Table of Contents

1.1 Python Installation

Python Installation & Setup

Installing Python on Windows

To install Python on Windows, visit the official Python website at python.org/downloads. Download the latest version (e.g., Python 3.11). Run the installer and ensure you check the box for `Add Python to PATH` before clicking "Install Now".

Verify the installation by opening Command Prompt and running:

```
Command
python --version
```

This should display the installed Python version (e.g., Python 3.11.4).

Installing Python on macOS

Download the macOS installer from python.org/downloads. Run the installer and follow the prompts. Alternatively, use Homebrew for a more streamlined installation:

```
Command
brew install python
```

Verify the installation by opening Terminal and running:

```
Command
python3 --version
```

Note: Use `python3` to avoid conflicts with the system Python (Python 2).

Installing Python on Linux

Most Linux distributions come with Python pre-installed. To install or upgrade Python, use your package manager. For Ubuntu/Debian, run:

```
Command
sudo apt update
sudo apt install python3
```

For Fedora, use:

```
Command
sudo dnf install python3
```

Verify the installation:

```
Command
python3 --version
```

Setting Up a Python Development Environment

A good Integrated Development Environment (IDE) enhances Python development. Popular choices include:

- **Visual Studio Code (VS Code):** Lightweight, with Python extensions for linting, debugging, and IntelliSense. Download from code.visualstudio.com.
- **PyCharm:** Feature-rich IDE for professional developers, available at jetbrains.com/pycharm.

For VS Code, install the Python extension and configure it to use your Python interpreter:

```
Command
python3 -m pip install --user pylint
```

Using Python Virtual Environments

Virtual environments isolate project dependencies. Create a virtual environment using:

```
Command
python3 -m venv myenv
```

Activate it:

- **Windows:**

  ```
  Command
  myenv\Scripts\activate
  ```

- **macOS/Linux:**

  ```
  Command
  source myenv/bin/activate
  ```

Deactivate by running:

```
Command
deactivate
```

Installing and Managing Packages with pip

`pip` is Python's package manager. Install a package (e.g., `requests`) with:

```
Command
pip install requests
```

List installed packages:

```
Command
pip list
```

Upgrade a package:

```
Command
pip install --upgrade requests
```

Use `requirements.txt` to manage dependencies:

```
# requirements.txt
requests==2.28.1
numpy==1.24.3
```

Install from `requirements.txt`:

```
Command
pip install -r requirements.txt
```

Challenging Questions

> **Question 1:** What happens if you forget to check "Add Python to PATH" during Windows installation, and how can you fix it?
>
> Show Answer

> **Question 2:** Why might `pip` install a package globally instead of in a virtual environment, and how can you ensure it installs in the virtual environment?

Show Answer

Question 3: What is the output of the following command in a virtual environment, and why?

```Command
pip list
```

Assume the virtual environment was just created and no additional packages were installed.

Show Answer

Print Page

1.2 Variables

Python Variables Explained

What Are Variables in Python

Variables in Python are named containers that store data values, acting as references to objects in memory. They allow you to store and manipulate data such as numbers, strings, or lists.

```python
 # Example: Storing different types of data
name = "Alice"  # String
age = 25        # Integer
height = 5.6    # Float
print(name, age, height)
```

Variable Naming Rules and Conventions

Python enforces specific rules and recommends conventions for naming variables:

- **Rules:**
  - Names must start with a letter or underscore (`_`).
  - Names can include letters, numbers, and underscores.
  - Names are case-sensitive (e.g., `age` and `Age` are distinct).
  - Cannot use Python keywords (e.g., `if` , `for`).
- **Conventions (PEP 8):**
  - Use lowercase with underscores (e.g., `user_name`).
  - Avoid single-character names except for counters (e.g., `i` in loops).
  - Choose descriptive names for clarity.

```python
 # Valid variable names
user_name = "Bob"
_count = 10
total_price2 = 99.99

# Invalid variable names
# 2price = 50  # Starts with a number
# my-var = 5   # Contains hyphen
```

Dynamic Typing in Python

Python uses dynamic typing, so you don’t need to declare a variable’s type, and its type can change during execution.

```python
 # Example: Variable type changes
x = 10        # x is an integer
print(type(x))  #
x = "Hello"   # x is now a string
print(type(x))  #
```

Variable Assignment and Reassignment

Variables are assigned using the `=` operator. Reassignment allows changing a variable’s value or type.

```python
 # Assignment
score = 100
print(score)  # Output: 100
```

```
# Reassignment
score = 200
print(score)  # Output: 200

# Multiple assignments
a, b, c = 1, 2, 3
print(a, b, c)  # Output: 1 2 3
```

Scope of Variables (Local vs Global)

Variable scope defines where a variable is accessible:

- **Local:** Defined inside a function, accessible only within it.
- **Global:** Defined outside functions, accessible everywhere.
- Use the `global` keyword to modify a global variable inside a function.

```python
 # Global variable
x = 10

def my_function():
    # Local variable
    y = 5
    print(f"Inside function: x = {x}, y = {y}")

my_function()
print(f"Outside function: x = {x}")  # y is not accessible here

# Modifying global variable
def update_global():
    global x
    x = 20
    print(f"Updated x inside function: {x}")

update_global()
print(f"Outside function: x = {x}")
```

Challenging Questions

Question 1: What will be the output of the following code?

```python
 a = 10
b = a
a = 20
print(b)
```

Show Answer

Question 2: What is wrong with this code, and how can it be fixed?

```python
 def increment():
    counter += 1

counter = 0
increment()
print(counter)
```

Show Answer

Question 3: Predict the output of this code:

```python
 x = 5
def outer():
    x = 10
    def inner():
        nonlocal x
        x = 15
    inner()
    print(x)

outer()
print(x)
```

Show Answer

1.3 Data Types

Python Data Types

Integers (int): Properties and Operations

Integers are whole numbers without a fractional part, with unlimited precision in Python. They support arithmetic operations like addition, subtraction, multiplication, and division.

```python
# Integer operations
a = 10
b = 3
print(a + b)  # Output: 13
print(a // b) # Output: 3 (floor division)
print(a ** b) # Output: 1000 (exponentiation)
```

Floats: Precision and Floating-Point Arithmetic

Floats represent numbers with decimal points. They are subject to floating-point precision issues due to IEEE 754 standards.

```python
# Float operations
x = 0.1 + 0.2
print(x)  # Output: 0.30000000000000004 (precision issue)
y = 3.14
print(round(y, 1))  # Output: 3.1
```

Strings (str): Creation, Methods, and Formatting

Strings are sequences of characters, created using single or double quotes. They are immutable and support various methods and formatting options.

```python
# String creation and methods
text = "Hello, Python!"
print(text.upper())  # Output: HELLO, PYTHON!
print(text[0:5])     # Output: Hello
name = "Alice"
print(f"Hi, {name}") # Output: Hi, Alice (f-string formatting)
```

Lists: Creation, Indexing, Slicing, and Mutability

Lists are ordered, mutable collections of items, allowing mixed data types. They support indexing, slicing, and modification.

```python
# List operations
fruits = ["apple", "banana", "cherry"]
print(fruits[1])      # Output: banana
fruits.append("date") # Add item
print(fruits[1:3])    # Output: ['banana', 'cherry'] (slicing)
fruits[0] = "avocado" # Modify item
print(fruits)         # Output: ['avocado', 'banana', 'cherry', 'date']
```

Tuples: Immutability and Use Cases

Tuples are ordered, immutable collections, often used for fixed data or as dictionary keys. They are created with parentheses.

```python
# Tuple operations
point = (3, 4)
print(point[0])  # Output: 3
# point[0] = 5   # Error: tuples are immutable
coords = (1, 2, 3)
x, y, z = coords # Unpacking
print(x, y, z)   # Output: 1 2 3
```

Dictionaries (dict): Key-Value Pairs and Operations

Dictionaries store key-value pairs, where keys are unique and immutable. They allow fast lookups and modifications.

```python
# Dictionary operations
person = {"name": "Bob", "age": 30}
print(person["name"])      # Output: Bob
person["city"] = "Paris"   # Add key-value pair
print(person.get("age"))   # Output: 30
del person["city"]         # Remove key-value pair
print(person)              # Output: {'name': 'Bob', 'age': 30}
```

Sets: Uniqueness, Set Operations (Union, Intersection)

Sets are unordered collections of unique items, ideal for membership testing and set operations like union and intersection.

```python
# Set operations
set1 = {1, 2, 3, 3}  # Duplicates ignored
set2 = {3, 4, 5}
print(set1)          # Output: {1, 2, 3}
print(set1 | set2)   # Output: {1, 2, 3, 4, 5} (union)
print(set1 & set2)   # Output: {3} (intersection)
set1.add(6)          # Add item
print(set1)          # Output: {1, 2, 3, 6}
```

Challenging Questions

Question 1: What will be the output of the following code, and why?

```python
a = [1, 2, 3]
b = a
b[0] = 10
print(a)
```

Show Answer

Question 2: Why does this code produce unexpected output, and how can it be fixed to add 0.3 accurately?

```python
x = 0.1 + 0.2
print(x)  # Output: 0.30000000000000004
```

Show Answer

Question 3: What will be the output of this code, and why?

```python
dict1 = {"a": 1, "b": 2}
dict2 = dict1.copy()
dict2["a"] = 10
print(dict1)
```

Show Answer

1.4 Operators

Python Operators

Arithmetic Operators (+, -, *, /, //, %, **)

Arithmetic operators perform mathematical operations on numbers (integers or floats).

```python
 a = 10
b = 3
print(a + b)   # Output: 13 (addition)
print(a - b)   # Output: 7 (subtraction)
print(a * b)   # Output: 30 (multiplication)
print(a / b)   # Output: 3.3333333333333335 (division)
print(a // b)  # Output: 3 (floor division)
print(a % b)   # Output: 1 (modulus)
print(a ** b)  # Output: 1000 (exponentiation)
```

Comparison Operators (==, !=, >, <, >=, <=)

Comparison operators compare values and return a boolean (`True` or `False`).

```python
 x = 5
y = 10
print(x == y)   # Output: False (equal to)
print(x != y)   # Output: True (not equal to)
print(x > y)    # Output: False (greater than)
print(x < y)    # Output: True (less than)
print(x >= 5)   # Output: True (greater than or equal to)
print(y <= 10)  # Output: True (less than or equal to)
```

Logical Operators (and, or, not)

Logical operators combine boolean expressions to evaluate conditions.

```python
 a = True
b = False
print(a and b)  # Output: False (both must be True)
print(a or b)   # Output: True (at least one must be True)
print(not a)    # Output: False (negates the boolean)
```

Bitwise Operators (&, |, ^, ~, <<, >>)

Bitwise operators manipulate individual bits of integers.

```python
 x = 10  # Binary: 1010
y = 4   # Binary: 0100
print(x & y)   # Output: 0 (AND, 0000)
print(x | y)   # Output: 14 (OR, 1110)
print(x ^ y)   # Output: 14 (XOR, 1110)
print(~x)      # Output: -11 (NOT, inverts bits)
print(x << 2)  # Output: 40 (left shift, 101000)
```

```
print(x >> 2)  # Output: 2 (right shift, 0010)
```

Assignment Operators (=, +=, -=, etc.)

Assignment operators assign values to variables, often combining with arithmetic or bitwise operations.

```Python
x = 5
x += 3  # Equivalent to x = x + 3
print(x)  # Output: 8
x *= 2  # Equivalent to x = x * 2
print(x)  # Output: 16
x //= 4 # Equivalent to x = x // 4
print(x)  # Output: 4
```

Membership and Identity Operators (in, not in, is, is not)

Membership operators check for presence in sequences, while identity operators compare object identity.

```Python
fruits = ["apple", "banana"]
print("apple" in fruits)      # Output: True
print("orange" not in fruits) # Output: True

a = [1, 2]
b = [1, 2]
c = a
print(a is b)      # Output: False (different objects)
print(a is c)      # Output: True (same object)
print(a is not b)  # Output: True
```

Challenging Questions

Question 1: What will be the output of the following code, and why?

```Python
x = 5
y = 2
print(x ** y % 3)
```

Show Answer

Question 2: Why does this code produce an unexpected result, and how can it be fixed to evaluate the condition correctly?

```Python
x = 10
if x > 5 or 15:
    print("Condition met")
else:
    print("Condition not met")
```

Show Answer

Question 3: What will be the output of this code, and why?

```python
 a = 12  # Binary: 1100
b = 10  # Binary: 1010
print(a & b)
print(a | b)
```

Show Answer

1.5 Control Flows

Python Control Flow

Conditional Statements (if, elif, else)

Conditional statements allow you to execute code based on conditions using `if`, `elif`, and `else`.

```python
# Conditional example
score = 85
if score >= 90:
    print("Grade: A")
elif score >= 80:
    print("Grade: B")
else:
    print("Grade: C")
# Output: Grade: B
```

Loops: for and while Loops

Loops iterate over sequences (`for`) or until a condition is met (`while`).

```python
# For loop
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
# Output: apple
#         banana
#         cherry

# While loop
count = 1
while count <= 3:
    print(count)
    count += 1
# Output: 1
#         2
#         3
```

Break, Continue, and Pass Statements

`break` exits a loop, `continue` skips to the next iteration, and `pass` is a placeholder that does nothing.

```python
# Break and continue
for i in range(5):
    if i == 3:
        break
    if i == 1:
        continue
    print(i)
# Output: 0
#         2
```

```
# Pass
for i in range(2):
    pass  # Placeholder, no action
```

Nested Control Structures

Nested structures combine conditionals and loops within each other for complex logic.

```Python
 # Nested loop and conditional
for i in range(3):
    if i % 2 == 0:
        for j in range(2):
            print(f"i={i}, j={j}")
# Output: i=0, j=0
#         i=0, j=1
#         i=2, j=0
#         i=2, j=1
```

Exception Handling (try, except, finally)

Exception handling manages errors using `try`, `except`, and `finally` blocks.

```Python
 # Exception handling
try:
    num = int(input("Enter a number: "))
    print(10 / num)
except ValueError:
    print("Invalid input, please enter a number")
except ZeroDivisionError:
    print("Cannot divide by zero")
finally:
    print("Execution complete")
# Example input: 0
# Output: Cannot divide by zero
#         Execution complete
```

Logical and Comparison Operators

Logical operators (`and`, `or`, `not`) and comparison operators (`==`, `!=`, `>`, etc.) evaluate conditions.

```Python
 # Logical and comparison operators
x = 10
if x > 5 and x < 15:
    print("x is between 5 and 15")
# Output: x is between 5 and 15
print(not x == 10)  # Output: False
```

Comprehensions (List, Dictionary)

Comprehensions provide a concise way to create lists or dictionaries using loops and conditionals.

```Python
 # List comprehension
squares = [x**2 for x in range(5)]
print(squares)  # Output: [0, 1, 4, 9, 16]
```

```
# Dictionary comprehension
numbers = {x: x*2 for x in range(3)}
print(numbers)  # Output: {0: 0, 1: 2, 2: 4}
```

Challenging Questions

Question 1: What will be the output of the following code, and why?

```python
for i in range(5):
    if i % 2 == 0:
        print(i)
    else:
        continue
```

Show Answer

Question 2: What is wrong with this code, and how can it be fixed to handle the error?

```python
numbers = [1, 2, "3", 4]
total = 0
for num in numbers:
    total += num
print(total)
```

Show Answer

Question 3: What will be the output of this code, and why?

```python
evens = [x for x in range(10) if x % 2 == 0]
print(evens)
```

Show Answer

1.6 Functions

Python Functions & Lambda Functions

Defining and Calling Functions

Functions are defined using the `def` keyword and called by their name followed by parentheses.

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

print(greet("Alice"))  # Output: Hello, Alice!
```

Function Parameters (Positional, Keyword, Default)

Functions can take positional, keyword, and default parameters for flexibility.

```python
def calculate_total(price, tax=0.05):
    return price * (1 + tax)

# Positional
print(calculate_total(100))       # Output: 105.0
# Keyword
print(calculate_total(price=100, tax=0.1))  # Output: 110.0
# Default
print(calculate_total(200))       # Output: 210.0
```

Return Statements and Multiple Returns

Functions use `return` to send back values. Multiple values can be returned as a tuple.

```python
def divide_and_remainder(a, b):
    quotient = a // b
    remainder = a % b
    return quotient, remainder

q, r = divide_and_remainder(10, 3)
print(q, r)  # Output: 3 1
```

Variable Scope in Functions

Variables defined inside a function are local; global variables can be accessed but modified only with the `global` keyword.

```python
x = 10
def modify():
    global x
    x = 20
    y = 5  # Local
    print(f"Inside: x={x}, y={y}")

modify()
```

```
print(f"Outside: x={x}")  # Output: Outside: x=20
```

Lambda Functions: Syntax and Use Cases

Lambda functions are anonymous, single-expression functions defined with the `lambda` keyword, often used for short operations.

```Python
square = lambda x: x ** 2
print(square(5))  # Output: 25

# Sorting with lambda
pairs = [(1, 'one'), (3, 'three'), (2, 'two')]
pairs.sort(key=lambda x: x[1])
print(pairs)  # Output: [(1, 'one'), (3, 'three'), (2, 'two')]
```

Higher-Order Functions (map, filter, reduce)

Higher-order functions like `map`, `filter`, and `reduce` take functions as arguments for processing iterables.

```Python
from functools import reduce

# Map: Apply function to each item
numbers = [1, 2, 3]
squares = list(map(lambda x: x ** 2, numbers))
print(squares)  # Output: [1, 4, 9]

# Filter: Select items based on condition
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # Output: [2]

# Reduce: Combine items
sum_all = reduce(lambda x, y: x + y, numbers)
print(sum_all)  # Output: 6
```

Challenging Questions

Question 1: What will be the output of the following code, and why?

```Python
def outer():
    x = 10
    def inner():
        nonlocal x
        x += 5
        return x
    return inner()

print(outer())  # Output: 15
```

Show Answer

Question 2: Why does this code produce an error, and how can it be fixed?

```Python
def multiply(x, y=2):
```

```
    return x * y

print(multiply(y=3, 5))
```

Show Answer

Question 3: What will be the output of this code, and why?

```python
 numbers = [1, 2, 3, 4]
result = list(map(lambda x: x * 2, filter(lambda x: x % 2 == 0, numbers)))
print(result)
```

Show Answer

1.7 Lists and Tuples

Python Lists and Tuples

Lists: Creating Lists

Lists are ordered, mutable collections created using square brackets `[]` or the `list()` constructor.

```python
fruits = ["apple", "banana", "cherry"]
numbers = list(range(3))
print(fruits)   # Output: ['apple', 'banana', 'cherry']
print(numbers)  # Output: [0, 1, 2]
```

Lists: Accessing Elements via Indexing and Slicing

Access list elements using zero-based indexing or slicing to extract sublists.

```python
fruits = ["apple", "banana", "cherry", "date"]
print(fruits[1])      # Output: banana
print(fruits[-1])     # Output: date (last element)
print(fruits[1:3])    # Output: ['banana', 'cherry']
print(fruits[:2])     # Output: ['apple', 'banana']
```

Lists: Modifying Lists (Mutability)

Lists are mutable, allowing changes to elements, addition, or removal of items.

```python
fruits = ["apple", "banana", "cherry"]
fruits[0] = "avocado"  # Modify element
fruits.append("date")  # Add to end
print(fruits)          # Output: ['avocado', 'banana', 'cherry', 'date']
```

Lists: Common List Methods

Lists have methods like `append`, `remove`, `pop`, and more for manipulation.

```python
numbers = [1, 2, 3, 2]
numbers.append(4)      # Add 4 to end
numbers.remove(2)      # Remove first 2
popped = numbers.pop() # Remove and return last item (4)
numbers.sort()         # Sort in place
print(numbers)         # Output: [1, 3]
print(popped)          # Output: 4
```

Tuples: Creating Tuples

Tuples are ordered, immutable collections created using parentheses `()` or the `tuple()` constructor.

```python
point = (3, 4)
colors = tuple(["red", "blue"])
print(point)   # Output: (3, 4)
```

```
print(colors)  # Output: ('red', 'blue')
```

Tuples: Accessing Elements via Indexing and Slicing

Like lists, tuples support indexing and slicing to access elements.

```python
coords = (10, 20, 30, 40)
print(coords[0])      # Output: 10
print(coords[-2])     # Output: 30
print(coords[1:3])    # Output: (20, 30)
```

Tuples: Immutability and Its Implications

Tuples cannot be modified after creation, ensuring data integrity but limiting flexibility.

```python
point = (1, 2)
# point[0] = 3  # Error: tuples are immutable
new_point = point + (3,)  # Create new tuple
print(new_point)          # Output: (1, 2, 3)
```

Tuples: Use Cases

Tuples are used for fixed data, as dictionary keys, or when immutability is desired.

```python
# Tuple as dictionary key
locations = {(0, 0): "Origin", (1, 2): "Point A"}
print(locations[(1, 2)])  # Output: Point A

# Fixed data
rgb = (255, 128, 0)
r, g, b = rgb  # Unpacking
print(r, g, b)  # Output: 255 128 0
```

Challenging Questions

Question 1: What will be the output of the following code, and why?

```python
lst = [1, 2, 3]
lst2 = lst
lst.append(4)
print(lst2)
```

Show Answer

Question 2: Why does this code raise an error, and how can it be fixed to modify the tuple?

```python
tup = (1, 2, 3)
tup[1] = 4
```

Show Answer

Question 3: What will be the output of this code, and why?

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

Show Answer

1.8 String and Dictionaries

Python Strings and Dictionaries

Strings: Creating Strings

Strings are created using single quotes `'`, double quotes `"`, or triple quotes `'''` for multiline strings.

```python
single = 'Hello'
double = "World"
multiline = '''Line 1
Line 2'''
print(single)     # Output: Hello
print(multiline)  # Output: Line 1
                  #         Line 2
```

Strings: Accessing Characters via Indexing and Slicing

Strings support zero-based indexing and slicing to access characters or substrings.

```python
text = "Python"
print(text[0])    # Output: P
print(text[-1])   # Output: n
print(text[1:4])  # Output: yth
print(text[:3])   # Output: Pyt
```

Strings: Immutability

Strings are immutable, meaning their characters cannot be changed after creation.

```python
text = "Python"
# text[0] = "J"  # Error: strings are immutable
new_text = "J" + text[1:]  # Create new string
print(new_text)  # Output: Jython
```

Strings: Common String Methods

Strings offer methods like `upper`, `lower`, `strip`, `split`, and `join` for manipulation.

```python
text = "  Hello, World!  "
print(text.upper())       # Output:   HELLO, WORLD!
print(text.strip())       # Output: Hello, World!
print(text.split(","))    # Output: ['  Hello', ' World!  ']
words = ["Python", "is", "fun"]
print(" ".join(words))    # Output: Python is fun
```

Strings: String Formatting

Strings can be formatted using f-strings, the `format()` method, or the `%` operator.

```python
name = "Alice"
```

```
age = 25
print(f"{name} is {age}")           # f-string: Alice is 25
print("{} is {}".format(name, age)) # format(): Alice is 25
print("%s is %d" % (name, age))    # % operator: Alice is 25
```

Dictionaries: Creating Dictionaries

Dictionaries are unordered collections of key-value pairs, created using curly braces `{}` or `dict()`.

Python
```
 person = {"name": "Bob", "age": 30}
scores = dict(math=90, science=85)
print(person)   # Output: {'name': 'Bob', 'age': 30}
print(scores)   # Output: {'math': 90, 'science': 85}
```

Dictionaries: Accessing and Modifying Key-Value Pairs

Access values using keys; modify or add key-value pairs directly.

Python
```
 person = {"name": "Bob", "age": 30}
print(person["name"])      # Output: Bob
person["age"] = 31         # Modify value
person["city"] = "Paris"   # Add new pair
print(person)              # Output: {'name': 'Bob', 'age': 31, 'city': 'Paris'}
```

Dictionaries: Dictionary Methods

Dictionary methods like `get`, `keys`, `values`, `items`, and `pop` facilitate operations.

Python
```
 person = {"name": "Bob", "age": 30}
print(person.get("city", "Unknown"))  # Output: Unknown
print(person.keys())                  # Output: dict_keys(['name', 'age'])
print(person.values())                # Output: dict_values(['Bob', 30])
print(person.items())                 # Output: dict_items([('name', 'Bob'), ('age', 30)])
person.pop("age")                     # Remove key-value pair
print(person)                         # Output: {'name': 'Bob'}
```

Dictionaries: Immutability for Keys

Dictionary keys must be immutable (e.g., strings, numbers, tuples), but values can be any type.

Python
```
 coords = {(0, 0): "Origin", (1, 2): "Point A"}
print(coords[(0, 0)])  # Output: Origin
# Invalid: lists cannot be keys
# d = {[1, 2]: "Invalid"}  # Error: unhashable type: 'list'
```

Dictionaries: Iteration over Dictionaries

Iterate over keys, values, or key-value pairs using loops.

Python
```
 scores = {"math": 90, "science": 85}
for subject in scores:
    print(subject)  # Output: math
                    #         science
for subject, score in scores.items():
    print(f"{subject}: {score}")
```

```
# Output: math: 90
#         science: 85
```

Challenging Questions

Question 1: What will be the output of the following code, and why?

```python
 text = "python"
new_text = text[:2] + text[2].upper() + text[3:]
print(new_text)
```

Show Answer

Question 2: Why does this code raise an error, and how can it be fixed to access the value safely?

```python
 d = {"name": "Alice"}
print(d["age"])
```

Show Answer

Question 3: What will be the output of this code, and why?

```python
 d = {"a": 1, "b": 2}
d2 = d
d2["a"] = 10
print(d)
```

Show Answer

1.9 Sets

Python Sets

Creating Sets

Sets are unordered collections of unique elements, created using curly braces `{}` or the `set()` constructor.

```python
 numbers = {1, 2, 3}
fruits = set(["apple", "banana", "apple"])
print(numbers)  # Output: {1, 2, 3}
print(fruits)   # Output: {'apple', 'banana'}
```

Uniqueness Property

Sets automatically remove duplicates, ensuring each element appears only once.

```python
 items = {1, 1, 2, 2, 3}
print(items)  # Output: {1, 2, 3}
mixed = set([1, "hello", 1, "hello"])
print(mixed)  # Output: {1, 'hello'}
```

Set Operations

Sets support operations like union, intersection, difference, and symmetric difference.

```python
 set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1 | set2)  # Union: {1, 2, 3, 4, 5}
print(set1 & set2)  # Intersection: {3}
print(set1 - set2)  # Difference: {1, 2}
print(set1 ^ set2)  # Symmetric difference: {1, 2, 4, 5}
```

Common Set Methods

Sets provide methods like `add`, `remove`, `discard`, and `pop` for manipulation.

```python
 numbers = {1, 2, 3}
numbers.add(4)        # Add element
print(numbers)        # Output: {1, 2, 3, 4}
numbers.remove(2)     # Remove element (raises KeyError if not found)
print(numbers)        # Output: {1, 3, 4}
numbers.discard(5)    # Remove if present, no error
popped = numbers.pop() # Remove and return arbitrary element
print(popped)         # Output: e.g., 1
print(numbers)        # Output: e.g., {3, 4}
```

Membership Testing and Iteration

Sets are optimized for membership testing and can be iterated over.

```python
fruits = {"apple", "banana", "cherry"}
print("apple" in fruits)  # Output: True
print("orange" not in fruits)  # Output: True
for fruit in fruits:
    print(fruit)
# Output: apple
#         banana
#         cherry
```

Interesting Programs Using Sets

Program 1: Duplicate Remover

This program takes a list of user inputs and removes duplicates using a set.

```python
def remove_duplicates():
    user_input = input("Enter items (space-separated): ").split()
    unique_items = set(user_input)
    return list(unique_items)

result = remove_duplicates()
print("Unique items:", result)
# Example input: apple apple banana cherry
# Output: Unique items: ['apple', 'banana', 'cherry']
```

Program 2: Common Elements Finder

This program finds common elements between two lists using set intersection.

```python
def find_common_elements(list1, list2):
    return list(set(list1) & set(list2))

list1 = [1, 2, 3, 4]
list2 = [3, 4, 5, 6]
common = find_common_elements(list1, list2)
print("Common elements:", common)  # Output: Common elements: [3, 4]
```

Program 3: Unique Word Counter

This program counts unique words in a sentence by converting it to a set.

```python
def count_unique_words(sentence):
    words = sentence.lower().split()
    unique_words = set(words)
    return len(unique_words)

sentence = "The cat and the dog and cat"
count = count_unique_words(sentence)
print("Unique word count:", count)  # Output: Unique word count: 4
```

Challenging Questions

> **Question 1:** What will be the output of the following code, and why?

```python
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5}
print(set1 - set2)
print(set2 - set1)
```

Show Answer

Question 2: Why does this code raise an error, and how can it be fixed?

```python
numbers = {1, 2, 3}
numbers.remove(4)
```

Show Answer

Question 3: What will be the output of this code, and why?

```python
set1 = {1, 2, 3}
set2 = set1.copy()
set2.add(4)
print(set1)
print(set2)
```

Show Answer

Questions Level-01

Python Assignments: Strings, Dictionaries, and Sets

Level 1 Assignments

Below are five beginner-friendly assignments on Strings, Dictionaries, and Sets. Each includes a creative twist to make them engaging. Try solving them to practice your Python skills!

Assignment 1: Secret Message Decoder (Strings)

Description: Write a program that takes a secret message as input and decodes it by extracting every second character starting from the first one (index 0). The message is guaranteed to be at least 2 characters long.

Twist: If the decoded message contains the word "hello" (case-insensitive), print "Special Greeting: Hello found!"; otherwise, print the decoded message.

Example:

- Input: "hpealnlod" → Output: "Special Greeting: Hello found!"
- Input: "sptayctkoy" → Output: "saty"

Hint: Use string slicing and the `lower()` method.

Assignment 2: Treasure Map Coordinates (Dictionaries)

Description: Create a program that stores a treasure map as a dictionary with tuple coordinates (x, y) as keys and location names as values. The program should take a coordinate as input (e.g., "1,2") and print the location name if it exists, or "Unknown location" if it doesn't.

Twist: If the coordinates form a square (x == y), print "Treasure Chest!" instead of the location name.

Example:

- Input: "1,2" → Output: "Cave"
- Input: "3,3" → Output: "Treasure Chest!"
- Input: "4,5" → Output: "Unknown location"

Hint: Use tuple keys and the `get()` method.

```python
treasure_map = {(1, 2): "Cave", (3, 3): "Island", (0, 1): "Beach"}
```

Assignment 3: Unique Potion Ingredients (Sets)

Description: Write a program that takes a space-separated list of potion ingredients as input and prints the unique ingredients in sorted order.

Twist: If the number of unique ingredients is exactly 3, print "Perfect Potion!" before the list.

Example:

- Input: "herb root herb leaf" → Output: Perfect Potion! ['herb', 'leaf', 'root']
- Input: "root root leaf" → Output: ['leaf', 'root']

Hint: Use a set to remove duplicates and convert to a sorted list.

Assignment 4: Word Mixer Formatter (Strings)

Description: Write a program that takes two words as input and combines them by alternating their characters (e.g., "cat" and "dog" → "cdaotg"). If the words are of different lengths, append the remaining characters of the longer word.

Twist: Format the output as an f-string in the form "Mixed word: {result}" and convert the result to uppercase.

Example:

- Input: "cat dog" → Output: Mixed word: CDAOTG
- Input: "hi there" → Output: Mixed word: HTIHERE

Hint: Use string indexing and loops.

Assignment 5: Common Friends Finder (Sets)

Description: Write a program that takes two space-separated lists of names (friends of two people) and finds the common friends using set intersection. Print the common friends in sorted order.

Twist: If there are no common friends, print "No common friends, time to introduce them!"

Example:

- Input: "Alice Bob Charlie" and "Bob Charlie Dave" → Output: ['Bob', 'Charlie']
- Input: "Alice Bob" and "Charlie Dave" → Output: No common friends, time to introduce them!

Hint: Use `set().intersection()` or the `&` operator.

Questions Level-02

Python Assignments: Strings, Dictionaries, and Sets (Level 2)

Level 2 Assignments

Below are five intermediate-level assignments on Strings, Dictionaries, and Sets. These challenges combine multiple concepts, handle edge cases, and include creative twists to test your Python skills. Perfect for learners comfortable with basic Python!

Assignment 1: Palindrome Sentence Analyzer (Strings)

Description: Write a program that takes a sentence as input and determines if it forms a palindrome when considering only alphabetic characters (ignoring spaces, punctuation, and case). Return the cleaned string (lowercase, alphabetic characters only) and whether it's a palindrome.

Twist: If the cleaned string is a palindrome and has more than 5 characters, append " - Epic Palindrome!" to the output.

Example:

- Input: "A man, a plan, a canal: Panama" → Output: "amanaplanacanalpanama is a palindrome - Epic Palindrome!"
- Input: "race a car" → Output: "raceacar is not a palindrome"
- Input: "deked" → Output: "deked is a palindrome"

Hint: Use `lower()`, `isalpha()`, and slicing.

Assignment 2: Inventory Manager (Dictionaries)

Description: Create a program that manages an inventory dictionary with item names as keys and quantities as values. The program takes a space-separated input of item names and updates the inventory: add 1 to the quantity if the item exists, or add the item with quantity 1 if it doesn't. Handle invalid inputs (e.g., empty strings).

Twist: If an item's quantity reaches exactly 5, mark it as "Fully Stocked" in the output instead of showing the number.

Example:

- Input: "apple banana apple" (initial inventory: {"apple": 3, "orange": 4}) → Output: {"apple": 5, "orange": 4, "banana": 1} → "apple: Fully Stocked, orange: 4, banana: 1"
- Input: " banana" → Output: "Invalid input: empty item name"

Hint: Use `get()` and dictionary iteration.

```python
inventory = {"apple": 3, "orange": 4}
```

Assignment 3: Set-Based Anagram Grouper (Sets)

Description: Write a program that takes a space-separated list of words and groups them by their anagrams using sets. Output a dictionary where keys are sorted character strings (representing the anagram) and values are sets of words that are anagrams of each other.

Twist: If a group has exactly two anagrams, prefix the group's output with "Twin Anagrams: ".

Example:

- Input: "cat act dog god" → Output: {"act": Twin Anagrams: {"cat", "act"}, "dgo": Twin Anagrams: {"dog", "god"}}
- Input: "cat hat mat" → Output: {"aht": {"hat"}, "amt": {"mat"}, "act": {"cat"}}

Hint: Use `sorted()` on word characters to create a key and sets to store anagrams.

Assignment 4: Custom String Encoder (Strings)

Description: Write a program that takes a string and encodes it by replacing each vowel (a, e, i, o, u, case-insensitive) with its position in the string (1-based indexing). Non-vowel characters remain unchanged.

Twist: If the encoded string contains three or more digits in a row, append " - Number Overload!" to the output.

Example:

- Input: "hello" → Output: "h2ll4 - Number Overload!" (e→2, o→5)
- Input: "python" → Output: "p2th3n" (o→2, y→3)

Hint: Use string iteration, `lower()`, and check for consecutive digits.

Assignment 5: Social Network Analyzer (Sets)

Description: Create a program that takes a dictionary of users and their friend sets, and a pair of user names as input. Output the union, intersection, and symmetric difference of their friend sets. Handle cases where a user doesn't exist.

Twist: If the intersection is empty, suggest a mutual friend from the symmetric difference (if any) to connect them.

Example:

- Input: "Alice Bob" (network: {"Alice": {"Charlie", "Dave"}, "Bob": {"Dave", "Eve"}}) → Output: Union: {"Charlie", "Dave", "Eve"}, Intersection: {"Dave"}, Symmetric Difference: {"Charlie", "Eve"}
- Input: "Alice Charlie" (network: {"Alice": {"Dave"}, "Charlie": {"Eve"}}) → Output: Union: {"Dave", "Eve"}, Intersection: {}, Symmetric Difference: {"Dave", "Eve"} - Suggest mutual friend: Dave

Hint: Use set operations (`|` , `&` , `^`) and `get()` .

```python
network = {"Alice": {"Charlie", "Dave"}, "Bob": {"Dave", "Eve"}, "Charlie": {"Eve"}}
```

2.1 File Handling

Python File Handling

Opening and Closing Files

Using the open() Function

The `open()` function opens a file in a specified mode, such as read, write, or append, and supports text or binary modes.

```Python
file = open("example.txt", "r")  # Open for reading
file.close()  # Always close the file
```

File Modes

Common file modes include: `r` (read), `w` (write, overwrites), `a` (append), `r+` (read and write), `wb` (write binary).

```Python
file = open("data.bin", "wb")  # Write binary mode
file.close()
file = open("log.txt", "a")    # Append mode
file.close()
```

Properly Closing Files with close()

Always close files using `close()` to free system resources, but errors can leave files open.

```Python
file = open("example.txt", "r")
content = file.read()
file.close()  # Ensure file is closed
```

Using the with Statement

The `with` statement automatically closes files, even if an error occurs, making it the preferred method.

```Python
with open("example.txt", "r") as file:
    content = file.read()
# File is automatically closed after the block
print("File closed:", file.closed)  # Output: File closed: True
```

Reading Files

Reading Entire Files with read()

The `read()` method reads the entire file content into a string.

```Python
with open("example.txt", "r") as file:
    content = file.read()
print(content)  # Output: Entire file content
```

Reading Line by Line with readline() or readlines()

`readline()` reads one line, while `readlines()` returns a list of all lines.

```python
with open("example.txt", "r") as file:
    line1 = file.readline()  # First line
    lines = file.readlines() # Remaining lines as list
print(line1.strip())  # Output: First line
print(lines)          # Output: ['Second line\n', 'Third line\n']
```

Iterating Over Files Line by Line

Iterate directly over a file object to read lines efficiently.

```python
with open("example.txt", "r") as file:
    for line in file:
        print(line.strip())  # Output: Each line without trailing newline
```

Handling Large Files Efficiently

For large files, use line-by-line iteration or `read(size)` to avoid loading the entire file into memory.

```python
with open("large_file.txt", "r") as file:
    while True:
        chunk = file.read(1024)  # Read 1024 bytes at a time
        if not chunk:
            break
        print(chunk, end="")
```

Writing to Files

Writing Strings with write()

The `write()` method writes a string to a file; it doesn't add newlines automatically.

```python
with open("output.txt", "w") as file:
    file.write("Hello, World!\n")
# File now contains: Hello, World!
```

Writing Multiple Lines with writelines()

`writelines()` writes a list of strings to a file without adding newlines.

```python
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
with open("output.txt", "w") as file:
    file.writelines(lines)
# File contains: Line 1
#                Line 2
#                Line 3
```

Appending to Existing Files

Use `a` mode to append data to the end of a file without overwriting.

```python
with open("log.txt", "a") as file:
    file.write("New entry\n")
# Appends "New entry" to the file
```

Overwriting vs. Appending

`w` mode overwrites the file, creating a new one if it doesn't exist, while `a` mode adds to the end.

```python
with open("data.txt", "w") as file:
    file.write("Overwritten\n")  # Replaces file content
with open("data.txt", "a") as file:
    file.write("Appended\n")     # Adds to existing content
# File contains: Overwritten
#                Appended
```

Challenging Questions

Question 1: What will be the output of the following code, and why?

```python
with open("test.txt", "w") as file:
    file.write("First line\nSecond line")
with open("test.txt", "r") as file:
    print(file.read(10))
```

Show Answer

Question 2: Why does this code raise an error, and how can it be fixed to append safely?

```python
file = open("log.txt", "r")
file.write("New log entry\n")
file.close()
```

Show Answer

Question 3: What will be the output of this code, and why?

```python
with open("data.txt", "w") as file:
    file.writelines(["A\n", "B\n"])
with open("data.txt", "a") as file:
    file.write("C\n")
with open("data.txt", "r") as file:
    lines = file.readlines()
print(len(lines))
```

Show Answer

2.2 Paths and File Operations

Python Advanced File Handling

Working with File Paths and Directories

Using the os and pathlib Modules

The `os` module provides functions for file and directory operations, while `pathlib` offers an object-oriented approach for path manipulation.

```Python
 import os
import pathlib

# os: Get current directory
print(os.getcwd())  # Output: Current working directory path

# pathlib: Create a path object
path = pathlib.Path("documents/data.txt")
print(path.name)    # Output: data.txt
```

Checking if Files/Directories Exist

Use `os.path.exists()` or `pathlib.Path.exists()` to check if a file or directory exists.

```Python
 import os
import pathlib

# os
print(os.path.exists("example.txt"))  # Output: True/False

# pathlib
path = pathlib.Path("example.txt")
print(path.exists())                  # Output: True/False
```

Creating, Renaming, and Deleting Files/Directories

Create directories with `os.makedirs()` or `pathlib.Path.mkdir()`, rename with `os.rename()`, and delete with `os.remove()` or `pathlib.Path.unlink()`.

```Python
 import os
import pathlib

# Create directory
os.makedirs("new_folder", exist_ok=True)

# Rename file
os.rename("old.txt", "new.txt")

# Delete file
path = pathlib.Path("new.txt")
if path.exists():
    path.unlink()  # Delete file
```

Handling Cross-Platform Path Compatibility

Use `os.path.join()` or `pathlib.Path` to create paths compatible across Windows, Linux, and macOS.

```Python
import os
import pathlib

# os
path = os.path.join("folder", "subfolder", "file.txt")
print(path)  # Output: folder/subfolder/file.txt (or folder\subfolder\file.txt on Windows)

# pathlib
path = pathlib.Path("folder") / "subfolder" / "file.txt"
print(path)  # Output: folder/subfolder/file.txt
```

Error Handling in File Operations

Common File-Related Exceptions

Common exceptions include `FileNotFoundError`, `PermissionError`, and `IsADirectoryError`.

```Python
try:
    with open("nonexistent.txt", "r") as file:
        content = file.read()
except FileNotFoundError:
    print("File not found!")
except PermissionError:
    print("Permission denied!")
```

Using try-except to Handle File Errors

Wrap file operations in `try-except` blocks to handle errors gracefully.

```Python
import os

try:
    os.remove("protected.txt")
except FileNotFoundError:
    print("File does not exist.")
except PermissionError:
    print("Cannot delete: Permission denied.")
```

Best Practices for Robust File Handling

Always use the `with` statement, check file existence before operations, handle specific exceptions, and use `pathlib` for cross-platform compatibility.

```Python
from pathlib import Path

path = Path("data.txt")
try:
    if path.exists():
        with path.open("r") as file:
            content = file.read()
    else:
        print("File not found.")
except Exception as e:
```

```
    print(f"Error: {e}")
```

Working with Different File Formats

Reading and Writing Text Files

Text files are read and written using standard file operations with encoding (default is UTF-8).

```python
from pathlib import Path

# Write text
with Path("notes.txt").open("w", encoding="utf-8") as file:
    file.write("Hello, Python!\n")

# Read text
with Path("notes.txt").open("r", encoding="utf-8") as file:
    content = file.read()
print(content)  # Output: Hello, Python!
```

Handling CSV Files (using csv module)

The `csv` module simplifies reading and writing CSV files.

```python
import csv

# Write CSV
with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age"])
    writer.writerow(["Alice", 25])

# Read CSV
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
# Output: ['Name', 'Age']
#         ['Alice', '25']
```

Working with JSON Files (using json module)

The `json` module handles serialization and deserialization of JSON data.

```python
import json

# Write JSON
data = {"name": "Bob", "scores": [90, 85]}
with open("data.json", "w") as file:
    json.dump(data, file, indent=4)

# Read JSON
with open("data.json", "r") as file:
    loaded_data = json.load(file)
print(loaded_data)  # Output: {'name': 'Bob', 'scores': [90, 85]}
```

Basic Binary File Operations

Binary files (e.g., images) are read and written in binary mode (`rb` , `wb`).

```python
# Copy binary file
with open("image.png", "rb") as source:
    data = source.read()
with open("copy.png", "wb") as destination:
    destination.write(data)
```

Challenging Questions

Question 1: What will be the output of the following code, and why?

```python
import os

try:
    os.makedirs("test_folder")
    with open("test_folder/file.txt", "w") as file:
        file.write("Test")
except IsADirectoryError:
    print("Directory error")
except FileNotFoundError:
    print("File not found")
```

Show Answer

Question 2: Why does this code raise an error, and how can it be fixed to read the CSV correctly?

```python
import csv

with open("data.csv", "r") as file:
    reader = csv.reader(file)
    print(reader[0])
```

Show Answer

Question 3: What will be the output of this code, and why?

```python
import json
from pathlib import Path

data = {"name": "Alice", "age": 30}
path = Path("config.json")
with path.open("w") as file:
    json.dump(data, file)
with path.open("r") as file:
    loaded = json.load(file)
print(loaded["name"])
```

Show Answer

2.3 Handling big files

Python Advanced File Handling

Buffering and Its Advantages

Buffering improves I/O performance by reducing the number of system calls. Python provides control using the `open` function's `buffering` parameter.

```python
with open("log.txt", "w", buffering=8192) as file:
    for i in range(10000):
        file.write(f"Line {i}\n")
```

Reading Huge Files Efficiently

To avoid loading the entire file into memory, read line by line using a generator approach.

```python
with open("huge_file.txt", "r") as file:
    for line in file:
        process(line)
```

Writing to Huge Files

Use buffered writing and flush periodically to avoid memory issues and ensure timely disk writes.

```python
with open("output.txt", "w") as file:
    for i in range(1000000):
        file.write(f"Record {i}\n")
        if i % 10000 == 0:
            file.flush()
```

Rotation of Data (Log Rotation)

Log rotation avoids huge log files by moving them and starting a new one. You can use `shutil` for manual rotation.

```python
import shutil
import os

if os.path.exists("app.log"):
    shutil.move("app.log", "app.log.1")
```

```
with open("app.log", "w") as file:
    file.write("New session started\n")
```

Reading Binary Files

Use binary mode (`rb`) to read non-text files like images or audio data.

```Python
with open("sample.wav", "rb") as binary_file:
    data = binary_file.read()
    print(data[:20])  # Display first 20 bytes
```

Reading from Another Machine Using SSH/SCP

You can use the `paramiko` or `scp` module to access files from a remote server.

```Python
from paramiko import SSHClient
from scp import SCPClient

ssh = SSHClient()
ssh.load_system_host_keys()
ssh.connect("remote_host", username="user")

with SCPClient(ssh.get_transport()) as scp:
    scp.get("/remote/path/file.txt", "local_file.txt")

ssh.close()
```

2.4 Regular Expressions

Python Regular Expressions (RegEx)

Substituting and Splitting with RegEx

Using re.sub() for Pattern Replacement

The `re.sub()` function replaces matches of a pattern with a specified string, useful for text manipulation.

```python
import re

# Replace digits with 'X'
text = "My number is 12345"
result = re.sub(r'\d', 'X', text)
print(result)  # Output: My number is XXXXX
```

Using re.split() to Split Strings by Patterns

The `re.split()` function splits a string based on a regex pattern, allowing flexible delimiter handling.

```python
import re

# Split on multiple delimiters
text = "apple,banana;orange grape"
result = re.split(r'[,;\s]+', text)
print(result)  # Output: ['apple', 'banana', 'orange', 'grape']
```

Practical Examples

Common use cases include anonymizing emails or splitting strings with varied delimiters.

```python
import re

# Replace email addresses
text = "Contact: alice@example.com or bob@domain.com"
result = re.sub(r'\b[\w\.-]+@[\w\.-]+\.\w+\b', '[EMAIL]', text)
print(result)  # Output: Contact: [EMAIL] or [EMAIL]

# Split on multiple delimiters
text = "one,two;three four"
result = re.split(r'[,;\s]+', text)
print(result)  # Output: ['one', 'two', 'three', 'four']
```

Grouping and Capturing in RegEx

Creating Groups with Parentheses ()

Parentheses `()` define capture groups, allowing extraction of specific parts of a match.

```python
import re

# Extract area code from phone number
text = "Call: (123) 456-7890"
match = re.search(r'\((\d{3})\)', text)
if match:
    print(match.group(1))  # Output: 123
```

Accessing Captured Groups in Matches

Captured groups can be accessed via `match.group()` or iterated with `match.groups()`.

```python
import re

# Extract date parts
text = "Date: 2023-05-12"
match = re.search(r'(\d{4})-(\d{2})-(\d{2})', text)
if match:
    print(match.groups())  # Output: ('2023', '05', '12')
    print(match.group(2))  # Output: 05
```

Named Groups and Backreferences

Named groups use `(?P...)` syntax, and backreferences like `\1` refer to captured groups.

```python
import re

# Named group for email username
text = "Email: alice@example.com"
match = re.search(r'(?P[\w\.-]+)@[\w\.-]+\.\w+', text)
if match:
    print(match.group('username'))  # Output: alice

# Backreference to match repeated word
text = "hello hello world"
result = re.sub(r'\b(\w+)\s+\1\b', r'\1', text)
print(result)  # Output: hello world
```

Non-Capturing Groups

Non-capturing groups `(?:...)` group patterns without capturing for later use.

```python
import re

# Non-capturing group for protocol
text = "https://example.com"
match = re.search(r'(?:http|https)://(\w+)', text)
if match:
    print(match.group(1))  # Output: example
```

Advanced RegEx Techniques

Lookaheads and Lookbehinds

Lookaheads `(?=...)` and lookbehinds `(?<=...)` assert conditions without including them in the match.

```python
import re

# Positive lookahead: match digits followed by 'px'
text = "12px 34em 56px"
result = re.findall(r'\d+(?=px)', text)
print(result)  # Output: ['12', '56']

# Positive lookbehind: match word after 'Mr.'
text = "Mr. Smith and Mr. Jones"
result = re.findall(r'(?<=Mr\.)\s*(\w+)', text)
print(result)  # Output: ['Smith', 'Jones']
```

Compiling Patterns with re.compile()

Use `re.compile()` to precompile patterns for efficiency in repeated use.

```python
import re

# Compile pattern for reuse
pattern = re.compile(r'\b\d{3}\b')
texts = ["123 is valid", "12 is not", "456 works"]
for text in texts:
    if pattern.search(text):
        print(f"Found in: {text}")  # Output: Found in: 123 is valid
                                    #         Found in: 456 works
```

Handling Multiline Text and Flags

Flags like `re.MULTILINE` and `re.IGNORECASE` modify regex behavior for multiline text or case sensitivity.

```python
import re

# Match start of each line with MULTILINE
text = "start line1\nstart line2"
pattern = re.compile(r'^start', re.MULTILINE)
result = pattern.findall(text)
print(result)  # Output: ['start', 'start']

# Case-insensitive match
text = "Hello WORLD"
pattern = re.compile(r'hello', re.IGNORECASE)
result = pattern.search(text)
print(result.group())  # Output: Hello
```

Debugging and Testing RegEx Patterns

Use `re.finditer()` to inspect matches or online tools like regex101.com for testing patterns.

```python
import re

# Debug with finditer
text = "Dates: 2023-05-12, 2024-06-15"
pattern = re.compile(r'\d{4}-\d{2}-\d{2}')
for match in pattern.finditer(text):
```

```
    print(f"Match: {match.group()} at {match.start()}-{match.end()}")
# Output: Match: 2023-05-12 at 7-17
#         Match: 2024-06-15 at 19-29
```

Challenging Questions

Question 1: What will be the output of the following code, and why?

```python
import re

text = "Contact: alice@example.com, bob@domain.com"
result = re.sub(r'\b(\w+)@([\w\.]+)\b', r'\1@REDACTED', text)
print(result)
```

Show Answer

Question 2: Why does this code produce unexpected results, and how can it be fixed?

```python
import re

text = "line1\nline2\nline3"
matches = re.findall(r'^line\d$', text)
print(matches)
```

Show Answer

Question 3: What will be the output of this code, and why?

```python
import re

text = "The year is 2023, not 2024."
pattern = re.compile(r'\b(?P\d{4})\b')
match = pattern.search(text)
if match:
    print(match.group('year'))
```

Show Answer

For more information and testing RegEx patterns, visit regex101.com.

2.5 Advanced Regex

Python Regular Expressions (RegEx)

Substituting and Splitting with RegEx

Using re.sub() for Pattern Replacement

The `re.sub()` function replaces all occurrences of a pattern with a specified string, ideal for text transformations.

```Python
import re

# Replace digits with asterisks
text = "Order #12345"
result = re.sub(r'\d', '*', text)
print(result)  # Output: Order #*****
```

Using re.split() to Split Strings by Patterns

The `re.split()` function splits a string based on a regex pattern, supporting complex delimiters.

```Python
import re

# Split on commas, semicolons, or spaces
text = "cat,dog;bird fish"
result = re.split(r'[,;\s]+', text)
print(result)  # Output: ['cat', 'dog', 'bird', 'fish']
```

Practical Examples

Practical applications include masking sensitive data like emails or splitting text with multiple separators.

```Python
import re

# Mask email addresses
text = "Reach me at john.doe@example.com or jane@company.org"
result = re.sub(r'\b[\w\.-]+@[\w\.-]+\.\w+\b', '[HIDDEN]', text)
print(result)  # Output: Reach me at [HIDDEN] or [HIDDEN]

# Split on multiple delimiters
text = "item1,item2;item3 item4"
result = re.split(r'[,;\s]+', text)
print(result)  # Output: ['item1', 'item2', 'item3', 'item4']
```

Grouping and Capturing in RegEx

Creating Groups with Parentheses ()

Parentheses `()` create capture groups to isolate parts of a matched pattern for further use.

```python
import re

# Capture area code from phone number
text = "Phone: (415) 555-1234"
match = re.search(r'\((\d{3})\)', text)
if match:
    print(match.group(1))  # Output: 415
```

Accessing Captured Groups in Matches

Use `match.group(n)` to access the nth captured group or `match.groups()` for all groups.

```python
import re

# Extract date components
text = "Event on 2025-10-25"
match = re.search(r'(\d{4})-(\d{2})-(\d{2})', text)
if match:
    print(match.groups())  # Output: ('2025', '10', '25')
    print(match.group(1))  # Output: 2025
```

Named Groups and Backreferences

Named groups `(?P...)` assign names to groups, and backreferences `\n` reuse captured text.

```python
import re

# Named group for username
text = "Email: user@domain.com"
match = re.search(r'(?P[\w\.-]+)@[\w\.-]+\.\w+', text)
if match:
    print(match.group('user'))  # Output: user

# Backreference to remove repeated words
text = "the the quick fox"
result = re.sub(r'\b(\w+)\s+\1\b', r'\1', text)
print(result)  # Output: the quick fox
```

Non-Capturing Groups

Non-capturing groups `(?:...)` group patterns without storing them for capture.

```python
import re

# Non-capturing group for protocol
text = "http://website.com"
match = re.search(r'(?:http|https)://(\w+)', text)
if match:
    print(match.group(1))  # Output: website
```

Advanced RegEx Techniques

Lookaheads and Lookbehinds

Positive/negative lookaheads `(?=...)` / `(?!...)` and lookbehinds `(?<=...)` / `(? assert conditions without including them in the match.`

```python
import re

# Positive lookahead: digits before 'cm'
text = "10cm 20px 30cm"
result = re.findall(r'\d+(?=cm)', text)
print(result)  # Output: ['10', '30']

# Negative lookbehind: words not after 'no-'
text = "no-stop go run no-jump"
result = re.findall(r'(?
```

Compiling Patterns with re.compile()

The `re.compile()` function precompiles regex patterns for faster execution in repetitive tasks.

```python
import re

# Compile pattern for repeated use
pattern = re.compile(r'\b\d{4}\b')
texts = ["Year: 2023", "Code: 12", "Date: 2025"]
for text in texts:
    if pattern.search(text):
        print(f"Match in: {text}")  # Output: Match in: Year: 2023
                                    #         Match in: Date: 2025
```

Handling Multiline Text and Flags

Flags like `re.MULTILINE` and `re.IGNORECASE` enable line-specific matching and case-insensitive searches.

```python
import re

# Match line starts with MULTILINE
text = "begin line1\nbegin line2"
pattern = re.compile(r'^begin', re.MULTILINE)
result = pattern.findall(text)
print(result)  # Output: ['begin', 'begin']

# Case-insensitive search
text = "Python PYTHON python"
pattern = re.compile(r'python', re.IGNORECASE)
result = pattern.findall(text)
print(result)  # Output: ['Python', 'PYTHON', 'python']
```

Debugging and Testing RegEx Patterns

Use `re.finditer()` to inspect match details or tools like regex101.com to test and refine patterns.

```python
import re

# Inspect matches with finditer
text = "Events: 2023-01-01, 2024-02-02"
pattern = re.compile(r'\d{4}-\d{2}-\d{2}')
for match in pattern.finditer(text):
    print(f"Found {match.group()} at {match.start()}-{match.end()}")
# Output: Found 2023-01-01 at 8-18
#         Found 2024-02-02 at 20-30
```

Challenging Questions

Question 1: What will be the output of this code, and why?

```python
import re

text = "Emails: user1@site.com, user2@site.com"
result = re.sub(r'\b(\w+)@[\w\.-]+\.\w+\b', r'\1@hidden', text)
print(result)
```

Show Answer

Question 2: Why does this code fail to match, and how can it be fixed?

```python
import re

text = "data1\ndata2\ndata3"
matches = re.findall(r'^data\d$', text)
print(matches)
```

Show Answer

Question 3: What will be the output of this code, and why?

```python
import re

text = "Price: $100 and $200"
pattern = re.compile(r'(?<=Price: \$)\d+')
matches = pattern.findall(text)
print(matches)
```

Show Answer

PythonInstallation

Python Installation & Setup

Step 1: Download Python

Go to the official Python website: python.org/downloads

Choose the version suitable for your operating system (Windows, macOS, Linux).

Step 2: Install Python

Windows: Run the installer and check `Add Python to PATH` before clicking install.

macOS/Linux: Use the installer or package manager like `brew install python` or `sudo apt install python3`.

Step 3: Verify Installation

Open a terminal or command prompt and run:

```
 python --version
python3 --version
```

Step 4: Install Code Editor

Install Visual Studio Code or any editor you like.

Step 5: Install pip and virtualenv (Optional)

```
pip install virtualenv
```

This helps you manage Python environments for different projects.

PythonInstallation

Python Installation & Setup

Step 1: Download Python

Go to the official Python website: python.org/downloads

Choose the version suitable for your operating system (Windows, macOS, Linux).

Step 2: Install Python

Windows: Run the installer and check `Add Python to PATH` before clicking install.

macOS/Linux: Use the installer or package manager like `brew install python` or `sudo apt install python3`.

Step 3: Verify Installation

Open a terminal or command prompt and run:

```
 python --version
python3 --version
```

Step 4: Install Code Editor

Install Visual Studio Code or any editor you like.

Step 5: Install pip and virtualenv (Optional)

```
pip install virtualenv
```

This helps you manage Python environments for different projects.

PythonInstallation

Python Installation & Setup

Step 1: Download Python

Go to the official Python website: python.org/downloads

Choose the version suitable for your operating system (Windows, macOS, Linux).

Step 2: Install Python

Windows: Run the installer and check `Add Python to PATH` before clicking install.

macOS/Linux: Use the installer or package manager like `brew install python` or `sudo apt install python3`.

Step 3: Verify Installation

Open a terminal or command prompt and run:

```
 python --version
python3 --version
```

Step 4: Install Code Editor

Install Visual Studio Code or any editor you like.

Step 5: Install pip and virtualenv (Optional)

```
pip install virtualenv
```

This helps you manage Python environments for different projects.

5.4 Introduction to Pandas

Introduction to Pandas

What is Pandas and Why Use It

Overview of Pandas as a Data Manipulation Library

Pandas is a powerful Python library for data manipulation and analysis, built on top of NumPy. It provides flexible data structures and tools for working with structured data.

```python
import pandas as pd

# Example: Create a simple DataFrame
data = {'Name': ['Alice', 'Bob'], 'Age': [25, 30]}
df = pd.DataFrame(data)
print(df)
# Output:
#     Name  Age
# 0  Alice   25
# 1    Bob   30
```

Key Features (Series, DataFrames, Data Analysis Tools)

Pandas offers `Series` for 1D data, `DataFrames` for 2D tabular data, and tools for filtering, grouping, and merging datasets.

```python
import pandas as pd

# Series example
series = pd.Series([10, 20, 30], index=['a', 'b', 'c'])
print(series)
# Output:
# a    10
# b    20
# c    30
# dtype: int64
```

Comparison with Other Tools (e.g., NumPy, Excel)

Unlike NumPy, Pandas handles heterogeneous data and labeled axes. Compared to Excel, Pandas is programmatic, scalable, and better for large datasets.

```python
import pandas as pd
import numpy as np

# Pandas vs NumPy
numpy_array = np.array([1, 2, 3])
pandas_series = pd.Series([1, 2, 3], index=['x', 'y', 'z'])
print(numpy_array)  # Output: [1 2 3]
print(pandas_series)
# Output:
# x    1
# y    2
```

```
# z    3
# dtype: int64
```

Installing and Setting Up Pandas

Installing Pandas Using pip or conda

Install Pandas via `pip install pandas` or `conda install pandas` in your Python environment.

```Python
 # Command line examples (not Python code)
# pip install pandas
# conda install pandas
```

Importing Pandas in Python

Import Pandas using the conventional alias `pd` for concise code.

```Python
 import pandas as pd

# Basic usage
df = pd.DataFrame({'A': [1, 2], 'B': [3, 4]})
print(df)
# Output:
#    A  B
# 0  1  3
# 1  2  4
```

Verifying Installation and Version

Check the installed Pandas version to ensure compatibility.

```Python
 import pandas as pd

# Check version
print(pd.__version__)  # Output: e.g., 2.2.2
```

Pandas Data Structures Overview

Introduction to Series and DataFrames

A `Series` is a one-dimensional labeled array, while a `DataFrame` is a two-dimensional table with rows and columns.

```Python
 import pandas as pd

# Series
s = pd.Series([100, 200, 300], index=['Jan', 'Feb', 'Mar'])
print(s)
# Output:
# Jan    100
```

```
# Feb    200
# Mar    300
# dtype: int64

# DataFrame
df = pd.DataFrame({'Sales': s})
print(df)
# Output:
#      Sales
# Jan    100
# Feb    200
# Mar    300
```

Differences Between Series and DataFrames

A `Series` is like a single column with an index, while a `DataFrame` is a collection of Series with shared indices.

Python

```
import pandas as pd

# Series vs DataFrame
series = pd.Series([1, 2, 3])
df = pd.DataFrame({'col1': [1, 2, 3], 'col2': [4, 5, 6]})
print(series)
# Output:
# 0    1
# 1    2
# 2    3
# dtype: int64
print(df)
# Output:
#    col1  col2
# 0     1     4
# 1     2     5
# 2     3     6
```

Basic Attributes (shape, dtypes, index)

Key attributes include `shape` (dimensions), `dtypes` (column data types), and `index` (row labels).

<

Python

```
import pandas as pd

# DataFrame attributes
df = pd.DataFrame({'A': [1, 2, 3], 'B': ['x', 'y', 'z']})
print(df.shape)   # Output: (3, 2)
print(df.dtypes)  # Output: A     int64
                  #         B    object
                  #         dtype: object
print(df.index)   # Output: RangeIndex(start=0, stop=3, step=1)
```

Challenging Questions

Question 1: What will be the output of this code, and why?

```python
import pandas as pd

data = {'A': [1, 2, None], 'B': [4, 5, 6]}
df = pd.DataFrame(data)
print(df.dtypes)
```

Show Answer

Question 2: Why does this code raise an error, and how can it be fixed?

```python
import pandas as pd

series = pd.Series([1, 2, 3], index=[0, 1, 2])
print(series[3])
```

Show Answer

Question 3: What will be the output of this code, and why?

```python
import pandas as pd

df = pd.DataFrame({'X': [10, 20, 30], 'Y': ['a', 'b', 'c']}, index=['p', 'q', 'r'])
print(df.index)
```

Show Answer

5.5 Pandas Series and Data Frames

Pandas: Creating and Modifying Series & DataFrames

Creating Series

From Lists, Arrays, or Dictionaries

A `Series` is a one-dimensional labeled array that can be created from lists, NumPy arrays, or dictionaries, offering flexibility for data representation.

```python
import pandas as pd
import numpy as np

# From list
s1 = pd.Series([10, 20, 30])
print(s1)
# Output:
# 0    10
# 1    20
# 2    30
# dtype: int64

# From NumPy array
s2 = pd.Series(np.array([1.5, 2.5, 3.5]))
print(s2)
# Output:
# 0    1.5
# 1    2.5
# 2    3.5
# dtype: float64

# From dictionary
s3 = pd.Series({'a': 100, 'b': 200, 'c': 300})
print(s3)
# Output:
# a    100
# b    200
# c    300
# dtype: int64
```

Specifying Custom Indices

Custom indices can be set during Series creation to provide meaningful labels, enhancing data interpretability.

```python
import pandas as pd

# Series with custom index
s = pd.Series([5, 10, 15], index=['Jan', 'Feb', 'Mar'])
print(s)
# Output:
# Jan     5
```

```
# Feb    10
# Mar    15
# dtype: int64
```

Series Attributes (values, index, name)

Attributes like `values` (data as a NumPy array), `index` (labels), and `name` (Series identifier) provide metadata access.

Python
```
 import pandas as pd

s = pd.Series([1, 2, 3], index=['x', 'y', 'z'], name='Data')
print(s.values)  # Output: [1 2 3]
print(s.index)   # Output: Index(['x', 'y', 'z'], dtype='object')
print(s.name)    # Output: Data
```

Creating DataFrames

From Dictionaries, Lists, or NumPy Arrays

`DataFrames` are two-dimensional tabular structures created from dictionaries (columns), lists (rows), or NumPy arrays for structured data.

Python
```
 import pandas as pd
import numpy as np

# From dictionary
df1 = pd.DataFrame({'Name': ['Alice', 'Bob'], 'Age': [25, 30]})
print(df1)
# Output:
#     Name  Age
# 0  Alice   25
# 1    Bob   30

# From list of lists
df2 = pd.DataFrame([[1, 'A'], [2, 'B']], columns=['ID', 'Grade'])
print(df2)
# Output:
#    ID Grade
# 0   1     A
# 1   2     B

# From NumPy array
df3 = pd.DataFrame(np.array([[10, 20], [30, 40]]), columns=['X', 'Y'])
print(df3)
# Output:
#     X   Y
# 0  10  20
# 1  30  40
```

Setting Column Names and Indices

Column names and indices can be specified during creation or modified later to align with the dataset's context.

```python
import pandas as pd

# DataFrame with custom columns and index
df = pd.DataFrame({'Score': [85, 90], 'Status': ['Pass', 'Pass']}, index=['Test1', 'Test2'])
print(df)
# Output:
#        Score Status
# Test1     85   Pass
# Test2     90   Pass
```

DataFrame Attributes (columns, index, shape)

Attributes like `columns` (column labels), `index` (row labels), and `shape` (dimensions) describe the DataFrame's structure.

```python
import pandas as pd

df = pd.DataFrame({'A': [1, 2, 3], 'B': ['x', 'y', 'z']}, index=['p', 'q', 'r'])
print(df.columns)  # Output: Index(['A', 'B'], dtype='object')
print(df.index)    # Output: Index(['p', 'q', 'r'], dtype='object')
print(df.shape)    # Output: (3, 2)
```

Modifying Series and DataFrames

Adding or Renaming Columns/Indices

New columns can be added via assignment, and `rename()` updates column names or indices without altering data.

```python
import pandas as pd

# Add new column
df = pd.DataFrame({'A': [10, 20, 30]})
df['B'] = df['A'] * 2
print(df)
# Output:
#     A   B
# 0  10  20
# 1  20  40
# 2  30  60

# Rename columns and indices
df = df.rename(columns={'A': 'X', 'B': 'Y'}, index={0: 'a', 1: 'b', 2: 'c'})
print(df)
# Output:
#     X   Y
# a  10  20
# b  20  40
# c  30  60
```

Converting Data Types

Use `astype()` or specific converters to change the data types of Series or DataFrame columns for compatibility

or analysis.

```python
import pandas as pd

# Convert data types
df = pd.DataFrame({'A': ['1', '2', '3'], 'B': [4.5, 5.5, 6.5]})
df['A'] = df['A'].astype(int)
df['B'] = df['B'].astype(int)
print(df)
# Output:
#    A  B
# 0  1  4
# 1  2  5
# 2  3  6
```

Copying vs. Modifying in Place

Modifications can be applied in place or on a copy using `copy()` to preserve the original data.

```python
import pandas as pd

# Modify in place
df = pd.DataFrame({'A': [1, 2, 3]})
df['A'] = df['A'] + 10
print(df)
# Output:
#     A
# 0  11
# 1  12
# 2  13

# Create a copy
df_copy = df.copy()
df_copy['A'] = df_copy['A'] * 2
print(df_copy)
# Output:
#     A
# 0  22
# 1  24
# 2  26
print(df)  # Original unchanged
# Output:
#     A
# 0  11
# 1  12
# 2  13
```

Challenging Questions

> **Question 1:** What will be the output of this code, and why?

```python
import pandas as pd

s = pd.Series([10, 20, 30], index=['x', 'y', 'z'], name='Values')
s.index = ['a', 'b', 'c']
print(s.name)
```

Show Answer

Question 2: Why does this code produce unexpected results, and how can it be fixed?

```python
import pandas as pd

df = pd.DataFrame({'A': [1, 2, 3]})
df['B'] = df['A']
df['B'][0] = 10
print(df)
```

Show Answer

Question 3: What will be the output of this code, and why?

```python
import pandas as pd

df = pd.DataFrame({'A': ['1', '2', '3']}, index=['x', 'y', 'z'])
df['A'] = df['A'].astype(float)
print(df.dtypes)
```

Show Answer

PythonInstallation

Python Installation & Setup

Step 1: Download Python

Go to the official Python website: python.org/downloads

Choose the version suitable for your operating system (Windows, macOS, Linux).

Step 2: Install Python

Windows: Run the installer and check `Add Python to PATH` before clicking install.

macOS/Linux: Use the installer or package manager like `brew install python` or `sudo apt install python3`.

Step 3: Verify Installation

Open a terminal or command prompt and run:

```
 python --version
python3 --version
```

Step 4: Install Code Editor

Install Visual Studio Code or any editor you like.

Step 5: Install pip and virtualenv (Optional)

```
pip install virtualenv
```

This helps you manage Python environments for different projects.

PythonInstallation

Python Installation & Setup

Step 1: Download Python

Go to the official Python website: python.org/downloads

Choose the version suitable for your operating system (Windows, macOS, Linux).

Step 2: Install Python

Windows: Run the installer and check `Add Python to PATH` before clicking install.

macOS/Linux: Use the installer or package manager like `brew install python` or `sudo apt install python3`.

Step 3: Verify Installation

Open a terminal or command prompt and run:

```
 python --version
python3 --version
```

Step 4: Install Code Editor

Install Visual Studio Code or any editor you like.

Step 5: Install pip and virtualenv (Optional)

```
pip install virtualenv
```

This helps you manage Python environments for different projects.

PythonInstallation

Python Installation & Setup

Step 1: Download Python

Go to the official Python website: python.org/downloads

Choose the version suitable for your operating system (Windows, macOS, Linux).

Step 2: Install Python

Windows: Run the installer and check `Add Python to PATH` before clicking install.

macOS/Linux: Use the installer or package manager like `brew install python` or `sudo apt install python3`.

Step 3: Verify Installation

Open a terminal or command prompt and run:

```
 python --version
python3 --version
```

Step 4: Install Code Editor

Install Visual Studio Code or any editor you like.

Step 5: Install pip and virtualenv (Optional)

```
pip install virtualenv
```

This helps you manage Python environments for different projects.

PythonInstallation

Python Installation & Setup

Step 1: Download Python

Go to the official Python website: python.org/downloads

Choose the version suitable for your operating system (Windows, macOS, Linux).

Step 2: Install Python

Windows: Run the installer and check `Add Python to PATH` before clicking install.

macOS/Linux: Use the installer or package manager like `brew install python` or `sudo apt install python3`.

Step 3: Verify Installation

Open a terminal or command prompt and run:

```
 python --version
python3 --version
```

Step 4: Install Code Editor

Install Visual Studio Code or any editor you like.

Step 5: Install pip and virtualenv (Optional)

```
pip install virtualenv
```

This helps you manage Python environments for different projects.