Prepared by Dr. Pravat Kumar Jena

Contents

1 PYTHON QUICK START

1.1 Theory

Python is a high-level, interpreted programming language known for its simplicity and readability. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming. Python is widely used for web development, data analysis, automation, and more.

To get started with Python:

- Install Python from the official website (python.org).
- Use an Integrated Development Environment (IDE) like IDLE, PyCharm, or VS Code.
- Run Python scripts from the command line using `python filename.py`.

The basic structure of a Python program starts with simple statements like printing output or performing calculations. Python uses indentation for code blocks instead of braces.

1.2 Examples

1.2.1 Hello World Program

This is the classic first program to print a greeting message.

```python
# Hello World example
print("Hello, World!")
```

Output:

```
Hello, World!
```

1.2.2 Simple Arithmetic

Perform basic calculations and print the result.

```python
# Simple arithmetic example
a = 10
b = 5
sum_ab = a + b
print("Sum of", a, "and", b, "is:", sum_ab)
```

Output:

```
Sum of 10 and 5 is: 15
```

1.2.3 User Input

Take input from the user and respond.

```python
# User input example
name = input("Enter your name: ")
print("Hello,", name, "! Welcome to Python.")
```

Sample Output (with input ”Alice”):

```
Enter your name: Alice
Hello, Alice! Welcome to Python.
```

2 VARIABLES AND NAMING

2.1 Theory

Variables in Python are used to store data values. Unlike other languages, Python has no explicit declaration; variables are created when assigned a value. Python uses dynamic typing, meaning the variable's type is determined at runtime based on the assigned value.

Naming conventions:

- Use lowercase letters for variable names.
- Separate words with underscores (snake_case).
- Avoid reserved keywords (e.g., if, for, while).
- Variable names should be descriptive and start with a letter or underscore.
- Constants are typically in uppercase (e.g., PI = 3.14).

Data types include integers, floats, strings, booleans, etc., and can be reassigned to different types.

2.2 Examples

2.2.1 Basic Variable Assignment

Assigning different types to variables.

```python
# Basic variable assignment
age = 25           # Integer
height = 5.9       # Float
name = "John"      # String
is_student = True # Boolean

print("Name:", name)
print("Age:", age)
print("Height:", height)
print("Is student?", is_student)
```

Output:

```
Name: John
Age: 25
Height: 5.9
Is student? True
```

2.2.2 Naming Conventions

Using proper snake_case naming.

```python
# Proper naming conventions
user_first_name = "Alice"
total_score = 95.5
max_attempts = 3
is_valid = False

print(f"User: {user_first_name}, Score: {total_score}")
```

Output:

```
User: Alice, Score: 95.5
```

2.2.3 Reassignment

Changing variable types.

```
# Variable reassignment
x = 10            # Initially integer
print("x as int:", x)

x = "ten"         # Reassigned to string
print("x as str:", x)
```

Output:

```
x as int: 10
x as str: ten
```

3 COMBINED ASSIGNMENT OPERATORS

3.1 Theory

Combined assignment operators (also known as augmented assignment operators) allow you to perform an operation and assign the result back to the variable in a concise way. They are shortcuts for common operations like addition, subtraction, etc.

Common operators:

- `+=` : Addition assignment (x += y is x = x + y)
- `-=` : Subtraction assignment
- `*=` : Multiplication assignment
- `/=` : Division assignment
- `//=` : Floor division assignment
- `%=` : Modulus assignment
- `**=` : Exponentiation assignment

These operators work with mutable types like lists and strings as well.

3.2 Examples

3.2.1 Arithmetic Combined Assignments

Using += and -= on numbers.

```
# Arithmetic combined assignments
counter = 0
counter += 5      # counter = counter + 5
print("After += 5:", counter)

counter -= 2      # counter = counter - 2
print("After -= 2:", counter)
```

Output:

```
After += 5: 5
After -= 2: 3
```

3.2.2 Multiplication and Division

Using *= and /=.

```
# Multiplication and division assignments
price = 10.0
quantity = 3
total = price * quantity

total *= 1.1      # Add 10% tax
print("Total after tax:", total)

total /= quantity # Per item
print("Price per item:", total)
```

Output:

```
Total after tax: 33.0
Price per item: 11.0
```

3.2.3 List Operations

Using combined assignments on lists.

```
# List combined assignments
numbers = [1, 2, 3]
numbers += [4, 5]  # Append list
print("After += [4,5]:", numbers)

numbers *= 2       # Duplicate list
print("After *= 2:", numbers)
```

Output:

```
After += [4,5]: [1, 2, 3, 4, 5]
After *= 2: [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
```

4 SUMMARY OF PYTHON ARITHMETIC OPERATORS

4.1 Theory

Python provides a rich set of arithmetic operators for numerical computations. These operators follow standard precedence rules (parentheses > exponents > multiplication/division > addition/subtraction), and associativity is left-to-right for most.

Operators summary:

- `+` : Addition
- `-` : Subtraction
- `*` : Multiplication
- `/` : True division (always returns float)
- `//` : Floor division (integer result)
- `%` : Modulus (remainder)
- `**` : Exponentiation (power)

These can be used with integers and floats. For complex numbers, additional support is available.

4.2 Examples

4.2.1 Basic Operators

Addition, subtraction, multiplication.

```python
# Basic arithmetic operators
a = 20
b = 6

add = a + b
sub = a - b
mul = a * b

print("Addition:", add)
print("Subtraction:", sub)
print("Multiplication:", mul)
```

Output:

```
Addition: 26
Subtraction: 14
Multiplication: 120
```

4.2.2 Division Operators

True division, floor division, modulus.

```python
# Division operators
a = 20
b = 6

div = a / b        # True division
floor_div = a // b # Floor division
mod = a % b        # Modulus

print("True division:", div)
print("Floor division:", floor_div)
print("Modulus:", mod)
```

Output:

```
True division: 3.3333333333333335
Floor division: 3
Modulus: 2
```

4.2.3 Exponentiation

Power operations.

```python
# Exponentiation operator
base = 2
exponent = 3

power = base ** exponent
print("2 raised to 3:", power)

# Square root approximation
sqrt_16 = 16 ** 0.5
print("Square root of 16:", sqrt_16)
```

Output:

```
2 raised to 3: 8
Square root of 16: 4.0
```

4.2.4 Operator Precedence

Demonstrating precedence.

```python
# Operator precedence
result = 2 + 3 * 4 ** 2 / 2 - 1
print("Result with precedence:", result)

# With parentheses
explicit = (2 + (3 * 4)) ** 2 / (2 - 1)
print("Explicit parentheses:", explicit)
```

Output:

```
Result with precedence: 25.0
Explicit parentheses: 400.0
```

5 ELEMENTARY DATA TYPES: INTEGER AND FLOATING POINT

5.1 Theory

Python supports two primary numeric data types for basic arithmetic: integers (int) and floating-point numbers (float).

Integers represent whole numbers with unlimited precision in Python 3 (no size limit). They can be positive, negative, or zero. Floating-point numbers represent real numbers with decimal points and are used for fractions and decimals. Floats have limited precision due to IEEE 754 standard, which can lead to rounding errors in calculations.

Key points:

- Integers: `int`, e.g., 42, -10, 0
- Floats: `float`, e.g., 3.14, -2.5, 1.0
- Type conversion: `int()` to convert to integer, `float()` to convert to float
- Division of two ints always returns a float in Python 3
- Floats can be written in scientific notation, e.g., 1.23e4

Remember: Avoid comparing floats for exact equality due to precision issues; use a small epsilon for comparisons.

5.2 Examples

5.2.1 Basic Integer and Float Operations

Working with ints and floats.

```python
# Integer and float examples
integer_num = 42
float_num = 3.14

print("Integer:", integer_num, "Type:", type(integer_num))
```

```python
print("Float:", float_num, "Type:", type(float_num))

# Operations
sum_result = integer_num + float_num
print("Sum:", sum_result, "Type:", type(sum_result))
```

Output:

```
Integer: 42 Type: <class 'int'>
Float: 3.14 Type: <class 'float'>
Sum: 45.14 Type: <class 'float'>
```

5.2.2 Type Conversion

Converting between types.

```python
# Type conversion
float_to_int = int(3.99)  # Truncates decimal
print("3.99 to int:", float_to_int)

int_to_float = float(42)
print("42 to float:", int_to_float)

# Division
result = 10 / 3
print("10 / 3:", result)
```

Output:

```
3.99 to int: 3
42 to float: 42.0
10 / 3: 3.3333333333333335
```

5.2.3 Precision Issues with Floats

Demonstrating floating-point precision.

```python
# Floating point precision
a = 0.1
b = 0.2
sum_ab = a + b
print("0.1 + 0.2 =", sum_ab)
print("Is equal to 0.3?", sum_ab == 0.3)

# Better comparison
epsilon = 1e-10
print("Close to 0.3?", abs(sum_ab - 0.3) < epsilon)
```

Output:

```
0.1 + 0.2 = 0.30000000000000004
Is equal to 0.3? False
Close to 0.3? True
```

6 BASIC INPUT AND OUTPUT

6.1 Theory

Python provides built-in functions for input and output operations. The `print()` function outputs data to the console, while `input()` reads user input as a string.

Key features:

- `print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`: Prints objects separated by sep, ending with end.
- `input(prompt='')`: Reads a line from input, returns string. Use `int(input())` or `float(input())` for numbers.
- Formatted output: f-strings (f""), `format()`, % formatting.
- For files: Use `open()` for reading/writing.

Remember: Always strip whitespace from input using `strip()` for robustness.

6.2 Examples

6.2.1 Basic Print Statements

Multiple ways to print.

```
# Basic print
print("Hello")
print("World")

# With sep and end
print("Hello", "World", sep=", ", end="!\n")
```

Output:

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

6.2.2 Input and Formatted Output

Reading input and formatting.

```
# Input and f-string
name = input("Enter name: ").strip()
age = int(input("Enter age: ").strip())

print(f"Hello {name}, you are {age} years old.")
```

Sample Output (inputs: Alice, 25):

```
Enter name: Alice
Enter age: 25
Hello Alice, you are 25 years old.
```

6.2.3 Reading Multiple Inputs

Handling multiple inputs.

```
# Multiple inputs
numbers = list(map(int, input("Enter numbers: ").split()))
print("You entered:", numbers)
print("Sum:", sum(numbers))
```

Sample Output (input: 1 2 3):

```
Enter numbers: 1 2 3
You entered: [1, 2, 3]
Sum: 6
```

7 FUNCTION DEFINITIONS

7.1 Theory

Functions are reusable blocks of code defined using the `def` keyword. They promote modularity and reduce redundancy.

Syntax: `def function_name(parameters):`

- Parameters: Optional, can have defaults, *args, **kwargs for variable arguments.
- Return: Use `return` to send back a value; default returns None.
- Scope: Local variables inside function.
- Docstrings: Use triple quotes for documentation.

Remember: Functions should follow the single responsibility principle; keep them focused.

7.2 Examples

7.2.1 Basic Function

Simple function without parameters.

```python
# Basic function
def greet():
    print("Hello from function!")

greet()
```

Output:

```
Hello from function!
```

7.2.2 Function with Parameters and Return

Using parameters and return.

```python
# Function with parameters
def add_numbers(a, b):
    """Add two numbers and return the sum."""
    return a + b

result = add_numbers(5, 3)
print("Sum:", result)
```

Output:

```
Sum: 8
```

7.2.3 Default Parameters and Variable Args

Advanced function features.

```python
# Default parameters
def multiply(x, y=2):
    return x * y

print("5 * 2 (default):", multiply(5))

# Variable args
```

```
def sum_all(*args):
    return sum(args)

print("Sum of 1,2,3,4:", sum_all(1,2,3,4))
```

Output:

```
5 * 2 (default): 10
Sum of 1,2,3,4: 10
```

8 THE PYTHON "IF" STATEMENT

8.1 Theory

The `if` statement implements conditional execution. It evaluates a boolean expression and executes the block if true.

Syntax:

```
if condition:
    # block
elif condition:
    # block
else:
    # block
```

Conditions use comparison operators (==, !=, >, <, >=, <=) and logical operators (and, or, not). Indentation defines the block.

Nested ifs and ternary operators (condition and value_if_true or value_if_false) are common.

Remember: Conditions should be boolean; non-empty strings, non-zero numbers are truthy.

8.2 Examples

8.2.1 Basic If-Elif-Else

Simple conditional.

```
# Basic if
score = 85
if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
else:
    grade = "C"

print("Grade:", grade)
```

Output:

```
Grade: B
```

8.2.2 Nested If

Nested conditions.

```
# Nested if
age = 20
has_license = True

```

```
if age >= 18:
    if has_license:
        print("Can drive")
    else:
        print("Need license")
else:
    print("Too young")
```

Output:

```
Can drive
```

8.2.3 Ternary Operator

Inline conditional.

```
# Ternary
x = 10
result = "Even" if x % 2 == 0 else "Odd"
print("x is", result)
```

Output:

```
x is Even
```

9 THE PYTHON "WHILE" STATEMENT

9.1 Theory

The `while` loop repeats a block as long as a condition is true. It's useful for indefinite iteration until a condition changes.

Syntax:

```
while condition:
    # block
    # update condition
```

Include a way to exit (e.g., break) to avoid infinite loops. Can use `else` clause that executes if no break occurred.

Remember: Use while when the number of iterations is unknown; prefer for loops for known ranges.

9.2 Examples

9.2.1 Basic While Loop

Counting loop.

```
# Basic while
count = 0
while count < 5:
    print("Count:", count)
    count += 1
```

Output:

```
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
```

9.2.2 While with Break and Continue

Control flow.

```
# While with break and continue
num = 0
while num < 10:
    num += 1
    if num == 5:
        continue  # Skip 5
    if num == 8:
        break     # Stop at 8
    print(num)
```

Output:

```
1
2
3
4
6
7
```

9.2.3 While-Else

Using else clause.

```
# While-else
found = False
i = 1
while i < 10:
    if i % 7 == 0:
        found = True
        break
    i += 1
else:
    print("7 not found")

if found:
    print("Found 7")
```

Output:

```
Found 7
```

10 A COUPLE OF COOL LITTLE APPS

10.1 Theory

This section presents simple applications demonstrating Python concepts. These apps combine input, conditionals, loops, and functions to solve practical problems.

Apps include:

- Number Guessing Game: User guesses a random number.
- Simple Calculator: Performs basic arithmetic based on user choice.

Remember: For real apps, handle errors with try-except; here we keep it simple.

10.2 Examples

10.2.1 Number Guessing Game

Guess a number between 1 and 100.

```
# Number Guessing Game
import random

secret = random.randint(1, 100)
guess = 0
attempts = 0

print("Guess the number between 1 and 100!")

while guess != secret:
    guess = int(input("Enter guess: "))
    attempts += 1
    if guess < secret:
        print("Too low!")
    elif guess > secret:
        print("Too high!")
    else:
        print(f"Correct! It took {attempts} attempts.")
```

Sample Output:

```
Guess the number between 1 and 100!
Enter guess: 50
Too low!
Enter guess: 75
Too high!
Enter guess: 62
Correct! It took 3 attempts.
```

10.2.2 Simple Calculator

Basic four-function calculator.

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

    if op == "+":
        result = num1 + num2
    elif op == "-":
        result = num1 - num2
    elif op == "*":
        result = num1 * num2
    elif op == "/":
        if num2 != 0:
            result = num1 / num2
        else:
            print("Division by zero!")
            return
    else:
        print("Invalid operator!")
```

```
        return

    print(f"Result: {num1} {op} {num2} = {result}")

calculator()
```

Sample Output (inputs: 10, +, 5):

```
Simple Calculator
Enter first number: 10
Enter operator (+, -, *, /): +
Enter second number: 5
Result: 10.0 + 5.0 = 15.0
```

11 SUMMARY OF PYTHON BOOLEAN OPERATORS, FUNCTION ARGUMENTS AND RETURN VALUES

11.1 Theory

This section summarizes boolean operators and function argument passing/return mechanisms.

Boolean Operators:

- `and`: True if both true
- `or`: True if at least one true
- `not`: Negates
- Short-circuit evaluation: and/or stop early

Function Arguments:

- Positional: By order
- Keyword: By name
- Defaults: Optional
- *args: Tuple of extras
- **kwargs: Dict of keyword extras

Return Values: Can return multiple via tuple; None if no return.

Remember: Mutable arguments can be modified in functions; pass copies if needed.

11.2 Examples

11.2.1 Boolean Operators

Using logical operators.

```
# Boolean operators
a = True
b = False

print("a and b:", a and b)
print("a or b:", a or b)
print("not a:", not a)

# Short-circuit
print("True and (1/0):", True and (1/0))  # Doesn't error
print("False or (1/0):", False or (1/0))  # Errors
```

Output (note: second print may error, but demonstrates concept):

```
a and b: False
a or b: True
not a: False
True and (1/0): True
```

11.2.2 Function Arguments

Different argument types.

```
# Function arguments
def func_example(required, optional="default", *args, **kwargs):
    print("Required:", required)
    print("Optional:", optional)
    print("Args:", args)
    print("Kwargs:", kwargs)
    return required + len(args)

# Calls
result1 = func_example(1)
print("Return 1:", result1)

result2 = func_example(2, "custom", 3, 4, key="value")
print("Return 2:", result2)
```

Output:

```
Required: 1
Optional: default
Args: ()
Kwargs: {}
Return 1: 1
Required: 2
Optional: custom
Args: (3, 4)
Kwargs: {'key': 'value'}
Return 2: 4
```

11.2.3 Multiple Returns

Returning tuples.

```
# Multiple returns
def divide_and_remainder(a, b):
    return a // b, a % b

quotient, remainder = divide_and_remainder(10, 3)
print("Quotient:", quotient, "Remainder:", remainder)
```

Output:

```
Quotient: 3 Remainder: 1
```

12 THE FORWARD REFERENCE PROBLEM

12.1 Theory

The forward reference problem occurs when code tries to use a variable or function before it is defined. In Python, this leads to NameError since variables are resolved at runtime.

Solutions:

- Define in order: Ensure definitions precede usage.
- Functions: Can call functions before definition if in same scope (hoisting-like, but not truly).
- Global variables: Use global keyword.
- For classes/methods: Forward references in type hints (from Python 3.7+ with postponed evaluation).

Remember: Always define before use; use functions to organize code.

12.2 Examples

12.2.1 Forward Reference Error

Demonstrating the error.

```python
# This will cause NameError
print(undefined_var)  # undefined_var not defined yet
undefined_var = 42
```

Output (Error):

```
NameError: name 'undefined_var' is not defined
```

12.2.2 Functions: Definition Order

Functions can be called before definition.

```python
# Function forward reference (works)
call_early()  # Defined later

def call_early():
    print("Function called!")
```

Output:

```
Function called!
```

12.2.3 Solution with If Guards

Using conditionals to avoid early reference.

```python
# Avoiding forward reference
use_later = False
if use_later:
    print(later_var)

later_var = 100
print("Later var defined:", later_var)
```

Output:

```
Later var defined: 100
```

13 PYTHON STRINGS

13.1 Theory

Strings are immutable sequences of characters, enclosed in quotes (' or "). They support indexing, slicing, and many methods.

Key features:

- Immutable: Cannot change in place.
- Indexing: s[0] for first char.
- Slicing: s[start:end:step]
- Methods: upper(), lower(), strip(), split(), join(), find(), replace()
- Formatting: f-strings, format(), %
- Raw strings: r"..." for escaping

Remember: Strings are unicode; use len() for length, which counts characters.

13.2 Examples

13.2.1 String Basics

Creation and indexing.

```
# String basics
s = "Hello, World!"
print("String:", s)
print("Length:", len(s))
print("First char:", s[0])
print("Slice:", s[7:12])
```

Output:

```
String: Hello, World!
Length: 13
First char: H
Slice: World
```

13.2.2 String Methods

Common operations.

```
# String methods
text = "  Python Programming  "
print("Upper:", text.upper())
print("Lower:", text.lower())
print("Stripped:", text.strip())
print("Split:", text.split())

words = ["Python", "is", "great"]
joined = " ".join(words)
print("Joined:", joined)
```

Output:

```
Upper:   PYTHON PROGRAMMING
Lower:   python programming
Stripped: Python Programming
Split: ['  Python', 'Programming  ']
Joined: Python is great
```

13.2.3 String Formatting

Different formatting methods.

```python
# String formatting
name = "Alice"
age = 30

# f-string
print(f"{name} is {age} years old.")

# format()
print("{} is {} years old.".format(name, age))

# %
print("%s is %d years old." % (name, age))
```

Output:

```
Alice is 30 years old.
Alice is 30 years old.
Alice is 30 years old.
```

14 PYTHON LISTS (AND A COOL SORTING APP)

14.1 Theory

Lists are mutable, ordered sequences that can hold mixed types. They are defined with [] and support dynamic sizing.

Key operations:

- Creation: [1, 2, 3]
- Indexing/Slicing: Similar to strings
- Methods: append(), extend(), insert(), remove(), pop(), sort(), reverse()
- List comprehensions: [expr for item in iterable]
- Mutability: Changes affect all references

Remember: Lists are passed by reference; use copy() for independent copies.

14.2 Examples

14.2.1 Basic List Operations

Creation and manipulation.

```python
# Basic lists
fruits = ["apple", "banana", "cherry"]
print("List:", fruits)
print("Length:", len(fruits))
```

```

# Append and access
fruits.append("date")
print("After append:", fruits)
print("Second item:", fruits[1])
```

Output:

```
List: ['apple', 'banana', 'cherry']
Length: 3
After append: ['apple', 'banana', 'cherry', 'date']
Second item: banana
```

14.2.2 List Methods and Comprehensions

Advanced features.

```
# List methods
numbers = [3, 1, 4, 1, 5]
numbers.sort()
print("Sorted:", numbers)

# Comprehension
squares = [x**2 for x in range(5)]
print("Squares:", squares)

# Filtered comprehension
evens = [x for x in range(10) if x % 2 == 0]
print("Evens:", evens)
```

Output:

```
Sorted: [1, 1, 3, 4, 5]
Squares: [0, 1, 4, 9, 16]
Evens: [0, 2, 4, 6, 8]
```

14.2.3 Cool Sorting App: Student Grades Sorter

Sort students by grades.

```
# Cool Sorting App: Sort students by grades
students = [
    ("Alice", 85),
    ("Bob", 92),
    ("Charlie", 78),
    ("David", 95)
]

# Sort by grade descending
sorted_students = sorted(students, key=lambda x: x[1], reverse=True)
print("Top students by grade:")
for name, grade in sorted_students:
    print(f"{name}: {grade}")
```

Output:

```
Top students by grade:
David: 95
Bob: 92
Alice: 85
Charlie: 78
```

15 THE "FOR" STATEMENT AND RANGES

15.1 Theory

The `for` loop iterates over iterables (lists, strings, ranges). It's ideal for known iteration counts.

`range(start, stop, step)` generates sequences efficiently.

Syntax:

```
for item in iterable:
    # block
```

Can use `else` if no break. With enumerate() for index-value pairs.

Remember: range() doesn't create a list; it's a generator for memory efficiency.

15.2 Examples

15.2.1 Basic For Loop

Iterating over list.

```python
# Basic for
colors = ["red", "green", "blue"]
for color in colors:
    print(color)
```

Output:

```
red
green
blue
```

15.2.2 Using Range

Generating numbers.

```python
# Range
for i in range(5):  # 0 to 4
    print(i)

for i in range(2, 8, 2):  # 2,4,6
    print(i)
```

Output:

```
0
1
2
3
4
2
4
6
```

15.2.3 Enumerate and For-Else

With indices and else.

```python
# Enumerate
fruits = ["apple", "banana"]
for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")

# For-else
for i in range(5):
    print(i)
else:
    print("Loop completed without break")
```

Output:

```
0: apple
1: banana
0
1
2
3
4
Loop completed without break
```

16 TUPLES

16.1 Theory

Tuples are immutable, ordered sequences similar to lists but cannot be modified after creation. Useful for fixed data, function returns.

Defined with ().

Key points:

- Immutable: No append, remove, etc.
- Indexing/Slicing: Same as lists
- Unpacking: a, b = (1, 2)
- Singletons: (1,) with comma
- Used as dict keys (hashable)

Remember: Tuples are faster than lists for fixed data; use for coordinates, returns.

16.2 Examples

16.2.1 Basic Tuple Operations

Creation and access.

```python
# Basic tuple
coords = (10, 20, 30)
print("Tuple:", coords)
print("Second:", coords[1])

# Unpacking
x, y, z = coords
print("Unpacked:", x, y, z)
```

Output:

```
Tuple: (10, 20, 30)
Second: 20
Unpacked: 10 20 30
```

16.2.2 Tuple Methods

Limited methods.

```
# Tuple methods (only count, index)
t = (1, 2, 2, 3)
print("Count of 2:", t.count(2))
print("Index of 3:", t.index(3))

# Immutable demo
# t[0] = 5  # This would error
```

Output:

```
Count of 2: 2
Index of 3: 3
```

16.2.3 Tuples as Dict Keys

Hashable example.

```
# Tuples as keys
locations = {(10, 20): "Home", (30, 40): "Work"}
print("Home location:", locations[(10, 20)])
```

Output:

```
Home location: Home
```

17 DICTIONARIES

17.1 Theory

Dictionaries are mutable, unordered (ordered since Python 3.7), hashable key-value pairs. Keys must be immutable (strings, tuples, etc.).

Syntax: {key: value}

Operations:

- Access: d[key]
- Add/Update: d[key] = value
- Delete: del d[key], pop()
- Methods: keys(), values(), items(), get(), update()
- Iteration: for k in d, for k,v in d.items()

Remember: Use get() to avoid KeyError; default dicts for missing keys.

17.2 Examples

17.2.1 Basic Dictionary

Creation and access.

```python
# Basic dict
person = {"name": "Alice", "age": 30, "city": "NY"}
print("Person:", person)
print("Name:", person["name"])

# Add
person["job"] = "Engineer"
print("Updated:", person)
```

Output:

```
Person: {'name': 'Alice', 'age': 30, 'city': 'NY'}
Name: Alice
Updated: {'name': 'Alice', 'age': 30, 'city': 'NY', 'job': 'Engineer'}
```

17.2.2 Dict Methods

Common methods.

```python
# Dict methods
print("Keys:", list(person.keys()))
print("Values:", list(person.values()))

# Get with default
job = person.get("job", "Unknown")
print("Job:", job)

# Items iteration
for k, v in person.items():
    print(f"{k}: {v}")
```

Output:

```
Keys: ['name', 'age', 'city', 'job']
Values: ['Alice', 30, 'NY', 'Engineer']
Job: Engineer
name: Alice
age: 30
city: NY
job: Engineer
```

17.2.3 Nested Dictionaries

Complex structure.

```python
# Nested dict
students = {
    "Alice": {"grade": 85, "subjects": ["Math", "Science"]},
    "Bob": {"grade": 92, "subjects": ["Math", "History"]}
}

print("Alice's grade:", students["Alice"]["grade"])
```

Output:

```
Alice's grade: 85
```

18 SETS

18.1 Theory

Sets are mutable, unordered collections of unique, hashable elements. No duplicates, fast membership testing.
Created with set() or {}.
Operations:

- Add/Remove: add(), remove(), discard()

- Set operations: union(|), intersection(), difference(-), symmetric_difference($^{)}Mutablevsimmutable : frozenset$
Remember: Sets are great for removing duplicates or membership checks; order not guaranteed.

18.2 Examples

18.2.1 Basic Set Operations

Creation and manipulation.

```
# Basic set
fruits = {"apple", "banana", "apple"}  # Duplicates removed
print("Set:", fruits)
print("Length:", len(fruits))

fruits.add("cherry")
print("After add:", fruits)

fruits.remove("banana")
print("After remove:", fruits)
```

Output:

- ```
  Set: {'apple', 'banana'}
  Length: 2
  After add: {'cherry', 'apple', 'banana'}
  After remove: {'cherry', 'apple'}
  ```

18.2.2 Set Operations

Mathematical sets.

```
# Set operations
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

union = set1 | set2
intersection = set1 & set2
difference = set1 - set2

print("Union:", union)
print("Intersection:", intersection)
print("Difference:", difference)
```

Output:

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

18.2.3 Frozenset

Immutable set.

```
# Frozenset
fs = frozenset([1, 2, 3])
print("Frozenset:", fs)

# Can be dict key
d = {fs: "value"}
print("Dict with frozenset key:", d[fs])
```

Output:

```
Frozenset: frozenset({1, 2, 3})
Dict with frozenset key: value
```

19 GLOBAL AND LOCAL VARIABLES

19.1 Theory

Python has local (function scope) and global (module scope) variables. Locals are preferred; globals for shared data.

Rules:

- Unassigned in function: Global if exists
- Assigned: Local (LEGB rule: Local, Enclosing, Global, Built-in)
- `global` keyword: To modify global inside function
- `nonlocal` for enclosing scopes (nested functions)

Remember: Avoid globals when possible; use parameters/returns for clarity. Use globals() to view all globals.

19.2 Examples

19.2.1 Local vs Global

Scope demonstration.

```
# Global variable
counter = 0

def increment():
    # local_counter = 1  # If uncommented, shadows global
    global counter
    counter += 1
    print("Inside function:", counter)

increment()
print("Outside function:", counter)
```

Output:

```
Inside function: 1
Outside function: 1
```

19.2.2 LEGB Rule

Resolution order.

```python
# LEGB example
x = "global"

def outer():
    x = "outer"  # Enclosing
    def inner():
        x = "inner"  # Local
        print("Inner:", x)
    inner()
    print("Outer:", x)

outer()
print("Global:", x)
```

Output:

```
Inner: inner
Outer: outer
Global: global
```

19.2.3 Nonlocal Keyword

For nested functions.

```python
# Nonlocal
def outer():
    count = 0
    def inner():
        nonlocal count
        count += 1
        return count
    return inner

inc = outer()
print("First call:", inc())
print("Second call:", inc())
```

Output:

```
First call: 1
Second call: 2
```