Python basics

Python's versatility and extensive library support make it a prime choice for academic and research purposes, especially in machine learning. For those new to this domain or Python, setting up a helpful environment is the first critical step. In the accompanied book *'Analysis of Farm Animal Behaviors : A Python Guide to Machine Learning and Deep Learning'*, we use Jupyter Notebook as our primary platform for demonstrations and experiments.

Installing and Setting Up the Python Environment

For the accompanied book, we work with Python version 3.11, and our primary platform for demonstrations and exercises is the Jupyter Notebook. The easiest way to get started with both Python and Jupyter is through the Anaconda environment.

Before we can install Anaconda, it is essential to have Python installed:

1. Go to Python's official download page https://www.python.org/.
2. Open https://www.python.org/downloads/ and download Python.
3. Run the downloaded installer for your operating system. Ensure you check the box that says "Add Python to PATH" during installation.

Installing Anaconda

Anaconda is a free, open-source Data Science Platform of Python and R, primarily for scientific computing. It offers a plethora of libraries and tools, ensuring that your machine learning environment is up-to-date and equipped for various tasks.

1. Go to the Anaconda official download page, https://www.anaconda.com/.
2. Download the installer suitable for your operating system (Windows, macOS, or Linux). You can find information and instructions about the installer at https://docs.anaconda.com/free/anaconda/install/index.html
3. Run the downloaded installer and follow the on-screen instructions. Ensure that you allow Anaconda to add a path to your environment (especially for Windows users).

Launching Jupyter Notebook via Anaconda Navigator

1. Start the Anaconda Navigator, which will be available in your applications or programs list after installation.
2. On the Navigator's home screen, click on "Launch" under Jupyter Notebook. Your default web browser will open, displaying Jupyter's dashboard.

Creating a New Notebook

Once you open the Jupyter dashboard:

1. Click on the "New" button (usually at the top-right).
2. From the drop-down, select "Python 3." This action will initiate a new notebook where you can start writing and executing Python code interactively.

3. Navigate through the interface to familiarize yourself with the commands

Familiarizing with Jupyter Notebook

Jupyter Notebook divides content into cells. These cells can either be code cells or markdown (text) cells.

1. To execute a code *cell*, click on it and press **Shift + Enter** (or ctrl + Enter).
2. To add new cells, use the "+" button on the toolbar or press B (to add below) or A (to add above) while you are in *command mode*.

Cells in Jupyter Notebook: In a Jupyter Notebook, you write and execute code in segments called "cells". Each cell can be run individually, and the output of the code will be displayed directly below the cell.

"+" button on the toolbar: When you open a Jupyter Notebook, you'll notice a toolbar at the top. One of the buttons on this toolbar is the "+" symbol, which is used to add a new cell. By clicking this "+" button, a new cell will be added immediately below the currently selected cell.

Command Mode in Jupyter Notebook:
A Jupyter Notebook has two primary modes:
a. **Edit Mode:** When you're typing inside a cell.
b. **Command Mode:** When you're navigating between cells without editing them. In this mode, the border of the selected cell is blue.
To enter Command Mode from Edit Mode, you can press the **Esc** key. Conversely, to switch from Command Mode to Edit Mode, you can press **Enter** on a selected cell.

To see the list of all available shortcuts in Jupyter notebook switch to Command Mode and press **H**.

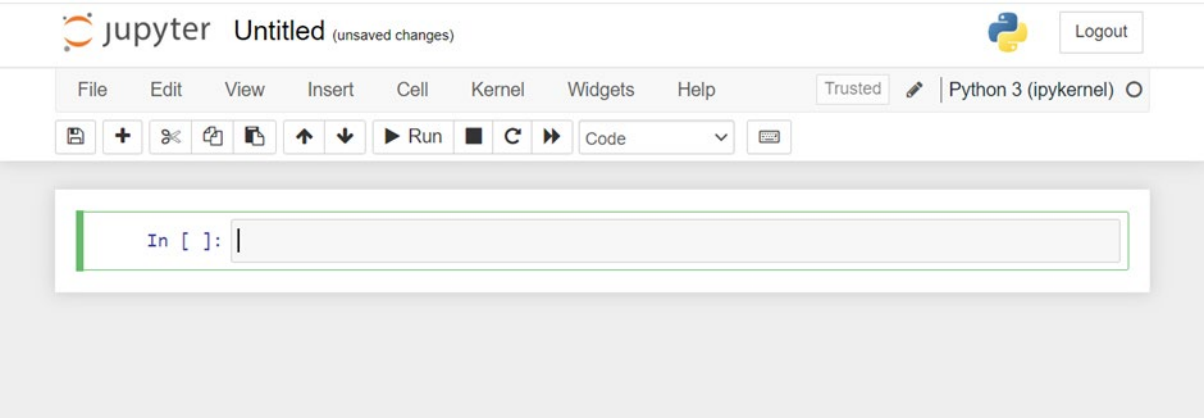

Figure 1 Jupyter notebook screenshot.

Installing Essential Libraries in Jupyter Notebook

Python libraries extend the capabilities of the Python language by providing additional functions, classes, and methods that simplify various tasks, from data manipulation to machine learning. To enhance the Python environment with these tools, one usually relies on package managers to download and install external libraries.

The two most popular package managers for Python are:

pip: The default package installer for Python. It allows you to install packages from the Python Package Index (PyPI) at https://pypi.org/.

conda: Especially useful for those using Anaconda, it can install packages from the Anaconda distribution and from PyPI. You can refer to its official website at https://docs.conda.io/en/latest/.

To install libraries in Jupyter Notebook, we will use a package manager for Python packages called **'pip'**. It is the standard package installer for Python, and it allows you to install packages that are not part of the Python standard library.

How to Install a Library:

Using pip

To install a library using pip in Jupyter notebook use the following command:

```
Jupyter notebook
!pip install library-name
```

For example, to install 'NumPy', you would write:

```
Jupyter notebook
!pip install numpy
```

The exclamation mark (!) at the start of each command is a Jupyter-specific syntax that allows you to run terminal commands directly from the notebook.

Using conda

```
Terminal
conda install library-name
```

Now proceed and install the following libraries through Jupyter notebook.

1. **NumPy (Numerical Python):** As its name suggests, NumPy provides support for large multi-dimensional arrays and matrices, along with mathematical functions to operate on these data structures. For a comprehensive understanding of NumPy, you can refer to its official documentation available at https://numpy.org/.
2. **Pandas:** This library offers data structures and functions necessary to efficiently manipulate large datasets. It's known for its data structure called DataFrame, which is like an in-memory representation of an Excel sheet. Check its documentation at https://pandas.pydata.org/docs/index.html.
3. **Matplotlib and Seaborn:** Both are popular data visualization tools in Python. Matplotlib (https://matplotlib.org/) is a comprehensive library for creating static, animated, and interactive visualizations, while Seaborn (https://seaborn.pydata.org/) is based on Matplotlib and provides a higher-level, more aesthetically pleasing interface for making statistical graphics.
4. **Scikit-learn:** It's a machine learning library that provides simple and efficient tools for predictive data analysis.

Command to install the libraries:

Jupyter notebook

```
!pip install numpy
!pip install pandas
!pip install matplotlib seaborn
!pip install scikit-learn
```

Importing and Testing the Libraries:

Once installed, you will want to **import** these libraries to ensure they have been installed correctly and are available for use in your notebook.

In Python, the **import** statement is used to load a library into a script. Once a library is imported, you can call its functions and methods.

Here's how to import and test the libraries:

```
In [1]: import numpy as np
        print(np.__version__)

        1.24.3

In [2]: import pandas as pd
        print(pd.__version__)

        1.5.3

In [3]: import matplotlib
        print(matplotlib.__version__)

        3.7.1

In [4]: import seaborn
        print(seaborn.__version__)

        0.12.2

In [5]: import sklearn
        print(sklearn.__version__)

        1.2.2
```

By using **as** during the import, we assign a shorter alias to the library, making it easier and more convenient to use in our code. The **.__version__** attribute is a way to retrieve the installed version of the library.

When you import a library in Python, you are making its functionalities available in your current workspace. By running the import commands and then utilizing the library's functions or attributes, you're effectively validating that the library has been installed and is functioning correctly.

The 'print' Function

In Python, the print function is used to output information to the console. Whether you are displaying the result of an operation, a string, or any other data type, print serves as one of the most basic and essential tools for debugging and data display.

When you encapsulate a variable or expression inside the print function, like **`print(variable)`** or **`print(expression)`**, Python evaluates it and displays the result. This is especially useful for checking values, confirming code operation, or just outputting data in a readable format.

For example, when we use:

```
In [1]: import numpy as np
        print(np.__version__)

        1.24.3
```

We are asking Python to output the value of the **__version__** attribute from the **numpy** library. This value typically displays the version of the library you have installed, which can be useful to confirm if you are using a specific version for compatibility or functionality reasons.

Python Basics

Python syntax:

In any programming language, syntax refers to the set of rules that state how programs in that language must be written to be considered valid and executable. Think of it as the grammar of a spoken language: just as misplaced punctuation can change the meaning of a sentence; incorrect syntax can lead to program errors.

Key Features of Python Syntax:

Indentation: Python relies on indentation. This means that the amount of white space at the start of a line matters. Code blocks under functions, loops, and conditionals are defined by their indentation.

Jupyter notebook

```
if x > 10:
    print("Some text")
```

In the above code, the **print** statement is executed only if the condition **x > 10** is true because it is indented under the if statement.

Comments: Comments are lines that are not executed by the Python interpreter. They are primarily used to explain code or prevent execution of code without deleting it. In Python, a comment starts with a **#** character.

Jupyter notebook

```
# This is a comment and will not be executed
print("Some text")
```

In Python, besides using the **#** character for single-line comments, there is another way to add comments, especially for multi-line descriptions or annotations: **the triple-quoted strings**.

Triple-quoted strings can be created using a set of triple single quotes **'''** or triple double quotes **"""**. Though primarily they represent multi-line strings, when they are not assigned to a variable or used as a string within the code, they effectively act as multi-line comments.

1. **Documenting Functions:**
   Triple-quoted strings are often used at the beginning of functions to provide a description, known as a **docstring**. This description can later be accessed using Python's built-in **help()** function.

Jupyter notebook

```
def add_numbers(x, y):
    """
    This function returns the sum of two numbers.
    Args:
        x: First number
        y: Second number
    Returns:
        Sum of x and y
    """
    return x + y
```

2. **Commenting Code Blocks:** If you want to temporarily disable a block of code without deleting it, you can wrap it in triple quotes.

Jupyter notebook

```
"""
This is a multiline comment
And will not be executed

"""
```

While triple-quoted strings can act as multi-line comments, their primary purpose is for multi-line strings and documentation (like docstrings). If you're just leaving a note or commenting out code, the **#** character might be more recognizable to others reading your code as a comment.

3. **Statements:** In Python, instructions written in the source code for execution are termed as statements. Python does not use semicolons **;** to indicate the end of the line or statement; the end of the line is the end of the statement.

Jupyter notebook

```
# This is a comment and will not be executed
x = 10
y = 20
sum = x + y
```

Each line above is a distinct statement.

4. **Line Continuation:** Python allows the splitting of a single statement across multiple lines, known as line continuation. This can be achieved using backslashes \, or by enclosing the statement within parentheses **()**, or using brackets **[]** and braces **{}** for lists and dictionaries respectively.

Jupyter notebook

```
total = 10 + 20 + 30 \
        + 100

days = ['Cows',
        'Horses',
        'Pigs']
```

5. **Quotation Marks:** Strings in Python can be created using single, double, or even triple (for multi-line strings) quotes. All are correct and can be used based on the programmer's preference, as long as the same type of quote starts and ends the string.

Jupyter notebook

```
single_quoted_string = 'Hello, world!'

double_quoted_string = "Hello, world!"

triple_quoted_string = """This is a multi-line string spanning several lines."""
```

6. **Colon:** In Python, colons are used to introduce a new block of code (for functions, loops, and conditionals).

Jupyter notebook

```
if x > 10:
    print("Some text")
```

Understanding Python's syntax is the first step to writing clean, error-free code. As you continue your journey with Python, you will find its syntax to be both intuitive and clear.

7. **Variable Names:** Variable names can start with a letter (a-z, A-Z) or an underscore (_). Numbers cannot be used at the start of a variable name. Apart from the initial character, variable names can have letters, numbers, and underscores. They are also case sensitive. For example, Python treats '**VariableName**', '**variablename**', and '**VARIABLENAME**' as different variables. Python has a set of reserved words (**keywords**) that cannot be used as variable names. Examples include '**if**', '**else**', '**while**', '**and**', '**def**', and so on.

Examples of valid and invalid variable names:

Jupyter notebook

```
# Start with a letter or an underscore

valid_variable = "This is valid"
_also_valid = "This is also valid"
10invalid = "This variable will raise an error" # INCORRECT

# Contain only Alpha-Numeric Characters and Underscores

var9999 = "Valid variable"
var_name_ = "Valid variable"
var@name = "This variable will raise an error" # INCORRECT

# Case-Sensitive

Var1 = 5
var1 = 10
print(Var1, var1)

# Cannot be Python Keywords

if = "ok" # SyntaxError: invalid syntax
```

Conventions and Best Practices

1. **Descriptive Names:** While you can name your variable anything, it is good practice to use descriptive names that indicate the purpose of your variable.

Jupyter notebook

```
# Good descriptive name
animal_id = 100
# Not descriptive
ai = 100
```

2. **Use Lowercase for Variables:** The most common convention is to use all lowercase letters for variable names and separate words using underscores.

Jupyter notebook

```
sample_rate = 12
```

3. **Use Uppercase for Constants:** If a variable represents a value that should not change, it is commonly written in uppercase.

Jupyter notebook

```
PI = 3.14159
```

4. **Avoid using Single Character Variables:** Except in specific cases (like loops counters), avoid using single characters as variable names

Jupyter notebook

```
# Good for loop counters
for i in range(10):
    print(i)

# But in general, be more descriptive
age = 11
```

While we have talked about some foundational elements of Python syntax and variable naming, it is essential to understand that these basics only scratch the surface. Python is a versatile and powerful language with many workings and features. Our intention here is not to provide a complete Python tutorial but rather to equip the reader with enough understanding to navigate the machine learning chapters of our accompanied book comfortably.

For readers who are keen to look into a more extensive study of Python, numerous dedicated resources and tutorials are available. One highly recommended starting point would be the official Python documentation and tutorial page, which provides an in-depth and well-structured guide for both beginners and seasoned programmers https://docs.python.org/3/tutorial/index.html .

Python keywords

In every programming language, certain words are reserved for specific functionalities. These words, often referred to as "**keywords**," have a predefined meaning in the language's syntax and structure. They play a vital role in defining the language's rules and cannot be used as identifiers like variable or function names. In Python, keywords are the building blocks of the language.

Here are some important things to remember about Python keywords:

Keywords are **case-sensitive**. For instance, **True** and **true** are not the same in Python; the former is a keyword, while the latter is not.

You cannot use keywords as variable names, function names, or any other identifier.

New keywords may be added, and some may be deprecated in future versions of Python, so it is a good practice to keep updated with the language's documentation.

Some commonly used Python keywords include:

- **and**: A logical operator.
- **break**: Used to break out of a loop prematurely.
- **def**: Used to define a function.
- **if, elif, else**: Conditional statements.
- **import**: Used to include external modules or libraries.
- **return**: Used to return a value from a function.
- **class**: Used to define a class in object-oriented programming.
- **True, False**: Boolean values representing truth and false values, respectively.
- **try, except**: For exception handling.
- **while, for**: Looping constructs.

This list is by no means exhaustive. To view a complete list of keywords in your Python version, you can do the following:

Jupyter notebook

```
import keyword
print(keyword.kwlist)
```

```
output: ['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await',
'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally',
'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not',
'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

Data Types

A data type, in the context of programming and computer science, refers to a classification that specifies which type of value a variable can hold. It denotes the nature of the data that can be

stored and how the computer should interpret and process that data. Different programming languages might have varying data types, but they typically comprise fundamental classifications such as **integers**, **floating-point numbers**, **characters**, **strings**, and **booleans**.

In Python, for instance, the data type of a variable is determined dynamically, without the need for explicit declaration by the programmer. This means that when you assign a value to a variable, Python will automatically understand the data type. The primary data types in Python include:

Integers (int): Whole numbers, e.g., 5, -2.

Floating-Point Numbers (float): Decimal numbers or numbers with a fractional component, e.g., 3.142, 2.714, 0.003.

Strings (str): A sequence of characters, e.g., "temperature", 'cows'.

Booleans (bool): Logical values indicating True or False.

Complex Numbers (complex): Numbers with a real and imaginary component, e.g., 1 + 2j.

Below are the Python code snippets you can run in Jupyter Notebook to explore the different data types and how to check a variable's data type:

Jupyter notebook

```
# Data types

# Assign an integer value to a variable
num_integer = 10

# Print the value
print(num_integer)

# Check and print the data type using the type() function
print(type(num_integer))

# Assign a floating-point value to a variable
num_float = 3.1426353

# Print the value
print(num_float)

# Check and print the data type
print(type(num_float))

# Assign a string value to a variable
text = "Hello, Jupyter!"

# Print the value
print(text)
```

```
# Check and print the data type
print(type(text))

# Assign a boolean value to a variable
is_python_amazing = True

# Print the value
print(is_python_amazing)

# Check and print the data type
print(type(is_python_amazing))

# Assign a complex number to a variable
num_complex = 1 + 2j

# Print the value
print(num_complex)

# Check and print the data type
print(type(num_complex))
```

When you run each of these code snippets in separate Jupyter Notebook cells, you'll get the value of the variable and its corresponding data type printed as the output. The **type()** function in Python is used to determine the class type of a variable or value.

In Python, you can change the data type of a value or variable, a process known as **type casting** or **type conversion**. There are built-in functions in Python to perform these conversions:

int(): Converts a value to an integer. This will truncate any decimal portion of a floating-point number.

float(): Converts a value to a floating-point number.

str(): Converts a value to a string.

complex(): Converts numbers to a complex number.

bool(): Converts a value to a boolean. Non-zero and non-empty values are converted to **True**, while **zero**, **None**, and empty values (like an empty string or empty list) are converted to **False**.

Jupyter notebook

```
# Convert a floating-point number to an integer
num_float = 5.89
num_int = int(num_float)
print(num_int)  # Output: 5
print(type(num_int))  # Output: <class 'int'>
```

```
# Convert an integer to a string
str_num = str(num_int)
print(str_num)  # Output: "5"
print(type(str_num))  # Output: <class 'str'>

# Convert an integer to a floating-point number
float_num = float(num_int)
print(float_num)  # Output: 5.0
print(type(float_num))  # Output: <class 'float'>

# Convert an integer to a boolean
bool_value = bool(num_int)
print(bool_value)  # Output: True (since num_int is non-zero)
print(type(bool_value))  # Output: <class 'bool'>
```

It is essential to be cautious while type casting. Not every conversion is valid. For example, trying to convert a string that doesn't represent a number (e.g., "age") to an integer will result in a **ValueError**.

Also, when you convert between types, you might lose information (as in the truncation of the decimal portion when converting a float to an integer), so always be aware of the implications of the conversions you are making.

Arithmetic Operations

Python can be used as a calculator:

```
# Arithmetic Operations
x, y = 2, 10

print(x + y)    # Output = 12
print(x - y)    # Output = -8
print(x * y)    # Output = 20
print(x / y)    # Output = 0.2
print(x // y)   # Output = 0
print(x % y)    # Output = 2
print(y % x)    # Output = 0
print(-x)       # Output = -2
print(abs(-x))  # Output = 2
print(int(2.10))# Output = 2
print(float(2)) # Output = 2.0
print(x ** y)   # Output = 1024
```

So far we have talked about foundational aspects like Python's syntax, variable naming conventions and basic data types. We also showed that Python can be used as a calculator. Remember, while we have navigated some vital concepts, Python offers so much more. Our aim here is not a complete mastery of Python, but to equip you with crucial basics that will facilitate the machine learning applications we aim to explore. In the next sections we will look at:

- **Data Structures:** We will be introduced to Python's primary **containers** such as **lists**, **dictionaries**, **sets**, and more. These structures are fundamental for organizing and processing data.
- **Control Structures:** To create meaningful programs, one must have the ability to conditionally execute sections of code and iterate over data. We will look into constructs like loops and conditionals that provide this capability.
- **Functions, Libraries, and Classes:** We will learn about creating reusable blocks of code through functions, using functionalities via libraries, and understanding the object-oriented nature of Python through classes.

Data Structures

In programming, the organization and management of data is crucial. The way we structure data impacts how effectively we can access, modify, or delete specific elements. This brings us to the importance of data structures.

Think of data structures as **containers** for data. However, not every container is suitable for every purpose. The choice of data structure often depends on the specific requirements of a program.

Python provides a variety of built-in data structures, each designed for specific tasks and with its own set of advantages. For example, lists in Python are used to store sequences of items, while dictionaries allow storage of key-value pairs.

In this section, we will explore the fundamental data structures provided by Python, understand their properties, and learn about the operations they support. Selecting the appropriate data structure is essential for creating efficient programs. We will start our discussion with a commonly used data structure in Python: the **list**.

Lists

A **list** is one of Python's build-in data structure that can store a collection of items. Lists are ordered and mutable (meaning their contents can be modified) and allow duplicate entries. Items in a list can be of any data type, including other lists which makes them very useful.

1. **Creating Lists**

Jupyter notebook

```
# Creating a list
farm_animals = ['pigs', 'cows', 'sheep', 'horses']

# List with mixed data types
mixed_list = [1, 'temp', 0.5, ['a', 'b']]
```

2. **Accessing Elements**

Lists are indexed with integers, starting from zero for the first element. For example, if there are **n** elements in the list, then the start index will be **0** and the last index will be **n-1**. Below you can visualize how indexes are treated.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	**N**	**I**	**M**	**A**	**L**	**B**	**E**	**H**	**A**	**V**	**I**	**O**	**R**
-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Jupyter notebook

```
# Creating a list
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

# Select all elements using ':'
my_list[:]
# Output: ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

# Select first element
my_list[0]
# Output: ['A']

# Select 6th element
my_list[5]
# Output: ['L']

# Select last element
my_list[-1]
# Output: ['R']
```

3. Modifying Lists

Being mutable, you can change, add, or remove elements from lists.

- **Changing an element**

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

my_list[0] = 'O'
print(my_list)

# Output: ['O','N','I','M','A','L','B','E','H','A','V','I','O','R']
```

- **Adding elements:**
  - Using **append()** it adds an element to the end of the list

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

my_list.append('!')
print(my_list)

# Output: ['A','N','I','M','A','L','B','E','H','A','V','I','O','R','!']
```

  - Using **insert()** inserts an element at a specified position.

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

# len()is used to determine the number of elements in an object
# Here we want to add the character '!' at the end of the list
# We len() is used to get the number of elements and be used as the last
index
my_list.insert(len(my_list),'!')
print(my_list)

# Output: ['A','N','I','M','A','L','B','E','H','A','V','I','O','R','!']
```

  - **Using extend()** you can append several elements or another list to the list.

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

my_list.extend(['!','!'])
print(my_list)

# Output:
['A','N','I','M','A','L','B','E','H','A','V','I','O','R','!', '!']
```

  - Using **list concatenation()**

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

new_list = my_list + ['!','!'])
print(new_list)

# Output:
['A','N','I','M','A','L','B','E','H','A','V','I','O','R','!', '!']
```

- **Removing elements**
  - Using **remove():** removes the **first** occurrence of the element with the specified value.

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

my_list.remove('A')
print(my_list)

# Output:
['N','I','M','A','L','B','E','H','A','V','I','O','R']
```

  - Using **pop()** it removes the element at the indicated index number. By default, the index number is **-1**, so if you use an empty **pop()**, the last item from the list will be removed.

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

my_list.pop(3)
print(my_list)

# Output:
['A','N','I','A','L','B','E','H','A','V','I','O','R']
```

  - Using **clear()** all elements from the list are removed

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

my_list.clear()
print(my_list)

# Output: []
```

- **Reverse elements** using **reverse()**

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

my_list.reverse()
print(my_list)

# Output: ['R','O','I','V','A','H','E','B','L','A','M','I','N','A']
```

- **Sort elements** using **sort()**

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

my_list.sort()
print(my_list)

# Output: ['A','A','A','B','E','H','I','I','L','M','N','O','R','V']
```

4. **List slicing**

List slicing is used to obtain a subset of the list.

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

# Select the 2nd to 10th element
# This will extract the elements from index 1 (inclusive) to index 10 (exclusive)
my_list[1:10]
# Output: ['N','I','M','A','L','B','E','H','A']

# Select the first 4 elements
my_list[:4]
```

```
# Output: ['A','N','I','M']

# Select the elements from index 5 to the end
my_list[5:]
# Output: ['L', 'B', 'E', 'H', 'A', 'V', 'I', 'O', 'R']

# Reversing the list
my_list[::-1]
# Output: ['R','O','I','V','A','H','E','B','L','A','M','I','N','A']
```

5. **List methods:** Some commonly used list methods include:

- **count():** Returns the number of occurrences of an element.

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

# Find the number of occurrences of A in the list
my_list.count('A')

# Output: 3
```

- **copy():** Returns a shallow copy of the list.
- **index():** Returns the index of the first element with the specified value.
    - You can use **index()** to get the position of an element in the list.
    - The **index()** method can also take optional parameters **start** and **end** to search within a subsequence of the list.
    - If the specified value is not present in the list (or within the specified range), the **index()** method will raise a **ValueError**.

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

# Find index A in the list
my_list.index('A')

# Output: 0

# Find index A in the list starting from position 5
my_list.index('A', 5)
```

```
# Output: 9

# Find index A in the list starting from position 1 to 6
my_list.index('A', 1, 6)

# Output: 4

# This will raise an error.
my_list.index('Z')

# Output:
ValueError
my_list.index('Z')
ValueError: 'Z' is not in list
```

6. **Iterating over a list:** You can iterate over the items of a list using a loop.

Jupyter notebook

```
my_list = ['A','N','I','M','A','L','B','E','H','A','V','I','O','R']

for letter in my_list:
    print(letter)

# Output:
A
N
I
M
A
L
B
E
H
A
V
I
O
R
```

7. **List Comprehension:** A concise way to create lists using a single line of code.

Jupyter notebook

```
some_numbers = [1,2,3,4,5]

squared_numbers = [x**2 for x in numbers]
print(squared_numbers)

# Output: [1, 4, 9, 16, 25, 36]
```

8. **Nested Lists:** Nested lists are quite common in Python, and they arise when a list contains another list as one of its elements. Essentially, it is a list inside a list. Nested lists are incredibly useful for representing more complex data structures like matrices or multi-level categories.
    - **Creating a nested list:** You simply include one or more lists as elements within another list.
    - **Accessing elements:** Accessing elements in a nested list requires specifying indices for each level of nesting.
    - **Modifying nested lists:** You can modify a nested list like you would a regular list but keep the levels of nesting in mind.
    - **Iterating over nested lists:** Looping over nested lists typically involves nested loops.
    - **List comprehensions with nested lists:** Nested list comprehensions can be used to modify nested lists more concisely.
    - **Flattening a nested list:** Sometimes, you might want to turn a nested list into a single, flat list. This can be achieved with a combination of loops or list comprehensions.

Examples:

Jupyter notebook

```
# Creating a nested list (3x3 matrix with three lists inside the main list)
nested_list = [[1,2,3],[4,5,6],[7,8,9]]

# Access the first list
print(nested_list[0])  # Outputs: [1, 2, 3]

# Access the second element of the first list
print(nested_list[0][1])  # Outputs: 2

# Change the value at position 2 of the first list
nested_list[0][1] = 10
print(nested_list[0])  # Outputs: [1, 10, 3]

# Iterating over bested lists
for sublist in nested_list:
    print(sublist)
```

```
# Outputs:
[1, 10, 3]
[4, 5, 6]
[7, 8, 9]

for sublist in nested_list:
    for item in sublist:
        print(item)
# Outputs:
1
10
3
4
5
6
7
8
9

# Square each number in the nested list
squared_list = [[item**2 for item in sublist] for sublist in nested_list]
print(squared_list)
# Outputs: [[1, 100, 9], [16, 25, 36], [49, 64, 81]]

# Flattening a nested list
flat_list = [item for sublist in nested_list for item in sublist]
print(flat_list)
# Outputs: [1, 10, 3, 4, 5, 6, 7, 8, 9]
```

Nested lists provide a way to store and manage multi-dimensional data. While they are a basic form of representing such data, Python offers more advanced tools, like **NumPy arrays**, for more efficient multi-dimensional operations. For the context of basic Python, however, understanding nested lists and their operations is essential.

9. **Basic Operations**
    - **Concatenation**
    - **Repetition**
    - **Membership**

Jupyter notebook

```
# Concatenation
[1, 2, 3] + [4, 5, 6]  # Output: [1, 2, 3, 4, 5, 6]

# Repetition
```

```
[A]*4 # Output: ['A', 'A', 'A', 'A']


# Membership
2 in [1,2,3]  # Output: True
```

10. Length: The len() function returns the number of items in a list.

Jupyter notebook

```
some_numbers = [1,2,3,4,5]

len(some_numbers)

# Output: 5
```

Lists are one of the most versatile and widely used data structures in Python. Their dynamic and flexible nature makes them suitable for a wide range of applications, from simple tasks to complex algorithms. By understanding lists, you are better equipped to manage collections of data in Python.

Dictionaries

A dictionary is an unordered collection of data values used to store data values in a pair form. Each pair consists of a unique key and an associated value. Since keys are unique, dictionaries can be highly effective in data retrieval, making it a valuable tool for tasks like categorizing animal behaviors based on certain criteria or features.

Syntax:

```
my_dict = {key1: value1, key2: value2, ...}
```

1. **Creating a Dictionary**

To define a dictionary, you use curly braces **{}** and specify **keys** and **values**:

Jupyter notebook

```
animal_behavior = {'chicken': 'omnivorous',
                   'cow': 'herbivorous',
                   'pig': 'omnivorous'
```

```
                    }
```

2. **Accessing Dictionary Values**

Values in a dictionary can be accessed by referring to their corresponding keys

Jupyter notebook

```
print(animal_behavior['chicken']

# Output: 'omnivorous'
```

3. **Modifying a Dictionary**

You can add new key-value pairs or modify existing ones with ease:

Jupyter notebook

```
# Adding a new entry
animal_behavior['sheep'] = 'herbivorous'

# Modifying an existing entry
animal_behavior['horse'] = 'herbivorous'
```

4. **Methods Associated with Dictionaries**

Some essential methods for dictionaries include:

- **keys(): Returns a list of all the keys.**
- **values(): Returns a list of all the values.**
- **items(): Returns a list of all key-value pairs.**
- **get(key): Retrieves the value for a given key.**

5. **Iterating Through a Dictionary**

To loop through a dictionary, you can iterate over its keys, values, or both:

Jupyter notebook

```
# Iterating over keys
for animal in animal_behavior.keys():
    print(animal)
```

```
# Output:
chicken
cow
pig


# Iterating over values
for behavior in animal_behavior.values():
    print(behavior)

# Output:
omnivorous
herbivorous
omnivorous


# Iterating over both keys and values
for animal, behavior in animal_behavior.items():
    print(f"The {animal} is {behavior}.")

# Output:
The chicken is omnivorous.
The cow is herbivorous.
The pig is omnivorous.
```

6. **Checking Key Existence**

Before accessing a key, it is practical to check if it exists in the dictionary to avoid errors:

Jupyter notebook

```
'lion' in animal_behavior

# Output: False
```

Tuples

While we have discussed lists and dictionaries, another integral data structure in Python is the **tuple**. A tuple is similar to a list. Both can store multiple items in a single unit (or container). However, there are some distinctions between lists and tuples:

1. **Immutability**: The most defining characteristic of tuples is their **immutability**. Once you define a tuple, you **cannot alter its contents**, i.e., you cannot add, modify, or delete elements from a tuple. This feature is particularly useful when you want to ensure that certain data remains constant throughout your program's execution.
2. **Syntax**: While lists use square brackets **[]**, tuples use parentheses **()**. For example, a list would be defined as **list_variable = [item1, item2, ...],** whereas a tuple would be **tuple_variable = (item1, item2, ...).**
3. **Usage Scenarios:** Tuples are generally used in scenarios where data should not change, for instance, fixed configurations or settings. It is also a common practice to use tuples for defining collections of constants.

Jupyter notebook

```
# Defining a tuple to store weight and height of a particular breed of horse
# Here, 500kg is the weight and 160cm is the height
horse_measurements = (500, 160)

# Accessing elements from the tuple
# This would assign 500 to the weight variable
weight = horse_measurements[0]

# This would assign 160 to the height variable
height = horse_measurements[1]
```

While you cannot modify the contents of a tuple directly, you can concatenate or combine tuples to create new ones:

Jupyter notebook

```
# A tuple storing sounds made by specific farm animals
animal_sounds = ("moo", "baa")

# Adding two more sounds to the tuple
animal_sounds += ("cluck", "squeak")

print(animal_sounds)

# Output: ('moo', 'baa', 'cluck', 'squeak')
```

Sets

A **set** is a collection data type that is both **unordered** and **unindexed**. The primary feature of a set is that it does **not allow duplicate values**, making it an ideal choice when you need to handle and represent unique elements.

Characteristics of Sets:

- **Uniqueness**: Sets do not allow duplicate values. If you try to add a duplicate value to a set, Python simply ignores it without raising an error.
- **Unordered**: The elements in a set are unordered. Therefore, you cannot access or change an element of a set using an index or a key.
- **Mutability**: While sets themselves are mutable (you can add or remove items after the set is created), the items contained within must be of a data type that is immutable, like numbers, strings, and tuples.

You can create a set by placing a comma-separated sequence of items inside curly braces **{}**. Remember not to confuse this with dictionaries. Dictionaries have key-value pairs, while sets only have individual elements.

You can also create a set using the built-in **set()** function. This is particularly useful if you're converting other data types (like lists or tuples) to sets.

Duplicates are automatically removed when creating a set, as sets can only contain **unique** elements.
Sets are unordered, so the elements might not appear in the same order in which they were added.

Usage and Examples:

Let's consider a scenario in farm animal behavior analysis. Imagine you have recorded the names of all the animals on a farm over multiple days, and you want to find out the unique animals.

Jupyter notebook

```
# List of animals recorded over multiple days
recorded_animals = ['sheep', 'horse', 'goat', 'horse', 'sheep', 'goat',
'turkey']

# Converting the list to a set to get unique animals
unique_animals = set(recorded_animals)

print(unique_animals)
# Output: {'turkey', 'sheep', 'goat', 'horse'}
```

Basic Operations with Sets:

- **Adding Elements**: You can add an element to a set using the **add()** method.
- **Removing Elements:** The **remove()** or **discard()** methods can be used to remove elements. The difference is that remove() raises an error if the element is not found, while discard() does not.
- **Union and Intersection**: Sets support mathematical operations like **union** (**|** or **union()**) and **intersection** (**&** or **intersection()**).

In the context of farm animal behavior, sets can be instrumental in tasks like identifying unique behaviors, filtering out repeated data entries, or comparing data sets for common elements.

Basic Operations examples:

1. **Creating a set:**

Jupyter notebook

```
animals = {"sheep", "horse", "goat"}
print(animals)  # Output: {'goat', 'horse', 'sheep'}

# Output: {'sheep', 'goat', 'horse'}
```

2. **Adding elements:**

Jupyter notebook

```
animals.add("turkey")
print(animals)

# Output: {'sheep', 'horse', 'goat', 'turkey'}
```

3. **Removing elements:**
    - Using **remove()**: Raises an error if the element is not found

Jupyter notebook

```
animals.remove("goat")
print(animals)

# Output: {'sheep', 'horse', 'turkey'}
```

    - Using **discard()**: Does not raise an error if there is not such element in the set

Jupyter notebook

```
animals.discard("goat")  # No error even though "goat" is already removed
print(animals)

# Output: {'turkey', 'sheep', 'horse'}
```

4. **Union:** Combines two sets without duplicates

Jupyter notebook

```
# List of animals recorded over multiple days
farm1 = {"sheep", "horse", "goat"}
farm2 = {"turkey", "horse", "duck"}

union_farm = farm1 | farm2
print(union_farm)
# Output: {'duck', 'goat', 'horse', 'sheep', 'turkey'}

# or
union_farm = farm1.union(farm2)
print(union_farm)
# Output: {'duck', 'goat', 'horse', 'sheep', 'turkey'}
```

5. **Intersection:** Finds common elements between two sets

Jupyter notebook

```
common_animals = farm1 & farm2
print(common_animals)
# Output: {'horse'}

# or

common_animals = farm1.intersection(farm2)
print(common_animals)
# Output: {'horse'}
```

6. **Difference:** Gets elements that are only in the first set and not in second

Jupyter notebook

```
unique_to_farm1 = farm1 - farm2
print(unique_to_farm1)
# Output: {'goat', 'sheep'}

# or
unique_to_farm1 = farm1.difference(farm2)
print(unique_to_farm1)
# Output: {'goat', 'sheep'}
```

7. **Subset and Superset:** Checks if a set is a subset or superset of another set

Jupyter notebook

```
set1 = {"sheep", "horse"}
print(set1.issubset(farm1))
# Output: True

print(farm1.issuperset(set1))
# Output: True
```

8. **Clearing all elements**

Jupyter notebook

```
animals.clear()
print(animals)

# Output: set()
```

These are some of the fundamental operations you can perform with sets in Python. Sets, due to their unique properties, offer efficient ways to handle and manipulate collections, especially when dealing with the uniqueness of data.

Control Structures

Control structures in Python allow you to execute certain code blocks conditionally and/or repeatedly. They are foundational concepts in programming, facilitating decision-making, looping, and error handling in code.

1. Conditional Statements

if Statement: Executes code if the specified condition is True.

Jupyter notebook

```
temperature = 15
if temperature < 20:
    print("It's cold outside.")

# Output: It's cold outside
```

if-else Statement: Provides an alternative code to execute if the initial condition is False.

Jupyter notebook

```
temperature = 15
if temperature < 20:
    print("It's cold outside.")
else:
    print("It's warm outside.")

# Output: It's cold outside
```

if-elif-else Statement: Allows multiple conditions to be checked in a sequence.

Jupyter notebook

```
if temperature < 10:
    print("It's freezing.")
elif temperature < 20:
    print("It's cold outside.")
else:
    print("It's warm outside.")

# Output: It's cold outside
```

Nested conditions:

Jupyter notebook

```
x = 5
y = 3
z = 4

if x > y:
    if x > z:
        print("x is the largest")
    else:
        print("z is larger than x but x is larger than y")

# Output: x is the largest
```

2. Loops

for Loop: Iterates over items of any sequence (like a list or a string).

Jupyter notebook

```
animals = ['sheep', 'horse', 'goat']
for animal in animals:
    print(animal)

# Output:
sheep
horse
goat
```

while Loop: Executes code as long as the condition remains True.

Jupyter notebook

```
count = 0
while count < 3:
    print(count)
    count += 1

# Output:
0
1
2
```

List Comprehension: Generates a new list by applying an expression to each item in a sequence.

Jupyter notebook

```
squared_numbers = [x*x for x in range(5)]
print(squared_numbers)  # [0, 1, 4, 9, 16]

# Output:
[0, 1, 4, 9, 16]
```

3. Exceptions and Exception Handling in Python

In programming, even with the most accurately written code, there are moments when errors occur during the execution of the program. These unexpected occurrences are called **exceptions**. Handling these exceptions without allowing the program to crash is a significant aspect of writing robust and reliable software.

What is an Exception?

In Python, exceptions are unwanted or unexpected events that can occur during the execution of a program. These events disrupt the normal flow of the program and usually result in terminating the execution.

Examples of common exceptions:

- **AttributeError:** This type of exception occurs when attempting to access a method or attribute of an object that does not exist. For instance, trying to access an undefined attribute of a class instance can trigger this.
- **ZeroDivisionError**: This arises when there is an effort to divide a number by zero.
- **KeyError**: Encountered when a specific key is not present in a dictionary.
- **IO Error**: Occurs due to an error in input/output operations, especially when there is a challenge in reading or writing a file.
- **Name Error**: This occurs when the interpreter cannot find a given function or variable name within the current context or scope.
- **ImportError**: This is triggered when a module fails to load or is not found during an import operation.
- **ValueError**: Seen when a method or function receives an argument or input that is unsuitable. For example, attempting to convert a non-integer string into an integer.
- **SyntaxError**: Encountered when there is a mistake in the code's structure. Examples include wrongly spelled keywords or a missing colon.
- **TypeError**: This arises when there is a mismatch between a function's expected data type and the provided data type, like adding a number to a text string.
- **IndexError**: This is flagged when trying to access an index outside the allowable range for sequence data types such as lists or tuples.

Exception handling

In Python, exception handling is a mechanism that allows programmers to handle run-time anomalies (exceptions) without terminating the program. This ensures that even in the event of unexpected issues, the program can continue to function or exit, providing informative feedback to users or developers.

An example of exception handling

Jupyter notebook

```
def compute_division(x, y):
    try:
        result = x / y
    except ZeroDivisionError as e:
        print(f"Error: {e}. Division by zero is not allowed!")
        return None
    except TypeError as e:
        print(f"Error: {e}. Ensure both x and y are numbers!")
        return None
    else:
        print(f"Division successful! Result is: {result}")
        return result
    finally:
        print("compute_division execution completed.")

# Test case 1
print(compute_division(10, 2))

# Output:
Division successful! Result is: 5.0
compute_division execution completed.
5.0

# Test case 2
print(compute_division(10, 0))

# Output:
Error: division by zero. Division by zero is not allowed!
compute_division execution completed.
None

# Test case 3
print(compute_division("10", 2))

# Output:
```

```
Error: unsupported operand type(s) for /: 'str' and 'int'. Ensure both x
and y are numbers!
compute_division execution completed.
None
```

What is happening in the code:

1. **try Block:** This is where we place the code that might raise exceptions. In the example, we are attempting to divide two numbers.
2. **except Block:** This block catches the exception. We can have multiple except blocks to catch different types of exceptions.

    - **ZeroDivisionError**: Triggered when we try to divide by zero.
    - **TypeError:** Triggered if the inputs are not numbers.

    Each except block can capture the raised exception instance (using **as e** in our case), which can be used to print or log more detailed error messages.
3. **else Block:** This block will run if no exceptions were raised in the try block. Here, we're simply printing the result of the division.
4. **finally Block:** This block is **executed no matter what**, whether an exception was raised or not. It is typically used for cleanup actions, such as closing files. In our example, we are simply printing a completion message.

The test cases at the end show three different scenarios:

- A valid division.
- A division by zero.
- A division with an inappropriate data type.

By utilizing this structured approach to exception handling, we can provide useful feedback on what went wrong, making it easier to debug and offering a better user experience.

Raising Exceptions: The 'raise' Statement

While the primary role of exception handling is to manage unexpected errors, there are times when you might need to intentionally trigger an exception in your code based on specific conditions. This is where the raise statement can be useful.

Using **raise**, you can manually throw an exception in your code. This can be helpful in situations where you want to enforce certain conditions or validate inputs.

How to Use raise

1. **Raising a Predefined Exception:** You can raise any of the built-in exceptions or even create your own.

Jupyter notebook

```
def validate_age(age):
    if age < 0:
```

```
        raise ValueError("Age cannot be negative!")
    return age
try:
    validate_age(-5)
except ValueError as e:
    print(e)

# Output:
Age cannot be negative!
```

In the above example, we are raising a **ValueError** if an invalid age (a negative value) is passed to the validate_age function.

2. **Raising a Custom Exception:** You can also define custom exception classes. These classes typically inherit from the base Exception class or its derivatives (classes will be discussed later in this section).

Jupyter notebook

```
class InvalidAgeError(Exception):
    pass

def validate_age_custom(age):
    if age < 0:
        raise InvalidAgeError("Age cannot be negative!")
    return age
try:
    validate_age_custom(-5)
except InvalidAgeError as e:
    print(e)

# Output:
Age cannot be negative!
```

In this example, we have created a custom exception called **InvalidAgeError**. The validate_age_custom function raises this custom exception if an invalid age is provided.

Why Use raise?

- **Enforce Constraints:** As shown in the examples, you might want to enforce certain conditions in your code, like not allowing negative ages.
- **Informative Feedback**: By raising exceptions with descriptive messages, you give the calling code (or the user) clear feedback about what went wrong.

- **Controlled Failures:** Instead of letting your program fail unpredictably when it encounters an unexpected state, you can proactively check for these states and raise an exception.

In summary, the raise statement is a powerful tool in Python's exception handling, allowing developers to trigger exceptions intentionally based on custom conditions or validations.

Functions

A function in programming is like a predefined procedure or routine that performs a specific task. In Python, functions are the foundations that enable modular and efficient coding. Functions encapsulate a set of operations into a single unit, making it **reusable** across various parts of the code. This encapsulation ensures that the code is **DRY** (**Don't Repeat Yourself**), and comforts debugging and maintenance.

In the context of a data mining project for animal behavior detection, functions can make repetitive tasks such as data extraction, preprocessing, and feature extraction more efficient.

The Syntax of a Function

A function in Python is defined using the **def** keyword, followed by the **function name** and parentheses **()**. The code block within every function is **indented** and starts with a colon **:**.

Jupyter notebook

```
def function_name(parameters):

    """docstring"""

    # Function body

    return result
```

- **Function Name**: Represents what the function does. Following a descriptive naming convention is crucial for code readability.
- **Parameters (or Arguments)**: These are the values you can pass into the function. A function can have no parameters or multiple parameters separated by commas.
- **Docstring:** It is a type of comment that describes what the function does and is enclosed between triple quotes. It is optional but recommended, especially for complex functions
- **Function Body**: Contains the statements that the function will execute.
- **Return Statement**: Outputs a value when the function is called. A function can also have no return statement, in which case it returns None.

Function Properties

1. **Reusability**: One of the primary benefits of functions is the reusability of code. Once a function is defined, it can be utilized multiple times throughout a program.

2. **Scope**: Variables defined inside a function are local to that function and cannot be accessed outside it. This encapsulation ensures variables in functions do not accidentally modify variables in the main program.
3. **Arguments**: Functions can accept multiple types of arguments:
    - **Positional Arguments:** Read in order from the function call.
    - **Keyword Arguments:** Passed by explicitly naming the variable in the function call.
    - **Default Arguments**: Default values can be assigned to parameters in the function definition. This means the function can be called without providing that particular argument.
    - **Variable-length Arguments**: Allows for an arbitrary number of arguments using ***args** for positional arguments and ****kwargs** for keyword arguments.
4. **Return Values:** A function can return multiple values using tuples.

With this foundational understanding, we can now proceed to explore the application of functions in the context of our animal behavior detection project.

Example: Creating dummy dataset to use

Often in data science, before obtaining real-world data or testing a new method, we might need to create simulated data. Simulated data can provide an understanding of how algorithms might perform or help in setting up the workflow.

The function below simulates a dummy dataset for animal behavior:

Jupyter notebook

```
import random
import csv

def simulate_data(num_samples=1000):
    """Simulate a dataset for animal behavior."""
    dataset = []
    for _ in range(num_samples):
        sample = {
            'activity_duration': random.uniform(0, 100),
            'proximity_to_others': random.uniform(0, 10),
            'noise_levels': random.uniform(50, 100)
        }
        dataset.append(sample)

    # Save the dataset to a CSV
    filename = "simulated_dataset.csv"
    keys = dataset[0].keys()
    with open(filename, 'w', newline='') as output_file:
        dict_writer = csv.DictWriter(output_file, keys)
        dict_writer.writeheader()
        dict_writer.writerows(dataset)

    return f"Data simulated and saved to {filename}"
```

```
# Use the function to generate simulated data
print(simulate_data())

# Output:
Data simulated and saved to simulated_dataset.csv
```

Breaking down the code step-by-step:

- **'import random'**: The '**random'** module provides functions to generate random numbers, which are utilized here to simulate data.
- **'import csv**': The '**csv'** module in Python is used to read and write CSV files, making it possible to store our simulated data in a structured format that's easy to share and analyze.
- **'def simulate_data(num_samples=1000):**': We are defining a function named **'simulate_data'**. This function accepts an optional argument **'num_samples'** that determines how many data samples to create. By default, we set it to 1000 but it could be any number.
- **'dataset = []'**: This initializes an empty **list** named dataset. As we proceed, this list will be filled with simulated data samples.
- **'for _ in range(num_samples):'**: A **for** loop iterating **'num_samples'** times. Here, '**_**' is a common convention in Python indicating that the loop variable is not going to be used inside the loop.
- **'sample = {...}'**: Inside the loop, we are creating a **dictionary** named sample. This dictionary contains three **key-value** pairs. Each pair corresponds to a feature of our data (like **'activity_duration'**) and its simulated value.
- **'random.uniform(a, b)'**: This function from the **random module** generates a random float number between **a** and **b**.
- **'dataset.append(sample)**': This line **adds** the generated sample **dictionary** to our dataset **list**.
- **'filename = "simulated_dataset.csv"'**: We are assigning the name of the CSV file where the data will be saved.
- **'keys = dataset[0].keys()'**: Here, we are extracting the **keys** (or **column names**) from the first dictionary in our dataset list.
- **'with open(filename, 'w', newline='') as output_file:'**: This is a context manager that opens the specified file in write mode (**'w'**). The **with** statement ensures that the file is properly closed after its operation finishes.
- **'csv.DictWriter(output_file, keys)'**: This creates a **writer object** for dictionaries. This means we can write our list of dictionaries (dataset) directly to a CSV file.
- **'dict_writer.writeheader()'**: This writes the titles of columns (headers).
- **'dict_writer.writerows(dataset)'**: This writes the dataset list to the CSV file.
- **'return f"Data simulated and saved to {filename}"':** The function ends with a **return** statement, which sends back a **string** indicating that the data was successfully simulated and saved.

- **'print(simulate_data())':** Here, we are calling the **simulate_data function** to simulate the data and print the result

This function simulates data and also saves it as a CSV file for future use. The advantage here is twofold – we are generating a placeholder dataset and ensuring it is stored in a persistent manner for subsequent analyses.

Example: Data extraction function

Once our data is saved, the next logical step in our workflow is to retrieve and preprocess it. This retrieval can be effectively done using a function:

Jupyter notebook

```
import pandas as pd

def extract_and_preprocess(filename="simulated_dataset.csv"):
    """Extract data from the provided file and preprocess it."""

    # Load data from CSV
    data = pd.read_csv(filename)

    # Extract features
    features    =    data[['activity_duration',    'proximity_to_others',
'noise_levels']]

    # For demonstration, we are normalizing the features.
    features_normalized = (features - features.mean()) / features.std()

    return features_normalized

# Use the function to extract and preprocess data
processed_data = extract_and_preprocess()
print(processed_data.head())

# Output:
  activity_duration  proximity_to_others  noise_levels
0         -0.750392            -1.557333      0.626784
1         -1.708669            -0.529892      1.078318
2         -0.725235             0.147800     -1.442999
3          1.559831            -0.196572      0.482220
4          0.940615             0.324937     -1.220871
```

Breaking down the code step-by-step:

- **import:** This command, as was introduced before, is used to include external libraries or modules in your Python script.
- **pandas**: A library in Python used for data manipulation and analysis.

- **as pd**: This is an alias. Instead of using **pandas.read_csv(...)**, you can use the shorthand **pd.read_csv(...**).
- **def**: This keyword is used to declare or define a function in Python.
- **extract_and_preprocess**: This is the name of the function.
- **(filename="simulated_dataset.csv")**: This is a parameter with a default value. If no argument is provided when calling the function, it will use "simulated_dataset.csv" as the filename.
- **pd.read_csv(...)**: This function from the **pandas** library reads a comma-separated values (CSV) file into a **DataFrame**. A **DataFrame** is a **2-dimensional** data structure with columns that can be of different types, similar to a spreadsheet or SQL table.
- **data[['activity_duration', 'proximity_to_others', 'noise_levels']]**: This line extracts specific columns from the DataFrame. Here, it is extracting the columns named **'activity_duration', 'proximity_to_others'**, and **'noise_levels'**.
- **features.mean()**: This calculates the **mean** (average) of each column in the **features** DataFrame.
- **features.std()**: This calculates the standard deviation of each column in the **features** DataFrame.
- **(features - features.mean()) / features.std()**: This is normalizing the data. Normalization is a technique often applied as part of data preparation for machine learning. The goal of normalization is to change the values of numeric columns in the dataset to use a **common scale**, without distorting differences in the ranges of values.
- **return**: This keyword is used to send a result back to the caller of the function. Here, it is returning the **features_normalized** DataFrame.
- **processed_data = extract_and_preprocess()**: This line calls the **extract_and_preprocess** function and stores the result in the **processed_data** variable.
- **print(processed_data.head())**: The **head()** function returns the first 5 rows of the DataFrame, and **print** displays it. This is useful for quickly inspecting the top few records of your processed data.

The above function **reads** data from the previously saved CSV file, **extracts** key features, and **normalizes** them. **Normalizing** is a common preprocessing step in machine learning, ensuring that all features have a similar scale.

Additional Information:

for loops and **dictionaries**: These were introduced in previous sections. While they are not explicitly used in this code snippet, understanding them is crucial for more complex operations and data handling in Python.

CSV (Comma-Separated Values): A CSV is a simple file format used to store tabular data, such as a spreadsheet or database. Data fields in a CSV file are separated by commas.

With the foundational knowledge of functions in place, and by viewing the given examples, we can appreciate the importance of these modular code blocks in the data science workflow.

Classes and Object-Oriented Programming in Python

Python, like many modern programming languages, supports both procedural and object-oriented programming (OOP). In OOP, Python uses classes as a mechanism to define objects and their behaviors. We will not investigate in deep the details of object-oriented programming, however, it is essential to have a foundational understanding of classes.

What is a Class?

A class can be thought of as a blueprint for creating objects. Objects, in turn, represent entities in your code that can hold data and perform actions. For instance, if you were building a simulation of a farm, you might have classes like **'Animal'**, **'Bird'**, and **'Mammal'**.

Why Use Classes?

- **Encapsulation**: Classes bundle data (**attributes**) and methods (**functions**) into a single entity. This **encapsulation** ensures that the data is firmly coupled with the operations that can be performed on it.
- **Inheritance**: Classes can **inherit** attributes and behaviors from other classes, allowing for code reuse and the establishment of relationships between classes.
- **Polymorphism**: Different classes can be treated as instances of the same class through inheritance. This allows for writing more generic and reusable code.

Basic Syntax:

Jupyter notebook

```
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass

class Bird(Animal):
    def speak(self):
        return "Cluck!"

class Mammal(Animal):
    def speak(self):
        return "moo!"

# Using the classes
hen = Bird("Hen")
cow = Mammal("Cow")

print(hen.speak())  # Output: Cluck!
print(cow.speak())  # Output: Moo!
```

In the above code:

- **Animal** is a base class.
- **Bird** and **Mammal** are derived **classes** that **inherit** from Animal.
- The **speak** method is defined in the base class but overridden in derived classes, showcasing **polymorphism**.

Here is another example to illustrate the concept:

Jupyter notebook

```
import pandas as pd

class AnimalBehaviorData:

    def __init__(self, filename="simulated_dataset.csv"):
        self.filename = filename
        self.data = None
        self.processed_data = None

    def extract_and_preprocess(self):
        self.data = pd.read_csv(self.filename)
        features = self.data[['activity_duration',
                              'proximity_to_others',
                              'noise_levels']]
        self.processed_data = (features - features.mean())
                              / features.std()

    def display_data(self):
        print(self.processed_data.head())

# Create an instance of the class and use its methods
dataset = AnimalBehaviorData()
dataset.extract_and_preprocess()
dataset.display_data()#


# Output:
   activity_duration  proximity_to_others  noise_levels
0          -0.750392            -1.557333      0.626784
1          -1.708669            -0.529892      1.078318
2          -0.725235             0.147800     -1.442999
3           1.559831            -0.196572      0.482220
4           0.940615             0.324937     -1.220871
```

Code Breakdown:

Jupyter notebook

```
class AnimalBehaviorData:
```

- **class:** This keyword is used to **declare** a class in Python.
- **AnimalBehaviorData:** This is the **name** of our class. It follows the convention of using CamelCase[1] for class names.

Jupyter notebook

```
def __init__(self, filename="simulated_dataset.csv"):
```

- **__init__:** This is a special method in Python, known as a **constructor**. It is called automatically when an instance of the class is created. It can have default and non-default parameters.
- **self:** Refers to the instance of the class. It binds the attributes with the given arguments.

Jupyter notebook

```
        self.filename = filename
        self.data = None
        self.processed_data = None
```

- These are instance variables. They hold data that belongs to an instance of the class.
- **self.filename:** This assigns the passed **filename** (or its default value) to the **filename** attribute of the instance.
- **self.data:** Initializes a placeholder to hold the raw data once it is read.
- **self.processed_data:** A placeholder to store the data after preprocessing.

[1] CamelCase is a convention used in programming to write compound words or phrases without spaces or punctuation, where each word within the phrase is capitalized except for the first one. This creates a distinct visual pattern resembling the humps on a camel's back, hence the name. For example, "myVariableName" and "calculateTotalAmount" are examples of identifiers written in CamelCase. It is commonly used for naming variables, functions, and class names in many programming languages.

Jupyter notebook

```
    def extract_and_preprocess(self):
        self.data = pd.read_csv(self.filename)
        features = self.data[['activity_duration', 'proximity_to_others',
                              'noise_levels']]
        self.processed_data = (features - features.mean())/features.std()
```

- **def extract_and_preprocess(self):** This is an instance method (a function defined inside a class). It operates on the attributes of the instance and may modify them.

Jupyter notebook

```
        self.data = pd.read_csv(self.filename)
        features = self.data[['activity_duration', 'proximity_to_others',
                              'noise_levels']]
        self.processed_data = (features - features.mean())/features.std()
```

- This method employs the previously discussed data extraction and preprocessing steps, now contained within a class structure.

Jupyter notebook

```
            def display_data(self):
```

- Another instance method designed to display the first few rows of the processed data.

Jupyter notebook

```
  print(self.processed_data.head())
```

- This line prints the first five rows of the processed data to the console.

Jupyter notebook

```
dataset = AnimalBehaviorData()
dataset.extract_and_preprocess()
dataset.display_data()
```

- **dataset = AnimalBehaviorData()**: Creates an instance of the **AnimalBehaviorData** class. This triggers the **__init__** method and sets up our instance with the default filename.
- **dataset.extract_and_preprocess()**: Calls the **extract_and_preprocess** method on our **dataset** instance. This reads the data from the given filename and processes it.
- **dataset.display_data()**: Calls up the **display_data** method on our **dataset** instance, which in turn prints the first five rows of the processed data to the console.

From the above demonstration, it may seem like just an alternative way of organizing the code. So, why bother with classes? The real power of classes lies in their ability to represent and **encapsulate** real-world entities and their behaviors in code.

Libraries

Pandas and **NumPy** are two libraries that form the base of data manipulation and numerical computing in Python.

Pandas

Pandas is an open-source data analysis and data manipulation library. It offers data structures for efficiently storing large and complex datasets and provides a rich set of functions to manipulate these structures. The primary data structures include **Series** and **DataFrame**. A Series represents a one-dimensional array like an object, whereas a DataFrame represents a two-dimensional tabular data structure. Pandas integrates seamlessly with other libraries such as Matplotlib and Seaborn for data visualization, as well as with scikit-learn for machine learning tasks.

NumPy

NumPy, short for Numerical Python, is another fundamental package for numerical computations in Python. It provides support for arrays (including multidimensional arrays), as well as an assortment of mathematical functions to operate on these arrays. NumPy serves as the foundational package for numerical computing and plays a significant role in scientific computing, which also makes it an essential part of our toolkit for machine learning experiments related to farm animal activity recognition.

Additional Resources

For a more comprehensive review of the methods, functions, and operations that Pandas and NumPy offer, kindly refer to the GitHub repository of the accompanied book. Each topic is elaborated there in detail, providing examples and elaborations for better understanding.

Jupyter Notebooks on GitHub

All code examples, including those demonstrating the use of Pandas and NumPy, can be found in the **https://github.com/nkcAna/WSDpython**.

Matplotlib

Matplotlib is one of the most popular data visualization libraries in Python. It provides an object-oriented API for embedding plots into applications and is also used for basic plotting. Matplotlib's flexibility allows us to generate line plots, scatter plots, bar plots, histograms, 3D plots, and much more.

For more information and to explore the capabilities it offers, you can visit the Matplotlib website and have a look at the available cheatsheets and handouts at https://matplotlib.org/cheatsheets/

Seaborn

Seaborn builds on Matplotlib and provides a higher-level, more convenient interface to create beautifully styled statistical plots. One of its greatest advantages is its ability to bring a whole new level of sophistication and visual appeal to Matplotlib plots. You may explore further by following the Seaborn official documentation at https://seaborn.pydata.org/.

scikit-learn

Scikit-learn is an open-source Python library offering efficient tools for data mining and data analysis. It supports numerous machine learning algorithms for both supervised and unsupervised learning. Known for its ease of use and versatility, it integrates well with NumPy and SciPy, making it a popular choice for data scientists and researchers for a broad range of applications. More details can be found in the scikit-learn official documentation https://scikit-learn.org/stable/.

By referring to the official documentation for each library, you can gain a deeper understanding of their comprehensive capabilities.