Advanced Python Programming - 1

Table of Contents

1. Python 2 Vs Python 3

- Python 3 is the newer and recommended version, while Python 2 is no longer supported.
- Python 3 handles text (Unicode) better, making it easier to work with different languages and special characters.
- It performs math operations more accurately, such as giving precise results for division by default.
- Python 3 offers better performance and compatibility with modern libraries.
- It is the preferred choice for new projects.

Feature	Python 2	Python 3
Print statement	`print` is a statement	`print()` is a function

Feature	Python 2	Python 3
Integer division	`5 / 2` returns `2`	`5 / 2` returns `2.5` (true division by default)
Unicode support	Strings are ASCII by default	Strings are Unicode by default
Iterating items	`dict.iteritems()`	`dict.items()`
xrange()	`xrange()` for iterating	`range()` replaces `xrange()`
Exception syntax	`except Exception, e:`	`except Exception as e:`
Input handling	`raw_input()` and `input()`	Only `input()` (equivalent to `raw_input()` in Python 2)
Libraries	Some libraries support Python 2 only	Modern libraries support Python 3
Print without parentheses	Allowed	Syntax error
Long integers	Explicit `L` suffix for longs	No `L` suffix; all integers are of type `int`
Compatibility	Outdated, no future support	Actively maintained and recommended
Performance	Slower due to legacy features	Faster and more efficient

2. Why Python

- **History of Python**
- Created in 1990 by **Guido van Rossum**.
- Python 3 released in 2008 with subsequent versions (3.1, 3.2, etc.).
- Designed for readability and ease of use.
- **Why Choose Python?**
- Clear, readable, and easy-to-learn syntax.
- Uses **whitespace and indentation** instead of braces for clean code.
- Python code is readable even by non-Python programmers.
- Large number of existing libraries and frameworks.
- Optimizes **developer time** over processing time.
- Extensive online documentation and support.
- **What Can You Do with Python?**
- **Base Python**:
- Core components for writing scripts and small programs.

- Comes with built-in modules like `random` and `math`.
- **Outside Libraries and Frameworks**:
- Expand Python's capabilities with external libraries.
- Easily downloadable and integrable.
- **Applications of Python**
- **Automation**:
- File handling, web scraping, working with Excel and PDFs.
- Automating emails, text messages, and form filling.
- **Data Science and Machine Learning**:
- Data analysis with libraries like **NumPy** and **Pandas**.
- Visualization with **Seaborn** and **Matplotlib**.
- Machine learning with **scikit-learn** and **TensorFlow**.
- **Web Development**:
- Backend development using **Django** and **Flask**.
- Interactive dashboards with **Plotly** and **Dash**.

3. Data Types

4. Numeric Types

Type	Description	Example
int	Integer numbers	`10`, `-3`, `0`
float	Decimal (floating point) numbers	`3.14`, `-0.5`, `2.0`
complex	Complex numbers	`3+4j`, `1-2j`

5. Text Type

Type	Description	Example
str	Sequence of characters	`"Hello"`, `'123'`

6. Boolean Type

Type	Description	Example
bool	Logical value	`True`, `False`

7. Sequence Types

Type	Description	Example
list	Ordered, mutable collection	`[1, 2, 3]`, `['a', 'b']`
tuple	Ordered, immutable collection	`(1, 2, 3)`
range	Sequence of numbers	`range(5)` → `0,1,2,3,4`

8. Mapping Type

Type	Description	Example
dict	Key-value pairs	`{"name": "Alice", "age": 25}`

9. Set Types

Type	Description	Example
set	Unordered, unique items	`{1, 2, 3}`
frozenset	Immutable set	`frozenset([1, 2, 3])`

10. None Type

Type	Description	Example
NoneType	Represents "no value"	`None`

11. Example Code

```
x = 5              # int
y = 3.14           # float
z = "Hello"        # str
flag = True        # bool
lst = [1, 2, 3]    # list
tpl = (4, 5, 6)    # tuple
d = {"a": 1, "b": 2} # dict
s = {1, 2, 3}      # set
none_value = None  # NoneType
```

12. Python Operators

Operators in Python are special symbols or keywords used to perform operations on variables and

values. Python supports several types of operators:

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Identity Operators
- Membership Operators
- Bitwise Operators

1. Arithmetic Operators

Arithmetic operators perform mathematical operations like addition, subtraction, multiplication, etc.

Operator	Description	Example
`+`	Addition	`x + y` → Adds two numbers
`-`	Subtraction	`x - y` → Subtracts the right-hand operand from the left-hand operand
`*`	Multiplication	`x * y` → Multiplies two numbers
`/`	Division	`x / y` → Divides left-hand operand by right-hand operand (returns float)
`//`	Floor Division	`x // y` → Divides and returns the integer part
`%`	Modulus	`x % y` → Returns the remainder
`*`	Exponentiation	`x * y` → Raises x to the power of y

Example:

```
x = 10
y = 3

print(x + y)    # 13
print(x - y)    # 7
print(x * y)    # 30
print(x / y)    # 3.33
print(x // y)   # 3
print(x % y)    # 1
print(x * y)   # 1000
```

2. Comparison (Relational) Operators

These operators compare two values and return a boolean result (`True` or `False`).

Operator	Description	Example
`==`	Equal to	`x == y` → `True` if x is equal to y
`!=`	Not equal	`x != y` → `True` if x is not equal to y
`>`	Greater than	`x > y` → `True` if x is greater than y
`<`	Less than	`x < y` → `True` if x is less than y
`>=`	Greater than or equal to	`x >= y` → `True` if x is greater than or equal to y
`⇐`	Less than or equal to	`x ⇐ y` → `True` if x is less than or equal to y

Example:

```
a = 10
b = 5

print(a > b)   # True
print(a < b)   # False
print(a == b)  # False
print(a != b)  # True
```

3. Assignment Operators

Assignment operators assign values to variables.

Operator	Description	Example
`=`	Assign	`x = 10` → Assigns value 10 to x
`+=`	Add and assign	`x += 5` → Equivalent to `x = x + 5`
`-=`	Subtract and assign	`x -= 3` → Equivalent to `x = x - 3`
`*=`	Multiply and assign	`x *= 2` → Equivalent to `x = x * 2`
`/=`	Divide and assign	`x /= 4` → Equivalent to `x = x / 4`

Operator	Description	Example
`//=`	Floor division and assign	`x //= 2` → Equivalent to `x = x // 2`
`%=`	Modulus and assign	`x %= 3` → Equivalent to `x = x % 3`
`*=`	Exponentiate and assign	`x *= 2` → Equivalent to `x = x * 2`

Example:

```
x = 10

x += 5   # x = x + 5 → 15
x *= 2   # x = x * 2 → 30
x /= 3   # x = x / 3 → 10.0
x %= 4   # x = x % 4 → 2
```

---

4. Logical Operators

Logical operators are used to combine conditional statements.

Operator	Description	Example
`and`	Returns `True` if both conditions are `True`	`x > 5 and x < 15`
`or`	Returns `True` if either condition is `True`	`x > 5 or x < 3`
`not`	Returns `True` if the condition is `False`	`not(x > 5)`

Example:

```
x = 10
y = 5

print(x > 5 and y < 10)  # True
print(x > 15 or y < 10)  # True
print(not(x < 5))        # True
```

---

5. Identity Operators

Identity operators compare the memory location of two objects.

Operator	Description	Example
`is`	Returns `True` if both variables refer to the same object	`x is y`
`is not`	Returns `True` if both variables do not refer to the same object	`x is not y`

Example:

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is z)      # True
print(x is y)      # False
print(x == y)      # True (content is the same)
```

---

6. Membership Operators

Membership operators check for the existence of a value in a sequence (string, list, tuple, etc.).

Operator	Description	Example
`in`	Returns `True` if a value is found in the sequence	`"a" in "apple"`
`not in`	Returns `True` if a value is not in the sequence	`"z" not in "apple"`

Example:

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

print(3 in x)         # True
print(10 not in x)    # True
```

---

7. Bitwise Operators

Bitwise operators perform operations on the **binary representation** of integers.

Operator	Description	Example
`&`	AND	`x & y`
`	`	`	OR

Operator	Description	Example
`x	y`	^
XOR	x ^ y	~
NOT	~x	<<
Left Shift	x << 2	>>

Example:

```
x = 6    # 110 in binary
y = 4    # 100 in binary

print(x & y)     # 4 → (100 in binary)
print(x | y)     # 6 → (110 in binary)
print(x ^ y)     # 2 → (010 in binary)
print(~x)        # -7 → Inverts all bits
print(x << 1)    # 12 → Left shift by 1
print(x >> 1)    # 3 → Right shift by 1
```

13. Python Functions

- **1. Introduction to Python Functions**
  - A **function** is a block of reusable code that performs a specific task.
  - Functions help in:
  - Reducing code duplication.
  - Improving readability and organization.
  - Enhancing reusability and modularity.

- **2. Syntax of a Function**

```
def function_name(parameters):
    """Docstring explaining the function"""
    # Code block
    return value
```

- `def` → Keyword used to define a function.
- `function_name` → Name of the function.
- `parameters` → Optional variables passed to the function.
- `return` → Returns the result (optional).

- **3. Types of Functions in Python**
  1. **Built-in Functions**
     - Pre-defined in Python.
     - Examples: `print()`, `len()`, `max()`, `min()`.
  2. **User-defined Functions**
     - Defined by the user to perform specific tasks.

- **4. Creating and Calling Functions**

Example 1: Simple Function

```
# Function to greet
def greet():
    print("Hello, welcome to Python!")

# Calling the function
greet()
```

Output:

```
Hello, welcome to Python!
```

- **5. Function with Parameters**

Example 2: Passing Parameters

```
# Function with parameters
def greet(name):
    print(f"Hello, {name}!")

# Calling the function with arguments
greet("Alice")
greet("Bob")
```

Output:

```
Hello, Alice!
Hello, Bob!
```

- **6. Function with Return Statement**
  - `return` is used to send the result back to the caller.

Example 3: Return Value

```
# Function to add two numbers
def add(a, b):
    return a + b

# Calling the function
result = add(5, 3)
print("Sum:", result)
```

Output:

```
Sum: 8
```

---

7. Default Parameter Values

- You can set **default values** for parameters.

Example 4: Default Parameter

```
# Function with default value
def greet(name="Guest"):
    print(f"Hello, {name}!")

# Calling the function
greet("Alice")     # Uses provided value
greet()            # Uses default value
```

Output:

```
Hello, Alice!
Hello, Guest!
```

- **8. Keyword Arguments**
  - You can specify parameter names during the function call.

Example 5: Keyword Arguments

```
# Function with multiple parameters
```

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

# Using keyword arguments
display_info(age=25, name="Alice")
```

Output:

```
Name: Alice, Age: 25
```

- **9. Variable-length Arguments**
  - `*args` → For multiple **positional** arguments.
  - `*kwargs` → For multiple **keyword** arguments.

Example 6: Using `*args`

```
# Function with *args
def add(*numbers):
    total = sum(numbers)
    print("Sum:", total)

# Calling the function with multiple arguments
add(1, 2, 3)
add(5, 10, 15, 20)
```

Output:

```
Sum: 6
Sum: 50
```

Example 7: Using `*kwargs`

```
# Function with *kwargs
def display_info(*data):
    for key, value in data.items():
        print(f"{key}: {value}")

# Calling with keyword arguments
display_info(name="Alice", age=25, city="New York")
```

Output:

```
name: Alice
```

```
age: 25
city: New York
```

- **Practical Examples**

Example 14: Temperature Conversion

```
def celsius_to_fahrenheit(celsius):
    return (celsius * 9/5) + 32

print("Temperature in Fahrenheit:", celsius_to_fahrenheit(30))
```

Example 15: Even or Odd Checker

```
def check_even_odd(num):
    if num % 2 == 0:
        print("Even")
    else:
        print("Odd")

check_even_odd(10)
check_even_odd(15)
```

Function Examples

```
def manage_set():
    # Create a set
    colors = {'red', 'green', 'blue'}

    # Add an element
    colors.add('yellow')

    # Remove an element
    colors.discard('green')

    print("Updated Set:", colors)

# Call the function
manage_set()
```

```python
def tuple_operations():
    # Create a tuple
    fruits = ('apple', 'banana', 'cherry', 'mango')

    # Accessing elements
    print("First fruit:", fruits[0])

    # Slicing tuple
    print("Last two fruits:", fruits[-2:])

# Call the function
tuple_operations()
```

```python
def dict_operations():
    # Create a dictionary
    student = {'name': 'Alice', 'age': 22, 'grade': 'A'}

    # Add a new key-value pair
    student['city'] = 'New York'

    # Retrieve a value
    print("Student's Name:", student['name'])

    # Print the entire dictionary
    print("Updated Dictionary:", student)

# Call the function
dict_operations()
```

- **Best Practices for Writing Functions**
  - Use meaningful names for functions.
  - Keep functions small and focused on a single task.
  - Use comments and docstrings to describe the function.
  - Avoid using global variables inside functions.
  - Use `return` wisely to send back necessary results.

14. Python Loops

Loops in Python are used to **iterate over a sequence** (like a list, tuple, dictionary, set, or string) and

execute a block of code repeatedly.

14.1. `for` Loop

The `for` loop iterates over a sequence and executes the block of code for each element.

Syntax:

```
for variable in sequence:
    # code block
```

Example:

```
# Loop through a list
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
# Output:
# apple
# banana
# cherry
```

Using `range()` with `for` loop:

```
# Iterating over a range of numbers
for i in range(5):
    print(i)  # 0 1 2 3 4
```

Iterating with step:

```
# Using step in range()
for i in range(1, 10, 2):
    print(i)  # 1 3 5 7 9
```

14.2. `while` Loop

The `while` loop executes the block of code repeatedly **as long as the condition is `True`.**

Syntax:

```
while condition:
    # code block
```

Example:

```
# Print numbers from 1 to 5
```

```
x = 1

while x <= 5:
    print(x)
    x += 1
# Output: 1 2 3 4 5
```

14.3. Loop Control Statements

Python provides **control statements** to alter the flow of loops.

- `break`: Terminates the loop immediately.
- `continue`: Skips the current iteration and continues with the next.
- `pass`: Does nothing and is used as a placeholder.

Using `break`*:*

```
# Exit loop when i == 3
for i in range(5):
    if i == 3:
        break
    print(i)  # 0 1 2
```

Using `continue`*:*

```
# Skip iteration when i == 3
for i in range(5):
    if i == 3:
        continue
    print(i)  # 0 1 2 4
```

Using `pass`*:*

```
# Placeholder in loop
for i in range(5):
    if i == 2:
        pass  # Do nothing
    print(i)  # 0 1 2 3 4
```

Purpose of `pass` in Python

Even though `pass` **does nothing** when executed, it serves specific purposes in Python programming:

- 1. **Placeholder for Incomplete Code**

- When you are writing code but **haven't implemented** a section yet, you can use `pass` to avoid syntax errors.
- Without `pass`, Python will throw an **IndentationError** or **SyntaxError** if a block is empty.

Example:

```
# Placeholder function
def future_function():
    pass  # To be implemented later

# Placeholder class
class Sample:
    pass
```

- Without `pass`, the interpreter would raise an **IndentationError** because Python expects at least one statement inside the function or class.

- 2. **Maintaining Structure**
  - When creating complex **control flow** (loops, conditionals, etc.), `pass` allows you to **define the structure** without adding logic immediately.

Example:

```
# Placeholder while structuring the code
for i in range(5):
    if i % 2 == 0:
        pass  # To be implemented later
    else:
        print(i)
```

- You can replace `pass` with actual code later, keeping the loop structure intact.

- 3. **Avoiding Syntax Errors**
  - Python **doesn't allow empty blocks**. Using `pass` helps you create empty blocks temporarily.

Example:

```
if True:
    pass  # No action yet
else:
    print("False")
```

- Without `pass`, the code will result in a **SyntaxError**.

Key Takeaway

- `pass` is useful as a **placeholder** when you are drafting the code.
- It helps **avoid syntax errors** in empty blocks.
- It allows you to **maintain the structure** of the code during development, even if the logic isn't implemented yet.

14.4. Nested Loops

A **nested loop** is a loop inside another loop. The inner loop executes **completely for each iteration** of the outer loop.

Example:

```
# Nested loop to create a multiplication table
for i in range(1, 4):
    for j in range(1, 4):
        print(i * j, end=" ")
    print()
# Output:
# 1 2 3
# 2 4 6
# 3 6 9
```

Conclusion

- `for` loop: Used for **iterating over sequences**.
- `while` loop: Repeats **as long as the condition is `True`**.
- Control statements: `break`, `continue`, and `pass` alter loop execution.
- Nested loops: Useful for **complex iterations**.

15. Tuples

- A **tuple** is a **collection of ordered and immutable elements**.
- It is similar to a list, but **tuples cannot be modified** after creation.
- Tuples are defined using **parentheses `()`**.

Creating Tuples

Basic Tuple

```
# Creating a tuple with integers
```

```
numbers = (1, 2, 3, 4)
print(numbers)  # Output: (1, 2, 3, 4)

# Tuple with mixed data types
person = ("Alice", 25, True)
print(person)  # Output: ('Alice', 25, True)
```

Tuple with One Element

- To create a tuple with a single element, add a **comma** after the value.

```
# Single element tuple
single = (5,)
print(type(single))  # Output: <class 'tuple'>

# Without the comma, it is treated as an integer
not_a_tuple = (5)
print(type(not_a_tuple))  # Output: <class 'int'>
```

Empty Tuple

```
# Empty tuple
empty = ()
print(empty)  # Output: ()
```

Tuple Indexing and Slicing

Accessing Elements by Index * Tuples are **indexed from 0**.

```
fruits = ('apple', 'banana', 'cherry')

# Accessing by index
print(fruits[0])  # Output: apple
print(fruits[2])  # Output: cherry
```

Negative Indexing

- You can use **negative indices** to access elements from the end.

```
colors = ('red', 'green', 'blue')

print(colors[-1])  # Output: blue
print(colors[-2])  # Output: green
```

Slicing * You can extract a **portion of a tuple** using slicing.

```
numbers = (0, 1, 2, 3, 4, 5)

# Slicing from index 1 to 4 (exclusive)
print(numbers[1:4])  # Output: (1, 2, 3)

# Slicing with step
print(numbers[::2])  # Output: (0, 2, 4)
```

---

4. Tuple Operations

Concatenation

```
# Combining two tuples
a = (1, 2, 3)
b = (4, 5, 6)
combined = a + b
print(combined)  # Output: (1, 2, 3, 4, 5, 6)
```

Repetition

```
# Repeating a tuple multiple times
t = ('Hello',) * 3
print(t)  # Output: ('Hello', 'Hello', 'Hello')
```

Membership Check

```
# Check if an element is in a tuple
fruits = ('apple', 'banana', 'cherry')
```

```
print('apple' in fruits)  # Output: True
print('grape' in fruits)  # Output: False
```

Tuple Methods

Tuples have **only two built-in methods**:

`count()` → **Counts the occurrences of an element**

```
numbers = (1, 2, 3, 1, 4, 1)

# Count occurrences of 1
print(numbers.count(1))  # Output: 3
```

`index()` → **Finds the index of the first occurrence**

```
letters = ('a', 'b', 'c', 'a', 'd')

# Get the index of the first 'a'
print(letters.index('a'))  # Output: 0
```

Tuple Unpacking

- Tuple unpacking allows you to **assign tuple elements** to multiple variables.

```
# Tuple unpacking
person = ('Alice', 25, 'Engineer')

name, age, job = person
print(name)  # Output: Alice
print(age)   # Output: 25
print(job)   # Output: Engineer
```

Nesting and Tuple of Tuples

- You can have **tuples inside tuples**.

```
# Nested tuple
nested = ((1, 2), (3, 4), (5, 6))
```

```
# Accessing elements
print(nested[0])       # Output: (1, 2)
print(nested[0][1])    # Output: 2
```

Tuple vs List

Feature	Tuple
List	**Mutability**
Immutable (cannot change)	Mutable (can modify)
Syntax	Uses `()`
Uses `[]`	**Performance**
Faster	Slightly slower
Methods	Only 2 methods
Many built-in methods	**Use case**
Fixed data	Dynamic data manipulation

When to Use Tuples

- When you need **immutable data** (e.g., coordinates, dates).
- For **faster performance** compared to lists.
- When using as **keys in dictionaries** (since they are hashable).

Tuple Exercises

Exercise 1: Create a tuple with the names of 5 cities and display the third city.

Exercise 2: Create a tuple of numbers `(1, 2, 3, 4, 5)` and square each element using a `for` loop.

Exercise 3: Create two tuples and concatenate them into a new tuple.

Exercise 4: Check if the number `10` is in the tuple `(5, 10, 15, 20)` using membership operator.

Exercise 5: Create a nested tuple and access an element inside the inner tuple.

- **Key Takeaways**
- Tuples are **immutable** and **ordered** collections.
- They support **indexing, slicing, and repetition**.

- Tuples are **faster and memory-efficient** compared to lists.
- Used in cases where **data should remain unchanged**.
- Ideal for **storing fixed collections of items**.

16. Sets

- A **set** is an **unordered collection of unique elements**.
- Sets are defined using **curly braces** `{}` or the `set()` function.
- Sets automatically remove duplicate values.
- Sets are **mutable**, but the elements must be **immutable** (e.g., numbers, strings, tuples).

Creating Sets

Basic Set

```
# Creating a set with integers
numbers = {1, 2, 3, 4, 5}
print(numbers)  # Output: {1, 2, 3, 4, 5}

# Set with mixed data types
mixed = {10, "Python", 3.14, True}
print(mixed)  # Output: {True, 10, 3.14, 'Python'}
```

Using the `set()` Function

```
# Using set() to create a set
fruits = set(['apple', 'banana', 'cherry'])
print(fruits)  # Output: {'apple', 'banana', 'cherry'}
```

Empty Set

```
# Empty set using set() function
empty = set()
print(type(empty))  # Output: <class 'set'>

# Empty curly braces create an empty dictionary, not a set
not_a_set = {}
print(type(not_a_set))  # Output: <class 'dict'>
```

Set Characteristics

Unordered Collection * The elements of a set are **unordered** (no guaranteed order).

```
# Set order is not guaranteed
items = {'pen', 'book', 'eraser'}
print(items)  # Output could be: {'eraser', 'pen', 'book'}
```

Unique Elements * Sets automatically remove **duplicate elements**.

```
numbers = {1, 2, 2, 3, 4, 4, 5}
print(numbers)  # Output: {1, 2, 3, 4, 5}
```

Adding and Removing Elements

Adding Elements

```
# Using add() to add a single element
languages = {'Python', 'Java'}
languages.add('C++')
print(languages)  # Output: {'Python', 'Java', 'C++'}
```

Removing Elements

```
# Using remove() → Raises an error if the element does not exist
fruits = {'apple', 'banana', 'cherry'}
fruits.remove('banana')
print(fruits)  # Output: {'apple', 'cherry'}

# Using discard() → No error if the element does not exist
fruits.discard('grape')  # No error even though 'grape' is not in the set
```

Clearing the Set

```
# Using clear() to remove all elements
numbers = {1, 2, 3, 4}
numbers.clear()
print(numbers)  # Output: set()
```

Set Operations

Union

- Combines **all unique elements** from two sets.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Using union() or |
result = set1.union(set2)
print(result)  # Output: {1, 2, 3, 4, 5}

# Using | operator
print(set1 | set2)  # Output: {1, 2, 3, 4, 5}
```

Intersection

- Finds **common elements** between two sets.

```
set1 = {1, 2, 3}
set2 = {2, 3, 4}

# Using intersection() or &
result = set1.intersection(set2)
print(result)  # Output: {2, 3}

# Using & operator
print(set1 & set2)  # Output: {2, 3}
```

Difference

- Finds elements that are in the **first set but not in the second**.

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

# Using difference() or -
result = set1.difference(set2)
print(result)  # Output: {1, 2}

# Using - operator
print(set1 - set2)  # Output: {1, 2}
```

Symmetric Difference

- Finds elements that are **in either set but not in both**.

```
set1 = {1, 2, 3}
```

```
set2 = {3, 4, 5}

# Using symmetric_difference() or ^
result = set1.symmetric_difference(set2)
print(result)  # Output: {1, 2, 4, 5}

# Using ^ operator
print(set1 ^ set2)  # Output: {1, 2, 4, 5}
```

Set Methods

`add()` → **Adds an element to the set.**

```
s = {1, 2, 3}
s.add(4)
print(s)  # Output: {1, 2, 3, 4}
```

`update()` → **Adds multiple elements from another set or iterable.**

```
s = {1, 2, 3}
s.update([4, 5])
print(s)  # Output: {1, 2, 3, 4, 5}
```

`pop()` → **Removes and returns a random element.**

```
s = {1, 2, 3, 4}
print(s.pop())  # Output: Random element
print(s)        # Set with remaining elements
```

`issubset()` → **Checks if one set is a subset of another.**

```
s1 = {1, 2}
s2 = {1, 2, 3, 4}

print(s1.issubset(s2))  # Output: True
```

`issuperset()` → **Checks if one set is a superset of another.**

```
s1 = {1, 2, 3, 4}
s2 = {2, 3}

print(s1.issuperset(s2))  # Output: True
```

🔹 Iterating Through a Set

```
# Iterating through a set
colors = {'red', 'green', 'blue'}

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

🔹 Converting Lists and Tuples to Sets

Converting List to Set

```
# Removing duplicates using set
numbers = [1, 2, 2, 3, 4, 4, 5]
unique_numbers = set(numbers)
print(unique_numbers)  # Output: {1, 2, 3, 4, 5}
```

Converting Tuple to Set

```
# Converting a tuple to a set
data = (1, 2, 3, 2, 1)
unique_data = set(data)
print(unique_data)  # Output: {1, 2, 3}
```

🔹 When to Use Sets

- When you need to **store unique values**.
- For **removing duplicates** from a list or tuple.
- When you need to **perform mathematical operations** like union, intersection, etc.
- Sets are **faster for membership testing** compared to lists.

🔹 10. Set Exercises

Exercise 1: Create two sets of student names and find the union and intersection.

Exercise 2: Create a set of numbers and remove all even numbers.

Exercise 3: Check if one set is a subset of another.

Exercise 4: Convert a list with duplicates into a set and display the unique values.

Exercise 5: Find the symmetric difference between two sets.

 Key Takeaways

- Sets are **unordered collections of unique elements**.
- Support operations like **union, intersection, and difference**.
- Useful for **removing duplicates** from data.
- Provide efficient **membership testing** compared to lists.
- Sets are **mutable** but can only hold **immutable elements**.

17. Dictionary

Introduction to Dictionary

- A **dictionary** is an **unordered**, **mutable** collection in Python.
- It stores data in **key-value pairs**.
- Each key is **unique** and used to access its corresponding value.

Creating Dictionaries

- Use **curly braces** `{}` or the `dict()` constructor to create dictionaries.
- Keys must be **immutable** (strings, numbers, or tuples), while values can be of any type.

```
# Creating dictionaries
empty_dict = {}  # Empty dictionary
person = {'name': 'Alice', 'age': 25, 'city': 'New York'}  # Dictionary with string
keys
numbers = {1: 'one', 2: 'two', 3: 'three'}  # Dictionary with integer keys
mixed = {'name': 'Bob', 42: [1, 2, 3], True: 'yes'}  # Mixed key types
```

Accessing Dictionary Elements

- Use **keys** to access values.
- Attempting to access a non-existent key raises a `KeyError`.

```
# Accessing values
```

```
person = {'name': 'Alice', 'age': 25, 'city': 'New York'}

print(person['name'])   # Alice
print(person['age'])    # 25

# Using get() to avoid KeyError
print(person.get('city'))       # New York
print(person.get('country'))    # None (no KeyError)
```

Modifying Dictionaries

- You can **add, update, and delete** key-value pairs.

```
# Adding new key-value pairs
person['email'] = 'alice@example.com'
print(person)  # {'name': 'Alice', 'age': 25, 'city': 'New York', 'email':
'alice@example.com'}

# Modifying existing values
person['age'] = 26
print(person)  # {'name': 'Alice', 'age': 26, 'city': 'New York', 'email':
'alice@example.com'}

# Deleting key-value pairs
del person['city']
print(person)  # {'name': 'Alice', 'age': 26, 'email': 'alice@example.com'}
```

Dictionary Methods

1. `keys()` → Returns all keys

```
person = {'name': 'Alice', 'age': 25, 'city': 'New York'}
print(person.keys())  # dict_keys(['name', 'age', 'city'])
```

2. `values()` → Returns all values

```
print(person.values())  # dict_values(['Alice', 25, 'New York'])
```

3. `items()` → Returns all key-value pairs

```
print(person.items())  # dict_items([('name', 'Alice'), ('age', 25), ('city', 'New
```

```
York')])
```

4. `pop()` → Removes and returns the value of a given key

```
age = person.pop('age')
print(age)        # 25
print(person)     # {'name': 'Alice', 'city': 'New York'}
```

5. `update()` → Merges another dictionary

```
person.update({'email': 'alice@example.com', 'age': 27})
print(person)  # {'name': 'Alice', 'city': 'New York', 'email': 'alice@example.com',
'age': 27}
```

Dictionary Iteration

- Use loops to iterate over keys, values, or both.

```
person = {'name': 'Alice', 'age': 25, 'city': 'New York'}

# Iterating over keys
for key in person:
    print(key)  # name, age, city

# Iterating over values
for value in person.values():
    print(value)  # Alice, 25, New York

# Iterating over key-value pairs
for key, value in person.items():
    print(f"{key}: {value}")
```

Key Takeaways

- **Dictionaries** store data in **key-value pairs**.
- Keys must be **unique** and immutable.
- Use dictionary methods like `keys()`, `values()`, and `items()` to access and modify data.

18. References

- https://drive.google.com/drive/folders/1CKqOQzst1cGURXGiRVivi2Xsc0n-X8CR
- https://github.com/Pierian-Data/Complete-Python-3-Bootcamp