Tutorial by Dr. Ayes Chinmay
Date: 15/09/2025

Chapter 1: Review of the Fundamentals - Complete Tutorial

Table of Contents

1.1 Python Quick Start

Overview

Python is an interpreted, high-level programming language that emphasizes code readability and simplicity. A quick start involves understanding the Python environment and how to execute basic commands.

Key Concepts

Python supports interactive and scripted programming. The simplest way to start is using the Python interpreter or an IDE like IDLE, PyCharm, or Jupyter Notebook.

Code Example 1: Your First Python Program

```
print("Hello, World!")
```

Output:

```
Hello, World!
```

Code Example 2: Basic Arithmetic

```
print(5 + 3)
print(10 - 4)
print(6 * 7)
print(20 / 4)
```

Output:

```
8
6
42
5.0
```

Code Example 3: Using Variables

```
x = 10
y = 20
print("x =", x)
print("y =", y)
print("x + y =", x + y)
```

Output:

```
x = 10
y = 20
x + y = 30
```

1.2 Variables and Naming

Overview

Variables are containers for storing data values. Python uses dynamic typing, meaning you don't need to declare the type of a variable explicitly.

Naming Rules

- Variable names must start with a letter or underscore (_)
- Names can contain letters, numbers, and underscores
- Names are case-sensitive
- Avoid using Python reserved words
- Use descriptive names (snake_case convention)

Code Example 1: Valid Variable Names

```
first_name = "John"
age = 25
_private_var = 100
userName = "john_doe"
print(first_name, age, _private_var, userName)
```

Output:

```
John 25 100 john_doe
```

Code Example 2: Variable Type Discovery

```
name = "Alice"
age = 30
height = 5.6
is_student = True

print(type(name))
print(type(age))
print(type(height))
print(type(is_student))
```

Output:

```
&lt;class 'str'&gt;
&lt;class 'int'&gt;
&lt;class 'float'&gt;
&lt;class 'bool'&gt;
```

Code Example 3: Multiple Assignment

```
x, y, z = 1, 2, 3
print(x, y, z)

a = b = c = 10
print(a, b, c)
```

Output:

```
1 2 3
10 10 10
```

1.3 Combined Assignment Operators

Overview

Combined assignment operators perform an operation and assign the result in a single step.

Common Combined Assignment Operators

Operator	Example	Equivalent
+=	x += 5	x = x + 5
-=	x -= 3	x = x - 3
*=	x *= 2	x = x * 2
/=	x /= 4	x = x / 4
//=	x //= 3	x = x // 3
%=	x %= 5	x = x % 5

Operator	Example	Equivalent
**=	x **= 2	x = x ** 2

Code Example 1: Addition Assignment

```
x = 10
x += 5
print("After += 5:", x)

x -= 3
print("After -= 3:", x)
```

Output:

```
After += 5: 15
After -= 3: 12
```

Code Example 2: Multiplication and Division Assignment

```
num = 20
num *= 2
print("After *= 2:", num)

num /= 4
print("After /= 4:", num)
```

Output:

```
After *= 2: 40
After /= 4: 10.0
```

Code Example 3: Modulus and Exponentiation

```
x = 17
x %= 5
print("After %= 5:", x)

y = 2
y **= 3
print("After **= 3:", y)
```

Output:

```
After %= 5: 2
After **= 3: 8
```

1.4 Summary of Python Arithmetic Operators

Overview

Arithmetic operators perform mathematical operations on numbers.

Arithmetic Operators Table

Operator	Name	Example	Result
+	Addition	10 + 3	13
-	Subtraction	10 - 3	7
*	Multiplication	10 * 3	30
/	Division	10 / 3	3.333...
//	Floor Division	10 // 3	3
%	Modulus	10 % 3	1
**	Exponentiation	2 ** 3	8

Code Example 1: Basic Arithmetic Operations

```
a = 15
b = 4

print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
print("Floor Division:", a // b)
print("Modulus:", a % b)
print("Exponentiation:", a ** b)
```

Output:

```
Addition: 19
Subtraction: 11
Multiplication: 60
Division: 3.75
Floor Division: 3
Modulus: 3
Exponentiation: 50625
```

Code Example 2: Order of Operations

```
result1 = 10 + 5 * 2
result2 = (10 + 5) * 2
result3 = 2 ** 3 * 2

print("10 + 5 * 2 =", result1)
print("(10 + 5) * 2 =", result2)
print("2 ** 3 * 2 =", result3)
```

Output:

```
10 + 5 * 2 = 20
(10 + 5) * 2 = 30
2 ** 3 * 2 = 16
```

1.5 Elementary Data Types: Integer and Floating Point

Overview

Python supports numeric data types: integers (whole numbers) and floats (decimal numbers).

Code Example 1: Integer Operations

```
x = 10
y = 3
z = -5

print("x:", x, "Type:", type(x))
print("y:", y, "Type:", type(y))
print("z:", z, "Type:", type(z))
print("x + y =", x + y)
print("x - y =", x - y)
print("x * y =", x * y)
```

Output:

```
x: 10 Type: &lt;class 'int'&gt;
y: 3 Type: &lt;class 'int'&gt;
z: -5 Type: &lt;class 'int'&gt;
x + y = 13
x - y = 7
x * y = 30
```

Code Example 2: Floating Point Numbers

```
a = 3.14
b = 2.71
c = -1.5

print("a:", a, "Type:", type(a))
print("b:", b, "Type:", type(b))
print("c:", c, "Type:", type(c))
print("a + b =", a + b)
print("a * b =", a * b)
print("a / b =", a / b)
```

Output:

```
a: 3.14 Type: &lt;class 'float'&gt;
b: 2.71 Type: &lt;class 'float'&gt;
c: -1.5 Type: &lt;class 'float'&gt;
a + b = 5.85
a * b = 8.4994
a / b = 1.1589854...
```

Code Example 3: Type Conversion

```
int_num = 10
float_num = 10.5

converted_to_float = float(int_num)
converted_to_int = int(float_num)

print("Original int:", int_num, "Converted to float:", converted_to_float)
print("Original float:", float_num, "Converted to int:", converted_to_int)
```

```
print(type(converted_to_float))
print(type(converted_to_int))
```

Output:

```
Original int: 10 Converted to float: 10.0
Original float: 10.5 Converted to int: 10
&lt;class 'float'&gt;
&lt;class 'int'&gt;
```

1.6 Basic Input and Output

Overview

Input/Output operations allow programs to interact with users. Python provides built-in functions for reading input and displaying output.

Code Example 1: Output with print()

```
print("Hello, Python!")
print("Number:", 42)
print("Float:", 3.14)
print(1, 2, 3, sep='-', end='!\n')
```

Output:

```
Hello, Python!
Number: 42
Float: 3.14
1-2-3!
```

Code Example 2: Input with input()

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

age = int(input("Enter your age: "))
print("Next year you will be", age + 1)
```

Output (with input "John" and "25"):

```
Enter your name: John
Hello, John
Enter your age: 25
Next year you will be 26
```

Code Example 3: Multiple Inputs

```
first = input("Enter first number: ")
second = input("Enter second number: ")

sum_result = int(first) + int(second)
print("Sum:", sum_result)
```

Output (with inputs "10" and "20"):

```
Enter first number: 10
Enter second number: 20
Sum: 30
```

1.7 Function Definitions

Overview

Functions are reusable blocks of code that perform specific tasks. They help organize code and reduce repetition.

Function Syntax

```
def function_name(parameters):
    """Docstring describing the function"""
    # Function body
    return value
```

Code Example 1: Simple Function

```
def greet():
    print("Hello, World!")

greet()
greet()
```

Output:

```
Hello, World!
Hello, World!
```

Code Example 2: Function with Parameters

```
def add(a, b):
    return a + b

result = add(5, 3)
print("5 + 3 =", result)

result = add(10, 20)
print("10 + 20 =", result)
```

Output:

```
5 + 3 = 8
10 + 20 = 30
```

Code Example 3: Function with Default Parameters

```
def greet_user(name, greeting="Hello"):
    print(greeting + ", " + name)

greet_user("Alice")
greet_user("Bob", "Hi")
```

Output:

```
Hello, Alice
Hi, Bob
```

Code Example 4: Multiple Return Values

```
def get_min_max(numbers):
    return min(numbers), max(numbers)

data = [5, 2, 8, 1, 9, 3]
minimum, maximum = get_min_max(data)
print("Min:", minimum, "Max:", maximum)
```

Output:

```
Min: 1 Max: 9
```

1.8 The Python "if" Statement

Overview

Conditional statements allow programs to execute different code based on conditions.

Code Example 1: Simple if Statement

```
age = 18
if age &gt;= 18:
    print("You are an adult")
```

Output:

```
You are an adult
```

Code Example 2: if-else Statement

```
temperature = 15

if temperature &gt; 25:
    print("It's hot")
else:
    print("It's cold")
```

Output:

```
It's cold
```

Code Example 3: if-elif-else Statement

```
score = 75

if score &gt;= 90:
    print("Grade: A")
elif score &gt;= 80:
    print("Grade: B")
elif score &gt;= 70:
```

```
    print("Grade: C")
else:
    print("Grade: F")
```

Output:

```
Grade: C
```

Code Example 4: Nested if Statements

```
age = 25
has_license = True

if age &gt;= 18:
    if has_license:
        print("You can drive")
    else:
        print("You need a license to drive")
else:
    print("You are too young to drive")
```

Output:

```
You can drive
```

1.9 The Python "while" Statement

Overview

The while loop repeatedly executes code as long as a condition is true.

Code Example 1: Basic while Loop

```
count = 1
while count &lt;= 5:
    print(count)
    count += 1
```

Output:

```
1
2
3
4
5
```

Code Example 2: while Loop with Condition

```
num = 10
while num &gt; 0:
    print(num)
    num -= 2
```

Output:

```
10
8
6
4
2
```

Code Example 3: break Statement

```
count = 0
while True:
    count += 1
    if count == 5:
        break
    print(count)
```

Output:

```
1
2
3
4
```

Code Example 4: continue Statement

```
count = 0
while count &amp;lt; 5:
    count += 1
    if count == 3:
        continue
    print(count)
```

Output:

```
1
2
4
5
```

1.10 A Couple of Cool Little Apps

Overview

This section demonstrates practical applications combining concepts learned so far.

Code Example 1: Simple Calculator

```
def calculator():
    print("Simple Calculator")
    num1 = float(input("Enter first number: "))
    operator = input("Enter operator (+, -, *, /): ")
    num2 = float(input("Enter second number: "))

    if operator == '+':
        print("Result:", num1 + num2)
    elif operator == '-':
        print("Result:", num1 - num2)
    elif operator == '*':
```

```
        print("Result:", num1 * num2)
    elif operator == '/':
        if num2 != 0:
            print("Result:", num1 / num2)
        else:
            print("Error: Division by zero")

calculator()
```

Output (with inputs 10, '+', 5):

```
Simple Calculator
Enter first number: 10
Enter operator (+, -, *, /): +
Enter second number: 5
Result: 15.0
```

Code Example 2: Countdown Timer

```
def countdown(n):
    while n &gt; 0:
        print(n)
        n -= 1
    print("Blastoff!")

countdown(3)
```

Output:

```
3
2
1
Blastoff!
```

1.11 Summary of Python Boolean Operators

Overview

Boolean operators combine multiple conditions to form complex logical expressions.

Boolean Operators Table

Operator	Meaning	Example	Result
and	Both conditions true	True and False	False
or	At least one true	True or False	True
not	Reverses the boolean	not True	False

Code Example 1: and Operator

```
x = 10
if x &gt; 5 and x &lt; 15:
    print("x is between 5 and 15")
```

Output:

```
x is between 5 and 15
```

Code Example 2: or Operator

```
day = "Saturday"
if day == "Saturday" or day == "Sunday":
    print("It's a weekend")
```

Output:

```
It's a weekend
```

Code Example 3: not Operator

```
is_raining = False
if not is_raining:
    print("Let's go outside")
```

Output:

```
Let's go outside
```

Code Example 4: Combined Boolean Operators

```
age = 25
has_license = True

if (age &gt;= 18) and (has_license):
    print("Can rent a car")
```

Output:

```
Can rent a car
```

1.12 Function Arguments and Return Values

Overview

Functions can accept multiple arguments and return values to communicate with calling code.

Code Example 1: Positional Arguments

```
def introduce(name, age, city):
    print(f"My name is {name}, I'm {age} years old, from {city}")

introduce("Alice", 30, "New York")
introduce("Bob", 25, "Los Angeles")
```

Output:

```
My name is Alice, I'm 30 years old, from New York
My name is Bob, I'm 25 years old, from Los Angeles
```

Code Example 2: Keyword Arguments

```
def display_info(name, age, city="Unknown"):
    print(f"Name: {name}, Age: {age}, City: {city}")

display_info(name="Charlie", age=35)
display_info(name="Diana", age=28, city="Chicago")
```

Output:

```
Name: Charlie, Age: 35, City: Unknown
Name: Diana, Age: 28, City: Chicago
```

Code Example 3: Variable Length Arguments

```
def sum_numbers(*args):
    total = sum(args)
    return total

print(sum_numbers(1, 2, 3))
print(sum_numbers(5, 10, 15, 20))
```

Output:

```
6
50
```

Code Example 4: Return Multiple Values

```
def calculate(a, b):
    sum_val = a + b
    product = a * b
    return sum_val, product

s, p = calculate(5, 3)
print("Sum:", s, "Product:", p)
```

Output:

```
Sum: 8 Product: 15
```

1.13 The Forward Reference Problem

Overview

The forward reference problem occurs when a function calls another function that is defined later in the code.

Code Example 1: Forward Reference Issue

```
def func_a():
    print("In func_a")
    func_b()

def func_b():
    print("In func_b")
```

```
func_a()
```

Output:

```
In func_a
In func_b
```

Code Example 2: Avoiding Forward Reference

```
def greet(name):
    return f"Hello, {name}"

def main():
    message = greet("Alice")
    print(message)

main()
```

Output:

```
Hello, Alice
```

Code Example 3: Function Definitions Before Calls

```
def helper():
    return "Helper function"

def main():
    result = helper()
    print(result)

main()
```

Output:

```
Helper function
```

1.14 Python Strings

Overview

Strings are sequences of characters used to represent text data.

Code Example 1: String Creation and Basic Operations

```
str1 = "Hello"
str2 = "World"

print(str1)
print(str1 + " " + str2)
print(str1 * 3)
print(len(str1))
```

Output:

```
Hello
Hello World
HelloHelloHello
5
```

Code Example 2: String Indexing and Slicing

```
text = "Python"

print(text[^0])
print(text[-1])
print(text[1:4])
print(text[:3])
print(text[3:])
```

Output:

```
P
n
yth
Pyt
hon
```

Code Example 3: String Methods

```
text = "hello world"

print(text.upper())
print(text.capitalize())
print(text.replace("world", "Python"))
print(text.split())
```

Output:

```
HELLO WORLD
Hello world
hello Python
['hello', 'world']
```

Code Example 4: String Formatting

```
name = "Alice"
age = 30

print(f"My name is {name} and I'm {age}")
print("My name is {} and I'm {}".format(name, age))
print("My name is %s and I'm %d" % (name, age))
```

Output:

```
My name is Alice and I'm 30
My name is Alice and I'm 30
My name is Alice and I'm 30
```

1.15 Python Lists (and a Cool Sorting App)

Overview

Lists are ordered, mutable collections of items. They are one of the most useful data structures in Python.

Code Example 1: Creating and Accessing Lists

```
fruits = ["apple", "banana", "orange", "grape"]

print(fruits)
print(fruits[^0])
print(fruits[-1])
print(len(fruits))
```

Output:

```
['apple', 'banana', 'orange', 'grape']
apple
grape
4
```

Code Example 2: List Modification

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

numbers.append(6)
print("After append:", numbers)

numbers.insert(0, 0)
print("After insert:", numbers)

numbers.remove(3)
print("After remove:", numbers)
```

Output:

```
After append: [1, 2, 3, 4, 5, 6]
After insert: [0, 1, 2, 3, 4, 5, 6]
After remove: [0, 1, 2, 4, 5, 6]
```

Code Example 3: List Slicing and Iteration

```
items = [10, 20, 30, 40, 50]

print(items[1:4])
print(items[:3])

for item in items:
    print(item, end=" ")
```

Output:

```
[20, 30, 40]
[10, 20, 30]
10 20 30 40 50
```

Code Example 4: Cool Sorting App

```
def sort_numbers():
    numbers = []
    n = int(input("How many numbers? "))

    for i in range(n):
        num = int(input(f"Enter number {i+1}: "))
        numbers.append(num)

    print("Original list:", numbers)
    numbers.sort()
    print("Sorted list:", numbers)

sort_numbers()
```

Output (with inputs 3, 5, 2, 8):

```
How many numbers? 3
Enter number 1: 5
Enter number 2: 2
Enter number 3: 8
Original list: [5, 2, 8]
Sorted list: [2, 5, 8]
```

1.16 The "for" Statement and Ranges

Overview

The for loop iterates over sequences. The range() function generates sequences of numbers.

Code Example 1: for Loop with range()

```
for i in range(5):
    print(i)
```

Output:

```
0
1
2
3
4
```

Code Example 2: range() with Start and Stop

```
for i in range(2, 7):
    print(i, end=" ")
```

Output:

```
2 3 4 5 6
```

Code Example 3: range() with Step

```
for i in range(0, 10, 2):
    print(i, end=" ")
```

Output:

```
0 2 4 6 8
```

Code Example 4: for Loop with Lists

```
colors = ["red", "green", "blue"]

for color in colors:
    print(color)
```

Output:

```
red
green
blue
```

Code Example 5: Nested for Loops

```
for i in range(3):
    for j in range(2):
        print(f"({i}, {j})", end=" ")
    print()
```

Output:

```
(0, 0) (0, 1)
(1, 0) (1, 1)
(2, 0) (2, 1)
```

1.17 Tuples

Overview

Tuples are immutable sequences similar to lists but cannot be modified after creation.

Code Example 1: Creating Tuples

```
tuple1 = (1, 2, 3)
tuple2 = ("a", "b", "c")
tuple3 = (1, "hello", 3.14)

print(tuple1)
print(tuple2)
print(tuple3)
```

Output:

```
(1, 2, 3)
('a', 'b', 'c')
```

```
(1, 'hello', 3.14)
```

Code Example 2: Tuple Indexing and Slicing

```
colors = ("red", "green", "blue", "yellow")

print(colors[^0])
print(colors[1:3])
print(len(colors))
```

Output:

```
red
('green', 'blue')
4
```

Code Example 3: Tuple Unpacking

```
coordinates = (10, 20, 30)
x, y, z = coordinates

print(f"x={x}, y={y}, z={z}")
```

Output:

```
x=10, y=20, z=30
```

Code Example 4: Immutability of Tuples

```
tuple_data = (1, 2, 3)

try:
    tuple_data[^0] = 10
except TypeError:
    print("Cannot modify tuple - immutable!")
```

Output:

```
Cannot modify tuple - immutable!
```

1.18 Dictionaries

Overview

Dictionaries are unordered collections of key-value pairs. Keys are unique identifiers for accessing values.

Code Example 1: Creating Dictionaries

```
student = {"name": "Alice", "age": 25, "city": "NYC"}

print(student)
print(student["name"])
print(student["age"])
```

Output:

```
{'name': 'Alice', 'age': 25, 'city': 'NYC'}
Alice
25
```

Code Example 2: Dictionary Methods

```
student = {"name": "Bob", "age": 30}

student["age"] = 31
student["grade"] = "A"

print(student)
print(student.keys())
print(student.values())
```

Output:

```
{'name': 'Bob', 'age': 31, 'grade': 'A'}
dict_keys(['name', 'age', 'grade'])
dict_values(['Bob', 31, 'A'])
```

Code Example 3: Iterating Over Dictionaries

```
person = {"name": "Charlie", "age": 28, "job": "Engineer"}

for key, value in person.items():
    print(f"{key}: {value}")
```

Output:

```
name: Charlie
age: 28
job: Engineer
```

Code Example 4: Dictionary with Multiple Data Types

```
company = {
    "name": "TechCorp",
    "employees": 100,
    "founded": 2010,
    "active": True
}

print(company)
```

Output:

```
{'name': 'TechCorp', 'employees': 100, 'founded': 2010, 'active': True}
```

1.19 Sets

Overview

Sets are unordered collections of unique elements. They are useful for removing duplicates and performing set operations.

Code Example 1: Creating Sets

```
set1 = {1, 2, 3, 4, 5}
set2 = {"apple", "banana", "orange"}

print(set1)
print(set2)
print(type(set1))
```

Output:

```
{1, 2, 3, 4, 5}
{'apple', 'banana', 'orange'}
&lt;class 'set'&gt;
```

Code Example 2: Adding and Removing Elements

```
numbers = {1, 2, 3}

numbers.add(4)
print("After add:", numbers)

numbers.remove(2)
print("After remove:", numbers)
```

Output:

```
After add: {1, 2, 3, 4}
After remove: {1, 3, 4}
```

Code Example 3: Set Operations

```
set_a = {1, 2, 3, 4}
set_b = {3, 4, 5, 6}

print("Union:", set_a | set_b)
print("Intersection:", set_a &amp;amp; set_b)
print("Difference:", set_a - set_b)
```

Output:

```
Union: {1, 2, 3, 4, 5, 6}
Intersection: {3, 4}
Difference: {1, 2}
```

Code Example 4: Removing Duplicates

```
numbers = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]

unique = set(numbers)
```

```
print("Original:", numbers)
print("Unique:", unique)
print("Sorted unique:", sorted(unique))
```

Output:

```
Original: [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
Unique: {1, 2, 3, 4}
Sorted unique: [1, 2, 3, 4]
```

1.20 Global and Local Variables

Overview

Variable scope determines where a variable can be accessed. Global variables are accessible everywhere; local variables are confined to functions.

Code Example 1: Global Variables

```
global_var = 100

def display():
    print("Global var:", global_var)

display()
```

Output:

```
Global var: 100
```

Code Example 2: Local Variables

```
def my_function():
    local_var = 50
    print("Local var:", local_var)

my_function()

try:
    print(local_var)
except NameError:
    print("Local variable not accessible outside function")
```

Output:

```
Local var: 50
Local variable not accessible outside function
```

Code Example 3: Global and Local with Same Name

```
x = 100

def modify():
    x = 50
    print("Inside function:", x)
```

```
modify()
print("Outside function:", x)
```

Output:

```
Inside function: 50
Outside function: 100
```

Code Example 4: Using global Keyword

```
counter = 0

def increment():
    global counter
    counter += 1

increment()
increment()
print("Counter:", counter)
```

Output:

```
Counter: 2
```

Code Example 5: Nested Function Scope

```
def outer():
    outer_var = "outer"

    def inner():
        inner_var = "inner"
        print(outer_var, inner_var)

    inner()

outer()
```

Output:

```
outer inner
```

Chapter 1 Summary

This chapter covered the fundamental concepts of Python programming, including:

- **Python Quick Start**: Introduction to Python and basic programs
- **Variables and Naming**: Creating and naming variables with proper conventions
- **Assignment Operators**: Combined operators for efficient code
- **Arithmetic Operations**: Mathematical operations on numeric data
- **Data Types**: Integer, float, string, and boolean types
- **Input/Output**: Reading from users and displaying results
- **Functions**: Creating reusable code blocks with parameters and returns

- **Conditional Statements**: Using if, elif, and else for branching logic
- **Loops**: Implementing while and for loops for repetition
- **Boolean Operators**: Combining conditions with and, or, not
- **Advanced Functions**: Multiple arguments, return values, and scope
- **Data Structures**: Lists, tuples, dictionaries, and sets
- **Scope Management**: Understanding global and local variables

These fundamentals form the foundation for all Python programming and problem-solving.
[1]

⁂