.keys():
    print mydict[key]
```

Running the example prints:

NumPy Crash Course

A value: 1

A value: 11

Keys: ['a', 'c', 'b']

Values: [11, 3, 2]

11

3

2

## **\_ Functions**

The biggest gotcha with Python is the whitespace. Ensure that you have an empty new line after indented code. The example below defines a new function to calculate the sum of two values and calls the function with two arguments.

```
# Sum function
```

```
def mysum(x, y):
```

```
    return x + y
```

```
# Test sum function
```

```
result = mysum
```

Running the example prints:

4

## **NumPy Crash Course**

NumPy provides the foundation data structures and operations for SciPy. These are arrays (ndarrays) that are efficient to define and manipulate.

### **Create array**

```
# define an array
```

```
import numpy
```

```
mylist = [1, 2, 3]
```



```
myarray = numpy.array(mylist)
```

```
print(myarray)
```

```
print(myarray.shape)
```

Notice how we easily converted a Python list to a NumPy array.

Running the example prints:

```
[1 2 3] (3,)
```

## **Access data**

Array notation and ranges can be used to efficiently access data in a NumPy array.

```
# access values
```

```
import numpy
```

```
mylist = [[1, 2, 3], [3, 4, 5]]
```

```
myarray = numpy.array(mylist)
```

```
print(myarray)
```

```
print(myarray.shape)
```

```
print("First row: %s" % myarray[0])
```

```
print("Last row: %s" % myarray[-1])
```

```
print("Specific row and col: %s" % myarray[0, 2])
```

```
print("Whole col: %s" % myarray[:, 2])
```

Running the example prints:

```
[[1 2 3]
```

```
[3 4 5]]
```

```
(2, 3)
```

First row: [1 2 3]

Last row: [3 4 5]

Specific row and col: 3

Whole col: [3 5]

## Arithmetic

NumPy arrays can be used directly in arithmetic.

# arithmetic

```
import numpy
```

```
myarray1 = numpy.array([2, 2, 2])
```

```
myarray2 = numpy.array([3, 3, 3])
```

```
print("Addition: %s") % (myarray1 + myarray2)
```

```
print("Multiplication: %s") % (myarray1 * myarray2)
```

Running the example prints:

Addition: [5 5 5]

Multiplication: [6 6 6]

There is a lot more to NumPy arrays but these examples give you a flavor of the efficiencies they provide when working with lots of numerical data.

## Matplotlib Crash Course

Matplotlib can be used for creating plots and charts. The library is generally used as follows:

- Call a plotting function with some data (e.g. `.plot()`).
- Call many functions to setup the properties of the plot (e.g. labels and colors).

— Make the plot visible (e.g. `.show()`).

## Line Plot

The example below creates a simple line plot from one dimensional data.

```
# basic line plot
import matplotlib.pyplot as plt
import numpy
myarray = numpy.array([1, 2, 3])
plt.plot(myarray)
plt.xlabel('some x axis')
plt.ylabel('some y axis')
plt.show()
```

Running the example produces:

### 3.3. Matplotlib Crash Course

## Scatter plot

Below is a simple example of creating a scatter plot from two dimensional data.

```
# basic scatter plot
import matplotlib.pyplot as plt
import numpy
x = numpy.array([1, 2, 3])
y = numpy.array([2, 4, 6])
plt.scatter(x,y)
plt.xlabel('some x axis')
```

```
plt.ylabel('some y axis')
```

```
plt.show()
```

Running the example produces:

There are many more plot types and many more properties that can be set on a plot to configure it.

## Pandas Crash Course

Pandas provides data structures and functionality to quickly manipulate and analyze data. The key to understanding Pandas for machine learning is understanding the Series and DataFrame data structures.

### Series

A series is a one dimensional array where the rows and columns can be labeled.

```
# series
```

```
import numpy
```

```
import pandas
```

```
myarray = numpy.array([1, 2, 3])
```

```
rownames = ['a', 'b', 'c']
```

```
myseries = pandas.Series(myarray, index=rownames)
```

```
print(myseries)
```

Running the example prints:

Listing 3.34: Output of example of creating a Pandas Series.

You can access the data in a series like a NumPy array and like a dictionary, for example:

```
print(myseries[0]) print(myseries['a'])
```

Listing 3.35: Example of accessing data in a Pandas Series.

Running the example prints:

```
1
```

```
1
```

## DataFrame

A data frame is a multi-dimensional array where the rows and the columns can be labeled.

```
# dataframe
```

```
import numpy
```

```
import pandas
```

```
myarray = numpy.array([[1, 2, 3], [4, 5, 6]])
```

```
rownames = ['a', 'b']
```

```
colnames = ['one', 'two', 'three']
```

```
mydataframe = pandas.DataFrame(myarray, index=rownames,  
                                columns=colnames)
```

```
print(mydataframe)
```

Running the example prints:

	one	two	three
a	1	2	3
b	4	5	6

Data can be index using column names.

```
print("method 1:")
```

```
print("one column: %s" % mydataframe['one'])
```

```
print("method 2:")
```

```
print("one column: %s") % mydataframe.one
```

Pandas is a very powerful tool for slicing and dicing you data.

By now, you should have discovered basic syntax and usage of Python and three key Python libraries used for machine learning – NumPy, Matplotlib, and Pandas.

Next, you now know enough syntax and usage information to read and understand Python code for machine learning and to start creating your own scripts. In the next lesson you will discover how you can very quickly and easily load standard machine learning datasets in Python.

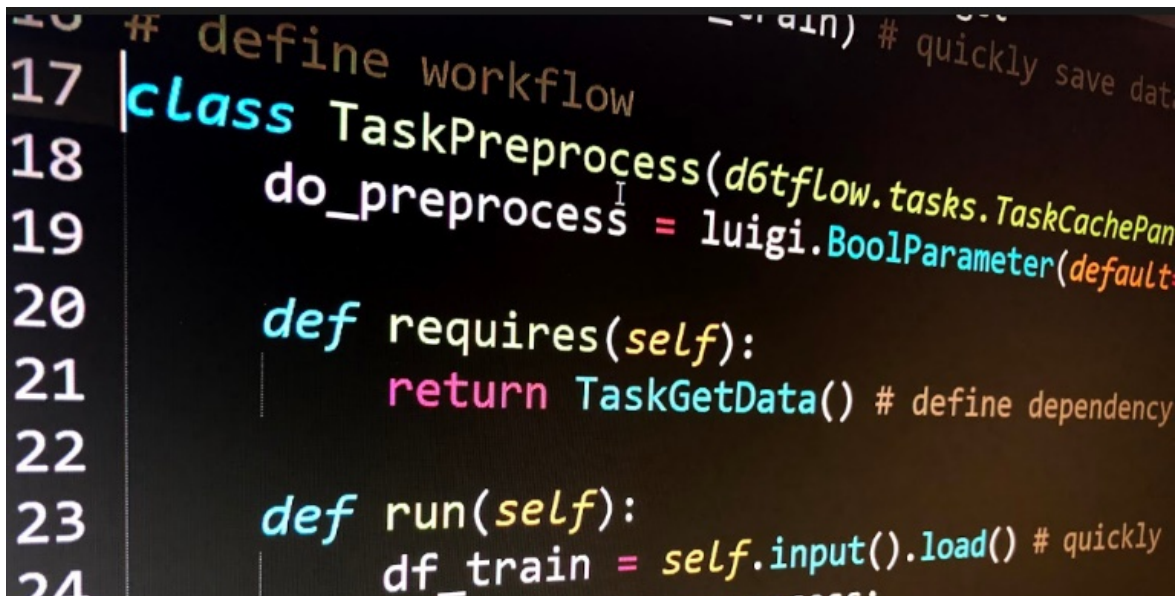
## Loading Machine Learning Data

You must be able to load your data before you can start your machine learning project. The most common format for machine learning data is CSV files. There are a number of ways to load a CSV file in Python. In this section you will learn three ways that you can use to load your CSV data in Python:

- Load CSV Files with the Python Standard Library.
- Load CSV Files with NumPy.
- Load CSV Files with Pandas.

But firstly, we need to understand the different data collection/retrieval mechanisms for different data types.

Let's proceed!.



```
17 # define workflow
18 class TaskPreprocess(d6tflow.tasks.TaskCachePan
19     do_preprocess = luigi.BoolParameter(default
20
21     def requires(self):
22         return TaskGetData() # define dependency
23
24     def run(self):
25         df_train = self.input().load() # quickly
```

## Data Collection

Data collection is where it all begins. Though listed as a step that comes post business understanding and problem definition, data collection often happens in parallel. This is done in order to assist in augmenting the business understanding process with facts like availability, potential value, and so on before a complete use case can be formed and worked upon. Of course, data collection takes a

formal and better form once the problem statement is defined and the project gets underway.

Data is at the center of everything around us, which is a tremendous opportunity. Yet this also presents the fact that it must be present in different formats, shapes, and sizes. Its omnipresence also means that it exists in systems such as legacy machines (say mainframes), web (say web sites and web applications), databases, flat files, sensors, mobile devices, and so on.

Let's look at some of the most commonly occurring data formats and ways of collecting such data.

## **CSV**

A CSV data file is one of the most widely available formats of data. It is also one of the oldest formats still used and preferred by different systems across domains. Comma Separated Values (CSV) are data files that contain data with each of its attributes delimited by a “,” (a comma). Figure 3-1 depicts a quick snapshot of how a typical CSV file looks.

The sample CSV shows how data is typically arranged. It contains attributes of different data types separated/delimited by a comma. A CSV may contain an optional header row (as shown in the example).

CSVs may also optionally enclose each of the attributes in single or double quotes to better demarcate. Though usually CSVs are used to store tabular data, i.e., data in the form of rows and columns, this is not the only way.

CSVs come in different variations and just changing the delimiter to a tab makes one a TSV (or a tab separated values) file. The basic ideology here is to use a unique symbol to delimit/separate different attributes.

Now that we know how a CSV looks, let's employ some Python magic to read/extract this data for use.



One of the advantages of using a language like Python is its ability to abstract and handle a whole lot of stuff. Unlike other languages where specific libraries or a lot of code is required to get basic stuff done, Python handles it with élan. Along the same lines is reading a CSV file. The simplest way to read a CSV is through the Python csv module. This module provides an abstraction function called the reader().

The reader function takes a file object as input to return an iterator containing the information read from the csv file. The following code snippet uses the csv.reader() function to read a given file.

```
csv_reader = csv.reader(open(file_name, 'rb'), delimiter=',')
```

Once the iterator is returned, we can easily iterate through the contents and get the data in the form/format required. For the sake of completeness let's go through an example where we read the contents of the CSV shown in Figure 3-1 using the csv module. We will then extract each of its attributes and convert the data into a dict with keys representing them. The following snippet forms the actions.

```
csv_rows = list()
```

```
csv_attr_dict = dict()
```

```
csv_reader = None
```

```
# read csv
```

```
csv_reader = csv.reader(open(file_name, 'rb'), delimiter=delimiter)
```

```
# iterate and extract data
```

```
for row in csv_reader:
```

```
    print(row)
```

```
    csv_rows.append(row)
```

```
# iterate and add data to attribute lists
```

```
for row in csv_rows[1:] :
```

```
csv_attr_dict['sno'].append(row[0])
csv_attr_dict['fruit'].append(row[1])
csv_attr_dict['color'].append(row[2])
csv_attr_dict['price'].append(row[3])
```

The output is a dict containing each attribute as a key with values and as an ordered list of values read from the CSV file.

CSV Attributes::

```
{'color': ['red', 'yellow', 'yellow', 'orange', 'green', 'yellow', 'green'],
'fruit': ['apple', 'banana', 'mango', 'orange', 'kiwi', 'pineapple', 'guava'],
'price': ['110.85', '50.12', '70.29', '80.00', '150.00', '90.00', '20.00'],
'sno': ['1', '2', '3', '4', '5', '6', '7']}
```

The extraction of data from a CSV and its transformation depends on the use case requirements. The conversion of our sample CSV into a dict of attributes is one way. We may choose different output format depending on the data and our requirements.

Though the workflow to handle and read a CSV file is pretty straightforward and easy to use, we would like to standardize and speed up our process. Also, more often than not, it is easier to understand data in a tabular format. We were introduced to the pandas library in the previous chapter with some amazing capabilities. Let's now utilize pandas to read a CSV as well.

The following snippet shows how pandas makes reading and extracting data from a CSV that's simpler and consistent as compared to the csv module.

```
df = pd.read_csv(file_name, sep=delimiter)
```

With a single line and a few optional parameters (as per requirements), pandas extracts data from a CSV file into a dataframe, which is a tabular representation of the same data. One of the major advantages of using pandas is the fact that it can handle a lot of

different variations in CSV files, such as files with or without headers, attribute values enclosed in quotes, inferring data types, and many more. Also, the fact that various machine learning libraries have the capability to directly work on pandas dataframes, makes it virtually a de facto standard package to handle CSV files.

The previous snippet generates the following output dataframe:

```
sno fruit color price
```

```
0 1 apple red 110.85
```

```
1 2 banana yellow 50.12
```

```
2 3 mango yellow 70.29
```

```
3 4 orange orange 80.00
```

```
4 5 kiwi green 150.00
```

```
5 6 pineapple yellow 90.00
```

```
6 7 guava green 20.00
```

Note pandas makes the process of reading CSV files a breeze, yet the csv module comes in handy when we need more flexibility. For example, not every use case requires data in tabular form or the data might not be consistently formatted and requires a flexible library like csv to enable custom logic to handle such data.

Along the same lines, data from flat files containing delimiters other than ',' (comma) like tabs or semicolons can be easily handled with these two modules. We will use these utilities while working on specific use cases in further chapters; until then, you are encouraged to explore and play around with these for a better understanding.

## JSON

Java Script Object Notation (JSON) is one of the most widely used data interchange formats across the digital realm. JSON is a lightweight alternative to legacy formats like XML (we shall discuss this format next). JSON is a text format that is language independent

with certain defined conventions. JSON is a human-readable format that is easy/simple to parse in most programming/scripting languages. A JSON file/object is simply a collection of name(key)-value pairs. Such key-value pair structures have corresponding data structures available in programming languages in the form of dictionaries (Python dict), struct, object, record, keyed lists, and so on. More details are available at <http://www.json.org/>.

JSONs are widely to send information across systems. The Python equivalent of a JSON object is the dict data type, which itself is a key-value pair structure. Python has various JSON related libraries that provide abstractions and utility functions. The json library is one such option that allows us to handle JSON files/objects.

The JSON object depicts a fairly nested structure that contains values of string, numeric, and array type. JSON also supports objects, Booleans, and other data types as values as well. The following snippet reads the contents of the file and then utilizes json.loads() utility to parse and convert it into a standard Python dict.

```
json_filedata = open(file_name).read() json_data =  
json.loads(json_filedata)
```

json\_data is a Python dict with keys and values of the JSON file parsed and type casted as Python data types. The json library also provides utilities to write back Python dictionaries as JSON files with capabilities of error checking and typecasting. The output of the previous operation is as follows.

```
outer_col_1 :      nested_inner_col_1 :  
val_1      nested_inner_col_2 : 2  
nested_inner_col_1 : val_2  
nested_inner_col_2 : 2 outer_col_2 :  
inner_col_1 : 3 outer_col_3 : 4
```

Before we move on to our next format, it is worth noting that pandas also provides utilities to parse JSONs. The pandas read\_json() is a

very powerful utility that provides multiple options to handle JSONs created in different styles.

We can easily parse such a JSON using pandas by setting the orientation parameter to “records”, as shown here.

```
df = pd.read_json(file_name,orient="records")
```

The output is a tabular dataframe with each data point represented by two attribute values as follows.

	col_1	col_2	0	a	b
1		c		d	
2		e		f	
3		g		h	
4		i		j	
5		k		l	

You are encouraged to read more about pandas read\_json() at [https://pandas.pydata.org/pandasdocs/stable/generated/pandas.read\\_json.html](https://pandas.pydata.org/pandasdocs/stable/generated/pandas.read_json.html) .

## XML

Having covered two of the most widely used data formats, so now let's take a look at XML. XMLs are quite a dated format yet is used by a lot many systems. XML or eXtensible Markup Language is a markup language that defines rules for encoding data/documents to be shared across the Internet. Like JSON, XML is also a text format that is human readable. Its design goals involved strong support for various human languages (via Unicode), platform independence, and simplicity. XMLs are widely used for representing data of varied shapes and sizes.

XMLs are widely used as configuration formats by different systems, metadata, and data representation format for services like RSS, SOAP, and many more.

XML is a language with syntactic rules and schemas defined and refined over the years. The most important components of an XML are

as follows:

- **Tag** : A markup construct denoted by strings enclosed with angled braces (“<” and “>”).
- **Content** : Any data not marked within the tag syntax is the content of the XML file/object.
- **Element** : A logical construct of an XML. An element may be defined with a start and an end tag with or without attributes, or it may be simply an empty tag.
- **Attribute** : Key-value pairs that represent the properties or attributes of the element in consideration. These are enclosed within a start or an empty tag.

More details on key concepts and details can be browsed at t <https://www.w3schools.com/xml/> .

XMLs can be viewed as tree structures, starting with one root element that branches off into various elements, each with their own attributes and further branches, the content being at leaf nodes.

Most XML parsers use this tree-like structure to read XML content. The following are the two major types of XML parsers:

- **DOM parser** : The Document Object Model parser is the closest form of tree representation of an XML. It parses the XML and generates the tree structure. One big disadvantage with DOM parsers is their instability with huge XML files.
- **SAX parser** : The Simple API for XML (or SAX for short) is a variant widely used on the web. This is an event-based parser that parses an XML element by element and provides hooks to trigger events based on tags. This overcomes the memory-based restrictions of DOM but lacks overall representation power.

There are multiple variants available that derive from these two types. To begin with, let's take a look at the ElementTree parser available from Python's xml library. The ElementTree parser is an

optimization over the DOM parser and it utilizes Python data structures like lists and dicts to handle data in a concise manner.

The following snippet uses the ElementTree parser to load and parse the sample XML file we saw previously. The parse() function returns a tree object, which has various attributes, iterators, and utilities to extract root and further components of the parsed XML.

```
tree = ET.parse(file_name) root = tree.getroot()

print("Root tag:{0}".format(root.tag)) print("Attributes of Root::
{0}".format(root.attrib))
```

The two print statements provide us with values related to the root tag and its attributes (if there are any). The root object also has an iterator attached to it which can be used to extract information related to all child nodes. The following snippet iterates the root object to print the contents of child nodes.

```
for child in xml:root:      print("{0}tag:{1}, attribute:{2}".format(
                                "\t"*indent_level,      child.tag,
                                child.attrib))

    print("{0}tag                                data:
{1}".format("\t"*indent_level,      child.text))
```

The final output generated by parsing the XML using ElementTree is as follows. We used a custom print utility to make the output more readable, the code for which is available on the repository.

Root tag:records

Attributes of Root:: {'attr': 'sample xml records'}

tag:record, attribute: {'name': 'rec\_1'}

tag data:

```
    tag:sub_element, attribute:{}      tag data:
    tag:detail1, attribute:{}          tag data:Attribute
1    tag:detail2, attribute:{ }
```



```

tag data:2
tag:sub_element_with_attr, attribute: {'attr': 'complex'} tag
data: Sub_Element_Text
tag:sub_element_only_attr, attribute: {'attr_val': 'only_attr'}
tag data:None
tag:record, attribute: {'name': 'rec_2'} tag data:
tag:sub_element, attribute: {} tag data:
tag:detail1, attribute: {} tag data:Attribute
1 tag:detail2, attribute: {}
tag data:2
tag:sub_element_with_attr, attribute: {'attr': 'complex'} tag
data: Sub_Element_Text
tag:sub_element_only_attr, attribute: {'attr_val':
'only_attr'} tag data:None

```

The xml library provides very useful utilities exposed through the ElementTree parser, yet it lacks a lot of fire power. Another Python library, xmltodict, provides similar capabilities but uses Python's native data structures like dicts to provide a more Pythonic way to handle XMLs. The following is a quick snippet to parse the same XML. Unlike ElementTree, the parse() function of xmltodict reads a file object and converts the contents into nested dictionaries.

```

xml_filedata = open(file_name).read()
ordered_dict = xmltodict.parse(xml_filedata)

```

The output generated is similar to the one generated using ElementTree with the exception that xmltodict uses the @ symbol to mark elements and attributes automatically. The following is the sample output.

```

records : @attr : sample xml records record : @name
: rec_1

```

sub\_element :

detail1 : Attribute 1

detail2 : 2 sub\_element\_with\_attr :

@attr : complex

#text : Sub\_Element\_Text sub\_element\_only\_attr :

@attr\_val : only\_attr

## **HTML and Scraping**

We began the chapter talking about the immense amount of information/data being generated at breakneck speeds. The Internet or the web is one of the driving forces for this revolution coupled with immense reach due to computers, smartphones and tablets.

The Internet is a huge interconnected web of information connected through hyperlinks. A large amount of data on the Internet is in the form of web pages. These web pages are generated, updated, and consumed millions of times day in and day out. With information residing in these web pages, it is imperative that we must learn how to interact and extract this information/data as well.

So far we have dealt with formats like CSV, JSON, and XML, which can be made available/extracted through various methods like manual downloads, APIs, and so on. With web pages, the methods change. In this section we will discuss the HTML format (the most common form of web page related format) and web-scraping techniques.

### **HTML**

The Hyper Text Markup Language (HTML) is a markup language similar to XML. HTML is mainly used by web browsers and similar applications to render web pages for consumption.

HTML defines rules and structure to describe web pages using markup. The following are standard components of an HTML page:

- Element: Logical constructs that form the basic building blocks of an HTML page
- Tags: A markup construct defined by angled braces (< and >). Some of the important tags are:
  - <html></html>: This pair of tags contains the whole of HTML document. It marks the start and end of the HTML page.
  - <body></body>: This pair of tags contains the main content of the HTML page rendered by the browser.

There are many more standard set of tags defined in the HTML standard; further information is available at [https://www.w3schools.com/html/html\\_intro.asp](https://www.w3schools.com/html/html_intro.asp) .

The following is a snippet to generate an HTML page that's rendered by a web browser.

```
<!DOCTYPE html>

<html>

<head>

<title>Sample HTML Page</title>

</head>

<body>

<h1>Sample WebPage</h1>

<p>HTML has been rendered</p>

</body>

</html>
```

Browsers use markup tags to understand special instructions like text formatting, positioning, hyperlinks, and so on but only renders the content for the end user to see. For use cases where data/information resides in HTML pages, we need special techniques to extract this content.

## Web Scraping

Web scraping is a technique to scrape or extract data from the web, particularly from web pages. Web scraping may involve manually copying the data or using automation to crawl, parse, and extract information from web pages. In most contexts, web scraping refers to automatically crawling a particular web site or a portion of the web to extract and parse information that can be later on used for analytics or other use cases. A typical web scraping flow can be summarized as follows:

**Crawl:** A bot or a web crawler is designed to query a web server using the required set of URLs to fetch the web pages. A crawler may employ sophisticated techniques to fetch information from pages linked from the URLs in question and even parse information to a certain extent. Web sites maintain a file called robots.txt to employ what is called as the “Robots Exclusion Protocol” to restrict/provide access to their content. More details are available at <http://www.robotstxt.org/robotstxt.html>.

**Scrape:** Once the raw web page has been fetched, the next task is to extract information from it. The task of scraping involves utilizing techniques like regular expressions, extraction based on XPath, or specific tags and so on to narrow down to the required information on the page.

Web scraping involves creativity from the point of view of narrowing down to the exact piece of information required. With web sites changing constantly and web pages becoming dynamic (see asp, jsp, etc.), presence of access controls (username/password, CAPTCHA, and so on) complicate the task even more. Python is a very powerful programming language, which should be evident by now, and scraping the web is another task for which it provides multiple utilities. Let's begin with extracting a blog post's text from the Apress blog to better understand web scraping.

## SQL

Databases date back to the 1970s and represent a large volume of data stored in relational form. Data available in the form of tables in databases, or to be more specific, relational databases, comprise of another format of structured data that we encounter when working on different use cases. Over the years, there have been various flavors of databases available, most of them conforming to the SQL standard.

The Python ecosystem handles data from databases in two major ways. The first and the most common way used while working on data science and related use cases is to access data using SQL queries directly. To access data using SQL queries, powerful libraries like sqlalchemy and pyodbc provide convenient interfaces to connect, extract, and manipulate data from a variety of relational databases like MS SQL Server, MySQL, Oracle, and so on. The sqlite3 library provides a lightweight easy-to-use interface to work with SQLite databases, though the same can be handled by the other two libraries as well.

The second way of interacting with databases is the ORM or the Object Relational Mapper method. This method is synonymous to the object oriented model of data, i.e., relational data is mapped in terms of objects and classes. SQLAlchemy provides a high-level interface to interact with databases in the ORM fashion.

## **Considerations When Loading CSV Data**

There are a number of considerations when loading your machine learning data from CSV files. For reference, you can learn a lot about the expectations for CSV files by reviewing the [CSV request for comment](#) titled Common Format and MIME Type for Comma-Separated Values (CSV) Files .

### **File header**

Does your data have a file header? If so this can help in automatically assigning names to each column of data. If not, you may need to name your attributes manually. Either way, you should

explicitly specify whether or not your CSV file had a file header when loading your data.

## **Comments**

Does your data have comments? Comments in a CSV file are indicated by a hash (#) at the start of a line. If you have comments in your file, depending on the method used to load your data, you may need to indicate whether or not to expect comments and the character to expect to signify a comment line.

## **Delimiter**

The standard delimiter that separates values in fields is the comma (,) character. Your file could use a different delimiter like tab or white space in which case you must specify it explicitly.

## **Quotes**

Sometimes field values can have spaces. In these CSV files the values are often quoted. The default quote character is the double quote character. Other characters can be used, and you must specify the quote character used in your file.

## **Loading CSV Files with the Python Standard Library**

The Python API provides the module CSV and the function reader() that can be used to load CSV files. Once loaded, you can convert the CSV data to a NumPy array and use it for machine learning. For example, you can download the Pima Indians dataset into your local directory with the filename pima-indians-diabetes.data.csv. All fields in this dataset are numeric and there is no header line.

```
# Load CSV Using Python Standard Library
```

```
import csv
```

```
import numpy
```

```

filename = 'pima-indians-diabetes.data.csv'
raw_data = open(filename, 'rb')
reader = csv.reader(raw_data, delimiter=',', quoting=csv.QUOTE_NONE)
x = list(reader)
data = numpy.array(x).astype('float')
print(data.shape)

```

The example loads an object that can iterate over each row of the data and can easily be converted into a NumPy array. Running the example prints the shape of the array.

```
(768, 9)
```

## Loading CSV Files with NumPy

You can load your CSV data using NumPy and the `numpy.loadtxt()` function. This function assumes no header row and all data has the same format. The example below assumes that the file `pima-indians-diabetes.data.csv` is in your current working directory.

```

# Load CSV using NumPy
from numpy import loadtxt
filename = 'pima-indians-diabetes.data.csv'
raw_data = open(filename, 'rb')
data = loadtxt(raw_data, delimiter=",")
print(data.shape)

```

Running the example will load the file as a `numpy.ndarray` and print the shape of the data:

```
(768, 9)
```

This example can be modified to load the same dataset directly from a URL as follows:

```
# Load CSV from URL using NumPy
from numpy import loadtxt
from urllib import urlopen
url = 'https://goo.gl/vhm1eU'
raw_data = urlopen(url)
dataset = loadtxt(raw_data, delimiter=",")
print(dataset.shape)
```

Again, running the example produces the same resulting shape of the data.

(768, 9)

For more information on the `numpy.loadtxt()` function see the API documentation.

## Loading CSV Files with Pandas

You can load your CSV data using Pandas and the `pandas.read_csv()` function. This function is very flexible and is perhaps my recommended approach for loading your machine learning data. The function returns a `pandas.DataFrame` that you can immediately start summarizing and plotting. The example below assumes that the `pima-indians-diabetes.data.csv` file is in the current working directory.

```
# Load CSV using Pandas
from pandas import read_csv
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = read_csv(filename, names=names)
```



```
print(data.shape)
```

Note that in this example we explicitly specify the names of each attribute to the DataFrame. Running the example displays the shape of the data:

```
(768, 9)
```

We can also modify this example to load CSV data directly from a URL.

```
# Load CSV using Pandas from URL
```

```
from pandas import read_csv
```

```
url = 'https://goo.gl/vhm1eU'
```

```
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
```

```
data = read_csv(url, names=names)
```

```
print(data.shape)
```

Again, running the example downloads the CSV file, parses it and displays the shape of the loaded DataFrame.

```
(768, 9)
```

To learn more about the `pandas.read_csv()` function you can refer to the API documentation.

Now that you know how to load your CSV data using Python it is time to start looking at it. In the next lesson you will discover how to use simple descriptive statistics to better understand your data.

## Understand Your Data With Descriptive Statistics

You must understand your data in order to get the best results. In this chapter you will discover the different recipes that you can use in Python to better understand your machine learning data. After reading this lesson you will know how to:

Each recipe is demonstrated by loading the Pima Indians Diabetes classification dataset from the UCI Machine Learning repository. Open your Python interactive environment and try each recipe out in turn. Let's get started.



### Peek at Your Data

There is no substitute for looking at the raw data. Looking at the raw data can reveal insights that you cannot get any other way. It can also plant seeds that may later grow into ideas on how to better pre-process and handle the data for machine learning tasks. You can review the first 20 rows of your data using the `head()` function on the Pandas DataFrame.

```
# View first 20 rows
```

```
from pandas import read_csv
filename = "pima-indians-diabetes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = read_csv(filename, names=names)
peek = data.head(20)
print(peek)
```

## Dimensions of Your Data

You must have a very good handle on how much data you have, both in terms of rows and columns.

Too many rows and algorithms may take too long to train. Too few and perhaps you do not have enough data to train the algorithms.

Too many features and some algorithms can be distracted or suffer poor performance due to the curse of dimensionality.

You can review the shape and size of your dataset by printing the shape property on the Pandas DataFrame.

# Dimensions of your data

```
from pandas import read_csv
filename = "pima-indians-diabetes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = read_csv(filename, names=names)
shape = data.shape
print(shape)
```

The results are listed in rows then columns. You can see that the dataset has 768 rows and 9 columns.

(768, 9)

## Data Type For Each Attribute

The type of each attribute is important. Strings may need to be converted to floating point values or integers to represent categorical or ordinal values. You can get an idea of the types of attributes by peeking at the raw data, as above. You can also list the data types used by the DataFrame to characterize each attribute using the dtypes property.

```
# Data Types for Each Attribute
```

```
from pandas import read_csv
```

```
filename = "pima-indians-diabetes.data.csv"
```

```
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
```

```
data = read_csv(filename, names=names)
```

```
types = data.dtypes
```

```
print(types)
```

You can see that most of the attributes are integers and that mass and pedi are floating point types.

```
preg int64
```

```
plas int64
```

```
pres int64
```

```
skin int64
```

```
test int64
```

```
mass float64
```

```
pedi float64
```

```
age int64
```

```
class int64
```

```
dtype: object
```

## Descriptive Statistics

Descriptive statistics can give you great insight into the shape of each attribute. Often you can create more summaries than you have time to review. The `describe()` function on the Pandas DataFrame lists 8 statistical properties of each attribute. They are:

- Count
- Mean
- Standard Deviation
- Minimum Value
- 25th Percentile
- 50th Percentile (Median)
- 75th Percentile
- Maximum Value

# Statistical Summary

```
from pandas import read_csv
from pandas import set_option

filename = "pima-indians-diabetes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = read_csv(filename, names=names)

set_option('display.width', 100)
set_option('precision', 3)

description = data.describe()

print(description)
```

You can see that you do get a lot of data. You will note some calls to `pandas.set_option()` in the recipe to change the precision of the numbers and the preferred width of the output. This is to make it more readable for this example. When describing your data this way, it is worth taking some time and reviewing observations from the

results. This might include the presence of NA values for missing data or surprising distributions for attributes.

### Class Distribution (Classification Only)

On classification problems you need to know how balanced the class values are. Highly imbalanced problems (a lot more observations for one class than another) are common and may need special handling in the data preparation stage of your project. You can quickly get an idea of the distribution of the class attribute in Pandas.

## Correlations Between Attributes

Correlation refers to the relationship between two variables and how they may or may not change together. The most common method for calculating correlation is Pearson's Correlation Coefficient, that assumes a normal distribution of the attributes involved. A correlation of -1 or 1 shows a full negative or positive correlation respectively. Whereas a value of 0 shows no correlation at all. Some machine learning algorithms like linear and logistic regression can suffer poor performance if there are highly correlated attributes in your dataset. As such, it is a good idea to review all of the pairwise correlations of the attributes in your dataset. You can use the `corr()` function on the Pandas DataFrame to calculate a correlation matrix.

```
# Pairwise Pearson correlations
```

```
from pandas import read_csv
```

```
from pandas import set_option
```

```
filename = "pima-indians-diabetes.data.csv"
```

```
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
```

```
data = read_csv(filename, names=names)
```

```
set_option('display.width', 100)
```

```
set_option('precision', 3)
```

```
correlations = data.corr(method='pearson')
```

```
print(correlations)
```

The matrix lists all attributes across the top and down the side, to give correlation between all pairs of attributes (twice, because the matrix is symmetrical). You can see the diagonal line through the matrix from the top left to bottom right corners of the matrix shows perfect correlation of each attribute with itself.

## Skew of Univariate Distributions

Skew refers to a distribution that is assumed Gaussian (normal or bell curve) that is shifted or squashed in one direction or another. Many machine learning algorithms assume a Gaussian distribution. Knowing that an attribute has a skew may allow you to perform data preparation to correct the skew and later improve the accuracy of your models. You can calculate the skew of each attribute using the `skew()` function on the Pandas DataFrame.

```
# Skew for each attribute
```

```
from pandas import read_csv
```

```
filename = "pima-indians-diabetes.data.csv"
```

```
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
```

```
data = read_csv(filename, names=names)
```

```
skew = data.skew()
```

```
print(skew)
```

The skew result show a positive (right) or negative (left) skew. Values closer to zero show less skew.

```
preg 0.901674
```

```
plas 0.173754
```

```
pres -1.843608
```

```
skin 0.109372
```

test 2.272251

mass -0.428982

pedi 1.919911

age 1.129597

class 0.635017

### Tips To Remember

This section gives you some tips to remember when reviewing your data using summary statistics.

- Review the numbers. Generating the summary statistics is not enough. Take a moment to pause, read and really think about the numbers you are seeing.
- Ask why. Review your numbers and ask a lot of questions. How and why are you seeing specific numbers. Think about how the numbers relate to the problem domain in general and specific entities that observations relate to.
- Write down ideas. Write down your observations and ideas. Keep a small text file or note pad and jot down all of the ideas for how variables may relate, for what numbers mean, and ideas for techniques to try later. The things you write down now while the data is fresh will be very valuable later when you are trying to think up new things to try.

Another excellent way that you can use to better understand your data is by generating plots and charts. In the next lesson you will discover how you can visualize your data for machine learning in Python.



# Understand Your Data With Visualization

You must understand your data in order to get the best results from machine learning algorithms. The fastest way to learn more about your data is to use data visualization. In this chapter you will discover exactly how you can visualize your machine learning data in Python using Pandas. Recipes in this chapter use the Pima Indians onset of diabetes dataset introduced in Chapter 4. Let's get started.



## Univariate Plots

In this section we will look at three techniques that you can use to understand each attribute of your dataset independently.

- Histograms.
- Density Plots.
- Box and Whisker Plots.

## Histograms

A fast way to get an idea of the distribution of each attribute is to look at histograms. Histograms group data into bins and provide you a count of the number of observations in each bin. From the shape of

the bins you can quickly get a feeling for whether an attribute is Gaussian, skewed or even has an exponential distribution. It can also help you see possible outliers.

### # Univariate Histograms

```
from matplotlib import pyplot
from pandas import read_csv
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = read_csv(filename, names=names)
data.hist()
pyplot.show()
```

We can see that perhaps the attributes age, pedi and test may have an exponential distribution. We can also see that perhaps the mass and pres and plas attributes may have a Gaussian or nearly Gaussian distribution. This is interesting because many machine learning techniques assume a Gaussian univariate distribution on the input variables.

## Density Plots

Density plots are another way of getting a quick idea of the distribution of each attribute. The plots look like an abstracted histogram with a smooth curve drawn through the top of each bin, much like your eye tried to do with the histograms.

### # Univariate Density Plots

```
from matplotlib import pyplot
from pandas import read_csv
filename = 'pima-indians-diabetes.data.csv'
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
```

```
data = read_csv(filename, names=names)

data.plot(kind='density', subplots=True, layout=(3,3), sharex=False)

pyplot.show()
```

## Box and Whisker Plots

Another useful way to review the distribution of each attribute is to use Box and Whisker Plots or boxplots for short. Boxplots summarize the distribution of each attribute, drawing a line for the median (middle value) and a box around the 25th and 75th percentiles (the middle 50% of the data). The whiskers give an idea of the spread of the data and dots outside of the whiskers show candidate outlier values (values that are 1.5 times greater than the size of spread of the middle 50% of the data).

```
# Box and Whisker Plots
```

```
from matplotlib import pyplot

from pandas import read_csv

filename = "pima-indians-diabetes.data.csv"

names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']

data = read_csv(filename, names=names)

data.plot(kind='box', subplots=True, layout=(3,3), sharex=False,
sharey=False)

pyplot.show()
```

We can see that the spread of attributes is quite different. Some like age, test and skin appear quite skewed towards smaller values.

## Multivariate Plots

This section provides examples of two plots that show the interactions between multiple variables in your dataset.

## Correlation matrix plot

Correlation gives an indication of how related the changes are between two variables. If two variables change in the same direction they are positively correlated. If they change in opposite directions together (one goes up, one goes down), then they are negatively correlated. You can calculate the correlation between each pair of attributes. This is called a correlation matrix. You can then plot the correlation matrix and get an idea of which variables have a high correlation with each other. This is useful to know, because some machine learning algorithms like linear and logistic regression can have poor performance if there are highly correlated input variables in your data.

```
# Correction Matrix Plot
```

```
from matplotlib import pyplot
```

```
from pandas import read_csv
```

```
import numpy
```

```
filename = 'pima-indians-diabetes.data.csv'
```

```
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
```

```
data = read_csv(filename, names=names)
```

```
correlations = data.corr()
```

```
# plot correlation matrix
```

```
fig = pyplot.figure()
```

```
ax = fig.add_subplot(111)
```

```
cax = ax.matshow(correlations, vmin=-1, vmax=1)
```

```
fig.colorbar(cax)
```

```
ticks = numpy.arange(0,9,1)
```

```
ax.set_xticks(ticks)
```

```
ax.set_yticks(ticks)
ax.set_xticklabels(names)
ax.set_yticklabels(names)
pyplot.show()
```

We can see that the matrix is symmetrical, i.e. the bottom left of the matrix is the same as the top right. This is useful as we can see two different views on the same data in one plot. We can also see that each variable is perfectly positively correlated with each other (as you would have expected) in the diagonal line from top left to bottom right.

## Scatter plot matrix

A scatter plot shows the relationship between two variables as dots in two dimensions, one axis for each attribute. You can create a scatter plot for each pair of attributes in your data. Drawing all these scatter plots together is called a scatter plot matrix. Scatter plots are useful for spotting structured relationships between variables, like whether you could summarize the relationship between two variables with a line. Attributes with structured relationships may also be correlated and good candidates for removal from your dataset.

```
# Scatterplot Matrix
```

```
from matplotlib import pyplot
from pandas import read_csv
from pandas.tools.plotting import scatter_matrix
filename = "pima-indians-diabetes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
data = read_csv(filename, names=names)
scatter_matrix(data)
pyplot.show()
```

Like the Correlation Matrix Plot above, the scatter plot matrix is symmetrical. This is useful to look at the pairwise relationships from different perspectives. Because there is little point of drawing a scatter plot of each variable with itself, the diagonal shows histograms of each attribute.

Now that you know two ways to learn more about your data, you are ready to start manipulating

it. In the next lesson you will discover how you can prepare your data to best expose the

structure of your problem to modeling algorithms.

## Preparing Your Data For Machine Learning

Many machine learning algorithms make assumptions about your data. It is often a very good idea to prepare your data in such way to best expose the structure of the problem to the machine learning algorithms that you intend to use. In this chapter you will discover how to prepare your data for machine learning in Python using scikit-learn. After completing this lesson you will know how to rescale data, standardize data, normalize data, binarize data.



## Need For Data Pre-processing

You almost always need to pre-process your data. It is a required step. A difficulty is that different algorithms make different assumptions about your data and may require different transforms. Further, when you follow all of the rules and prepare your data, sometimes algorithms can deliver better results without pre-processing.

Generally, I would recommend creating many different views and transforms of your data, then exercise a handful of algorithms on each view of your dataset. This will help you to flush out which data transforms might be better at exposing the structure of your problem in general.

## Data Transforms

In this lesson you will work through 4 different data pre-processing recipes for machine learning. The Pima Indian diabetes dataset is used in each recipe. Each recipe follows the same structure:

- Load the dataset from a URL.
- Split the dataset into the input and output variables for machine learning.
- Apply a pre-processing transform to the input variables.
- Summarize the data to show the change.

The scikit-learn library provides two standard idioms for transforming data. Each are useful in different circumstances. The transforms are calculated in such a way that they can be applied to your training data and any samples of data you may have in the future. The scikit-learn documentation has some information on how to use various different pre-processing methods:

- Fit and Multiple Transform.
- Combined Fit-And-Transform.

The Fit and Multiple Transform method is the preferred approach. You call the `fit()` function to prepare the parameters of the transform once on your data. Then later you can use the `transform()` function on the same data to prepare it for modeling and again on the test or validation dataset or new data that you may see in the future. The Combined Fit-And-Transform is a convenience that you can use for one off tasks. This might be useful if you are interested in plotting or summarizing the transformed data. You can review the preprocess API in scikit-learn [here](#) .

## Rescale Data

When your data is comprised of attributes with varying scales, many machine learning algorithms can benefit from rescaling the attributes to all have the same scale. Often this is referred to as normalization and attributes are often rescaled into the range between 0 and 1. This is useful for optimization algorithms used in the core of machine



learning algorithms like gradient descent. It is also useful for algorithms that weight inputs like regression and neural networks and algorithms that use distance measures like k-Nearest Neighbors. You can rescale your data using scikit-learn using the MinMaxScaler class .

```
# Rescale data (between 0 and 1)

from pandas import read_csv

from numpy import set_printoptions

from sklearn.preprocessing import MinMaxScaler

filename = 'pima-indians-diabetes.data.csv'

names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']

dataframe = read_csv(filename, names=names)

array = dataframe.values

# separate array into input and output components

X = array[:,0:8]

Y = array[:,8]

scaler = MinMaxScaler(feature_range=(0, 1))

rescaledX = scaler.fit_transform(X)

# summarize transformed data

set_printoptions(precision=3)
```

## Standardize Data

Standardization is a useful technique to transform attributes with a Gaussian distribution and differing means and standard deviations to a standard Gaussian distribution with a mean of 0 and a standard deviation of 1. It is most suitable for techniques that assume a Gaussian distribution in the input variables and work better with rescaled data, such as linear regression, logistic regression and

linear discriminate analysis. You can standardize data using scikit-learn with the StandardScaler class .

```
# Standardize data (0 mean, 1 stdev)

from sklearn.preprocessing import StandardScaler

from pandas import read_csv

from numpy import set_printoptions

filename = 'pima-indians-diabetes.data.csv'

names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']

dataframe = read_csv(filename, names=names)

array = dataframe.values

# separate array into input and output components

X = array[:,0:8]

Y = array[:,8]

scaler = StandardScaler().fit(X)

rescaledX = scaler.transform(X)

# summarize transformed data

set_printoptions(precision=3)

print(rescaledX[0:5,:])
```

## **Normalize Data**

Normalizing in scikit-learn refers to rescaling each observation (row) to have a length of 1 (called a unit norm or a vector with the length of 1 in linear algebra). This pre-processing method can be useful for sparse datasets (lots of zeros) with attributes of varying scales when using algorithms that weight input values such as neural networks and algorithms that use distance measures such as k-Nearest

Neighbors. You can normalize data in Python with scikit-learn using the Normalizer class .

```
# Normalize data (length of 1)

from sklearn.preprocessing import Normalizer

from pandas import read_csv

from numpy import set_printoptions

filename = 'pima-indians-diabetes.data.csv'

names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']

dataframe = read_csv(filename, names=names)

array = dataframe.values

# separate array into input and output components

X = array[:,0:8]

Y = array[:,8]

scaler = Normalizer().fit(X)

normalizedX = scaler.transform(X)

# summarize transformed data

set_printoptions(precision=3)

print(normalizedX[0:5,:])
```

## **Binarize Data (Make Binary)**

You can transform your data using a binary threshold. All values above the threshold are marked 1 and all equal to or below are marked as 0. This is called binarizing your data or thresholding your data. It can be useful when you have probabilities that you want to make crisp values. It is also useful when feature engineering and you want to add new features that indicate something meaningful.

You can create new binary attributes in Python using scikit-learn with the Binarizer class.

```
# binarization
```

```
from sklearn.preprocessing import Binarizer
```

```
from pandas import read_csv
```

```
from numpy import set_printoptions
```

```
filename = 'pima-indians-diabetes.data.csv'
```

```
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
```

```
dataframe = read_csv(filename, names=names)
```

```
array = dataframe.values
```

```
# separate array into input and output components
```

```
X = array[:,0:8]
```

```
Y = array[:,8]
```

```
binarizer = Binarizer(threshold=0.0).fit(X)
```

```
binaryX = binarizer.transform(X)
```

```
# summarize transformed data
```

```
set_printoptions(precision=3)
```

```
print(binaryX[0:5,:])
```

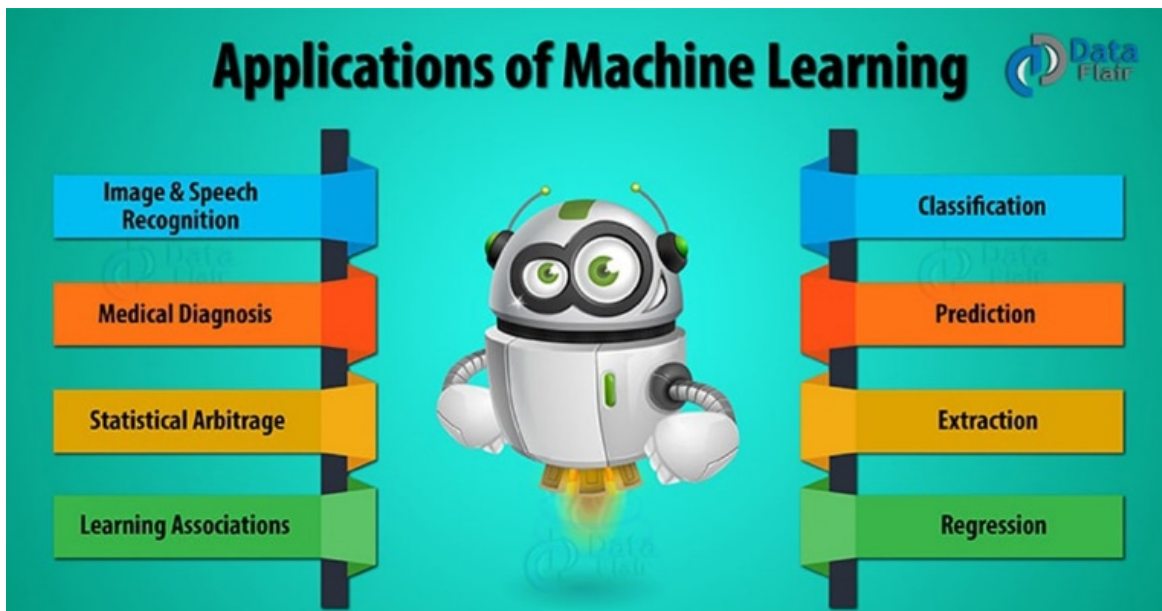
You now know how to transform your data to best expose the structure of your problem to the modeling algorithms. In the next lesson you will discover the real world applications of machine learning.

## Real-World Applications of Machine Learning

As we move forward into the digital age, one of the modern innovations we've seen is the creation of Machine Learning. It is widely being applied and used in the real world today to solve complex problems that would otherwise have been impossible to solve based on traditional approaches and rule-based systems.

This incredible form of artificial intelligence is already being used in various industries and professions. For Example, Image and Speech Recognition, Medical Diagnosis, Prediction, Classification, Learning Associations, Statistical Arbitrage, Extraction, Regression. Today we're looking at all these Machine Learning Applications in today's modern world.

These are the real world Machine Learning Applications, let's see them one by one-



### Image Recognition

It is one of the most common machine learning applications. There are many situations where you can classify the object as a digital image. For digital images, the measurements describe the outputs of each pixel in the image.

In the case of a black and white image, the intensity of each pixel serves as one measurement. So if a black and white image has  $N \times N$  pixels, the total number of pixels and hence measurement is  $N^2$ .

In the coloured image, each pixel considered as providing 3 measurements of the intensities of 3 main colour components ie RGB. So  $N \times N$  coloured image there are  $3 N^2$  measurements.

- For face detection – The categories might be face versus no face present. There might be a separate category for each person in a database of several individuals.
- For character recognition – We can segment a piece of writing into smaller images, each containing a single character. The categories might consist of the 26 letters of the English alphabet, the 10 digits, and some special characters.

## **Speech Recognition**

Speech recognition (SR) is the translation of spoken words into text. It is also known as “automatic speech recognition” (ASR), “computer speech recognition”, or “speech to text” (STT).

In speech recognition, a software application recognizes spoken words. The measurements in this Machine Learning application might be a set of numbers that represent the speech signal. We can segment the signal into portions that contain distinct words or phonemes. In each segment, we can represent the speech signal by the intensities or energy in different time-frequency bands.

Although the details of signal representation are outside the scope of this program, we can represent the signal by a set of real values.

Speech recognition, Machine Learning applications include voice user interfaces. Voice user interfaces are such as voice dialing, call routing, domestic appliance control. It can also use as simple data entry, preparation of structured documents, speech-to-text processing, and plane.

## **Medical Diagnosis**

ML provides methods, techniques, and tools that can help in solving diagnostic and prognostic problems in a variety of medical domains. It is being used for the analysis of the importance of clinical parameters and of their combinations for prognosis, e.g. prediction of disease progression, for the extraction of medical knowledge for outcomes research, for therapy planning and support, and for overall patient management. ML is also being used for data analysis, such as detection of regularities in the data by appropriately dealing with imperfect data, interpretation of continuous data used in the Intensive Care Unit, and for intelligent alarming resulting in effective and efficient monitoring.

It is argued that the successful implementation of ML methods can help the integration of computer-based systems in the healthcare environment providing opportunities to facilitate and enhance the work of medical experts and ultimately to improve the efficiency and quality of medical care.

In medical diagnosis, the main interest is in establishing the existence of a disease followed by its accurate identification. There is a separate category for each disease under consideration and one category for cases where no disease is present. Here, machine learning improves the accuracy of medical diagnosis by analyzing data of patients.

The measurements in this Machine Learning applications are typically the results of certain medical tests (example blood pressure, temperature and various blood tests) or medical diagnostics (such as medical images), presence/absence/intensity of various symptoms and basic physical information about the patient (age, sex, weight etc). On the basis of the results of these measurements, the doctors narrow down on the disease inflicting the patient.

## **Online Customer Support**

A number of websites nowadays offer the option to chat with customer support representative while they are navigating within the site. However, not every website has a live executive to answer your queries. In most of the cases, you talk to a chatbot. These bots tend

to extract information from the website and present it to the customers. Meanwhile, the chatbots advances with time. They tend to understand the user queries better and serve them with better answers, which is possible due to its machine learning algorithms.

## **Statistical Arbitrage**

In finance, statistical arbitrage refers to automated trading strategies that are typical of a short-term and involve a large number of securities. In such strategies, the user tries to implement a trading algorithm for a set of securities on the basis of quantities such as historical correlations and general economic variables. These measurements can be cast as a classification or estimation problem. The basic assumption is that prices will move towards a historical average.

We apply machine learning methods to obtain an index arbitrage strategy. In particular, we employ linear regression and support vector regression (SVR) onto the prices of an exchange-traded fund and a stream of stocks. By using principal component analysis (PCA) in reducing the dimension of feature space, we observe the benefit and note the issues in the application of SVR. To generate trading signals, we model the residuals from the previous regression as a mean reverting process.

In the case of classification, the categories might be sold, buy or do nothing for each security. In the case of estimation one might try to predict the expected return of each security over a future time horizon. In this case, one typically needs to use the estimates of the expected return to make a trading decision(buy, sell, etc.)

## **Learning Associations**

Learning association is the process of developing insights into various associations between products. A good example is how seemingly unrelated products may reveal an association to one another. When analyzed in relation to buying behaviors of customers.



One application of machine learning- Often studying the association between the products people buy, which is also known as basket analysis. If a buyer buys 'X', would he or she force to buy 'Y' because of a relationship that can identify between them? This leads to the relationship that exists between fish and chips etc. when new products launch in the market a Knowing these relationships it develops a new relationship. Knowing these relationships could help in suggesting the associated product to the customer. For a higher likelihood of the customer buying it, It can also help in bundling products for a better package.

This learning of associations between products by a machine is learning associations. Once we found an association by examining a large amount of sales data, Big Data analysts. It can develop a rule to derive a probability test in learning a conditional probability.

## **Search Engine Result Refining**

Google and other search engines use machine learning to improve the search results for you. Every time you execute a search, the algorithms at the backend keep a watch at how you respond to the results. If you open the top results and stay on the web page for long, the search engine assumes that the the results it displayed were in accordance to the q uery. Similarly, if you reach the second or third page of the search results but do not open any of the results, the search engine estimates that the results served did not match re q uirement. This way, the algorithms working at the backend improve the search results.

## **Email Spam and Malware Filtering**

There are a number of spam filtering approaches that email clients use. To ascertain that these spam filters are continuously updated, they are powered by machine learning. When rule-based spam filtering is done, it fails to track the latest tricks adopted by spammers. Multi Layer Perceptron, C 4.5 Decision Tree Induction are some of the spam filtering techni q ues that are powered by ML.

Over 325, 000 malwares are detected everyday and each piece of code is 90–98% similar to its previous versions. The system security programs that are powered by machine learning understand the coding pattern. Therefore, they detect new malware with 2–10% variation easily and offer protection against them.

## **Classification**

Classification is a process of placing each individual from the population under study in many classes. This is identified as independent variables.

Classification helps analysts to use measurements of an object to identify the category to which that object belongs. To establish an efficient rule, analysts use data. Data consists of many examples of objects with their correct classification.

For example, before a bank decides to disburse a loan, it assesses customers on their ability to repay the loan. By considering factors such as customer's earning, age, savings and financial history we can do it. This information is taken from the past data of the loan. Hence, Seeker uses to create a relationship between customer attributes and related risks.

## **Social Media Services**

From personalizing your news feed to better ads targeting, social media platforms are utilizing machine learning for their own and user benefits. Here are a few examples that you must be noticing, using, and loving in your social media accounts, without realizing that these wonderful features are nothing but the applications of ML.

**People You May Know:** Machine learning works on a simple concept: understanding with experiences. Facebook continuously notices the friends that you connect with, the profiles that you visit very often, your interests, workplace, or a group that you share with someone etc. On the basis of continuous learning, a list of Facebook users are suggested that you can become friends with.

Face Recognition: You upload a picture of you with a friend and Facebook instantly recognizes that friend. Facebook checks the poses and projections in the picture, notice the unique features, and then match them with the people in your friend list. The entire process at the backend is complicated and takes care of the precision factor but seems to be a simple application of ML at the front end.

Similar Pins: Machine learning is the core element of Computer Vision, which is a technique to extract useful information from images and videos. Pinterest uses computer vision to identify the objects (or pins) in the images and recommend similar pins accordingly.

## **Prediction**

Consider the example of a bank computing the probability of any of loan applicants faulting the loan repayment. To compute the probability of the fault, the system will first need to classify the available data in certain groups. It is described by a set of rules prescribed by the analysts.

Once we do the classification, as per need we can compute the probability. These probability computations can compute across all sectors for varied purposes

The current prediction is one of the hottest machine learning algorithms. Let's take an example of retail, earlier we were able to get insights like sales report last month / year / 5-years / Diwali / Christmas. These type of reporting is called as historical reporting. But currently business is more interested in finding out what will be my sales next month / year / Diwali, etc.

So that business can take a required decision (related to procurement, stocks, etc.) on time.

## **Extraction**

Information Extraction (IE) is another application of machine learning. It is the process of extracting structured information from

unstructured data. For example web pages, articles, blogs, business reports, and e-mails. The relational database maintains the output produced by the information extraction.

The process of extraction takes input as a set of documents and produces a structured data. This output is in a summarized form such as an excel sheet and table in a relational database.

Nowadays extraction is becoming a key in the big data industry.

As we know that the huge volume of data is getting generated out of which most of the data is unstructured. The first key challenge is handling unstructured data. Now conversion of unstructured data to structured form based on some pattern so that the same can stored in RDBMS.

Apart from this in current days data collection mechanism is also getting change. Earlier we collected data in batches like End-of-Day (EOD), but now business wants the data as soon as it is getting generated, i.e. in real time.

## **Regression**

We can apply Machine learning to regression as well.

Assume that  $x = x_1, x_2, x_3, \dots, x_n$  are the input variables and  $y$  is the outcome variable. In this case, we can use machine learning technology to produce the output ( $y$ ) on the basis of the input variables ( $x$ ). You can use a model to express the relationship between various parameters as below:

$Y = g(x)$  where  $g$  is a function that depends on specific characteristics of the model.

In regression, we can use the principle of machine learning to optimize the parameters. To cut the approximation error and calculate the closest possible outcome.

We can also use Machine learning for function optimization. We can choose to alter the inputs to get a better model. This gives a new

and improved model to work with. This is known as response surface design.

The following list depicts some other real-world applications of Machine Learning.

- Product recommendations in online shopping platforms
- Sentiment and emotion analysis
- Anomaly detection
- Fraud detection and prevention
- Content recommendation (news, music, movies, and so on)
- Weather forecasting
- Stock market forecasting
- Market basket analysis
- Customer segmentation
- Churn analytics
- Click through predictions
- Failure/defect detection and prevention
- E-mail spam filtering

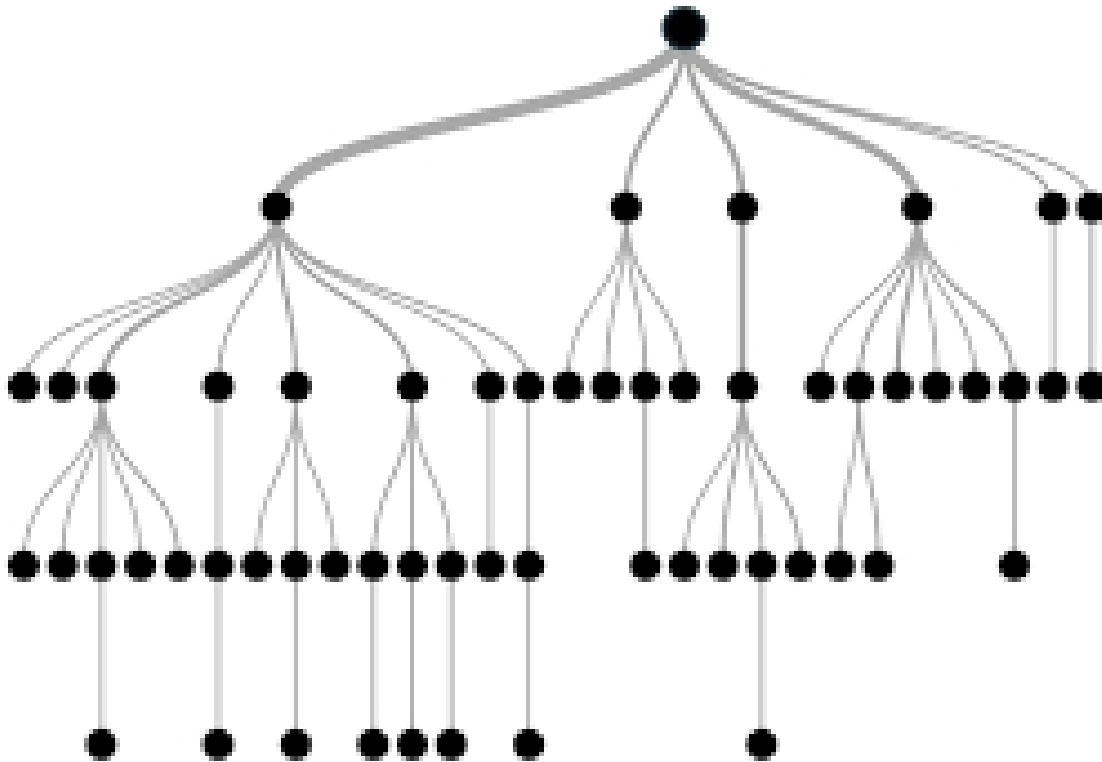
By now, you should be familiarized with the foundations of Machine Learning before taking a deep dive into Machine Learning pipelines and solving real-world problems. The need for Machine Learning in today's world is introduced in the chapter with a focus on making data-driven decisions at scale.

So, this was all about Machine Learning Applications. Hope you like our explanation. Machine learning is an incredible breakthrough in the field of artificial intelligence. While it does have some frightening implications when you think about it, these Machine Learning Applications are several of the many ways this technology can improve our lives.

# Using Tree-Based Algorithms for Advertising Click-Through Prediction

Online display advertising is a multibillion-dollar industry. It comes in different formats including banner ads composed of text, images, flash, and rich media such as audio and video. Advertisers or their agencies place advertisements on a variety of websites, even mobile apps across the Internet, to reach potential customers and deliver an advertising message.

In this chapter and the next, we will be solving one of the most important machine learning problems in digital online advertising, click-through prediction—given a user and the page they are visiting, how likely they will click on a given ad. We will be herein focusing on learning tree-based algorithms, decision tree and random forest, and utilizing them to tackle the billion dollar problem.



Display online advertising has served as one of the greatest examples for machine learning utilization. Obviously, advertisers as well as consumers ourselves, are keenly interested in well-targeted ads. The industry has heavily relied on the ability of machine learning models to predict the ad targeting effectiveness: how likely the audience in a certain age group will be interested in this product, customers with certain household income will purchase this product after seeing its ad, frequent sport sites visitors will spend more time in reading this ad, and so on. The most common measurement of effectiveness is the click-through rate (CTR), which is the ratio of clicks on a specific ad to its total number of views. The higher CTR in general, the better targeted an ad is, the more successful an online advertising campaign is.

Click-through prediction holds both promise of and challenges for machine learning. It mainly involves binary classification of whether a given ad on a given page (or app) will be clicked by a given user, with predictive features from these three aspects, including:

Ad content and information (category, position, text, format, and so on) Page content and publisher information (category, context, domain, and so on)

User information (age, gender, location, income, interests, search history, browsing history, device, and so on)

Suppose we, as an agency, are operating ads on behalf of several advertisers and our job is to display the right ads to the right audience. With an existing dataset in hand (the following small chunk as an example, whose number of predictive features can easily go up to thousands in reality) taken from millions of records of campaigns running last month, we need to develop a classification model to learn and predict the future ad placement outcome.

## **Getting Started With Numerical And Categorical Data**

At first glance, the features in the preceding dataset are categorical, for example, male or female, one of four age groups, one of the

predefined site categories, whether or not being interested in sports. Such types of data are different from the numerical type of feature data that we have worked with until now.

Categorical (also called qualitative) features represent characteristics, distinct groups, and a countable number of options. Categorical features may or may not have logical order. For example, household income from low, median to high, is an ordinal feature, while the category of an ad is not ordinal. Numerical (also called quantitative) features, on the other hand, have mathematical meaning as a measurement and of course are ordered. For instance, term frequency and the tf-idf variant are respectively discrete and continuous numerical features; the cardiotocography dataset contains both discrete (such as number of accelerations per second, number of fetal movements per second) and continuous (such as mean value of long term variability) numerical features.

Categorical features can also take on numerical values. For example, 1 to 12 represents months of the year, 1 and 0 indicates male and female. Still, these values do not contain mathematical implication.

Among the two classification algorithms, naive Bayes and SVM, which we learned about previously, the naive Bayes classifier works for both numerical and categorical features as likelihoods or are calculated in the same way, while SVM requires features to be numerical in order to compute margins.

Now if we think of predicting click or not click with naive Bayes, and try to explain the model to our advertiser clients, our clients will find it hard to understand the prior likelihood of individual attributes and their multiplication. Is there a classifier that is easy to interpret, explain to clients, and also able to handle categorical data?

Decision tree!

## **Decision Tree Classifier**

A decision tree is a tree-like graph, a sequential diagram illustrating all of the possible decision alternatives and the corresponding



outcomes. Starting from the root of a tree, every internal node represents what a decision is made based on; each branch of a node represents how a choice may lead to the next nodes; and finally, each terminal node, the leaf, represents an outcome yielded.

For example, we have just made a couple of decisions that brought us to the action of learning decision tree to solve our advertising problem:

The decision tree classifier operates in the form of a decision tree. It maps observations to class assignments (symbolized as leaf nodes), through a series of tests (represented as internal nodes) based on feature values and corresponding conditions (represented as branches). In each node, a question regarding the values and characteristics of a feature is asked; based on the answer to the question, observations are split into subsets. Sequential tests are conducted until a conclusion about the observations' target label is reached. The paths from root to end leaves represent the decision making process, the classification rules.

The following figure shows a much simplified scenario where we want to predict click or no click on a self-driven car ad, we manually construct a decision tree classifier that works for an available dataset. For example, if a user is interested in technology and they have a car, they will tend to click the ad; for a person outside of this subset, if the person is a high-income female, then she is unlikely to click the ad. We then use the learned tree to predict two new inputs, whose results are click and no click, respectively.

After a decision tree has been constructed, classifying a new sample is straightforward as we just saw: starting from the root, apply the test condition and follow the branch accordingly until a leaf node is reached and the class label associated will be assigned to the new sample.

So how can we build an appropriate decision tree?

## **The Construction Of A Decision Tree**

A decision tree is constructed by partitioning the training samples into successive subsets. The partitioning process is repeated in a recursive fashion on each subset. For each partitioning at a node, a condition test is conducted based on a value of a feature of the subset. When the subset shares the same class label, or no further splitting can improve the class purity of this subset, recursive partitioning on this node is finished.

Theoretically, for a partitioning on a feature (numerical or categorical) with  $n$  different values, there are  $n$  different ways of binary splitting (yes or no to the condition test), not to mention other ways of splitting. Without considering the order of features partitioning takes place on, there are already possible trees for an  $m$ -dimensional dataset.

Many algorithms have been developed to efficiently construct an accurate decision tree. Popular ones include:

- ID3 (Iterative Dichotomiser 3): which uses a greedy search in a topdown manner by selecting the best attribute to split the dataset on each iteration backtracking
- C4.5: an improved version on ID3 by introducing backtracking where it traverses the constructed tree and replaces branches with leaf nodes if purity is improved this way.
- CART (Classification and Regression Tree): which we will discuss in detail
- CHAID (Chi-squared Automatic Interaction Detector): which is often used in direct marketing in practice. It involves complicated statistical concepts, but basically determines the optimal way of merging predictive variables in order to best explain the outcome.

The basic idea of these algorithms is to grow the tree greedily by making a series of local optimizations on choosing the most significant feature to use for partitioning the data. The dataset is then split based on the optimal value of that feature. We will discuss the measurement of significant features and optimal splitting value of a feature in the next section.

We will now study in detail and implement the CART algorithm as the most notable decision tree algorithm in general. It constructs the tree by binary splitting and growing each node into left and right children. In each partition, it greedily searches for the most significant combination of features and their values, where all different possible combinations are tried and tested using a measurement function. With the selected feature and value as a splitting point, it then divides the data in a way that:

- Samples with the feature of this value (for a categorical feature) or greater value (for a numerical feature) becomes the right child
- The remainder becomes the left child

The preceding partitioning process repeats and recursively divides up the input samples into two subgroups. When the dataset becomes unmixed, a splitting process stops at a subgroup where any of the following two criteria meet:

- Minimum number of samples for a new node: When the number of samples is not greater than the minimum number of samples required for a further split, a partitioning stops in order to prevent the tree from excessively tailoring to the training set and as a result overfitting.
- Maximum depth of the tree: A node stops growing when its depth, which is defined as the number of partitioning taking place top-down starting from the root node, is not less than the maximum tree depth. Deeper trees are more specific to the training set and lead to overfitting.

A node with no branch out becomes a leaf and the dominant class of samples at this node is served as the prediction. Once all splitting processes finish, the tree is constructed and is portrayed with the information of assigned labels at terminal nodes and splitting points (feature + value) at all preceding internal nodes.

We will implement the CART decision tree algorithm from scratch after studying the metrics of selecting the optimal splitting features

and values as promised.

## The Metrics To Measure A Split

When selecting the best combination of features and values as the splitting point, two criteria, Gini impurity and information gain, can be used to measure the quality of separation.

Gini impurity as its name implies, measures the class impurity rate, the class mixture rate. For a dataset with K classes, suppose data from class (

$$1 \leq k \leq K$$

) takes up a fraction

$$f_k$$

(

$$0 \leq f_k \leq 1$$

) of the entire dataset, the Gini impurity of such a dataset is written as follows:

$$\text{Gini impurity} = 1 - \sum_{k=1}^K f_k^2$$

Lower Gini impurity indicates a purer dataset. For example, when the dataset contains only one class, say the fraction of this class is 1 and that of others is 0, its Gini impurity becomes  $1 - (1^2 + 0^2) = 0$ . In another example, a dataset records a large number of coin flips where heads and tails take up half of the samples, the Gini impurity is  $1 - (0.5^2 + 0.5^2) = 0.5$ .

In binary cases, Gini impurity under different values of the positive class's fraction can be visualized by the following code:

```
>>> import matplotlib.pyplot as plt >>>
```

```
import numpy as np
```

A fraction of the positive class varies from 0 to 1:

```
>>> pos_fraction = np.linspace(0.00, 1.00, 1000)
```

Gini impurity is calculated accordingly, followed by the plot of Gini impurity versus positive proportion:

```
>>> gini = 1 - pos_fraction**2 - (1-pos_fraction)**2
```

```
>>> plt.plot(pos_fraction, gini)
```

```
>>> plt.ylim(0, 1)
```

```
>>> plt.xlabel('Positive fraction')
```

```
>>> plt.ylabel('Gini Impurity')
```

```
>>> plt.show()
```

Given labels of a dataset, we can implement the Gini impurity calculation function as follows:

```
>>> def gini_impurity(labels):
```

```
...     # When the set is empty, it is also pure
```

```
...     if not labels:
```

```
...         return 0
```

```
...     # Count the occurrences of each label
```

```
...     counts = np.unique(labels, return_counts=True)[1]
```

```
...     fractions = counts / float(len(labels))
```

```
...     return 1 - np.sum(fractions ** 2)
```

Test it out with some examples:

```
>>> print('{0:.4f}'.format(gini_impurity([1, 1, 0, 1, 0]))) 0.4800
```

```
>>> print('{0:.4f}'.format(gini_impurity([1, 1, 0, 1, 0, 0])))
```

```
0.5000
```

```
>>> print('{0:.4f}'.format(gini_impurity([1, 1, 1, 1])))
```

0.0000

In order to evaluate the quality of a split, we simply add up the Gini impurity of all resulting subgroups combining the proportions of each subgroup as corresponding weight factors. And again, the smaller weighted sum of Gini impurity, the better the split.

Another metric, information gain, measures the improvement of purity after splitting, in other words, the reduction of uncertainty due to a split. Higher information gain implies a better splitting. We obtain the information gain of a split by comparing the entropy before and after the split.

Entropy after a split is calculated as the weighted sum of entropy of each child, similarly to the weighted Gini impurity.

During the process of constructing a node at a tree, our goal is to search for a splitting point where the maximal information gain is obtained. As the entropy of the parent node is unchanged, we just need to measure the entropy of resulting children due to a split. The better split is the one with a less entropy of resulting children.

To understand it better, we will look at the self-driving car ad example again:

For the first option, the entropy after split can be calculated as follows:

$$\begin{aligned} \#1 \text{ entropy} &= \frac{3}{5} \left[ -\left( \frac{2}{3} * \log_2 \frac{2}{3} + \frac{1}{3} * \log_2 \frac{1}{3} \right) \right] + \frac{2}{5} \left[ -\left( \frac{1}{2} * \log_2 \frac{1}{2} + \frac{1}{2} * \log_2 \frac{1}{2} \right) \right] \\ &= 0.951 \end{aligned}$$

The second split can be calculated as follows:

$$\#2 \text{ entropy} = \frac{2}{5} \left[ -(1 * \log_2 1 + 0) \right] + \frac{3}{5} \left[ -\left( \frac{1}{3} * \log_2 \frac{1}{3} + \frac{2}{3} * \log_2 \frac{2}{3} \right) \right] = 0.551$$

For exploration, we can also calculate their information gain as follows:

$$\text{Entropy before} = -\left(\frac{3}{5} * \log_2 \frac{2}{3} + \frac{2}{5} * \log_2 \frac{2}{5}\right) = 0.971$$

$$\text{\#1 information gain} = 0.971 - 0.951 = 0.020$$

$$\text{\#2 information gain} = 0.971 - 0.551 = 0.420$$

According to the information gain/entropy-based evaluation, the second split is preferable, which is also concluded based on the Gini impurity criterion.

In general, the choice of two metrics, Gini impurity and information gain, has little effect on the performance of the trained decision tree. They both measure the weighted impurity of children after a split. We can combine them into one function calculating the weighted impurity:

```
>>> criterion_function = {'gini': gini_impurity,
                           'entropy': entropy}

>>> def weighted_impurity(groups, criterion='gini'):
...     """ Calculate weighted impurity of children after a spl
...     Args:
...
...         groups (list of children, and a child consists a
li             of class label ...         criterion
(metric to measure the quality of a split
...
...         'gini' for Gini Impurity or 'entropy'
for             Information Gain)
...
...     Returns:
...         float, weighted impurity
...
...     """
...
...     total = sum(len(group) for group in groups)
...     weighted_sum = 0.0 ...     for group in groups:
...         weighted_sum += len(group) / float(total )
```

```

        * criterion_function[criterion](gro ...    return
weighted_sum

```

Test it with the example we just hand calculated:

```

>>> children_1 = [[1, 0, 1], [0, 1]]
>>> children_2 = [[1, 1], [0, 0, 1]]
>>> print('Entropy of #1 split:
        {0:.4f}'.format(weighted_impurity(children_1, 'entropy
Entropy of #1 split: 0.9510 >>> print('Entropy of #2 split:
        {0:.4f}'.format(weighted_impurity(children_2,      'entropy
Entropy of #2 split: 0.5510

```

## The Implementations of Decision Tree

With a solid understanding of partitioning evaluation metrics, let's practice the CART tree algorithm by hand on a simulated dataset:

To begin, we decide on the first splitting point, the root, by trying out all possible values for each of two features. We utilize the `weighted_impurity` function we just defined to calculate the weighted Gini impurity for each possible combination:

*Gini(interest, Tech) = weighted\_impurity([[1, 1, 0], [0, 0, 0, 1]]) = 0.405*

*Gini(interest, Fashion) = weighted\_impurity([[0, 0], [1, 0, 1, 0, 1]]) = 0.343*

*Gini(interest, Sports) = weighted\_impurity([[0, 1], [1, 0, 0, 1, 0]]) = 0.486*

*Gini(occupation, Professional) = weighted\_impurity([[0, 0, 1, 0], [1, 0, 1]]) = 0.405*



$$Gini(occupation, Student) = weighted\_impurity([[0, 0, 1, 0], [1, 0, 1]]) =$$

$$0.405$$

$$Gini(occupation, Retired) = weighted\_impurity([[1, 0, 0, 0, 1, 1], [1]]) =$$

$$0.429$$

The root goes to the user interest feature with the fashion value. We can now build the first level of the tree:

If we are satisfied with a one level deep tree, we can stop here by assigning the right branch label 0 and the left branch label 1 as the majority class. Alternatively, we can go further down the road constructing the second level from the left branch (the right branch cannot be further split):

$$Gini(interest, Tech) = weighted\_impurity([[0, 1], [1, 1, 0]]) = 0.467$$

$$Gini(interest, Sports) = weighted\_impurity([[1, 1, 0], [0, 1]]) = 0.467$$

$$Gini(occupation, Professional) = weighted\_impurity([[0, 1, 0], [1, 1]]) = 0.267$$

$$Gini(occupation, Student) = weighted\_impurity([[1, 0, 1], [0, 1]]) = 0.467$$

$$Gini(occupation, Retired) = weighted\_impurity([[1, 0, 1, 1], [0]]) = 0.300$$

With the second splitting point specified by (occupation, professional) with the least Gini impurity, our tree will now look as follows:

We can repeat the splitting process as long as the tree does not exceed the maximal depth and the node contains enough samples.

It is now time for coding after the process of tree construction is clear.

We start with the criterion of best splitting point: the calculation of weighted impurity of two potential children is as what we defined previously, while that of two metrics is slightly different where the inputs now become numpy arrays for computational efficiency:

```
>>> def gini_impurity(labels):  
...     # When the set is empty, it is also pure  
...     if labels.size == 0:  
...         return 0  
...     # Count the occurrences of each label  
...     counts = np.unique(labels, return_counts=True)[1]  
...     fractions = counts / float(len(labels)) ...     return 1 -  
np.sum(fractions ** 2)
```

```
>>> def entropy(labels):  
...     # When the set is empty, it is also pure  
...     if labels.size == 0:  
...         return 0  
...     counts = np.unique(labels, return_counts=True)[1]  
...     fractions = counts / float(len(labels))  
...     return - np.sum(fractions * np.log2(fractions))
```

Next, we define a utility function to split a node into left and right child based on a feature and a value:

```
>>> def split_node(X, y, index, value):  
...     """ Split data set X, y based on a feature and a value  
...     Args:  
...         X, y (numpy.ndarray, data set)
```

```

...     index (int, index of the feature used for splitting ...
value (value of the feature used for splitting) ...    Returns:

...         list, list: left and right child, a child is in
the         format of [X, y]
...     """
...     x_index = X[:, index]
...     # if this feature is numerical
...     if X[0, index].dtype.kind in ['i', 'f']:
...         mask = x_index >= value
...     # if this feature is categorical ...    else:
...         mask = x_index == value
...     # split into left and right child
...     left = [X[~mask, :], y[~mask]]
...     right = [X[mask, :], y[mask]]
...     return left, right

```

Note, that we check whether the feature is numerical or categorical and split the data accordingly.

With the splitting measurement and generation functions available, we now define the greedy search function trying out all possible splits and returning the best one given a selection criterion, along with the resulting children:

```

>>> def get_best_split(X, y, criterion):
...     """ Obtain the best splitting point and resulting
child         for the data set X, y ...    Args:
...     X, y (numpy.ndarray, data set) ...    criterion (gini or
entropy) ...    Returns:

```

```

...         dict {index: index of the feature, value:
feature      value, children: left and right children}
...         """
...
...         best_index, best_value, best_score, children
=            None, None, 1, None
...     for index in range(len(X[0])):
...         for value in np.sort(np.unique(X[:, index])):
...             groups = split_node(X, y, index, value)
...             impurity = weighted_impurity(
...                 [groups[0][1], groups[1][1]], criteri
...             if impurity < best_score:
...                 best_index, best_value, best_score,
childre      index, value, impurity, gro
...     return {'index': best_index, 'value': best_value,
...             'children': children}

```

The preceding selection and splitting process occurs in a recursive manner on each of the subsequent children. When a stopping criterion meets, a process at a node stops and the major label will be assigned to this leaf node:

```

>>> def get_leaf(labels):
...     # Obtain the leaf as the majority of the labels ...     return
np.bincount(labels).argmax()

```

And finally the recursive function that links all these together by:

```

:
    Assigning a leaf node if one of two children nodes is empty

```

Assigning a leaf node if the current branch depth exceeds the maximal

depth allowed

- Assigning a leaf node if it does not contain sufficient samples required for a further split
- Otherwise, proceeding with further splits with the optimal splitting point

```
>>> def split(node, max_depth, min_size, depth, criterion):
...     """ Split children of a node to construct new nodes
or      assign them terminals ...   Args:
...     node (dict, with children info)
...           max_depth (int, maximal depth of the tree)
...           min_size (int, minimal samples required to
further          split a child)
...     depth (int, current depth of the node)
...     criterion (gini or entropy)
...     """
...     left, right = node['children'] ...   del (node['children'])
...     if left[1].size == 0:
...         node['right'] = get_leaf(right[1])
...         return
...     if right[1].size == 0:
...         node['left'] = get_leaf(left[1])
...         return
...     # Check if the current depth exceeds the maximal depth
...     if depth >= max_depth:
...         node['left'], node['right'] =
...             get_leaf(left[1]), get_leaf(right[1])
```

```

...     return
...     # Check if the left child has enough samples
...     if left[1].size <= min_size: ...         node['left'] =
get_leaf(left[1]) ...     else:
...     # It has enough samples, we further split it
...         result = get_best_split(left[0], left[1], criterion
... result_left, result_right = result['children']
...     if result_left[1].size == 0:
...         node['left'] = get_leaf(result_right[1])
...     elif result_right[1].size == 0:
...         node['left'] = get_leaf(result_left[1])
...     else:
...         node['left'] = result
...
...         split(node['left'], max_depth,
min_size,
depth + 1, criterion ...     #
Check if the right child has enough samples
...     if right[1].size <= min_size: ...         node['right'] =
get_leaf(right[1]) ...     else:
...     # It has enough samples, we further split it
...         result = get_best_split(right[0], right[1], criteri
... result_left, result_right = result['children']
...     if result_left[1].size == 0:
...         node['right'] = get_leaf(result_right[1])
...     elif result_right[1].size == 0:
...         node['right'] = get_leaf(result_left[1])
...     else:

```

```
...     node['right'] = result
```

```
...                                     split(node['right'], max_depth,  
min_size,                             depth + 1, criteri
```

Plus, the entry point of the tree construction:

```
>>> def train_tree(X_train, y_train, max_depth,  
min_size, criterion='gini'): ... """ Construction of a  
tree starts here ... Args:
```

```
...     X_train, y_train (list, list, training data)
```

```
...         max_depth (int, maximal depth of the tree)  
...         min_size (int, minimal samples required to  
further             split a child)
```

```
...     criterion (gini or entropy)
```

```
...     """
```

```
...     X = np.array(X_train)
```

```
...     y = np.array(y_train)
```

```
...     root = get_best_split(X, y, criterion)
```

```
...     split(root, max_depth, min_size, 1, criterion) ... return root
```

Now let's test it with the preceding hand-calculated example:

```
>>> X_train = [['tech', 'professional'],
```

```
...           ['fashion', 'student'],
```

```
...           ['fashion', 'professional'],
```

```
...           ['sports', 'student'],
```

```
...           ['tech', 'student'],
```

```
...           ['tech', 'retired'],
```

```
...           ['sports', 'professional']]
```

```
>>> y_train = [1, 0, 0, 0, 1, 0, 1]
```

```
>>> tree = train_tree(X_train, y_train, 2, 2)
```

To verify that the trained tree is identical to what we constructed by hand, we will write a function displaying the tree:

```
>>> CONDITION = {'numerical': {'yes': '>=', 'no': '<'},
```

```
...         'categorical': {'yes': 'is', 'no': 'is not'}}
```

```
>>> def visualize_tree(node, depth=0):
```

```
...     if isinstance(node, dict):
```

```
...         if node['value'].dtype.kind in ['i', 'f']: ...         condition =
CONDITION['numerical']
```

```
...         else:
```

```
...             condition = CONDITION['categorical'] ...             print('{}|-
X{} {} {}'.format(depth * ' ', node['index'] + 1,
condition['no'], node['value
```

```
...         if 'left' in node:
```

```
...             visualize_tree(node['left'], depth + 1) ...             print('{}|-
X{} {} {}'.format(depth * ' ', node['index'] + 1,
condition['yes'], node['value
```

```
...         if 'right' in node:
```

```
...             visualize_tree(node['right'], depth + 1) ...         else:
```

```
...             print('{}[{}]'.format(depth * ' ', node))
```

```
>>> visualize_tree(tree)
```

```
|- X1 is not fashion
```

```
    |- X2 is not professional
```

```
        [0]
```

```
    |- X2 is professiona l
```



**[1]**

**|- X1 is fashion [0]**

We can test it with a numerical example:

```
>>> X_train_n = [[6, 7],
```

```
...      [2, 4],
```

```
...      [7, 2],
```

```
...      [3, 6],
```

```
...      [4, 7],
```

```
...      [5, 2],
```

```
...      [1, 6],
```

```
...      [2, 0],
```

```
...      [6, 3],
```

```
...      [4, 1]]
```

```
>>> y_train_n = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
```

```
>>> tree = train_tree(X_train_n, y_train_n, 2, 2)
```

```
>>> visualize_tree(tree)
```

**|- X2 < 4**

**|- X1 < 7**

**[1]**

**|- X1 >= 7**

**[0]**

**|- X2 >= 4**

**|- X1 < 2**

[1 ]

|- X1 >= 2

[0]

Now that we have a more solid understanding of decision tree by realizing it from scratch, we can try the decision tree package from scikit-learn, which is already well developed:

```
>>> from sklearn.tree import DecisionTreeClassifier >>> tree_sk  
=  
DecisionTreeClassifier(criterion='gini',                                max_  
depth=2,    min_samples_split    >>>    tree_sk.fit(X_train_n,  
y_train_n)
```

To visualize the tree we just built, we utilize the built-in function, `export_graphviz`, as follows:

```
>>>                                export_graphviz(tree_sk,  
out_file='tree.dot',                feature_names=['X1', 'X2'],  
impurity=False, filled=True         class_names=['0', '1'])
```

This will generate a `tree.dot` file, which can be converted to a PNG image file using GraphViz software (installation instructions can be found in <http://www.graphviz.org/>) by running the command `dot -Tpng tree.dot -o tree.png` in the Terminal.

The tree generated is essentially the same as the one we had before.

## Click-Through Prediction With Decision Tree

After several examples, it is now time to predict ad click-through with the decision tree algorithm we just thoroughly learned and practiced. We will use the dataset from a Kaggle machine learning competition Click-Through Rate Prediction (<https://www.kaggle.com/c/avazu-ctr-prediction>).

For now, we only take the first 100,000 samples from the train file (unzipped from the `train.gz` file from <https://www.kaggle.com/c/>

avazu-ctrprediction/data) for training the decision tree and the first 100,000 samples from the test file (unzipped from the test.gz file from the same page) for prediction purposes.

The data fields are described as follows:

- id: ad identifier, such as 1000009418151094273, 10000169349117863715
- click: 0 for non-click, 1 for click
- hour: in the format of YYMMDDHH, for example, 14102100
- C1: anonymized categorical variable, such as 1005, 1002
- banner\_pos: where a banner is located, 1 and 0
- site\_id: site identifier, such as 1fbe01fe, fe8cc448, d6137915
- site\_domain: hashed site domain, such as 'bb1ef334', 'f3845767
- site\_category: hashed site category, such as 28905ebd, 28905ebd
- app\_id: mobile app identifier
- app\_domain app\_category
- device\_id: mobile device identifier
- device\_ip: IP address
- device\_model: such as iPhone 6, Samsung, hashed by the way
- device\_type: such as tablet, smartphone, hashed
- device\_conn\_type: Wi-Fi or 3G for example, again hashed in the data
- C14-C21: anonymized categorical variables

We take a glance at the data by running the command `head train | sed 's/,,/, ,/g;s/,,/, ,/g' | column -s, -t:`

```
id app_domain app_category device_id device_ip device_model dev
1000009418151094273 0 14102100 1005 0 1fbe01fe f3845767
289
10000169349117863715 0 14102100 1005 0 1fbe01fe
f3845767 28
```

10000371904215119486	0	14102100	1005	0	1fbe01fe f3845767 28
10000640724480838376	0	14102100	1005	0	1fbe01fe f3845767 28
10000679056417042096	0	14102100	1005	1	fe8cc448 9166c161 05
10000720757801103869	0	14102100	1005	0	d6137915 bb1ef334 f0
10000724729988544911	0	14102100	1005	0	8fda644b 25d4cfcd f0
10000918755742328737	0	14102100	1005	1	e151e245 7e091613 f0
10000949271186029916	1	14102100	1005	0	1fbe01fe f3845767 28

Don't be scared by the anonymized and hashed values. They are categorical features and each possible value of them corresponds to a real and meaningful value, but it is presented this way due to the privacy policy. Maybe C1 means user gender, and 1005 and 1002 represent male and female respectively.

Now let's get started with reading the dataset:

```
>>> import csv

>>> def read_ad_click_data(n, offset=0):
...     X_dict, y = [], []
...     with open('train', 'r') as csvfile:
...         reader = csv.DictReader(csvfile)
...         for i in range(offset):
...             reader.next()
...         i = 0
```

```

...     for row in reader:
...         i += 1
...         y.append(int(row['click']))
...         del row['click'], row['id'],
row['hour'], row['device_id'], row['device_ip']
...         X_dict.append(row)
...         if i >= n:
...             break
...     return X_dict, y

```

Note, that we at this moment exclude the id, hour, and device\_id, device\_ip from features: >>> n\_max = 100000

```
>>> X_dict_train, y_train = read_ad_click_data('train', n_max)
```

```
>>> print(X_dict_train[0])
```

```
{'C21': '79', 'site_id': '1fbe01fe', 'app_id': 'ecad2386', 'C19':
'35', 'C18': '0', 'device_type': '1', 'C17': '1722', 'C15': '32',
'C14': '15706', 'C16': '50', 'device_conn_type': '2', 'C1':
'1005', 'app_category': '07d7df22', 'site_category': '28905ebd',
'app_domain': '7801e8d9', 'site_domain': 'f3845767', 'banner_po':
'0', 'C20': '-1', 'device_model': '44956a24'}
```

```
>>> print(X_dict_train[1])
```

```
{'C21': '79', 'site_id': '1fbe01fe', 'app_id': 'ecad2386', 'C19':
'35', 'C18': '0', 'device_type': '1', 'C17': '1722', 'C15': '32',
'C14': '15704', 'C16': '50', 'device_conn_type': '0', 'C1':
'1005', 'app_category': '07d7df22', 'site_category': '28905ebd',
'app_domain': '7801e8d9', 'site_domain': 'f3845767', 'banner_po':
'0', 'C20': '-1', 'device_model': '44956a24'}
```

```
'0', 'C20': '100084', 'device_model': '711ee120'}
```

Next, we transform these dictionary objects (feature: value) into one-hot encoded vectors using DictVectorizer. We will talk about one-hot encoding in the next chapter. It basically converts a categorical feature with  $k$  possible values into  $k$  binary features. For example, the site category feature with three possible values, news, education, and sports, will be encoded into three binary features, is\_news, is\_education, and is\_sports. The reason we do such transformation is because the tree-based algorithms in scikit-learn (current version 0.18.1) only allow numerical feature input:

```
>>> from sklearn.feature_extraction import DictVectorizer
>>> dict_one_hot_encoder = DictVectorizer(sparse=False)
>>> X_train = dict_one_hot_encoder.fit_transform(X_dict_train)
>>> print(len(X_train[0]))
5725
```

We transformed the original 19-dimension categorical features into 5725dimension binary features.

Similarly, we construct the testing dataset:

```
>>> X_dict_test, y_test = read_ad_click_data(n, n)
>>> X_test = dict_one_hot_encoder.transform(X_dict_test)
>>> print(len(X_test[0]))
5725
```

Next, we train a decision tree model using the grid search techniques we learned in the last chapter. For demonstration, we will only tweak the max\_depth parameter, but other parameters, for example min\_samples\_split and class\_weight are recommended. Note that the classification metric should be AUC of ROC, as it is an imbalanced binary case (only 17490 out of 100000 training samples are clicks):

```

>>> from sklearn.tree import DecisionTreeClassifier

>>> parameters = {'max_depth': [3, 10, None]}

>>> decision_tree =
DecisionTreeClassifier(criterion='gini', min
_samples_split=30

>>> from sklearn.model_selection import GridSearchCV >>>
grid_search = GridSearchCV(decision_tree,
parameters, n_jobs=-1, cv=3, scoring='roc_auc')

>>> grid_search.fit(X_train, y_train)

>>> print(grid_search.best_params_) {'max_depth': 10}

Use the model with the optimal parameter to predict unseen cases:

>>> decision_tree_best = grid_search.best_estimator_

>>> pos_prob = decision_tree_best.predict_proba(X_test)[:, 1] >>>
from sklearn.metrics import roc_auc_score

>>> print('The ROC AUC on testing set is:

{0:.3f}'.format(roc_auc_score(y_test, pos_prob)

```

The ROC AUC on testing set is: 0.692

The AUC we can achieve with the optimal decision tree model is 0.69. It does not seem perfect, but click-through involves many intricate human factors and its prediction is a very difficult problem.

Looking back, a decision tree is a sequence of greedy searches for the best splitting point at each step based on the training dataset. However, this tends to cause overfitting as it is likely that the optimal points only work for the training samples. Fortunately, random forest is the technique to correct this, and it provides a better-performing tree model.

## Random Forest - Feature Bagging Of Decision Tree

The ensemble technique, bagging (which stands for bootstrap aggregating), which we briefly mentioned in the first chapter, can effectively overcome overfitting. To recap, different sets of training samples are randomly drawn with replacement from the original training data; each set is used to train an individual classification model. Results of these separate models are then combined together via majority vote to make the final decision.

Tree bagging, as previously described, reduces the high variance that a decision tree model suffers from and hence in general performs better than a single tree. However, in some cases where one or more features are strong indicators, individual trees are constructed largely based on these features and as a result become highly correlated. Aggregating multiple correlated trees will not make much difference. To force each tree to be uncorrelated, random forest only considers a random subset of the features when searching for the best splitting point at each node. Individual trees are now trained based on different sequential sets of features, which guarantees more diversity and better performance. Random forest is a variant tree bagging model with additional feature-based bagging.

To deploy random forest to our click-through prediction project, we will use the package from scikit-learn. Similar to the way we previously implemented decision tree, we only tweak the `max_depth` parameter:

```
>>> from sklearn.ensemble import RandomForestClassifier >>>
random_forest =
RandomForestClassifier(n_estimators=100, criterion='gini',
min_samples_split=30, n_jobs= >>> grid_search =
GridSearchCV(random_forest,
parameters, n_jobs=-1, cv=3, scoring='roc_auc')
>>> grid_search.fit(X_train, y_train)
>>> print(grid_search.best_params_)
{'max_depth': None}
```



Use the model with the optimal parameter None for max\_depth (nodes are expanded until other stopping criteria are met) to predict unseen cases:

```
>>> random_forest_best = grid_search.best_estimator_  
>>> pos_prob = random_forest_best.predict_proba(X_test)[:, 1]  
>>> print('The ROC AUC on testing set is:  
      {0:.3f}'.format(roc_auc_score(y_test, pos_prob))) The ROC AUC  
on testing set is: 0.724
```

It turns out that the random forest model gives a lift in the performance.

Although for demonstration, we only played with the max\_depth parameter, there are another three important parameters that we can tune to improve the performance of a random forest model:

- max\_features: The number of features to consider at each best splitting point search. Typically, for an m-dimensional dataset, (rounded) is a recommended value for max\_features. This can be specified as max\_features="sqrt" in scikit-learn. Other options include "log2", 20% of the original features to 50%.
- n\_estimators: The number of trees considered for majority voting. Generally speaking, the more the number trees, the better is the performance, but it takes more computation time. It is usually set as 100, 200, 500, and so on.
- min\_samples\_split: The minimal number of samples required for further split at a node. Too small a value tends to cause overfitting, while a large one is likely to introduce under fitting. 10, 30, and 50 might be good options to start with.

## Best Practices to Follow

After working on multiple projects covering important machine learning concepts, techniques, and widely used algorithms, we have gathered a broad picture of the machine learning ecosystem, and solid experience in tackling practical problems using machine learning algorithms and Python. However, there will be issues once we start working on projects from scratch in the real world. This chapter aims to get us ready for it with best practices to follow throughout the entire machine learning solution workflow.



## Best Practices in the Data Preparation Stage

Apparently, no machine learning system can be built without data. Data collection should be our first focus.

### — Completely understand the project goal

Before starting to collect data, we should make sure that the goal of the project, the business problem, is completely understood. As it will

guide us to what data sources to look into, and where sufficient domain knowledge and expertise is also required. For example, in the previous chapter, our goal was to predict future prices of the DJIA index, so we collected data of its past performance, instead of past performance of a European stock; in Chapter 5, Click-Through Prediction with Tree-Based Algorithms and Chapter 6, ClickThrough Prediction with Logistic Regression, the business problem was to optimize advertising targeting efficiency measured in a click-through rate, so we collected click stream data of who clicked or did not click on what ad in what page, instead of merely what ads were displayed on what page.

### **— Collect all fields that are relevant**

With a goal to achieve in mind, we have narrowed down potential data sources to investigate. Now the question becomes: Is it necessary to collect data of all fields available in a data source, or is a subset of attributes enough? It would be perfect if we could know in advance which attributes are key indicators or key predictive factors. However, it is very difficult to ensure that the attributes hand-picked by a domain expert will yield the best prediction results. Hence, for each data source, it is recommended to collect all fields that are related to the project, especially in cases where recollecting the data is time consuming, or even impossible. For example, in the stock price prediction example, we collected data of all fields including Open, High, Low, and Volume even though initially we were uncertain of how useful High and Low are. Retrieving the stock data is quick and easy with the API though.

In another example, if we ever want to collect data ourselves by scraping online articles for news topic classification, we should store as much information as possible. Otherwise, if a piece of information is not collected but is later found to provide value, such as hyperlinks in an article, the article might be already removed from the web page; if it still exists, rescraping those pages can be costly. After collecting the datasets that we think are useful, we need to assure the data quality by inspecting its consistency and completeness.

### **— Maintain consistency of field values**

In a dataset that exists or we collect from scratch, often we see values representing the same meaning. For example, there are "American", "US", and

"U.S.A" in the Country field, and "male" and "M" in the "Gender" field. It is necessary to unify values in a field. For example, we can only keep "M" and

"F" in the "Gender" field and replace other alternatives. Otherwise, it will mess up the algorithms in later stages as different feature values will be treated differently even if they have the same meaning. It is also a great practice to keep track of what values are mapped to the default value of a field.

In addition, the format of values in the same field should also be consistent. For instance, in the "Age" field, there are true age values such as 21, 35, and mistaken year values such as 1990, 1978; the "Rating" field, both cardinal numbers and English numerals are found, such as 1, 2, 3, and "one", "two", "three". Transformation and reformatting should be conducted in order to ensure data consistency.

### **— Deal with missing data**

Due to various reasons, datasets in the real world are rarely completely clean and often contain missing or corrupt values. They are usually presented as blanks, "Null", "-1", "999999", "unknown", or any placeholder. Samples with missing data not only provide incomplete predictive information, but also might confuse the machine learning model as it cannot tell whether -1 or "unknown" holds a meaning. It is significant to pinpoint and deal with missing data in order to avoid jeopardizing the performance of models in later stages.

Here are three basic strategies that we can use to tackle the missing data issue:

- Discarding samples containing any missing value
- Discarding fields containing missing values in any sample

These two strategies are simple to implement, however, at the expense of lost data, especially when the original dataset is not large enough. The third strategy does not abandon any data, but tries to fill in the blanks:

- Inferring the missing values based on the known part from the attribute. The process is called missing data imputation. Typical imputation methods include replacing missing values with the mean or the median value of the field across all samples, or the most frequent value for categorical data.

Let's look at how each strategy is applied in an example where we have a dataset (age, income) consisting of six samples (30, 100), (20, 50),

(35, unknown), (25, 80), (30, 70), and (40, 60). If we process this dataset using the first strategy, it becomes (30, 100), (20, 50), (25, 80), (30, 70), and (40, 60). If we employ the second strategy, the dataset becomes (30), (20), (35), (25), (30), and (40) where only the first field remains. If we decide to complete the unknown value instead of skipping it, the sample (35, unknown) can be transformed into (35, 72) with the mean of the rest values in the second field, or (35, 70) with the median value in the second field.

In scikit-learn, the Imputer class provides a nicely written imputation transformer. We will herein use it for the preceding small example:

```
>>> import numpy as np
>>> from sklearn.preprocessing import Imputer
>>> # Represent the unknown value by np.nan in numpy
>>> data_origin = [[30, 100],
...                 [20, 50],
...                 [35, np.nan],
...                 [25, 80],
```

```
...          [30, 70],  
...          [40, 60]]
```

Initialize the imputation transformer with the mean value and obtain such information from the original data:

```
>>> # Imputation with the mean value
```

```
>>> imp_mean = Imputer(missing_values='NaN', strategy='mean')
```

```
>>> imp_mean.fit(data_origin) Complete the missing value:
```

```
>>> data_mean_imp = imp_mean.transform(data_origin)
```

```
>>> print(data_mean_imp)
```

```
[[ 30. 100.]
```

```
 [ 20.  50.]
```

```
 [ 35.  72.]
```

```
 [ 25.  80.]
```

```
 [ 30.  70.]
```

```
 [ 40.  60.]]
```

Similarly, initialize the imputation transformer with the median value:

```
>>> # Imputation with the median value
```

```
>>> imp_median = Imputer(missing_values='NaN', strategy='median')
```

```
>>> imp_median.fit(data_origin)
```

```
>>> data_median_imp = imp_median.transform(data_origin)
```

```
>>> print(data_median_imp)
```

```
[[ 30. 100.]
```

```
 [ 20.  50.]
```

```
 [ 35.  70.]
```

```
[ 25.  80.]
```

```
[ 30.  70.]
```

```
[ 40.  60.]]
```

When new samples come in, missing values (in any attribute) can be imputed using the trained transformer, for example, with the mean value:

```
>>> new = [[20, np.nan],
```

```
...      [30, np.nan],
```

```
...      [np.nan, 70],
```

```
...      [np.nan, np.nan]]
```

```
>>> new_mean_imp = imp_mean.transform(new)
```

```
>>> print(new_mean_imp)
```

```
[[ 20.  72.]
```

```
[ 30.  72.]
```

```
[ 30.  70.]
```

```
[ 30.  72.]]
```

Note that 30 in the age field is the mean of those six age values in the original dataset. Now that we have seen how imputation works and its implementation, let's see how the strategy of imputing missing values and discarding missing data affects the prediction results through the following example. First, we load the diabetes dataset and simulate a corrupted dataset with missing values:

```
>>> from sklearn import datasets
```

```
>>> dataset = datasets.load_diabetes()
```

```
>>> X_full, y = dataset.data, dataset.target
```

```
>>> # Simulate a corrupted data set by adding 25% missing value
```

```

>>> m, n = X_full.shape
>>> m_missing = int(m * 0.25)
>>> print(m, m_missing)
442 110
>>> # Randomly select m_missing samples
>>> np.random.seed(42)
>>> missing_samples = np.array([True] * m_missing +
                                [False] * (m - m_missing))
>>> np.random.shuffle(missing_samples)
>>> # For each missing sample, randomly select 1 out of n features
missing_features = np.random.randint(low=0,
                                     high=n,
                                     size=m_missing)
>>> # Represent missing values by nan
>>> X_missing = X_full.copy()
>>> X_missing[np.where(missing_samples)[0], missing_features] =

```

Then we deal with this corrupted dataset by discarding samples containing a missing value:

```

>>> X_rm_missing = X_missing[~missing_samples, :]
>>> y_rm_missing = y[~missing_samples]

```

We then measure the effects of using this strategy by estimating the averaged regression score, the

with a regression forest model in a cross-validation manner:

```

>>> # Estimate R^2 on the data set with missing samples removed
>>> from sklearn.ensemble import RandomForestRegressor

```



```
>>> from sklearn.model_selection import cross_val_score

>>> regressor = RandomForestRegressor(random_state=42,
max_depth=10, n_estimators=100)
>>> score_rm_missing = cross_val_score(regressor, X_rm_missing, y_rm_missing).mean
>>> print('Score with the data set with missing samples removed
{0:.2f}'.format(score_rm_missi
```

Score with the data set with missing samples removed: 0.39

Now we approach the corrupted dataset differently by imputing missing values with the mean:

```
>>> imp_mean = Imputer(missing_values='NaN', strategy='mean')
>>> X_mean_imp = imp_mean.fit_transform(X_missing)
```

And similarly we measure the effects of using this strategy by estimating the averaged

:

```
>>> # Estimate R^2 on the data set with missing samples removed
>>> regressor = RandomForestRegressor(random_state=42, max_depth=10, n_estimators=100)
>>> score_mean_imp = cross_val_score(regressor, X_mean_imp, y).mean
>>> print('Score with the data set with missing values replaced by mean: {0:.2f}'.format(score_mean_imp))
Score with the data set with missing values replaced by mean: 0
```

Imputation strategy works better than discarding in this case. So how far is the imputed dataset from the original full one? We can check it again by estimating the averaged regression score on the original dataset:

```
>>> # Estimate R^2 on the full data set
>>> regressor = RandomForestRegressor(random_state=42, max_depth=10, n_estimators=100)
```

```
ax_depth=10,      n_estimators=50      >>>      score_full      =  
cross_val_score(regressor, X_full, y).mean()  
  
>>> print('Score with the full data set:  
        {0:.2f}'.format(score_full))
```

Score with the full data set: 0.44

It turns out that little information is comprised in the completed dataset. However, there is no guarantee that the imputation strategy always works better and sometimes dropping samples with missing values can be more effective. Hence, it is a great practice to compare the performances of different strategies via cross-validation as we have practiced previously.

## **Best Practices In The Training Sets Generation Stage**

With well-prepared data, it is safe to move on with the training sets generation stage. Typical tasks in this stage can be summarized into two major categories, data preprocessing and feature engineering.

Data preprocessing usually involves categorical feature encoding, feature scaling, feature selection, and dimensionality reduction.

### **— Determine categorical features with numerical values**

In general, categorical features are easy to spot, as they convey qualitative information, such as risk level, occupation, and interests. However, it gets tricky if the feature takes on a discrete and countable (limited) number of numerical values, for instance, 1 to 12 representing months of the year, and 1 and 0 indicating true and false. The key to identifying whether such a feature is categorical or numerical is whether it provides mathematical implication: if so, it is a numerical feature, such as product rating from 1 to 5; otherwise, categorical, such as the month or day of the week.

### **— Decide on whether or not to encode categorical features**

If a feature is considered categorical, we need to decide on whether or not to encode it. It depends on what prediction algorithm(s) we will use in a later stage. Naive Bayes and tree-based algorithms can directly work with categorical features, while other algorithms in general cannot, in which case encoding is essential.

As the output of the feature generation stage is the input of the algorithm training stage, steps taken in the feature generation stage should be compatible with the prediction algorithm. Therefore, we should look at two stages, feature generation and prediction algorithm training as a whole, instead of two isolated components. The next two practical tips also suggest this point.

- **Decide on whether or not to reduce dimensionality and if so how**

Dimensionality reduction has advantages similar to feature selection:

- Reducing training time of prediction models, as redundant or correlated features are merged into new ones
- Reducing overfitting for the same reason
- Likely improving performance as prediction models will learn from data with less redundant or correlated features

Again, it is not certain that dimensionality reduction will yield better prediction results. In order to examine its effects integrating dimensionality reduction in the model training stage is recommended. Reusing the preceding handwritten digits example, we measure the effects of PCA-based dimensionality reduction, where we keep a different number of top components to construct a new dataset, and estimate the accuracy on each dataset:

```
>>> from sklearn.decomposition import PCA

>>> # Keep different number of top components >>> N = [10, 15, 25,
35, 45]

>>> for n in N:

...     pca = PCA(n_components=n)
```

```

... X_n_kept = pca.fit_transform(X)
... # Estimate accuracy on the data set with top n componen
... classifier = SVC(gamma=0.005) ... score_n_components =
        cross_val_score(classifier, X_n_kept, y).mean
... print('Score with the data set of top {0} components:
        {1:.2f}'.format(n, score_n_componen

```

Score with the data set of top 10 components: 0.95

Score with the data set of top 15 components: 0.95

Score with the data set of top 25 components: 0.91

Score with the data set of top 35 components: 0.89

Score with the data set of top 45 components: 0.88

- **Decide on whether or not to scale features**

SGD-based linear regression and SVR models require features to be standardized by removing the mean and rescaling to unit variance. So when is feature scaling needed and when is it not?

In general, naive Bayes and tree-based algorithms are not sensitive to features at different scales, as they look at each feature independently. Logistic or linear regression normally is not affected by the scales of input features, with one exception, when the weights are optimized with stochastic gradient descent.

In most cases, an algorithm that involves any form of distance (separation in spaces) of samples in learning factors requires scaled/standardized input features, such as SVC and SVR. Feature scaling is also a must for any algorithm using SGD for optimization. We have so far covered tips regarding data preprocessing and we will now discuss best practices of feature engineering as another major aspect of training sets generation. We will do so from two perspectives:

- **Perform feature engineering with domain expertise**

Luckily enough, if we possess sufficient domain knowledge, we can apply it in creating domain-specific features; we utilize our business experience and insights to identify what in the data and formulate what correlates to the prediction target from the data. For example, in Chapter 9, Stock Prices Prediction with Regression Algorithms, we designed and constructed feature sets for stock prices prediction based on factors investors usually look at when making investment decisions.

While particular domain knowledge is required, sometimes we can still apply some general tips in this category. For example, in fields related to customer analytics, such as market and advertising, time of the day, day of the week, month are usually important signals. Given a data point with the value

2017/02/05 in the date column and 14:34:21 in the time column, we can create new features including afternoon, Sunday, and February. In retail, information over a period of time is usually aggregated to provide better insights. The number of times a customer visits a store for the past three months, average number of products purchased weekly for the previous year, for instance, can be good predictive indicators for customer behavior prediction.

- **Perform feature engineering without domain expertise**

If unfortunately, we have very little domain knowledge, how can we generate features? Don't panic. There are several generic approaches:

- **Binarization:** a process of converting a numerical feature to a binary one with a preset threshold. For example, in spam email detection, for the feature (or term) prize, we can generate a new feature whether prize occurs: any term frequency value greater than 1 becomes 1, otherwise 0. Feature number of visits per week can be used to produce a new feature is frequent

visitor by judging whether the value is greater than or equal to 3. We implement such binarization as follows using scikit-learn:

```
>>> from sklearn.preprocessing import Binarizer
```

```
>>> X = [[4], [1], [3], [0]]
```

```
>>> binarizer = Binarizer(threshold=2.9)
```

```
>>> X_new = binarizer.fit_transform(X)
```

```
>>> print(X_new)
```

```
[[1]
```

```
[0]
```

```
[1]
```

```
[0]]
```

- Discretization: a process of converting a numerical feature to a categorical feature with limited possible values. Binarization can be viewed as a special case of discretization. For example, we can generate an age group feature from age: 18-24 for age from 18 to 24, 25-34 for age from 25 to 34, 34-54 and 55+.
- Interaction: includes sum, multiplication, or any operations of two numerical features, joint condition check of two categorical features. For example, number of visits per week and number of products purchased per week can be used to generate number of products purchased per visit feature; interest and occupation, such as sports and engineer, can form occupation and interest, such as engineer interested in sports.
- Polynomial transformation: a process of generating polynomial and interaction features. For two features

and

*b*

, the two degree of polynomial features generated are

$a^2$

,

$ab$

and

$b^2$

. In scikit-learn, we can use the PolynomialFeatures class to perform polynomial transformation:

```
>>> from sklearn.preprocessing import
PolynomialFeature

>>> X = [[2, 4],
... [1, 3],
... [3, 2],
... [0, 3]]

>>> poly = PolynomialFeatures(degree=2)
>>> X_new = poly.fit_transform(X)
>>> print(X_new)

[[ 1.  2.  4.  4.  8. 16. ]
 [ 1.  1.  3.  1.  3.  9.]
 [ 1.  3.  2.  9.  6.  4.]
 [ 1.  0.  3.  0.  0.  9.]]
```

Note that the resulting new features consist of 1 (bias, intercept),

$a$

,

$b$

,

$a^2$

,

$ab,$

and

$b^2$

#### — **Document how each feature is generated**

We've covered rules of feature engineering with domain knowledge and in general, there is one more thing worth noting: document how each feature is generated. It sounds trivial, but often we just forget how a feature is obtained or created. We usually need to go back to this stage after some fail trials in the model training stage and attempt to create more features with the hope of performance improvement. We have to be clear of what and how features are generated, in order to remove those that do not quite work out and to add new ones with potential.



# Best Practices In The Model Training, Evaluation, and Selection Stage

Given a machine learning problem, the first question many people ask is usually: what is the best classification/regression algorithm to solve it? However, there is no one-size-fits-all solution or free lunch. No one could know which algorithm will work the best before trying multiple methods and fine-tuning the optimal one. We will be looking into best practices around this in the following sections.

## – Choose the right algorithm(s) to start with

Due to the fact that there are several parameters to tune for an algorithm, exhausting all algorithms and fine-tuning each one can be extremely timeconsuming and computationally expensive. We should instead short-list one to three algorithms to start with following the general guidelines in the following list (note we herein focus on classification, but the theory transcends in regression and there is usually a counterpart algorithm in regression).

There are several things that we need to be clear about before short-listing potential algorithms:

- Size of the training dataset
- Dimensionality of the dataset
- Whether the data is linearly separable
- Whether features are independent
- Tolerance and tradeoff of bias and variance
- Whether online learning is required

## Naive Bayes

It is a very simple algorithm. For a relatively small training dataset, if features are independent, naive Bayes will usually perform well. For a large dataset, naive Bayes will still work well as feature independence can be assumed in this case regardless of the truth. Training of naive Bayes is usually faster than any other algorithms due to its computational simplicity. However, this may lead to high bias (low variance though).

## **Logistic regression**

It is probably the most widely used classification algorithm, and the first algorithm a machine learning practitioner usually tries given a classification problem. It performs well when data is linearly separable or approximately linearly separable. Even if it is not linearly separable, we can if possible, convert the linearly non-separable features into separable ones and apply logistic regression afterwards (see the following example). Also logistic regression is extremely scalable to large datasets with SGD optimization, which makes it efficient in solving big data problems. Plus, it makes online learning feasible.

Although logistic regression is a low bias, high variance algorithm, we overcome the potential overfitting by adding L1, L2, or a mix of two regularizations.

## **SVM**

It is versatile to adapt to the linear separability of data. For a separable dataset, SVM with linear kernel performs comparably to logistic regression. Beyond this, SVM also works well for a non-separable one, if equipped with a non-linear kernel, such as RBF. For a high-dimensional dataset, the performance of logistic regression is usually compromised, while SVM still performs well. A good example could be news classification where the feature dimension is tens of thousands. In general, very high accuracy can be achieved by SVM with the right kernel and parameters. However, this might be at the expense of intense computation and high memory consumption.

## **Random forest (or decision tree)**

Linear separability of data does not matter to the algorithm. And it works directly with categorical features without encoding, which provides great ease of use. Also, the trained model is very easy to interpret and explain to nonmachine learning practitioners, which cannot be achieved with most other algorithms. Additionally, random forest boosts decision tree, which might lead to overfitting by

assembling a collection of separate trees. Its performance is comparable to SVM, while fine-tuning a random forest model is less difficult compared to SVM and neural networks.

## **Neural networks**

It is extremely powerful, especially with the development of deep learning. However, finding the right topology (layers, nodes, activation functions, and so on) is not easy, not to mention the time-consuming model training and tuning. Hence, it is not recommended as an algorithm to start with.

### **— Reduce overfitting**

We've touched on ways to avoid overfitting when discussing the pros and cons of algorithms in the last practice. We will now formally summarize them:

- Cross-validation, a good habit we have built on throughout the chapters in this book.
- Regularization.
- Simplification if possible. The more complex the model is, the higher the chance of overfitting is. Complex models include a tree or forest with excessive depth, a linear regression with high degree polynomial transformation, and SVM with a complicated kernel.
- Ensemble learning, combining a collection of weak models to form a stronger one.

### **— Diagnose overfitting and underfitting**

So how can we tell whether a model suffers from overfitting, or the other extreme, underfitting? Learning curve is usually used to evaluate bias and variance of a model. Learning curve is a graph that compares the crossvalidated training and testing scores over a variety of training samples.

For a model that fits well on the training samples, the performance of training samples should be above what is desired. Ideally, as the number of training samples increases, the model performance on

testing samples improves; eventually the performance on testing samples becomes close to that on training samples.

When the performance on testing samples converges at a value far from the performance on training samples, overfitting can be concluded. In this case, the model fails to generalize to instances that are not seen. For a model that does not even fit well on the training samples, underfitting is easily spotted: both performances on training and testing samples are below what is desired in the learning curve.

## **Best Practices In The Deployment And Monitoring Stage**

After all the processes in the former three stages, we now have a well established data preprocessing pipeline and a correctly trained prediction model. The last stage of a machine learning system involves saving those resulting models from previous stages and deploying them on new data, as well as monitoring the performance, updating the prediction models regularly.

### **\_ Save, load, and reuse models**

When the machine learning is deployed, new data should go through the same data preprocessing procedures (scaling, feature engineering, feature selection, dimensionality reduction, and so on) as in previous stages. The preprocessed data is then fed in the trained model. We simply cannot rerun the entire process and retrain the model every time new data comes in. Instead, we should save the established preprocessing models and trained prediction models after the corresponding stages complete. In deployment mode, these models are loaded in advance, and they are used to produce prediction results of the new data.

We illustrate it via the diabetes example where we standardize the data and employ an SVR model:

```
>>> dataset = datasets.load_diabetes()
```

```
>>> X, y = dataset.data, dataset.target
```

```
>>> num_new = 30 # the last 30 samples as new data set
```

```
>>> X_train = X[:-num_new, :]
```

```
>>> y_train = y[:-num_new]
```

```
>>> X_new = X[-num_new:., :] >>> y_new = y[-num_new:]
```

Preprocessing the training data with scaling:

```
>>> from sklearn.preprocessing import StandardScaler
```

```
>>> scaler = StandardScaler()
```

```
>>> scaler.fit(X_train)
```

Now save the established standardize, the scaler object with pickle:

```
>>> import pickle
```

```
>>> pickle.dump(scaler, open("scaler.p", "wb" ))
```

This generates the scaler.p file. Move on with training a SVR model on the scaled data:

```
>>> X_scaled_train = scaler.transform(X_train)
```

```
>>> from sklearn.svm import SVR
```

```
>>> regressor = SVR(C=20)
```

```
>>> regressor.fit(X_scaled_train, y_train)
```

Save the trained regressor, the regressor object with pickle:

```
>>> pickle.dump(regressor, open("regressor.p", "wb"))
```

This generates the regressor.p file. In the deployment stage, we first load in the saved standardizer and regressor from the two preceding files:

```
>>> my_scaler = pickle.load(open("scaler.p", "rb" )) >>>  
my_regressor = pickle.load(open("regressor.p", "rb"))
```

Then preprocess the new data using the standardizer and make a prediction with the regressor just loaded:

```
>>> X_scaled_new = my_scaler.transform(X_new) >>>
predictions = my_regressor.predict(X_scaled_new) Best
practice 17 - monitor model performance
```

The machine learning system is now up and running. To make sure everything is on the right track, we need to conduct a performance check on a regular basis. To do so, besides making a prediction in real time, we should record the ground truth at the same time.

Continue the diabetes example with a performance check:

```
>>> from sklearn.metrics import r2_score
>>> print('Health check on the model, R^2:
{0:.3f}'.format(r2_score(y_new, predictions))) Health check
on the model, R^2: 0.613
```

We should log the performance and set an alert for a decayed performance.

## **\_ Update Models Regularly**

If the performance is getting worse, chances are the pattern of data has changed. We can work around this by updating the model. Depending on whether online learning is feasible or not with the model, the model can be modernized with the new set of data (online updating) or retrained completely with the most recent data.

## Conclusion

We hope we have convinced you of the usefulness of machine learning in a wide variety of applications, and how easily machine learning can be implemented in practice.

Learning python machine learning is not a tough task even for beginners. So, take the leap and master the ***Python Machine Learning*** .

Keep digging into the data, and don't lose sight of the larger picture.

# Python Data Science

Andrew Lee

## **Understanding focus on data analysis.**

First, we will start by discussing some of The toolbox of all any data scientist, as for any kind of programmer, is an essential ingredient for success and enhanced performance. Choosing the right tools can save a lot of time, allowing us to





Many people use just one programming language in their entire life, which is usually the first and only one that they learn. Recall, however, that also tedious tasks must be performed properly, if success is to follow.

Single language is that many basic tools simply will not be available at it, and eventually you will have into either reimplement them or create a 'bridge' so you can use some other language for a specific task. To get this book, we have selected Python as the programming language, as it offers a great level of versatility to your data science programmer.

However it also has excellent properties for newbie programmers, making it ideal for people who have never programmed before. Some of that the most remarkable of those properties are easy-to-read code, suppression of non-mandatory delimiters, dynamic typing, and dynamic memory usage. Python is an interpreted language, so that the code is executed immediately in the Python console without needing the compilation step into machine language. Currently, Python is one of that the most One of all its main characteristics that makes it so flexible is that it can be seen as

a multiparadigm language. This is especially useful for people who already know how to program with other languages, as they can rapidly start programming with Python in that the same way.

Back in this book, we have decided to focus on

That the Python language because, as explained earlier, it is a mature programming language, easy for the newbies, and can be used as a specific platform for data scientists, thanks to its large ecosystem of scientific libraries and its vibrant community.

The Python community is one of all the most Active programming communities, with a enormous number of developed toolboxes. scientific computing with Python. NumPy provides, among other things, support such as multidimensional arrays with basic operations and useful linear algebra functions.

Many toolboxes use the NumPy array representations as an efficient basic data structure. Meanwhile, SciPy provides a collection of numerical algorithms and domain-specific toolboxes, including signal processing, optimization, statistics, and much more. Another core toolbox at SciPy is the plotting library Matplotlib. This toolbox has many tools for data visualization. SCIKIT-Learn: Machine Learning in Python Scikit-learn is a Machine Learning library built from NumPy, SciPy, and Matplotlib. Scikit-learn offers simple and

efficient tools for common tasks in data analysis, such as classification, regression, clustering, dimensionality reduction, model selection, and preprocessing.

## PANDAS: Python Data Analysis Library

Pandas provides high-performance data structures and data analysis tools. The key feature of Pandas is a fast and efficient DataFrame object for data manipulation with integrated indexing. The DataFrame structure can be seen as a spreadsheet, which offers very flexible ways of working with it. You can easily transform any dataset in the way you want, by reshaping it and adding or removing columns or rows. It also provides high-performance functions for

aggregating, blending, and joining datasets. Pandas also has tools for importing and exporting data from different formats:

comma-separated value (CSV), text files, Microsoft Excel, SQL databases, and the fast HDF5 format. Back in many situations, the data you have in such formats will not be complete or totally structured. For such cases, Pandas offers handling of missing data and intelligent data alignment. Furthermore, Pandas

provides a convenient Matplotlib interface.

## Data Science Ecosystem Installation

Before we can get started on solving our own data-oriented problems, we will need to set up our programming environment.

The first question we want to answer concerns that the Python language itself. There are currently two different versions of Python: Python 2.X and Python 3.X. The differences between the versions are important, so there is no compatibility between the codes, i.e., code written in Python 2.X does not work in Python 3.X and vice versa. Python 3.X was introduced in late 2008; by then, a lot of code and many toolboxes had already been deployed using Python 2.X (Python 2.0 was initially introduced in 2000). Therefore, much of the scientific community did not change to Python 3.0 immediately, and they were stuck with Python 2.7. By today, almost all libraries have been ported into Python 3.0; however Python 2.7 is still maintained, so possibly version can be chosen.

However, those who already have a large amount of code in 2.X rarely change to Python

3.X. In our examples through this book, we will use Python 2.7. Once we have chosen one of the Python Versions, the next item to decide is whether we want to install the data scientist Python ecosystem by individual tool-boxes, or to perform a package installation with all the needed toolboxes (and a lot more). For newbies, the second option is recommended. In case the first option is chosen, then it is only

necessary to install all the said toolboxes at the previous section, in exactly that order. However, if a bundle installation is chosen, the Anaconda Python distribution is a good option. The Anaconda distribution provides integration of all the Python toolboxes and applications needed for data scientists into a single directory, without mixing with other Python toolboxes installed on the machine. It contains, of course, the core toolboxes and applications such as NumPy, Pandas, SciPy, Matplotlib, Scikit-learn, IPython, Spyder, etc., however also more specific tools for other related tasks such as data visualization, code optimization, and big data processing.

## Integrated Development Environments (IDE)

For any programmer, and by extension, for any data scientist, the integrated development environment (IDE) is an essential tool. IDEs are designed to maximize programmer productivity. Thus, over the years, this software has evolved in order to make coding tasks less complicated.

Choosing the right IDE for each person is crucial, however unfortunately, there is no "one-size-fits-all" programming environment. The best solution is to try the most popular IDEs among the community and keep that the ones that fit better in each case. Back in general, the basic pieces of any IDE are Three: the editor, the compiler (or interpreter), and the debugger. Some IDEs can be used in multiple programming languages, provided by language-specific plugins, such as Netbeans<sup>7</sup> or Eclipse<sup>8</sup>. Others are specific to one language, or even to a specific programming task. In the case of Python, there are a large number of specific IDEs, both commercial (PyCharm, WingIDE<sup>10</sup>) and open-source. The open-source community helps IDEs spring up; anyone can customize their own environment and share it with the rest of the community. For example, Spyder<sup>11</sup> (Scientific Python Development EnviRonment) is an IDE customized with the task of the data scientist in mind. Together with the advent of all web applications, a new Starting in the academic and e-learning communities, web-based IDEs were developed considering the way not just your code, however also all of your environment and executions, can be stored in a server.

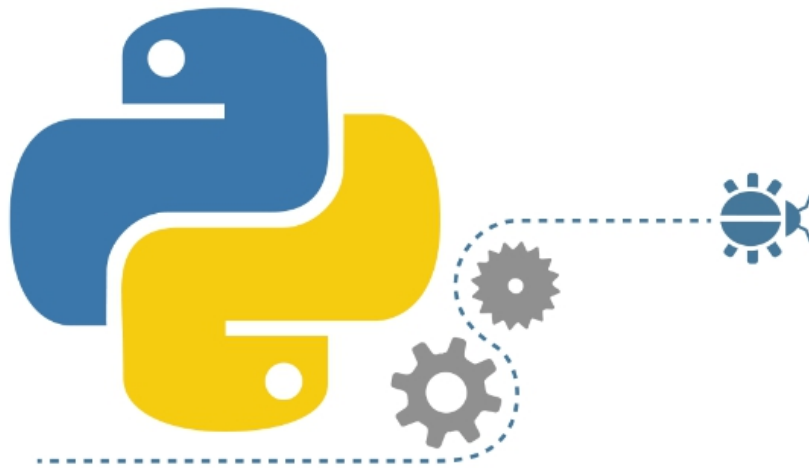
Back in SageMath, a server can be set up in a center, such as a university or school, and students can work on their homework either in that the classroom or at home, starting from exactly the same point they left off. Nowadays, such sessions are called 'notebooks' And they are not only used in classrooms, however also used to show results in presentations or on business dashboards. The recent spread of such notebooks is mainly due to IPython. Since December 2011, IPython has been issued as a browser version of its interactive console, called IPython Notebook, which shows Python execution results very clearly and concisely by means of cells. Cells can contain content other than code. For example, markdown (a wiki text language) cells can be added to introduce algorithms. It is also possible into insert Matplotlib graphics to illustrate examples or even web pages. Recently, some scientific journals have started to accept notebooks in order to show experimental results, complete using their code and data sources. Back in this way, experiments can become completely replicabledown into this last detail. Since the project has grown so much, IPython Laptop has been separated from IPython software and has become a part of a larger project: Jupyter<sup>12</sup>. Jupyter (for Julia, Python and R) aims to reuse the same WIDE for all these interpreted languages, and not just Python. All old IPython notebooks are automatically imported to the new version if they are opened with the Jupyter platform; however once they are converted to the new version, they cannot be used again in old IPython notebook versions. In this novel, all the examples shown use Jupyter notebook style.

## **Get Started Using Python to Get Data Scientists**

many practical examples. Back in this chapter, we will use a very basic example to

help you start a data science ecosystem from scratch. To execute our examples,

we will use Jupyter notebook, although any other console or IDE can be used.



Can start by launching that the Jupyter notebook platform. In case we chose the bundle installation, we can In case we use the command line, the root directory is the same directory where we launched the Jupyter notebook. Otherwise, if we use that the Anaconda launcher, the root directory is the current user directory. Now, to start a new notebook, we only need into press the (New Notebooks Python two ) button at the top on the right of the home page. Untitled. First of all all, we are going to change the name of that the notebook to something more appropriate. To perform this, just click on the notebook name and rename it: DataScience-GetStartedExample.

```
import matplotlib. pyplot as plt  
import matplotlib. pyplot as plt
```

Since this was

the first cell executed, the number shown will be 1. If the process of importing the libraries is correct, no output cell is produced.

Reading,

## Selecting & Filtering Your Data

### Reading

Let's start reading the data we downloaded.

First of all, we have to create a new notebook called Open Government Data Analysis

and open it. Then, after ensuring that the `educ_figdp_1_Data. Csv` file is stored

in the same directory as that our notebook directory, we will write the following

code to read and show that the content:

```
edu = pd.read_csv('files/ch02/ educ_figdp_1_Data. csv',na_values =  
':',usecols = ["TIME","GEO","Value"])edu
```

The way to read CSV (or any other separated

value, providing the separator character) files in Pandas is by using the

`read_csv` method. Besides the name of the file, we add the `na_values` key

argument to this method along with the character that represents "non available

data" in the file. Normally, CSV files have a header with the names of the

columns. In case this is the case, we can use the `usecols` parameter to select which

columns in the file will be used.

Back in this case, the DataFrame resulting from reading our data is stored in `edu`. The output of that the execution shows that the `edu` DataFrame size is 384 rows × 3 columns. Since the DataFrame is too large to be fully displayed, three dots appear at the middle of each row.

Beside this, Pandas also has functions for reading files with formats such as Excel, HDF5, tabulated files, or even the content from the clipboard (`read_excel()`, `read_hdf()`, `read_table()`, `read_clipboard()`). Whichever function we use, the result of reading a document is stored as a DataFrame structure.

To see how the data looks, we can use the `Head()` method, which shows just that the first five rows. In case we use a number as an argument, this will be the amount of rows that will be listed:

`edu.head()` Similarly, you can use the `tail()` method, which returns the last five rows by default.

`edu.tail()` In case we want to know the names of the columns or the names of the indexes, we can use the DataFrame attributes `columns` and `index` respectively. The names of the columns or indexes can be changed by



assigning a new list of that the same length into these attributes. The values of any

DataFrame can be retrieved as a Python array by attracting up its values attribute.

In case we just want quick statistical information on

all the numeric columns in a DataFrame, we can use the function `describe()`. This

result shows the count, the mean, the standard deviation, the minimum and

maximum, and the percentiles, by default, of the 25th, 50th, and 75th, for all

the values in each column or series.

## Selecting

If we want into select a subset of data from a

DataFrame, it is necessary to indicate this subset using square brackets (`[]`) after the DataFrame.

The subset can be specified in several ways. In case we want to select only one

column out of a DataFrame, we only want to put its name involving the square brackets. The result will

be a Series data structure, not a DataFrame, because only one column is retrieved.

`edu['Value']` In case we want into select a subset of rows from a

DataFrame, we can do so by indicating a range of rows separated by a colon (:-RRB-

inside the square

brackets. This is commonly known as a 'slice' of rows:

This instruction returns the slice of rows

from the 10th to the 13th position. Notice that the slice does not use the index

labels as references, but the position. Back in this case, the labels of that the rows

simply coincide with the position of the rows.

When we want to select a subset of columns and

rows, using the labels as our references instead of the positions, we can use

ix indexing:

This returns all the rows between the

indexes specified in the slice before the comma, with the columns specified as a

list after the comma. In this case, ix references that the index labels, which means

that ix does not reunite the 90th to 94th rows, but it returns all the rows

between the row labeled 90 and the row labeled 94; so if that the index '100' is

placed between the rows labeled as 90 and 94, this row would also be returned.

## Filtering

Another way to select a subset of data is

by applying Boolean indexing. This indexing is commonly known as a 'filter.'

For instance, if we want to filter those values less than or equal to 6.5, we can take action like

this:

`edu[edu['Value'] > 6.5]. tail()` Boolean indexing uses the result of a

Boolean operation over the data, returning a mask with True or False for each

row. The rows marked True in the mask will be selected. Back in the previous

example, the Boolean operation `edu['Value'] > 6.5` produces a Boolean mask.

When an component in the "Value" column is greater than 6.5, the corresponding

value at the mask is set to 'True,' otherwise it is set to 'False.' Then, when

this mask is applied as an index in `edu[edu['Value'] > 6.5]`, the result is a

filtered DataFrame containing only rows with values higher than 6.5. Of course course,

any of that the usual Boolean operators can be used for filtering: < (less

than), <= (less than or equal to), > (greater than), >= (greater than or equal to),

respectively (equal to),  
and `! ) = (not equal to)`.  
to).

## Filtering

### Missing Values

Pandas uses the special value NaN (not a number) to represent missing values. Back in Python, NaN is a special floating-point value returned by certain operations when one of their results ends at an undefined value. A subtle feature of NaN values is that two NaN are not ever equal. Because of this, the only safe way to tell whether a value is missing in a DataFrame is by using the `isnull()` function. Indeed, this function can be used into filter rows with missing values.

## Manipulating & Sorting Data

### Manipulating

Once

we know how to select the desired data, the next item we need to know is how

to manipulate data. One of the most straightforward things we can do is into operate with columns or rows using aggregation functions.

The result of all

these functions applied to a row or column is always a number. Meanwhile, if a

function is applied to a

Columns, then you can specify if the function should be applied to the rows for

each column (setting that the `axis=0` keyword on the invocation of the function), or if

it should be applied on the columns for each row (setting the `axis=1` keyword on

that the invocation of the function).

Note that these are functions specific into Pandas,

Back in Python, NaN values propagate through all operations, without raising an exception. Back in contrast, Pandas

operations exclude NaN values representing missing data. For example, the Pandas

`max` function excludes NaN values, thus they are interpreted as missing values;

whereas the standard Python max function will take the mathematical

interpretation of NaN and return it as the maximum:

edu. Besides these aggregation functions we can

Apply operations over all that the values in rows, columns, or a combination of

both. For example, we can apply any binary arithmetical operation (+, -, \*, /) to an whole row:

```
s.head()
```

DataFrame or Series just by setting its name as that the argument of the apply

method. For example, from the next code, we apply that the sqrt function from the NumPy

library to perform the square

root of each value in that the Value column.

If we want to design a specific function to

```
apply ( lambda
```

```
Id: p **two )
```

```
s.head()
```

This will

produce a new column from the DataFrame, displayed after all the others. You must

be aware that if a column with that the same

name already exists, its previous values will be overwritten. In the

following example, we assign the Series that results from dividing the Value

column by that the maximum value in the same column to a new column named 'ValueNorm.'

Back in Pandas, all the functions that

change the contents of a DataFrame, such as the drop function, will normally

return a copy of the altered data, instead of overwriting the DataFrame.

In case you do not want to keep the old values, you can set the keyword inplace to 'True'.

Authentic )

edu.head()

A new row at the bottom of that the DataFrame, we can use the Pandas 'append'

function. You must be sure to set that the 'ignore\_index' flag in the append method to 'True', otherwise the index 0 is given to this new row, which

will produce an error if it already exists:

edu

edu.tail()

Want to use the drop function again. Now, we have to set that the axis to 0, and

specify the index of the row we want to remove. Since we want to eliminate the

last row, we can use the max function over the indexes to determine which row it

is.

The drop() function is also used to remove

Missing values by applying it over that the result of that the isnull() function. This has

a similar effect to filtering the NaN values, as we explained above, however here

the difference is that a copy of the DataFrame without that the NaN values is returned, instead of a view.

To remove NaN values, instead of the

generic drop function, we can use the specific dropna() function. In case we want to

erase any row that contains an NaN value, we have to set the just how keyword into 'any'.

To restrict it to a subset of columns, we can specify it using the subset

keyword. As we can see below, the result will be the same as using the drop

function:

```
EduDrop = edu.dropna(exactly='any', subset =  
["Value"])
```

eduDrop. head() If, instead of removing the rows containing

NaN, we want to fill them with another value, then we can use the fillna()



method, specifying which value is to be used. Should we want to fill only some

specific columns, we have to set as that the argument to the fillna() function a

dictionary with the names of the columns as the key, and which character is to

be used for satisfying as the value.

## Sorting

Another important functionality we will

Need if inspecting our data is to sort by columns. We can sort a DataFrame

using any column, using the sort function. In case we want to see the first five

rows of data sorted in descending order (i.e., from the largest into the smallest

values) and using the Value column, then we just need to do this:

Note that that the inplace keyword means that

the DataFrame will be overwritten, and hence no new DataFrame is returned. In case instead of ascending = False we use ascending = True, the values are sorted in

ascending order (i.e., from the smallest into the largest values).

When we want to return to the original order,

we can sort by an index using the sort\_index function and specifying axis=0:

```
edu.sort_index( axis = 0, ascending = True ,  
inplace = True)
```

## edu.head()Grouping & Rearranging Data

### Grouping

Another very useful way to inspect data is to group it according to established criteria. For instance, in our example, it could be nice to group all the data by country, regardless of the year. Pandas has the groupby function that allows us to do exactly this. The value returned by this function is a special, grouped DataFrame. To have a proper DataFrame as a result, it is necessary to apply an aggregation function. Thus, this function will be applied to all that the values in the same group.

For example, in our case, if we want a DataFrame showing the mean of the values for each country over all the years, we can obtain it by grouping according to country, and using the mean function as the aggregation method for each group. The result would be a DataFrame with countries as indexes and the mean values as the column:

```
group = edu[["GEO",  
"Value"]].groupby('GEO').mean()
```

`group.head()`Rearranging

Up until today our indexes have been just a numeration of rows without much meaning. We can transform the arrangement of

our data, redistributing the indexes and columns for better manipulation of our

data, which normally leads into better performance. We can rearrange our data

using the `pivot_table` function. Here, we can specify which columns will be the

new indexes, the fresh values, and the new columns.

To get example, imagine that we want to transform

our DataFrame to a spreadsheet- like structure with the country names as the

index, while the columns will be the years starting from 2006, and the values

will be the previous Value column. To do this, first we want to filter out the

data, and then pivot it in this way:

```
filtered_data = edu[edu["TIME"]  
> 2005]
```

```
pivedu = pd. pivot_table( filtered_data , values  
= 'Value',
```

```
index = ['GEO'] ,columns = ['TIME'])pivedu.head()
```

Today we can use the new index to select

specific rows by label, using the ix operator:

```
pivedu.ix[['Spain','Portugal'],  
[2006,2011]]
```

Pivot also offers the option of providing

an argument `aggr_function` that allows us to perform an aggregation function

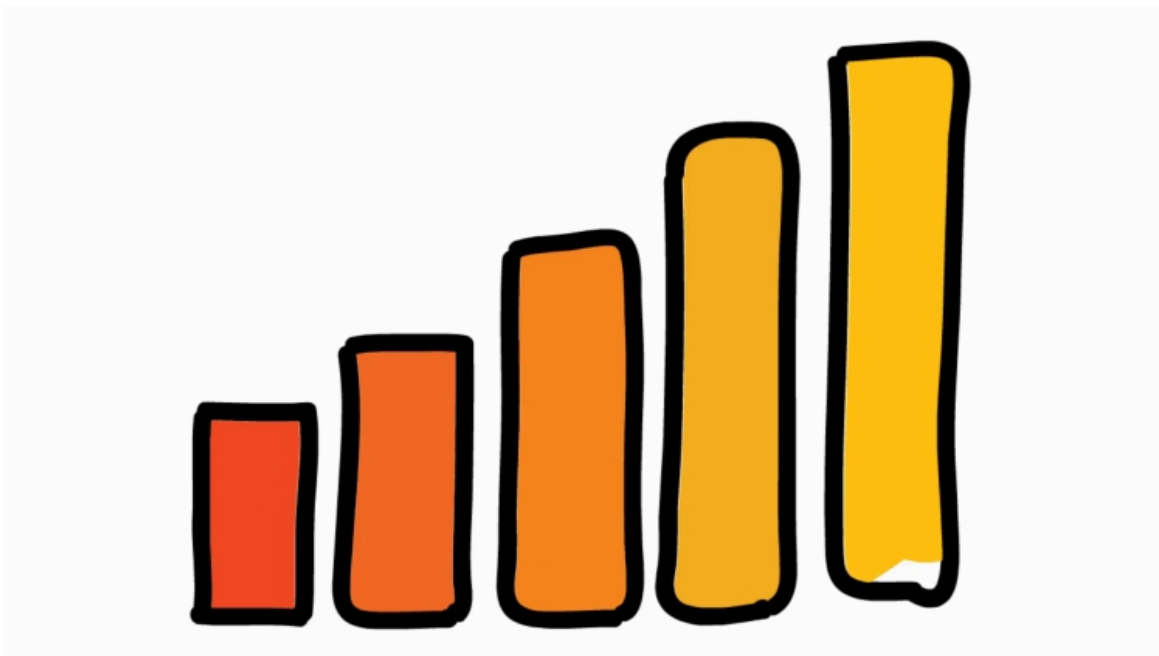
between the values, if there is more than one value for the given row and

column after the transformation. As usual, you can design any custom function

you want, just giving its name or using a  $\lambda$ -function.

## Descriptive Statistics

Descriptive statistics helps to simplify large amounts of data in a sensible way. In contrast to inferential statistics, which will be introduced in a later chapter, descriptive statistics do not draw conclusions beyond the data we are analyzing; nor do we reach any conclusions regarding any hypotheses we may make. We do not try to infer characteristics of the “population” (see below) of the data, but claim to present quantitative descriptions of it in a manageable form. It is simply a way to describe the data.



Statistics, and in particular descriptive statistics, is based on two main concepts:

- A population is a collection of objects, items (“units”) about which information is sought;
- A sample is a part of the population that is observed.

Descriptive statistics applies the concepts, measures, and terms that are used to describe the basic features of the samples in a study. These procedures are essential to provide summaries about the samples as an approximation of the population. Together with simple graphics, they form the basis of every quantitative analysis of data.

In order to describe the sample data and to be able to infer a conclusion, we must go through several steps:

- Data preparation: Given a specific example, we need to prepare the data for generating statistically valid descriptions.
- Descriptive statistics: This generates different statistics to describe and summarize the data concisely, and evaluate different ways to visualize them.

## **Data Preparation**

One of the first tasks when analyzing data is to collect and prepare the data in a format appropriate for analysis of the samples. The most common steps for data preparation involve the following operations.

- Obtaining the data: Data can be read directly from a file, or might be obtained by scraping the web.
- Parsing the data: The right parsing procedure depends on what format the data are in: plain text, fixed columns, CSV, XML, HTML, etc.
- Cleaning the data: Survey responses and other data files are almost always incomplete. Sometimes, there are multiple codes for things such as 'not asked', 'did not know', and 'declined to answer'. There are almost always errors. A simple strategy is to remove or ignore incomplete records.
- Building data structures: Once you've read the data, it is necessary to store it in a data structure that lends itself to the analysis we are interested in. If the data fits into the memory, building a data structure is usually the way to go. If not, usually a database is built, which is an out-of-memory data structure. Most databases provide a mapping from keys to values, so they serve as dictionaries.

## **Exploratory Data Analysis**

The data that comes from performing a particular measurement on all the subjects in a sample represents our observations for a single characteristic like country, age, education, etc. These measurements and categories represent a sample distribution of the variable, which in turn approximately represents the population distribution of the variable. One of the main goals of exploratory data analysis is to visualize and summarize the sample distribution, thereby allowing us to make tentative assumptions about the population distribution.

## Summarizing the Data

The data can be categorical or quantitative. For categorical data, a simple tabulation of the frequency of each category is the best non-graphical exploration for data analysis. For example, we can ask ourselves to identify the proportion of high-income professionals in our database:

```
df1 = df [( df. income == ' >50K\n')]  
  
print 'The rate of people with high income is: ', int ( len ( df1 )/ float ( len (df)) *100) , '%. '  
  
print 'The rate of men with high income is: ', int ( len ( ml1 )/ float ( len (ml)) *100) , '%. '  
  
print 'The rate of women with high income is: ',  
int ( len ( fm1 )/ float ( len (fm)) *100) , '%. '  
  
The rate of people with high income is: 24 %.  
  
The rate of men with high income is: 30 %.  
  
The rate of women with high income is: 10 %.
```

Given a quantitative variable, exploratory data analysis is a way to make preliminary assessments about the population distribution of the variable, using the data of the observed samples. The characteristics of the population distribution of a quantitative variable are its mean, deviation, histograms, outliers, etc. Our observed data represent a finite set of samples of an often infinite

number of possible samples. The characteristics of our randomly observed samples are interesting only to the degree that they represent the population of the data they came from.

- **Mean**

One of the first measurements to obtain sample statistics from the data, is the sample mean. Given a sample of  $n$  values,  $\{x_i\}$ ,  $i = 1, \dots, n$ , the mean,  $\mu$ , is the sum of the values divided by the number of values.

The terms 'mean' and 'average' are often used interchangeably. In fact, the main distinction between them is that the mean of a sample is the summary statistic computed by Eq. (3.1), while an average is not strictly defined and could be one of many summary statistics that can be chosen to describe the central tendency of a sample.

In our case, we can consider what the average age of men and women in our dataset would be in terms of their means:

```
print 'The average age of men is: ', ml['age'].mean ()
print 'The average age of women is: ', fm['age'].mean ()
print 'The average age of high - income men is: ', ml1 ['age'].mean ()
print 'The average age of high - income women is: ',
fm1 ['age'].mean ()
```

The average age of men is: 39.4335474989

The average age of women is: 36.8582304336

The average age of high-income men is: 44.6257880516

The average age of high-income women is: 42.1255301103

This difference in the sample means can be considered initial evidence that there are differences between men and women with high income!



Comment: Later, we will work with both concepts: the population mean and the sample mean. We should not confuse them! The first is the mean of samples taken from the population; the second is the mean of the entire population.

- **Sample Variance**

The mean is not usually a sufficient descriptor of the data. We can go further by knowing two numbers: mean and variance. The variance  $\sigma^2$  describes the spread of the data.

The term  $(x_i - \mu)$  is called the deviation from the mean, so the variance is the mean squared deviation. The square root of the variance,  $\sigma$ , is called the standard deviation. We consider the standard deviation, because the variance is hard to interpret (e.g., if the units are grams, the variance is in grams squared).

Let us compute the mean and the variance of hours per week men and women in our dataset work:

```
ml_mu = ml['age'].mean()
fm_mu = fm['age'].mean()
ml_var = ml['age'].var()
fm_var = fm['age'].var()
ml_std = ml['age'].std()
fm_std = fm['age'].std()
print 'Statistics of age for men: mu:',
ml_mu , 'var:', ml_var , 'std:', ml_std
print 'Statistics of age for women: mu:',
fm_mu , 'var:', fm_var , 'std:', fm_std
```

We can see that the mean number of hours worked per week by women is significantly less than that worked by men, but with a much

higher variance and standard deviation.

- **Sample Median**

The mean of the samples is a good descriptor, but it has a significant shortcoming: what will happen if, in the sample set, there is an error with a value very different from the rest? For example, considering hours worked per week, it would normally be in a range between 20 and 80; but what would happen if, by mistake, there was a value of 1000? An item of data that is significantly different from the rest of the data is called an outlier. In this case, the mean,  $\mu$ , will be drastically shifted towards the outlier. One solution to this drawback is offered by the statistical median,  $\mu_{12}$ , which is an order statistic giving the middle value of a sample. In this case, all values are ordered by their magnitude, and the median is defined as the value that is in the middle of the ordered list. Hence, it is a value that is much more robust in the face of outliers.

Let us see, then, the median age of working men and women in our dataset and the median age of high-income men and women:

```
ml_median = ml['age'].median()
fm_median = fm['age'].median()
print "Median age per men and women : ", ml_median , fm_median

ml_median_age = ml1 ['age'].median()
fm_median_age = fm1 ['age'].median()
print "Median age per men and women with high - income : ",
ml_median_age , fm_median_age
```

Median age per men and women: 38.0 35.0

Median age per men and women with high-income: 44.0 41.0

As expected, the median age of high-income people is higher than the whole set of working people, although the difference between

men and women in both sets is the same.

- **Quantiles and Percentiles**

Sometimes we are interested in observing how sample data are distributed in general. In this case, we can order the samples  $\{x_i\}$ , then find the  $x_p$  so that it divides the data into two parts, where:

- a fraction  $p$  of the data values is less than or equal to  $x_p$  and
- the remaining fraction  $(1 - p)$  is greater than  $x_p$ .

That value,  $x_p$ , is the  $p$ -th quantile, or the  $100 \times p$ -th percentile. For example, a 5-number summary is defined by the values  $x_{\min}$ ,  $Q1$ ,  $Q2$ ,  $Q3$ ,  $x_{\max}$ , where  $Q1$  is the 25  $\times p$ -th percentile,  $Q2$  is the 50  $\times p$ -th percentile and  $Q3$  is the 75  $\times p$ -th percentile.

## **Data Distributions**

Summarizing data by just looking at their mean, median, and variance can be dangerous: very different data can be described by the same statistics. The best thing to do is to validate the data by inspecting each facet. We can have a look at the data distribution, which describes how often each value appears (i.e., what is its frequency).

## **Outlier Treatment**

As mentioned before, outliers are data samples with a value that is far from the central tendency. Different rules can be followed to detect outliers, such as:

- Computing samples that are far from the median.
- Computing samples whose values exceed the mean by 2 or 3 standard deviations.

For example, in our case, we are interested in the age statistics of men versus women with high incomes and we can see that, in our dataset, the minimum age is 17 years and the maximum is 90 years. We can consider that some of these samples are due to errors or are

not representative. Applying our domain knowledge, we focus on the median age (37, in our case) up to 72 and down to 22 years old, and we consider the rest as outliers.

In [17]:

```
df2 = df. drop (df. index [  
(df. income == ' >50K\n') &  
(df['age '] > df['age ']. median () + 35) & (df['age '] > df['age ']. median  
( ) -15)  
)  
ml1_age = ml1 ['age ']  
fm1_age = fm1 ['age ']  
ml2_age = ml1_age . drop ( ml1_age . index [  
( ml1_age > df['age ']. median () + 35) & ( ml1_age > df['age '].  
median () - 15)  
)  
fm2_age = fm1_age . drop ( fm1_age . index [  
( fm1_age > df['age ']. median () + 35) & ( fm1_age > df['age '].  
median () - 15)  
)
```

We can check how the mean and the median changed once the data were cleaned:

In:

```
mu2ml = ml2_age . mean ()  
std2ml = ml2_age . std ()  
md2ml = ml2_age . median ()  
mu2fm = fm2_age . mean ()
```

```

std2fm = fm2_age . std ()
md2fm = fm2_age . median ()

print " Men statistics :"
```

```

print " Mean :", mu2ml , " Std :", std2ml print " Median :", md2ml
print " Min :", ml2_age . min () , " Max :", ml2_age . max ()

print " Women statistics :"
```

```

print " Mean :", mu2fm , " Std :", std2fm print " Median :", md2fm
print " Min :", fm2_age . min () , " Max :", fm2_age . max ()
```

Out: Men statistics: Mean: 44.3179821239 Std: 10.0197498572  
Median:

44.0 Min: 19 Max: 72

Women statistics: Mean: 41.877028181 Std: 10.0364418073  
Median:

41.0 Min: 19 Max: 72

Let us visualize how many outliers are removed from the whole data  
by:

In:

```

plt . figure ( figsize = (13.4 , 5))
df. age [( df. income == ' >50K\n')]
. plot ( alpha = .25 , color = 'blue ')
df2 . age [( df2 . income == ' >50K\n')]
. plot ( alpha = .45 , color = 'red ')
```

Next, we can see that by removing the outliers, the difference between the populations (men and women) actually decreased. In our case, there were more outliers in men than women. If the difference in the mean values before removing the outliers is 2.5, then after removing them, it slightly decreased to 2.44:

In:

```
print 'The mean difference with outliers is: %4.2 f.  
,  
  
% ( ml_age . mean () - fm_age . mean () )  
print 'The mean difference without outliers is:  
%4.2 f.'  
  
% ( ml2_age . mean () - fm2_age . mean () )
```

Out: The mean difference with outliers is: 2.58.

The mean difference without outliers is: 2.44.

Let us observe the difference of men and women incomes in the cleaned subset with some more details.

In:

```
countx , divisionx = np. histogram ( ml2_age , normed = True )  
county , divisiony = np. histogram ( fm2_age , normed = True )  
  
val = [( divisionx [i] + divisionx [i +1]) /2  
for i in range ( len ( divisionx ) - 1)]  
plt . plot (val , countx - county , 'o-')
```

One can see that the differences between male and female values are slightly negative before age 42, and positive after it. Hence, women tend to be promoted (receive more than 50K) earlier than men.

## Measuring Asymmetry: Skewness and Pearson's Median Skewness Coefficient

For univariate data, the formula for skewness is a statistic that measures the asymmetry of the set of  $n$  data samples,  $x_i$  :

$$g_1 = \frac{1}{n} \frac{\sum_i (x_i - \mu)^3}{\sigma^3},$$

where  $\mu$  is the mean,  $\sigma$  is the standard deviation, and  $n$  is the number of data points. Negative deviation indicates that the distribution “skews left” (it extends further to the left than to the right). One can easily see that the skewness for a normal distribution is zero, and any symmetric data must have a skewness of zero. Note that skewness can be affected by outliers! A simpler alternative is to look at the relationship between the mean  $\mu$  and the median  $\mu_{12}$  .

```
def skewness (x):
```

```
    res = 0
```

```
    m = x. mean ()
```

```
    s = x. std ()
```

```
    for i in x:
```

```
        res += (i-m) * (i-m) * (i-m)
```

```
    res /= ( len (x) * s * s * s)
```

```
    return res
```

```
print " Skewness of the male population = ", skewness ( ml2_age )
```

```
print " Skewness of the female population is = ", skewness (
fm2_age )
```

Out:

Skewness of the male population = 0.266444383843

Skewness of the female population = 0.386333524913

That is, the female population is more skewed than the male, probably since men could be more prone to retire later than women.

The Pearson's median skewness coefficient is a more robust alternative to the skewness coefficient and is defined as follows:

$$g_p = 3(\mu - \mu_{12})/\sigma.$$

There are many other definitions for skewness that will not be discussed here. In our case, if we check the Pearson's skewness coefficient for both men and women, we can see that the difference between them actually increases:

In:

```
def pearson (x):
return 3*( x. mean () - x. median ())*x. std ()
print " Pearson 's coefficient of the male population
= ",
pearson ( ml2_age )
print " Pearson 's coefficient of the female population = ",
pearson ( fm2_age )
```

Out: Pearson's coefficient of the male population = 9.55830402221

Pearson's coefficient of the female population = 26.4067269073



After exploring the data, we obtained some apparent effects that seem to support our initial assumptions. For example, the mean age for men in our dataset is 39.4 years; while for women, is 36.8 years. When analyzing for high-income salaries, the mean age for men increased to 44.6 years; while for women, it increased to 42.1 years. When the data were cleaned of outliers, we obtained a mean age for high-income men: 44.3, and for women: 41.8. Moreover, histograms and other statistics show the skewness of the data and the fact that women used to be promoted a little bit earlier than men, in general.

## Continuous Distribution

The distributions we have considered up to now are based on empirical observations and thus are called 'empirical distributions'. As an alternative, we may be interested in considering distributions that are defined by a continuous function and are called continuous distributions [2]. Remember that we defined the PMF,  $f_X(x)$ , of a discrete random variable  $X$  as  $f_X(x) = P(X = x)$  for all  $x$ . In the case of a continuous random variable  $X$ , we speak of the Probability Density Function (PDF), which is defined as  $f_X(x)$  where this satisfies:  $F_X(x) = \int_{-\infty}^x f_X(t) dt$  for all  $x$ . There are many continuous distributions; here, we will examine both exponential and normal distributions.

- **The Exponential Distribution**

Exponential distributions are well known, since they describe the interval time between events. When events are equally likely to occur at any time, the distribution of the interval time tends to be an exponential distribution. The CDF and the PDF of the exponential distribution are defined by the following equations:

$$CDF(x) = 1 - e^{-\lambda x}, \quad PDF(x) = \lambda e^{-\lambda x}.$$

The parameter  $\lambda$  defines the shape of the distribution. It is easy to show that the mean of the distribution is  $1/\lambda$ , the variance is  $1/\lambda^2$  and the median is  $\ln(2)/\lambda$ .

Note that, for a small number of samples, it is difficult to see that the exact empirical distribution fits a continuous distribution. The best way to observe this match is to generate samples from the continuous distribution and see if these samples match the data. As an exercise, you can consider the birthdays of a large group of people, sorting them and computing the interval time in days. If you plot the CDF of the interval times, you will observe the exponential distribution.

There are a lot of real-world events that can be described with this distribution, including the time until a radioactive particle decays; the time it takes before your next telephone call; and the time until default (on payment to company debt holders) in reduced-form credit risk modeling.

- **The Normal Distribution**

The normal distribution, also called the Gaussian distribution, is the most common, since it represents many real phenomena: economic, natural, social, and others. Some well-known examples of real phenomena with a normal distribution are as follows:

- The size of living tissue (length, height, weight).
- The length of inert appendages (hair, nails, teeth) of biological specimens.
- Different physiological measurements (e.g., blood pressure), etc.

## Data Analysis and Libraries

There are numerous data analysis libraries that help us to process and analyze data. They use different programming languages, and have different advantages and disadvantages when solving various data analysis problems. Now, we will introduce some common libraries that may be useful for you. They should give you an overview of the libraries in the field. However, the rest of this module focuses on Python-based libraries.



Some of the libraries that use the Java language for data analysis are as follows:

- **Weka** : This is the library that I became familiar with when I first learned about data analysis. It has a graphical user interface that allows you to run experiments on a small dataset. This is great if you want to get a feel for what is possible in the data processing space. However, if you build a complex product, it is not the best choice, because of its performance, sketchy API design, non-optimal algorithms, and little documentation (<http://www.cs.waikato.ac.nz/ml/weka/>).
- **Mallet** : This is another Java library that is used for statistical natural language processing, document classification, clustering, topic modeling, information extraction, and other Machine Learning applications on

text. There is an add-on package for Mallet, called GRMM, that contains support for inference in general, graphical models, and training of Conditional Random Fields (CRF) with arbitrary graphical structures. In my experience, the library performance and the algorithms are better than Weka. However, its only focus is on text-processing problems. The reference page is at <http://mallet.cs.umass.edu/>.

- **Mahout** : This is Apache's Machine Learning framework built on top of Hadoop; its goal is to build a scalable Machine Learning library. It looks promising, but comes with all the baggage and overheads of Hadoop. The homepage is at <http://mahout.apache.org/>.
- **Spark** : This is a relatively new Apache project, supposedly up to a hundred times faster than Hadoop. It is also a scalable library that consists of common Machine Learning algorithms and utilities. Development can be done in Python as well as in any JVM language. The reference page is at <https://spark.apache.org/docs/1.5.0/mllib-guide.html>.

Here are a few libraries that are implemented in C++:

- **Vowpal Wabbit** : This library is a fast, out-of-core learning system sponsored by Microsoft Research and, previously, Yahoo! Research. It has been used to learn a tera-feature (10<sup>12</sup>) dataset on 1,000 nodes in one hour. More information can be found in the publication at <http://arxiv.org/abs/1110.4198>.
- **MultiBoost** : This package is a multiclass, multilabel, and multitask classification boosting software implemented in C++. If you use this software, you should refer to the paper published in 2012 in the *Journal of Machine Learning Research* ,
- ' MultiBoost: A Multi-purpose Boosting Package', D.Benbouzid, R. Busa-Fekete, N. Casagrande, F.-D. Collin, and B. Kégl.

- **MLpack** : This is also a C++ machine-learning library, developed by the Fundamental Algorithmic and Statistical Tools Laboratory (FASTLab) at Georgia Tech. It focuses on scalability, speed, and ease-of-use, and was presented at the Big Learning workshop of NIPS 2011. Its homepage is at <http://www.mlpack.org/about.html>.
- **Caffe** : The last C++ library we want to mention is Caffe. This is a deep learning framework made with expression, speed, and modularity in mind. It is developed by the Berkeley Vision and Learning Center (BVLC) and community contributors. You can find more information about it at <http://caffe.berkeleyvision.org/>.

Other libraries for data processing and analysis are as follows:

- **Statsmodels** : This is a great Python library for statistical modeling, and is mainly used for predictive and exploratory analysis.
- **Modular toolkit for data processing (MDP)**: This is a collection of supervised and unsupervised learning algorithms and other data processing units that can be combined into data processing sequences and more complex feed-forward network architectures (<http://mdp-toolkit.sourceforge.net/index.html>).
- **Orange** : This is an open-source data visualization and analysis for both novices and experts. It is packed with features for data analysis and has add-ons for bioinformatics and text mining. It contains the implementation of self-organizing maps, which sets it apart from the other projects (<http://orange.biolab.si/>).
- **Mirador** : This is a tool for the visual exploration of complex datasets, supporting Mac and Windows. It enables users to discover correlation patterns and derive new hypotheses from data (<http://orange.biolab.si/>).
- **RapidMiner** : This is another GUI-based tool for data mining, machine learning, and predictive analysis

(<https://rapidminer.com/>).

- **Theano** : This bridges the gap between Python and lower- level languages. Theano gives very significant performance gains, particularly for large matrix operations, and is therefore, a good choice for deep learning models. However, it is not easy to debug because of the additional compilation layer.
- **Natural language processing toolkit (NLTK)** : This is written in Python with unique and very salient features.

I could not list all libraries for data analysis here. However, the preceding libraries are enough to help you learn and build data analysis applications. I hope you will enjoy them after reading this module.

## Data Analysis and Processing

Data is getting bigger and more diverse every day. Therefore, analyzing and processing data to advance human knowledge or to create value is a big challenge. To tackle these challenges, you will need domain knowledge and a variety of skills, drawing from areas such as Computer Science, Artificial Intelligence (AI) and Machine Learning (ML), statistics and mathematics, and domain knowledge.

Let's go through data analysis and its domain knowledge:

- **Computer Science** : We need this knowledge to provide abstractions for efficient data processing. Basic Python programming experience is required to follow the next chapters. We will introduce the Python libraries used in data analysis.
- **Artificial Intelligence and Machine Learning** : If Computer Science knowledge helps us to program data analysis tools, Artificial Intelligence and Machine Learning help us to model the data and learn from it in order to build smart products.
- **Statistics and mathematics** : We cannot extract useful information from raw data if we do not use statistical

techniques or mathematical functions.

- **Domain Knowledge** : Besides technology and general techniques, it is important to have insight into the specific domain. What do the data fields mean? What data do we need to collect? Based on this expertise, we explore and analyze raw data by applying the preceding techniques, step by step.

Data analysis is a process composed of the following steps:

- **Data requirements** : We have to define what kind of data will be collected based on the requirements or problem analysis. For example, if we want to detect a user's behavior while reading news on the Internet, we should be aware of visited article links, dates and times, article categories, and the time the user spends on different pages.
- **Data collection** : Data can be collected from a variety of sources: mobile, personal computer, camera, or recording devices. It may also be obtained in different ways: communication, events, and interactions between person and person, person and device, or device and device. Data appears at all times in all places across the world. The problem is how to find and gather it to solve our problem. This is the mission of this step.
- **Data processing** : Data that is initially obtained must be processed or organized for analysis. This process is performance-sensitive. How fast can we create, insert, update, or query data? When building a real product that has to process big data, we should consider this step carefully. What kind of database should we use to store data? What kind of data structure, such as analysis, statistics, or visualization, is suitable for our purposes?
- **Data cleaning** : After being processed and organized, the data may still contain duplicates or errors. Therefore, we need a cleaning step to reduce those situations and increase the quality of the results in the

following steps. Common tasks include record matching, de-duplication, and column segmentation. Depending on the type of data, we can apply several types of data cleaning. For example, a user's history of visits to a news website might contain a lot of duplicate rows, because the user might have refreshed certain pages many times. For our specific issue, these rows might not carry any meaning when we explore the user's behavior, so we should remove them before saving it to our database. Another situation we may encounter is click fraud on news—someone just wants to improve their website ranking or sabotage a website. In this case, the data will not help us to explore a user's behavior. We can use thresholds to check whether a visit page event comes from a real person or from malicious software.

- **Exploratory data analysis** : Now, we can start to analyze data through a variety of techniques, referred to as 'exploratory data analysis.' We may detect additional problems in data cleaning, or discover requests for further data. Therefore, these steps may be iterative and repeated throughout the whole data analysis process. Data visualization techniques are also used to examine the data in graphs or charts. Visualization often facilitates understanding of data sets, especially if they are large or multidimensional.
- **Modelling and algorithms** : A lot of mathematical formulas and algorithms may be applied to detect or predict useful knowledge from raw data. For example, we can use similarity measures to cluster users who have exhibited similar news-reading behavior, and recommend articles of interest to them next time. Alternatively, we can detect users' genders based on their news reading behavior by applying classification models such as the Support Vector Machine (SVM) or linear regression. Depending on the problem, we may use different algorithms to get an acceptable result. It



can take a lot of time to evaluate the accuracy of the algorithms and choose the best one to implement for a certain product.

- **Data product** : The goal of this step is to build data products that receive data input and generate output according to the problem requirements. We will apply computer science knowledge to implement our selected algorithms as well as manage the data storage.

## Python Libraries in Data Analysis

Python is a multi-platform, general-purpose programming language that can run on Windows, Linux/Unix, and Mac OS X, and has been ported to Java and .NET virtual machines as well. It has a powerful standard library. In addition, it has many libraries for data analysis: Pylearn2, Hebel, Pybrain, Pattern, MontePython, and MILK. In this module, we will cover some common Python data analysis libraries such as Numpy, Pandas, Matplotlib, PyMongo, and Scikit-learn. Now, to help you get started, I will briefly present an overview of each library for those who are less familiar with the scientific Python stack.

### NumPy

One of the fundamental packages used for scientific computing in Python is Numpy. Among other things, it contains the following:

- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions for performing array computations
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra operations, Fourier transformations, and random number capabilities

Besides this, it can also be used as an efficient multidimensional container of generic data. Arbitrary data types can be defined and integrated with a wide variety of databases.

### Pandas

Pandas is a Python package that supports rich data structures and functions for analyzing data, and is developed by the PyData Development Team. It is focused on the improvement of Python's data libraries. Pandas consists of the following things:

- A set of labeled array data structures; the primary of which are Series, DataFrame, and Panel
- Index objects enabling both simple axis indexing and multilevel/hierarchical axis indexing
- An integrated group by engine for aggregating and transforming datasets
- Date range generation and custom date offsets
- Input/output tools that load and save data from flat files or PyTables/HDF5 format
- Optimal memory versions of the standard data structures
- Moving window statistics and static and moving window linear/panel regression

Due to these features, Pandas is an ideal tool for systems that need complex data structures or high-performance time series functions such as financial data analysis applications.

## **Matplotlib**

Matplotlib is the single most used Python package for 2D-graphics. It provides both a very quick way to visualize data from Python and publication-quality figures in many formats: line plots, contour plots, scatter plots, and Basemap plots. It comes with a set of default settings, but allows customization of all kinds of properties. However, we can easily create our charts with the defaults of almost every property in Matplotlib.

## **PyMongo**

MongoDB is a type of NoSQL database. It is highly scalable, robust, and perfect to work with JavaScript-based web applications, because we can store data as JSON documents and use flexible schemas.

PyMongo is a Python distribution containing tools for working with MongoDB. Many tools have also been written for working with PyMongo to add more features such as MongoKit, Humongolus, MongoAlchemy, and Ming.

## **The Scikit-learn library**

The scikit-learn is an open source machine-learning library using the Python programming language. It supports various Machine Learning models, such as classification, regression, and clustering algorithms, interoperated with the Python numerical and scientific libraries NumPy and SciPy.

## NumPy Arrays and Vectorized Computation

NumPy is the fundamental package supported for presenting and computing data with high performance in Python. It provides some interesting features.



- Extension package to Python for multidimensional arrays (ndarrays), various derived objects (such as masked arrays), matrices providing vectorization operations, and broadcasting capabilities. Vectorization can significantly increase the performance of array computations by taking advantage of Single Instruction Multiple Data (SIMD) instruction sets in modern CPUs.
- Fast and convenient operations on arrays of data, including mathematical manipulation, basic statistical operations, sorting, selecting, linear algebra, random number generation, discrete Fourier transforms, and so on.
- Efficiency tools that are closer to hardware because of integrating C/C++/Fortran code.

NumPy is a good starting package for you to get familiar with arrays and array-oriented computing in data analysis. Also, it is the basic

step to learn other, more effective tools such as Pandas, which we will see in the next chapter. We will be using NumPy version 1.9.1.

## NumPy arrays

An array can be used to contain values of a data object in an experiment or simulation step, pixels of an image, or a signal recorded by a measurement device. For example, the latitude of the Eiffel Tower, Paris is 48.858598 and the longitude is 2.294495. It can be presented in a NumPy array object as p:

```
>>> import numpy as np
```

```
>>> p = np.array([48.858598, 2.294495])
```

```
>>> p
```

```
Output: array([48.858598, 2.294495])
```

This is a manual construction of an array using the `np.array` function. The standard convention to import NumPy is as follows:

```
>>> import numpy as np
```

You can, of course, put `from numpy import *` in your code to avoid having to write `np`. However, you should be careful with this habit because of the potential code conflicts (further information on code conventions can be found in the Python Style Guide, also known as PEP8, at <https://www.python.org/dev/peps/pep-0008/>).

There are two requirements of a NumPy array: a fixed size at creation, and a uniform, fixed data type, with a fixed size in memory. The following functions help you to get information on the p matrix:

```
>>> p.ndim # getting dimension of array p
```

```
1
```

```
>>> p.shape # getting size of each array dimension
```

```
(2,)
```

```
>>> len(p) # getting dimension length of array p
```

```
2 >>> p.dtype    # getting data type of array p dtype('float64')
```

## Data Types

There are five basic numerical types, including Booleans (bool), integers (int), unsigned integers (uint), floating point (float), and complex. They indicate how many bits are needed to represent elements of an array in memory. Besides that, NumPy also has some types, such as intc and intp, that have different bit sizes, depending on the platform.

## Array Creation

There are various functions provided to create an array object. They are very useful to create and store data in a multidimensional array in different situations.

## Indexing and Slicing

As with other Python sequence types, such as lists, it is very easy to access and assign a value of each array's element:

```
>>> a = np.arange(7) >>> a array([0, 1, 2, 3, 4, 5, 6]) >>> a[1], a[4],  
a[-1]  
(1, 4, 6)
```

In Python, array indices start at 0. This is in contrast to Fortran or Matlab, where indices begin at 1.

As another example, if our array is multidimensional, we need tuples of integers to index an item:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
>>> a[0, 2]    # first row, third column  
3  
>>> a[0, 2] = 10  
>>> a
```

```
array([[1, 2, 10], [4, 5, 6], [7, 8, 9]])
```

```
>>> b = a[2]
```

```
>>> b
```

```
array([7, 8, 9]) >>> c = a[:2]
```

```
>>> c array([[1, 2, 10], [4, 5, 6]])
```

We call `b` and `c` 'array slices', which are views on the original one. It means that the data is not copied to `b` or `c`, and whenever we modify their values, it will be reflected in the array `a` as well:

```
>>> b[-1] = 11 >>> a
```

```
array([[1, 2, 10], [4, 5, 6], [7, 8, 11]])
```

when we omit the index number.

## Fancy Indexing

Besides indexing with slices, NumPy also supports indexing with Boolean or integer arrays (masks). This method is called fancy indexing. It creates copies, not views.

First, we take a look at an example of indexing with a Boolean mask array:

```
>>> a = np.array([3, 5, 1, 10])
```

```
>>> b = (a % 5 == 0) >>> b array([False, True, False, True],  
dtype=bool) >>> c = np.array([[0, 1], [2, 3], [4, 5], [6, 7]]) >>> c[b]  
array([[2, 3], [6, 7]])
```

The second example is an illustration of using integer masks on arrays:

```
>>> a = np.array([[1, 2, 3, 4],
```

```
[5, 6, 7, 8],
```

```
[9, 10, 11, 12],
```

```
[13, 14, 15, 16]]) >>> a[[2, 1]]
```

```
array([[9, 10, 11, 12], [5, 6, 7, 8]]) >>> a[[-2, -1]]      # select rows  
from the end array([[ 9, 10, 11, 12], [13, 14, 15, 16]]) >>> a[[2, 3], [0,  
1]]    # take elements at (2, 0) and (3, 1) array([9, 14])
```

## Numerical Operations on Arrays

We are getting familiar with creating and accessing ndarrays. Now, we continue to the next step, applying some mathematical operations to array data without writing any for loops, of course, with higher performance.

Scalar operations will propagate the value to each element of the array:

```
>>> a = np.ones(4) >>> a * 2 array([2., 2., 2., 2.]) >>> a + 3 array([4.,  
4., 4., 4.])
```

All arithmetic operations between arrays apply the operation element wise:

```
>>> a = np.ones([2, 4]) >>> a * a array([[1., 1., 1., 1.], [1., 1., 1., 1.]])  
>>> a + a  
array([[2., 2., 2., 2.], [2., 2., 2., 2.]])
```

Also, here are some examples of comparisons and logical operations:

```
>>> a = np.array([1, 2, 3, 4])  
>>> b = np.array([1, 1, 5, 3]) >>> a == b  
array([True, False, False, False], dtype=bool)  
>>> np.array_equal(a, b)    # array-wise comparison  
False  
>>> c = np.array([1, 0])
```



```
>>> d = np.array([1, 1]) >>> np.logical_and(c, d)      # logical
operations array([True, False])
```

## Array Functions

Many helpful array functions are supported in NumPy for analyzing data. We will list the parts of them that are commonly used. First, the transposing function is another kind of reshaping form that returns a view on the original data array without copying anything:

```
>>> a = np.array([[0, 5, 10], [20, 25, 30]]) >>> a.reshape(3, 2)
array([[0, 5], [10, 20], [25, 30]]) >>> a.T
array([[0, 20], [5, 25], [10, 30]])
```

We also have the `swapaxes` method that takes a pair of axis numbers and returns a view on the data, without making a copy:

```
>>> a = np.array([[[0, 1, 2], [3, 4, 5]],
[[6, 7, 8], [9, 10, 11]]]) >>> a.swapaxes(1, 2) array([[[0, 3],  [1, 4],
[2, 5]],
[[6, 9],
[7, 10],
[8, 11]])
```

The transposing function is used to do matrix computations; for example, computing the inner matrix product  $XTX$  using `np.dot`:

```
>>> a = np.array([[1, 2, 3],[4,5,6]]) >>> np.dot(a.T, a) array([[17, 22,
27],  [22, 29, 36],
[27, 36, 45]])
```

Sorting data in an array is also an important demand when processing data. Let's take a look at some sorting functions and their use:

```
>>> a = np.array ([[6, 34, 1, 6], [0, 5, 2, -1]])
```

```
>>> np.sort(a)    # sort along the last axis array([[1, 6, 6, 34], [-1, 0,
2, 5]])

>>> np.sort(a, axis=0)    # sort along the first axis array([[0, 5, 1, -1],
[6, 34, 2, 6]])

>>> b = np.argsort(a)    # fancy indexing of sorted array >>> b
array([[2, 0, 3, 1], [3, 0, 2, 1]]) >>> a[0][b[0]] array([1, 6, 6, 34])

>>> np.argmax(a)    # get index of maximum element 1
```

## Data Processing Using Arrays

With the NumPy package, we can easily solve many kinds of data processing tasks without writing complex loops. It is very helpful for us to control our code, as well as the performance of the program. In this section, we want to introduce some mathematical and statistical functions.

## Loading And Saving Data

We can also save and load data to and from a disk, either in text or binary format, by using different supported functions in the NumPy package.

### Saving an array

Arrays are saved by default in an uncompressed raw binary format, with the file extension `.npy` by the `np.save` function:

```
>>> a = np.array([[0, 1, 2], [3, 4, 5]])

>>> np.save('test1.npy', a)
```

If we want to store several arrays into a single file in an uncompressed `.npz` format, we can use the `np.savez` function, as shown in the following example:

```
>>> a = np.arange(4)

>>> b = np.arange(7)
```

```
>>> np.savez('test2.npz', arr0=a, arr1=b)
```

The .npz file is a zipped archive of files named after the variables they contain. When we load an .npz file, we get back a dictionary-like object that can be queried for its lists of arrays:

```
>>> dic = np.load('test2.npz') >>> dic['arr0'] array([0, 1, 2, 3])
```

Another way to save array data into a file is using the np.savetxt function that allows us to set format properties in the output file:

```
>>> x = np.arange(4)
```

```
>>> # e.g., set comma as separator between elements
```

```
>>> np.savetxt('test3.out', x, delimiter=',')
```

## Loading an Array

We have two common functions (np.load and np.loadtxt) which correspond to the saving functions, for loading an array:

```
>>> np.load('test1.npy') array([[0, 1, 2], [3, 4, 5]]) >>>
np.loadtxt('test3.out', delimiter=',') array([0., 1., 2., 3.]
```

Similar to the np.savetxt function, the np.loadtxt function also has a lot of options for loading an array from a text file.

## Linear algebra with NumPy

Linear algebra is a branch of mathematics concerned with vector spaces and mapping between those spaces. NumPy has a package called linalg that supports powerful linear algebra functions. We can use these functions to find eigenvalues and eigenvectors, or to perform singular value decomposition:

```
>>> A = np.array([[1, 4, 6],
                  [5, 2, 2],
                  [-1, 6, 8]])
```

```
>>> w, v = np.linalg.eig(A) >>> w                                # eigenvalues
array([-0.111 + 1.5756j, -0.111 - 1.5756j, 11.222+0.j]) >>>
v                                # eigenvector array([[-0.0981 + 0.2726j, -0.0981 -
0.2726j, 0.5764+0.j], [0.7683+0.j, 0.7683-0.j, 0.4591+0.j] ,
[-0.5656 - 0.0762j, -0.5656 + 0.00763j, 0.6759+0.j]])
```

The function is implemented using the geev Lapack routines that compute the eigenvalues and eigenvectors of general square matrices.

Another common problem is solving linear systems, such as  $Ax = b$  with  $A$  as a matrix and  $x$  and  $b$  as vectors. The problem can be solved easily by using the `numpy.linalg.solve` function:

```
>>> A = np.array([[1, 4, 6], [5, 2, 2], [-1, 6, 8]])
>>> b = np.array([[1], [2], [3]])
>>> x = np.linalg.solve(A, b) >>> x
array([[-1.77635e-16], [2.5], [-1.5]])
```

## NumPy Random Numbers

An important part of any simulation is the ability to generate random numbers.

For this purpose, NumPy provides various routines in the submodule `random`. It uses a particular algorithm, called the Mersenne Twister, to generate pseudorandom numbers.

First, we need to define a seed that makes the random numbers predictable. When the value is reset, the same numbers will appear every time. If we do not assign the seed, NumPy automatically selects a random seed value based on the system's random number generator device or on the clock:

```
>>> np.random.seed(20)
```

An array of random numbers in the  $[0.0, 1.0]$  interval can be generated as follows:

```
>>> np.random.rand(5) array([0.5881308, 0.89771373, 0.89153073,
0.81583748,      0.03588959])
```

```
>>> np.random.rand(5) array([0.69175758, 0.37868094,
0.51851095, 0.65795147 ,
0.19385022])
```

```
>>> np.random.seed(20)      # reset seed number >>>
np.random.rand(5) array([0.5881308, 0.89771373, 0.89153073,
0.81583748,      0.03588959])
```

If we want to generate random integers in the half-open interval [min, max], we can use the randint(min, max, length) function:

```
>>> np.random.randint(10, 20, 5) array([17, 12, 10, 16, 18])
```

NumPy also provides for many other distributions, including the Beta, binomial, chi-square, Dirichlet, exponential, F, Gamma, geometric, and Gumbel.

We can also use random number generation to shuffle items in a list. Sometimes this is useful when we want to sort a list in random order:

```
>>> a = np.arange(10)
```

```
>>> np.random.shuffle(a) >>> a
```

```
array([7, 6, 3, 1, 4, 2, 5, 0, 9, 8])
```

The following figure shows two distributions, binomial and poisson, side by side with various parameters.

## Data Analysis with Pandas

In this chapter, we will explore another data analysis library called Pandas. The goal of this chapter is to give you some basic knowledge and concrete examples for getting started with Pandas.

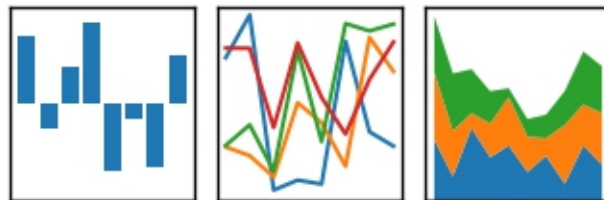
### An Overview of the Pandas Package

Pandas is a Python package that supports fast, flexible, and expressive data structures, as well as computing functions for data analysis. The following are some prominent features that Pandas supports:

- Data structure with labeled axes. This makes the program clean and clear and avoids common errors resulting from misaligned data.
- Flexible handling of missing data.
- Intelligent, label-based slicing, fancy indexing, and subset creation of large datasets.
- Powerful arithmetic operations and statistical computations on a custom axis via axis label.
- Robust input and output support for loading or saving data from and to files, databases, or HDF5 format.

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Related to Pandas installation, we recommend an easy method, which is to install it as a part of Anaconda, a cross-platform distribution for data analysis and scientific computing. You can refer to the reference at <http://docs.continuum.io/anaconda/> to download and install the library.

After installation, we can use it like other Python packages. First, we have to import the following packages at the beginning of the program:

```
>>> import pandas as pd
```

```
>>> import numpy as np
```

## The Pandas Data Structure

Let's first get acquainted with two of Pandas' primary data structures: the Series and the DataFrame. They can handle the majority of use cases in finance, statistics, social science, and many areas of engineering.

### Series

A Series is a one-dimensional object, similar to an array, list, or column in table. Each item in a Series is assigned to an entry in an index:

```
>>> s1 = pd.Series(np.random.rand(4), index=['a', 'b', 'c', 'd'])
>>> s1
a    0.6122 b    0.98096 c    0.3350 d    0.7221 dtype: float64
```

By default, if no index is identified, one will be created with values ranging from 0 to N-1, where N is the length of the Series:

```
>>> s2 = pd.Series(np.random.rand(4))
```

```
>>> s2
```

```
0 0.6913
```

```
1 0.8487
```

```
2 0.8627 3    0.7286 dtype: float64
```

We can access the value of a Series by using the index:

```
>>> s1['c']
```

```
0.3350
```

```
>>> s1['c'] = 3.14 >>> s1['c', 'a', 'b']
c    3.14 a    0.6122 b    0.98096
```

This access method is similar to a Python dictionary. Pandas also allows us to initialize a Series object directly from a Python dictionary:

```
>>> s3 = pd.Series({'001': 'Nam', '002': 'Mary',  
                    '003': 'Peter'})
```

```
>>> s3
```

```
001    Nam
```

```
002    Mary 003    Peter dtype: object
```

Sometimes, we want to filter or rename the index of a Series created from a Python dictionary. At such times, we can send the selected index list directly to the initial function, similarly to the process in the preceding example. Only elements that exist in the index list will be in the Series object. Conversely, indexes that are missing in the dictionary are initialized to default NaN values by Pandas:

```
>>> s4 = pd.Series({'001': 'Nam', '002': 'Mary',  
                    '003': 'Peter'}, index=[  
                    '002', '001', '024', '065'])
```

```
>>> s4
```

```
002    Mary
```

```
001    Nam
```

```
024    NaN 065    NaN dtype: object
```

The library also supports functions that detect missing data:

```
>>> pd.isnull(s4)
```

```
002    False
```

```
001    False
```

```
024    True 065    True dtype: bool
```



Similarly, we can also initialize a Series from a scalar value:

```
>>> s5 = pd.Series(2.71, index=['x', 'y']) >>> s5 x    2.71 y    2.71
dtype: float64
```

A Series object can be initialized with NumPy objects as well, such as ndarray. In addition, Pandas can automatically align data indexed in different ways in arithmetic operations:

```
>>> s6 = pd.Series(np.array([2.71, 3.14]), index=['z', 'y']) >>> s6
z    2.71 y    3.14 dtype: float64 >>> s5 + s6 x    NaN y    5.85
z    NaN dtype: float64
```

## The DataFrame

The DataFrame is a tabular data structure comprising a set of ordered columns and rows. It can be thought of as a group of Series objects that share an index (the column names). There are a number of ways to initialize a DataFrame object. First, let's take a look at the common example of creating a DataFrame from a dictionary of lists:

```
>>> data = {'Year': [2000, 2005, 2010, 2014],
            'Median_Age': [24.2, 26.4, 28.5, 30.3],
            'Density': [244, 256, 268, 279]}
```

```
>>> df1 = pd.DataFrame(data)
```

```
>>> df1
```

	Density	Median_Age	Year
0	244	24.2	2000
1	256	26.4	2005
2	268	28.5	2010
3	279	30.3	2014

By default, the DataFrame constructor will order the column alphabetically. We can edit the default order by applying the

column's attribute to the initializing function:

```
>>> df2 = pd.DataFrame(data, columns=['Year', 'Density' ,  
                                   'Median_Age'])
```

```
>>> df2
```

	Year	Density	Median_Age
0	2000	244	24.2
1	2005	256	26.4
2	2010	268	28.5
3	2014	279	30.3

```
>>> df2.index
```

```
Int64Index([0, 1, 2, 3], dtype='int64')
```

We can provide the index labels of a DataFrame similar to a Series:

```
>>> df3 = pd.DataFrame(data, columns=['Year', 'Density',  
                                   'Median_Age'], index=['a', 'b', 'c', 'd'])
```

```
>>> df3.index
```

```
Index([u'a', u'b', u'c', u'd'], dtype='object')
```

We can construct a DataFrame out of nested lists, as well:

```
>>> df4 = pd.DataFrame([  
    ['Peter', 16, 'pupil', 'TN', 'M', None],  
    ['Mary', 21, 'student', 'SG', 'F', None],  
    ['Nam', 22, 'student', 'HN', 'M', None],  
    ['Mai', 31, 'nurse', 'SG', 'F', None],    ['John', 28, 'lawyer', 'SG', 'M',  
None]], columns=['name', 'age', 'career', 'province', 'sex', 'award'])
```

Columns can be accessed by column name, just as a Series can, either by dictionary-like notation or as an attribute, if the column name is a syntactically valid attribute name:

```
>>> df4.name # or df4['name']
```

```
0 Peter
```

```
1 Mary
```

```
2 Nam
```

```
3 Mai
```

```
4 John
```

```
Name: name, dtype: object
```

To modify or append a new column to the created DataFrame, we specify the column name and the value we want to assign:

```
>>> df4['award'] = None >>> df4
```

	name	age	career	province	sex	award
0	Peter	16	pupil	TN	M	None

```
1 Mary 21 student SG F None
```

```
2 Nam 22 student HN M None
```

```
3 Mai 31 nurse SG F None
```

```
4 John 28 lawer SG M None
```

Using a couple of methods, rows can be retrieved by position or name:

```
>>> df4.ix[1]
```

name	age	21	career	student
province	SG	sex	F	award

```
None Name: 1, dtype: object
```

A DataFrame object can also be created from different data structures, such as a list of dictionaries, a dictionary of Series, or a record array. The method to initialize a DataFrame object is similar to the preceding examples.

Another common method is to provide a DataFrame with data from a location, such as a text file. In this situation, we use the `read_csv` function that expects the column separator to be a comma, by default. However, we can change that by using the `sep` parameter:

```
# person.csv file name,age,career,province,sex Peter,16,pupil,TN,M
Mary,21,student,SG,F
Nam,22,student,HN,M
Mai,31,nurse,SG,F
John,28,lawyer,SG,M
```

```
# loading person.csv into a DataFrame
```

```
>>> df4 = pd.read_csv('person.csv') >>>
df4
  name  age  career  province  sex
0 Peter  16  pupil    TN      M
1 Mary  21  student  SG      F
2 Nam   22  student  HN      M
3 Mai   31  nurse    SG      F
4 John  28  lawyer   SG      M
```

While reading a data file, we sometimes want to skip a line or an invalid value. As for Pandas 0.16.2, `read_csv` supports over 50 parameters for controlling the loading process. Some common useful parameters are as follows:

- `sep`: This is a delimiter between columns. The default is a comma symbol.
- `dtype`: This is a data type for data or columns.
- `header`: This sets row numbers to use as the column names.
- `skiprows`: This defines which line numbers to skip at the start of the file.

- `error_bad_lines`: This shows invalid lines (too many fields) that will, by default, cause an exception, so that no DataFrame will be returned. If we set the value of this parameter as false, any bad lines will be skipped.

Moreover, Pandas also has support for reading and writing a DataFrame directly from or to a database such as the `read_frame` or `write_frame` function within the Pandas module. We will come back to these methods later.

## Essential Basic Functionality

Pandas supports many essential functions that are useful to manipulate Pandas data structures. In this module, we will focus on the most important features regarding exploration and analysis.

### Reindexing and Altering Labels

Reindexing is a critical method in the Pandas data structures. It confirms whether the new or modified data satisfies a given set of labels along a particular axis of Pandas objects.

First, let's view a reindex example on a Series object:

```
>>> s2.reindex([0, 2, 'b', 3])
0    0.6913 2    0.8627 b    NaN 3    0.7286 dtype: float64
```

When reindexed labels do not exist in the data object, a default value of NaN will be automatically assigned to the position; this holds true for the DataFrame case as well:

```
>>> df1.reindex(index=[0, 2, 'b', 3],
                  columns=['Density', 'Year', 'Median_Age', 'C'])
```

	Density	Year	Median_Age	C
0	244	2000	24.2	NaN
2	268	2010	28.5	NaN
b	NaN	NaN	NaN	NaN
3	279	2014	30.3	NaN

We can change the NaN value in the missing index case to a custom value, by setting the `fill_value` parameter.

## Head and Tail

In common data analysis situations, our data structure objects contain many columns and a large number of rows. Therefore, we cannot view or load all the information of the objects. Pandas supports functions that allow us to inspect a small sample. By default, the functions return five elements, but we can set a custom number as well. The following example shows how to display the first five and the last three rows of a longer Series:

```
>>> s7 = pd.Series(np.random.rand(10000))
>>> s7.head()
0 0.631059
1 0.766085
2 0.066891
3 0.867591
4 0.339678
dtype: float64 >>> s7.tail(3)
9997 0.412178
9998 0.800711 9999    0.438344 dtype: float64
```

We can also use these functions for DataFrame objects in the same way.

## Binary Operations

First, we will consider arithmetic operations between objects. In different indexes' objects case, the expected result will be the union of the index pairs. We will not explain this again because we had an example of it in the previous section (`s5 + s6`). This time, we will show another example with a DataFrame:

```
>>> df5 =
pd.DataFrame(np.arange(9).reshape(3,3), columns=
['a','b','c']) >>> df5
```

```
   a  b  c
0  0  1  2
1  3  4  5
```

```
>>> df6 =
pd.DataFrame(np.arange(8).reshape(2,4), columns=
['a','b','c','d']) >>> df6
```

```
   a  b  c  d
0  0  1  2  3
1  4  5  6  7
```

```
>>> df5 + df6
```

```
   a  b  c  d
0  0  2  4  6
1  7  9 11 NaN
```

The mechanisms for returning the result between two kinds of data structures are similar. A problem that we need to consider is the missing data between objects. In this case, if we want to fill with a fixed value, such as 0, we can use arithmetic functions such as add, sub, div, and mul, and the function's supported parameters such as fill\_value:

```
>>> df7 = df5.add(df6, fill_value=0) >>> df7
```

```
   a  b  c  d
0  0  2  4  6
1  7  9 11  7
```

```
2  6  7  8 NaN
```

Next, we will discuss comparison operations between data objects. We have some supported functions, such as equal (eq), not equal (ne), greater than (gt), less than (lt), less equal (le), and greater equal (ge). Here is an example:

```
>>> df5.eq(df6)
```

```
   a  b  c  d
0  True  True  True  False
1  False  False  False  False
```

```
2  False  False  False  False
```

## Functional Statistics

The supported statistics method of a library is really important in data analysis. To get inside a big data object, we need to know some summarized information such as the mean, sum, or q uantile. Pandas supports a large number of methods to compute them. Let's consider a simple example of calculating the sum information of df5, which is a DataFrame object:

```
>>> df5.sum() a    9
b   12 c   15 dtype: int64
```

When we do not specify which axis we want to calculate sum information, by default, the function will calculate on an index axis, which is axis 0:

- Series: We do not need to specify the axis.
- DataFrame: Columns (axis = 1) or index (axis = 0). The default setting is axis 0.

We also have the skipna parameter, which allows us to decide whether to exclude missing data or not. By default, it is set as true:

```
>>> df7.sum(skipna=False) a    13 b    18 c    23 d   NaN dtype:
float64
```

Another function that we want to consider is describe(). It is very convenient for us to summarize most of the statistical information of a data structure, such as the Series and DataFrame, as well:

```
>>> df5.describe()
          a      b      c count  3.0  3.0  3.0
mean    3.0  4.0  5.0 std    3.0  3.0  3.0 min    0.0  1.0  2.0
25%     1.5  2.5  3.5
50%     3.0  4.0  5.0 75%    4.5  5.5  6.5 max    6.0  7.0  8.0
```

We can specify percentiles to include or exclude in the output by using the percentiles parameter; for example, consider the following:

```
>>> df5.describe(percentiles=[0.5, 0.8])
          a      b      c
count  3.0  3.0  3.0
```



```
mean    3.0  4.0  5.0 std    3.0  3.0  3.0 min    0.0  1.0  2.0
50%    3.0  4.0  5.0 80%    4.8  5.8  6.8 max    6.0  7.0  8.0
```

## Function Application

Pandas supports function application that allows us to apply some functions supported in other packages such as NumPy, or our own functions on data structure objects. Here, we illustrate two examples of these cases; first, using apply to execute the std() function, which is the standard deviation calculating function of the NumPy package:

```
>>> df5.apply(np.std, axis=1) # default: axis=0
```

```
0 0.816497
```

```
1 0.816497 2 0.816497 dtype: float64
```

Second, if we want to apply a formula to a data object, we can also use apply function by following these steps:

1. Define the function or formula that you want to apply on a data object.

2. Call the defined function or formula via apply. In this step, we also need to figure out the axis that we want to apply the calculation to:

```
>>> f = lambda x: x.max() - x.min() # step 1
```

```
>>> df5.apply(f, axis=1) # step 2
```

```
0 2
```

```
1 2 2 2 dtype: int64 >>> def sigmoid(x): return 1/(1 + np.exp(x))
```

```
>>> df5.apply(sigmoid) a b c
```

```
0 0.500000 0.268941 0.119203
```

```
1 0.047426 0.017986 0.006693
```

```
2 0.002473 0.000911 0.000335
```

## Sorting

There are two sorting methods that we are interested in: sorting by row or column index, and sorting by data value.

First, we will consider methods for sorting by row and column index. In this case, we have the `sort_index()` function. We also have the `axis` parameter to set (whether the function should sort by row or column). The ascending option with the `True` or `False` value will allow us to sort data in ascending or descending order. The default setting for this option is `True`:

```
>>> df7 =
pd.DataFrame(np.arange(12).reshape(3,4), columns=
['b', 'd', 'a', 'c'], index=['x', 'y', 'z']) >>> df7
b d a c
x 0 1 2 3
y 4 5 6 7
z 8 9 10 11

>>> df7.sort_index(axis=1)
a b c d
x 2 0 3 1
y 6 4 7 5
z 10 8 11 9
```

Series has a method `order` that sorts by value. For `NaN` values in the object, we can also have a special treatment via the `na_position` option:

```
>>> s4.order(na_position='first')
024 NaN
065 NaN
002 Mary 001 Nam dtype: object >>> s4
002 Mary
001 Nam
024 NaN 065 NaN dtype: object
```

Besides that, Series also has the `sort()` function that sorts data by value. However, the function will not return a copy of the sorted data:

```
>>> s4.sort(na_position='first')
>>> s4
024 NaN
065 NaN
```

```
002    Mary 001    Nam dtype: object
```

If we want to apply the sort function to a DataFrame object, we need to figure out which columns or rows will be sorted:

```
>>> df7.sort(['b', 'd'], ascending=False)    b  d  a  c z  8  9  10  11
y  4  5  6  7 x  0  1  2  3
```

If we do not want to automatically save the sorting result to the current data object, we can change the setting of the inplace parameter to False.

## Indexing and Selecting Data

In this section, we will focus on how to get, set, or slice subsets of Pandas data structure objects. As we learned in previous sections, Series or DataFrame objects have axis labeling information. This information can be used to identify items that we want to select or assign a new value to in the object:

```
>>> s4[['024', '002']]    # selecting data of Series object
```

```
024    NaN 002    Mary dtype: object >>> s4[['024', '002']] =
'unknown' # assigning data
```

```
>>> s4
```

```
024    unknown
```

```
065    NaN
```

```
002    unknown 001    Nam dtype: object
```

If the data object is a DataFrame structure, we can also proceed in a similar way:

```
>>> df5[['b', 'c']]    b  c 0  1  2
```

```
1  4  5
```

```
2  7  8
```

For label indexing on the rows of DataFrame, we use the ix function, which enables us to select a set of rows and columns in the object. There are two parameters that we need to specify: the row and column labels that we want to get. By default, if we do not specify the selected column names, the function will return selected rows with all columns in the object:

```
>>> df5.ix[0] a    0 b    1 c    2 Name: 0, dtype: int64 >>> df5.ix[0, 1:3]
b    1 c    2 Name: 0, dtype: int64
```

We also have many ways to select and edit data contained in a Pandas object.

## Computational Tools

Let's start with correlation and covariance computation between two data objects. Both the Series and DataFrame have a cov method. On a DataFrame object, this method will compute the covariance between the Series inside the object:

```
>>> s1 = pd.Series(np.random.rand(3))
>>> s1
0 0.460324
1 0.993279 2 0.032957 dtype: float64 >>> s2 =
pd.Series(np.random.rand(3))
>>> s2
0 0.777509
1 0.573716 2 0.664212 dtype: float64 >>> s1.cov(s2)
-0.024516360159045424

>>> df8 =
pd.DataFrame(np.random.rand(12).reshape(4,3), columns=['a','b','c']) >>> df8
a b c
0 0.200049 0.070034 0.978615
```

```
1 0.293063 0.609812 0.788773
```

```
2 0.853431 0.243656 0.978057
```

```
0.985584 0.500765 0.481180
```

```
>>> df8.cov()          a          b          c a 0.155307 0.021273
-0.048449 b 0.021273 0.059925 -0.040029 c -0.048449
-0.040029 0.055067
```

Usage of the correlation method is similar to the covariance method. It computes the correlation between Series inside a data object, in case the data object is a DataFrame. However, we need to specify which method will be used to compute the correlations. The available methods are Pearson, kendall, and spearman. By default, the function applies the spearman method:

```
>>> df8.corr(method = 'spearman')          a          b          c a 1.0 0.4 -0.8
b 0.4 1.0 -0.8 c -0.8 -0.8 1.0
```

We also have the `corrwith` function that supports calculating correlations between Series that have the same label contained in different DataFrame objects:

```
>>> df9 = pd.DataFrame(np.arange(8).reshape(4,2), columns=['a', 'b'])
>>> df9          a  b
0 0 0 1
```

```
1 2 3
```

```
2 4 5
```

```
3 6 7 >>> df8.corrwith(df9) a 0.955567 b 0.488370 c NaN
dtype: float64
```

## Working With Missing Data

In this section, we will discuss missing, NaN, or null values, in Pandas data structures. It is a very common situation to have missing data in an object. One such case that creates missing data is reindexing:

```

>>> df8 = pd.DataFrame(np.arange(12).reshape(4,3), columns=['a', 'b', 'c'])
a b c
0 0 1 2
1 3 4 5
2 6 7 8
3 9 10 11
>>> df9 = df8.reindex(columns = ['a', 'b', 'c', 'd'])
a b c d
0 0 1 2 NaN
1 3 4 5 NaN
2 6 7 8 NaN
4 9 10 11 NaN
>>> df10 = df8.reindex([3, 2, 'a', 0])
a b c
3 9 10 11
2 6 7 8
a NaN NaN NaN
0 0 1 2

```

To manipulate missing values, we can use the `isnull()` or `notnull()` functions to detect the missing values in a Series object, as well as in a DataFrame object:

```

>>> df10.isnull()
a b c
3 False False False
2 False False False
a True True True
0 False False False

```

On a Series, we can drop all null data and index values, by using the `dropna` function:

```

>>> s4 = pd.Series({'001': 'Nam', '002': 'Mary', '003': 'Peter'}, index=['002', '001', '024', '065'])
>>> s4
002    Mary
001    Nam
024    NaN
065    NaN
dtype: object
>>> s4.dropna()
002    Mary
001    Nam
dtype: object

```

With a DataFrame object, it is a little bit more complex than with Series. We can tell which rows or columns we want to drop, and also

if all entries must be null, or if a single null value is enough. By default, the function will drop any row containing a missing value:

```
>>> df9.dropna() # all rows will be dropped
```

Empty DataFrame

Columns: [a, b, c, d]

```
Index: [] >>> df9.dropna(axis=1)  a  b  c 0 0 1 2
```

```
1 3 4 5
```

```
2 6 7 8
```

```
3 9 10 11
```

Another way to control missing values is to use the supported parameters of functions that we introduced in the previous section. They are also very useful to help solve this problem. In our experience, we should assign a fixed value in missing cases when we create data objects. This will make our objects cleaner for later processing steps. For example, consider the following:

```
>>> df11 = df8.reindex([3, 2, 'a', 0], fill_value = 0) >>> df11  a  b  c
3 9 10 11 2 6 7 8 a 0 0 0 0 0 1 2
```

We can also use the fillna function to fill a custom value in missing values:

```
>>> df9.fillna(-1)  a  b  c d 0 0 1 2 -1
```

```
1 3 4 5 -1
```

```
2 6 7 8 -1
```

```
3 9 10 11 -1
```

## Data Visualization

Data visualization is concerned with the presentation of data in a pictorial or graphical form. It is one of the most important tasks in data analysis, since it enables us to see analytical results, detect outliers, and make decisions for model building. There are many Python libraries for visualization, of which Matplotlib, seaborn, bokeh, and ggplot are among the most popular. However, in this chapter, we mainly focus on the Matplotlib library that is used by many people in many different contexts.



Matplotlib produces publication-quality figures in a variety of formats, and interactive environments across Python platforms. Another advantage is that Pandas comes equipped with useful wrappers around several Matplotlib plotting routines, allowing for quick and handy plotting of Series and DataFrame objects.

The IPython package started as an alternative to the standard interactive Python shell, but has since evolved into an indispensable tool for data exploration, visualization, and rapid prototyping. It is possible to use the graphical capabilities offered by Matplotlib from IPython through various options, of which the simplest to get started with is the `pylab` flag:



```
$ ipython --pylab
```

This flag will preload Matplotlib and Numpy for interactive use with the default Matplotlib backend. IPython can run in various environments: in a terminal, as a Qt application, or inside a browser. These options are worth exploring, since IPython has been adopted for many use cases, such as prototyping, interactive slides for more engaging conference talks or lectures, and as a tool for sharing research.

## The Matplotlib API Primer

The easiest way to get started with plotting using Matplotlib is often by using the MATLAB API that is supported by the package:

```
>>> import matplotlib.pyplot as plt
>>> from numpy import *
>>> x = linspace(0, 3, 6) >>> x array([0., 0.6, 1.2, 1.8, 2.4, 3.]) >>> y
= power(x,2) >>> y array([0., 0.36, 1.44, 3.24, 5.76, 9.]) >>> figure()
>>> plot(x, y, 'r')
>>> xlabel('x')
>>> ylabel('y')
>>> title('Data visualization in MATLAB-like API')
>>> plt.show()
```

However, star imports should not be used unless there is a good reason for doing so. In the case of Matplotlib, we can use the canonical import:

```
>>> import matplotlib.pyplot as plt
```

The preceding example could then be written as follows:

```
>>> plt.plot(x, y)
>>> plt.xlabel('x')
```

```
>>> plt.ylabel('y')
```

```
>>> plt.title('Data visualization using Pyplot of Matplotlib')
```

```
>>> plt.show()
```

If we only provide a single argument to the plot function, it will automatically use it as the y values and generate the x values from 0 to N-1, where N is equal to the number of values:

```
>>> plt.plot(y)
```

```
>>> plt.xlabel('x')
```

```
>>> plt.ylabel('y')
```

```
>>> plt.title('Plot y value without given x values')
```

```
>>> plt.show()
```

By default, the range of the axes is constrained by the range of the input x and y data. If we want to specify the viewport of the axes, we can use the axis() method to set custom ranges. For example, in the previous visualization, we could increase the range of the x axis from [0, 5] to [0, 6], and that of the y axis from [0, 9] to [0, 10], by writing the following command:

```
>>> plt.axis([0, 6, 0, 10])
```

## Line Properties

The default line format when we plot data in Matplotlib is a solid blue line, which is abbreviated as b-. To change this setting, we only need to add the symbol code, which includes letters as color string, and symbols as line style string, to the plot function. Let us consider a plot of several lines with different format styles:

```
>>> plt.plot(x*2, 'g^', x*3, 'rs', x**x, 'y-')
```

```
>>> plt.axis([0, 6, 0, 30])
```

```
>>> plt.show()
```

The output for the preceding command is as follows:

There are many line styles and attributes, such as color, line width, and dash style, that we can choose from to control the appearance of our plots. The following example illustrates several ways to set line properties:

```
>>> line = plt.plot(y, color='red', linewidth=2.0)
>>> line.set_linestyle('--')
>>> plt.setp(line, marker='o')
>>> plt.show()
```

## Figures and Subplots

By default, all plotting commands apply to the current figure and axes. In some situations, we want to visualize data in multiple figures and axes in order to compare different plots, or to use the space on a page more efficiently. There are two steps required before we can plot the data. First, we have to define which figure we want to plot. Second, we need to figure out the position of our subplot in the figure:

```
>>> plt.figure('a') # define a figure, named 'a'
>>> plt.subplot(221) # the first position of 4 subplots in 2x2 figure
>>> plt.plot(y+y, 'r--')
>>> plt.subplot(222) # the second position of 4 subplots
>>> plt.plot(y*3, 'ko')
>>> plt.subplot(223) # the third position of 4 subplots
>>> plt.plot(y*y, 'b^')
>>> plt.subplot(224)
>>> plt.show()
```

In this case, we currently have figure a. If we want to modify any subplot in figure a, we first use the command to select the figure and subplot, and then execute the function to modify the subplot. Here, for example, we change the title of the second plot of our four-plot figure:

```
>>> plt.figure('a')
>>> plt.subplot(222)
>>> plt.title('visualization of  $y^3$ ')
>>> plt.show()
```

There is a convenience method, `plt.subplots()`, to creating a figure that contains a given number of subplots. As in the previous example, we can use the `plt.subplots(2,2)` command to create a 2x2 figure that consists of four subplots.

We can also create the axes manually, instead of using the default rectangular grid, by using the `plt.axes([left, bottom, width, height])` command, where all input parameters are in fractional `[0, 1]` coordinates:

```
>>> plt.figure('b') # create another figure, named 'b'
>>> ax1 = plt.axes([0.05, 0.1, 0.4, 0.32])
>>> ax2 = plt.axes([0.52, 0.1, 0.4, 0.32])
>>> ax3 = plt.axes([0.05, 0.53, 0.87, 0.44])
>>> plt.show()
```

However, when you manually create axes, it takes more time to balance coordinates and sizes between subplots to arrive at a well-proportioned figure.

## Exploring Plot Types

We have looked at how to create simple line plots so far. The Matplotlib library supports many more plot types that are useful for

data visualization. However, our goal is to provide the basic knowledge that will help you to understand and use the library for visualizing data in the most common situations. Therefore, we will only focus on four plot types: scatter plots, bar plots, contour plots, and histograms.

## Scatter Plots

A scatter plot is used to visualize the relationship between variables measured in the same dataset. It is easy to plot a simple scatter plot, using the `plt.scatter()` function, that requires numeric columns for both the x and y axis:

Let's take a look at the command for the preceding output:

```
>>> X = np.random.normal(0, 1, 1000)
>>> Y = np.random.normal(0, 1, 1000)
>>> plt.scatter(X, Y, c = ['b', 'g', 'k', 'r', 'c'])
>>> plt.show()
```

## Bar Plots

A bar plot (sometimes known as a 'bar graph') is used to present grouped data with rectangular bars, which can be either vertical or horizontal, with the lengths of the bars corresponding to their values. We use the `plt.bar()` command to visualize a vertical bar, and the `plt.barh()` command for a horizontal bar:

The command for such output is as follows:

```
>>> X = np.arange(5)
>>> Y = 3.14 + 2.71 * np.random.rand(5)
>>> plt.subplots(2)
>>> # the first subplot
>>> plt.subplot(211)
```

```
>>> plt.bar(X, Y, align='center', alpha=0.4, color='y')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.title('bar plot in vertical')
>>> # the second subplot
>>> plt.subplot(212)
>>> plt.barh(X, Y, align='center', alpha=0.4, color='c')
>>> plt.xlabel('x')
>>> plt.ylabel('y')
>>> plt.title('bar plot in horizontal')
>>> plt.show()
```

## Contour Plots

We use contour plots to present the relationship between three numeric variables in two dimensions. Two variables are drawn along the x and y axes, and the third variable, z, is used for contour levels, which are then plotted as curves in different colors:

```
>>> x = np.linspace(-1, 1, 255)
>>> y = np.linspace(-2, 2, 300)
>>> z = np.sin(y[:, np.newaxis]) * np.cos(x)
>>> plt.contour(x, y, z, 255, linewidth=2)
>>> plt.show()
```

## Legends and Annotations

Legends are an important element that is used to identify the plot elements in a figure. The easiest way to show a legend inside a

figure is to use the label argument of the plot function, and show the labels by calling the plt.legend() method:

```
>>> x = np.linspace(0, 1, 20)
>>> y1 = np.sin(x)
>>> y2 = np.cos(x)
>>> y3 = np.tan(x)
>>> plt.plot(x, y1, 'c', label='y=sin(x)')
>>> plt.plot(x, y2, 'y', label='y=cos(x)')
>>> plt.plot(x, y3, 'r', label='y=tan(x)')
>>> plt.legend(loc='upper left')
>>> plt.show()
```

The loc argument in the legend command is used to figure out the position of the label box. There are several valid location options: lower left, right, upper left, lower center, upper right, center, lower right, upper right, center right, best, upper center, and center left. The default position setting is upper right. However, when we set an invalid location option that does not exist in the above list, the function automatically falls back to the best option.

If we want to split the legend into multiple boxes in a figure, we can manually set our expected labels for plot lines, as shown in the following image:

The output for the preceding command is as follows:

```
>>> p1 = plt.plot(x, y1, 'c', label='y=sin(x)')
>>> p2 = plt.plot(x, y2, 'y', label='y=cos(x)')
>>> p3 = plt.plot(x, y3, 'r', label='y=tan(x)')
>>> lsin = plt.legend(handles=p1, loc='lower right')
>>> lcos = plt.legend(handles=p2, loc='upper left')
```

```

>>> ltan = plt.legend(handles=p3, loc='upper right')
>>> # with above code, only 'y=tan(x)' legend appears in the figure
>>> # fix: add lsin, lcos as separate artists to the axes
>>> plt.gca().add_artist(lsin)
>>> plt.gca().add_artist(lcos)
>>> # automatically adjust subplot parameters to specified padding
>>> plt.tight_layout()
>>> plt.show()

```

The other element in a figure that we want to introduce is the annotations, which can consist of text, arrows, or other shapes to explain parts of the figure in detail, or to emphasize some special data points. There are different methods for showing annotations, such as text, arrow, and annotation.

- The text method draws text at the given coordinates (x, y) on the plot; optionally with custom properties. There are some common arguments in the function: x, y, label text, and font-related properties that can be passed in via fontdict, such as family, fontsize, and style.
- The annotate method can draw both text and arrows, arranged appropriately. Arguments of this function are s (label text), xy (the position of element to annotation), xytext (the position of the label s), xycoords (the string that indicates what type of coordinate xy is), and arrowprops (the dictionary of line properties for the arrow that connects the annotation).

Here is a simple example to illustrate the annotate and text functions:

```

>>> x = np.linspace(-2.4, 0.4, 20)
>>> y = x*x + 2*x + 1

```



```
>>> plt.plot(x, y, 'c', linewidth=2.0) >>> plt.text(-1.5, 1.8, 'y=x^2 + 2*x
+ 1',
            fontsize=14, style='italic') >>> plt.annotate('minima
point', xy=(-1, 0),
            xytext=(-1,
0.3),
            horizontalalignment='center',
            verticalalign
ment='top',
            arrowprops=dict(arrowstyle='->',
connectionstyle='arc3'))

>>> plt.show()
```

## Plotting Functions With Pandas

We have covered most of the important components in a plot figure using Matplotlib. In this section, we will introduce another powerful plotting method for directly creating standard visualization from Pandas data objects that are often used to manipulate data.

For Series or DataFrame objects in Pandas, most plotting types are supported, such as line, bar, box, histogram, scatter plots, and pie charts. To select a plot type, we use the kind argument of the plot function. With no kind of plot specified, the plot function will generate a line-style visualization by default, as in the following example:

```
>>> s = pd.Series(np.random.normal(10, 8, 20))

>>> s.plot(style='ko—', alpha=0.4, label='Series plotting')

>>> plt.legend()

>>> plt.show()
```

## Additional Python Data Visualization Tools

Besides Matplotlib, there are other powerful data visualization toolkits based on Python. While we cannot dive deeper into these libraries, we would like to at least briefly introduce them in this session.

### Bokeh

Bokeh is a project by Peter Wang, Hugo Shi, and others at Continuum Analytics. It aims to provide elegant and engaging

visualizations in the style of D3.js. The library can quickly and easily create interactive plots, dashboards, and data applications. Here are a few differences between Matplotlib and Bokeh:

- Bokeh achieves cross-platform ubiquity through IPython's new model of in-browser client-side rendering
- Bokeh uses a syntax familiar to R and ggplot users, while Matplotlib is more familiar to Matlab users
- Bokeh can build a ggplot-inspired, in-browser interactive visualization tool, while Matplotlib focuses on 2D cross-platform graphics.

The basic steps for creating plots with Bokeh are as follows:

- Prepare some data in a list, series, and DataFrame
- Tell Bokeh where you want to generate the output
- Call `figure()` to create a plot with some overall options, similar to the Matplotlib options discussed earlier
- Add renderers for your data, with visual customizations such as colors, legends, and width
- Ask Bokeh to `show()` or `save()` the results

## MayaVi

MayaVi is a library for interactive scientific data visualization and 3D plotting, built on top of the award-winning visualization toolkit (VTK), which is a traits-based wrapper for the open-source visualization library. It offers the following:

- The ability to interact with the data and object in the visualization through dialogs.
- An interface in Python for scripting. MayaVi can work with Numpy and scipy for 3D plotting out of the box and can be used within IPython notebooks, which is similar to Matplotlib.
- An abstraction over VTK that offers a simpler programming model.

## Data Mining

We are collecting information at a scale that has never been seen before in the history of mankind and placing more day-to-day importance on the use of this information in everyday life. We expect our computers to translate Web pages into other languages, predict the weather, suggest books we would like, and diagnose our health issues. These expectations will grow, both in the number of applications and also in the efficacy we expect. Data mining is a methodology that we can employ to train computers to make decisions with data, and forms the backbone of many high-tech systems of today.



The Python language is fast growing in popularity, for good reason. It gives the programmer a lot of flexibility; it has a large number of modules to perform different tasks; and Python code is usually more readable and concise than most other languages. There is a large and an active community of researchers, practitioners, and beginners using Python for data mining.

In this chapter, we will introduce data mining with Python. We will cover the following topics:

- What is data mining, and where can it be used?

- Setting up a Python-based environment to perform data mining
- An example of affinity analysis, which is recommending products based on purchasing habits
- An example of (a classic) classification problem, predicting a plant species based on its measurements

## Introducing Data Mining

Data mining provides a way for a computer to learn how to make decisions with data. This decision could be predicting tomorrow's weather, blocking a spam e-mail from entering your inbox, detecting the language of a website, or finding a new romance on a dating site. There are many different applications of data mining, with new applications being discovered all the time.

Data mining is part of algorithms, statistics, engineering, optimization, and computer science. We also use concepts and knowledge from other fields, such as linguistics, neuroscience, or town planning. Applying it effectively usually requires domain-specific knowledge to be integrated with the algorithms.

Most data mining applications work with the same high-level view, although the details often change quite considerably. We start our data mining process by creating a dataset, describing an aspect of the real world. Datasets comprise two aspects:

- Samples that are objects in the real world. This can be a book, photograph, animal, person, or any other object.
- Features that are descriptions of the samples in our dataset. Features could be the length, frequency of a given word, number of legs, date it was created, and so on.

The next step is tuning the data mining algorithm. Each data mining algorithm has parameters, either within the algorithm or supplied by the user. This tuning allows the algorithm to learn how to make decisions about the data.

## **A Simple Affinity Analysis Example**

In this section, we jump into our first example. A common-use case for data mining is to improve sales by asking a customer who is buying a product if he/she would like another similar product as well. This can be done through affinity analysis, which is the study of when things exist together.

What is affinity analysis?

Affinity analysis is a type of data mining that gives similarity between samples (objects). This could be the similarity between the following:

- Users on a website, in order to provide varied services or targeted advertising
- Items to sell to those users, in order to provide recommended movies or products
- Human genes, in order to find people that share the same ancestors

We can measure affinity in a number of ways. For instance, we can record how frequently two products are purchased together. We can also record the accuracy of the statement when a person buys object 1, and also when they buy object 2.

## **Product Recommendations**

One of the issues with moving a traditional business online, such as commerce, is that tasks that used to be done by humans need to be automated in order for the online business to scale. One example of this is up-selling, or selling an extra item to a customer who is already buying. Automated product recommendations through data mining are one of the driving forces behind the e-commerce revolution that is turning billions of dollars per year into revenue.

In this example, we are going to focus on a basic product recommendation service. We design this based on the following idea: when two items are historically purchased together, they are more likely to be purchased together in the future. This sort of

thinking is behind many product recommendation services, in both online and offline businesses.

A very simple algorithm for this type of product recommendation algorithm is to simply find any historical case where a user has brought an item and to recommend other items that the historical user brought. In practice, simple algorithms such as this can do well, or at least better than choosing random items to recommend. However, they can be improved upon significantly, which is where data mining comes in.

To simplify the coding, we will consider only two items at a time. As an example, people may buy bread and milk at the same time at the supermarket. In this early example, we wish to find simple rules of the form:

If a person buys product X, then they are likely to purchase product Y

More complex rules involving multiple items will not be covered, such as people buying sausages and burgers being more likely to buy tomato sauce.

## **Loading the Dataset with NumPy**

The dataset can be downloaded from the code package supplied with the course. Download this file and save it on your computer, noting the path to the dataset. For this example, I recommend that you create a new folder on your computer to put your dataset and code in. From here, open your IPython Notebook, navigate to this folder, and create a new notebook.

The dataset we are going to use for this example is a NumPy two-dimensional array, which is a format that underlies most of the examples in the rest of the module. The array looks like a table, with rows representing different samples, and columns representing different features.

The cells represent the value of a particular feature of a particular sample. To illustrate, we can load the dataset with the following

code:

```
import numpy as np
```

```
dataset_filename = "affinity_dataset.txt" X =  
np.loadtxt(dataset_filename)
```

For this example, run the IPython Notebook and create an IPython Notebook. Enter the above code into the first cell of your Notebook. You can then run the code by pressing Shift + Enter (which will also add a new cell for the next batch of code). After the code is run, the square brackets to the left-hand side of the first cell will be assigned an incrementing number, letting you know that this cell has been completed.

For later code that will take more time to run, an asterisk will be placed to denote that this code is either running or scheduled to run. This asterisk will be replaced by a number when the code has completed running.

You will need to save the dataset into the same directory as the IPython Notebook. If you choose to store it somewhere else, you will need to change the dataset\_filename value to the new location.

Next, we can show some of the rows of the dataset to get a sense of what the dataset looks like. Enter the following line of code into the next cell and run it, in order to print the first five lines of the dataset:

```
print(X[:5])
```

The dataset can be read by looking at each row (horizontal line) at a time. The first row (0, 0, 1, 1, 1) shows the items purchased in the first transaction. Each column (vertical row) represents each of the items. They are bread, milk, cheese, apples, and bananas, respectively. So, in the first transaction, the person bought cheese, apples, and bananas, but not bread or milk.

Each of these features contain binary values, stating only whether the items were purchased and not how many of them were purchased. A 1 indicates that "at least 1" item was bought of this

type, while a 0 indicates that absolutely none of that item was purchased.

## **Implementing a Simple Ranking of Rules**

We wish to find rules of the type ‘If a person buys product X, then they are likely to purchase product Y.’ We can quite easily create a list of all of the rules in our dataset by simply finding all occasions when two products were purchased together. However, we then need a way to determine good rules from bad ones. This will allow us to choose specific products to recommend.

Rules of this type can be measured in many ways, of which we will focus on two: support and confidence.

Support is the number of times that a rule occurs in a dataset, which is computed by simply counting the number of samples that the rule is valid for. It can sometimes be normalized by dividing by the total number of times the premise of the rule is valid, but we will simply count the total for this implementation.

While the support measures how often a rule exists, confidence measures how accurate they are and when they can be used. It can be computed by determining the percentage of times the rule applies when the premise applies. We first count how many times a rule applies in our dataset, and divide it by the number of samples where the premise (the if statement) occurs.

As an example, we will compute the support and confidence for the rule ‘if a person buys apples, they also buy bananas.’

As the following example shows, we can tell whether someone bought apples in a transaction by checking the value of `sample[3]`, where a sample is assigned to a row of our matrix:

Similarly, we can check if bananas were bought in a transaction by seeing if the value for `sample[4]` is equal to 1 (and so on). We can now compute the number of times our rule exists in our dataset and, from that, we can determine confidence and support.



Now, we need to compute these statistics for all the rules in our database. We will do this by creating a dictionary for both valid rules and invalid rules. The key to this dictionary will be a tuple (premise and conclusion). We will store the indices, rather than the actual feature names. Therefore, we would store (3 and 4) to signify the previous rule 'If a person buys apples, they will also buy bananas.' If the premise and conclusion are given, the rule is considered valid. However, if the premise is given but the conclusion is not, the rule is considered invalid for that sample.

To compute the confidence and support for all possible rules, we must first set up some dictionaries to store the results. We will use defaultdict for this, which sets a default value if a key is accessed that doesn't yet exist. We record the number of valid rules, invalid rules, and occurrences of each premise:

```
from collections import defaultdict
valid_rules = defaultdict(int)
invalid_rules = defaultdict(int)
num_occurrences = defaultdict(int)
```

Next, we compute these values in a large loop. We iterate over each sample and feature in our dataset. This first feature forms the premise of the rule—if a person buys a product premise:

```
for sample in X:
    for premise in range(4):
```

We check whether the premise exists for this sample. If not, we do not have any more processing to do on this sample/premise combination, and move to the next iteration of the loop:

```
        if sample[premise] == 0: continue
```

If the premise is valid for this sample (it has a value of 1), then we record this and check each conclusion of our rule. We skip over any conclusion that is the same as the premise—this would give us rules such as 'If a person buys apples, then they buy apples,' which obviously doesn't help us much;

```
        num_occurrences[premise] += 1
        for conclusion in range(n_features):
            if premise == conclusion: continue
```

If the conclusion exists for this sample, we increment our valid count for this rule. If not, we increment our invalid count for this rule:

```
if sample[conclusion] == 1:    valid_rules[(premise, conclusion)] += 1
else:    invalid_rules[(premise, conclusion)] += 1
```

We have now completed computing the necessary statistics, and can now compute the support and confidence for each rule. As before, the support is simply our valid\_rules value: `support = valid_rules`

The confidence is computed in the same way, but we must loop over each rule to compute this:

```
confidence = defaultdict(float)
for premise, conclusion in valid_rules.keys():
```

```
    rule = (premise, conclusion)    confidence[rule] = valid_rules[rule] / num_occurrences[premise]
```

We now have a dictionary showing the support and confidence for each rule.

We can create a function that will print out the rules in a readable format. The signature of the rule takes the premise and conclusion indices, the support and confidence dictionaries we just computed, and the features array that tells us what the features mean:

```
def print_rule(premise, conclusion, support, confidence, features):
```

We get the names of the features for the premise and conclusion and print out the rule in a readable format:

```
    premise_name = features[premise]    conclusion_name = features[conclusion]
```

```
    print("Rule: If a person buys {0} they will also buy {1}".format(premise_name, conclusion_name))
```

Then we print out the Support and Confidence of this rule:

```

    print("
{0}".format(support[(premise,
)])
{0:.3f}".format(confidence[(premise,
conclusion)]))
    print("
-
Support:
conclusion)
Confidence:
-

```

We can test the code by calling it in the following way—feel free to experiment with different premises and conclusions:

## Ranking to Find the Best Rules

Now that we can compute the support and confidence of all rules, we want to be able to find the best rules. To do this, we perform a ranking, and print the rules with the highest values. We can do this for both the support and confidence values.

To find the rules with the highest support, we first sort the support dictionary. Dictionaries do not support ordering by default; the `items()` function gives us a list containing the data in the dictionary. We can sort this list, using the `itemgetter` class as our key, which allows for the sorting of nested lists such as this one. Using `itemgetter(1)` allows us to sort based on the values. Setting `reverse=True` gives us the highest values first:

```
from operator import itemgetter
```

```
sorted_support = sorted(support.items(), key=itemgetter(1), reverse=True)
```

We can then print out the top five rules:

```
for index in range(5):
```

```

    print("Rule #{0}".format(index + 1))
    premise, conclusion = sorted_support[index][0]
    print_rule(premise, conclusion, support, confidence, features)

```

## A Simple Classification Example

In the affinity analysis example, we looked for correlations between different variables in our dataset. In classification, we instead have a

single variable that we are interested in, that we call the 'class' (also called the 'target'). If, in the previous example, we were interested in how to make people buy more apples, we could set that variable to be the class and look for classification rules that obtain that goal. We would then look only for rules that relate to that goal.

## **What Is Classification?**

Classification is one of the largest uses of data mining, both in practical use and in research. As before, we have a set of samples that represents objects or things we are interested in classifying. We also have a new array, the class values. These class values give us a categorization of the samples. Some examples are as follows:

- Determining the species of a plant by looking at its measurements. The class value here would be 'Which species is this?'.
- Determining if an image contains a dog. The class would be 'Is there a dog in this image?'.
- Determining if a patient has cancer based on the test results. The class would be 'Does this patient have cancer?'.

While many of the examples above are binary (yes/no) questions, they do not have to be, as in the case of plant species classification in this section.

The goal of classification applications is to train a model on a set of samples with known classes, and then apply that model to new, unseen samples with unknown classes. For example, we want to train a spam classifier on my past e-mails, which I have labeled as 'spam' or 'not spam'. I then want to use that classifier to determine whether my next e-mail is spam, without me needing to classify it myself.

## **Loading and Preparing the Dataset**

The dataset we are going to use for this example is the famous Iris database of plant classification. In this dataset, we have 150 plant

samples and four measurements of each: sepal length, sepal width, petal length, and petal width (all in centimeters). This classic dataset (first used in 1936!) is one of the classic datasets for data mining. There are three classes: Iris Setosa, Iris Versicolour, and Iris Virginica. The aim is to determine which type of plant a sample is, by examining its measurements.

The Scikit-learn library contains this dataset built-in, making the loading of the dataset straightforward:

```
from sklearn.datasets import load_iris
dataset = load_iris()
X = dataset.data
y = dataset.target
```

You can also print(`dataset.DESCR`) to see an outline of the dataset, including some details about the features.

The features in this dataset are continuous values, meaning they can take any range of values. Measurements are a good example of this type of feature, where a measurement can take the value of 1, 1.2, or 1.25 and so on. Another aspect about continuous features is that feature values that are close to each other indicate similarity. A plant with a sepal length of 1.2 cm is similar to a plant with a sepal width of 1.25 cm.

In contrast are categorical features. These features, while often represented as numbers, cannot be compared in the same way. In the Iris dataset, the class values are an example of a categorical feature. The class 0 represents Iris Setosa, class 1 represents Iris Versicolour, and class 2 represents Iris Virginica. This doesn't mean that Iris Setosa is more similar to Iris Versicolour than it is to Iris Virginica—despite the class value being more similar. The numbers here represent categories. All we can say is whether categories are the same or different.

There are other types of features too, some of which will be covered in later chapters.

While the features in this dataset are continuous, the algorithm we will use in this example requires categorical features. Turning a

continuous feature into a categorical feature is a process called discretization.

A simple discretization algorithm is to determine a threshold, and any values below this threshold are given a value 0. Meanwhile, any above this are given the value 1. For our threshold, we will compute the mean (average) value for that feature. To start with, we compute the mean for each feature:

```
attribute_means = X.mean(axis=0)
```

This will give us an array of length 4, which is the number of features we have. The first value is the mean of the values for the first feature, and so on. Next, we use this to transform our dataset from one with continuous features to one with discrete categorical features:

```
X_d = np.array(X >= attribute_means, dtype='int')
```

We will use this new `X_d` dataset (for `X` discretized) for our training and testing, rather than the original dataset (`X`).

## Implementing the OneR algorithm

OneR is a simple algorithm that simply predicts the class of a sample by finding the most frequent class for the feature values. OneR is a shorthand for One Rule, indicating that we only use a single rule for this classification, by choosing the feature with the best performance. While some of the later algorithms are significantly more complex, this simple algorithm has been shown to have good performance in a number of real-world datasets.

The algorithm starts by iterating over every value of every feature. For that value, count the number of samples from each class that have that feature value. Record the most frequent class for the feature value, and the error of that prediction.

For example, if a feature has two values, 0 and 1, we first check all samples that have the value 0. For that value, we may have 20 in class A, 60 in class B, and 20 in class C. The most frequent class

for this value is B, and there are 40 instances that have different classes. The prediction for this feature value is B with an error of 40, as there are 40 samples that have a different class from the prediction. We then do the same procedure for the value 1 for this feature, and then for all other feature value combinations.

Once all of these combinations are computed, we compute the error for each feature by summing up the errors for all values for that feature. The feature with the lowest total error is chosen as the One Rule, and is then used to classify other instances.

In code, we will first create a function that computes the class prediction and error for a specific feature value. We have two necessary imports, defaultdict and itemgetter, that we used in earlier code:

```
from collections import defaultdict from operator import itemgetter
```

Next, we create the function definition, which requires the dataset, classes, the index of the feature we are interested in, and the value we are computing: `def train_feature_value(X, y_true, feature_index, value):`

We then iterate over all the samples in our dataset, counting the actual classes for each sample with that feature value:

```
class_counts = defaultdict(int)
for sample, y in zip(X, y_true):
    if sample[feature_index] == value:
        class_counts[y] += 1
```

We then find the most frequently assigned class by sorting the `class_counts` dictionary and finding the highest value:

```
sorted_class_counts = sorted(class_counts.items(),
                              key=itemgetter(1), reverse=True)
most_frequent_class = sorted_class_counts[0][0]
```

Finally, we compute the error of this rule. In the OneR algorithm, any sample with this feature value would be predicted as being the most frequent class. Therefore, we compute the error by summing up

the counts for the other classes (not the most frequent). These represent training samples that this rule does not work on:

```
incorrect_predictions = [class_count for class_value, class_count  
in class_counts.items() if class_value != most_frequent_class]  
error = sum(incorrect_predictions)
```

Finally, we return both the predicted class for this feature value and the number of incorrectly classified training samples, the error, of this rule:

```
return most_frequent_class, error
```

With this function, we can now compute the error for an entire feature by looping over all the values for that feature, summing the errors, and recording the predicted classes for each value.

The function header needs the dataset, classes, and feature index we are interested in:

```
def train_on_feature(X, y_true, feature_index):
```

Next, we find all of the unique values that the given feature takes. The indexing in the next line looks at the whole column for the given feature and returns it as an array. We then use the set function to find only the unique values:

```
values = set(X[:,feature_index])
```

Next, we create our dictionary that will store the predictors. This dictionary will have feature values as the keys, and classification as the values. An entry with key 1.5 and value 2 would mean that, when the feature has value set to 1.5, it will classify an object/data point as belonging to class 2. We also create a list storing the errors for each feature value:

```
predictors = {}    errors = []
```

As the main section of this function, we iterate over all the unique values for this feature and use our previously defined `train_feature_value()` function to find the most frequent class and



the error for a given feature value. We store the results as outlined above:

```
for current_value in values:
    most_frequent_class, error =
    train_feature_value(X, y_true, feature_index,
    current_value) predictors[current_value] =
    most_frequent_class errors.append(error)
```

Finally, we compute the total errors of this rule and return the predictors along with this value :

```
total_error = sum(errors) return predictors, total_error
```

## Testing the Algorithm

When we evaluated the affinity analysis algorithm for the last section, our aim was to explore the current dataset. With this classification, our problem is different. We want to build a model that will allow us to classify previously unseen samples by comparing them to what we know about the problem.

For this reason, we split our Machine Learning workflow into two stages: training and testing. In training, we take a portion of the dataset and create our model. In testing, we apply that model and evaluate how effectively it worked on the dataset. As our goal is to create a model that is able to classify previously unseen samples, we cannot use our testing data for training the model. If we do, we run the risk of overfitting.

Overfitting is the problem of creating a model that classifies our training dataset very well, but performs poorly on new samples. The solution is quite simple: never use training data to test your algorithm. This simple rule has some complex variants, which we will cover in later chapters; but for now, we can evaluate our OneR implementation by simply splitting our dataset into two small datasets: a training one and a testing one. This workflow is given in this section.

The Scikit-learn library contains a function to split data into training and testing components:

```
from sklearn.cross_validation import train_test_split
```

This function will split the dataset into two sub-datasets, according to a given ratio (which by default uses 25 percent of the dataset for testing). It does this randomly, which improves the confidence that the algorithm is being appropriately tested:

```
Xd_train, Xd_test, y_train, y_test = train_test_split(X_d, y, random_state=14)
```

We now have two smaller datasets: `Xd_train` contains our data for training and `Xd_test` contains our data for testing. `y_train` and `y_test` give the corresponding class values for these datasets.

We also specify a specific `random_state`. Setting the random state will give the same split every time the same value is entered. It will appear random, but the algorithm used is deterministic, and the output will be consistent. For this module, I recommend setting the random state to the same value that I do, as it will give you the same results that I get, allowing you to verify your results. To get truly random results that change every time you run it, set `random_state` to none.

Next, we compute the predictors for all the features for our dataset. Remember to only use the training data for this process. We iterate over all the features in the dataset and use our previously defined functions to train the predictors and compute the errors:

```
all_predictors = {} errors = {} for feature_index in range(Xd_train.shape[1]):
```

```
    predictors, total_error = train_on_feature(Xd_train, y_train, feature_index)
```

```
    all_predictors[feature_index] = predictors errors[feature_index] = total_error
```

Next, we find the best feature to use as our "One Rule", by finding the feature with the lowest error:

```
best_feature, best_error = sorted(errors.items(), key=itemgetter(1))  
[0]
```

We then create our model by storing the predictors for the best feature:

```
model = {'feature': best_feature, 'predictor':  
all_predictors[best_feature][0]}
```

Our model is a dictionary that tells us which feature to use for our One Rule, and the predictions that are made based on the values it has. Given this model, we can predict the class of a previously unseen sample by finding the value of the specific feature and using the appropriate predictor. The following code does this for a given sample:

```
variable = model['variable'] predictor = model['predictor'] prediction =  
predictor[int(sample[variable])]
```

Often, we want to predict a number of new samples at one time, which we can do using the following function; use the above code, but iterate over all the samples in a dataset, obtaining the prediction for each sample:

```
def predict(X_test, model):  
    variable = model['variable']    predictor = model['predictor']  
    y_predicted = np.array([predictor[int(sample[variable])]  
for    sample in X_test])    return y_predicted
```

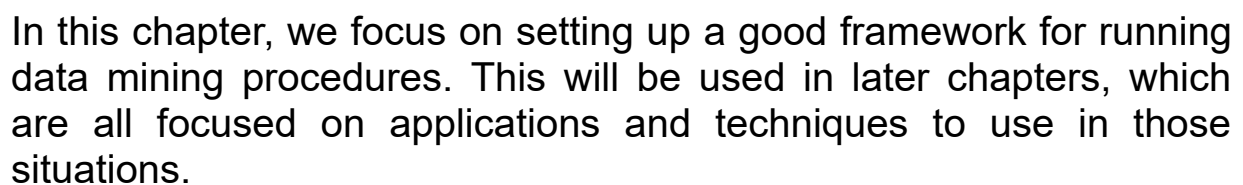
For our testing dataset, we get the predictions by calling the following function: `y_predicted = predict(X_test, model)`

We can then compute the accuracy of this by comparing it to the known classes:

```
accuracy = np.mean(y_predicted == y_test) * 100 print("The test  
accuracy is {:.1f}%".format(accuracy))
```

This gives an accuracy of 68 percent, which is not bad for a single rule!

The Scikit-learn library is a collection of data mining algorithms, written in Python and using a common programming interface. This allows users to easily try different algorithms, as well as to utilize standard tools for effective testing and parameter searching. There are a large number of algorithms and utilities in scikit-learn.



- Estimators: This is to perform classification, clustering, and regression.
- Transformers: This is to perform preprocessing and data alterations.
- Pipelines: This is to put together your workflow into a replicable format Scikit-learn estimators.

Estimators are Scikit-learn's abstractions, allowing for the standardized implementation of a large number of classification algorithms. Estimators are used for classification. Estimators have two main functions:

- `fit()`: This performs the training of the algorithm and sets internal parameters. It takes two inputs, the training sample dataset and the corresponding classes for those samples.
- `predict()`: This predicts the class of the testing samples that is given as input. This function returns an array with the predictions of each input testing sample.

Most Scikit-learn estimators use the NumPy arrays or a related format for input and output.

There are a large number of estimators in Scikit-learn. These include support vector machines (SVM), random forests, and neural networks. Many of these algorithms will be used in later chapters. In this chapter, we will use a different estimator from Scikit-learn: nearest neighbor.

For this chapter, you will need to install Matplotlib (if you haven't already done so). The easiest way to install it is to use pip3 to install Scikit-learn:

```
$pip3 install matplotlib
```

If you have any difficulty installing Matplotlib, seek the official installation instructions at <http://matplotlib.org/users/installing.html>.

## Nearest Neighbors

Nearest neighbors is one of the most intuitive algorithms in the set of standard data mining algorithms. To predict the class of a new sample, we look through the training dataset for the samples that are most similar to our new sample. We take the most similar sample and predict the class that the majority of those samples have.

As an example, we wish to predict the class of a triangle, based on which class it is more similar to (represented here by having similar objects closer together). We seek the three nearest neighbors, which are two diamonds and one square. There are more diamonds than circles, and the predicted class for the triangle is, therefore, a diamond:

Nearest neighbors can be used for nearly any dataset--however, it can be very computationally expensive to compute the distance between all pairs of samples. For example, if there are 10 samples in the dataset, there are 45 unique distances to compute. However, if there are 1000 samples, there are nearly 500,000! Various methods exist for improving this speed dramatically; some of which are covered in the later sections of this module.

It can also do poorly in categorical-based datasets, and another algorithm should be used for these instead.

## **Distance Metrics**

A key underlying concept in data mining is that of distance. If we have two samples, we need to know how close they are to each other. Furthermore, we need to answer questions such as 'Are these two samples more similar than the other two?' Answering questions like these is important to the outcome of the case.

The most common distance metric that the people are aware of is Euclidean distance, which is the real-world distance. If you were to plot the points on a graph and measure the distance with a straight ruler, the result would be the Euclidean distance. A little more formally, it is the square root of the sum of the squared distances for each feature.

Euclidean distance is intuitive, but provides poor accuracy if some features have larger values than others. It also gives poor results when lots of features have a value of 0, known as a sparse matrix. There are other distance metrics in use; two commonly employed ones are the Manhattan and Cosine distance.

The Manhattan distance (also called City Block) is the sum of the absolute differences in each feature (with no use of square distances). Intuitively, it can be thought of as the number of moves a rook (or castle) in chess would take to move between the points, if it were limited to moving one square at a time. While the Manhattan distance does suffer if some features have larger values than others, the effect is not as dramatic as in the case of Euclidean.

The Cosine distance is better suited to cases where some features are larger than others, and when there are lots of zeros in the dataset. Intuitively, we draw a line from the origin to each of the samples, and measure the angle between those lines. This can be seen in the following diagram:

## **Loading the Dataset**

The dataset we are going to use is called Ionosphere, which is the recording of many high-frequency antennas. The aim of the antennas is to determine whether there is a structure in the ionosphere, and a region in the upper atmosphere. Those that have a structure are deemed good, while those that do not are deemed bad. The aim of this application is to build a data mining classifier that can determine whether an image is good or bad.

This can be downloaded from the UCL Machine Learning data repository, which contains a large number of datasets for different data mining applications. Go to <http://archive.ics.uci.edu/ml/datasets/Ionosphere>, and click on Data Folder. Download the ionosphere.data and ionosphere.names files to a folder on your computer. For this example, I'll assume that you have put the dataset in a directory called Data in your home folder.

The location of your home folder depends on your operating system. For Windows, it is usually at C:\Documents and Settings\username. For Mac or Linux machines, it is usually at /home/username. You can get your home folder by running this Python code: `import os  
print(os.path.expanduser("~"))`



For each row in the dataset, there are 35 values. The first 34 are measurements taken from the 17 antennas (two values for each antenna). The last is either 'g' or 'b' ('good' or 'bad').

Start the IPython Notebook server and create a new notebook called Ionosphere Nearest Neighbors for this chapter.

First, we load up the NumPy and csv libraries that we will need for our code:

```
import numpy as np
import csv
```

To load the dataset, we first get the filename of the dataset. First, get the folder the dataset is stored in from your data folder:

```
data_filename = os.path.join(data_folder, "Ionosphere",
                              "ionosphere.data")
```

We then create the X and y NumPy arrays to store the dataset in. The sizes of these arrays are known from the dataset. Don't worry if you don't know the size of future datasets—we will use other methods to load the dataset in future sections and you won't need to know this size beforehand:

```
X = np.zeros((351, 34), dtype='float')
y = np.zeros((351,), dtype='bool')
```

The dataset is in a Comma-Separated Values (CSV) format, which is a commonly used format for datasets. We are going to use the csv module to load this file. Import it and set up a csv reader object:

```
with open(data_filename, 'r') as input_file:
    reader = csv.reader(input_file)
```

Next, we loop over the lines in the file. Each line represents a new set of measurements, which is a sample in this dataset. We use the `enumerate` function to get the line's index as well, so we can update the appropriate sample in the dataset (X):

```
for i, row in enumerate(reader):
```

We take the first 34 values from this sample, turn each into a float, and save that to our dataset :

```
data = [float(datum) for datum in row[:-1]] X[i] = data
```

Finally, we take the last value of the row and set the class. We set it to 1 (or True) if it is a good sample, and 0 if it is not:

```
y[i] = row[-1] == 'g'
```

We now have a dataset of samples and features in X, and the corresponding classes in y, as we did in the classification example in 'Data Mining'.

## **Moving Towards a Standard Workflow**

Estimators in Scikit-learn have two main functions: `fit()` and `predict()`. We train the algorithm using the `fit` method and our training set. We evaluate it using the `predict` method on our testing set.

First, we need to create these training and testing sets. As before, import and run the `train_test_split` function:

```
from sklearn.cross_validation import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_  
state=14)
```

Then, we import the nearest neighbor class and create an instance for it. We leave the parameters as defaults for now, and will choose good parameters later in this chapter. By default, the algorithm will choose the five nearest neighbors to predict the class of a testing sample:

```
from sklearn.neighbors import KNeighborsClassifier estimator =  
KNeighborsClassifier()
```

After creating our estimator, we must then fit it on our training dataset. For the nearest neighbor class, this records our dataset, allowing us to find the nearest neighbor for a new data point, by

comparing that point to the training dataset: `estimator.fit(X_train, y_train)`

We then train the algorithm with our test set and evaluate with our testing set:

```
y_predicted = estimator.predict(X_test) accuracy = np.mean(y_test == y_predicted) * 100 print("The accuracy is {0:.1f}%".format(accuracy))
```

This scores 86.4 percent accuracy, which is impressive for a default algorithm and just a few lines of code! Most Scikit-learn default parameters are chosen explicitly to work well with a range of datasets. However, you should always aim to choose parameters based on knowledge of the application experiment.

## Running the Algorithm

In our earlier experiments, we set aside a portion of the dataset as a testing set, with the rest being the training set. We train our algorithm on the training set and evaluate how effective it will be based on the testing set. However, what happens if we happen to choose an easy testing set? Alternately, what if it was particularly troublesome? We can discard a good model due to poor results resulting from such an "unlucky" split of our data.

The cross-fold validation framework is a way to address the problem of choosing a testing set and a standard methodology in data mining. The process works by doing a number of experiments with different training and testing splits, but using each sample in a testing set only once. The procedure is as follows:

- Split the entire dataset into a number of sections, called 'folds'.
- For each fold in the dataset, execute the following steps:
  - Set that fold aside as the current testing set ° Train the algorithm on the remaining folds

- Evaluate on the current testing set
- Report on all the evaluation scores, including the average score.
- In this process, each sample is used in the testing set only once. This reduces (but doesn't completely eliminate) the likelihood of choosing 'lucky' testing sets.

Throughout this module, the code examples build upon each other within a chapter. Each chapter's code should be entered into the same IPython Notebook, unless otherwise specified.

The Scikit-learn library contains a number of cross-fold validation methods. A helper function is provided that performs the preceding procedure. We can import it now to our IPython Notebook: `from sklearn.cross_validation import cross_val_score`

By default, `cross_val_score` uses a specific methodology called 'Stratified K Fold' to split the dataset into folds. This creates folds that have approximately the same proportion of classes in each fold, again reducing the likelihood of choosing poor folds. This is a great default, so we won't mess with it right now.

Next, we use this function, applying it to the original (full) dataset and classes:

```
scores = cross_val_score(estimator, X, y, scoring='accuracy')
average_accuracy = np.mean(scores) * 100
print("The average accuracy is {0:.1f}%".format(average_accuracy))
```

This gives a slightly more modest result of 82.3 percent, but it is still quite good considering we have not yet tried setting better parameters. In the next section, we will see how we can go about changing the parameters to achieve a better outcome.

## Setting Parameters

Almost all data mining algorithms have parameters that the user can set. This serves to generalize an algorithm, to allow it to be applicable in a wide variety of circumstances. Setting these

parameters can be quite difficult, as choosing good parameter values is often highly reliant on features of the dataset.

The nearest neighbor algorithm has several parameters, but the most important one is that of the number of nearest neighbors to use when predicting the class of an unseen attribution. In Scikit-learn, this parameter is called `n_neighbors`. In the following figure, we show that, when this number is too low, a randomly labeled sample can cause an error. In contrast, when it is too high, the actual nearest neighbors have a smaller effect on the result:

On the left-hand side, we would usually expect the test sample (the triangle) to be classified as a circle. However, if `n_neighbors` is 1, the single red diamond in this area (likely a noisy sample) causes the sample to be predicted as being a diamond, while it appears to be in a red area. In figure (b), on the right-hand side, we would usually expect the test sample to be classified as a diamond. However, if `n_neighbors` is 7, the three nearest neighbors (which are all diamonds) are overridden by the large number of circle samples.

If we want to test a number of values for the `n_neighbors` parameter, for example, each of the values from 1 to 20, we can rerun the experiment many times by setting `n_neighbors` and observing the result:

```
avg_scores = [] all_scores = []

parameter_values = list(range(1, 21)) # Include 20 for n_neighbors
in parameter_values:

    estimator =
    KNeighborsClassifier(n_neighbors=n_neighbors)
    scores =
    cross_val_score(estimator, X, y, scoring='accuracy')
```

Compute and store the average in our list of scores. We also store the full set of scores for later analysis:

```
    avg_scores.append(np.mean(scores))    all_scores.append(scores)
```

We can then plot the relationship between the value of `n_neighbors` and the accuracy. First, we tell the IPython Notebook that we want to show plots inline in the notebook itself:

```
%matplotlib inline
```

We then import `pyplot` from the `Matplotlib` library and plot the parameter values alongside average scores:

```
from matplotlib import pyplot as plt plt.plot(parameter_values,  
avg_scores, '-o')
```

While there is a lot of variance, the plot shows a decreasing trend as the number of neighbors increases.

## Preprocessing Using Pipelines

When taking measurements of real-world objects, we often get features in very different ranges. For instance, if we are measuring the qualities of an animal, we might have several features, as follows:

- Number of legs: This is between the range of 0-8 for most animals, while some have many more!
- Weight: This is between the range of only a few micrograms, all the way to a blue whale with a weight of 190,000 kilograms!
- Number of hearts: This can be between zero to five, in the case of the earthworm.

For a mathematical-based algorithm to compare each of these features, the differences in the scale, range, and units can be difficult to interpret. If we used the above features in many algorithms, the weight would probably be the most influential feature, largely due to the larger numbers, and not anything to do with the actual effectiveness of the feature.

One of the methods to overcome this is to use a process called 'preprocessing' to normalize features so that they all have the same range, or are put into categories like small, medium and large.

Suddenly, the large difference in the types of features has less of an impact on the algorithm, and can lead to large increases in the accuracy.

Preprocessing can also be used to choose only the more effective features, create new features, and so on. Preprocessing in Scikit-learn is done through Transformer objects, which take a dataset in one form and return an altered dataset after some transformation of the data. These don't have to be numerical, as Transformers are also used to extract features--however, in this section, we will stick with preprocessing.

## An Example

We can show an example of the problem by breaking the Ionosphere dataset. While this is only an example, many real-world datasets have problems of this form. First, we create a copy of the array so that we do not alter the original dataset:

```
X_broken = np.array(X)
```

Next, we break the dataset by dividing every second feature by 10:

```
X_broken[:,::2] /= 10
```

In theory, this should not have a great effect on the result. After all, the values for these features are still relatively similar. The major issue is that the scale has changed, and the odd features are now larger than the even features. We can see the effect of this by computing the accuracy:

```
estimator = KNeighborsClassifier()
```

```
original_scores = cross_val_score(estimator, X, y,  
scoring='accuracy')
```

```
print("The original average accuracy for is
```

```
{0:.1f}%".format(np.mean(original_scores) * 100)) broken_scores =  
cross_val_score(estimator, X_broken, y, scoring='accuracy')
```

```
print("The 'broken' average accuracy for is  
{0:.1f}%".format(np.mean(broken_scores) * 100))
```

This gives a score of 82.3 percent for the original dataset, which drops down to 71.5 percent on the broken dataset. We can fix this by scaling all the features to the range 0 to 1.

## Standard Preprocessing

The preprocessing we will perform for this experiment is called 'feature-based normalization' through the `MinMaxScaler` class. Continuing with the IPython notebook for the rest of this section; first, we import this class:

```
from sklearn.preprocessing import MinMaxScaler
```

This class takes each feature and scales it to the range 0 to 1. The minimum value is replaced with 0, the maximum with 1, and the other values somewhere in between.

To apply our preprocessor, we run the transform function on it. While `MinMaxScaler` doesn't, some Transformers need to be trained first, in the same way that the classifiers do. We can combine these steps by running the `fit_transform` function instead:

```
X_transformed = MinMaxScaler().fit_transform(X)
```

Here, `X_transformed` will have the same shape as `X`. However, each column will have a maximum of 1 and a minimum of 0.

There are various other forms of normalizing in this way, which is effective for other applications and feature types:

- Ensure the sum of the values for each sample equals 1, using `sklearn.preprocessing.Normalizer`
- Force each feature to have a zero mean and a variance of 1, using `sklearn.preprocessing.StandardScaler`, which is a commonly used starting point for normalization
- Turn numerical features into binary features, where any value above a threshold is 1 and any below is 0, using



`sklearn.preprocessing.Binarizer`

We will use combinations of these preprocessors in later chapters, along with other types of Transformers.

## Putting It All Together

We can now create a workflow by combining the code from the previous sections, using the broken dataset previously calculated:

```
X_transformed = MinMaxScaler().fit_transform(X_broken) estimator  
= KNeighborsClassifier()
```

```
transformed_scores = cross_val_score(estimator, X_transformed,  
y, scoring='accuracy')
```

```
print("The average accuracy for is  
{0:.1f}%".format(np.mean(transformed_scores) * 100))
```

This gives us back our score of 82.3 percent accuracy. The MinMaxScaler resulted in features of the same scale, meaning that no features overpowered others by simply being bigger values. While the nearest neighbor algorithm can be confused with larger features, some algorithms handle scale differences better. But be careful--some are much worse!

## Pipelines

As experiments grow, so does the complexity of the operations. We may split up our dataset, binarize features, perform feature-based scaling, perform sample-based scaling, and many more operations.

Keeping track of all of these operations can get quite confusing, and can result in being unable to replicate the result. Common problems include forgetting a step, incorrectly applying a transformation, or adding a transformation that wasn't needed.

Another issue is the order of the code. In the previous section, we created our `X_transformed` dataset, and then created a new estimator for the cross validation. If we had multiple steps, we would need to track all of these changes to the dataset in the code.

Pipelines are a construct that addresses these problems (and others, which we will see later). Pipelines store the steps in your data mining workflow. They can take your raw data in, perform all the necessary transformations, and then create a prediction. This allows us to use pipelines in functions such as `cross_val_score`, where they expect an Estimator. First, import the Pipeline object:

```
from sklearn.pipeline import Pipeline
```

Pipelines take a list of steps as input, representing the chain of the data mining application. The last step needs to be an Estimator, while all previous steps are Transformers. The input dataset is altered by each Transformer, with the output of one step being the input of the next step. Finally, the samples are classified by the last step's Estimator. In our pipeline, we have two steps:

- Use `MinMaxScaler` to scale the feature values from 0 to 1
- Use `KNeighborsClassifier` as the classification algorithms

Each step is then represented by a tuple ('name', step). We can then create our pipeline:

```
scaling_pipeline = Pipeline([('scale',  
MinMaxScaler()),  
('predict', KNeighborsClassifier())])
```

The key here is the list of tuples. The first tuple is our scaling step, and the second tuple is the predicting step. We give each step a name: the first we call 'scale' and the second we call 'predict', but you can choose your own names. The second part of the tuple is the actual Transformer or Estimator object.

Running this pipeline is now very easy, using the cross validation code from before:

```
scores = cross_val_score(scaling_pipeline, X_broken, y,  
scoring='accuracy') print("The pipeline scored an average accuracy  
for is {0:.1f}%".format(np.mean(transformed_scores) * 100))
```

This gives us the same score as before (82.3 percent), which is expected, as we are effectively running the same steps.

We will soon use more advanced testing methods, and setting up pipelines is a great way to ensure that the code complexity does not grow unmanageably.

## **Giving Computers the Ability to Learn from Data**

In my opinion, Machine Learning, the application and science of algorithms that makes sense of data, is the most exciting field of all the computer sciences! We are living in an age where data comes in abundance; using the self-learning algorithms from the field of Machine Learning, we can turn this data into knowledge. Thanks to the many powerful open source libraries that have been developed in recent years, there has probably never been a better time to break into the Machine Learning field and learn how to utilize powerful algorithms to spot patterns in data and make predictions about future events.

[illegible]

In this chapter, we will learn about the main concepts and different types of Machine Learning. Together with a basic introduction to the relevant terminology, we will lay the groundwork for successfully using Machine Learning techniques for practical problem solving.

In this chapter, we will cover the following topics:

- The general concepts of Machine Learning
- The three types of learning and basic terminology
- The building blocks for successfully designing Machine Learning systems

## **How to Transform Data into Knowledge**

In this age of modern technology, there is one resource that we have in abundance: a large amount of structured and unstructured data. In the second half of the twentieth century, Machine Learning evolved as a subfield of Artificial Intelligence that involved the development of self-learning algorithms to gain knowledge from that data in order to make predictions. Instead of requiring humans to manually derive rules and build models from analyzing large amounts of data, Machine Learning offers a more efficient alternative for capturing the knowledge in data to gradually improve the performance of predictive models, and make data-driven decisions. Not only is Machine Learning becoming increasingly important in computer science research, it also plays an ever-expanding role in our everyday life. Thanks to Machine Learning, we enjoy robust e-mail spam filters, convenient text and voice recognition software, reliable Web search engines, challenging chess players, and, hopefully soon, safe and efficient self-driving cars.

## **The Three Different Types of Machine Learning**

In this section, we will take a look at the three types of Machine Learning: supervised learning, unsupervised learning, and reinforcement learning. We will learn about the fundamental differences between the three different learning types and, using conceptual examples, we will develop an intuition for the practical problem domains where these can be applied:

### **Making Predictions about the Future with Supervised Learning**

The main goal in supervised learning is to learn a model from labeled training data that allows us to make predictions about unseen or future data. Here, the term ‘supervised’ refers to a set of samples where the desired output signals (labels) are already known.

Consider the example of e-mail spam filtering--we can train a model using a supervised machine learning algorithm on a body of labeled e-mail, e-mail that are correctly marked as 'spam' or 'not-spam', to predict whether a new e-mail belongs to either of the two categories. A supervised learning task with discrete class labels, such as in the previous e-mail spam-filtering example, is also called a classification task. Another subcategory of supervised learning is regression, where the outcome signal is a continuous value:

- **Classification for Predicting Class Labels**

Classification is a subcategory of supervised learning, where the goal is to predict the categorical class labels of new instances based on past observations. These class labels are discrete, unordered values that can be understood as the 'group memberships' of the instances. The previously mentioned example of e-mail-spam detection represents a typical example of a binary classification task, where the Machine Learning algorithm learns a set of rules in order to distinguish between two possible classes: spam and non-spam e-mail.

However, the set of class labels does not have to be of a binary nature. The predictive model learned by a supervised learning algorithm can assign any class label that was present in the training dataset to a new, unlabeled instance. A typical example of a multi-class classification task is handwritten character recognition. Here, we could collect a training dataset that consists of multiple handwritten examples of each letter in the alphabet. Now, if a user provides a new handwritten character via an input device, our predictive model will be able to predict the correct letter in the alphabet with a certain accuracy. However, our Machine Learning system would be unable to correctly recognize any of the digits zero to nine, for example, if they were not part of our training dataset.

The following figure illustrates the concept of a binary classification task given 30 training samples: 15 training samples are labeled as 'negative' class (circles) and 15 training samples are labeled as 'positive' class (plus signs). In this scenario, our dataset is two-dimensional, which means that each sample has two values

associated with it:  $x_1$  and  $x_2$ . Now, we can use a supervised Machine Learning algorithm to learn a rule—the decision boundary represented as a black dotted line—that can separate those two classes and classify new data into each of those two categories, given its  $x_1$  and  $x_2$  values:

- **Regression for Predicting Continuous Outcomes**

We learned in the previous section that the task of classification is to assign categorical, unordered labels to instances. A second type of supervised learning is the prediction of continuous outcomes, which is also called regression analysis. In regression analysis, we are given a number of predictor (explanatory) variables and a continuous response variable (outcome); and we try to find a relationship between those variables that allows us to predict an outcome.

For example, let's assume that we are interested in predicting the Math SAT scores of our students. If there is a relationship between the time spent studying for the test and the final scores, we could use it as training data to learn a model that uses study time to predict the test scores of future students who are planning to take this test.

The term 'regression' was devised by Francis Galton in his article 'Regression Towards Mediocrity' in *Hereditary Stature* in 1886. Galton described the biological phenomenon that the variance of height in a population does not increase over time. He observed that the height of parents is not passed on to their children, but that children's height is regressing towards the population mean.

The following figure illustrates the concept of linear regression. Given a predictor variable  $x$  and a response variable  $y$ , we fit a straight line to this data that minimizes the distance—most commonly the average squared distance—between the sample points and the fitted line. We can now use the intercept and slope learned from this data to predict the outcome variable of new data:

## **Solving Interactive Problems with Reinforcement Learning**



Another type of Machine Learning is reinforcement learning. In reinforcement learning, the goal is to develop a system (agent) that improves its performance based on interactions with the environment. Since the information about the current state of the environment typically also includes a so-called 'reward' signal, we can think of reinforcement learning as a field related to supervised learning. However, in reinforcement learning, this feedback is not the correct ground truth label or value, but a measure of how well the action was measured by a reward function. Through interaction with the environment, an agent can then use reinforcement learning to learn a series of actions that maximizes this reward, via an exploratory trial-and-error approach, or by deliberative planning.

A popular example of reinforcement learning is a chess engine. Here, the agent decides upon a series of moves depending on the state of the board (the environment), and the reward can be defined as 'win or lose' at the end of the game:

## **Discovering Hidden Structures with Unsupervised Learning**

In supervised learning, we know the right answer beforehand when we train our model, and in reinforcement learning, we define a measure of reward for particular actions by the agent. In unsupervised learning, however, we are dealing with unlabeled data or data of unknown structure. Using unsupervised learning techniques, we are able to explore the structure of our data to extract meaningful information, without the guidance of a known outcome variable or reward function.

- **Finding Subgroups with Clustering**

Clustering is an exploratory data analysis technique that allows us to organize a pile of information into meaningful subgroups (clusters), without having any prior knowledge of their group memberships. Each cluster that may arise during the analysis defines a group of objects that share a certain degree of similarity but are more dissimilar to objects in other clusters, which is why

clustering is also sometimes called "unsupervised classification." Clustering is a great technique for structuring information and deriving meaningful relationships among data. For example, it allows marketers to discover customer groups based on their interests in order to develop distinct marketing programs.

The figure below illustrates how clustering can be applied to organizing unlabeled data into three distinct groups, based on the similarity of their features  $x_1$  and  $x_2$ :

- **Dimensionality Reduction for Data Compression**

Another subfield of unsupervised learning is dimensionality reduction. Often, we are working with data of high dimensionality—each observation comes with a high number of measurements—that can present a challenge for limited storage space and the computational performance of Machine Learning algorithms. Unsupervised dimensionality reduction is a commonly used approach in feature preprocessing to remove noise from data, which can also degrade the predictive performance of certain algorithms, and compress the data onto a smaller dimensional subspace, while still retaining most of the relevant information.

Sometimes, dimensionality reduction can also be useful for visualizing data—for example, a high-dimensional feature set can be projected onto one-, two-, or three-dimensional feature spaces in order to visualize it via 3D- or 2D-scatterplots or histograms. The figure below shows an example where non-linear dimensionality reduction was applied to compress a 3D Swiss Roll onto a new 2D feature subspace:

## **An Introduction to Basic Terminology and Notations**

Now that we have discussed the three broad categories of machine learning—supervised, unsupervised, and reinforcement learning—let's have a look at the basic terminology that we will be using. The following table shows an excerpt of the Iris dataset, which is a classic example in the field of Machine Learning. The Iris dataset

contains the measurements of 150 iris flowers from three different species: Setosa, Versicolor, and Virginica. Here, each flower sample represents one row in our data set, and the flower measurements in centimeters are stored as columns, which we also call the features of the dataset:

To keep the notation and implementation simple yet efficient, we will make use of some of the basics of linear algebra. For our purposes, we will use a matrix and vector notation to refer to our data. We will follow the common convention to represent each sample as separate row in a feature matrix  $X$ , where each feature is stored as a separate column.

The Iris dataset, consisting of 150 samples and 4 features, can then be written as a

150×4 matrix  $X$  :

For the rest of this module, we will use the superscript ( $i$ ) to refer to the  $i$ th training sample, and the subscript  $j$  to refer to the  $j$ th dimension of the training dataset.

We use lower-case, bold-face letters to refer to vectors ( $x \in \mathbb{R}^{n \times 1}$ ) and upper-case, bold-face letters to refer to matrices. To refer to single elements in a vector or matrix, we write the letters in italics ( $x_i$  or  $x_{ij}$ ), respectively).

## **A Roadmap for Building Machine Learning Systems**

In the previous sections, we discussed the basic concepts of Machine Learning and the three different types of learning. In this section, we will discuss other important parts of a Machine Learning system that accompanies the learning algorithm. The diagram below shows a typical workflow diagram for using Machine Learning in predictive modeling, which we will discuss in the following subsections:

### **Preprocessing – Getting Data into Shape**

Raw data rarely comes in the form and shape that is necessary for the optimal performance of a learning algorithm. Thus, the preprocessing of the data is one of the most crucial steps in any Machine Learning application. If we take the Iris flower dataset from the previous section as an example, we could think of the raw data as a series of flower images from which we want to extract meaningful features. Useful features could be the color, the hue, the intensity of the flowers, the height, and the flower lengths and widths. Many Machine Learning algorithms also require that the selected features are on the same scale for optimal performance, which is often achieved by transforming the features in the range  $[0, 1]$  or a standard normal distribution with zero mean and unit variance.

Some of the selected features may be highly correlated, and therefore redundant to a certain degree. In those cases, dimensionality reduction techniques are useful for compressing the features onto a lower dimensional subspace. Reducing the dimensionality of our feature space has the advantage of requiring less storage space, so the learning algorithm can run much faster.

To determine whether our Machine Learning algorithm not only performs well on the training set but also generalizes well to new data, we also want to randomly divide the dataset into a separate training and test set. We use the training set to train and optimize our Machine Learning model, while we keep the test set until the very end to evaluate the final model.

## **Training and Selecting a Predictive Model**

Many different machine learning algorithms have been developed to solve different problem tasks. An important point that can be summarized from David Wolpert's famous 'No Free Lunch Theorems' is that we can't get learning 'for free' ('The Lack of A Priori Distinctions Between Learning Algorithms', D.H. Wolpert 1996; 'No Free Lunch Theorems for Optimization', D.H. Wolpert and W.G. Macready, 1997). Intuitively, we can relate this concept to the popular saying, "I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail" (Abraham Maslow,

1966). For example, each classification algorithm has its inherent biases, and no single classification model enjoys superiority. In practice, it is therefore essential to compare at least a handful of different algorithms in order to train and select the best performing model. But before we can compare different models, we first have to decide upon a metric to measure performance. One commonly used metric is classification accuracy, which is defined as the proportion of correctly classified instances.

One legitimate question to ask is: how do we know which model performs well on the final test dataset and real-world data if we don't use this test set for the model selection, but keep it for the final model evaluation? In order to address the issue embedded in this question, different cross-validation techniques can be used where the training dataset is further divided into training and validation subsets, in order to estimate the generalization performance of the model. Finally, we also cannot expect that the default parameters of the different learning algorithms provided by software libraries will be optimal for our specific problem task. Therefore, we will make frequent use of hyperparameter optimization techniques that help us fine-tune the performance of our model. Intuitively, we can think of those hyperparameters as parameters that are not learned from the data, but represent the knobs of a model that we can turn to improve its performance, which will become much clearer when working with actual examples.

## **Evaluating Models and Predicting Unseen Data Instances**

After we have selected a model that has been fitted on the training dataset, we can use the test dataset to estimate how well it performs on this unseen data to estimate the generalization error. If we are satisfied with its performance, we can now use this model to predict new, future data. It is important to note that the parameters for the previously mentioned procedures—such as feature scaling and dimensionality reduction—are solely obtained from the training dataset, and the same parameters are later reapplied to transform

the test dataset, as well as any new data samples—the performance measured on the test data may be overoptimistic otherwise.

## **Using Python for Machine Learning**

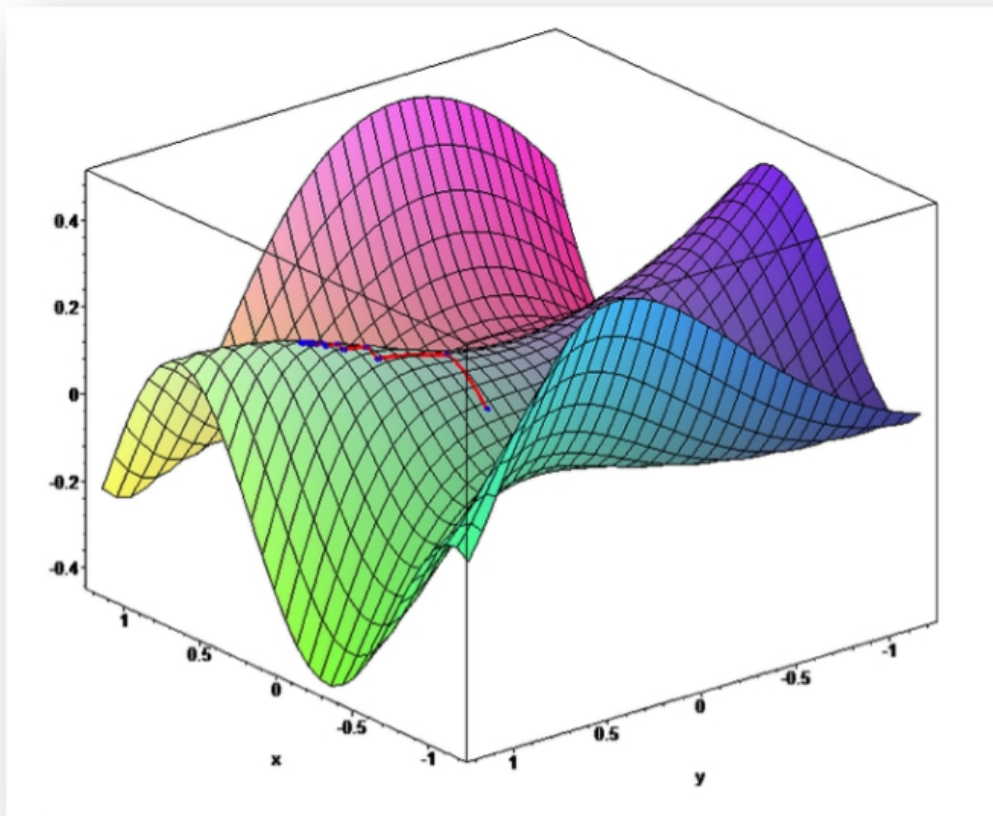
Python is one of the most popular programming languages for data science and therefore enjoys a large number of useful add-on libraries developed by its great community.

Although the performance of interpreted languages, such as Python, for computation-intensive tasks is inferior to lower-level programming languages, extension libraries such as NumPy and SciPy have been developed that build upon lower layer Fortran and C implementations for fast and vectorized operations on multidimensional arrays.

For Machine Learning programming tasks, we mostly refer to the Scikit-learn library, which is one of the most popular and accessible open source machine learning libraries as of today.

# Training Machine Learning Algorithms

In this chapter, we will make use of one of the first algorithmically described Machine Learning algorithms for classification, the 'perceptron' and 'adaptive linear neurons'. We will start by implementing a perceptron step-by-step in Python and training it to classify different flower species in the Iris dataset. This will help us to understand the concept of Machine Learning algorithms for classification, and how they can be efficiently implemented in Python.



## Artificial Neurons – a Brief Glimpse into the Early History of Machine Learning

Before we discuss the perceptron and related algorithms in more detail, let us take a brief tour through the early beginnings of Machine Learning. Trying to understand how the biological brain works to design artificial intelligence, Warren McCulloch and Walter Pitts published the first concept of a simplified brain cell, the so-called McCulloch-Pitts (MCP) neuron, in 1943 (W. S. McCulloch and W. Pitts. 'A Logical Calculus of the Ideas Immanent in Nervous Activity'. *The Bulletin of Mathematical Biophysics* , 5(4):115–133, 1943). Neurons are interconnected nerve cells in the brain that are involved in the processing and transmitting of chemical and electrical signals, which is illustrated in the following figure:

McCulloch and Pitts described such a nerve cell as a simple logic gate with binary outputs; multiple signals arrive at the dendrites, are then integrated into the cell body, and, if the accumulated signal exceeds a certain threshold, an output signal is generated that will be passed on by the axon.

Only a few years later, Frank Rosenblatt published the first concept of the perceptron learning rule based on the MCP neuron model (F. Rosenblatt, 'The Perceptron, a Perceiving and Recognizing Automaton'. Cornell Aeronautical Laboratory, 1957). With his perceptron rule, Rosenblatt proposed an algorithm that would automatically learn the optimal weight coefficients that are then multiplied with the input features in order to make the decision of whether a neuron fires or not. In the context of supervised learning and classification, such an algorithm could then be used to predict if a sample belonged to one class or the other.

More formally, we can pose this problem as a binary classification task, where we refer to our two classes as 1 (positive class) and -1 (negative class) for simplicity. We can then define an activation function  $\phi(\cdot)$  that takes a linear combination of certain input values  $x$  and a corresponding weight vector  $w$ , where  $z$  is the so-called net input ( $z = w \cdot x$ ):

$$z = w_1 x_1 + w_2 x_2 + \dots + w_m x_m$$



Now, if the activation of a particular sample  $x^{(i)}$ , that is, the output of  $\phi(z)$ , is greater than a defined threshold  $\theta$ , we predict class 1 and if it's not greater than the threshold, we predict class -1. In the perceptron algorithm, the activation function  $\phi(\cdot)$  is a simple unit step function, which is sometimes also called the Heaviside step function:

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq \theta \\ -1 & \text{otherwise} \end{cases}$$

For simplicity, we can bring the threshold  $\theta$  to the left side of the equation and define a weight-zero as  $w_0 = -\theta$  and  $x_0 = 1$ , so that we write  $z$  in a more compact form  $z = w_0 x_0 + w_1 x_1 + \dots + w_m x_m = w^T x$  and  $\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases}$ .

In the following sections, we will often make use of basic notations from linear algebra. For example, we will abbreviate the sum of the products of the values in  $x$  and  $w$  using a vector dot product, whereas superscript  $T$  stands for transpose, which is an operation that transforms a column vector into a row vector, and vice versa:  $z = w_0 x_0 + w_1 x_1 + \dots + w_m x_m = \sum_{j=0}^m x_j w_j = w^T x$

$$\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 4 \\ 6 \end{bmatrix}$$

For example:  $\begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \cdot \begin{bmatrix} 5 \\ 4 \\ 6 \end{bmatrix} = 1 \cdot 5 + 2 \cdot 4 + 3 \cdot 6 = 32$ .

$$\begin{bmatrix} 5 & 4 & 6 \end{bmatrix}^T$$

Furthermore, the transpose operation can also be applied to a matrix to reflect it over its diagonal, for example:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}^T$$

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 3 & 6 & 4 \end{bmatrix}^T = \begin{bmatrix} 5 & 6 & 3 & 4 \end{bmatrix}$$

In this book, we will only use the very basic concepts from linear algebra. However, if you need a quick refresher, please take a look at Zico Kolter's excellent *Linear Algebra Review and Reference*,

which is freely available at [http://www.cs.cmu.edu/~zkolter/course/linalg/linalg\\_notes.pdf](http://www.cs.cmu.edu/~zkolter/course/linalg/linalg_notes.pdf).

The following figure illustrates how the net input  $z = w \cdot x^T$  is squashed into a binary output (-1 or 1) by the activation function of the perceptron (left subfigure) and how it can be used to discriminate between two linearly separable classes (right subfigure):

The whole idea behind the MCP neuron and Rosenblatt's thresholded perceptron model is to use a reductionist approach to mimic how a single neuron in the brain works: it either fires or it doesn't. Thus, Rosenblatt's initial perceptron rule is fairly simple, and can be summarized by the following steps:

- Initialize the weights to 0 or small random numbers.
- For each training sample  $x^{(i)}$ , perform the following steps:
  - Compute the output value  $\hat{y}^{(i)}$
  - Update the weights

Here, the output value is the class label predicted by the unit step function that we defined earlier, and the simultaneous update of each weight  $w_j$  in the weight vector

$w$  can be more formally written as:

$$w_j := w_j + \Delta w_j$$

The value of  $\Delta w_j$ , which is used to update the weight  $w_j$ , is calculated by the perceptron learning rule:

$$\Delta w_j = \eta (y^{(i)} - \hat{y}^{(i)}) x_j^{(i)}$$

Where  $\eta$  is the learning rate (a constant between 0.0 and 1.0),  $y^{(i)}$  is the true class label of the  $i$ th training sample, and  $\hat{y}^{(i)}$  is the predicted class label. It is important to note that all weights in the weight vector are being updated simultaneously, which means that we don't recompute the  $\hat{y}^{(i)}$  until all of the weights  $\Delta w_j$  are updated. Concretely, for a 2D dataset, we would write the update as follows:

$$\Delta w_0 = \eta(y_i - \text{output}(i))$$

$$\Delta w_1 = \eta(y_i - \text{output}(i))x_1(i)$$

$$\Delta w_2 = \eta(y_i - \text{output}(i))x_2(i)$$

Before we implement the perceptron rule in Python, here's a simple thought experiment to illustrate how beautifully simple this learning rule really is. In the two scenarios where the perceptron predicts the class label correctly, the weights remain unchanged:

$$\Delta w_j = \eta(-1 - 1)x(j) = 0$$

$$\Delta w_j = \eta(1 - 1)x(j) = 0$$

However, in the case of a wrong prediction, the weights are being pushed towards the direction of the positive or negative target class, respectively:

$$\Delta w_j = \eta(1 - (-1))x(j) = \eta(2)x(j)$$

$$\Delta w_j = \eta(-1 - 1)x(j) = \eta(-2)x(j)$$

To gain better intuition for the multiplicative factor  $x(j)$ , let us go through another simple example, where:

$$y(i) = +1, \hat{y}(j) = -1, \eta = 1$$

Let's assume that  $x(j) = 0.5$ , and we misclassify this sample as -1. In this case, we would increase the corresponding weight by 1, so that the activation  $x(j) \times w(j)$  will be more positive the next time we encounter this sample, and will be more likely to be above the threshold of the unit step function to classify the sample as +1:

$$\Delta = w(j)(1 - (-1))0.5 = (2)0.5 = 1$$

The weight update is proportional to the value of  $x(j)$ . For example, if we have another sample  $x(j) = 2$  that is incorrectly classified as -1, we'd push the decision boundary by an even larger extent to classify this sample correctly the next time:

$$\Delta = w_j(1 - (-1))2 = (2)2 = 4$$

It is important to note that the convergence of the perceptron is only guaranteed if the two classes are linearly separable and the learning rate is sufficiently small. If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset (epochs) and/or a threshold for the number of tolerated misclassifications—otherwise, the perceptron would never stop updating the weights

Now, before we jump into implementation in the next section, let's summarize what we just learned in a simple figure that illustrates the general concept of the perceptron:

The preceding figure illustrates how the perceptron receives the inputs of a sample  $x$  and combines them with the weights  $w$  to compute the net input. The net input

is then passed on to the activation function (here: the unit step function), which generates a binary output  $-1$  or  $+1$ —the predicted class label of the sample. During the learning phase, this output is used to calculate the error of the prediction and update the weights.

## **Implementing a Perceptron Learning Algorithm in Python**

In the previous section, we learned how Rosenblatt's perceptron rule works; let's go ahead and implement it in Python, and apply it to the Iris dataset that we introduced earlier. We will take an objected-oriented approach to define the perceptron interface as a Python Class, which allows us to initialize new perceptron objects that can learn from data via a fit method, and make predictions via a separate predict method. As a convention, we add an underscore to attributes that are not being created upon the initialization of the object, but by calling the object's other methods—for example, `self.w_`.

If you are not yet familiar with Python's scientific libraries or need a refresher, please see the following resources:

NumPy: [http://wiki.scipy.org/Tentative\\_NumPy\\_Tutorial](http://wiki.scipy.org/Tentative_NumPy_Tutorial)

Pandas: <http://pandas.pydata.org/pandas-docs/stable/tutorials.html>

Matplotlib: <http://matplotlib.org/users/beginner.html>

Also, to better follow the code examples, I recommend you download the IPython notebooks from the Packt website. For a general introduction to IPython notebooks, please visit <https://ipython.org/ipython-doc/3/notebook/index.html>.

```
import numpy as np    class Perceptron(object):    """Perceptron classifier.
```

Parameters

----- eta : float

Learning rate (between 0.0 and 1.0) n\_iter : int Passes over the training dataset.

Attributes

----- w\_ : 1d-array

Weights after fitting.

errors\_ : list Number of misclassifications in every epoch.

```
"""    def __init__(self, eta=0.01, n_iter=10):
```

```
    self.eta = eta    self.n_iter = n_iter
```

```
def fit(self, X, y):    """Fit training data.
```

Parameters

-----

X : {array-like}, shape = [n\_samples, n\_features]

Training vectors, where n\_samples is the number of samples and n\_features is the number of features.

y : array-like, shape = [n\_samples] Target values.

Returns ----- self : object

```
""" "
```

```

self.w_ = np.zeros(1 + X.shape[1])    self.errors_ = []

for _ in range(self.n_iter):
    errors = 0
    for xi, target in zip(X, y):
        update = self.eta * (target - self.predict(xi))
        self.w_[1:] += update * xi
        errors += int(update != 0.0)
    self.errors_.append(errors)
    return self

def net_input(self, X):
    """Calculate net input"""
    return np.dot(X, self.w_[1:]) + self.w_[0]

def predict(self, X):
    """Return class label after unit step"""
    return np.where(self.net_input(X) >= 0.0, 1, -1)

```

Using this perceptron implementation, we can now initialize new perceptron objects with a given learning rate `eta` and `n_iter`, which is the number of epochs (passes over the training set). Via the `fit` method, we initialize the weights in `self.w_` to a zero-vector  $m+1$  where  $m$  stands for the number of dimensions (features) in the dataset where we add 1 for the zero-weight (that is, the threshold).

NumPy indexing for one-dimensional arrays works similarly to Python lists, using the square-bracket (`[]`) notation. For two-dimensional arrays, the first indexer refers to the row number, and the second indexer to the column number. For example, we would use `X[2, 3]` to select the third row and fourth column of a 2D array `X`.

After the weights have been initialized, the `fit` method loops over all individual samples in the training set and updates the weights according to the perceptron learning rule that we discussed in the previous section. The class labels are predicted by the `predict` method, while we also called in the `fit` method to predict the class label for the weight update, but `predict` can also be used to predict the class labels of new data after we have fitted our model. Furthermore, we also collect the number of misclassifications during each epoch in the list `self.errors_` so that we can later analyze how

well our perceptron performed during the training. The `np.dot` function that is used in the `net_input` method simply calculates the vector dot product  $w \cdot X_t$ .

## Conclusion

There is no doubt that Python is one of the best suited programming languages when it comes to data science. It has been said, time and again, that Python is the one of the most common programming languages. But often, the question of why one should study this language comes into play.

Data science has a great future, and there will be many jobs in the future, as the demand for data scientists is increasing day by day. So, one can dream of their future in this field. You should learn Python if you want to venture into the field of data science, because Python is a flexible language; it is free and powerful, along with being an open source language.

Python cuts development time in half with its simplicity, as well as making it easy to read the syntax. With the help of Python, one can manipulate and analyze your data, as well as carrying out data visualization. Python provides libraries that are essential for the applications of Machine Learning, as well as other scientific processing of data.

The best part about learning Python is that it is a high-level language that is quite easy to learn, and is procedure oriented along with being object oriented.

Learning Python Data Science is not a tough task, even for beginners. So, do not hesitate to take the leap and master **Python Data Science**.

*To your success!*