MASTER OF COMPUTER APPLICATIONS

SEMESTER 1

PYTHON PROGRAMMING

Unit 12

Python for Data Science

Table of Contents

SL No	Topic	Fig No / Table / Graph	SAQ / Activity	Page No
1	Introduction	-	-	4 - 5
	1.1 Learning Objectives	-	-	
2	Why Python for Data Analysis?	-	-	6 - 7
	2.1 Python as Glue	-	-	
	2.2 Why Not Python?	-	-	
3	Essential Libraries for Data Analysis	-	-	8 - 10
	3.1 NumPy	-	-	
	3.2 pandas	-	-	
	3.3 matplotlib	-	-	
4	NumPy Basics: Arrays and Vectorized Computation	-	-	11 - 25
	4.1 The NumPy ndarray: A Multidimensional Array Object	-	-	
	4.2 Array-Oriented Programming with Arrays	-	-	
5	Pandas	-	-	26 - 37
	5.1 Introduction to pandas Data Structures	-	-	
	5.2 Essential Functionality	-	-	
6	Data Loading, Storage, and File Formats	-	-	38 - 42
	6.1 Reading and Writing Data in Text Format	-	-	
	6.2 Reading Large Files in Chunks	-	-	

1. INTRODUCTION

Welcome to Unit 12 of our series! By now, in Unit 11, you've gained a foundational grasp of Python for web development. You learned how to choose between popular frameworks like Flask and Django, both of which offer unique advantages depending on your project needs. You explored the essentials of building basic web applications, understanding MVC architecture, and grasping the basics of the chosen framework. We delved deep into routing, learning how to map URLs to functions efficiently, and you also tackled templates—creating dynamic content and mastering template inheritance to keep your codebase clean and maintainable.

Now, in Unit 12, we shift gears to a critical and exciting domain: Data Analysis with Python. Why learn data analysis? In the digital age, data is ubiquitous, and being able to manipulate and understand this data can provide significant advantages in any field. Python, with its simplicity and powerful libraries, is a favorite tool for data analysts around the world. This unit will cover why Python is an excellent choice for data analysis, explore its role as a glue language, and discuss situations where Python might not be the best fit. Expect to learn about essential libraries like NumPy, pandas, and matplotlib, which are pillars for any data analyst using Python.

To effectively learn and master the concepts in this unit, you will engage with a variety of resources and exercises. The journey will begin with understanding Python as a tool for data analysis—exploring its integrative capabilities and limitations. From there, you will dive into practical tutorials and examples using NumPy for numerical data manipulation, pandas for data structuring and manipulation, and matplotlib for data visualization. Each section is designed to build on the last, with practical, real-world applications that reinforce theoretical knowledge. Interactive exercises, coding challenges, and mini-projects will help cement your understanding and enable you to harness the power of Python to analyze and interpret complex data sets. Let's embark on this analytical journey, enhancing your skill set and preparing you to tackle real-world data challenges.

1.1 Learning Objectives

At the end of this topic, you will be able to:

- *Identify key Python libraries used for data analysis such as NumPy, pandas, and matplotlib.*
- *Explain the concept of data frames and series within pandas.*
- *Demonstrate the use of pandas to perform basic data manipulation tasks such as sorting, filtering, and grouping data.*
- *Compare and contrast different methods for handling missing data in a dataset using pandas.*
- *Critique various data visualization techniques using matplotlib to determine the most effective strategy for presenting data.*
- *Design and execute a comprehensive data analysis workflow that incorporates data cleaning, transformation, and visualization, culminating in actionable insights.*

2. Why Python for Data Analysis?

For many people, the Python programming language has strong appeal. Since its first appearance in 1991, Python has become one of the most popular interpreted programming languages, along with Perl, Ruby, and others. Python and Ruby have become especially popular since 2005 or so for building websites using their numerous web frameworks, like Rails (Ruby) and Django (Python). Such languages are often called scripting languages, as they can be used to quickly write small programs, or scripts to automate other tasks. I don't like the term "scripting language," as it carries a connotation that they cannot be used for building serious software. Among interpreted languages, for various historical and cultural reasons, Python has developed a large and active scientific computing and data analysis community. In the last 10 years, Python has gone from a bleeding-edge or "at your own risk" scientific computing language to one of the most important languages for data science, machine learning, and general software development in academia and industry.

For data analysis and interactive computing and data visualization, Python will inevitably draw comparisons with other open source and commercial programming languages and tools in wide use, such as R, MATLAB, SAS, Stata, and others. In recent years, Python's improved support for libraries (such as pandas and scikit-learn) has made it a popular choice for data analysis tasks. Combined with Python's overall strength for general-purpose software engineering, it is an excellent option as a primary language for building data applications.

2.1 Python as Glue

Part of Python's success in scientific computing is the ease of integrating C, C++, and FORTRAN code. Most modern computing environments share a similar set of legacy FORTRAN and C libraries for doing linear algebra, optimisation, integration, fast Fourier transforms, and other such algorithms. The same story has held true for many companies and national labs that have used Python to glue together decades' worth of legacy software.

Many programs consist of small portions of code where most of the time is spent, with large amounts of "glue code" that doesn't run often. In many cases, the execution time of the glue code is insignificant; effort is most fruitfully invested in optimising the computational bottlenecks, sometimes by moving the code to a lower-level language like C.

2.2 Why Not Python?

While Python is an excellent environment for building many kinds of analytical applications and general-purpose systems, there are a number of uses for which Python may be less suitable.

As Python is an interpreted programming language, in general most Python code will run substantially slower than code written in a compiled language like Java or C++. As programmer time is often more valuable than CPU time, many are happy to make this trade-off. However, in an application with very low latency or demanding resource utilization requirements (e.g., a high-frequency trading system), the time spent programming in a lower-level (but also lower-productivity) language like C++ to achieve the maximum possible performance might be time well spent.

Python can be a challenging language for building highly concurrent, multithreaded applications, particularly applications with many CPU-bound threads. The reason for this is that it has what is known as the global interpreter lock (GIL), a mechanism that prevents the interpreter from executing more than one Python instruction at a time. The technical reasons for why the GIL exists are beyond the scope of this book. While it is true that in many big data processing applications, a cluster of computers may be required to process a dataset in a reasonable amount of time, there are still situations where a single-process, multithreaded system is desirable.

This is not to say that Python cannot execute truly multithreaded, parallel code. Python C extensions that use native multithreading (in C or C++) can run code in parallel without being impacted by the GIL, so long as they do not need to regularly interact with Python objects.

3. ESSENTIAL LIBRARIES FOR DATA ANALYSIS

For those who are less familiar with the Python data ecosystem and the libraries used throughout the book, I will give a brief overview of some of them.

3.1 NumPy

NumPy, short for Numerical Python, has long been a cornerstone of numerical com- puting in Python. It provides the data structures, algorithms, and library glue needed for most scientific applications involving numerical data in Python. NumPy contains, among other things:

- A fast and efficient multidimensional array object ndarray
- Functions for performing element-wise computations with arrays or mathematical operations between arrays
- Tools for reading and writing array-based datasets to disk
- Linear algebra operations, Fourier transform, and random number generation
- A mature C API to enable Python extensions and native C or C++ code to access NumPy's data structures and computational facilities

Beyond the fast array-processing capabilities that NumPy adds to Python, one of its primary uses in data analysis is as a container for data to be passed between algorithms and libraries. For numerical data, NumPy arrays are more efficient for storing and manipulating data than the other built-in Python data structures. Also, libraries written in a lower-level language, such as C or Fortran, can operate on the data stored in a NumPy array without copying data into some other memory representation. Thus, many numerical computing tools for Python either assume NumPy arrays as a primary data structure or else target seamless interoperability with NumPy.

3.2 pandas

pandas provides high-level data structures and functions designed to make working with structured or tabular data fast, easy, and expressive. Since its emergence in 2010, it has helped enable Python to be a powerful and productive data analysis environment. The primary objects in pandas that will be used in this book are the DataFrame, a tabular, column-oriented data structure with both row and column labels, and the Series, a one-dimensional labelled array object.

pandas blends the high-performance, array-computing ideas of NumPy with the flexible data manipulation capabilities of spreadsheets and relational databases (such as SQL). It provides sophisticated indexing functionality to make it easy to reshape, slice and dice, perform aggregations, and select subsets of data. Since data manipulation, preparation, and cleaning is such an important skill in data analysis, pandas is one of the primary focuses of this book.

As a bit of background, I started building pandas in early 2008 during my tenure at AQR Capital Management, a quantitative investment management firm. At the time, I had a distinct set of requirements that were not well addressed by any single tool at my disposal:

- Data structures with labeled axes supporting automatic or explicit data alignment
- —this prevents common errors resulting from misaligned data and working with differently indexed data coming from different sources
- Integrated time series functionality
- The same data structures handle both time series data and non–time series data
- Arithmetic operations and reductions that preserve metadata
- Flexible handling of missing data
- Merge and other relational operations found in popular databases (SQL-based, for example)

I wanted to be able to do all of these things in one place, preferably in a language well suited to general-purpose software development. Python was a good candidate lan- guage for this, but at that time there was not an integrated set of data structures and tools providing this functionality. As a result of having been built initially to solve finance and business analytics problems, pandas features especially deep time series functionality and tools well suited for working with time-indexed data generated by business processes.

For users of the R language for statistical computing, the DataFrame name will be familiar, as the object was named after the similar R data.frame object. Unlike Python, data frames are built into the R programming language and its standard library. As a result, many features found in pandas are typically either part of the R core implementation or provided by add-on packages.

The pandas name itself is derived from panel data, an econometrics term for multidimensional structured datasets, and a play on the phrase Python data analysis itself.

3.3 matplotlib

matplotlib is the most popular Python library for producing plots and other two-dimensional data visualizations. It was originally created by John D. Hunter and is now maintained by a large team of developers. It is designed for creating plots suit- able for publication. While there are other visualization libraries available to Python programmers, matplotlib is the most widely used and as such has generally good integration with the rest of the ecosystem. I think it is a safe choice as a default visualisation tool.

4. NUMPY BASICS: ARRAYS AND VECTORIZED COMPUTATION

NumPy, short for Numerical Python, is a foundational package for numerical computing in Python. It is widely used as the standard for data exchange in scientific computing. Key features of NumPy include:

- ndarray: A multidimensional array for fast arithmetic operations and flexible broadcasting.
- Mathematical functions: Perform operations on entire arrays efficiently without loops.
- Data handling: Tools for reading/writing array data to disk and working with memory-mapped files.
- Advanced functionalities: Linear algebra, random number generation, and Fourier transform capabilities.
- C API: Connects NumPy with libraries written in C, C++, or FORTRAN, facilitating the integration of legacy codebases.

Understanding NumPy arrays is crucial for using array-oriented tools like pandas effectively. NumPy excels in areas such as:

- Fast vectorized operations for data manipulation.
- Common array algorithms (sorting, unique, set operations).
- Efficient descriptive statistics and data summarization.
- Data alignment and relational data manipulations.
- Conditional logic as array expressions.
- Group-wise data manipulations.

NumPy's design for efficiency on large data sets includes:

Storing data in contiguous memory blocks, reducing memory usage and overhead.

Performing complex computations on entire arrays without Python loops.

Performance comparison:

- NumPy: Multiplying a million-element array by 2 takes approximately 72.4 ms.

- Python list: The same operation with a list takes approximately 1.05 seconds.

NumPy-based algorithms are generally 10 to 100 times faster and use significantly less memory than their pure Python counterparts.

4.1. The NumPy ndarray: A Multidimensional Array Object

NumPy's ndarray is a fast, flexible container for large datasets in Python. It allows for batch computations using syntax similar to operations between scalar elements.

Example: Creating an ndarray

To create an ndarray, first import NumPy and generate a small array of random data:

```
import numpy as np

data = np.random.randn(2, 3)

print(data)
```

Output:

```
array([[-0.2047, 0.4789, -0.5194],

    [-0.5557, 1.9658, 1.3934]])
```

4.1.1 Performing Operations on Arrays

NumPy allows you to perform mathematical operations on whole blocks of data using similar syntax to scalar operations.

Multiplying elements:

```
print(data * 10)
```

Output:

```
array([[-2.0471, 4.7894, -5.1944],

    [-5.5573, 19.6578, 13.9341]])
```

Adding arrays:

```
print(data + data)
```

Output:

```
array([[-0.4094, 0.9579, -1.0389],
   [-1.1115, 3.9316, 2.7868]])
```

4.1.2 Array Attributes

Every ndarray has a shape, a tuple indicating the size of each dimension, and a dtype, an object describing the data type of the array.

Shape: Indicates the size of each dimension.

```
print(data.shape)
```

Output:

```
(2, 3)
```

dtype: Describes the data type of the array.

```
print(data.dtype)
```

Output:

```
dtype('float64')
```

4.1.3 Creating ndarrays

The easiest way to create an array is to use the array function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data.

From a list:

```
data1 = [6, 7.5, 8, 0, 1]

arr1 = np.array(data1)
```

```
print(arr1)
```

Output:

```
array([ 6. , 7.5, 8. , 0. , 1. ])
```

From nested lists:

Nested sequences, like a list of equal-length lists, will be converted into a multidimensional array:

```
data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

arr2 = np.array(data2)

print(arr2)
```

Output:

```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

```
print(arr2.ndim)
```

Output:

```
2
```

```
print(arr2.shape)
```

Output:

```
(2, 4)
```

4.1.4 Creating Arrays with Functions

NumPy provides several functions for creating arrays with specific values or shapes:

zeros: Creates an array of all zeros.

```
print(np.zeros(10))
```

Output:

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

empty: Creates an array without initializing its values.

```
print(np.empty((2, 3, 2)))
```

Output:

```
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],
       [[0., 0.],
        [0., 0.],
        [0., 0.]]])
```

arange: Creates an array with a range of values.

```
print(np.arange(15))
```

Output:

```
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14])
```

4.1.5 Data Types for ndarrays

The data type or dtype is a special object containing the information the ndarray needs to interpret a chunk of memory as a particular type of data.

```
arr1 = np.array([1, 2, 3], dtype=np.float64)

arr2 = np.array([1, 2, 3], dtype=np.int32)

print(arr1.dtype)
```

Output:

```
dtype('float64')
```

```
print(arr2.dtype)
```

Output:

dtype('int32')

4.1.6 Type Conversion

You can explicitly convert or cast an array from one dtype to another using ndarray's astype method.

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

print(arr.dtype)
```

Output:

dtype('int64')

```
float_arr = arr.astype(np.float64)

print(float_arr.dtype)
```

Output:

dtype('float64')

4.1.7 Arithmetic with NumPy Arrays

Arrays enable you to express batch operations on data without writing any for loops. This is known as vectorization.

Element-wise operations:

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])

print(arr * arr)
```

Output:

```
array([[ 1.,  4.,  9.],

    [16., 25., 36.]])
```

```
print(arr - arr)
```

Output:

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Scalar operations:

```
print(1 / arr)
```

Output:

```
array([[1.        , 0.5       , 0.33333333],
       [0.25      , 0.2       , 0.16666667]])
```

```
print(arr ** 0.5)
```

Output:

```
array([[1.        , 1.41421356, 1.73205081],
       [2.        , 2.23606798, 2.44948974]])
```

Comparisons:

```
arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
print(arr2 > arr)
```

Output:

```
array([[False,  True, False],
       [ True, False,  True]])
```

4.1.8 Indexing and Slicing

NumPy array indexing is a rich topic, as there are many ways you may want to select a subset of your data or individual elements.

Basic indexing:

```
arr = np.arange(10)

print(arr[5])
```

Output:

```
5
```

```
print(arr[5:8])
```

Output:

```
array([5, 6, 7])
```

Higher-dimensional arrays:

In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays.

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(arr2d[2])
```

Output:

```
array([7, 8, 9])
```

print(arr2d[0, 2])

Output:

```
3
```

4.1.9 Boolean Indexing

Boolean indexing allows you to select data based on the values of another array.

Using a condition:

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe',
'Joe'])
```

```
data = np.random.randn(7, 4)

print(names == 'Bob')
```

Output:

```
array([ True, False, False,  True, False, False, False])
```

```
print(data[names == 'Bob'])
```

Output:

```
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.669 , -0.4386, -0.5397,  0.477 ]])
```

Negating a condition:

You can use != or negate the condition using ~ to select everything except a certain value.

```
print(data[~(names == 'Bob')])
```

Output:

```
array([[ 1.0072, -1.2962,  0.275 ,  0.2289],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 3.2489, -1.0212, -0.5771,  0.1241],
       [ 0.3026,  0.5238,  0.0009,  1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Multiple conditions:

Use boolean arithmetic operators like & (and) and | (or) to combine multiple conditions.

```
mask = (names == 'Bob') | (names == 'Will')

print(data[mask])
```

Output:

```
array([[ 0.0929,  0.2817,  0.769 ,  1.2464],
       [ 1.3529,  0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397,  0.477 ],
       [ 3.2489, -1.0212, -0.5771,  0.1241]])
```

Setting values with boolean arrays:

You can set values in an array based on a condition.

```
data[data < 0] = 0
print(data)
```

Output:

```
array([[0.0929, 0.2817, 0.769 , 1.2464],
       [1.0072, 0.    , 0.275 , 0.2289],
       [1.3529, 0.8864, 0.    , 0.    ],
       [1.669 , 0.    , 0.    , 0.477 ],
       [3.2489, 0.    , 0.    , 0.1241],
       [0.3026, 0.5238, 0.0009, 1.3438],
       [0.    , 0.    , 0.    , 0.    ]])
```

4.2. Array-Oriented Programming with Arrays

Using NumPy arrays allows you to express many data processing tasks as concise array expressions, replacing explicit loops with array expressions, known as vectorization. Vectorized operations are often much faster than their pure Python equivalents, especially in numerical computations.

Example: Vectorized Computation

Evaluate the function $\sqrt{x^2 + y^2}$ across a grid of values:

```
import numpy as np

points = np.arange(-5, 5, 0.01)

xs, ys = np.meshgrid(points, points)

z = np.sqrt(xs ** 2 + ys ** 2)
```

Output:

```
array([[7.0711, 7.064 , 7.0569, ..., 7.0499, 7.0569, 7.064 ],
    [7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569],
    [7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
    ...,
    [7.0499, 7.0428, 7.0357, ..., 7.0286, 7.0357, 7.0428],
    [7.0569, 7.0499, 7.0428, ..., 7.0357, 7.0428, 7.0499],
    [7.064 , 7.0569, 7.0499, ..., 7.0428, 7.0499, 7.0569]])
```

4.2.1 Expressing Conditional Logic as Array Operations

The numpy.where function is a vectorized version of the ternary expression x if condition else y.

```
xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])

yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])

cond = np.array([True, False, True, True, False])

result = np.where(cond, xarr, yarr)

print(result)
```

Output

array([1.1, 2.2, 1.3, 1.4, 2.5])

Replace positive values with 2 and negative values with -2:

arr = np.random.randn(4, 4)

result = np.where(arr > 0, 2, -2)

print(result)

Output:

```
array([[-2, -2, -2, -2],
       [ 2,  2, -2,  2],
       [ 2,  2,  2, -2],
       [ 2, -2,  2,  2]])
```

4.2.2 Mathematical and Statistical Methods

NumPy provides methods for computing statistics about an array or along an axis, such as sum, mean, and std.

```
arr = np.random.randn(5, 4)
print(arr.mean())
print(arr.sum())
```

Output:

```
0.19607051119998253
3.9214102239996507
```

Compute statistics along an axis:

```
print(arr.mean(axis=1))
print(arr.sum(axis=0))
```

Output:

```
array([ 1.022 , 0.1875, -0.502 , -0.0881, 0.3611])

array([ 3.1693, -2.6345, 2.2381, 1.1486])
```

4.2.3 Methods for Boolean Arrays

Boolean values are coerced to 1 (True) and 0 (False) in methods like sum, often used to count True values.

```
arr = np.random.randn(100)

print((arr > 0).sum())  # Number of positive values
```

Output:

```
42
```

4.2.4 Sorting

Sort arrays in-place with the sort method:

```
arr = np.random.randn(6)

arr.sort()

print(arr)
```

Output:

```
array([-0.8469, -0.4938, -0.1357, 0.6095, 1.24 , 1.43 ])
```

Sort along an axis in a multidimensional array:

```
arr = np.random.randn(5, 3)

arr.sort(axis=1)

print(arr)
```

Output:

```
array([[-0.2555, 0.6033, 1.2636],
[-0.9616, -0.4457, 0.4684],

[-1.8245, 0.6254, 1.0229],

[-0.3501, 0.0909, 1.1074],

[-1.7415, -0.8948, 0.218 ]])
```

4.2.5 Unique and Other Set Logic

Use np.unique to find sorted unique values in an array:

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

print(np.unique(names))
```

Output:

```
array(['Bob', 'Joe', 'Will'], dtype='<U4')
```

Check membership of values in one array within another using np.in1d:

```
values = np.array([6, 0, 0, 3, 2, 5, 6])

print(np.in1d(values, [2, 3, 6]))
```

Output:

```
array([ True, False, False,  True,  True, False,  True], dtype=bool)
```

4.2.6 Summary of Functions

Basic Array Statistical Methods:

- sum: Sum of elements.
- mean: Arithmetic mean.
- std, var: Standard deviation and variance.
- min, max: Minimum and maximum.
- argmin, argmax: Indices of minimum and maximum elements.
- cumsum: Cumulative sum.

- cumprod: Cumulative product.

Boolean Array Methods:

- any: Test if any value is True.
- all: Test if all values are True.

Set Operations:

- unique: Compute the sorted, unique elements.
- intersect1d: Compute the sorted, common elements.
- union1d: Compute the sorted union of elements.
- in1d: Boolean array indicating if each element of one array is in another.
- setdiff1d: Elements in one array not in another.
- setxor1d: Elements in either of the arrays, but not both.

5. PANDAS

Pandas is a key tool for data cleaning and analysis in Python, providing data structures and manipulation tools designed for efficiency and ease of use. It is often used alongside NumPy, SciPy, statsmodels, scikit-learn, and matplotlib.

Key Features of pandas:

- Tabular and Heterogeneous Data: Unlike NumPy, which is optimized for homogeneous numerical data, pandas excels in handling tabular and diverse datasets.
- Integration with Other Libraries: Pandas integrates well with numerical computing and data visualization tools.
- Array-Based Functions: It adopts many of NumPy's idiomatic styles, including array-based functions and operations that avoid for loops.

Since its open-source launch in 2010, pandas has grown significantly, with over 800 contributors enhancing its capabilities for various real-world applications.

Importing pandas

Common conventions for importing pandas in code:

import pandas as pd

from pandas import Series, DataFrame

pd. refers to pandas, and importing Series and DataFrame directly is convenient due to their frequent use.

5.1. Introduction to pandas Data Structures

To get started with pandas, you will need to get comfortable with its two workhorse data structures: Series and DataFrame. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

5.1.1 Series

A Series is a one-dimensional array-like object with an associated array of data labels, called its index. It can hold a sequence of values similar to NumPy types.

Example:

```
import pandas as pd

obj = pd.Series([4, 7, -5, 3])

print(obj)
```

Output:

```
0   4

1   7

2  -5

3   3

dtype: int64
```

A Series can also be created with custom index labels:

```
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

print(obj2)
```

Output:

```
d   4

b   7

a  -5

c   3

dtype: int64
```

Series supports array-like operations, preserving the index-value link:

```
obj2[obj2 > 0]  # Filter positive values

obj2 * 2  # Scalar multiplication
```

Series can be created from dictionaries:

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah':
5000}

obj3 = pd.Series(sdata)
```

Output:

```
Ohio     35000

Oregon   16000

Texas    71000

Utah      5000

dtype: int64
```

5.1.2 DataFrame

A DataFrame is a two-dimensional table of data with labeled axes (rows and columns). Each column can be of a different type (numeric, string, boolean, etc.).

Example:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada',
'Nevada'],

        'year': [2000, 2001, 2002, 2001, 2002, 2003],

        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}

frame = pd.DataFrame(data)

print(frame)
```

Output:

```
  state  year  pop
0   Ohio  2000  1.5
1   Ohio  2001  1.7
2   Ohio  2002  3.6
3 Nevada  2001  2.4
4 Nevada  2002  2.9
5 Nevada  2003  3.2
```

Columns can be selected and manipulated:

```
frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'])
frame2['debt'] = 16.5
```

Output:

```
  year  state  pop  debt
0  2000   Ohio  1.5  16.5
1  2001   Ohio  1.7  16.5
2  2002   Ohio  3.6  16.5
3  2001  Nevada  2.4  16.5
4  2002  Nevada  2.9  16.5
5  2003  Nevada  3.2  16.5
```

5.1.3 Index Objects

Index objects hold the axis labels and metadata. They are immutable and provide methods for set operations.

Example:

```
obj = pd.Series(range(3), index=['a', 'b', 'c'])

index = obj.index

print(index)
```

Output:

```
Index(['a', 'b', 'c'], dtype='object')
```

Index objects support operations like append, difference, intersection, and union.

Pandas provides powerful and flexible data structures, Series and DataFrame, that make data manipulation and analysis easy and efficient. These structures integrate well with NumPy and other analytical libraries, providing robust tools for handling both homogeneous and heterogeneous data.

5.2. Essential Functionality

This section covers the core mechanics of interacting with data in pandas Series and DataFrames. We will highlight key features and functionalities, keeping our focus on the most essential aspects for data analysis and manipulation using pandas.

5.2.1 Reindexing

Reindexing creates a new object with data conforming to a new index.

Example:

```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])

obj.reindex(['a', 'b', 'c', 'd', 'e'])
```

Output:

```
a   -5.3
```

```
b    7.2

c    3.6

d    4.5

e    NaN

dtype: float64
```

For time series, use methods like ffill for forward filling:

```
obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])

obj3.reindex(range(6), method='ffill')
```

Output:

```
0     blue

1     blue

2   purple

3   purple

4   yellow

5   yellow

dtype: object
```

DataFrame Reindexing:

```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'], columns=['Ohio', 'Texas', 'California'])

frame.reindex(['a', 'b', 'c', 'd'])
```

Output:

```
Ohio    Texas  California
```

```
a    0.0   1.0    2.0

b    NaN   NaN    NaN

c    3.0   4.0    5.0

d    6.0   7.0    8.0
```

5.2.2 Dropping Entries from an Axis

Drop Entries with drop method:

```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])

obj.drop('c')
```

Output:

```
a   0.0

b   1.0

d   3.0

e   4.0

dtype: float64
```

For DataFrames, drop entries from rows or columns:

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)), index=['Ohio', 
'Colorado', 'Utah', 'New York'], columns=['one', 'two', 'three', 
'four'])

data.drop(['Colorado', 'Ohio'])
```

Output:

```
  one  two  three  four

Utah  8   9   10   11

New York  12   13   14   15
```

5.2.3 Indexing, Selection, and Filtering

Series Indexing:

```
obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])

obj['b']

obj[2:4]

obj[['b', 'a', 'd']]
```

Outputs:

```
b   1.0

dtype: float64

c   2.0

d   3.0

dtype: float64

b   1.0

a   0.0

d   3.0

dtype: float64
```

DataFrame Indexing:

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)), index=['Ohio',
'Colorado', 'Utah', 'New York'], columns=['one', 'two', 'three',
'four'])

data['two']

data[['three', 'one']]

data[data['three'] > 5]
```

Outputs:

```
Ohio      1

Colorado   5

Utah      9

New York  13

Name: two, dtype: int64


three  one

Ohio   2   0

Colorado  6   4

Utah  10   8

New York  14  12


one  two  three  four

Colorado  4  5  6  7

Utah  8  9  10  11

New York  12 13 14 15
```

5.2.4 Integer Indexes

Handling Integer Indexes:

```
ser = pd.Series(np.arange(3.))

ser.iloc[-1]
```

Output:

```
2.0
```

Use loc and iloc for precise indexing:

```
ser.iloc[:1]

ser.loc[:1]

ser.iloc[:1]
```

Outputs:

```
0    0.0

dtype: float64


0    0.0

1    1.0

dtype: float64


0    0.0

dtype: float64
```

5.2.5 Arithmetic and Data Alignment

Arithmetic with Different Indexes:

```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])

s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e',
'f', 'g'])

s1 + s2
```

Output:

```
a.   5.2
b.   1.1
c.   NaN
d.   0.0
e.   NaN
f.   NaN
```

dtype: float64

DataFrame and Series Operations:

```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'), index=['Ohio', 'Texas', 'Colorado'])

df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'), index=['Utah', 'Ohio', 'Texas', 'Oregon'])

df1 + df2
```

Output:

```
  b  c  d  e
Colorado NaN NaN NaN NaN
Ohio     3.0 NaN 6.0 NaN
Oregon   NaN NaN NaN NaN
Texas    9.0 NaN 12.0 NaN
Utah     NaN NaN NaN NaN
```

Handling Missing Values in Arithmetic:

```
df1.add(df2, fill_value=0)
```

Output:

```
  a  b  c  d  e
```

```
0   0.0  2.0 4.0 6.0 4.0

1   9.0 5.0 13.0 15.0 9.0

2  18.0 20.0 22.0 24.0 14.0

3  15.0 16.0 17.0 18.0 19.0
```

This concise guide covers the core functionality of pandas for effective data manipulation and analysis. For more advanced features, explore pandas documentation and additional resources.

6. DATA LOADING, STORAGE, AND FILE FORMATS

Accessing data is a necessary first step for using most of the tools in this book. I'm going to be focused on data input and output using pandas, though there are numer- ous tools in other libraries to help with reading and writing data in various formats.

Input and output typically falls into a few main categories: reading text files and other more efficient on-disk formats, loading data from databases, and interacting with net- work sources like web APIs.

6.1. Reading and Writing Data in Text Format

pandas offers several functions to read tabular data into a DataFrame. Commonly used functions include read_csv and read_table.

Table of Parsing Functions:

Function	Description
read_csv	Load delimited data from a file, URL, or file-like object.
read_table	Load delimited data from a file, URL, or file-like object.
read_fwf	Read data in fixed-width column format.
read_clipboard	Read data from the clipboard.
read_excel	Read tabular data from an Excel file.
read_hdf	Read HDF5 files written by pandas.
read_html	Read all tables from an HTML document.
read_json	Read data from a JSON string representation.
read_pickle	Read an object stored in Python pickle format.
read_sql	Read SQL query results as a DataFrame.
read_stata	Read a dataset from Stata file format.

read_feather Read the Feather binary file format.

Key Concepts in Reading Data

Indexing

Specify one or more columns as the DataFrame index.

Type Inference and Data Conversion

Custom value conversions and handling missing value markers.

Datetime Parsing

Combine date and time information spread over multiple columns.

Iterating

Support for reading large files in chunks.

Handling Unclean Data

Skip rows, footers, comments, and handle special numeric formats.

Examples

Reading a CSV file:

```
df = pd.read_csv('examples/ex1.csv')
```

Specifying column names:

```
pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd',
'message'])
```

Using a column as an index:

```
pd.read_csv('examples/ex2.csv', names=['a', 'b', 'c', 'd',
'message'], index_col='message')
```

Handling hierarchical indexes:

```
parsed = pd.read_csv('examples/csv_mindex.csv', index_col=['key1', 'key2'])
```

Reading data with whitespace as a delimiter:

```
result = pd.read_table('examples/ex3.txt', sep='\s+')
```

Skipping rows:

```
pd.read_csv('examples/ex4.csv', skiprows=[0, 2, 3])
```

Handling missing values:

```
result = pd.read_csv('examples/ex5.csv', na_values=['NULL'])
```

Common Arguments for read_csv and read_table:

Argument	Description
path	Location of the file.
sep	Delimiter to use.
header	Row number to use as column names.
index_col	Columns to use as the row index.
names	Column names for the result.
skiprows	Rows to skip at the beginning of the file.
na_values	Values to consider as NA.
comment	Character(s) to split comments off the end of lines.
parse_dates	Attempt to parse data to datetime.
dayfirst	Treat dates as international format.
nrows	Number of rows to read from the beginning.
iterator	Return a TextParser object for reading file piecemeal.

chunksize	Size of file chunks for iteration.
skip_footer	Lines to ignore at the end of the file.
encoding	Text encoding for Unicode.
squeeze	If the parsed data contains one column, return a Series.
thousands	Separator for thousands.

6.2 Reading Large Files in Chunks

To read large files, use the chunksize parameter to read in small pieces:

```
chunker = pd.read_csv('examples/ex6.csv', chunksize=1000)
```

Iterate over chunks:

```
tot = pd.Series([])

for piece in chunker:

    tot = tot.add(piece['key'].value_counts(), fill_value=0)

tot = tot.sort_values(ascending=False)
```

6.3. Writing Data to Text Format

Export data to a CSV file:

```
data.to_csv('examples/out.csv')
```

Custom delimiters and handling missing values:

```
data.to_csv(sys.stdout, sep='|', na_rep='NULL')
```

Disable row and column labels:

```
data.to_csv(sys.stdout, index=False, header=False)
```

Write a subset of columns:

```
data.to_csv(sys.stdout, index=False, columns=['a', 'b', 'c'])
```

Write a Series to a CSV file:

```
ts.to_csv('examples/tseries.csv')
```

Working with Delimited Formats

For files with custom delimiters, use Python’s built-in csv module:

```
import csv

with open('examples/ex7.csv') as f:

    reader = csv.reader(f)

    for line in reader:

        print(line)
```

Define a custom CSV dialect:

```
class my_dialect(csv.Dialect):

    lineterminator = '\n'

    delimiter = ';'

    quotechar = '"'

    quoting = csv.QUOTE_MINIMAL


with open('mydata.csv', 'w') as f:

    writer = csv.writer(f, dialect=my_dialect)

    writer.writerow(('one', 'two', 'three'))

    writer.writerow(('1', '2', '3'))

    writer.writerow(('4', '5', '6'))

    writer.writerow(('7', '8', '9'))
```

This concise guide provides an overview of reading and writing data in text format using pandas, along with handling common data processing scenarios.

7. DATA CLEANING AND PREPARATION

During data analysis and modeling, a significant portion of time is spent on data preparation: loading, cleaning, transforming, and rearranging data. This can take up 80% or more of an analyst's time. Data may not always be in the right format for a task, leading researchers to use general-purpose programming languages like Python, Perl, R, or Java, or Unix tools like sed or awk for ad hoc processing. Fortunately, pandas, along with Python's built-in features, provides a high-level, flexible, and fast set of tools for data manipulation.

If you encounter a data manipulation need not covered in this book or the pandas library, consider sharing your use case on Python mailing lists or the pandas GitHub site. The design and implementation of pandas are often driven by real-world application needs.

This chapter discusses tools for handling missing data, duplicate data, string manipulation, and other analytical transformations. The next chapter focuses on combining and rearranging datasets in various ways..

7.1. Handling Missing Data

Missing data is common in data analysis. Pandas aims to make handling missing data easy, with descriptive statistics excluding missing data by default. Pandas represents missing numeric data as NaN (Not a Number).

```
string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])
```

```
string_data.isnull()
```

Pandas uses the term NA (not available) for missing data, which can indicate data that does not exist or was not observed. The built-in Python None is also treated as NA in object arrays.

```
string_data[0] = None
```

```
string_data.isnull()
```

Table below lists some functions related to missing data handling.

Argument	Description

dropna	Filter out missing data
fillna	Fill in missing data
isnull	Return boolean values indicating missing data
notnull	Negation of isnull

7.1.1 Filtering Out Missing Data

Use dropna to filter out missing data in Series and DataFrames.

```
data = pd.Series([1, np.nan, 3.5, np.nan, 7])

data.dropna()



data = pd.DataFrame([[1., 6.5, 3.], [1., np.nan, np.nan], [np.nan,
np.nan, np.nan], [np.nan, 6.5, 3.]])

data.dropna()
```

You can specify to drop rows or columns with all or any NA values using how and axis parameters.

```
data.dropna(how='all')

data.dropna(axis=1, how='all')
```

For time series data, use thresh to specify the minimum number of non-NA values required.

```
df.dropna(thresh=2)
```

7.1.2 Filling In Missing Data

Use fillna to fill in missing data with a constant, a dictionary, or through interpolation methods like ffill.

```
df.fillna(0)

df.fillna({1: 0.5, 2: 0})
df.fillna(method='ffill')
```

```
data.fillna(data.mean())
```

Table below provides a reference for fillna arguments.

Argument	Description
value	Scalar value or dict-like object to fill missing values
method	Interpolation method, e.g., 'ffill'
axis	Axis to fill on
inplace	Modify the calling object without producing a copy
limit	Maximum number of consecutive periods to fill

7.2. Data Transformation

So far in this chapter we've been concerned with rearranging data. Filtering, cleaning, and other transformations are another class of important operations.

7.2.1 Removing Duplicates

Duplicate rows in a DataFrame can be identified and removed using duplicated and drop_duplicates.

```
data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'], 'k2': [1,
1, 2, 3, 3, 4, 4]})

data.duplicated()

data.drop_duplicates()

data.drop_duplicates(['k1'])

data.drop_duplicates(['k1', 'k2'], keep='last')
```

7.2.2 Transforming Data Using a Function or Mapping

To transform data, you can use the map method on a Series with a function or dictionary.

```
data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon', 'Pastrami', 'corned beef', 'Bacon', 'pastrami', 'honey ham', 'nova lox'], 'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})

meat_to_animal = {'bacon': 'pig', 'pulled pork': 'pig', 'pastrami': 'cow', 'corned beef': 'cow', 'honey ham': 'pig', 'nova lox': 'salmon'}

data['animal'] = data['food'].str.lower().map(meat_to_animal)
```

7.2.3 Replacing Values

Use the replace method to replace values in a Series or DataFrame.

```
data = pd.Series([1., -999., 2., -999., -1000., 3.])

data.replace(-999, np.nan)

data.replace([-999, -1000], np.nan)

data.replace([-999, -1000], [np.nan, 0])

data.replace({-999: np.nan, -1000: 0})
```

7.2.4 Renaming Axis Indexes

Axis labels can be transformed using the map method or rename.

```
data = pd.DataFrame(np.arange(12).reshape((3, 4)), index=['Ohio', 'Colorado', 'New York'], columns=['one', 'two', 'three', 'four'])

data.index = data.index.map(lambda x: x[:4].upper())

data.rename(index={'OHIO': 'INDIANA'}, columns={'three': 'peekaboo'})

data.rename(index={'OHIO': 'INDIANA'}, inplace=True)
```

7.2.5 Discretization and Binning

The cut and qcut functions are used for binning continuous data.

```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

```
bins = [18, 25, 35, 60, 100]

cats = pd.cut(ages, bins)

pd.value_counts(cats)

pd.cut(ages, [18, 26, 36, 61, 100], right=False)

group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']

pd.cut(ages, bins, labels=group_names)

data = np.random.rand(20)

pd.cut(data, 4, precision=2)

data = np.random.randn(1000)

cats = pd.qcut(data, 4)

pd.qcut(data, [0, 0.1, 0.5, 0.9, 1.])
```

7.2.6 Detecting and Filtering Outliers

Filter or transform outliers using array operations.

```
data = pd.DataFrame(np.random.randn(1000, 4))

col = data[2]

col[np.abs(col) > 3]

data[(np.abs(data) > 3).any(1)]

data[np.abs(data) > 3] = np.sign(data) * 3
```

7.2.7 Permutation and Random Sampling

Use numpy.random.permutation to permute rows and sample to select a random subset.

```
df = pd.DataFrame(np.arange(20).reshape((5, 4)))

sampler = np.random.permutation(5)

df.take(sampler)

df.sample(n=3)
```

```
choices = pd.Series([5, 7, -1, 6, 4])

choices.sample(n=10, replace=True)
```

7.2.8 Computing Indicator/Dummy Variables

Convert categorical variables into dummy/indicator variables using get_dummies.

```
df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'], 'data1':
range(6)})

pd.get_dummies(df['key'])

dummies = pd.get_dummies(df['key'], prefix='key')

df_with_dummy = df[['data1']].join(dummies)
```

For multi-category rows, use a more complex method:

```
mnames = ['movie_id', 'title', 'genres']

movies = pd.read_table('datasets/movielens/movies.dat', sep='::',
header=None, names=mnames)

all_genres = []

for x in movies.genres:

    all_genres.extend(x.split('|'))

genres = pd.unique(all_genres)

zero_matrix = np.zeros((len(movies), len(genres)))

dummies = pd.DataFrame(zero_matrix, columns=genres)

for i, gen in enumerate(movies.genres):

    indices = dummies.columns.get_indexer(gen.split('|'))

    dummies.iloc[i, indices] = 1

movies_windic = movies.join(dummies.add_prefix('Genre_'))
```

Use get_dummies with cut for discretization:

```
values = np.random.rand(10)
```

```
bins = [0, 0.2, 0.4, 0.6, 0.8, 1]

pd.get_dummies(pd.cut(values, bins))
```

7.3. String Manipulation

Python is popular for raw data manipulation due to its ease of use for string and text processing with built-in string methods. For complex pattern matching, regular expressions are used. Pandas enhances this by allowing the application of string methods and regular expressions on entire arrays of data, while also handling missing data efficiently.

7.3.1 String Object Methods

Built-in string methods in Python are often sufficient for string manipulation. For example, split breaks a comma-separated string into pieces:

```
val = 'a,b, guido'

val.split(',')

# Output: ['a', 'b', ' guido']
```

You can combine split with strip to remove whitespace:

```
pieces = [x.strip() for x in val.split(',')]

# Output: ['a', 'b', 'guido']
```

To concatenate these substrings with a delimiter, use join:

```
'::'.join(pieces)

# Output: 'a::b::guido'
```

Other methods include substring location with in, index, and find:

```
'guido' in val

# Output: True

val.index(',')

# Output: 1
```

```
val.find(':')

# Output: -1
```

count returns the number of occurrences of a substring:

```
val.count(',')

# Output: 2
```

replace substitutes one pattern for another:

```
val.replace(',', '::')

# Output: 'a::b:: guido'

val.replace(',', '')

# Output: 'ab guido'
```

Summary of String Methods

- count: Returns the number of non-overlapping occurrences of a substring.
- endswith: Returns True if the string ends with the specified suffix.
- startswith: Returns True if the string starts with the specified prefix.
- join: Uses the string as a delimiter to concatenate a sequence of other strings.
- index: Returns the position of the first character in a substring if found; raises ValueError if not found.
- find: Returns the position of the first character of the first occurrence of a substring; returns -1 if not found.
- rfind: Returns the position of the first character of the last occurrence of a substring; returns -1 if not found.
- replace: Replaces occurrences of a substring with another string.
- strip, rstrip, lstrip: Trims whitespace, including newlines.
- split: Breaks the string into a list of substrings using the specified delimiter.
- lower: Converts alphabet characters to lowercase.
- upper: Converts alphabet characters to uppercase.
- casefold: Converts characters to lowercase and regional-specific comparable forms.

- ljust, rjust: Left or right justify the string, padding with spaces or another fill character.

7.3.2 Regular Expressions

Regular expressions (regex) provide a flexible way to search or match complex string patterns in text. Python's built-in re module is used for this purpose, with functions for pattern matching, substitution, and splitting.

Example: Splitting Strings

To split a string by a variable number of whitespace characters:

```
import re

text = "foo\tbar\t baz \tqux"

re.split('\s+', text)

# Output: ['foo', 'bar', 'baz', 'qux']
```

Compiling Regex

You can compile a regex for reuse:

```
regex = re.compile('\s+')

regex.split(text)

# Output: ['foo', 'bar', 'baz', 'qux']
```

Finding Patterns

To find all matches of the regex in a text:

```
regex.findall(text)

# Output: ['\t', '\t ', ' \t']
```

Match and Search

- findall returns all matches.
- search returns the first match.

- match matches only at the beginning of the string.

Example to find email addresses:

```
text = """Dave dave@google.com Steve steve@gmail.com

Rob rob@gmail.com Ryan ryan@yahoo.com """

pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'

regex = re.compile(pattern, flags=re.IGNORECASE)

regex.findall(text)

# Output: ['dave@google.com', 'steve@gmail.com', 'rob@gmail.com',
'ryan@yahoo.com']
```

Substitution

To replace occurrences of a pattern with a new string:

```
regex.sub('REDACTED', text)

# Output: 'Dave REDACTED Steve REDACTED Rob REDACTED Ryan REDACTED'
```

Grouping

To segment email addresses into components:

```
pattern = r'([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})'

regex = re.compile(pattern, flags=re.IGNORECASE)

m = regex.match('wesm@bright.net')

m.groups()

# Output: ('wesm', 'bright', 'net')
```

Using findall with groups:

```
regex.findall(text)

# Output: [('dave', 'google', 'com'), ('steve', 'gmail', 'com'),
('rob', 'gmail', 'com'), ('ryan', 'yahoo', 'com')]
```

Using sub with groups:

```
regex.sub(r'Username: \1, Domain: \2, Suffix: \3', text)
```

```
# Output: 'Dave Username: dave, Domain: google, Suffix: com Steve
Username: steve, Domain: gmail, Suffix: com Rob Username: rob,
Domain: gmail, Suffix: com Ryan Username: ryan, Domain: yahoo,
Suffix: com'
```

Summary of Regular Expression Methods

- findall: Return all matching patterns in a string as a list.
- finditer: Like findall, but returns an iterator.
- match: Match pattern at the start of the string.
- search: Scan string for pattern match.
- split: Split string at each pattern occurrence.
- sub, subn: Replace all (sub) or first n (subn) pattern occurrences with a replacement.
- These methods allow efficient and powerful text processing, making regex an essential tool for data manipulation.

7.3.3 Vectorized String Functions in pandas

Cleaning a messy dataset often involves string manipulation, even with missing data:

```
data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',
'Rob': 'rob@gmail.com', 'Wes': np.nan}
```

```
data = pd.Series(data)
```

Handling Missing Data

Check for missing data:

data.isnull()

# Output:

# Dave   False

```
# Rob     False

# Steve   False

# Wes     True

# dtype: bool
```

String Operations

Use str attribute for string operations, which skip NA values:

```
data.str.contains('gmail')
```

```
# Output:

# Dave    False

# Rob     True

# Steve    True

# Wes     NaN

# dtype: object
```

Regular Expressions

Apply regex to each string:

```
pattern = '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

data.str.findall(pattern, flags=re.IGNORECASE)
```

```
# Output:

# Dave    [(dave, google, com)]

# Rob     [(rob, gmail, com)]

# Steve   [(steve, gmail, com)]

# Wes     NaN
# dtype: object
```

Element Retrieval

Retrieve elements using str.get or indexing:

```
matches = data.str.match(pattern, flags=re.IGNORECASE)

matches.str.get(1)

# Output:

# Dave    NaN

# Rob     NaN

# Steve   NaN

# Wes     NaN

# dtype: float64
```

String Slicing

Slice strings:

```
data.str[:5]

# Output:

# Dave    dave@

# Rob     rob@g

# Steve   steve

# Wes     NaN

# dtype: object
```

Vectorized String Methods

Table of common vectorized string methods:

Method	Description
cat	Concatenate strings element-wise with optional delimiter
contains	Return boolean array if each string contains pattern/regex
count	Count occurrences of pattern
extract	Extract one or more strings from a Series using regex with groups
endswith	Equivalent to x.endswith(pattern) for each element
startswith	Equivalent to x.startswith(pattern) for each element
findall	Compute list of all occurrences of pattern/regex for each string
get	Index into each element (retrieve i-th element)
isalnum	Equivalent to built-in str.isalnum
isalpha	Equivalent to built-in str.isalpha
isdecimal	Equivalent to built-in str.isdecimal
isdigit	Equivalent to built-in str.isdigit
islower	Equivalent to built-in str.islower
isnumeric	Equivalent to built-in str.isnumeric
isupper	Equivalent to built-in str.isupper
join	Join strings in each element with passed separator
len	Compute length of each string
lower, upper	Convert cases
match	Use re.match with regex on each element, returning matched groups as list
pad	Add whitespace to left, right, or both sides of strings

center	Equivalent to pad(side='both')
repeat	Duplicate values (e.g., s.str.repeat(3) is equivalent to x * 3 for each string)
replace	Replace occurrences of pattern/regex with another string
slice	Slice each string
split	Split strings on delimiter or regex
strip	Trim whitespace from both sides, including newlines
rstrip	Trim whitespace on right side
lstrip	Trim whitespace on left side

These methods simplify and enhance string processing in pandas.

8. SUMMARY

As we wrap up Unit 12 on Python for Data Analysis, let's take a moment to reflect on what we've covered and how these powerful tools and techniques can be applied in real-world data analysis projects. Python, with its simplicity and robust ecosystem, stands out as an excellent choice for both novice and seasoned data analysts.

Throughout this unit, we delved deep into the essentials of Python's data analysis libraries. Starting with NumPy, we explored how its multidimensional arrays and vectorized computation provide the backbone for numerical operations in Python. We learned how to create, manipulate, and transform data efficiently, setting a strong foundation for more complex operations.

Transitioning to pandas, we uncovered the power of data frames and series for handling structured data. We practiced various data manipulation techniques, from reindexing and sorting to dealing with missing values and filtering outliers. These skills are crucial for preparing data for analysis, ensuring that insights drawn are based on clean and well-structured datasets.

Our journey also included a dive into data visualization with matplotlib, where we learned how to create compelling visual narratives of our data. Whether it's through simple line graphs or complex scatter plots, the ability to visualize data not only aids in understanding trends and patterns but also in communicating findings effectively to any audience.

Moreover, we covered practical aspects of data analysis such as data loading, storage, and file formats, which are essential for managing data in different stages of a data analysis project. Understanding how to read, write, and store data efficiently ensures that our analysis processes are scalable and manageable.

Python's role as 'glue' in data analysis was also emphasized, showcasing its capability to integrate with other languages and tools, bridging the gap between data collection, analysis, and application. This versatility makes Python an invaluable tool in the toolkit of a data analyst.

To fully leverage the potential of Python for data analysis, practice is key. I encourage you to apply these techniques in various projects, experiment with different types of data, and continuously seek out new challenges. The real-world applications of these skills are vast, from finance and healthcare to marketing and beyond, where data-driven decisions are pivotal.

The journey through Python for Data Analysis has equipped you with the tools to tackle real-world data challenges. The skills you've acquired are not just applicable but also indispensable in today's data-driven world. Keep exploring, keep analysing, and let the data guide you to new insights and innovations.

9. GLOSSARY

- NumPy: A library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.
- pandas: An open-source data analysis and manipulation tool, built on top of the Python programming language. It offers data structures and operations for manipulating numerical tables and time series.
- matplotlib: A plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications.
- DataFrame: A two-dimensional, size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns) in pandas.
- Series: A one-dimensional array-like object containing a sequence of values and an associated array of data labels, called its index, in pandas.
- Index Objects: Immutable array structures that hold the axis labels and other metadata (like the axis name or names) in pandas.
- Universal Functions (ufunc): Functions that operate on ndarrays in an element-by-element fashion, supporting array broadcasting, typecasting, and several other standard features.
- Vectorized Computation: Operations performed on arrays primarily in NumPy and pandas that are optimized to perform complex computations on whole arrays without the need for Python for loops.
- CRUD Operations: An acronym for Create, Read, Update, and Delete. It is used to refer to the basic operations that applications need to perform on database data.
- Transactions: A sequence of database operations that are executed as a single unit. In SQL, transactions are managed to ensure data integrity and to handle errors in multi-step operations.
- ACID Properties: A set of properties that guarantee database transactions are processed reliably. ACID stands for Atomicity, Consistency, Isolation, and Durability.

- SQLite: A C library that provides a lightweight disk-based database that doesn't require a separate server process and allows access to the database using a nonstandard variant of the SQL query language.
- Boolean Indexing: A kind of indexing in pandas that uses actual boolean values rather than labels or integer locations.
- Sorting and Ranking: Data manipulation methods in pandas that rearrange the data based on some criteria or assign ranks to elements in the data according to their value.
- Data Cleaning: The process of detecting and correcting (or removing) corrupt or inaccurate records from a record set, table, or database.
- Data Transformation: Converting data from one format or structure into another. This often includes combining datasets, creating new variables, and converting types.
- Regular Expressions: Patterns used to match character combinations in strings, often utilized for searching or pattern matching capabilities.
- Flask: A micro web framework written in Python. It is classified as a microframework because it does not require particular tools or libraries and provides support for building web applications.
- Django: A high-level Python web framework that encourages rapid development and clean, pragmatic design. It is built by experienced developers and handles much of the hassle of web development.

10. SELF-ASSESSMENT QUESTIONS

1. _________ is a Python library used for working with arrays.
2. Pandas is used primarily for data _________ and manipulation.
3. _________ is a method in pandas used to drop entries from an axis.
4. A _________ function in NumPy applies operations element-wise.
5. The primary data structure in pandas used to represent tables of data is called _________.
6. The matplotlib library is typically used for _________ in Python.
7. To update the price of all records in a SQL database where the symbol is 'AAPL' to 130.00, the SQL command used is _________.
8. In a Flask web application, the @app.route decorator is used to _________ a URL to a view function.
9. _________ indexing allows you to select data in a DataFrame based on actual data values rather than purely on their positional index.
10. _________ is a process in data analysis where you remove or fill missing data entries.
11. Which of the following is not a Python data analysis library?
    A) NumPy
    B) pandas
    C) Ruby
    D) matplotlib
12. Which method is used to read data into a pandas DataFrame?
    A) pandas.read_csv()
    B) pandas.load_data()
    C) pandas.open_file()
    D) pandas.data_read()
13. Which of the following is a use of the groupby() function in pandas?
    A) To concatenate two DataFrames
    B) To divide data into distinct groups
    C) To plot data

D) To load data from a file

14. What does CRUD stand for?

A) Create, Read, Use, Delete

B) Create, Return, Update, Disable

C) Create, Read, Update, Delete

D) Connect, Read, Update, Delete

15. What does ACID stand for in database transactions?

A) Atomicity, Consistency, Isolation, Durability

B) Assembly, Consistency, Isolation, Durability

C) Atomicity, Connection, Isolation, Durability

D) Atomicity, Consistency, Integration, Durability

16. Which of the following is true about the fillna() method in pandas?

A) It removes rows with null values.

B) It replaces null values with a specified value.

C) It generates a report of null values.

D) It deletes the DataFrame.

17. Which library provides the ability to perform vectorized operations on data?

A) Flask

B) Django

C) NumPy

D) SQLite

18. Which SQL command is used to remove data from a database?

A) REMOVE

B) DELETE

C) DROP

D) CUT

19. In the context of pandas, what is a Series?

A) A single row of data

B) A single column of data

C) A type of database

D) A Python function

20. Which function is used to save changes in a database with Python's sqlite3 library?
    A) save()
    B) write()
    C) commit()
    D) push()

11. TERMINAL QUESTIONS

1. Explain the concept of "Python as Glue" in the context of data analysis.
2. Discuss the main reasons someone might choose not to use Python for data analysis.
3. Describe the architecture and primary features of NumPy arrays.
4. Elaborate on the importance of data types for ndarrays in NumPy.
5. What are universal functions in NumPy, and why are they significant?
6. Explain the process of indexing and slicing in NumPy arrays.
7. Discuss the concept of data alignment in pandas and its significance.
8. Describe how the pandas library handles missing data.
9. What are the advantages of using pandas for data manipulation and analysis over other libraries?
10. Explain the role and functionality of matplotlib in data visualization.
11. Write a Python script using NumPy to create a 3x3 matrix with values ranging from 1 to 9.
12. Using pandas, write a Python script to read a CSV file and display the first five rows.
13. Write a Python function to update a specific entry in a pandas DataFrame.
14. Using matplotlib, write a Python script to plot a simple line graph using arbitrary values.
15. Demonstrate how to perform a matrix multiplication using NumPy.
16. Write a Python script using pandas to handle missing values in a DataFrame by replacing them with the average of the remaining values.
17. Using NumPy, write a Python script to generate a four-dimensional array and access its second element.
18. Demonstrate the use of conditional logic as array operations in NumPy.
19. Write a Python script using pandas to create a DataFrame and add a new column that is a function of existing columns.
20. Using matplotlib, demonstrate how to create a scatter plot to visualize two variables from a DataFrame.

12. ANSWERS

Self-Assessment Questions

1. NumPy
2. Analysis
3. Drop
4. Universal
5. DataFrame
6. Plotting
7. "UPDATE stocks SET price = 130.00 WHERE symbol = 'AAPL'"
8. Map
9. Boolean
10. Cleaning
11. C) Ruby
12. A) pandas.read_csv()
13. B) To divide data into distinct groups
14. C) Create, Read, Update, Delete
15. A) Atomicity, Consistency, Isolation, Durability
16. B) It replaces null values with a specified value.
17. C) NumPy
18. B) DELETE
19. B) A single column of data
20. C) commit()

Terminal Questions

Long Answer Questions

1. Python as Glue - Refer to Unit 12, Section 2.1 "Python as Glue".
2. Why Not Python? - Refer to Unit 12, Section 2.2 "Why Not Python?".
3. NumPy Arrays Architecture - Refer to Unit 12, Section 4.1 "The NumPy ndarray: A Multidimensional Array Object".
4. Data Types for ndarrays - Refer to Unit 12, Section 4.1.2 "Data Types for ndarrays".

5. Universal Functions in NumPy - Refer to Unit 12, Section 4.2 "Universal Functions: Fast Element-Wise Array Functions".
6. Indexing and Slicing in NumPy - Refer to Unit 12, Section 4.1.4 "Basic Indexing and Slicing".
7. Data Alignment in pandas - Refer to Unit 12, Section 5.2.4 "Arithmetic and Data Alignment".
8. Handling Missing Data in pandas - Refer to Unit 12, Section 7.1 "Handling Missing Data".
9. Advantages of pandas - Refer to Unit 12, Section 5 "pandas".
10. Functionality of matplotlib - Refer to Unit 12, Section 3.3 "matplotlib".

Programming Challenges

11. 3x3 Matrix with NumPy - Refer to Unit 12, Section 4.1.1 "Creating ndarrays".
12. Reading CSV with pandas - Refer to Unit 12, Section 6.1 "Reading and Writing Data in Text Format".
13. Update DataFrame Entry - Refer to Unit 12, Section 5.2 "Essential Functionality".
14. Line Graph with matplotlib - Refer to Unit 12, Section 3.3 "matplotlib".
15. Matrix Multiplication with NumPy - Refer to Unit 12, Section 4.5 "Linear Algebra".
16. Handling Missing Values in pandas - Refer to Unit 12, Section 7.1.2 "Filling In Missing Data".
17. Four-dimensional Array in NumPy - Refer to Unit 12, Section 4.1 "The NumPy ndarray: A Multidimensional Array Object".
18. Conditional Logic as Array Operations - Refer to Unit 12, Section 4.3.1 "Expressing Conditional Logic as Array Operations".
19. Creating DataFrame and Adding Column - Refer to Unit 12, Section 5.2 "Essential Functionality".
20. Scatter Plot with matplotlib - Refer to Unit 12, Section 3.3 "matplotlib".

13. REFERENCES

- Mckinney, W. (2017). Python for data analysis : data wrangling with pandas, NumPy, and IPython. O'reilly Media, Inc., October.