Python and Statistics for Climate Informatics

Beginner's Guide for Python Data Analysis - 1

Dr. Jangho Lee

University of Illinois Chicago
Department of Earth and Environmental Sciences

November 2024

1 Basic Data Types

1.1 Basic Calculation with Python

Python can be used as a calculator, allowing us to perform calculations just like on a scientific calculator. It understands mathematical operations such as addition (+), subtraction (-), multiplication (*), and division (/). The symbols used for these operations are called **operators**.

For example, if we want to add the numbers 3 and 4, we can type the following command, and the result will be displayed in the terminal or the output section of your Integrated Development Environment (IDE).

Code 1.1

```
3 + 4
```

```
7
```

It is important to note that any incomplete **expressions** will return an error as the code below.

Code 1.2

```
12+3+
```

```
SyntaxError: invalid syntax
```

As you might have guessed, Python supports many operators beyond addition (+). These include:

- **Addition (+)**: Used to add two numbers.
- **Subtraction (-)**: Used to subtract one number from another.
- **Multiplication (*)**: Used to multiply two numbers.
- **Division (/)**: Used to divide one number by another, resulting in a float (decimal value).
- **Floor Division (//)**: Used to divide one number by another and return the largest whole number less than or equal to the result (integer division).
- **Modulus (%)**: Used to find the remainder of a division operation.
- **Exponentiation (**)**: Used to raise one number to the power of another.

Python also supports comparison and logical operators, which will be introduced later. When writing Python expressions, the order of operations follows the general rules of mathematics, specifically the PEMDAS rule: Parentheses, Exponents, Multiplication, Division, Addition, and Subtraction. This means Python evaluates expressions starting with parentheses, followed by exponents, then multiplication and division (from left to right), and finally addition and subtraction (also from left to right).

Example 1.1

> *If the atmospheric pressure at sea level is 1013.25 hPa, and it decreases at an average rate of 6.5 hPa per kilometer of altitude, what would the pressure be at an altitude of 5 kilometers?*

Code 1.3

```
1013.25 - (6.5*5)
```

```
980.75
```

1.2 Basic Built-In Functions in Python

In addition to operators, Python provides a wide range of built-in functions that can perform various operations on data. These functions are pre-defined by Python and can be used without the need for importing external libraries.

- **print()**: Displays output to the screen or terminal.
- **round()**: Rounds a number to the nearest integer or specified decimal places.
- **abs()**: Returns the absolute value of a number (removes any negative sign).
- **max()**: Returns the largest value from a list or series of values.
- **min()**: Returns the smallest value from a list or series of values.
- **sum()**: Calculates the total (sum) of all elements in a list or tuple.

Below are some examples using these functions.

Code 1.4

```
print("Hello World")
```

```
Hello World
```

Code 1.5

```
print(round(1.872, 2))
```

```
1.87
```

Code 1.6

```
print(max(1,5,7,3,10,2))
```

```
10
```

1.3 Assigning Variables

In Python, we can store values in **variables** to use them later in calculations or operations. A variable is simply a name that refers to a value. Assigning a value to a variable is straightforward in Python, using the `=` symbol in the format `variable_name = value`. This process is called **variable assignment** and allows us to reuse values without repeatedly typing them. Variables make programs more readable, flexible, and efficient, as you can store data and refer to it by name instead of hardcoding values into every calculation or operation.

For example, instead of writing the number `5` multiple times in a program, you can assign it to a variable like `x` and use `x` in calculations. If the value changes later, you only need to update the variable assignment, and the change will automatically apply throughout the program. This is especially useful in complex calculations or when working with dynamic data.

Additionally, variables allow us to:

- Store intermediate results: Instead of calculating the same value multiple times, you can store it in a variable and reuse it.
- Improve code readability: Variables with descriptive names (e.g., length, width) help make code self-explanatory and easier to understand.
- Make code more maintainable: By updating a variable's value in one place, you can avoid errors that might occur from changing hardcoded values throughout your code.

Here's a simple example to demonstrate variable assignment:

Code 1.7

```
a = 10
b = 50
print(a+b)
```

```
60
```

Note that the python syntax does not work the other way and it will raise a Syntax Error:

Code 1.8

```
10 = c
```

```
SyntaxError: cannot assign to literal here
```

Example 1.2

If the air temperature is 30°C, the relative humidity is 70%, and the wind speed is 2 m/s, calculate the heat index (HI) using the formula:

$$HI = T + 0.33 \cdot RH - 0.70 \cdot v - 4.00$$

where: T is the air temperature in Celsius, RH is the relative humidity as a decimal (e.g., 70% is 0.70), v is the wind speed in meters per second.

Code 1.9

```
# Assign variables
temperature = 30  # Air temperature in Celsius
humidity = 0.70     # Relative humidity as a decimal
wind_speed = 2      # Wind speed in m/s
# Calculate heat index using the formula
heat_index = temperature + 0.33 * humidity - 0.70 * wind_speed - 4.00
# Print the result
print(f"The heat index is: {heat_index:.2f} °C")
```

```
The heat index is: 24.83 °C
```

1.4 Data Types in Python

In Python, **data types** define the kind of value a variable can hold. Python is a **dynamically typed** language, meaning you don't need to explicitly declare the type of a variable. The type is determined automatically based on the value assigned.

Understanding data types is essential, as they affect how variables are stored, manipulated, and used in computations. Here, we'll explore Python's basic data types and their practical applications.

Basic Data Types in Python:

- **Integer (`int`)**: Represents whole numbers, positive or negative, without decimals.
- **Floating-Point (`float`)**: Represents numbers with decimals.
- **String (`str`)**: Represents text, enclosed in either single quotes (`' '`) or double quotes (`" "`).
- **Boolean (`bool`)**: Represents logical values: `True` or `False`.
- **None (`NoneType`)**: Represents the absence of a value or a null value.

Let's explore each data type in detail:

1.4.1 Integer (`int`)

Integers are whole numbers that can be positive, negative, or zero. They are typically used for counting, indexing, or performing arithmetic operations that don't require precision.

For example, integers are often used to:

- Count items (e.g., the number of students in a class).
- Represent days, years, or other whole units of time.
- Perform basic arithmetic operations, such as addition or subtraction.

Code 1.10

```python
# Examples of integers
x = 10
y = -5
z = x + y
print(z)  # Output: 5
```

```
5
```

1.4.2 Floating-Point (`float`)

Floating-point numbers represent real numbers with decimals. They are used when calculations require more precision than integers, such as in scientific computations or measuring continuous values.

Use cases for `float` include:

- Representing mathematical constants (e.g., $\pi = 3.14159$).
- Calculating areas, distances, or rates that involve decimals.
- Handling money-related calculations where precision matters.

Code 1.11

```
# Examples of floating-point numbers
pi = 3.14159
radius = 5
area = pi * (radius ** 2)
print("The area of the circle is:, round(area, 2)")
```

```
The area of the circle is: 78.54
```

1.4.3 String (`str`)

Strings are used to store text. They are essential for working with textual data, such as names, messages, or file paths. Strings can be enclosed in single quotes (`' '`) or double quotes (`" "`).

Some operations you can perform on strings include:

- Concatenation (joining two or more strings together).
- Slicing (extracting specific parts of a string).
- Formatting (inserting variables into strings, as discussed in the previous section).

Code 1.12

```
# Examples of strings
first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
print(f"Full Name: {full_name}")
```

```
Full Name: John Doe
```

1.4.4 Boolean (`bool`)

Booleans represent logical values: `True` or `False`. They are primarily used in conditions, comparisons, and logical operations.

Boolean values often arise from:

- Comparisons (e.g., `x > y` returns `True` or `False`).
- Logical operators (`and`, `or`, `not`).
- Control flow, such as in `if` statements.

Code 1.13

```
# Examples of booleans
is_sunny = True
is_rainy = False
print(is_sunny and is_rainy)  # Output: False
```

```
False
```

1.4.5 None (`NoneType`)

The `NoneType` represents the absence of a value. This is useful in scenarios where a variable is declared but doesn't yet hold meaningful data.

Common use cases for `None` include:

- Placeholder values in code.
- Indicating the absence of a result or error state in functions.

Code 1.14

```python
# Example of None
value = None
print(value)  # Output: None
```

```
None
```

1.4.6 Checking Data Types

Python provides the `type()` function to check the type of a variable. This is especially useful when debugging or validating input.

Code 1.15

```python
# Checking data types
x = 10
y = 3.14
z = "Python"
print(type(x))  # Output: <class 'int'>
print(type(y))  # Output: <class 'float'>
print(type(z))  # Output: <class 'str'>
```

```
<class 'int'>
<class 'float'>
<class 'str'>
```

1.5 String Formatting

In Python, **string formatting** is a way to insert variables, expressions, or other values into a string. It allows you to create dynamic and readable strings without manually concatenating parts. Python provides several ways to format strings, and the most common methods are:

- **f-strings (formatted string literals)**
- `str.format()` **method**
- **Old-style % formatting**

Let's explore each method in detail.

1.5.1 f-Strings (Recommended)

Introduced in Python 3.6, f-strings are a concise and readable way to format strings. You prefix the string with `f` or `F`, and variables or expressions are placed inside curly braces {}.

Code 1.16

```python
name = "Alice"
age = 25
print(f"My name is {name}, and I am {age} years old.")
```

```
My name is Alice, and I am 25 years old.
```

You can also use expressions inside f-strings:

Code 1.17

```python
length = 5
width = 3
area = length * width
print(f"The area of the rectangle is {area} square units.")
```

```
The area of the rectangle is 15 square units.
```

1.5.2 The `str.format()` Method

The `str.format()` method provides a more traditional way to format strings. Placeholders {} are used in the string, and values are inserted using the `.format()` method.

Code 1.18

```python
name = "Bob"
age = 30
print("My name is {}, and I am {} years old.".format(name, age))
```

```
My name is Bob, and I am 30 years old.
```

You can use positional or keyword arguments to format strings:

Code 1.19

```python
# Positional arguments
print("The dimensions are {} x {}.".format(10, 20))

# Keyword arguments
print("The dimensions are {length} x {width}.".format(length=10, width=20))
```

```
The dimensions are 10 x 20.
The dimensions are 10 x 20.
```

1.5.3 Old-Style % Formatting (Legacy)

The % operator is an older method of string formatting. While still supported, it is less readable and flexible than the other methods.

Code 1.20

```python
name = "Charlie"
age = 35
print("My name is %s, and I am %d years old." % (name, age))
```

```
My name is Charlie, and I am 35 years old.
```

Here, `%s` is used for strings, and `%d` is used for integers.

1.5.4 Formatting Numbers

Python string formatting methods can also handle numbers, allowing you to specify the number of decimal places or add padding for alignment.

Code 1.21

```python
value = 3.14159
print(f"Value rounded to two decimal places: {value:.2f}")
```

```
Value rounded to two decimal places: 3.14
```

2 Control Statements in Python

Control statements allow you to make decisions and repeat actions in your code. They form the foundation of most programming tasks by enabling dynamic and flexible behavior in programs. Python provides two primary types of control statements:

- **Conditional Statements:** Used to execute code based on specific conditions.
- **Loops:** Used to repeat a block of code multiple times.

Let's explore these in detail.

2.1 Conditional Statements: `if`, `elif`, `else`

Conditional statements allow your program to make decisions by evaluating conditions that result in either `True` or `False`. Python supports three main constructs for conditional statements: `if`, `elif`, and `else`.

2.1.1 The `if` Statement

The `if` statement is the simplest form of a conditional. It checks a condition, and if the condition evaluates to `True`, the block of code following the `if` statement is executed.

The general structure is as follows:

```
if condition:
    # Code to execute if the condition is True
```

This allows your program to behave differently depending on the input or context.

Code 2.1

```python
temperature = 30  # Temperature in °C

if temperature > 25:
    print("It's a hot day.")
```

```
It's a hot day.
```

In this example, the message "It's a hot day." is printed only if the condition `temperature > 25` is satisfied.

2.1.2 The `if-else` Statement

The `if-else` statement allows you to execute one block of code when the condition is `True`, and another block when the condition is `False`. This adds a second branch to handle alternative scenarios.

The general structure is:

```
if condition:
    # Code to execute if the condition is True
else:
    # Code to execute if the condition is False
```

Code 2.2

```python
humidity = 40  # Humidity in percentage

if humidity > 50:
    print("It's humid.")
else:
    print("The air is dry.")
```

```
The air is dry.
```

Here, the program checks whether the humidity is above 50%. If true, it prints "It's humid." Otherwise, it prints "The air is dry."

2.1.3 The `if-elif-else` Statement

The `if-elif-else` construct is used when there are multiple conditions to evaluate. Only the first condition that evaluates to `True` will execute. If none of the conditions are true, the `else` block runs.

The structure is:

```
if condition1:
    # Code to execute if condition1 is True
elif condition2:
    # Code to execute if condition2 is True
else:
    # Code to execute if all conditions are False
```

Code 2.3

```python
temperature = 15  # Temperature in °C

if temperature > 25:
    print("It's hot.")
elif temperature > 15:
    print("It's warm.")
else:
    print("It's cold.")
```

```
It's warm.
```

This example evaluates the temperature. If it's greater than 25, it prints "It's hot." If not, it checks if it's greater than 15 and prints "It's warm." Otherwise, it prints "It's cold."

2.2 Loops: `for` and `while`

Loops allow you to execute a block of code repeatedly. This is especially useful when working with sequences or when the same operation needs to be performed multiple times.

2.2.1 `for` Loop

A `for` loop is used to iterate over a sequence (such as a list, tuple, or range) and perform a block of code for each element.

The structure is:

```
for item in sequence:
    # Code to execute for each item in the sequence
```

Code 2.4

```python
# Temperatures recorded over 3 days
temperatures = [25, 28, 22]

# Print each temperature
for temp in temperatures:
    print(f"Temperature: {temp} °C")
```

```
Temperature: 25 °C
Temperature: 28 °C
Temperature: 22 °C
```

In this example, the `for` loop iterates over the list of temperatures and prints each value.

2.2.2 `while` Loop

A `while` loop executes as long as its condition remains `True`. This is useful when the number of iterations is not known in advance.

The structure is:

```
while condition:
    # Code to execute as long as the condition is True
```

Code 2.5

```python
# Count down from 5
count = 5

while count > 0:
    print(f"Countdown: {count}")
    count -= 1

print("Liftoff!")
```

```
Countdown: 5
Countdown: 4
Countdown: 3
Countdown: 2
Countdown: 1
Liftoff!
```

This example counts down from 5 to 1 and then prints ”Liftoff!”.

2.2.3 `break` and `continue` Statements

The `break` statement exits a loop immediately, while the `continue` statement skips the rest of the current iteration and moves to the next one.

Code 2.6

```python
# Find the first temperature above 25 °C
temperatures = [22, 24, 27, 23]

```

```python
for temp in temperatures:
    if temp > 25:
        print(f"First hot day: {temp} °C")
        break
```

```
First hot day: 27 °C
```

Code 2.7

```python
# Skip temperatures below 25 °C
temperatures = [22, 24, 27, 23]

for temp in temperatures:
    if temp < 25:
        continue
    print(f"Warm day: {temp} °C")
```

```
Warm day: 27 °C
```

2.3 Summary

Control statements in Python allow you to:

- Make decisions using `if`, `elif`, and `else`.
- Iterate over sequences or repeat actions using `for` and `while` loops.
- Use `break` and `continue` to manage loop execution.

These statements enable dynamic and efficient programming.

3 Functions in Python

A **function** is a reusable block of code that performs a specific task. Functions help to organize your code, avoid repetition, and make your programs more readable and efficient.

Python allows you to define your own functions using the `def` keyword.

3.1 Defining a Function

To create a function, you use the `def` keyword, followed by the function name, parentheses (which may include parameters), and a colon. The code inside the function must be indented.

The general structure is:

```
def function_name(parameters):
    # Code block (function body)
    return value  # Optional
```

Key components of a function:

- **`def`:** Keyword used to define the function.
- **Function Name:** The name you choose for your function, which should describe its purpose.
- **Parameters:** Values that can be passed into the function (optional).
- **Return Statement:** The value or result that the function outputs (optional).

3.2 A Basic Function

Let's define a simple function to calculate the square of a number.

Code 3.1

```python
# Define a function to calculate the square of a number
def square(number):
    return number * number

# Call the function
result = square(4)
print(result)
```

```
16
```

Here:

- The function `square()` takes one parameter, `number`.
- It multiplies the parameter by itself and returns the result.
- The function is then called using `square(4)`, and the result (16) is printed.

3.3 Functions with Multiple Parameters

Functions can take multiple parameters, separated by commas. These parameters allow you to perform more complex tasks.

For example, let's create a function to calculate the average of two temperatures:

Code 3.2

```python
# Define a function to calculate the average of two temperatures
def average_temperature(temp1, temp2):
    return (temp1 + temp2) / 2

# Call the function
average = average_temperature(25, 30)
print(f"The average temperature is {average} °C")
```

```
The average temperature is 27.5 °C
```

3.4 Functions with Default Parameters

You can define default values for function parameters. If a value is not provided when the function is called, the default value is used.

Let's modify the previous example to use a default value for the second temperature:

Code 3.3

```python
# Define a function with a default parameter
def average_temperature(temp1, temp2=25):
    return (temp1 + temp2) / 2

# Call the function with one argument
average = average_temperature(30)
print(f"The average temperature is {average} °C")
```

```
The average temperature is 27.5 °C
```

3.5 Returning Multiple Values

A function can return multiple values as a tuple. This is useful for performing multiple calculations at once.

For example, let's create a function that calculates both the sum and the product of two numbers:

Code 3.4

```python
# Define a function to calculate the sum and product
def sum_and_product(a, b):
    return a + b, a * b

# Call the function
result_sum, result_product = sum_and_product(4, 5)
print(f"Sum: {result_sum}, Product: {result_product}")
```

```
Sum: 9, Product: 20
```

3.6 The Importance of Functions

Functions are powerful tools for structuring your programs. They enable:

- Code reuse: Write a function once and use it multiple times.
- Modularity: Divide your program into smaller, manageable pieces.
- Readability: Make your code easier to understand by naming functions descriptively.
- Flexibility: Customize behavior with parameters and default values.

3.7 Summary

In this section, we covered:

- How to define and call functions using the `def` keyword.
- Passing arguments to functions, including default parameters.
- Returning values, including multiple values.

Functions are essential for building efficient and maintainable Python programs. They provide a way to encapsulate logic and make your code more organized and reusable.

4 Classes in Python

Classes are the cornerstone of object-oriented programming (OOP), a paradigm that models real-world entities in terms of data (attributes) and behavior (methods). Python's OOP capabilities allow you to create reusable, modular, and organized programs.

A **class** is a blueprint for creating objects, while an **object** is an instance of a class. This relationship enables abstraction and encapsulation of complex operations.

4.1 What is a Class?

A class is like a template that defines the properties and behaviors of a set of objects. For instance, you could create a class called `Weather` to represent weather data. Each `Weather` object would then store specific information about temperature and humidity.

A class in Python is defined using the `class` keyword. Inside a class, you can define:

- **Attributes:** Variables that store data about the object.
- **Methods:** Functions that define the behavior of the object.

The general syntax for a class is:

```
class ClassName:
    def __init__(self, parameters):
        # Initialize attributes
        self.attribute = value

    def method(self):
        # Define behavior
        pass
```

The `__init__` method is a special method known as a constructor. It initializes the attributes of the object when it is created.

4.2 Defining a Simple Class

Let's define a class called `Weather` to store and display weather data.

Code 4.1

```python
# Define a Weather class
class Weather:
    def __init__(self, temperature, humidity):
        self.temperature = temperature  # Temperature in °C
        self.humidity = humidity        # Humidity in percentage

# Create an object of the Weather class
today = Weather(temperature=30, humidity=70)

# Access attributes
print(f"Today's temperature: {today.temperature} °C")
print(f"Today's humidity: {today.humidity}%")
```

```
Today's temperature: 30 °C
Today's humidity: 70%
```

Explanation:

- `class Weather`: This defines the `Weather` class.
- `__init__`: This constructor initializes the `temperature` and `humidity` attributes for each object.
- `today`: This is an object (instance) of the `Weather` class, with temperature set to 30 and humidity set to 70.
- `self`: Represents the object itself, allowing you to access its attributes and methods.

4.3 Adding Methods to a Class

A class can include methods to define its behavior. Methods are functions defined inside a class that operate on the object's attributes.

Let's enhance the `Weather` class by adding a method to calculate the heat index, a measure of how hot it feels when relative humidity is factored in.

Code 4.2

```python
# Define a Weather class with a method
class Weather:
    def __init__(self, temperature, humidity):
        self.temperature = temperature
        self.humidity = humidity

    def calculate_heat_index(self):
        # Simplified heat index formula
        return self.temperature + 0.33 * self.humidity - 4

# Create an object and calculate the heat index
today = Weather(temperature=30, humidity=70)
heat_index = today.calculate_heat_index()

print(f"Heat Index: {heat_index:.2f} °C")
```

```
Heat Index: 49.10 °C
```

Explanation:

- `calculate_heat_index`: This method uses the object's `temperature` and `humidity` attributes to compute the heat index.
- `today.calculate_heat_index()`: Calls the method on the `today` object and returns the calculated value.

4.4 Encapsulation and Access Control

Encapsulation is the process of restricting access to an object's internal state and allowing it to be modified only through well-defined methods. In Python:

- Attributes are **public** by default, meaning they can be accessed directly.

- To make an attribute private, prefix it with __ (double underscore).

Example: *Code 4.3*

```python
# Define a class with private attributes
class Weather:
    def __init__(self, temperature, humidity):
        self.__temperature = temperature  # Private attribute
        self.humidity = humidity

    def get_temperature(self):
        return self.__temperature  # Access private attribute

# Create an object and access attributes
today = Weather(temperature=30, humidity=70)
print(f"Today's temperature: {today.get_temperature()} °C")
```

```
Today's temperature: 30 °C
```

This demonstrates how private attributes can be accessed only through methods like `get_temperature()`.

4.5 Inheritance

Inheritance allows you to define a new class based on an existing class. The new class (child) inherits attributes and methods from the parent class but can also have its own additional functionality.

Example: *Code 4.4*

```python
# Parent class
class Weather:
    def __init__(self, temperature, humidity):
        self.temperature = temperature
        self.humidity = humidity

# Child class
class WeatherStation(Weather):
    def __init__(self, temperature, humidity, station_name):
        super().__init__(temperature, humidity)  # Call parent constructor
        self.station_name = station_name

# Create an object of the child class
station = WeatherStation(25, 60, "Station A")
print(f"Station Name: {station.station_name}")
print(f"Temperature: {station.temperature} °C")
```

```
Station Name: Station A
Temperature: 25 °C
```

Explanation:

- `Weather`: Parent class that stores temperature and humidity.
- `WeatherStation`: Child class that inherits from `Weather` and adds the `station_name` attribute.
- `super()`: Calls the parent class's constructor to initialize inherited attributes.

4.6 Why Use Classes?

Classes provide several advantages:

- **Modularity:** Divide your program into smaller, reusable components.
- **Encapsulation:** Protect the internal state of objects.
- **Inheritance:** Reuse and extend existing code.
- **Abstraction:** Hide unnecessary details from the user.

4.7 Summary

In this section, we learned:

- How to define and use classes to model real-world entities.
- The importance of attributes, methods, encapsulation, and inheritance.
- How classes promote code reuse and modularity in Python.

Classes are essential for building structured and scalable Python programs, enabling you to model complex systems with ease.

5 Data Structures: Advanced Usage

In Python, data structures such as lists, tuples, sets, and dictionaries are versatile tools for organizing and manipulating data. This chapter explores advanced techniques for using these data structures, including nested structures, comprehensions, and practical applications.

5.1 Nested Data Structures

Data structures can be nested within each other to represent complex data. For instance, a list can contain dictionaries, a dictionary can store lists, and so on. This is particularly useful for working with tabular or hierarchical data.

5.1.1 Example: Nested List of Dictionaries

Let's store temperature and humidity readings for multiple cities:

Code 5.1

```python
# Weather data for multiple cities
weather_data = [
    {"city": "Chicago", "temperature": 22, "humidity": 60},
    {"city": "Los Angeles", "temperature": 25, "humidity": 50},
    {"city": "New York", "temperature": 18, "humidity": 70}
]

# Access data for each city
for entry in weather_data:
    print(f"City: {entry['city']}, "
          f"Temperature: {entry['temperature']} °C, "
          f"Humidity: {entry['humidity']}%")
```

```
City: Chicago, Temperature: 22 °C, Humidity: 60%
City: Los Angeles, Temperature: 25 °C, Humidity: 50%
City: New York, Temperature: 18 °C, Humidity: 70%
```

Explanation:

- The outer structure is a `list`, representing a collection of cities.
- Each item in the list is a `dictionary`, containing weather data for a specific city.
- Data is accessed using nested indexing and dictionary keys.

5.1.2 Example: Dictionary of Lists

Dictionaries can store lists as values. For instance, let's group temperature readings by city:

Code 5.2

```python
# Temperature readings grouped by city
city_temperatures = {
    "Chicago": [22, 23, 21],
    "Los Angeles": [25, 26, 24],
    "New York": [18, 19, 17]
}

# Calculate the average temperature for Chicago
```

```python
chicago_temps = city_temperatures["Chicago"]
average_temp = sum(chicago_temps) / len(chicago_temps)

print(f"Average temperature in Chicago: {average_temp:.2f} °C")
```

```
Average temperature in Chicago: 22.00 °C
```

Explanation:

- The dictionary's keys are city names, and the values are lists of temperature readings.
- The `sum()` and `len()` functions are used to calculate the average.

5.2 Comprehensions

Comprehensions provide a concise way to create lists, sets, or dictionaries. They are often used to transform data or filter elements.

5.2.1 List Comprehensions

A list comprehension is a compact way to create a new list by applying an operation to each element in an existing sequence.

Code 5.3

```python
# Convert temperatures from Celsius to Fahrenheit
temperatures_celsius = [22, 25, 18]
temperatures_fahrenheit = [(temp * 9/5) + 32 for temp in temperatures_celsius]

print(f"Temperatures in Fahrenheit: {temperatures_fahrenheit}")
```

```
Temperatures in Fahrenheit: [71.6, 77.0, 64.4]
```

5.2.2 Dictionary Comprehensions

Dictionary comprehensions create dictionaries from existing sequences.

Code 5.4

```python
# Map city names to temperature ranges
temperature_ranges = {
    city: max(temps) - min(temps)
    for city, temps in city_temperatures.items()
}

print(f"Temperature ranges: {temperature_ranges}")
```

```
Temperature ranges: {'Chicago': 2, 'Los Angeles': 2, 'New York': 2}
```

5.3 Sorting and Searching in Data Structures

Python provides efficient ways to sort and search through data structures. Let's explore these techniques.

5.3.1 Sorting a List

The `sorted()` function sorts a list in ascending or descending order.

Code 5.5

```python
# Sort cities by temperature
sorted_weather = sorted(
    weather_data, key=lambda x: x["temperature"], reverse=True
)

# Print sorted cities
for entry in sorted_weather:
    print(f"City: {entry['city']}, Temperature: {entry['temperature']} °C")
```

```
City: Los Angeles, Temperature: 25 °C
City: Chicago, Temperature: 22 °C
City: New York, Temperature: 18 °C
```

Explanation:

- The `key=lambda x: x["temperature"]` specifies that sorting should be based on the `temperature` key.
- The `reverse=True` parameter sorts in descending order.

5.3.2 Searching in a Dictionary

You can use dictionary methods like `get()` to retrieve values efficiently.

Code 5.6

```python
# Search for a city's temperature
city = "Chicago"
temperature = city_temperatures.get(city, "Not found")
print(f"Temperature readings for {city}: {temperature}")
```

```
Temperature readings for Chicago: [22, 23, 21]
```

5.4 Summary

In this section, we explored:

- Nested data structures, such as lists of dictionaries and dictionaries of lists.
- Comprehensions for concise data transformation.
- Sorting and searching techniques in Python.

These tools and techniques allow you to work with complex datasets effectively, making them indispensable for data analysis and real-world programming tasks.

6 Exercises

This chapter presents 20 advanced exercises to challenge your understanding of Python concepts such as control statements, data structures, functions, and string formatting. Each problem is designed to simulate real-world scenarios or advanced use cases.

6.1 Exercise Problems

Exercise 6.1

Write a Python program to determine if a given year is a leap year. A year is a leap year if it is divisible by 4 but not divisible by 100, unless it is also divisible by 400.

Exercise 6.2

Create a program to calculate the wind chill index given:

$$WCI = 13.12 + 0.6215T - 11.37v^{0.16} + 0.3965Tv^{0.16}$$

where T is the air temperature in $^\circ C$ and v is the wind speed in km/h. Test it with $T = -5$ and $v = 20$.

Exercise 6.3

Store temperature readings for a week in a nested dictionary, where the keys are the days of the week, and the values are dictionaries containing high and low temperatures. Write a program to:

- *Calculate the average high temperature.*
- *Find the day with the largest temperature difference.*

Exercise 6.4

Write a function `classify_pressure()` that takes an atmospheric pressure value (in hPa) and returns:

- *`"Low"` if the pressure is below 1000 hPa.*
- *`"Normal"` if the pressure is between 1000 and 1020 hPa.*
- *`"High"` if the pressure exceeds 1020 hPa.*

Use a list of sample pressures (`[995, 1015, 1030, 1005]`) to test your function.

Exercise 6.5

Given a list of wind speeds (`[12, 18, 5, 23, 35]`), write a program to:

- *Sort the wind speeds in descending order.*
- *Print the three highest wind speeds.*

Exercise 6.6

Define a function `calculate_dew_point()` that computes the dew point temperature using the formula:

$$T_d = T - \frac{100 - RH}{5}$$

where T is the temperature in $^\circ C$ and RH is the relative humidity. Test it with $T = 25$, $RH = 60$, and $T = 18$, $RH = 80$.

Exercise 6.7

Store weather station data in a dictionary where keys are station names, and values are dictionaries containing temperature, humidity, and wind speed. Write a program to:

- *Calculate the average temperature across all stations.*
- *Identify the station with the highest wind speed.*

Exercise 6.8

Write a program that accepts a list of temperatures (in $^\circ C$) and creates a new list with only those temperatures above 30 $^\circ C$. Use list comprehensions.

Exercise 6.9

Define a function `convert_weather_data()` that takes a list of temperatures in (@°@*)C and returns a dictionary where keys are the original temperatures and values are the corresponding temperatures in $^\circ F$.*

Exercise 6.10

Create a program to simulate a weather log. Store daily high and low temperatures in a list of tuples for one week. Write a function to:

- *Return the day with the highest high temperature.*
- *Return the day with the lowest low temperature.*

Exercise 6.11

Create a dictionary of precipitation data for 10 days (`[0, 5, 12, 0, 3, 8, 0, 7, 15, 0]`). Write a program to:

- *Count the number of dry days (precipitation = 0).*
- *Calculate the total precipitation for the period.*

Exercise 6.12

Write a Python program that simulates a weather alert system:

- *Input: Current temperature and humidity.*
- *Output: An alert if the heat index exceeds 40 $^\circ C$.*

Use the formula:

$$HI = T + 0.33 \cdot RH - 4$$

Exercise 6.13

Write a program to determine if a given list of temperatures is in ascending order (e.g., `[18, 20, 22, 24]`) or not.

Exercise 6.14

Create a program that generates a report of average monthly temperatures. Input: A dictionary where keys are months and values are lists of daily temperatures. Output: The average temperature for each month.

Exercise 6.15

Write a program that creates a set of unique temperature readings from a list of duplicates (e.g., `[22, 24, 22, 25, 24]`).

Exercise 6.16

Define a function `filter_cities()` that takes a dictionary of city weather data and returns a list of cities with temperatures above 25 °C and humidity below 50%.

Exercise 6.17

Create a program to calculate the pressure at a given altitude using the barometric formula:

$$P = P_0 \cdot \exp\left(-\frac{M \cdot g \cdot h}{R \cdot T}\right)$$

where $P_0 = 1013.25$ *hPa,* $M = 0.02896$ *kg/mol,* $g = 9.8$ m/s^2*,* $R = 8.314$ *J/(mol·K), and* $T = 293$ *K.*

Exercise 6.18

Write a Python program to:

- *Input a list of temperatures and humidity levels for five cities.*
- *Use a function to calculate the heat index for each city.*
- *Store the results in a dictionary and print it.*

Exercise 6.19

Write a program to reverse a list of temperatures (e.g., `[22, 24, 25, 20]`) without using the `reverse()` function.

Exercise 6.20

Using a dictionary, store the names and average temperatures of 5 cities. Write a program to find the city with the highest average temperature.

6.2 Solutions

Below are the solutions to the exercises provided in the previous chapter. Each solution is written in Python and includes its corresponding output.

1. **Leap Year Check**

```python
year = 2024
if (year % 4 == 0 and year % 100 != 0) or year % 400 == 0:
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")
```

```
2024 is a leap year.
```

2. **Wind Chill Index**

```python
T = -5  # Temperature in °C
v = 20  # Wind speed in km/h
WCI = 13.12 + 0.6215 * T - 11.37 * v**0.16 + 0.3965 * T * v**0.16
print(f"Wind Chill Index: {WCI:.2f}")
```

```
Wind Chill Index: -10.57
```

3. **Weekly Temperature Analysis**

```python
temperatures = {
    "Monday": {"high": 25, "low": 18},
    "Tuesday": {"high": 28, "low": 20},
    "Wednesday": {"high": 26, "low": 19},
    "Thursday": {"high": 27, "low": 21},
    "Friday": {"high": 30, "low": 22},
}

# Average high temperature
average_high = sum(day["high"] for day in temperatures.values()) /
    len(temperatures)

# Day with largest temperature difference
largest_difference_day = max(temperatures, key=lambda day:
    temperatures[day]["high"] - temperatures[day]["low"])

print(f"Average high temperature: {average_high:.2f} °C")
print(f"Day with largest difference: {largest_difference_day}")
```

```
Average high temperature: 27.20 °C
Day with largest difference: Friday
```

4. **Pressure Classification**

```python
def classify_pressure(pressure):
    if pressure < 1000:
        return "Low"
    elif 1000 <= pressure <= 1020:
        return "Normal"
    else:
        return "High"

pressures = [995, 1015, 1030, 1005]
results = [classify_pressure(p) for p in pressures]
print(results)
```

```
['Low', 'Normal', 'High', 'Normal']
```

5. **Top Three Wind Speeds**

```
wind_speeds = [12, 18, 5, 23, 35]
wind_speeds.sort(reverse=True)
top_three = wind_speeds[:3]
print(f"Top three wind speeds: {top_three}")
```

```
Top three wind speeds: [35, 23, 18]
```

6. **Dew Point Calculation**

```
def calculate_dew_point(T, RH):
    return T - (100 - RH) / 5

print(calculate_dew_point(25, 60))
print(calculate_dew_point(18, 80))
```

```
17.0
10.0
```

7. **Weather Station Analysis**

```
stations = {
    "Station A": {"temperature": 22, "humidity": 60, "wind_speed": 15},
    "Station B": {"temperature": 25, "humidity": 50, "wind_speed": 20},
    "Station C": {"temperature": 18, "humidity": 70, "wind_speed": 10},
}

average_temp = sum(station["temperature"] for station in stations.values()) /
    len(stations)
highest_wind = max(stations, key=lambda s: stations[s]["wind_speed"])

print(f"Average temperature: {average_temp:.2f} °C")
print(f"Highest wind speed recorded at: {highest_wind}")
```

```
Average temperature: 21.67 °C
Highest wind speed recorded at: Station B
```

8. **Temperatures Above 30 (*@°@*)C**

```
temperatures = [22, 31, 35, 28, 32]
above_30 = [temp for temp in temperatures if temp > 30]
print(above_30)
```

```
[31, 35, 32]
```

9. **Convert Temperatures to Fahrenheit**

```
def convert_weather_data(temps):
    return {C: (C * 9 / 5) + 32 for C in temps}

temperatures = [22, 25, 18]
print(convert_weather_data(temperatures))
```

```
{22: 71.6, 25: 77.0, 18: 64.4}
```

10. **Weather Log Analysis**

```
weather_log = [
    ("Monday", 25, 18),
    ("Tuesday", 28, 20),
    ("Wednesday", 26, 19),
    ("Thursday", 27, 21),
    ("Friday", 30, 22),
]

highest_day = max(weather_log, key=lambda x: x[1])[0]
lowest_day = min(weather_log, key=lambda x: x[2])[0]

print(f"Day with highest temperature: {highest_day}")
print(f"Day with lowest temperature: {lowest_day}")
```

```
Day with highest temperature: Friday
Day with lowest temperature: Monday
```

11. **Precipitation Analysis**

```
precipitation = [0, 5, 12, 0, 3, 8, 0, 7, 15, 0]

# Count dry days
dry_days = len([p for p in precipitation if p == 0])

# Total precipitation
total_precipitation = sum(precipitation)

print(f"Dry days: {dry_days}")
print(f"Total precipitation: {total_precipitation} mm")
```

```
Dry days: 4
Total precipitation: 50 mm
```

12. **Weather Alert System**

```
# Input: Temperature and humidity
T = 35  # Temperature in °C
RH = 85  # Relative humidity in %

# Heat Index calculation
HI = T + 0.33 * RH - 4

# Alert system
if HI > 40:
    print("Heat Index Alert! Take precautions.")
else:
    print("Conditions are normal.")
```

```
Heat Index Alert! Take precautions.
```

13. **Ascending Order Check**

```
temperatures = [18, 20, 22, 24]

# Check if the list is in ascending order
is_ascending = all(temperatures[i] <= temperatures[i + 1] for i in
    range(len(temperatures) - 1))

print(f"Is the list in ascending order? {is_ascending}")
```

```
Is the list in ascending order? True
```

14. **Monthly Average Temperatures**

```
# Daily temperatures for each month
monthly_data = {
    "January": [22, 24, 21],
    "February": [19, 20, 18],
}

# Calculate the average temperature for each month
```

```python
averages = {month: sum(days) / len(days) for month, days in monthly_data.items()}

# Print results
for month, avg in averages.items():
    print(f"Average temperature in {month}: {avg:.2f} °C")
```

```
Average temperature in January: 22.33 °C
Average temperature in February: 19.00 °C
```

15. **Unique Temperature Readings**

```python
temperatures = [22, 24, 22, 25, 24]

# Create a set of unique temperatures
unique_temperatures = set(temperatures)

print(f"Unique temperatures: {unique_temperatures}")
```

```
Unique temperatures: {24, 25, 22}
```

16. **Filter Cities by Temperature and Humidity**

```python
# City weather data
weather_data = {
    "City A": {"temperature": 26, "humidity": 45},
    "City B": {"temperature": 22, "humidity": 60},
    "City C": {"temperature": 28, "humidity": 40},
}

# Filter cities with temperature > 25 °C and humidity < 50%
filtered_cities = [city for city, data in weather_data.items() if
    data["temperature"] > 25 and data["humidity"] < 50]

print(f"Cities meeting the criteria: {filtered_cities}")
```

```
Cities meeting the criteria: ['City A', 'City C']
```

17. **Barometric Formula for Pressure at Altitude**

```python
import math

# Constants
P0 = 1013.25  # Sea level pressure in hPa
M = 0.02896   # Molar mass of air in kg/mol
g = 9.8       # Gravitational acceleration in m/s^2
R = 8.314     # Universal gas constant in J/(mol*K)
T = 293       # Temperature in K
h = 1000      # Altitude in meters

# Barometric formula
P = P0 * math.exp((-M * g * h) / (R * T))
print(f"Pressure at {h} meters: {P:.2f} hPa")
```

```
Pressure at 1000 meters: 898.76 hPa
```

18. **Heat Index for Multiple Cities**

```python
# Weather data for cities
cities = [
    {"city": "City A", "temperature": 30, "humidity": 70},
    {"city": "City B", "temperature": 32, "humidity": 80},
]

# Calculate heat index for each city
heat_indices = {
    city["city"]: city["temperature"] + 0.33 * city["humidity"] - 4 for city in
        cities
}

print("Heat indices:")
for city, hi in heat_indices.items():
    print(f"{city}: {hi:.2f} °C")
```

```
Heat indices:
City A: 49.10 °C
City B: 56.40 °C
```

19. Reverse a List of Temperatures

```python
# List of temperatures
temperatures = [22, 24, 25, 20]

# Reverse the list manually
reversed_temperatures = temperatures[::-1]

print(f"Reversed temperatures: {reversed_temperatures}")
```

```
Reversed temperatures: [20, 25, 24, 22]
```

20. City With the Highest Average Temperature

```python
# Dictionary of cities with average temperatures
city_data = {
    "City A": 25,
    "City B": 30,
    "City C": 28,
    "City D": 22,
    "City E": 27,
}

# Find the city with the highest temperature
highest_city = max(city_data, key=city_data.get)

print(f"The city with the highest average temperature is {highest_city}.")
```

```
The city with the highest average temperature is City B.
```