CHAPTER 1

Introduction to Data Science with Python

The amount of digital data that exists is growing at a rapid rate, doubling every two years, and changing the way we live. It is estimated that by 2020, about 1.7MB of new data will be created every second for every human being on the planet. This means we need to have the technical tools, algorithms, and models to clean, process, and understand the available data in its different forms for decision-making purposes. *Data science* is the field that comprises everything related to cleaning, preparing, and analyzing unstructured, semistructured, and structured data. This field of science uses a combination of statistics, mathematics, programming, problem-solving, and data capture to extract insights and information from data.

The Stages of Data Science

Figure 1-1 shows different stages in the field of data science. Data scientists use programming tools such as Python, R, SAS, Java, Perl, and C/C++ to extract knowledge from prepared data. To extract this information, they employ various fit-to-purpose models based on machine leaning algorithms, statistics, and mathematical methods.

O. Embarak, *Data Analysis and Visualization Using Python*,
https://doi.org/10.1007/978-1-4842-4109-7_1

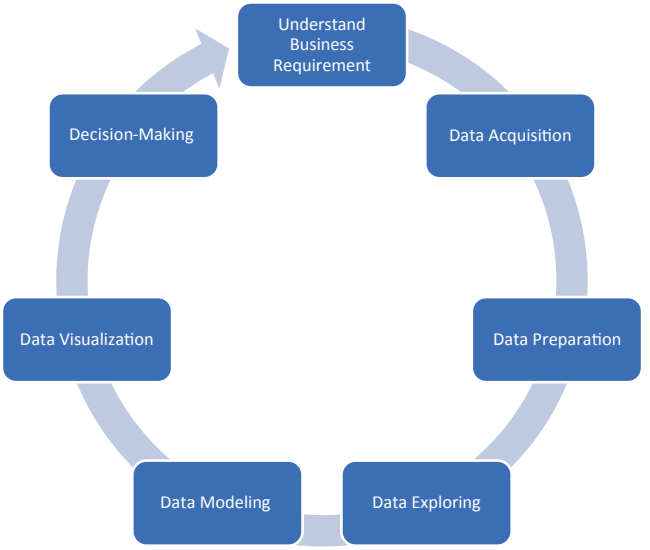


Figure 1-1. *Data science project stages*

Data science algorithms are used in products such as internet search engines to deliver the best results for search queries in less time, in recommendation systems that use a user's experience to generate recommendations, in digital advertisements, in education systems, in healthcare systems, and so on. Data scientists should have in-depth knowledge of programming tools such as Python, R, SAS, Hadoop platforms, and SQL databases; good knowledge of semistructured formats such as JSON, XML, HTML. In addition to the knowledge of how to work with unstructured data.

Why Python?

Python is a dynamic and general-purpose programming language that is used in various fields. Python is used for everything from throwaway scripts to large, scalable web servers that provide uninterrupted service 24/7. It is used for GUI and database programming, client- and server-side

web programming, and application testing. It is used by scientists writing applications for the world's fastest supercomputers and by children first learning to program. It was initially developed in the early 1990s by Guido van Rossum and is now controlled by the not-for-profit Python Software Foundation, sponsored by Microsoft, Google, and others.

The first-ever version of Python was introduced in 1991. Python is now at version 3.*x*, which was released in February 2011 after a long period of testing. Many of its major features have also been backported to the backward-compatible Python 2.6, 2.7, and 3.6.

Basic Features of Python

Python provides numerous features; the following are some of these important features:

- *Easy to learn and use*: Python uses an elegant syntax, making the programs easy to read. It is developer-friendly and is a high-level programming language.
- *Expressive*: The Python language is expressive, which means it is more understandable and readable than other languages.
- *Interpreted*: Python is an interpreted language. In other words, the interpreter executes the code line by line. This makes debugging easy and thus suitable for beginners.
- *Cross-platform*: Python can run equally well on different platforms such as Windows, Linux, Unix, Macintosh, and so on. So, Python is a portable language.
- *Free and open source*: The Python language is freely available at `www.python.org`. The source code is also available.

- *Object-oriented*: Python is an object-oriented language with concepts of classes and objects.
- *Extensible*: It is easily extended by adding new modules implemented in a compiled language such as C or C++, which can be used to compile the code.
- *Large standard library*: It comes with a large standard library that supports many common programming tasks such as connecting to web servers, searching text with regular expressions, and reading and modifying files.
- *GUI programming support*: Graphical user interfaces can be developed using Python.
- *Integrated*: It can be easily integrated with languages such as C, C++, Java, and more.

Python Learning Resources

Numerous amazing Python resources are available to train Python learners at different learning levels. There are so many resources out there, though it can be difficult to know how to find all of them. The following are the best general Python resources with descriptions of what they provide to learners:

- *Python Practice Book* is a book of Python exercises to help you learn the basic language syntax. (See `https://anandology.com/python-practice-book/index.html`.)
- *Agile Python Programming*: *Applied for Everyone* provides a practical demonstration of Python programming as an agile tool for data cleaning, integration, analysis, and visualization fits for academics, professionals, and

researchers. (See http://www.lulu.com/shop/ossama-embarak/agile-python-programming-applied-for-everyone/paperback/product-23694020.html.)

- "A Python Crash Course" gives an awesome overview of the history of Python, what drives the programming community, and example code. You will likely need to read this in combination with other resources to really let the syntax sink in, but it's a great resource to read several times over as you continue to learn. (See https://www.grahamwheeler.com/posts/python-crash-course.html.)
- "A Byte of Python" is a beginner's tutorial for the Python language. (See https://python.swaroopch.com/.)
- The O'Reilly book *Think Python*: *How to Think Like a Computer Scientist* is available in HTML form for free on the Web. (See https://greenteapress.com/wp/think-python/.)
- *Python for You and Me* is an approachable book with sections for Python syntax and the major language constructs. The book also contains a short guide at the end teaching programmers to write their first Flask web application. (See https://pymbook.readthedocs.io/en/latest/.)
- Code Academy has a Python track for people completely new to programming. (See www.codecademy.com/catalog/language/python.)
- *Introduction to Programming with Python* goes over the basic syntax and control structures in Python. The free book has numerous code examples to go along with each topic. (See www.opentechschool.org/.)

- Google has a great compilation of material you should read and learn from if you want to be a professional programmer. These resources are useful not only for Python beginners but for any developer who wants to have a strong professional career in software. (See techdevguide.withgoogle.com.)
- Looking for ideas about what projects to use to learn to code? Check out the five programming projects for Python beginners at knightlab.northwestern.edu.
- There's a Udacity course by one of the creators of Reddit that shows how to use Python to build a blog. It's a great introduction to web development concepts. (See mena.udacity.com.)

Python Environment and Editors

Numerous integrated development environments (IDEs) can be used for creating Python scripts.

Portable Python Editors (No Installation Required)

These editors require no installation:

Azure Jupyter Notebooks: The open source Jupyter Notebooks was developed by Microsoft as an analytic playground for analytics and machine learning.

Python(x,y): Python(x,y) is a free scientific and engineering development application for numerical computations, data analysis, and data visualization based on the Python programming language, Qt graphical user interfaces, and Spyder interactive scientific development environment.

WinPython: This is a free Python distribution for the Windows platform; it includes prebuilt packages for ScientificPython.

Anaconda: This is a completely free enterprise-ready Python distribution for large-scale data processing, predictive analytics, and scientific computing.

PythonAnywhere: PythonAnywhere makes it easy to create and run Python programs in the cloud. You can write your programs in a web-based editor or just run a console session from any modern web browser.

Anaconda Navigator: This is a desktop graphical user interface (GUI) included in the Anaconda distribution that allows you to launch applications and easily manage Anaconda packages (as shown in Figure 1-2), environments, and channels without using command-line commands. Navigator can search for packages on the Anaconda cloud or in a local Anaconda repository. It is available for Windows, macOS, and Linux.

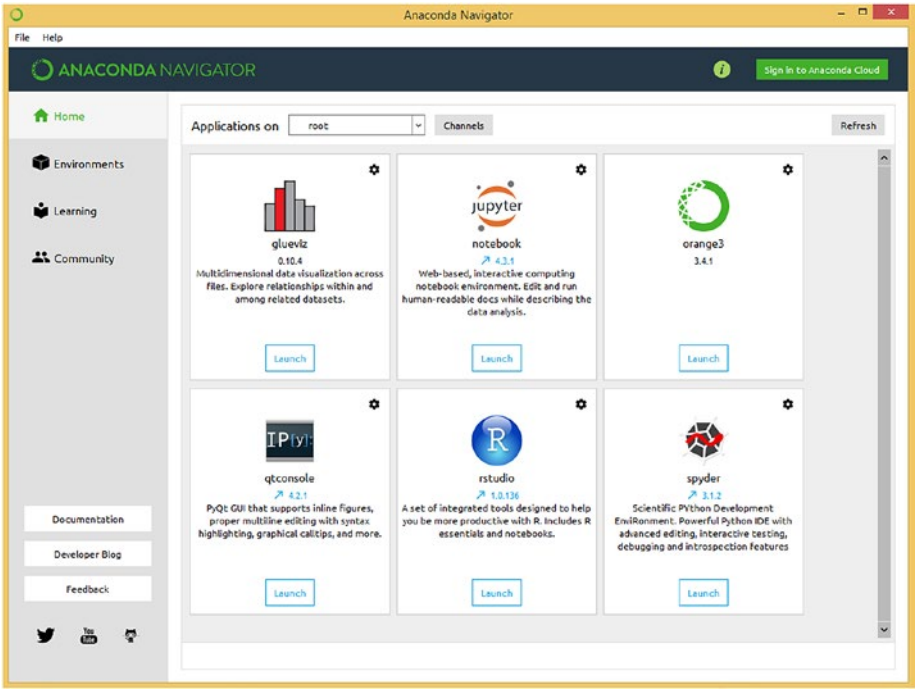


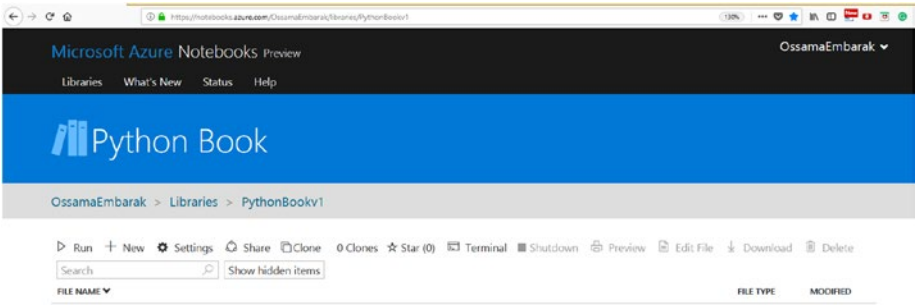
Figure 1-2. *Anaconda Navigator*

The following sections demonstrate how to set up and use Azure Jupyter Notebooks.

Azure Notebooks

The Azure Machine Learning workbench supports interactive data science experimentation through its integration with Jupyter Notebooks.

Azure Notebooks is available for free at `https://notebooks.azure.com/`. After registering and logging into Azure Notebooks, you will get a menu that looks like this:



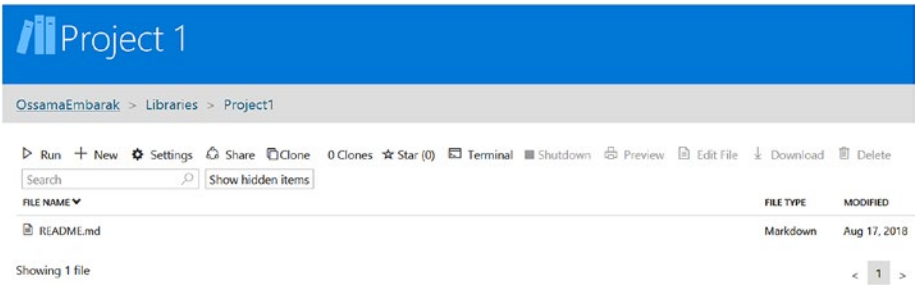
Once you have created your account, you can create a library for any Python project you would like to start. All libraries you create can be displayed and accessed by clicking the Libraries link.

Let's create a new Python script.

1. Create a library.

Click New Library, enter your library details, and click Create, as shown here:

A new library is created, as shown in Figure 1-3.



2. Create a project folder container.

   Organizing the Python library scripts is important. You can create folders and subfolders by selecting +New from the ribbon; then for the item type select Folder, as shown in Figure 1-3.

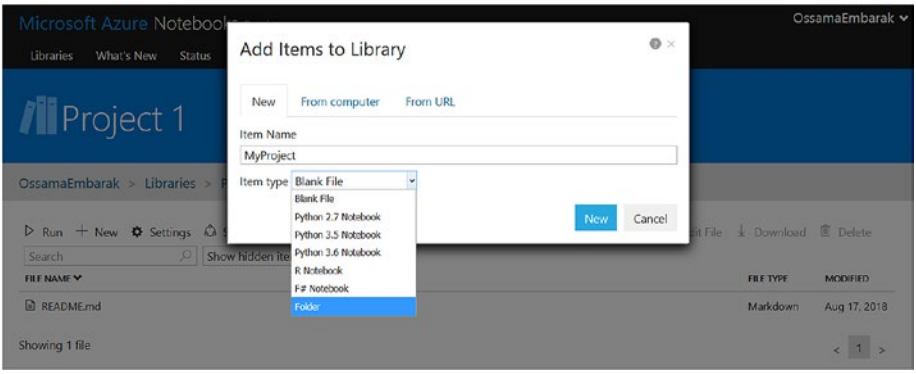
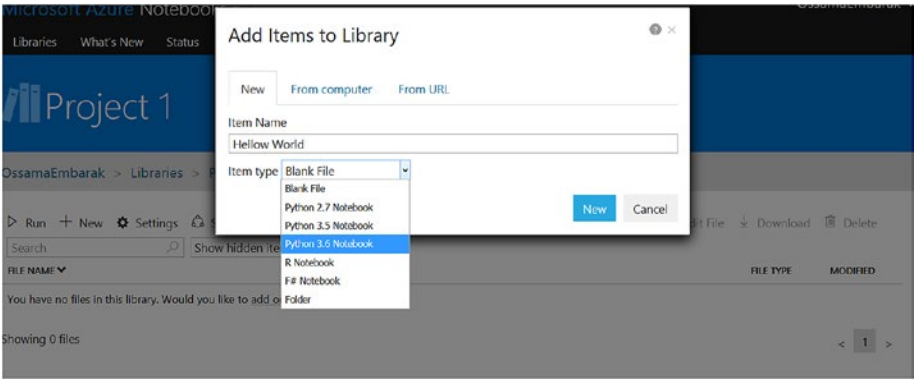


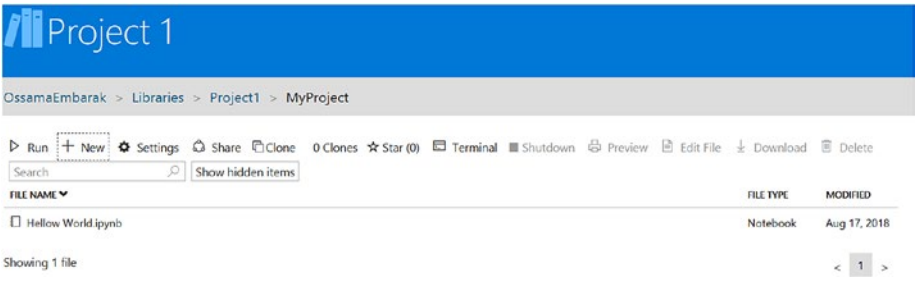
Figure 1-3. *Creating a folder in an Azure project*

3. Create a Python project.

Move inside the created folder and create a new Python project.



Your project should look like this:



4. Write and run a Python script.

Open the Created Hello World script by clicking it, and start writing your Python code, as shown in Figure 1-4.

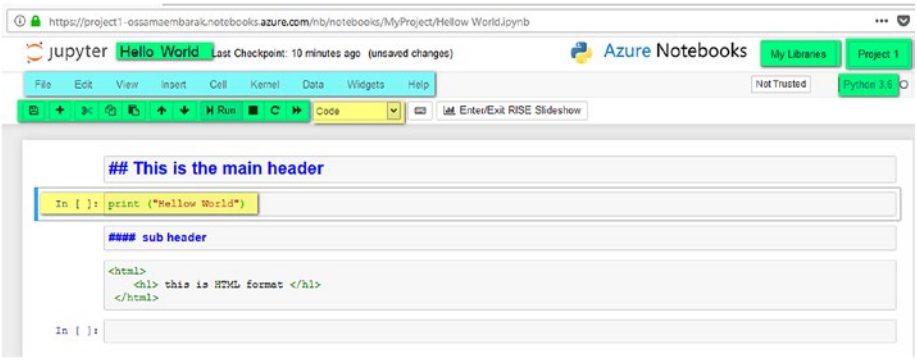
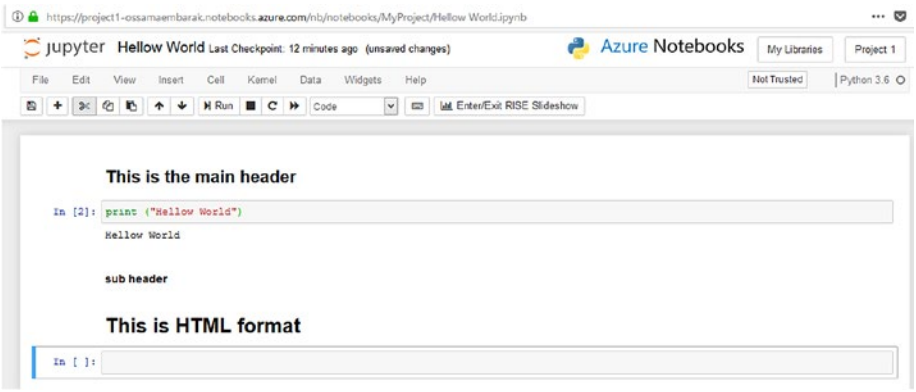


Figure 1-4. *A Python script file on Azure*

In Figure 1-4, all the green icons show the options that can be applied on the running file. For instance, you can click + to add new lines to your file script. Also, you can save, cut, and move lines up and down. To execute any segment of code, press Ctrl+Enter, or click Run on the ribbon.



Offline and Desktop Python Editors

There are many offline Python IDEs such as Spyder, PyDev via Eclipse, NetBeans, Eric, PyCharm, Wing, Komodo, Python Tools for Visual Studio, and many more.

The following steps demonstrate how to set up and use Spyder. You can download Anaconda Navigator and then run the Spyder software, as shown in Figure 1-5.

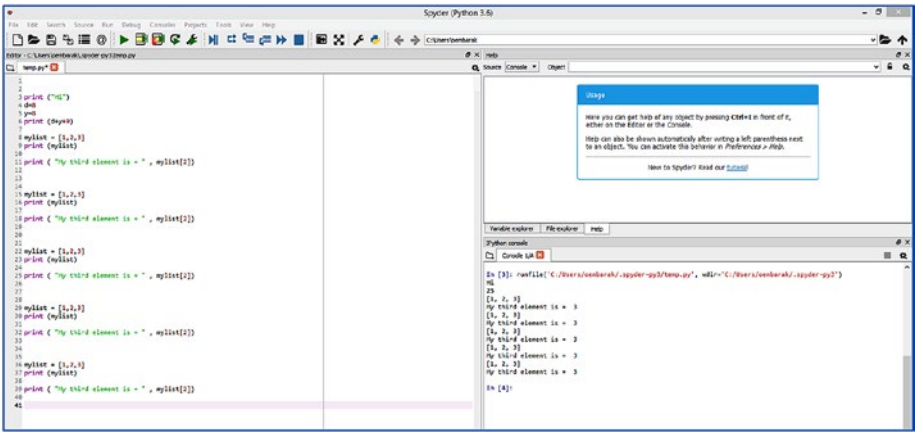


Figure 1-5. *Python Spyder IDE*

On the left side, you can write Python scripts, and on the right side you can see the executed script in the console.

The Basics of Python Programming

This section covers basic Python programming.

Basic Syntax

A Python *identifier* is a name used to identify a variable, function, class, module, or other object in the created script. An identifier starts with a letter from *A* to *Z* or from *a* to *z* or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).

Python does not allow special characters such as @, $, and % within identifiers. Python is a case-sensitive programming language. Thus, `Manpower` and `manpower` are two different identifiers in Python.

The following are the rules for naming Python identifiers:

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

The `help?` method can be used to get support from the Python user manual, as shown in Listing 1-1.

Listing 1-1. Getting Help from Python

```
In [3]:      help?

Signature:   help(*args, **kwds)
Type:        _Helper
String form: Type help() for interactive help, or help(object)
for help about object.
Namespace:   Python builtin
```

```
File:        ~/anaconda3_501/lib/python3.6/_sitebuiltins.py
Docstring:
Define the builtin 'help'.

This is a wrapper around pydoc.help that provides a helpful
message
when 'help' is typed at the Python interactive prompt.

Calling help() at the Python prompt starts an interactive help
session.
Calling help(thing) prints help for the python object 'thing'.
```

The smallest unit inside a given Python script is known as a *token*, which represents punctuation marks, reserved words, and each individual word in a statement, which could be keywords, identifiers, literals, and operators.

Table 1-1 lists the reserved words in Python. Reserved words are the words that are reserved by the Python language already and can't be redefined or declared by the user.

Table 1-1. *Python Reserved Keywords*

and	exec	not	continue	global	with	yield	in
assert	finally	or	def	if	return	else	is
break	for	pass	except	lambda	while	try	
class	from	print	del	import	raise	elif	

Lines and Indentation

Line indentation is important in Python because Python does not depend on braces to indicate blocks of code for class and function definitions or flow control. Therefore, a code segment block is denoted by line indentation, which is rigidly enforced, as shown in Listing 1-2.

Listing 1-2. Line Indentation Syntax Error

```
In [4]:age, mark, code=10,75,"CIS2403"
       print (age)
       print (mark)
            print (code)
```

```
File "<ipython-input-4-5e544bb51da0>", line 4
print (code)
IndentationError: unexpected indent
```

Multiline Statements

Statements in Python typically end with a new line. But a programmer can use the line continuation character (\) to denote that the line should continue, as shown in Listing 1-3. Otherwise, a syntax error will occur.

Listing 1-3. Multiline Statements

```
In [5]:TV=15
            Mobile=20 Tablet = 30
total = TV +
Mobile +
      Tablet
print (total)
```

```
File "<ipython-input-5-68bc7095f603>", line 5
total = TV +
SyntaxError: invalid syntax
```

The following is the correct syntax:

```
In [6]: TV=15
       Mobile=20
       Tablet = 30
       total = TV + \
```

```
        Mobile + \
        Tablet
        print (total)
```

```
65
```

The code segment with statements contained within the [], {}, or () brackets does not need to use the line continuation character, as shown in Listing 1-4.

Listing 1-4. Statements with Quotations

```
In [7]: days = ['Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday']
print (days)
```

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Quotation Marks in Python

Python accepts single ('), double ("), and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string. However, triple quotes are used to span the string across multiple lines, as shown in Listing 1-5.

Listing 1-5. Quotation Marks in Python

```
In [8]:sms1 = 'Hellow World'
       sms2 = "Hellow World"
       sms3 = """ Hellow World"""
       sms4 = """ Hellow
             World"""
       print (sms1)
       print (sms2)
       print (sms3)
       print (sms4)
```

```
Hellow World
Hellow World
Hellow World
Hellow
World
```

Multiple Statements on a Single Line

Python allows the use of \n to split line into multiple lines. In addition, the semicolon (;) allows multiple statements on a single line if neither statement starts a new code block, as shown in Listing 1-6.

Listing 1-6. The Use of the Semicolon and New Line Delimiter

```
In [9]: TV=15; name="Nour"; print (name); print ("Welcome
to\nDubai Festival 2018")
```

```
Nour
Welcome to
Dubai Festival 2018
```

Read Data from Users

The line code segment in Listing 1-7 prompts the user to enter a name and age, converts the age into an integer, and then displays the data.

Listing 1-7. Reading Data from the User

```
In [10]:name = input("Enter your name ")
       age = int (input("Enter your age "))
       print ("\nName =", name); print ("\nAge =", age)
```

```
Enter your name Nour
Enter your age 12

Name = Nour

Age = 12
```

Declaring Variables and Assigning Values

There is no restriction to declaring explicit variables in Python. Once you assign a value to a variable, Python considers the variable according to the assigned value. If the assigned value is a string, then the variable is considered a string. If the assigned value is a real, then Python considers the variable as a double variable. Therefore, Python does not restrict you to declaring variables before using them in the application. It allows you to create variables at the required time.

Python has five standard data types that are used to define the operations possible on them and the storage method for each of them.

- Number
- String
- List
- Tuple
- Dictionary

The equal (=) operator is used to assign a value to a variable, as shown in Listing 1-8.

Listing 1-8. Assign Operator

```
In [11]: age = 11
      name ="Nour"
      tall=100.50
In [12]: print (age)
      print (name)
      print (tall)
```

```
11
Nour
100.5
```

Multiple Assigns

Python allows you to assign a value to multiple variables in a single statement, which is also known as *multiple assigns*. You can assign a single value to multiple variables or assign multiple values to multiple variables, as shown in Listing 1-9.

Listing 1-9. Multiple Assigns

```
In [13]:age= mark = code =25
      print (age)
      print (mark)
      print (code)
```

```
25
25
25
```

```
In [14]:age, mark, code=10,75,"CIS2403"
      print (age)
      print (mark)
      print (code)
```

```
10
75
CIS2403
```

Variable Names and Keywords

A *variable* is an identifier that allocates specific memory space and assigns a value that could change during the program runtime. Variable names should refer to the usage of the variable, so if you want to create a variable for student age, then you can name it as `age` or `student_age`. There are many rules and restrictions for variable names. It's not allowed to use special characters or white spaces in variable naming. For instance, variable names shouldn't start with any special character and shouldn't be any of the Python reserved keywords. The following example shows incorrect naming: {`?age, 1age, age student, and, if, 1_age, etc`}. The following shows correct naming for a variable: {`age, age1, age_1, if_age, etc`}.

Statements and Expressions

A *statement* is any unit of code that can be executed by a Python interpreter to get a specific result or perform a specific task. A program contains a sequence of statements, each of which has a specific purpose during program execution. The expression is a combination of values, variables, and operators that are evaluated by the interpreter to do a specific task, as shown in Listing 1-10.

Listing 1-10. Expression and Statement Forms

```
In [16]:# Expressions
      x=0.6              # Statement
      x=3.9 * x * (1-x)  # Expressions
      print (round(x, 2))
```

```
0.94
```

Basic Operators in Python

Operators are the constructs that can manipulate the value of operands. Like different programming languages, Python supports the following operators:

- Arithmetic operators
- Relational operators
- Assign operators
- Logical operators
- Membership operators
- Identity operators
- Bitwise operators

Arithmetic Operators

Table 1-2 shows examples of arithmetic operators in Python.

Table 1-2. *Python Arithmetic Operators*

Operators	Description	Example	Output
`//`	Performs floor division (gives the integer value after division)	`print (13//5)`	2
`+`	Performs addition	`print (13+5)`	18
`-`	Performs subtraction	`print (13-5)`	8
`*`	Performs multiplication	`print (2*5)`	10
`/`	Performs division	`print (13/5)`	2.6
`%`	Returns the remainder after division (modulus)	`print (13%5)`	3
`**`	Returns an exponent (raises to a power)	`print (2**3)`	8

Relational Operators

Table 1-3 shows examples of relational operators in Python.

Table 1-3. *Python Relational Operators*

Operators	Description	Example	Output
<	Less than	`print (13<5)`	False
>	Greater than	`print (13>5)`	True
<=	Less than or equal to	`print (13<=5)`	False
>=	Greater than or equal to	`print (2>=5)`	False
==	Equal to	`print (13==5)`	False
!=	Not equal to	`print (13! =5)`	True

Assign Operators

Table 1-4 shows examples of assign operators in Python.

Table 1-4. *Python Assign Operators*

Operators	Description	Example	Output
=	Assigns	`x=10` `print (x)`	10
/=	Divides and assigns	`x=10; x/=2` `print (x)`	5.0
+=	Adds and assigns	`x=10; x+=7` `print (x)`	17
-=	Subtracts and assigns	`x=10; x-=6` `print (x)`	4

(*continued*)

Table 1-4. (*continued*)

Operators	Description	Example	Output
*=	Multiplies and assigns	`x=10; x*=5` `print (x)`	50
%=	Modulus and assigns	`x=13; x%=5` `print (x)`	3
**=	Exponent and assigns	`x=10; x**=3` `print(x)`	1000
//=	Floor division and assigns	`x=10; x//=2` `print(x)`	5

Logical Operators

Table 1-5 shows examples of logical operators in Python.

Table 1-5. *Python Logical Operators*

Operators	Description	Example	Output
and	Logical AND (when both conditions are true, the output will be true)	`x=10>5 and 4>20` `print (x)`	False
or	Logical OR (if any one condition is true, the output will be true)	`x=10>5 or 4>20` `print (x)`	True
not	Logical NOT (complements the condition; i.e., reverses it)	`x=not (10<4)` `print (x)`	True

A Python *program* is a sequence of Python statements that have been crafted to do something. It can be one line of code or thousands of code segments written to perform a specific task by a computer. Python statements are executed immediately and do not wait for the entire

program to be executed. Therefore, Python is an interpreted language that executes line per line. This differs from other languages such as C#, which is a compiled language that needs to handle the entire program.

Python Comments

There are two types of comments in Python: single-line comments and multiline comments.

The # symbol is used for single-line comments.

Multiline comments can be given inside triple quotes, as shown in Listing 1-11.

Listing 1-11. Python Comment Forms

```
In [18]: # Python single line comment
In [19]: ''' This
      Is
      Multi-line comment'''
```

Formatting Strings

The Python special operator % helps to create formatted output. This operator takes two operands, which are a formatted string and a value. The following example shows that you pass a string and the 3.14259 value in string format. It should be clear that the value can be a single value, a tuple of values, or a dictionary of values.

```
In [20]: print ("pi=%s"%"3.14159")

pi=3.14159
```

Conversion Types

You can convert values using different conversion specifier syntax, as summarized in Table 1-6.

Table 1-6. *Conversion Syntax*

Syntax	Description
%c	Converts to a single character
%d, %i	Converts to a signed decimal integer or long integer
%u	Converts to an unsigned decimal integer
%e, %E	Converts to a floating point in exponential notation
%f	Converts to a floating point in fixed-decimal notation
%g	Converts to the value shorter of %f and %e
%G	Converts to the value shorter of %f and %E
%o	Converts to an unsigned integer in octal
%r	Converts to a string generated with `repr()`
%s	Converts to a string using the `str()` function
%x, %X	Converts to an unsigned integer in hexadecimal

For example, the conversion specifier %s says to convert the value to a string. Therefore, to print a numerical value inside string output, you can use, for instance, `print("pi=%s" % 3.14159)`. You can use multiple conversions within the same string, for example, to convert into double, float, and so on.

```
In [1]:print("The value of %s is = %02f" % ("pi", 3.14159))

The value of pi is = 3.141590
```

You can use a dot (.) followed by a positive integer to specify the precision. In the following example, you can use a tuple of different data types and inject the output in a string message:

```
In [21]:print ("Your name is %s, and your height is %.2f while
your weight is %.2d" % ('Ossama', 172.156783, 75.56647))

Your name is Ossama, and your height is 172.16 while your
weight is 75
```

In the previous example, you can see that `%.2f` is replaced with the value 172.16 with two decimal fractions after the decimal point, while `%2d` is used to display decimal values only but in a two-digit format.

You can display values read directly from a dictionary, as shown next, where `%(name)s` says to take as a string the dictionary value of the key `Name` and `%(height).2f` says to take it as a float with two fraction values, which are the dictionary values of the key `height`:

```
In [23]:print ("Hi %(Name)s, your height is %(height).2f"
%{'Name':"Ossama", 'height': 172.156783})

Hi Ossama, your height is 172.16
```

The Replacement Field, {}

You can use the replacement field, {}, as a name (or index). If an index is provided, it is the index of the list of arguments provided in the field. It's not necessary to have indices with the same sequence; they can be in a random order, such as indices 0, 1, and 2 or indices 2, 1, and 0.

```
In [24]:x = "price is"
        print ("{1} {0} {2}".format(x, "The", 1920.345))

The price is 1920.345
```

Also, you can use a mix of values combined from lists, dictionaries, attributes, or even a singleton variable. In the following example, you will create a class called `A()`, which has a single variable called `x` that is assigned the value 9.

Then you create an instance (*object*) called `w` from the class `A()`. Then you print values indexed from variable `{0}` and the `{1[2]}` value from the list of values `["a," "or," "is"]`, where 1 refers to the index of printing and 2 refers to the index in the given list where the string index is 0. `{2[test]}` refers to index 2 in the print string and reads its value from the passed dictionary from the key `test`. Finally, `{3.x}` refers to the third index, which takes its value from `w`, which is an instance of the class `A()`.

```
In [34]:class A():x=9 w=A()
      print ("{0} {1[2]} {2[test]} {3.x}".format("This", ["a",
      "or", "is"], {"test": "another"},w))
```

```
This is another 9
```

```
In [34]:print ("{1[1]} {0} {1[2]} {2[test]}{3.x}".
format("This", ["a", "or", "is"], {"test": "another"},w))
```

```
or This is another 9
```

The Date and Time Module

Python provides a time package to deal with dates and times. You can retrieve the current date and time and manipulate the date and time using the built-in methods.

The example in Listing 1-12 imports the time package and calls its `.localtime()` function to retrieve the current date and time.

Listing 1-12. Time Methods

```
In [42]:import time localtime = time.asctime(time.
localtime(time.time()))
print ("Formatted time :", localtime)
print(time.localtime())
print (time.time())
```

```
Formatted time : Fri Aug 17 19:12:07 2018
```

```
time.struct_time(tm_year=2018, tm_mon=8, tm_mday=17,
tm_hour=19, tm_min=12, tm_sec=7, tm_wday=4, tm_yday=229,
tm_isdst=0)
```

```
1534533127.8304486
```

Time Module Methods

Python provides various built-in time functions, as in Table 1-7, that can be used for time-related purposes.

Table 1-7. *Built-in Time Methods*

Methods	Description
`time()`	Returns time in seconds since January 1, 1970.
`asctime(time)`	Returns a 24-character string, e.g., Sat Jun 16 21:27:18 2018.
`sleep(time)`	Used to stop time for the given interval of time.
`strptime (String,format)`	Returns a tuple with nine time attributes. It receives a string of date and a format. `time.struct_time(tm_year=2018, tm_mon=6, tm_mday=16, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=3, tm_yday=177, tm_isdst=-1)`

(*continued*)

Table 1-7. *(continued)*

Methods	Description
`gtime()/ gtime(sec)`	Returns `struct_time`, which contains nine time attributes.
`mktime()`	Returns the seconds in floating point since the epoch.
`strftime (format)/ strftime (format,time)`	Returns the time in a particular format. If the time is not given, the current time in seconds is fetched.

Python Calendar Module

Python provides a calendar module, as in Table 1-8, which provides many functions and methods to work with a calendar.

Table 1-8. *Built-in Calendar Module Functions*

Methods	Description
`prcal(year)`	Prints the whole calendar of the year.
`firstweekday()`	Returns the first weekday. It is by default 0, which specifies Monday.
`isleap(year)`	Returns a Boolean value, i.e., true or false. Returns true in the case the given year is a leap year; otherwise, false.
`monthcalendar(year,month)`	Returns the given month with each week as one list.
`leapdays(year1,year2)`	Returns the number of leap days from `year1` to `year2`.
`prmonth(year,month)`	Prints the given month of the given year.

You can use the Calendar package to display a 2018 calendar as shown here:

```
In [45]:import calendar
            calendar.prcal(2018)
```

```
                                  2018

      January               February               March
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7                1  2  3  4                1  2  3  4
 8  9 10 11 12 13 14   5  6  7  8  9 10 11   5  6  7  8  9 10 11
15 16 17 18 19 20 21  12 13 14 15 16 17 18  12 13 14 15 16 17 18
22 23 24 25 26 27 28  19 20 21 22 23 24 25  19 20 21 22 23 24 25
29 30 31              26 27 28              26 27 28 29 30 31

      April                 May                   June
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
                   1     1  2  3  4  5  6                1  2  3
 2  3  4  5  6  7  8   7  8  9 10 11 12 13   4  5  6  7  8  9 10
 9 10 11 12 13 14 15  14 15 16 17 18 19 20  11 12 13 14 15 16 17
16 17 18 19 20 21 22  21 22 23 24 25 26 27  18 19 20 21 22 23 24
23 24 25 26 27 28 29  28 29 30 31           25 26 27 28 29 30
30

      July                  August                September
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
                   1           1  2  3  4  5                1  2
 2  3  4  5  6  7  8   6  7  8  9 10 11 12   3  4  5  6  7  8  9
 9 10 11 12 13 14 15  13 14 15 16 17 18 19  10 11 12 13 14 15 16
16 17 18 19 20 21 22  20 21 22 23 24 25 26  17 18 19 20 21 22 23
23 24 25 26 27 28 29  27 28 29 30 31        24 25 26 27 28 29 30
30 31

      October               November              December
Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su  Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7                1  2  3  4                1  2
 8  9 10 11 12 13 14   5  6  7  8  9 10 11   3  4  5  6  7  8  9
15 16 17 18 19 20 21  12 13 14 15 16 17 18  10 11 12 13 14 15 16
22 23 24 25 26 27 28  19 20 21 22 23 24 25  17 18 19 20 21 22 23
29 30 31              26 27 28 29 30        24 25 26 27 28 29 30
                                            31
```

Fundamental Python Programming Techniques

This section demonstrates numerous Python programming syntax structures.

Selection Statements

The `if` statement is used to execute a specific statement or set of statements when the given condition is true. There are various forms of `if` structures, as shown in Table 1-9.

Table 1-9. *if Statement Structure*

Form	if statement	if-else Statement	Nested if Statement

Structure	`if(condition):`
	`statements`
		`if(condition):`
		`statements`
		`else:`
		`statements`
			`if (condition):`
			`statements`
			`elif (condition):`
			`statements`
			`else:`
			`statements`

The `if` statement is used to make decisions based on specific conditions occurring during the execution of the program. An action or set of actions is executed if the outcome is true or false otherwise. Figure 1-6 shows the general form of a typical decision-making structure found in most programming languages including Python. Any nonzero and non-null values are considered true in Python, while either zero or null values are considered false.

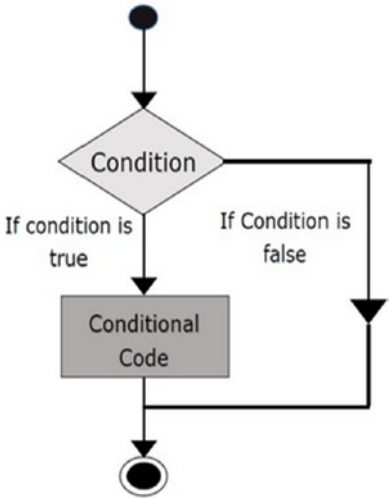


Figure 1-6. *Selection statement structure*

Listing 1-13 demonstrates two examples of a selection statement, remember the indentation is important in the Python structure. The first block shows that the value of x is equal to 5; hence, the condition is testing whether x equals 5 or not. Therefore, the output implements the statement when the condition is true.

Listing 1-13. The if-else Statement Structure

```
In [13]:#Comparison operators
      x=5
      if x==5:
             print ('Equal 5')
elif x>5:
      print ('Greater than 5')
elif x<5:
      print ('Less than 5')

Equal 5
```

```
In [14]:year=2000
      if year%4==0:
            print("Year(", year ,")is Leap")
else:
            print (year , "Year is not Leap" )

Year( 2000 )is Leap
```

Indentation determines which statement should be executed. In Listing 1-14, the `if` statement condition is false, and hence the outer `print` statement is the only executed statement.

Listing 1-14. Indentation of Execution

```
In [12]:#Indentation
      x=2
      if x>2:
             print ("Bigger than 2")
             print (" X Value bigger than 2")
      print ("Now we are out of if block\n")

Now we are out of if block
```

The nested `if` statement is an `if` statement that is the target of another `if` statement. In other words, a nested `if` statement is an `if` statement inside another `if` statement, as shown in Listing 1-15.

Listing 1-15. Nested Selection Statements

```
In [2]:a=10
      if a>=20:
             print ("Condition is True" )
else:
             if a>=15:
                   print ("Checking second value" )
```

```
else:
                    print ("All Conditions are false" )
All Conditions are false
```

Iteration Statements

There are various iteration statement structures in Python. The `for` loop is one of these structures; it is used to iterate the elements of a collection in the order that they appear. In general, statements are executed sequentially, where 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 numbers of times.

Control structures allow you to execute a statement or group of statements multiple times, as shown by Figure 1-7.

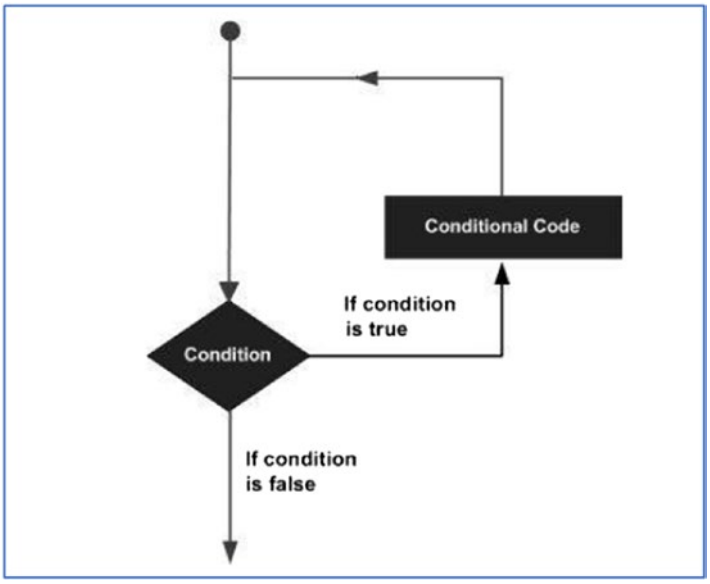


Figure 1-7. *A loop statement*

Table 1-10 demonstrates different forms of iteration statements. The Python programming language provides different types of loop statements to handle iteration requirements.

Table 1-10. *Iteration Statement Structure*

1	`for` **loop** Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
2	**Nested loops** You can use one or more loop inside any another `while`, `for`, or `do.. while` loop.
3	`while` **loop** Repeats a statement or group of statements while a given condition is true. It tests the condition *before* executing the loop body.
4	`do {....} while ()` Repeats a statement or group of statements while a given condition is true. It tests the condition *after* executing the loop body.

Python provides various support methods for iteration statements where it allows you to terminate the iteration, skip a specific iteration, or pass if you do not want any command or code to execute. Table 1-11 summarizes control statements within the iteration execution.

Table 1-11. *Loop Control Statements*

1	**Break statement** Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	**Continue statement** Causes the loop to skip the remainder of its body and immediately retests its condition prior to reiterating.
3	**Pass statement** The pass statement is used when a statement is required syntactically but you do not want any command or code to execute.

The `range()` statement is used with `for` loop statements where you can specify one value. For example, if you specify 4, the loop statement starts from 1 and ends with 3, which is n-1. Also, you can specify the start and end values. The following examples demonstrate loop statements.

Listing 1-16 displays all numerical values starting from 1 up to n-1, where n=4.

Listing 1-16. for Loop Statement

```
In [23]:# use the range statement
       for a in range (1,4):
       print ( a )

1
2
3
```

Listing 1-17 displays all numerical values starting from 0 up to n-1, where n=4.

Listing 1-17. Using the range() Method

```
In [24]:# use the range statement
      for a in range (4):
            print ( a )
```

```
0
1
2
3
```

Listing 1-18 displays the `while` iteration statement.

Listing 1-18. while Iteration Statement

```
In [32]:ticket=4
      while ticket>0:
            print ("Your ticket number is ", ticket)
            ticket -=1
```

```
Your ticket number is 4
Your ticket number is 3
Your ticket number is 2
Your ticket number is 1
```

Listing 1-19 iterates all numerical values in a list to find the maximum value.

Listing 1-19. Using a Selection Statement Inside a Loop Statement

```
In [2]:largest = None
      print ('Before:', largest)
      for val in [30, 45, 12, 90, 74, 15]:
if largest is None or val>largest:
      largest = val
      print ("Loop", val, largest)
print ("Largest", largest)
```

```
Before: None
Loop 30 30
Loop 45 45
Loop 90 90
Largest 90
```

In the previous examples, the first and second iterations used the `for` loop with a range statement. In the last example, iteration goes through a list of elements and stops once it reaches the last element of the iterated list.

A *break* statement is used to jump statements and transfer the execution control. It breaks the current execution, and in the case of an inner loop, the inner loop terminates immediately. However, a *continue* statement is a jump statement that skips execution of current iteration. After skipping, the loop continues with the next iteration. The `pass` keyword is used to execute nothing. The following examples demonstrate how and when to employ each statement.

The Use of Break, Continues, and Pass Statements

Listing 1-20 shows the break, continue, and pass statements.

Listing 1-20. Break, Continue, and Pass Statements

```
In [44]:for letter in 'Python3':
       if letter == 'o':
             break
        print (letter)
```

```
P
y
t
h
```

```
In [45]: a=0
         while a<=5:
              a=a+1
              if a%2==0:
               continue
              print (a)
 print ("End of Loop" )
```

```
1
3
5
End of Loop
```

```
In [46]: for i in [1,2,3,4,5]:
              if i==3:
                   pass
              print ("Pass when value is", i )
            print (i)
```

```
1
2
Pass when value is 3
3
4
5
```

As shown, you can iterate over a list of letters, as shown in Listing 1-20, and you can iterate over the word *Python3* and display all the letters. You stop iteration once you find the condition, which is the letter *o*. In addition,

you can use the pass statement 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.

try and except

try and except are used to handle unexpected values where you would like to validate entered values to avoid error occurrence. In the first example of Listing 1-21, you use try and except to handle the string "Al Fayoum," which is not convertible into an integer, while in the second example, you use try and except to handle the string 12, which is convertible to an integer value.

Listing 1-21. try and except Statements

```
In [14]: # Try and Except
astr='Al Fayoum'
      errosms=''
try:
             istr=int(astr) # error
except:
             istr=-1
             errosms="\nIncorrect entry"
print ("First Try:", istr , errosms)
```

```
First Try: -1
Incorrect entry
```

```
In [15]:# Try and Except
                   astr='12'
                   errosms=' '
                   try:
                    istr=int(astr) # error
                   except:
```

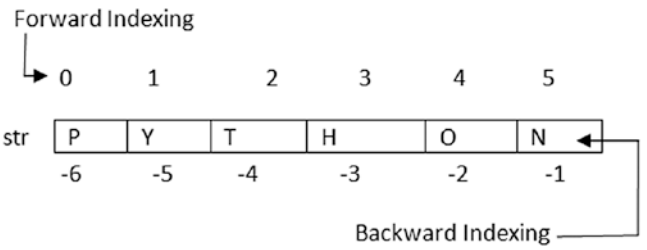
```
 istr=-1
 errosms="\nIncorrect entry"
                  print ("First Try:", istr , errosms)
```

```
First Try: 12
```

String Processing

A *string* is a sequence of characters that can be accessed by an expression in brackets called an *index*. For instance, if you have a string variable named `var1`, which maintains the word `PYTHON`, then `var1[1]` will return the character *Y*, while `var1[-2]` will return the character *O*. Python considers strings by enclosing text in single as well as double quotes. Strings are stored in a contiguous memory location that can be accessed from both directions (forward and backward), as shown in the following example, where

- Forward indexing starts with 0, 1, 2, 3, and so on.
- Backward indexing starts with -1, -2, -3, -4, and so on.



String Special Operators

Table 1-12 lists the operators used in string processing. Say you have the two variables `a= 'Hello'` and `b = 'Python'`. Then you can implement the operations shown in Table 1-12.

Table 1-12. *String Operators*

Operator	Description	Outputs
+	Concatenation: adds values on either side of the operator	`a + b` will give `HelloPython`.
*	Repetition: creates new strings, concatenating multiple copies of the same string	`a*2` will give `-HelloHello`.
[]	Slice: gives the character from the given index	`a[1]` will give `e`.
[:]	Range slice: gives the characters from the given range	`a[1:4]` will give `ell`.
in	Membership: returns true if a character exists in the given string	`H in a` will give true.
not in	Membership: returns true if a character does not exist in the given string	`M not in a` will give true.

Various symbols are used for string formatting using the operator `%`. Table 1-13 gives some simple examples.

Table 1-13. *String Format Symbols*

Format Symbol	Conversion
`%c`	Character
`%s`	String conversion via `str()` prior to formatting
`%i`	Signed decimal integer
`%d`	Signed decimal integer
`%u`	Unsigned decimal integer

(*continued*)

Table 1-13. (*continued*)

Format Symbol	Conversion
%o	Octal integer
%x	Hexadecimal integer (lowercase letters)
%X	Hexadecimal integer (uppercase letters)
%e	Exponential notation (with lowercase e)
%E	Exponential notation (with uppercase E)
%f	Floating-point real number
%g	The shorter of %f and %e
%G	The shorter of %f and %E

String Slicing and Concatenation

String slicing refers to a segment of a string that is extracted using an index or using search methods. In addition, the `len()` method is a built-in function that returns the number of characters in a string. Concatenation enables you to join more than one string together to form another string.

The operator `[n:m]` returns the part of the string from the *n*th character to the *m*th character, including the first but excluding the last. If you omit the first index (before the colon), the slice starts at the beginning of the string. In addition, if you omit the second index, the slice goes to the end of the string. The examples in Listing 1-22 show string slicing and concatenation using the + operator.

Listing 1-22. String Slicing and Concatenation

```
In [3]:var1 = 'Welcome to Dubai'
       var2 = "Python Programming"
       print ("var1[0]:", var1[0])
       print ("var2[1:5]:", var2[1:5])

       var1[0]: W
       var2[1:5]: ytho
```

```
In [5]:st1="Hello"
       st2=' World'
       fullst=st1 + st2
       print (fullst)
```

```
Hello World
```

```
In [11]:# looking inside strings
       fruit = 'banana'
       letter= fruit[1]
       print (letter)
       index=3
       w = fruit[index-1]
       print (w)
       print (len(fruit))
```

```
a
n
6
```

String Conversions and Formatting Symbols

It is possible to convert a string value into a float, double, or integer if the string value is applicable for conversion, as shown in Listing 1-23.

Listing 1-23. String Conversion and Format Symbols

```
In [14]:#Convert string to int
      str3 = '123'
      str3= int (str3)+1
      print (str3)
```

```
124
```

```
In [15]:#Read and convert data
      name=input('Enter your name: ')
      age=input('Enter your age: ')
      age= int(age) + 1
      print ("Name: %s"% name ,"\t Age:%d"% age)
```

```
Enter your name: Omar
Enter your age: 41
```

```
Name: Omar    Age:42
```

Loop Through String

You can use iteration statements to go through a string forward or backward. A lot of computations involve processing a string one character at a time. String processing can start at the beginning, select each character in turn, do something to it, and continue until the end. This pattern of processing is called a *traversal*. One way to write a traversal is with a `while` loop, as shown in Listing 1-24.

Listing 1-24. Iterations Through Strings

```
In [30]:# Looking through string
        fruit ='banana'
        index=0
        while index< len(fruit):
               letter = fruit [index]
```

```
            print (index, letter)
            index=index+1
0 b
1 a
2 n
3 a
4 n
5 a

In [31]:print ("\n Implementing iteration with continue")
      while True:
            line = input('Enter your data>')
            if line[0]=='#':
                  continue
            if line =='done':
                  break
            print (line )
      print ('End!')

Implementing iteration with continue

Enter your data>Higher Colleges of Technology
Higher Colleges of Technology

Enter your data>#

Enter your data>done
End!

In [32]:print ("\nPrinting in reverse order")
      index=len(fruit)-1
      while index>=0 :
            letter = fruit [index]
            print (index, letter )
            index=index-1
```

```
Printing in reverse order
5 a
4 n
3 a
2 n
1 a
0 b

Letterwise iteration
In [33]:Country='Egypt'
      for letter in Country:
            print (letter)

E
g
y
p
t
```

You can use iterations as well to count letters in a word or to count words in lines, as shown in Listing 1-25.

Listing 1-25. Iterating and Slicing a String

```
In [2]:# Looking and counting
      word='banana'
      count=0
      for letter in word:
            if letter =='a':
                  count +=1
      print ("Number of a in ", word, "is :", count )

Number of a in banana is : 3
```

```
In [3]:# String Slicing
       s="Welcome to Higher Colleges of Technology"
       print (s[0:4])
       print (s[6:7])
       print (s[6:20])
       print (s[:12])
       print (s[2:])
       print (s [:])
       print (s)
```

```
Welc
e
e to Higher Co Welcome to H
lcome to Higher Colleges of Technology Welcome to Higher
Colleges of Technology
Welcome to Higher Colleges of Technology
```

Python String Functions and Methods

Numerous built-in methods and functions can be used for string processing; Table 1-14 lists these methods.

Table 1-14. *Built-in String Methods*

Method/Function	Description
`capitalize()`	Capitalizes the first character of the string.
`count(string, begin,end)`	Counts a number of times a substring occurs in a string between the beginning and end indices.
`endswith(suffix, begin=0,end=n)`	Returns a Boolean value if the string terminates with a given suffix between the beginning and end.

(*continued*)

Table 1-14. (*continued*)

Method/Function	Description
find(substring, beginIndex, endIndex)	Returns the index value of the string where the substring is found between the begin index and the end index.
index(subsring, beginIndex, endIndex)	Throws an exception if the string is not found and works same as the find() method.
isalnum()	Returns true if the characters in the string are alphanumeric (i.e., letters or numbers) and there is at least one character. Otherwise, returns false.
isalpha()	Returns true when all the characters are letters and there is at least one character; otherwise, false.
isdigit()	Returns true if all the characters are digits and there is at least one character; otherwise, false.
islower()	Returns true if the characters of a string are in lowercase; otherwise, false.
isupper()	Returns false if the characters of a string are in uppercase; otherwise, false.
isspace()	Returns true if the characters of a string are white space; otherwise, false.
len(string)	Returns the length of a string.
lower()	Converts all the characters of a string to lowercase.
upper()	Converts all the characters of a string to uppercase.
startswith(str, begin=0,end=n)	Returns a Boolean value if the string starts with the given str between the beginning and end.

(*continued*)

Table 1-14. (*continued*)

Method/Function	Description
`swapcase()`	Inverts the case of all characters in a string.
`lstrip()`	Removes all leading white space of a string and can also be used to remove a particular character from leading white spaces.
`rstrip()`	Removes all trailing white space of a string and can also be used to remove a particular character from trailing white spaces.

Listing 1-26 shows how to use built-in methods to remove white space from a string, count specific letters within a string, check whether the string contains another string, and so on.

Listing 1-26. Implementing String Methods

```
In [29]:var1 =' Higher Colleges of Technology '
      var2='College'
      var3='g'
      print (var1.upper())
      print (var1.lower())
      print ('WELCOME TO'.lower())
      print (len(var1))
      print (var1.count(var3, 2, 29) ) # find how many g
      letters in var1
      print ( var2.count(var3) )
```

```
HIGHER COLLEGES OF TECHNOLOGY
higher colleges of technology
welcome to
```

```
31
3
1

In [33]:print (var1.endswith('r'))
      print (var1.startswith('O'))
      print (var1.find('h', 0, 29))
      print (var1.lstrip()) # It removes all leading whitespace
      of a string in var1
      print (var1.rstrip()) # It removes all trailing
      whitespace of a string in var1
      print (var1.strip()) # It removes all leading and
      trailing whitespace
      print ('\n')
      print (var1.replace('Colleges', 'University'))

False
False
4
Higher Colleges of Technology
 Higher Colleges of Technology
Higher Colleges of Technology

Higher University of Technology
```

The in Operator

The word in is a Boolean operator that takes two strings and returns true if the first appears as a substring in the second, as shown in Listing 1-27.

Listing 1-27. The in Method in String Processing

```
In [43]:var1 =' Higher Colleges of Technology '
      var2='College'
      var3='g'
```

```
      print ( var2 in var1)
      print ( var2 not in var1)
```

```
True
False
```

Parsing and Extracting Strings

The `find` operator returns the index of the first occurrence of a substring in another string, as shown in Listing 1-28. The `atpost` variable is used to maintain a returned index of the substring `@` as it appears in the `Maindata` string variable.

Listing 1-28. Parsing and Extracting Strings

```
In [39]:# Parsing and Extracting strings
            Maindata = 'From ossama.embarak@hct.ac.ae Sunday
            Jan 4 09:30:50 2017' atpost = Maindata.find('@')
            print ("\n<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>")
            print (atpost)
            print (Maindata[ :atpost])
            data = Maindata[ :atpost]
            name=data.split(' ')
            print (name)
            print (name[1].replace('.', ' ').upper())
            print ("\n<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>")
```

```
<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>
19
From ossama.embarak
['From', 'ossama.embarak']
OSSAMA EMBARAK
<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>
```

```
In [41]:# Another way to split strings
             Maindata = 'From ossama.embarak@hct.ac.ae Sunday
             Jan 4 09:30:50 2017'
             name= Maindata[ :atpost].replace('From','').upper()
             print (name.replace('.',' ').upper().lstrip())
             print ("\n<<<<<<<<<<<<<<>>>>>>>>>>>>>>")
             sppos=Maindata.find(' ', atpost)
             print (sppos)
             print (Maindata[ :sppos])
             host = Maindata [atpost + 1 : sppos ]
             print (host)
             print ("\n<<<<<<<<<<<<<<>>>>>>>>>>>>>>")

OSSAMA EMBARAK
<<<<<<<<<<<<<<>>>>>>>>>>>>>>
29
From ossama.embarak@hct.ac.ae
hct.ac.ae
<<<<<<<<<<<<<<>>>>>>>>>>>>>>
```

Tabular Data and Data Formats

Data is available in different forms. It can be unstructured data, semistructured data, or structured data. Python provides different structures to maintain data and to manipulate it such as variables, lists, dictionaries, tuples, series, panels, and data frames. Tabular data can be easily represented in Python using lists of tuples representing the records of the data set in a data frame structure. Though easy to create, these kinds of representations typically do not enable important tabular data manipulations, such as efficient column selection, matrix mathematics, or spreadsheet-style operations. Tabular is a package of Python modules for working with tabular data. Its main object is the `tabarray` class, which is a

data structure for holding and manipulating tabular data. You can put data into a `tabarray` object for more flexible and powerful data processing. The Pandas library also provides rich data structures and functions designed to make working with structured data fast, easy, and expressive. In addition, it provides a powerful and productive data analysis environment.

A Pandas data frame can be created using the following constructor:

```
pandas.DataFrame( data, index, columns, dtype, copy)
```

A Pandas data frame can be created using various input forms such as the following:

- List
- Dictionary
- Series
- Numpy ndarrays
- Another data frame

Chapter 3 will demonstrate the creation and manipulation of the data frame structure in detail.

Python Pandas Data Science Library

Pandas is an open source Python library providing high-performance data manipulation and analysis tools via its powerful data structures. The name Pandas is derived from "panel data," an econometrics term from multidimensional data. The following are the key features of the Pandas library:

- Provides a mechanism to load data objects from different formats
- Creates efficient data frame objects with default and customized indexing
- Reshapes and pivots date sets

- Provides efficient mechanisms to handle missing data
- Merges, groups by, aggregates, and transforms data
- Manipulates large data sets by implementing various functionalities such as slicing, indexing, subsetting, deletion, and insertion
- Provides efficient time series functionality

Sometimes you have to import the Pandas package since the standard Python distribution doesn't come bundled with the Pandas module. A lightweight alternative is to install Numpy using popular the Python package installer `pip`. The Pandas library is used to create and process series, data frames, and panels.

A Pandas Series

A *series* is a one-dimensional labeled array capable of holding data of any type (integer, string, float, Python objects, etc.). Listing 1-29 shows how to create a series using the Pandas library.

Listing 1-29. Creating a Series Using the Pandas Library

```
In [34]:#Create series from array using pandas and numpy
       import pandas as pd
       import numpy as np
       data = np.array([90,75,50,66])
       s = pd.Series(data,index=['A','B','C','D'])
       print (s)
A 90
B 75
C 50
D 66
dtype: int64
```

```
In [36]:print (s[1])

75

In [37]:#Create series from dictionary using pandas
      import pandas as pd
      import numpy as np
      data = {'Ahmed' : 92, 'Ali' : 55, 'Omar' : 83}
      s = pd.Series(data,index=['Ali','Ahmed','Omar'])
      print (s)

Ali 55
Ahmed 92
Omar 83
dtype: int64

In [38]:print (s[1:])


Ahmed 92
Omar 83
dtype: int64
```

A Pandas Data Frame

A *data frame* is a two-dimensional data structure. In other words, data is aligned in a tabular fashion in rows and columns. In the following table, you have two columns and three rows of data. Listing 1-30 shows how to create a data frame using the Pandas library.

Name	Age
Ahmed	35
Ali	17
Omar	25

Listing 1-30. Creating a Data Frame Using the Pandas Library

```
In [39]:import pandas as pd
        data = [['Ahmed',35],['Ali',17],['Omar',25]]
        DataFrame1 = pd.DataFrame(data,columns=['Name','Age'])
        print (DataFrame1)

    Name    Age
0   Ahmed   35
1   Ali     17
2   Omar    25
```

You can retrieve data from a data frame starting from index 1 up to the end of rows.

```
In [40]: DataFrame1[1:]


Out[40]:      Name    Age
          1   Ali     17
          2   Omar    25
```

You can create a data frame using a dictionary.

```
In [41]:import pandas as pd
        data = {'Name':['Ahmed', 'Ali', 'Omar',
        'Salwa'],'Age':[35,17,25,30]}
        dataframe2 = pd.DataFrame(data, index=[100, 101, 102, 103])
        print (dataframe2)

        Age     Name
100     35      Ahmed
101     17      Ali
102     25      Omar
103     30      Salwa
```

You can select only the first two rows in a data frame.

```
In [42]: dataframe2[:2]

Out[42]:     Age   Name
       100   35    Ahmed
       101   17    Ali
```

You can select only the name column in a data frame.

```
In [43]: dataframe2['Name']

Out[43]:100         Ahmed
101    Ali
102    Omar
103    Salwa
Name: Name, dtype: object
```

A Pandas Panels

A *panel* is a 3D container of data that can be created from different data structures such as from a dictionary of data frames, as shown in Listing 1-31.

Listing 1-31. Creating a Panel Using the Pandas Library

```
In [44]:# Creating a panel
      import pandas as pd
      import numpy as np
      data = {'Temperature Day1' : pd.DataFrame(np.random.
      randn(4, 3)),'Temperature Day2' : pd.DataFrame
      (np.random.randn(4, 2))}
      p = pd.Panel(data)
      print (p['Temperature Day1'])

        0         1          2
0   1.152400    -1.298529   1.440522
```

```
1   -1.404988   -0.105308   -0.192273
2   -0.575023   -0.424549    0.146086
3   -1.347784    1.153291   -0.131740
```

Python Lambdas and the Numpy Library

The lambda operator is a way to create small anonymous functions, in other words, functions without names. These functions are throwaway functions; they are just needed where they have been created. The lambda feature is useful mainly for Lisp programmers. Lambda functions are used in combination with the functions `filter()`, `map()`, and `reduce()`.

Anonymous functions refer to functions that aren't named and are created by using the keyword `lambda`. A lambda is created without using the `def` keyword; it takes any number of arguments and returns an evaluated expression, as shown in Listing 1-32.

Listing 1-32. Anonymous Function

```
In [34]:# Anonymous Function Definition
             summation=lambda val1, val2: val1 + val2#Call
             summation as a function
          print ("The summation of 7 + 10 = ", summation(7,10) )

The summation of 7 + 10 = 17

In [46]:result = lambda x, y : x * y
        result(2,5)

Out[46]: 10

In [47]:result(4,10)

Out[47]: 40
```

The map() Function

The `map()` function is used to apply a specific function on a sequence of data. The `map()` function has two arguments.

`r = map(func, seq)`

Here, `func` is the name of a function to apply, and `seq` is the sequence (e.g., a list) that applies the function `func` to all the elements of the sequence `seq`. It returns a new list with the elements changed by `func`, as shown in Listing 1-33.

Listing 1-33. Using the map() Function

```
In [65]:def fahrenheit(T):
             return ((float(9)/5)*T + 32)
          def celsius(T):
             return (float(5)/9)*(T-32)
        Temp = (15.8, 25, 30.5,25)
        F = list ( map(fahrenheit, Temp))
        C = list ( map(celsius, F))
        print (F)
        print (C)
```

```
[60.44, 77.0, 86.9, 77.0]
[15.799999999999999, 25.0, 30.500000000000004, 25.0]
```

```
In [72]:Celsius = [39.2, 36.5, 37.3, 37.8]
Fahrenheit = map(lambda x: (float(9)/5)*x + 32, Celsius)
for x in Fahrenheit:
      print(x)
```

```
102.56
97.7
99.14
100.03999999999999
```

The filter() Function

The `filter()` function is an elegant way to filter out all elements of a list for which the applied function returns true.

For instance, the function `filter(func, list1)` needs a function called `func` as its first argument. `func` returns a Boolean value, in other words, either true or false. This function will be applied to every element of the list `list1`. Only if `func` returns true will the element of the list be included in the result list.

The `filter()` function in Listing 1-34 is used to return only even values.

Listing 1-34. Using the filter() Function

```
In [79]:fib = [0,1,1,2,3,5,8,13,21,34,55]
        result = filter(lambda x: x % 2==0, fib)
        for x in result:
             print(x)
```

```
0
2
8
34
```

The reduce () Function

The `reduce()` function continually applies the function `func` to a sequence `seq` and returns a single value.

The `reduce()` function is used to find the max value in a sequence of integers, as shown in Listing 1-35.

Listing 1-35. Using the reduce() Function

```
In [81]: f = lambda a,b: a if (a > b) else b
reduce(f, [47,11,42,102,13])
```

```
102
```

```
In [82]: reduce(lambda x,y: x+y, [47,11,42,13])
```

```
113
```

Python Numpy Package

Numpy is a Python package that stands for "numerical Python." It is a library consisting of multidimensional array objects and a collection of routines for processing arrays.

The Numpy library is used to apply the following operations:

- Operations related to linear algebra and random number generation
- Mathematical and logical operations on arrays
- Fourier transforms and routines for shape manipulation

For instance, you can create arrays and perform various operations such as adding or subtracting arrays, as shown in Listing 1-36.

Listing 1-36. Example of the Numpy Function

```
In [83]:a=np.array([[1,2,3],[4,5,6]])
        b=np.array([[7,8,9],[10,11,12]])
        np.add(a,b)
```

```
Out[83]: array([[ 8, 10, 12], [14, 16, 18]])
```

```
In [84]:np.subtract(a,b) #Same as a-b
```

```
Out[84]: array([[-6, -6, -6], [-6, -6, -6]])
```

Data Cleaning and Manipulation Techniques

Keeping accurate data is highly important for any data scientist. Developing an accurate model and getting accurate predictions from the applied model depend on the missing values treatment. Therefore, handling missing data is important to make models more accurate and valid.

Numerous techniques and approaches are used to handle missing data such as the following:

- Fill NA forward
- Fill NA backward
- Drop missing values
- Replace missing (or) generic values
- Replace NaN with a scalar value

The following examples are used to handle the missing values in a tabular data set:

```
In [31]: dataset.fillna(0) # Fill missing values with zero value
In [35]: dataset.fillna(method='pad') # Fill methods Forward
In [35]: dataset.fillna(method=' bfill') # Fill methods Backward
In [37]: dataset.dropna() # remove all missing data
```

Chapter 5 covers different gathering and cleaning techniques.

Abstraction of the Series and Data Frame

A *series* is one of the main data structures in Pandas. It differs from lists and dictionaries. An easy way to visualize this is as two columns of data. The first is the special index, a lot like the dictionary keys, while the second is your actual data. You can determine an index for a series, or

Python can automatically assign indices. Different attributes can be used to retrieve data from a series' `iloc()` and `loc()` attributes. Also, Python can automatically retrieve data based on the passed value. If you pass an object, then Python considers that you want to use the index label–based `loc()`. However, if you pass an index integer parameter, then Python considers the `iloc()` attribute, as indicated in Listing 1-37.

Listing 1-37. Series Structure and Query

```
In [6]: import pandas as pd
         animals = ["Lion", "Tiger", "Bear"]
         pd.Series(animals)

Out[6]: 0 Lion
 1 Tiger
 2 Bear
dtype: object
```

You can create a series of numerical values.

```
In [5]: marks = [95, 84, 55, 75]
        pd.Series(marks)

Out[5]: 0   95
        1   84
        2   55
        3   75
        dtype: int64
```

You can create a series from a dictionary where indices are the dictionary keys.

```
In [11]: quiz1 = {"Ahmed":75, "Omar": 84, "Salwa": 70}
      q = pd.Series(quiz1)
      q
```

```
Out[11]: Ahmed   75
         Omar    84
         Salwa   70
         dtype: int64
```

The following examples demonstrate how to query a series.

You can query a series using a series label or the `lock()` attribute.

```
In [13]: q.loc['Ahmed']

Out[13]: 75

In [20]: q['Ahmed']


Out[20]: 75
```

You can query a series using a series index or the `ilock()` attribute.

```
In [19]: q.iloc[2]

Out[19]: 70

In [21]: q[2]

Out[21]: 70
```

You can implement a Numpy operation on a series.

```
In [25]:s = pd.Series([70,90,65,25, 99])
         s
Out[25]:0   70
         1   90
         2   65
         3   25
         4   99
         dtype: int64
```

```
In [27]:total =0
      for val in s:
             total += val
      print (total)
```

```
349
```

You can get faster results by using Numpy functions on a series.

```
In [28]: import numpy as np
          total = np.sum(s)
          print (total)
```

```
349
```

It is possible to alter a series to add new values; it is automatically detected by Python that the entered values are not in the series, and hence it adds it to the altered series.

```
In [29]:s = pd.Series ([99,55,66,88])
          s.loc['Ahmed'] = 85
          s
```

```
Out[29]: 0   99
         1   55
         2   66
         3   88
     Ahmed   85
     dtype: int64
```

You can append two or more series to generate a larger one, as shown here:

```
In [32]: test = [95, 84, 55, 75]
      marks = pd.Series(test)
      s = pd.Series ([99,55,66,88])
      s.loc['Ahmed'] = 85
```

```
NewSeries = s.append(marks)
NewSeries
```

```
Out[32]: 0   99
         1   55
         2   66
         3   88
     Ahmed   85
         0   95
         1   84
         2   55
         3   75
     dtype: int64
```

The *data frame* data structure is the main structure for data collection and processing in Python. A data frame is a two-dimensional series object, as shown in Figure 1-8, where there's an index and multiple columns of content each having a label.

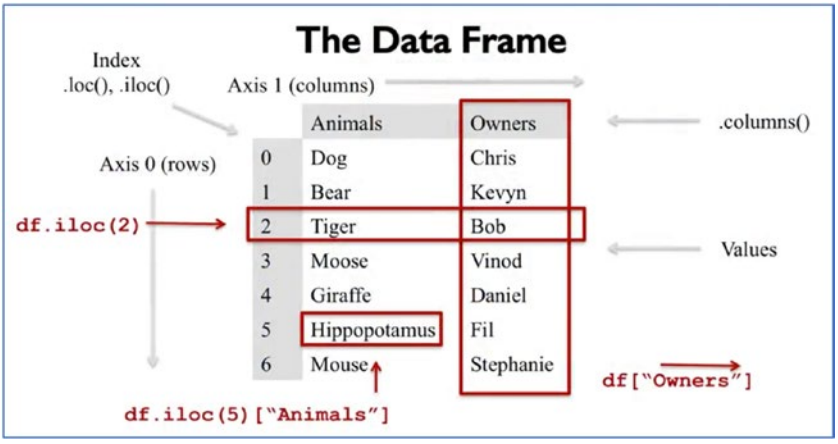


Figure 1-8. *Data frame virtual structure*

Data frame creation and queries were discussed earlier in this chapter and will be discussed again in the context of data collection structures in Chapter 3.

Running Basic Inferential Analyses

Python provides numerous libraries for inference and statistical analysis such as Pandas, SciPy, and Numpy. Python is an efficient tool for implementing numerous statistical data analysis operations such as the following:

- Linear regression
- Finding correlation
- Measuring central tendency
- Measuring variance
- Normal distribution
- Binomial distribution
- Poisson distribution
- Bernoulli distribution
- Calculating p-value
- Implementing a Chi-square test

Linear regression between two variables represents a straight line when plotted as a graph, where the exponent (power) of both of the variables is 1. A nonlinear relationship where the exponent of any variable is not equal to 1 creates a curve shape.

Let's use the built-in Tips data set available in the Seaborn Python library to find linear regression between a restaurant customer's total bill value and each bill's tip value, as shown in Figure 1-9. The function in Seaborn to find the linear regression relationship is `regplot`.

```
In [40]:import seaborn as sb
        from matplotlib import pyplot as plt
        df = sb.load_dataset('tips')
        sb.regplot(x = "total_bill", y = "tip", data = df)
        plt.xlabel('Total Bill')
        plt.ylabel('Bill Tips')
        plt.show()
```

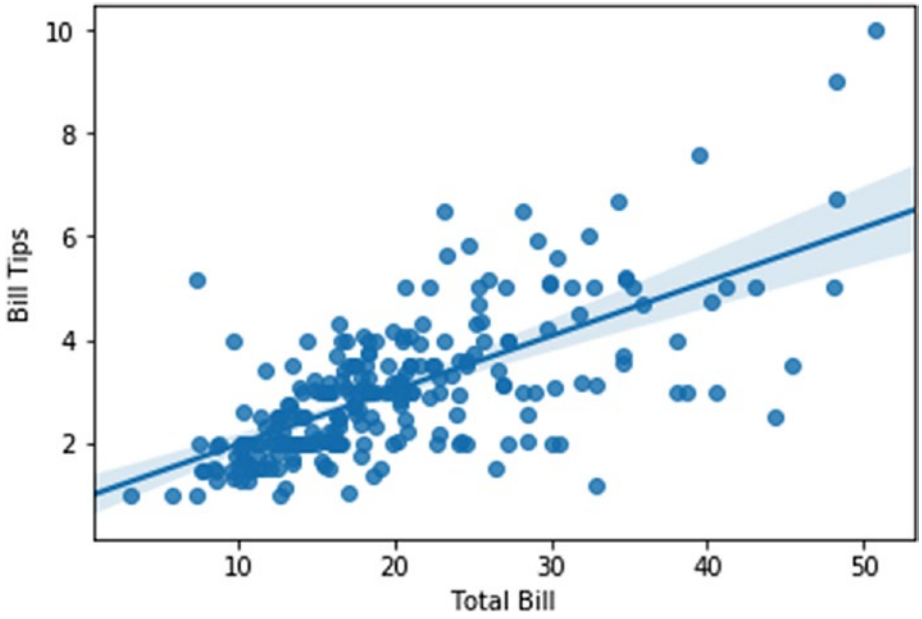


Figure 1-9. *Regression analysis*

Correlation refers to some statistical relationship involving dependence between two data sets, such as the correlation between the price of a product and its sales volume.

Let's use the built-in Iris data set available in the Seaborn Python library and try to measure the correlation between the length and the width of the sepals and petals of three species of iris, as shown in Figure 1-10.

```
In [42]: import matplotlib.pyplot as plt
         import seaborn as sns
         df = sns.load_dataset('iris')
         sns.pairplot(df, kind="scatter")
         plt.show()
```

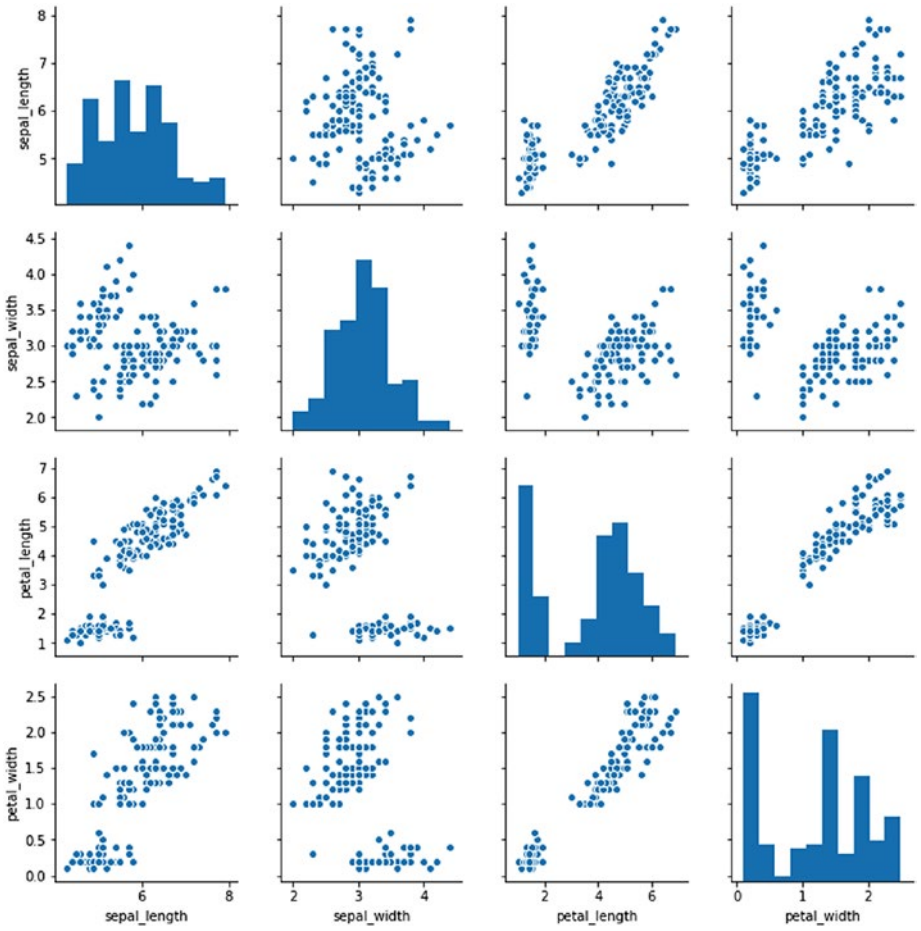


Figure 1-10. *Correlation analysis*

In statistics, *variance* is a measure of how dispersed the values are from the mean value. Standard deviation is the square root of variance. In other words, it is the average of the squared difference of values in a data set from the mean value. In Python, you can calculate this value by using the function `std()` from the Pandas library.

```
In [58]: import pandas as pd
d = {
'Name': pd.Series(['Ahmed','Omar','Ali','Salwa','Majid',
 'Othman','Gameel','Ziad','Ahlam','Zahrah',
 'Ayman','Alaa']),
'Age': pd.Series([34,26,25,27,30,54,23,43,40,30,28,46]),
'Height':pd.Series([114.23,173.24,153.98,172.0,153.20,164.6,
 183.8,163.78,172.0,164.8 ])}
df = pd.DataFrame(d) #Create a DataFrame

print (df.std())# Calculate and print the standard deviation

Age     9.740574
Height 18.552823
Out[46]: [Text(0,0.5,'Frequency'), Text(0.5,0,'Binomial')]
```

You can use the `describe()` method to find the full description of a data frame set, as shown here:

```
In [59]: print (df.describe())

       Age Height
count 12.000000 12.000000
mean 33.833333 164.448333
std 9.740574 18.552823
min 23.000000 114.230000
25% 26.750000 161.330000
```

```
50% 30.000000 168.400000
75% 40.750000 173.455000
max 54.000000 183.800000
```

Central tendency measures the distribution of the location of values of a data set. It gives you an idea of the average value of the data in the data set and an indication of how widely the values are spread in the data set.

The following example finds the mean, median, and mode values of the previously created data frame:

```
In [60]: print ("Mean Values in the Distribution")
         print (df.mean())
         print ("*******************************")
         print ("Median Values in the Distribution")
         print (df.median())
         print ("*******************************")
         print ("Mode Values in the Distribution")
         print (df['Height'].mode())
```

```
Mean Values in the Distribution
Age 33.833333
Height    164.448333
dtype: float64
*******************************
Median Values in the Distribution
Age 30.0
Height    168.4
dtype: float64
*******************************
Mode Values of height in the Distribution
0     172.0
dtype: float64
```

Summary

This chapter introduced the data science field and the use of Python programming for implementation. Let's recap what was covered in this chapter.

- The data science main concepts and life cycle
- The importance of Python programming and its main libraries used for data science processing
- Different Python data structure use in data science applications
- How to apply basic Python programming techniques
- Initial implementation of abstract series and data frames as the main Python data structure
- Data cleaning and its manipulation techniques
- Running basic inferential statistical analyses

The next chapter will cover the importance of data visualization in business intelligence and much more.

Exercises and Answers

1. Write a Python script to prompt users to enter two values; then perform the basic arithmetical operations of addition, subtraction, multiplication, and division on the values.

Answer:

```
In [2]: # Store input numbers:
num1 = input('Enter first number: ')
```

```
num2 = input('Enter second number: ')
sumval = float(num1) + float(num2)     # Add two numbers
minval = float(num1) - float(num2)     # Subtract two numbers
mulval = float(num1) * float(num2)     # Multiply two numbers
divval = float(num1) / float(num2)     #Divide two numbers

# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2,
sumval))

# Display the subtraction
print('The subtraction of {0} and {1} is {2}'.format(num1, num2,
minval))

# Display the multiplication
print('The multiplication of {0} and {1} is {2}'.format(num1,
num2, mulval))

# Display the division
print('The division of {0} and {1} is {2}'.format(num1, num2,
divval))

Enter first number: 10
Enter second number: 5
The sum of 10 and 5 is 15.0
The subtraction of 10 and 5 is 5.0
The multiplication of 10 and 5 is 50.0
The division of 10 and 5 is 2.0
```

2. Write a Python script to prompt users to enter the lengths of a triangle sides. Then calculate the semiperimeters. Calculate the triangle area and display the result to the user. The area of a triangle is (s*(s-a)*(s-b)*(s-c))-1/2.

Answer:

```
In [3]:a = float(input('Enter first side: '))
       b = float(input('Enter second side: '))
       c = float(input('Enter third side: '))
       s = (a + b + c) / 2 # calculate the semiperimeter
       area = (s*(s-a)*(s-b)*(s-c)) ** 0.5 # calculate the area
       print('The area of the triangle is %0.2f' %area)
```

```
Enter first side: 10
Enter second side: 9
Enter third side: 7
The area of the triangle is 30.59
```

3. Write a Python script to prompt users to enter the first and last values and generate some random values between the two entered values.

Answer:

```
In [7]:import random
a = int(input('Enter the starting value : '))
b = int(input('Enter the end value : '))
print(random.randint(a,b))
random.sample(range(a, b), 3)
```

```
Enter the starting value : 10
Enter the end value : 100
14
Out[7]: [64, 12, 41]
```

4. Write a Python program to prompt users to enter a distance in kilometers; then convert kilometers to miles, where 1 kilometer is equal to 0.62137 miles. Display the result.

Answer:

```
In [9]: # convert kilometers to miles
kilometers = float(input('Enter the distance in kilometers: '))
# conversion factor
Miles = kilometers * 0.62137
print('%0.2f kilometers is equal to %0.2f miles'
 %(kilometers, Miles))
```

```
Enter the distance in kilometers: 120
120.00 kilometers is equal to 74.56 miles
```

5. Write a Python program to prompt users to enter a Celsius value; then convert Celsius to Fahrenheit, where T(°F) = T(°C) x 1.8 + 32. Display the result.

Answer:

```
In [11]: # convert Celsius to Fahrenheit
       Celsius = float(input('Enter temperature in Celsius: '))
       # conversion factor
       Fahrenheit = (Celsius * 1.8) + 32
       print('%0.2f Celsius is equal to %0.2f Fahrenheit'
 %(Celsius, Fahrenheit))
```

```
Enter temperature in Celsius: 25
25.00 Celsius is equal to 77.00 Fahrenheit
```

6. Write a program to prompt users to enter their working hours and rate per hour to calculate gross pay. The program should give the employee 1.5 times the hours worked above 30 hours. If Enter Hours is 50 and Enter Rate is 10, then the calculated payment is Pay: 550.0.

Answer:

```
In [6]:Hflage=True
       Rflage=True
       while Hflage & Rflage :
              hours = input ('Enter Hours:')
              try:
                     hours = int(hours)
                     Hflage=False
              except:
                     print ("Incorrect hours number !!!!")

              try:
                     rate = input ('Enter Rate:')
                     rate=float(rate)
                     Rflage=False
              except:
                      print ("Incorrect rate !!")

       if hours>40:
              pay= 40 * rate + (rate*1.5) * (hours - 40)
       else:
               pay= hours * rate
       print ('Pay:',pay)

Enter Hours: 50
Enter Rate: 10

Pay: 550.0
```

7. Write a program to prompt users to enter a value; then check whether the entered value is positive or negative value and display a proper message.

Answer:

```
In [1]: Val = float(input("Enter a number: "))
      if Val > 0:
             print("{0} is a positive number".format(Val))
      elif Val == 0:
             print("{0} is zero".format(Val))
      else:
             print("{0} is negative number".format(Val))
```

```
Enter a number: -12
-12.0 is negative number
```

8. Write a program to prompt users to enter a value; then check whether the entered value is odd or even and display a proper message.

Answer:

```
In [4]:# Check if a Number is Odd or Even
      val = int(input("Enter a number: "))
      if (val % 2) == 0:
             print("{0} is an Even number".format(val))
      else:
             print("{0} is an Odd number".format(val))
```

```
Enter a number: 13
13 is an Odd number
```

9. Write a program to prompt users to enter an age; then check whether each person is a child, a teenager, an adult, or a senior. Display a proper message.

Age	Category
< 13	Child
13 to 17	Teenager
18 to 59	Adult
> 59	Senior

Answer:

```
In [6]:age = int(input("Enter age of a person : "))
      if(age < 13):
             print("This is a child")
      elif(age >= 13 and age <=17):
             print("This is a teenager")
      elif(age >= 18 and age <=59):
             print("This is an adult")
      else:
             print("This is a senior")
```

```
Enter age of a person : 40
```

```
This is an adult
```

10. Write a program to prompt users to enter a car's speed; then calculate fines according to the following categories, and display a proper message.

Speed Limit	Fine Value
< 80	0
81 to 99	200
100 to 109	350
> 109	500

Answer:

```
In [7]:Speed = int(input("Enter your car speed"))
         if(Speed < 80):
             print("No Fines")
         elif(Speed >= 81 and Speed <=99):
             print("200 AE Fine ")
         elif(Speed >= 100 and Speed <=109):
             print("350 AE Fine ")
         else:
             print("500 AE Fine ")

Enter your car speed120

500 AE Fine
```

11. Write a program to prompt users to enter a year; then find whether it's a leap year. A year is considered a leap year if it's divisible by 4 and 100 *and* 400. If it's divisible by 4 and 100 but not by 400, it's not a leap year. Display a proper message.

Answer:

```
In [11]:year = int(input("Enter a year: "))
      if (year % 4) == 0:
            if (year % 100) == 0:
                  if (year % 400) == 0:
                        print("{0} is a leap year".
                        format(year))
                  else:
                        print("{0} is not a leap year".
                        format(year))
```

```
            else:
                  print("{0} is a leap year".format(year))
      else:
         print("{0} is not a leap year".format(year))
```

```
Enter a year: 2000
```

```
2000 is a leap year
```

12. Write a program to prompt users to enter a Fibonacci sequence. The Fibonacci sequence is the series of numbers 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ... The next number is found by adding the two numbers before it. For example, the 2 is found by adding the two numbers before it (1+1). Display a proper message.

Answer:

```
In [14]:nterms = int(input("How many terms you want? "))
            # first two terms
            n1 = 0
            n2 = 1
            count = 2
            # check if the number of terms is valid
            if nterms <= 0:
                  print("Please enter a positive integer")
            elif nterms == 1:
                  print("Fibonacci sequence:")
                  print(n1)
```

```
        else:
              print("Fibonacci sequence:")
              print(n1,",",n2,end=', ') # end=', ' is used
              to continue printing in the same line
              while count < nterms:
                    nth = n1 + n2
                    print(nth,end=' , ')
                    # update values
                    n1 = n2
                    n2 = nth
                    count += 1

How many terms you want? 8

Fibonacci sequence:
0 , 1, 1 , 2 , 3 , 5 , 8 , 13 ,
```