Python Programming – Lecture Note

Chapter 1: Introduction to Python

Overview

- Python: High-level, interpreted, and dynamically-typed programming language created by Guido van Rossum in 1991.
- Key Features:
  - Easy to learn and use.
  - Free and open-source.
  - Supports multiple programming paradigms.
  - Large standard library.

Applications

- System scripting, GUI development, web applications, data science, machine learning, and more.

Programming Paradigms

1. Procedural: Step-by-step instructions (`if`, `for`, `while`).
2. Object-Oriented: Real-world modeling using `classes` and `objects`.
3. Functional: Use of higher-order functions like `map`, `filter`.
4. Event-Driven: Ideal for GUI-based applications.

Chapter 2: Getting Started

Installation

1. Windows: Download from python.org and ensure "Add Python to PATH" is checked.
2. Linux: Install using:
3. `sudo apt-get install python3`
4. Verify Installation:
5. `python --version`

Python Modes

1. Interactive Mode:
2. `>>> print("Hello, Python!")`
3. `Hello, Python!`
4. Script Mode: Save the code in a `.py` file and run:
5. `python script_name.py`

Development Tools

- IDEs: PyCharm, VS Code, IDLE.
- Online Platforms: Google Colab, Jupyter Notebook.

Chapter 3: Python Basics

Identifiers and Keywords

- Identifiers: Names for variables, functions, etc.
  - Start with a letter or _.
  - Cannot use reserved keywords (`if`, `else`, `for`).
- Keywords can be listed using:
- `import keyword`
- `print(keyword.kwlist)`

Data Types

1. Basic Types: `int`, `float`, `bool`, `str`.
2. Container Types: `list`, `tuple`, `set`, `dict`.

Operators

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

Examples

```
x, y = 10, 3
print(x + y)  # 13
print(x // y) # 3
```

Chapter 4: Strings

Overview

- Strings are immutable sequences of characters.
- Can be enclosed in single (') , double (") or triple (''' or """) quotes.

String Operations

- Concatenation:
- print("Hello" + "World") # HelloWorld
- Slicing:
- msg = "Python"
- print(msg[:3]) # Pyt
- Repetition:
- print("Hi" * 3) # HiHiHi

String Methods

- upper(), lower(), find(), replace(), split().

```
s = "Python"
print(s.upper())       # PYTHON
print(s.replace("Py", "Cy")) # Cyton
```

Chapter 5: Decision Control Instructions

Conditionals

- Syntax:
- `if condition:`
- `    # block`
- `elif another_condition:`
- `    # block`
- `else:`
- `    # block`
- Example:
- `x = 10`
- `if x > 0:`
- `    print("Positive")`
- `else:`
- `    print("Negative")`

Chapter 6: Repetition Control Instructions

Loops

1. `for` Loop:
2. `for i in range(5):`
3. `    print(i)`
4. `while` Loop:
5. `n = 3`
6. `while n > 0:`
7. `    print(n)`
8. `    n -= 1`

Loop Control

- `break`: Exit the loop early.
- `continue`: Skip the current iteration.

Chapter 7: Console Input/Output

Input and Output

```
name = input("Enter your name: ")
print(f"Hello, {name}")
```

Chapter 8: Lists

Overview

- Definition: A list is an ordered, mutable collection of elements that can hold items of different data types.
- Syntax:
- `lst = [1, 2, 3, "Python", True]`

Key Features

1. Dynamic: Can grow or shrink in size.
2. Indexed: Access elements using indices.
3. Mutable: Elements can be modified.

List Operations

1. Accessing Elements:

```
lst = [10, 20, 30]
print(lst[0])   # Output: 10
print(lst[-1])  # Output: 30
```

5. Modifying Elements:

```
lst[1] = 25
print(lst)  # Output: [10, 25, 30]
```

8. Adding Elements:

```
lst.append(40)        # Adds at the end
lst.insert(1, 15)     # Inserts at index 1
```

11. Removing Elements:

```
lst.pop()             # Removes last element
lst.remove(10)        # Removes first occurrence of 10
del lst[0]            # Deletes element at index 0
```

15. List Slicing:

```
lst = [1, 2, 3, 4, 5]
print(lst[1:4])  # Output: [2, 3, 4]
```

Common List Methods

- Examples:
- `lst = [1, 3, 2]`
- `lst.sort()      # Output: [1, 2, 3]`
- `lst.reverse()   # Output: [3, 2, 1]`
- `print(len(lst)) # Output: 3`

Chapter 9: Tuples

Overview

- Definition: A tuple is an ordered, immutable collection of elements.
- Syntax:
- `tpl = (1, 2, 3)`

Key Features

1. Immutable: Elements cannot be modified.
2. Indexed: Access elements using indices.
3. Allows Duplicates: Duplicate elements are permitted.

Tuple Operations

1. Accessing Elements:
2. `tpl = (10, 20, 30)`
3. `print(tpl[0])   # Output: 10`
4. `print(tpl[-1])  # Output: 30`
5. Slicing:
6. `tpl = (1, 2, 3, 4, 5)`
7. `print(tpl[1:4])  # Output: (2, 3, 4)`

Tuple Methods

- Examples:
- `tpl = (1, 2, 3, 2)`
- `print(tpl.count(2))  # Output: 2`
- `print(tpl.index(3))  # Output: 2`

Chapter 10: Sets

Overview

- Definition: A set is an unordered collection of unique elements.
- Syntax:
- `s = {1, 2, 3}`

Key Features

1. Unordered: No indexing.
2. Unique Elements: Duplicates are automatically removed.
3. Mutable: Elements can be added or removed.

Set Operations

1. Adding Elements:
2. `s.add(4)`
3. `print(s)  # Output: {1, 2, 3, 4}`
4. Removing Elements:
5. `s.remove(2)`
6. `print(s)  # Output: {1, 3, 4}`
7. Set Operations:
    - Union:
    - `a = {1, 2, 3}`
    - `b = {3, 4, 5}`
    - `print(a | b)  # Output: {1, 2, 3, 4, 5}`
    - Intersection:
    - `print(a & b)  # Output: {3}`
8. Membership Test:
9. `print(2 in s)  # Output: True`

Chapter 11: Dictionaries

Overview

- Definition: A dictionary is an unordered collection of key-value pairs.
- Syntax:
- `d = {"name": "Alice", "age": 25}`

Key Features

1. Key-Value Pairs: Accessed using keys, not indices.
2. Unique Keys: Keys must be unique.

Dictionary Operations

1. Accessing Values:
2. `print(d["name"])  # Output: Alice`
3. Adding/Updating:
4. `d["city"] = "New York"  # Add a new key-value pair`
5. `d["age"] = 26           # Update value`
6. Removing:
7. `del d["city"]           # Remove a key-value pair`
8. Iterating:
9. `for key, value in d.items():`
10. `    print(key, value)`

Chapter 12: Comprehensions

List Comprehension

- Definition: A concise way to create lists.
- Syntax:
- `[expression for item in iterable if condition]`
- Example:
- `squares = [x**2 for x in range(5)]`
- `print(squares)  # Output: [0, 1, 4, 9, 16]`

Set Comprehension

- Example:
- `s = {x**2 for x in range(5)}`
- `print(s)  # Output: {0, 1, 4, 9, 16}`

Dictionary Comprehension

- Example:
- `d = {x: x**2 for x in range(5)}`
- `print(d)  # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}`

Chapter 13: Functions

Overview

- Definition: A reusable block of code that performs a specific task.
- Syntax:
- def function_name(parameters):
-     # body
-     return value

Types of Functions

1. User-Defined Functions:
2. def greet(name):
3.   return f"Hello, {name}"
4.
5. print(greet("Alice")) # Output: Hello, Alice
6. Lambda Functions:
7. square = lambda x: x**2
8. print(square(5)) # Output: 25

Function Parameters

1. Positional Parameters:
2. def add(a, b):
3.   return a + b
4.
5. print(add(3, 5)) # Output: 8
6. Default Parameters:
7. def greet(name="World"):
8.   return f"Hello, {name}"
9.
10. print(greet()) # Output: Hello, World
11. Arbitrary Arguments:
12. def sum_all(*args):
13.   return sum(args)
14.
15. print(sum_all(1, 2, 3, 4)) # Output: 10
16. Keyword Arguments:
17. def greet(**kwargs):
18.   return f"Hello, {kwargs['name']}"
19.
20. print(greet(name="Alice")) # Output: Hello, Alice

Chapter 14: Recursion

Overview

- Definition: A function calling itself to solve smaller subproblems of the original problem.
- Base Case: The termination condition to avoid infinite recursion.
- Recursive Case: The part where the function calls itself with modified arguments.

Key Characteristics

1. A problem is divided into smaller subproblems.
2. Recursive calls must converge toward a base case.

Examples

Factorial

```
def factorial(n):
    if n == 0:
        return 1  # Base case
    return n * factorial(n - 1)  # Recursive case

print(factorial(5))  # Output: 120
```

Fibonacci Sequence

```
def fibonacci(n):
    if n <= 1:
        return n  # Base case
    return fibonacci(n - 1) + fibonacci(n - 2)  # Recursive case

print(fibonacci(6))  # Output: 8
```

Sum of a List

```
def sum_list(lst):
    if not lst:  # Base case: empty list
        return 0
    return lst[0] + sum_list(lst[1:])  # Recursive case

print(sum_list([1, 2, 3, 4, 5]))  # Output: 15
```

Chapter 15: Functional Programming

Overview

- Functional Programming:
  - Focuses on immutability, pure functions, and declarative style.
  - Common concepts: Higher-order functions, closures, and lambdas.
- Pure Functions: Functions with no side effects whose output depends solely on input.

Higher-Order Functions

- Functions that take other functions as arguments or return functions as results.

Key Built-in Functions

1. `map()`:
   - Applies a function to all elements in an iterable.

```
nums = [1, 2, 3]
squares = list(map(lambda x: x**2, nums))
print(squares)  # Output: [1, 4, 9]
```

5. `filter()`:
   - Filters elements from an iterable based on a condition.

```
nums = [1, 2, 3, 4]
evens = list(filter(lambda x: x % 2 == 0, nums))
print(evens)  # Output: [2, 4]
```

9. `reduce()`:
   - Combines elements from an iterable into a single value.

```
from functools import reduce

nums = [1, 2, 3, 4]
total = reduce(lambda x, y: x + y, nums)
print(total)  # Output: 10
```

Lambda Functions

- Anonymous, inline functions defined using the `lambda` keyword.
- `square = lambda x: x**2`
- `print(square(5))  # Output: 25`

Examples

Sort with Key

```
students = [("Alice", 85), ("Bob", 75), ("Charlie", 95)]
sorted_students = sorted(students, key=lambda x: x[1])
print(sorted_students)  # Output: [('Bob', 75), ('Alice', 85), ('Charlie', 95)]
```

Functional Approach to Summation

```
nums = [1, 2, 3, 4, 5]
sum_of_squares = sum(map(lambda x: x**2, nums))
print(sum_of_squares)  # Output: 55
```

Chapter 16: Modules and Packages

Modules

- Definition: A file containing Python code (functions, classes, variables) that can be imported and reused.
- Advantages:
  - Code reusability.
  - Better organization.
  - Separation of concerns.

Creating a Module

1. Create a file named `mymodule.py`:

```
# mymodule.py
def greet(name):
    return f"Hello, {name}"
```

2. Import and use the module in another script:

```
# main.py
import mymodule
print(mymodule.greet("Alice"))  # Output: Hello, Alice
```

Built-in Modules

- Python includes several built-in modules like `math`, `os`, `random`.

```
import math

print(math.sqrt(16))  # Output: 4.0
print(math.pi)        # Output: 3.141592653589793
```

Packages

- Definition: A package is a directory that contains multiple modules and an `__init__.py` file.
- Advantages:
  - Hierarchical organization of modules.
  - Encourages modularity in large projects.

Creating a Package

1. Create a directory named `mypackage`:

```
mypackage/
├── __init__.py
├── module1.py
├── module2.py
```

2. Define modules within the package:

```
# module1.py
def add(a, b):
    return a + b
# module2.py
def multiply(a, b):
    return a * b
```

Use the package:

```
from mypackage.module1 import add
from mypackage.module2 import multiply

print(add(2, 3))       # Output: 5
print(multiply(2, 3))  # Output: 6
```

Using `__init__.py`

- The `__init__.py` file initializes the package and can contain package-level variables or imports.
- `# __init__.py`
- `from .module1 import add`
- `from .module2 import multiply`

Installing and Using External Packages

- Use `pip` to install third-party packages:
- `pip install numpy`
- Example of using an installed package:
- `import numpy as np`
-
- `arr = np.array([1, 2, 3, 4])`
- `print(arr.mean())  # Output: 2.5`

Chapter 17: Namespaces

Overview

- A namespace is a mapping between variable names and their corresponding objects.
- Python has three main namespaces:
    1. Local: Inside a function.
    2. Global: At the module level.
    3. Built-in: Python's predefined functions and variables.

Scope Example

```
x = 10  # Global

def foo():
    x = 5  # Local
    print(x)  # 5

foo()
print(x)  # 10
```

Chapter 18: Classes and Objects

Classes

- Definition: A blueprint for creating objects.
- Example:
- ```
  class Person:
      def __init__(self, name, age):
          self.name = name
          self.age = age

      def greet(self):
          return f"Hello, my name is {self.name}."

  p = Person("Alice", 30)
  print(p.greet())  # Hello, my name is Alice.
  ```

  ```

Chapter 19: Intricacies of Classes and Objects

Special Methods

- `__str__`: String representation of an object.
- `class Person:`
- `    def __init__(self, name):`
- `        self.name = name`
-
- `    def __str__(self):`
- `        return f"Person({self.name})"`
-
- `print(Person("Alice"))  # Person(Alice)`

Encapsulation

- Protecting access to class variables and methods using _ and __.
- `class BankAccount:`
- `    def __init__(self, balance):`
- `        self.__balance = balance  # Private variable`
-
- `    def get_balance(self):`
- `        return self.__balance`

Chapter 20: Containership and Inheritance

Inheritance

- Allows a class to inherit attributes and methods from another class.
- ```
  class Animal:
      def speak(self):
          return "I make sounds."
  
  class Dog(Animal):
      def speak(self):
          return "Woof!"
  
  d = Dog()
  print(d.speak())  # Woof!
  ```

  ```

Chapter 21: Iterators and Generators

Iterators

- An object implementing `__iter__()` and `__next__()`.
- ```
  class MyIterator:
      def __init__(self, max):
          self.max = max
          self.current = 0
  
      def __iter__(self):
          return self
  
      def __next__(self):
          if self.current < self.max:
              self.current += 1
              return self.current
          else:
              raise StopIteration
  
  for num in MyIterator(3):
      print(num)  # 1 2 3
  ```

Generators

- Use the `yield` keyword to lazily produce values.
- ```
  def generate_numbers(n):
      for i in range(n):
          yield i
  
  for num in generate_numbers(3):
      print(num)  # 0 1 2
  ```

Chapter 22: Exception Handling

Try-Except Block

```
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error: {e}")
else:
    print("No exceptions occurred.")
finally:
    print("This block always executes.")
```

Chapter 23: File Input/Output

File Operations

- Writing to a File:
- with open("example.txt", "w") as file:
-     file.write("Hello, File!")
- Reading from a File:
- with open("example.txt", "r") as file:
-     content = file.read()
-     print(content) # Hello, File!

Chapter 24: Miscellany

Regular Expressions

```
import re
match = re.search(r'\d+', "Age: 25")
print(match.group())  # 25
```

Date and Time

```
from datetime import datetime
now = datetime.now()
print(now.strftime("%Y-%m-%d %H:%M:%S"))
```

Chapter 25: Concurrency and Parallelism

Threads

```
import threading

def print_numbers():
    for i in range(5):
        print(i)

t1 = threading.Thread(target=print_numbers)
t1.start()
```

Chapter 26: Synchronization

Using Locks

```
import threading

lock = threading.Lock()

def critical_section():
    with lock:
        print("Accessing shared resource.")

t1 = threading.Thread(target=critical_section)
t1.start()
```
