Python Notebook

This handbook covers essential Python topics with examples and explanations.

Table of Contents

- 1 Basic
- 1.1 Syntax
  - 1.1.1 Indentation
  - 1.1.2 Comments
  - 1.1.3 Variables
  - 1.1.4 Data Types
- 2 Operators
  - 2.1 Arithmetic Operators
  - 2.2 Comparison Operators
  - 2.3 Logical Operators
  - 2.4 Assignment Operators
- 3 Control Flow
  - 3.1 Conditional Statements
  - 3.2 Loops
- 4 Python Functions
- 5 Data Structures
  - 5.1 Lists
  - 5.2 Tuples
  - 5.3 Dictionaries
  - 5.4 Sets
- 6 File Handling
- 7 Error Handling
- 8 Object-Oriented Programming
- 9 Modules and Packages
- 10 Advanced Topics

1 Basic

1.1 Syntax

Python syntax is designed to be easy to read and understand. Key aspects include:

- **Indentation**: Python uses indentation to define code blocks.
- **Comments**: Helpful for documentation.
- **Variables**: Containers for storing data values.
- **Data Types**: Define the nature of data stored in variables.

1.1.1 Indentation

Indentation in Python is mandatory and helps define code blocks. Each block level is indented by 4 spaces.

In [6]:
```
def example_function():
    if True:
        print("This line is indented.")
example_function()
```

```
This line is indented.
```

1.1.2 Comments

Comments in Python start with `#` and are not executed by the interpreter. Multi-line comments can be written with triple quotes.

In [8]:
```
# This is a single-line comment
"""This is a multi-line comment
spanning multiple lines."""
print("Comments are ignored by the interpreter.")
```

```
Comments are ignored by the interpreter.
```

1.1.3 Variables

Variables are created when you assign a value to them using the `=` operator.

In [10]:
```
x = 10
y = "Hello, Python!"
z = True
print(x, y, z)
```

```
10 Hello, Python! True
```

1.1.4 Data Types

Python supports various data types, including:

- **Integers**: Whole numbers
- **Floats**: Decimal numbers
- **Strings**: Text
- **Booleans**: True or False
- **Lists**: Ordered, mutable collections
- **Tuples**: Ordered, immutable collections
- **Dictionaries**: Key-value pairs
- **Sets**: Unordered collections of unique elements

In [63]:
```
# Examples of different data types
int_var = 42
float_var = 3.14159
str_var = "Python"
bool_var = True
list_var = [1, 2, 3]
tuple_var = (1, 2, 3)
```

```
dict_var = {"name": "Alice", "age": 25}
set_var = {1, 2, 3}

print(f"int_var: {type(int_var).__name__}")
print(f"float_var: {type(float_var).__name__}")
print(f"str_var: {type(str_var).__name__}")
print(f"bool_var: {type(bool_var).__name__}")
print(f"list_var: {type(list_var).__name__}")
print(f"tuple_var: {type(tuple_var).__name__}")
print(f"dict_var: {type(dict_var).__name__}")
print(f"set_var: {type(set_var).__name__}")
```

```
int_var: int
float_var: float
str_var: str
bool_var: bool
list_var: list
tuple_var: tuple
dict_var: dict
set_var: set
```

2 Operators

Operators are special symbols used to perform operations on variables and values. Python supports the following types of operators:

- **Arithmetic Operators**: `+` , `-` , `*` , `/` , `//` , `%` , `**`
- **Comparison Operators**: `==` , `!=` , `>` , `<` , `>=` , `<=`
- **Logical Operators**: `and` , `or` , `not`
- **Assignment Operators**: `=` , `+=` , `-=` , `*=` , `/=` , `//=` , `%=` , `**=`

2.1 Arithmetic Operators

In [61]:
```
a, b = 10, 5

print(f"a + b = {a + b}")
print(f"a - b = {a - b}")
print(f"a * b = {a * b}")
print(f"a / b = {a / b}")
print(f"a // b = {a // b}")
print(f"a % b = {a % b}")
print(f"a ** b = {a ** b}")
```

```
a + b = 15
a - b = 5
a * b = 50
a / b = 2.0
a // b = 2
a % b = 0
a ** b = 100000
```

2.2 Comparison Operators

In [65]:
```
x, y = 10, 5

print(f"x == y: {x == y}")
```

```
print(f"x != y: {x != y}")
print(f"x > y: {x > y}")
print(f"x < y: {x < y}")
print(f"x >= y: {x >= y}")
print(f"x <= y: {x <= y}")
```

```
x == y: False
x != y: True
x > y: True
x < y: False
x >= y: True
x <= y: False
```

2.3 Logical Operators

In [67]:
```
a, b = True, False

print(f"a and b: {a and b}")
print(f"a or b: {a or b}")
print(f"not a: {not a}")
```

```
a and b: False
a or b: True
not a: False
```

2.4 Assignment Operators

In [69]:
```
x = 5
print(f"x = {x}")

x += 3
print(f"x += 3: {x}")

x -= 2
print(f"x -= 2: {x}")

x *= 4
print(f"x *= 4: {x}")

x /= 2
print(f"x /= 2: {x}")

x //= 2
print(f"x //= 2: {x}")

x %= 3
print(f"x %= 3: {x}")

x **= 2
print(f"x **= 2: {x}")
```

```
x = 5
x += 3: 8
x -= 2: 6
x *= 4: 24
x /= 2: 12.0
x //= 2: 6.0
x %= 3: 0.0
x **= 2: 0.0
```

3 Control Flow

Control flow allows you to control the execution order of code using conditional statements and loops.

3.1 Conditional Statements

- `if` checks a condition
- `elif` provides additional checks
- `else` executes if no conditions match

In [24]:
```
x = 10
if x > 10:
    print("Greater than 10")
elif x == 10:
    print("Equal to 10")
else:
    print("Less than 10")
```

```
Equal to 10
```

3.2 Loops

- **For Loop**: Iterates over a sequence.
- **While Loop**: Repeats as long as a condition is true.
- **break**: Exits the loop.
- **continue**: Skips to the next iteration.
- **pass**: Does nothing; placeholder.

In [26]:
```
# Example of for and while loop with break and continue
for i in range(5):
    if i == 3:
        continue  # Skips when i is 3
    print("For loop iteration:", i)

count = 0
while count < 5:
    print("While loop iteration:", count)
    if count == 3:
        break  # Exits the loop when count is 3
    count += 1

# pass statement example
for i in range(3):
    pass  # Placeholder; does nothing
```

```
For loop iteration: 0
For loop iteration: 1
For loop iteration: 2
For loop iteration: 4
While loop iteration: 0
While loop iteration: 1
While loop iteration: 2
While loop iteration: 3
```

4 Python Functions

Functions are reusable blocks of code that perform a specific task.

- **Defining Functions**: `def` keyword is used.
- **Parameters and Arguments**: Values that functions accept.
- **Return Statements**: Specify what a function returns.
- **Lambda Functions**: Anonymous, single-line functions.
- **Built-in Functions**: Python has built-in functions like `len`, `print`, `sum`, etc.

In [71]:
```python
# Defining a function with parameters and a return statement
# The `add` function takes two parameters `a` and `b` and returns their sum.
def add(a, b):
    return a + b

# Calling the `add` function with arguments 10 and 5 and storing the result in `
result = add(10, 5)
print("Addition Result:", result)

# Lambda function example
# Defining an anonymous (lambda) function to multiply two values `x` and `y`.
multiply = lambda x, y: x * y

# Calling the lambda function with arguments 4 and 3 and printing the result.
print("Multiplication Result:", multiply(4, 3))
```

```
Addition Result: 15
Multiplication Result: 12
```

5 Data Structures

Python provides various data structures to store collections of data.

- **Lists**: Mutable, ordered collections.
- **Tuples**: Immutable, ordered collections.
- **Dictionaries**: Key-value pairs.
- **Sets**: Unordered, unique items.

5.1 Lists

- Lists are defined with square brackets and can contain mixed data types.
- **Indexing** and **Slicing** allow you to access specific elements.
- Lists also support methods like `append`, `remove`, `pop`, and `sort`.

In [31]:
```
# List example with indexing and slicing
fruits = ["apple", "banana", "cherry", "date"]
print("First fruit:", fruits[0])
print("Last two fruits:", fruits[-2:])

# List methods
fruits.append("elderberry")
fruits.remove("banana")
print("Updated list:", fruits)
```

```
First fruit: apple
Last two fruits: ['cherry', 'date']
Updated list: ['apple', 'cherry', 'date', 'elderberry']
```

5.2 Tuples

- Tuples are defined with parentheses and are immutable (cannot be changed after creation).
- **Indexing** and **Slicing** allow you to access specific elements.
- Tuples **do not** support methods like `append` or `remove` because they are immutable.

In [77]:
```
# Tuple example with indexing and slicing
fruits = ("apple", "banana", "cherry", "date")
print("First fruit:", fruits[0])
print("Last two fruits:", fruits[-2:])

# Attempting to modify the tuple will result in an error
# fruits[0] = "avocado"  # Uncommenting this line would cause an error
```

```
First fruit: apple
Last two fruits: ('cherry', 'date')
```

5.3 Dictionaries

- Dictionaries are defined with curly braces and store data as key-value pairs.
- You can **access**, **add**, and **modify** elements using keys.
- Dictionaries support methods like `get`, `keys`, `values`, `items`, and `update`.

In [80]:
```
# Dictionary example with key-value access
person = {"name": "Alice", "age": 25, "city": "New York"}
print("Name:", person["name"])
print("Age:", person["age"])

# Dictionary methods
person["age"] = 26  # Updating age
person["profession"] = "Engineer"  # Adding a new key-value pair
print("Updated dictionary:", person)
```

```
Name: Alice
Age: 25
Updated dictionary: {'name': 'Alice', 'age': 26, 'city': 'New York', 'professio
n': 'Engineer'}
```

5.4 Sets

- Sets are defined with curly braces and contain unordered, unique items.
- Sets **do not** support **indexing** or **slicing** due to their unordered nature.
- Sets support methods like `add`, `remove`, `pop`, `union`, `intersectio`, and `difference`.

In [83]:
```
# Set example with unique elements
fruits = {"apple", "banana", "cherry"}
print("Initial set:", fruits)

# Set methods
fruits.add("date")  # Adding an element
fruits.remove("banana")  # Removing an element
print("Updated set:", fruits)
```

```
Initial set: {'apple', 'banana', 'cherry'}
Updated set: {'date', 'apple', 'cherry'}
```

6 File Handling

Python provides functionality to work with files.

- **Opening and Closing Files**: `open()` and `close()` methods.
- **Reading from Files**: Using `read()` or `readlines()`.
- **Writing to Files**: Using `write()`.
- **Working with CSV Files**: Using `csv` library.

File Modes

When opening a file, you can specify the mode:

- `'r'`: Read (default)
- `'w'`: Write (creates a new file or truncates an existing file)
- `'a'`: Append
- `'b'`: Binary mode
- `'x'`: Exclusive creation

Opening and Closing Files

Files must be opened before they can be manipulated. Always close files after operations to free up system resources.

6.1 Opening a File

```
# Open a file in read mode
file = open('example.txt', 'r')
```

6.2 Close the file

```
file.close()
```

6.3 Reading File

6.3.1 Reading the Entire File

```
# Open the file and read its content
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)
```

6.3.2 Reading Line by Line

```
with open('example.txt', 'r') as file:
    for line in file:
        print(line.strip())  # Strip removes leading/trailing
whitespace
```

6.3.3 Reading Specific Lines

```
# Read specific lines using readlines()
with open('example.txt', 'r') as file:
    lines = file.readlines()
    print(lines[0])  # Print the first line
    print(lines[1])  # Print the second line
```

6.4 Writing to a File

6.4.1 Writing New Content

```
# Write new content to a file
with open('output.txt', 'w') as file:
    file.write('Hello, World!\n')
    file.write('Welcome to file handling in Python.\n')
```

6.4.2 Appending Content

```
# Append content to an existing file
with open('output.txt', 'a') as file:
    file.write('This line is appended to the file.\n')
```

6.5 Working with Binary Files

6.5.1 Writing a Binary File

```
data = bytearray([120, 3, 255, 0, 100])
with open('output.bin', 'wb') as file:
    file.write(data)
```

6.5.2 Reading a Binary File

```
with open('output.bin', 'rb') as file:
    binary_data = file.read()
    print(binary_data)
```

6.6 Exception Handling

In [68]:
```
try:
    with open('non_existent_file.txt', 'r') as file:
```

```
        content = file.read()
except FileNotFoundError:
    print('File not found. Please check the file name and path.')
```

```
File not found. Please check the file name and path.
```

7 Error Handling

Python uses `try`, `except`, `finally`, and `raise` for error handling.

7.1 Types of Errors

There are two main types of errors in Python:

1. **Syntax Errors**: Occur when the code violates the syntax rules.
2. **Exceptions**: Runtime errors that occur when the program is running.

Using Try and Except

The `try` block lets you test a block of code for errors, while the `except` block lets you handle the error.

Basic Example

```
try:
    result = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
```

7.2 Catching Multiple Exceptions

```
try:
    value = int(input("Enter a number: "))
    result = 10 / value
except (ZeroDivisionError, ValueError) as e:
    print(f"Error: {e}")
```

7.3 Using Finally

The `finally` block allows you to execute code regardless of whether an exception occurred or not. It's typically used for cleanup actions.

```
try:
    file = open('example.txt', 'r')
    content = file.read()
except FileNotFoundError:
    print("File not found.")
finally:
    file.close()  # Ensure the file is closed
```

7.4 Raising Exceptions

You can `raise` exceptions intentionally using the raise statement.

```
def check_age(age):
    if age < 18:
        raise ValueError("Age must be 18 or older.")
    return "Access granted."

try:
    print(check_age(15))
except ValueError as e:
    print(f"Error: {e}")
```

8 Object-Oriented Programming

8.1 Class

A class in Python can be thought of as a blueprint for creating objects. Objects have two main characteristics: `variables` and `methods`. Variables are the characteristics of the object, while methods are functions that perform operations on the object.

8.1.1 Instance

An instance is a specific object created from a class.

In [37]:
```
# Defining a simple class and creating an object
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return f"{self.name} makes a sound"

cat = Animal("Cat")
print(cat.speak())
```

```
Cat makes a sound
```

9 Modules and Packages

Modules and packages allow you to organize and reuse code across files.

In [39]:
```
# Importing and using a module
import math
print(math.sqrt(16))
```

```
4.0
```

10 Advanced Topics

Advanced Python includes list comprehensions, generator expressions, and decorators.

In [41]:
```
# List comprehension
squares = [x**2 for x in range(10)]
print(squares)
```

```
# Generator expression
gen = (x**2 for x in range(10))
print(list(gen))

# Decorator example
def decorator(func):
    def wrapper():
        print("Before function call")
        func()
        print("After function call")
    return wrapper

@decorator
def say_hello():
    print("Hello!")

say_hello()
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
Before function call
Hello!
After function call
```

11 Machine Learning Using Python

Libraries Required

To get started, we need the following libraries:

- `numpy` : For numerical computations
- `pandas` : For data manip/ulation
- `matplotlib` : For data visualization
- `scikit-learn` : For implementing machine learning algorithms

In [10]:
```
# Generating a Dataset
from sklearn.datasets import make_classification
import pandas as pd

# Generate synthetic dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=2, n_red

# Create a DataFrame
df = pd.DataFrame(X, columns=[f'feature_{i}' for i in range(20)])
df['target'] = y

# Display the first few rows of the dataset
df.head()
```

Out[10]:

	feature_0	feature_1	feature_2	feature_3	feature_4	feature_5	feature_6	feature_7
0	-0.669356	-1.495778	-0.870766	1.141831	0.021606	1.730630	-1.251698	0.289305
1	0.093372	0.785848	0.105754	1.272354	-0.846316	-0.979093	1.263707	0.264020
2	-0.905797	-0.608341	0.295141	0.943716	0.092936	1.370397	-0.064772	0.287273
3	-0.585793	0.389279	0.698816	0.436236	-0.315082	0.459505	1.448820	0.505558
4	1.146441	0.515579	-1.222895	-0.396230	-1.293508	-0.352428	0.071254	1.239584

5 rows × 21 columns

11.1 Splitting the Data

In [15]:
```
from sklearn.model_selection import train_test_split

# Split the dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_
```

11.2 Feature Scaling

In [18]:
```
from sklearn.preprocessing import StandardScaler

# Scale features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

11.3 Implementing Machine Learning Algorithms

11.3.1 Logistic Regression

In [22]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Initialize and fit the model
model = LogisticRegression()
model.fit(X_train_scaled, y_train)

# Make predictions
y_pred = model.predict(X_test_scaled)

# Evaluate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')
```

```
Accuracy: 0.85
```

11.3.2 Decision Tree Classifier

In [25]:
```
from sklearn.tree import DecisionTreeClassifier
```

```
# Initialize and fit the model
dt_model = DecisionTreeClassifier(random_state=42)
dt_model.fit(X_train, y_train)

# Make predictions
dt_y_pred = dt_model.predict(X_test)

# Evaluate accuracy
dt_accuracy = accuracy_score(y_test, dt_y_pred)
print(f'Decision Tree Accuracy: {dt_accuracy:.2f}')
```

```
Decision Tree Accuracy: 0.88
```

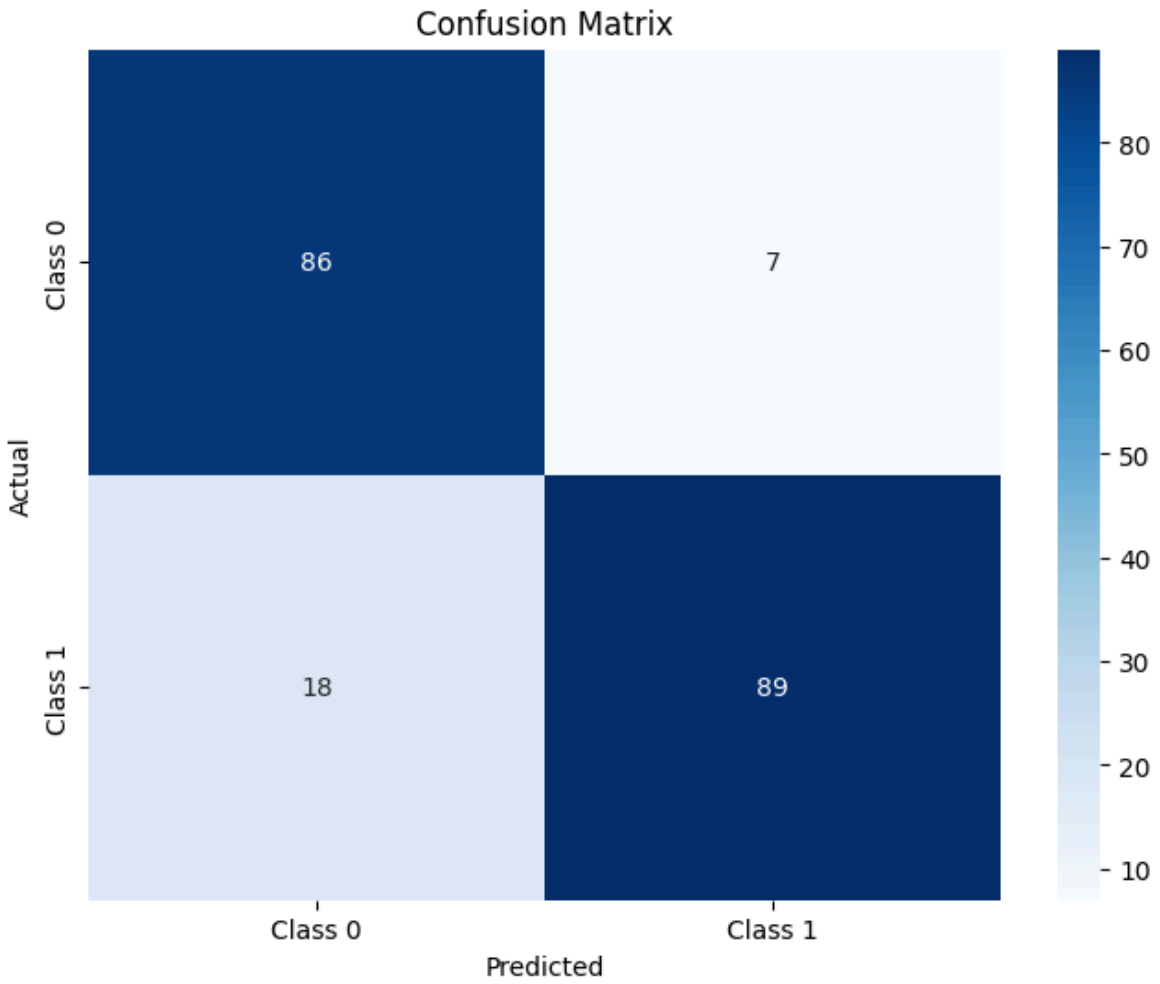
11.4 Model Evaluation

11.4.1 Confusion Matrix

In [38]:
```
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Generate confusion matrix
cm = confusion_matrix(y_test, dt_y_pred)

# Plot confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Class 0', 'Clas
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.title('Confusion Matrix')
plt.show()
```



11.4.2 Classification Report

In [32]:
```
from sklearn.metrics import classification_report

# Print classification report
report = classification_report(y_test, dt_y_pred)
print(report)
```

```
              precision    recall  f1-score   support

           0       0.83      0.92      0.87        93
           1       0.93      0.83      0.88       107

    accuracy                           0.88       200
   macro avg       0.88      0.88      0.87       200
weighted avg       0.88      0.88      0.88       200
```