Python Lists

Concept	Description & Syntax
Definition	Data structure written using square brackets `[]`. Allows storage of multiple objects. `my_list = [1, "apple", True]`
Indexing	Zero-based. First element is at index 0. `my_list[0]`
Negative Indexing	Access from the end. `-1` is the last element. `my_list[-1]`
Slicing	Access a range of elements. `list[start:end]`. `my_list[1:3]`
Adding Elements	`append(item)` adds to the end. `insert(index, item)` adds at a position. `my_list.append("banana")`
Removing Elements	`pop(index)` removes and returns item. `remove(value)` removes first occurrence. `del list[index]` deletes. `my_list.pop()`
Length	Returns the number of items. `len(my_list)`
Sorting	`list.sort()` sorts in-place. `sorted(list)` returns a new sorted list. `my_list.sort()`

For Loops

Concept	Description & Syntax
Basic Syntax	Iterates over a sequence. `for item in sequence:`
Range Function	Generates a sequence of numbers. `range(start, stop, step)`. `for i in range(5):`
Indentation	Code inside the loop must be indented (standard is 4 spaces).

List Comprehension

Concept	Description & Syntax
Basic	Concise way to create lists. `[expression for item in iterable]`. `[x**2 for x in range(5)]`
With Condition	Filter items. `[expression for item in iterable if condition]`. `[x for x in numbers if x % 2 == 0]`

Tuples

Concept	Description & Syntax
Definition	Immutable sequence. Defined with parentheses `()` . `my_tuple = (1, 2, 3)`
Immutability	Cannot be changed after creation (no append, remove, or assignment).
Unpacking	Assign tuple values to variables. `a, b, c = my_tuple`

PEP 8 (Style Guide)

Concept	Description & Syntax
Indentation	Use 4 spaces per indentation level.
Naming	Variables/Functions: `snake_case` . Classes: `CamelCase` . Constants: `UPPER_CASE` .
Whitespace	Avoid extraneous whitespace. Space after commas, around operators.

In [ ]:

```
# 1. Lists: Creation and Manipulation
shopping_list = ["apples", "bread", "milk"]
shopping_list.append("eggs")         # Add to end
shopping_list.insert(1, "butter")    # Insert at index 1
removed_item = shopping_list.pop()   # Remove last item ('eggs')

print(f"Shopping List: {shopping_list}")
print(f"First item: {shopping_list[0]}")
print(f"Last item: {shopping_list[-1]}")

# 2. For Loops and Range
print("\n--- Loop Output ---")
for item in shopping_list:
    print(f"I need to buy {item}")

print("\n--- Range Output ---")
for i in range(1, 4): # 1 to 3
    print(f"Count: {i}")

# 3. List Comprehension
numbers = [1, 2, 3, 4, 5]
# Create a new list of squares
squares = [n**2 for n in numbers]
# Filter for even numbers
evens = [n for n in numbers if n % 2 == 0]

print(f"\nNumbers: {numbers}")
print(f"Squares: {squares}")
print(f"Evens: {evens}")

# 4. Tuples (Immutable)
dimensions = (1920, 1080)
width, height = dimensions # Unpacking
```

```
print(f"\nScreen Dimensions: {width}x{height}")
# dimensions[0] = 1280 # This would cause a TypeError
```

Data Visualization with Matplotlib

Basic Plotting

Concept	Description & Syntax
Importing	Standard alias is `plt`. `import matplotlib.pyplot as plt`
Line Plot	Best for trends over time. `plt.plot(x_values, y_values)`
Scatter Plot	Best for relationships/correlations between two variables. `plt.scatter(x_values, y_values)`
Bar Chart	Best for comparing categorical data. `plt.bar(categories, values)`
Displaying	Renders the plot window. `plt.show()`

Customization

Concept	Description & Syntax

Titles & Labels	Add context to the chart.
	`plt.title("My Chart")`
	`plt.xlabel("X Axis")`
	`plt.ylabel("Y Axis")`

Grid	Adds gridlines for easier reading. `plt.grid(True)`

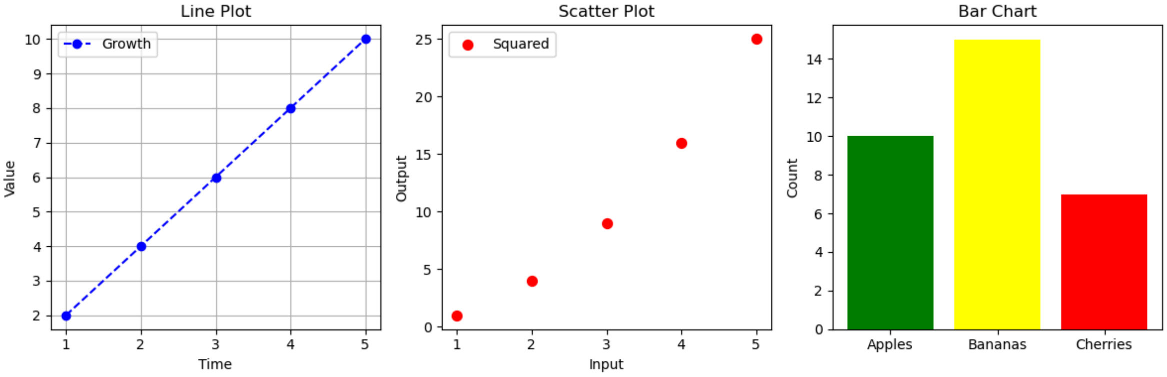
Legends	Identifies multiple datasets. Requires `label` arg in plot.
	`plt.plot(x, y, label="Data 1")`
	`plt.legend()`

Colors & Markers	Customize appearance. `color='red'`, `marker='o'`, `linestyle='--'`
Saving	Save the plot to a file instead of showing it. `plt.savefig("my_plot.png")`

In [1]:

```
import matplotlib.pyplot as plt

# 1. Line Plot Example
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]

plt.figure(figsize=(12, 4)) # Set figure size

# Subplot 1: Line Plot
```

```
plt.subplot(1, 3, 1)
plt.plot(x, y, marker='o', linestyle='--', color='blue', label='Growth')
plt.title("Line Plot")
plt.xlabel("Time")
plt.ylabel("Value")
plt.grid(True)
plt.legend()

# Subplot 2: Scatter Plot
plt.subplot(1, 3, 2)
plt.scatter(x, [v**2 for v in x], color='red', s=50, label='Squared')
plt.title("Scatter Plot")
plt.xlabel("Input")
plt.ylabel("Output")
plt.legend()

# Subplot 3: Bar Chart
categories = ['Apples', 'Bananas', 'Cherries']
counts = [10, 15, 7]
plt.subplot(1, 3, 3)
plt.bar(categories, counts, color=['green', 'yellow', 'red'])
plt.title("Bar Chart")
plt.ylabel("Count")

plt.tight_layout() # Adjust spacing
plt.show()
```



Control Flow and Dictionaries

Conditional Statements

Concept	Description & Syntax
If Statement	Executes code if condition is True. `if age >= 18:`

Elif / Else	Handle alternative conditions.
	`elif age < 13:`
	`else:`

Comparison Operators	`==` (equal), `!=` (not equal), `>`, `<`, `>=`, `<=`
Logical Operators	`and` (both true), `or` (one true), `not` (inverse)

Dictionaries

Concept	Description & Syntax
Definition	Key-value pairs in curly braces `{}`. Keys must be unique/immutable. `user = {'name': 'Alice', 'age': 25}`
Accessing	Use key in square brackets. `user['name']`
Get Method	Safe access (returns None if key missing). `user.get('height')`
Adding/Updating	Assign value to key. `user['city'] = 'London'`
Removing	`pop(key)` removes item. `del dict[key]` deletes. `user.pop('age')`
Looping	Iterate keys, values, or items. `for k, v in user.items():`

While Loops & Input

Concept	Description & Syntax
While Loop	Repeats while condition is True. `while count < 5:`
Break	Exits the loop immediately. `if x == 'quit': break`
Continue	Skips to the next iteration.
User Input	Pauses program to get string from user. `name = input("Enter name: ")`
Type Conversion	Convert input (string) to other types. `age = int(input("Age: "))`

In [2]:

```
# 1. Control Flow (If/Elif/Else)
age = 20
if age >= 18:
    status = "Adult"
elif age >= 13:
    status = "Teenager"
else:
    status = "Child"
print(f"Age {age}: {status}")

# 2. Dictionary Operations
student = {'name': 'Emma', 'course': 'Business', 'grades': [85, 90, 88]}

# Accessing and Updating
print(f"Student: {student['name']}")
student['grade_avg'] = sum(student['grades']) / len(student['grades']) # Add new
print(f"Average Grade: {student['grade_avg']:.2f}")
```

```python
# Looping through a dictionary
print("Student Details:")
for key, value in student.items():
    print(f"  - {key}: {value}")

# 3. While Loop (Counter Example)
# (Using a counter instead of input() to allow 'Run All' without blocking)
count = 3
print("Starting countdown:")
while count > 0:
    print(f"  {count}...")
    count -= 1
print("Liftoff!")
```

```
Age 20: Adult
Student: Emma
Average Grade: 87.67
Student Details:
  - name: Emma
  - course: Business
  - grades: [85, 90, 88]
  - grade_avg: 87.66666666666667
Starting countdown:
  3...
  2...
  1...
Liftoff!
```

Functions

Function Basics

Concept	Description & Syntax
Definition	Block of reusable code. Defined with `def`. `def my_func():`
Parameters	Variables passed into function. `def greet(name):`
Arguments	Values sent to function when called. `greet("Alice")`
Return	Sends a result back to the caller. `return x + y`
Docstring	Documentation string explaining the function. `"""Description"""`

Advanced Arguments

Concept	Description & Syntax
Default Args	Parameter has a default value if not provided. `def power(base, exp=2):`

Concept	Description & Syntax
Keyword Args	Arguments passed by name. `func(name="Bob", age=30)`
`*args`	Variable number of positional arguments (tuple). `def sum_all(*args):`
** `**kwargs` **	Variable number of keyword arguments (dictionary). `def config(**kwargs):`

In [3]:

```python
# 1. Basic Function with Return
def calculate_area(length, width):
    """Returns the area of a rectangle."""
    return length * width

area = calculate_area(10, 5)
print(f"Area: {area}")

# 2. Default Arguments & Keyword Arguments
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

print(greet("Alice"))              # Uses default
print(greet("Bob", greeting="Hi")) # Overrides default

# 3. *args (Variable Positional Arguments)
def sum_numbers(*args):
    """Sums any number of arguments."""
    return sum(args)

print(f"Sum: {sum_numbers(1, 2, 3, 4, 5)}")

# 4. **kwargs (Variable Keyword Arguments)
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Charlie", age=28, city="Paris")
```

```
Area: 50
Hello, Alice!
Hi, Bob!
Sum: 15
name: Charlie
age: 28
city: Paris
```

Object-Oriented Programming (OOP)

Core Concepts

Concept	Description & Syntax
Class	Blueprint for creating objects. `class Car:`

Concept	Description & Syntax
Object	Instance of a class. `my_car = Car()`
`__init__`	Constructor method, runs when object is created. `def __init__(self, make):`
`self`	Reference to the current instance. Used to access attributes/methods. `self.make = make`
Attributes	Variables belonging to an object. `self.color = "Red"`
Methods	Functions belonging to an object. `def drive(self):`
`__str__`	String representation of the object (for printing). `def __str__(self): return "Car"`

In [4]:

```python
# 1. Defining a Class
class Student:
    def __init__(self, name, student_id):
        self.name = name          # Attribute
        self.student_id = student_id
        self.grades = []          # Default attribute

    def add_grade(self, grade):   # Method
        self.grades.append(grade)

    def average_grade(self):
        if not self.grades:
            return 0
        return sum(self.grades) / len(self.grades)

    def __str__(self):            # String representation
        return f"Student: {self.name} (ID: {self.student_id})"

# 2. Creating Objects (Instances)
s1 = Student("Alice", "S001")
s2 = Student("Bob", "S002")

# 3. Using Methods
s1.add_grade(85)
s1.add_grade(90)
s2.add_grade(78)

print(s1)  # Uses __str__
print(f"Alice's Average: {s1.average_grade():.2f}")
print(f"Bob's Average: {s2.average_grade():.2f}")
```

```
Student: Alice (ID: S001)
Alice's Average: 87.50
Bob's Average: 78.00
```

Files and Exceptions

File Handling (Standard `open`)

Concept	Description & Syntax
Opening Files	Use `with open(filename, mode) as file:` to ensure proper closing.
Modes	`'r'` (read), `'w'` (write - overwrites), `'a'` (append).
Reading	`file.read()` (all), `file.readlines()` (list of lines), `for line in file:` (iterate).
Writing	`file.write(string)`

JSON Handling

Concept	Description & Syntax
Importing	`import json`
Saving (Dump)	`json.dump(data, file)` or `json.dumps(data)` (to string).
Loading (Load)	`json.load(file)` or `json.loads(string)`.

Exception Handling

Concept	Description & Syntax
Try/Except	Catches errors. `try: ... except FileNotFoundError:`
Multiple Excepts	`except (ValueError, ZeroDivisionError):`
Else	Runs if NO exception occurs.
Finally	Runs always (cleanup).

In [10]:

```python
import json

# 1. Reading from a File (with Error Handling)
try:
    # Create a dummy file first for demonstration
    with open("animals.txt", "w") as f:
        f.write("Cat\nDog\nElephant")

    with open("animals.txt", "r") as file:
        animals_list = [line.strip() for line in file]
except FileNotFoundError:
    print("The file 'animals.txt' was not found.")
else:
    print("Animals found:", animals_list)

# 2. Writing and Appending
foods = ["Pizza", "Sushi", "Tacos"]
with open("favourites.txt", "w") as file:
    for food in foods:
        file.write(food + "\n")

with open("favourites.txt", "a") as file:
    file.write("Ice Cream\n")

print("\nFavourites saved.")

# 3. JSON Storage
user_data = {"username": "Alice", "score": 95}
with open("user.json", "w") as file:
    file.write(json.dumps(user_data))

# Loading JSON
try:
    with open("user.json") as file:
        content = file.read()
        data = json.loads(content)
        print(f"\nLoaded JSON: User {data['username']} has score {data['score']}
except FileNotFoundError:
    pass

# 4. Exception Handling (ZeroDivision)
def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero.")
        return None

print("\nDivision Test:", safe_divide(10, 0))
```

```
Animals found: ['Cat', 'Dog', 'Elephant']

Favourites saved.

Loaded JSON: User Alice has score 95
Error: Cannot divide by zero.

Division Test: None
```

Comprehensive Revision Example

This example integrates key concepts from the course into a single application:

1. **Data Structures**: Lists and Dictionaries to store sales data.
2. **OOP**: A `SalesManager` class to organize logic.
3. **Control Flow**: Loops to calculate totals and find top products.
4. **File I/O**: Using standard `open` and `json` to save/load data.
5. **Visualization**: Using `matplotlib` to plot revenue.

In [9]:

```
import json
import matplotlib.pyplot as plt

class SalesManager:
    def __init__(self, filename="sales_data.json"):
        self.filename = filename
        self.sales = [] # List of dictionaries
```

```python
    def add_sale(self, product, amount, quantity):
        """Adds a sale record."""
        sale = {
            "product": product,
            "amount": amount,
            "quantity": quantity
        }
        self.sales.append(sale)

    def save_data(self):
        """Saves sales data to a JSON file."""
        with open(self.filename, 'w') as file:
            json.dump(self.sales, file, indent=4)
        print(f"Data saved to {self.filename}")

    def load_data(self):
        """Loads data from file if it exists."""
        try:
            with open(self.filename, 'r') as file:
                self.sales = json.load(file)
            print(f"Data loaded from {self.filename}")
        except FileNotFoundError:
            print("No existing data found. Starting fresh.")

    def get_total_revenue(self):
        """Calculates total revenue."""
        total = 0
        for sale in self.sales:
            total += sale['amount'] * sale['quantity']
        return total

    def get_product_summary(self):
        """Returns a dictionary of total revenue per product."""
        summary = {}
        for sale in self.sales:
            prod = sale['product']
            revenue = sale['amount'] * sale['quantity']
            if prod in summary:
                summary[prod] += revenue
            else:
                summary[prod] = revenue
        return summary

    def plot_revenue(self):
        """Plots revenue per product."""
        summary = self.get_product_summary()
        products = list(summary.keys())
        revenues = list(summary.values())

        plt.figure(figsize=(8, 5))
        plt.bar(products, revenues, color='skyblue')
        plt.title("Total Revenue by Product")
        plt.xlabel("Product")
        plt.ylabel("Revenue (£)")
        plt.show()

# --- Main Execution ---
if __name__ == '__main__':
    print("--- Running Sales Application ---")
    app = SalesManager("my_shop_sales.json")

    # Try loading existing data
    app.load_data()

    # Add some sample data
    app.add_sale("Coffee", 3.50, 10)
    app.add_sale("Tea", 2.50, 15)
    app.add_sale("Cake", 4.00, 5)

    # Calculate and print revenue
    print(f"Total Revenue: £{app.get_total_revenue():.2f}")

    # Save data to file
    app.save_data()

    # Uncomment the line below to see the plot
    # app.plot_revenue()
```

```
--- Running Sales Application ---
No existing data found. Starting fresh.
Total Revenue: £92.50
Data saved to my_shop_sales.json
```