Mastering Python Through Errors A Practical Guide

Kai Guo

Table of contents

Chapter 1

Python Error Guide - Complete File Index

1.1 Package Contents

This directory contains everything you need to publish your 20-chapter Python Error Guide.

1.2 Files in This Package

1.2.1 1. Core Documentation

1.2.1.1 README.md

- **Purpose**: Main guide overview and table of contents
- **Use for**: Landing page, GitHub README
- **Size**: 5.0 KB
- **Contains**:
  - Complete guide introduction
  - Full 20-chapter table of contents
  - Learning paths
  - Error index
  - Quick start guide

View README.md

1.2.1.2 PACKAGE_SUMMARY.md

- **Purpose**: Overview of this publishing package
- **Use for**: Understanding what you have and next steps
- **Size**: ~10 KB
- **Contains**:
  - What's included
  - Step-by-step next steps
  - Publishing platform options
  - Launch plan
  - Success tips

View PACKAGE_SUMMARY.md

1.2.2 2. Guide Content

1.2.2.1 chapter-01-variables-data-types.md

- **Status**: COMPLETE
- **Purpose**: Full chapter 1 - example/template
- **Size**: 16 KB
- **Contains**:
  - Variables and data types
  - Common errors: NameError, TypeError, ValueError
  - Code examples (wrong and correct)
  - Practice problems with solutions
  - Key takeaways

View Chapter 1

1.2.2.2 Chapters 2-20

- **Status**: IN CONVERSATION ABOVE
- **Purpose**: Remaining 19 chapters
- **Location**: Scroll up in the conversation
- **How to extract**: See EXTRACTION_GUIDE.md

Chapter List: - Chapter 2: Operators and Expressions - Chapter 3: Strings and String Methods - Chapter 4: Lists and List Methods - Chapter 5: Dictionaries and Sets - Chapter 6: Tuples and Immutability - Chapter 7: Conditional Statements - Chapter 8: Loops - Chapter 9: Functions - Chapter 10: File I/O - Chapter 11: Regular Expressions - Chapter 12: Pandas Basics - Chapter 13: Pandas Advanced - Chapter 14: NumPy - Chapter 15: Matplotlib - Chapter 16: Object-Oriented Programming - Chapter 17: Modules and Imports - Chapter 18:

Exception Handling - Chapter 19: Debugging Techniques - Chapter 20: Testing and Code Quality

---

Happy Publishing!

Last Updated: October 26, 2025 Package Version: 1.0

Chapter 2

Chapter 1: Variables and Data Types - The Foundation of Python Errors

2.1 Introduction

Welcome to your journey of mastering Python errors! Before we can understand errors, we need to understand the basics: **variables** and **data types**. Most Python errors stem from misunderstanding how Python handles data.

Think of variables as labeled boxes that store information. The **type** of information (number, text, list, etc.) determines what operations you can perform. Using the wrong type leads to errors - and that's what we'll learn to avoid!

2.2 1.1 Understanding Variables

2.2.1 What is a Variable?

```python
# A variable is a name that refers to a value
name = "Alice"
age = 25
is_student = True
```

```
# Variables can change (they're "variable")
age = 26  # Changed the value
```

Key Concepts: - Variables are created when you first assign a value - Python is **dynamically typed** - you don't declare types - Variable names are case-sensitive (`age` `Age`) - Use descriptive names (`user_age` not `x`)

2.2.2 Error Type 1: `NameError: name 'X' is not defined`

Error Message:

```
>>> print(age)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'age' is not defined
```

What Happened: You tried to use a variable that doesn't exist yet.

Why It Happens: - Variable was never created - Typo in variable name - Variable used before assignment - Wrong scope (variable defined in function, used outside)

Code Example - WRONG:

```
# Using before defining
print(username)  # ERROR! username doesn't exist yet
username = "Alice"

# Typo in variable name
user_name = "Bob"
print(user_nane)  # ERROR! Typo: 'nane' instead of 'name'

# Case sensitivity
Age = 30
print(age)  # ERROR! Python sees 'age' and 'Age' as different

# Forgetting to assign
result  # ERROR! Just naming isn't enough
result = 10  # Need to assign a value
```

Code Example - CORRECT:

```
# Define before using
username = "Alice"
print(username)  #   Works

# Check spelling carefully
```

```
user_name = "Bob"
print(user_name)  #   Correct spelling

# Match case exactly
age = 30
print(age)  #   Correct case

# Always assign a value
result = 10
print(result)  #   Works
```

Prevention Pattern:

```
# Check if variable exists before using
try:
    print(username)
except NameError:
    username = "Default User"
    print(username)

# Or use getattr with default for attributes
# (We'll cover this later)
```

2.3 1.2 Basic Data Types

2.3.1 Numbers: int and float

```
# Integers (whole numbers)
age = 25
year = 2025
temperature = -10

# Floats (decimal numbers)
price = 19.99
pi = 3.14159
temperature = -10.5

# Python automatically chooses the right type
x = 5      # int
y = 5.0    # float
z = 5.     # float (same as 5.0)
```

2.3.2 Error Type 2: `TypeError: unsupported operand type(s)`

Error Message:

```
>>> "5" + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

What Happened: You tried to combine incompatible types (string + number).

Why It Happens: - Mixing strings and numbers in operations - Wrong type from user input (always strings) - Forgetting to convert types

Code Example - WRONG:

```
# Mixing types
age = "25"
next_year = age + 1  # ERROR! Can't add string + int

# User input is always string
user_input = input("Enter a number: ")  # Returns string
result = user_input + 10  # ERROR! String + int

# Forgetting conversion
price = "19.99"
tax = price * 0.1  # ERROR! Can't multiply string by float
```

Code Example - CORRECT:

```
# Convert string to int
age = "25"
next_year = int(age) + 1  #   Works: 26

# Convert user input
user_input = input("Enter a number: ")
number = int(user_input)  # Convert to int first
result = number + 10  #   Works

# Convert string to float
price = "19.99"
price_float = float(price)
tax = price_float * 0.1  #   Works

# Convert number to string (for concatenation)
age = 25
message = "Age: " + str(age)  #   Works
# Or use f-strings (better):
```

```
message = f"Age: {age}"  #  Automatic conversion
```

Type Conversion Functions:

```
# String to number
int("123")       # 123 (integer)
float("3.14")    # 3.14 (float)
int("3.14")      # ERROR! Can't convert float string directly to int
int(float("3.14"))  # 3 (convert to float first, then int)

# Number to string
str(123)         # "123"
str(3.14)        # "3.14"

# Check type
type(5)          # <class 'int'>
type(5.0)        # <class 'float'>
type("5")        # <class 'str'>

isinstance(5, int)      # True
isinstance(5.0, int)    # False
isinstance(5.0, float) # True
```

2.3.3 Error Type 3: `ValueError: invalid literal for int()`

Error Message:

```
>>> int("hello")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hello'
```

What Happened: You tried to convert a string to a number, but the string doesn't contain a valid number.

Why It Happens: - String contains non-numeric characters - Empty string - String with spaces (leading/trailing) - User input validation issues

Code Example - WRONG:

```
# Non-numeric string
age = int("twenty-five")  # ERROR! Not a number

# Empty string
value = int("")  # ERROR! Empty
```

```
# String with spaces
number = int("  42  ")  # ERROR! (Actually this works, but not always safe)

# Decimal string to int
value = int("3.14")  # ERROR! Use float first
```

Code Example - CORRECT:

```
# Validate before converting
text = "hello"
if text.isdigit():
    number = int(text)
else:
    print("Not a valid number")  #   Handles error

# Handle conversion errors with try/except
user_input = "invalid"
try:
    number = int(user_input)
except ValueError:
    print("Please enter a valid number")
    number = 0  # Default value

# Strip whitespace before converting
text = "  42  "
number = int(text.strip())  #   Works

# Convert float string to int (two steps)
text = "3.14"
number = int(float(text))  #   Works: 3

# Validate with helper function
def safe_int(text, default=0):
    """Safely convert to int with default"""
    try:
        return int(text)
    except ValueError:
        return default

value = safe_int("invalid", default=0)  #   Returns 0
value = safe_int("42")  #   Returns 42
```

2.4 1.3 Strings

2.4.1 String Basics

```
# Creating strings
name = "Alice"
message = 'Hello, World!'
multiline = """This is
a multiline
string"""

# String operations
greeting = "Hello" + " " + "World"  # Concatenation
repeated = "Ha" * 3  # "HaHaHa"
length = len("Python")  # 6
```

2.4.2 Error Type 4: String Index Errors

Error Message:

```
>>> text = "Hello"
>>> print(text[10])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

What Happened: You tried to access a character position that doesn't exist.

Why It Happens: - Index too large (beyond string length) - Forgetting Python uses 0-based indexing - Empty string

Code Example - WRONG:

```
text = "Hello"  # Length is 5, indices are 0-4

# Index too large
char = text[5]  # ERROR! Valid indices: 0-4

# Wrong assumption about length
first = text[1]  # This is 'e', not 'H' (0-based!)

# Empty string
empty = ""
char = empty[0]  # ERROR! No characters
```

Code Example - CORRECT:

```
text = "Hello"

# Use valid indices (0 to len-1)
first = text[0]   #   'H' (first character)
last = text[4]    #   'o' (last character)
last = text[-1]   #   'o' (negative index from end)

# Check length before accessing
if len(text) > 5:
    char = text[5]
else:
    print("Index out of range")

# Safe access with try/except
try:
    char = text[10]
except IndexError:
    char = None  # Or default value

# Use slicing (doesn't raise error)
substring = text[10:15]  #   Returns empty string, no error
```

String Indexing:

```
text = "Python"
#       012345  (positive indices)
#      -654321  (negative indices)

text[0]    # 'P'
text[5]    # 'n'
text[-1]   # 'n' (last character)
text[-6]   # 'P' (first character)

# Slicing [start:stop:step]
text[0:3]   # 'Pyt' (indices 0, 1, 2)
text[:3]    # 'Pyt' (from start)
text[3:]    # 'hon' (to end)
text[::2]   # 'Pto' (every 2nd character)
text[::-1]  # 'nohtyP' (reversed)
```

2.5 1.4 Booleans

2.5.1 Boolean Basics

```
# Boolean values
is_valid = True
is_empty = False

# Comparison operations create booleans
age = 25
is_adult = age >= 18  # True
is_teen = 13 <= age < 20  # True

# Logical operations
x = True
y = False
result = x and y  # False
result = x or y   # True
result = not x    # False
```

2.5.2 Error Type 5: Type Confusion with Booleans

What Happened: Treating non-boolean values as if they were boolean, or vice versa.

Code Example - WRONG:

```
# Confusing truthy/falsy with boolean
value = "False"  # This is a string!
if value:
    print("This runs!")  # String "False" is truthy!

# Comparing boolean wrong way
is_valid = True
if is_valid == "True":  # Comparing bool to string
    print("Won't work")  # Never executes

# Using assignment in condition
x = 5
if x = 10:  # ERROR! Assignment, not comparison
    print("Won't work")
```

Code Example - CORRECT:

```
# Use actual boolean values
value = False  #  Boolean, not string
```

```python
# Compare correctly
is_valid = True
if is_valid:  #   Direct boolean check
    print("Valid!")

# Use == for comparison
x = 5
if x == 10:  #   Comparison operator
    print("Equal to 10")

# Understand truthy/falsy
# Falsy: False, None, 0, "", [], {}, ()
# Truthy: Everything else

if "":  # Empty string is falsy
    print("Won't print")

if "text":  # Non-empty string is truthy
    print("Will print")  #

# Explicit boolean conversion
text = "hello"
bool(text)  # True (non-empty string)
bool("")    # False (empty string)
bool(0)     # False
bool(42)    # True
```

2.6 1.5 None Type

2.6.1 Understanding None

```python
# None represents absence of value
result = None
name = None

# Checking for None
if result is None:
    print("No result yet")

# Don't use == for None
if result == None:  # Works but not recommended
    print("Better use 'is'")
```

```python
if result is None:  #  Preferred way
    print("No result")
```

2.6.2 Error Type 6: `TypeError: 'NoneType' object is not...`

Error Message:

```python
>>> result = None
>>> result + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

What Happened: You tried to use None as if it were another type (number, string, etc.).

Why It Happens: - Function returns None (implicit or explicit) - Variable not initialized properly - Forgetting to return value from function

Code Example - WRONG:

```python
# Function returns None implicitly
def calculate():
    result = 5 + 3
    # Oops, forgot to return!

answer = calculate()  # Gets None
total = answer + 10  # ERROR! None + 10

# Using None in operations
value = None
doubled = value * 2  # ERROR! Can't multiply None

# Method that returns None
numbers = [3, 1, 2]
result = numbers.sort()  # sort() returns None!
print(result[0])  # ERROR! Can't index None
```

Code Example - CORRECT:

```python
# Always return a value from functions
def calculate():
    result = 5 + 3
    return result  #  Explicit return
```

```
answer = calculate()
total = answer + 10  #   Works

# Check for None before using
value = None
if value is not None:
    doubled = value * 2
else:
    doubled = 0  # Default value

# Some methods modify in-place and return None
numbers = [3, 1, 2]
numbers.sort()  # Modifies list, returns None
print(numbers[0])  #   Use the modified list: 1

# Or use function that returns new value
numbers = [3, 1, 2]
sorted_numbers = sorted(numbers)  # Returns new sorted list
print(sorted_numbers[0])  #   Works: 1

# Provide default value
def get_value():
    return None

result = get_value() or 0  # Use 0 if None
result = get_value() if get_value() is not None else 0  # More explicit
```

2.7 1.6 Practice Problems - Fix These Errors!

2.7.1 Problem 1: NameError

```
print(user_name)
user_name = "Alice"
```

What's wrong?

Click for Answer

Error: `NameError: name 'user_name' is not defined`

Why: Variable is used before it's defined.

Fix:

```
user_name = "Alice"  # Define first
print(user_name)     # Then use
```

2.7.2 Problem 2: TypeError

```
age = "25"
next_year = age + 1
print(next_year)
```

What's wrong?

Click for Answer

Error: `TypeError: can only concatenate str (not "int") to str`

Why: Can't add string and integer.

Fix:

```
age = "25"
next_year = int(age) + 1  # Convert to int first
print(next_year)  # 26
```

2.7.3 Problem 3: ValueError

```
user_input = "twenty-five"
age = int(user_input)
```

What's wrong?

Click for Answer

Error: `ValueError: invalid literal for int() with base 10: 'twenty-five'`

Why: String doesn't contain a valid number.

Fix:

```
user_input = "twenty-five"
try:
    age = int(user_input)
except ValueError:
    print("Please enter a numeric value")
    age = 0  # Default value
```

2.7.4 Problem 4: IndexError

```
text = "Hi"
print(text[5])
```

What's wrong?

Click for Answer

Error: `IndexError: string index out of range`

Why: Index 5 doesn't exist (string has indices 0 and 1 only).

Fix:

```
text = "Hi"
if len(text) > 5:
    print(text[5])
else:
    print("Index out of range")  #

# Or use negative indexing
print(text[-1])  # Last character: 'i'
```

2.7.5 Problem 5: NoneType Error

```
def get_discount():
    discount = 0.1
    # Missing return statement

price = 100
final_price = price - get_discount()
```

What's wrong?

Click for Answer

Error: `TypeError: unsupported operand type(s) for -: 'int' and 'NoneType'`

Why: Function doesn't return a value (returns None implicitly).

Fix:

```
def get_discount():
    discount = 0.1
    return discount  #  Add return statement
```

```
price = 100
final_price = price - get_discount()  #   Works now
```

2.8 1.7 Key Takeaways

2.8.1 What You Learned

1. **Define variables before using them** - Avoid `NameError`
2. **Match types in operations** - Convert when needed
3. **Validate before converting** - Use try/except for conversions
4. **Check string length before indexing** - Avoid `IndexError`
5. **Always return values from functions** - Avoid `NoneType` errors
6. **Use appropriate comparison** - `is` for None, `==` for values

2.8.2 Common Patterns

```
# Pattern 1: Safe type conversion
try:
    number = int(user_input)
except ValueError:
    number = 0  # Default value

# Pattern 2: Safe string indexing
if len(text) > index:
    char = text[index]

# Pattern 3: Check for None
if value is not None:
    # Use value

# Pattern 4: Provide defaults
result = function() or default_value
```

2.8.3 Error Summary Table

Error Type	Common Cause	Prevention
`NameError`	Using undefined variable	Define before use
`TypeError`	Mixing incompatible types	Convert types explicitly
`ValueError`	Invalid conversion	Validate before converting

Error Type	Common Cause	Prevention
`IndexError`	Invalid string index	Check length first
`NoneType` errors	Using None in operations	Check for None, return values

---

2.9 1.8 Moving Forward

You now understand the basics of variables and data types, and how to avoid common errors. In **Chapter 2**, we'll explore **Operators and Expressions** and learn about more error types!

Chapter 3

Chapter 2: Operators and Expressions - Mathematical and Logical Errors

3.1 Introduction

You've mastered variables and types. Now let's explore **operators** - the symbols that let you perform operations on data (+, -, *, /, ==, etc.). Understanding operators prevents calculation errors and logic bugs.

Operators in Python: - **Arithmetic operators**: +, -, *, /, //, %, **- Comparison operators: ==, !=, <, >, <=, >= - Logical operators: and, or, not -** Assignment operators**: =, +=, -=, *=, etc.

Let's master operators by understanding their errors!

3.2 2.1 Arithmetic Operators

3.2.1 Basic Math Operations

```
# Addition
result = 5 + 3  # 8

# Subtraction
result = 10 - 4  # 6

# Multiplication
```

```
result = 3 * 4  # 12

# Division
result = 10 / 3  # 3.333... (always returns float)

# Floor division (rounds down)
result = 10 // 3  # 3 (integer division)

# Modulo (remainder)
result = 10 % 3  # 1

# Exponentiation
result = 2 ** 3  # 8 (2 to the power of 3)
```

3.2.2 Error Type 1: `ZeroDivisionError: division by zero`

Error Message:

```
>>> result = 10 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

What Happened: You tried to divide by zero, which is mathematically undefined.

Why It Happens: - Literal zero in denominator - Variable that becomes zero - User input that's zero - Calculation result that's zero

Code Example - WRONG:

```
# Direct division by zero
result = 10 / 0  # ERROR!

# Variable is zero
denominator = 0
result = 100 / denominator  # ERROR!

# User input
number = int(input("Enter divisor: "))  # User enters 0
result = 50 / number  # ERROR!

# Calculation results in zero
x = 5
y = 5
result = 10 / (x - y)  # ERROR! (5 - 5 = 0)
```

```python
# Modulo by zero
remainder = 10 % 0  # ERROR! Also causes ZeroDivisionError
```

Code Example - CORRECT:

```python
# Check before dividing
denominator = 0
if denominator != 0:
    result = 100 / denominator
else:
    result = 0  # Or handle appropriately
    print("Cannot divide by zero")

# Using try/except
try:
    result = 10 / denominator
except ZeroDivisionError:
    print("Error: Division by zero")
    result = None

# Function with validation
def safe_divide(numerator, denominator):
    """Safely divide two numbers"""
    if denominator == 0:
        return None  # Or raise custom error
    return numerator / denominator

result = safe_divide(10, 0)  # Returns None
result = safe_divide(10, 2)  # Returns 5.0

# Using a default value
def divide_with_default(a, b, default=0):
    """Divide with default value if b is zero"""
    try:
        return a / b
    except ZeroDivisionError:
        return default

result = divide_with_default(10, 0, default=0)  # 0
result = divide_with_default(10, 2)  # 5.0

# For modulo
def safe_modulo(a, b):
    """Safe modulo operation"""
    if b == 0:
```

```
        return None
    return a % b
```

Prevention Pattern:

```
# Always validate divisor
def calculate_average(total, count):
    if count == 0:
        return 0  # Or appropriate default
    return total / count

# Check in mathematical expressions
x = 10
y = 5
denominator = (x - y)
if denominator != 0:
    result = 100 / denominator
```

3.2.3 Error Type 2: `TypeError: unsupported operand type(s) for X`

Error Message:

```
>>> result = "5" + 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'str' and 'int'
```

What Happened: You tried to perform an arithmetic operation on incompatible types.

Why It Happens: - Mixing strings and numbers - Using operators on wrong types - Forgetting type conversions

Code Example - WRONG:

```
# String + number
result = "5" + 5  # ERROR!

# String * string
result = "hello" * "world"  # ERROR!

# List + number
result = [1, 2, 3] + 5  # ERROR!

# Division with strings
result = "10" / "2"  # ERROR!
```

```
# Subtraction with strings
result = "10" - 5  # ERROR!
```

Code Example - CORRECT:

```
# Convert string to number
result = int("5") + 5  #   10
result = float("5.5") + 5  #   10.5

# Or convert number to string
result = "5" + str(5)  #   "55"

# String repetition (works!)
result = "hello" * 3  #   "hellohellohello"

# List repetition (works!)
result = [1, 2] * 3  #   [1, 2, 1, 2, 1, 2]

# List concatenation
result = [1, 2, 3] + [4, 5]  #   [1, 2, 3, 4, 5]

# Proper string operations
result = int("10") / int("2")  #   5.0

# Type checking before operation
value = "5"
if isinstance(value, str):
    value = int(value)
result = value + 5  #   10
```

Understanding Operator Compatibility:

```
# What works with each operator:

# + (Addition/Concatenation)
5 + 3  #   Numbers
"a" + "b"  #   Strings
[1] + [2]  #   Lists
# "5" + 5  #   String + Number

# * (Multiplication/Repetition)
5 * 3  #   Numbers
"a" * 3  #   String repetition
[1] * 3  #   List repetition
# "a" * "b"  #   String * String
```

```
# - (Subtraction)
5 - 3  #   Numbers only
# "5" - "3"  #   Not for strings

# / (Division)
10 / 2  #   Numbers only
# "10" / "2"  #   Not for strings
```

3.3 2.2 Comparison Operators

3.3.1 Understanding Comparisons

```
# Equality
5 == 5  # True
"hello" == "hello"  # True

# Inequality
5 != 3  # True

# Greater than / Less than
10 > 5  # True
3 < 7  # True

# Greater than or equal / Less than or equal
5 >= 5  # True
3 <= 3  # True

# Chaining comparisons
1 < 5 < 10  # True (same as: 1 < 5 and 5 < 10)
```

3.3.2 Error Type 3: Wrong Comparison Operator

What Happened: Using = instead of ==, or other comparison mistakes.

Code Example - WRONG:

```
# Using = instead of ==
if x = 5:  # ERROR! SyntaxError
    print("Five")

# Comparing with wrong type
if "5" == 5:  # False (no error, but wrong logic)
```

```
    print("Equal")  # Doesn't print

# Using is for value comparison
x = 1000
y = 1000
if x is y:  # False (checks identity, not value)
    print("Same")  # Doesn't print

# Wrong negation
if not x == 5:  # Works but verbose
    print("Not five")
```

Code Example - CORRECT:

```
# Use == for comparison
x = 5
if x == 5:  #   Correct
    print("Five")

# Convert types before comparing
if int("5") == 5:  #   True
    print("Equal")

# Use == for value comparison
x = 1000
y = 1000
if x == y:  #   True
    print("Same")

# Use is only for None, True, False
value = None
if value is None:  #   Correct
    print("No value")

# Use != for not equal
if x != 5:  #   Better than: not x == 5
    print("Not five")

# String comparison (case-sensitive)
name = "Alice"
if name == "Alice":  #   True
    print("Hello Alice")

if name.lower() == "alice":  #   Case-insensitive
    print("Hello Alice")
```

Comparison Best Practices:

```
# For numbers: use ==, !=, <, >, <=, >=
age = 25
if age >= 18:  #
    print("Adult")

# For None: use is/is not
value = None
if value is None:  #  Preferred
    print("No value")

# For strings: use ==, consider case
name = "Alice"
if name == "Alice":  #  Case-sensitive
    print("Match")

if name.lower() == "alice":  #  Case-insensitive
    print("Match")

# For booleans: use == or direct
is_valid = True
if is_valid:  #  Direct
    print("Valid")

if is_valid == True:  #  Works but verbose
    print("Valid")

# For collections: use == for value, is for identity
list1 = [1, 2, 3]
list2 = [1, 2, 3]
if list1 == list2:  #  True (same values)
    print("Same values")

if list1 is list2:  # False (different objects)
    print("Same object")
```

3.4 2.3 Logical Operators

3.4.1 Understanding and, or, not

```
# and - Both must be True
True and True  # True
True and False  # False
```

```
False and False  # False

# or - At least one must be True
True or False  # True
False or False  # False
True or True  # True

# not - Reverses the boolean
not True  # False
not False  # True

# Combining operators
x = 10
if x > 5 and x < 15:  # True
    print("Between 5 and 15")

# Short-circuit evaluation
if x > 5 and expensive_function():  # expensive_function only called if x > 5
    print("Both conditions true")
```

3.4.2 Error Type 4: Logical Operator Mistakes

Code Example - WRONG:

```
# Using bitwise operators instead of logical
if True & False:  # Using & instead of 'and'
    print("Wrong operator")

# Wrong order of operations
if not x == 5:  # Confusing - applies 'not' first
    print("Not five")

# Chaining with wrong logic
age = 25
if age > 18 and < 65:  # ERROR! SyntaxError
    print("Working age")

# Missing parentheses
if x > 5 and y > 3 or z > 10:  # Ambiguous
    print("Condition met")

# Comparing with boolean literals unnecessarily
if is_valid == True:  # Verbose
    print("Valid")
```

Code Example - CORRECT:

```
# Use logical operators correctly
if True and False:  #  Logical AND
    print("Both true")

# Clear negation
if x != 5:  #  Better than: not x == 5
    print("Not five")

# Proper chaining
age = 25
if age > 18 and age < 65:  #  Correct
    print("Working age")

# Or use chaining
if 18 < age < 65:  #  More Pythonic
    print("Working age")

# Use parentheses for clarity
if (x > 5 and y > 3) or z > 10:  #  Clear intent
    print("Condition met")

# Direct boolean check
if is_valid:  #  Simple and clear
    print("Valid")

# Combining conditions clearly
username = "admin"
password = "secret"
if username == "admin" and password == "secret":  #
    print("Login successful")

# Short-circuit evaluation
if user is not None and user.is_active():  #  Safe
    print("Active user")
```

Logical Operator Truth Tables:

```
# AND truth table
True and True    # True
True and False   # False
False and True   # False
False and False  # False

# OR truth table
True or True     # True
```

```
True or False    # True
False or True    # True
False or False   # False

# NOT truth table
not True         # False
not False        # True

# Combining operators
not (True and False)  # True
True and not False    # True
False or not False    # True
```

3.5 2.4 Assignment Operators

3.5.1 Compound Assignment

```
# Basic assignment
x = 5

# Addition assignment
x += 3  # Same as: x = x + 3
# x is now 8

# Subtraction assignment
x -= 2  # Same as: x = x - 2
# x is now 6

# Multiplication assignment
x *= 2  # Same as: x = x * 2
# x is now 12

# Division assignment
x /= 3  # Same as: x = x / 3
# x is now 4.0

# Floor division assignment
x //= 2  # Same as: x = x // 2
# x is now 2.0

# Modulo assignment
x %= 2  # Same as: x = x % 2
# x is now 0.0
```

```
# Exponentiation assignment
x = 2
x **= 3  # Same as: x = x ** 3
# x is now 8
```

3.5.2 Error Type 5: Assignment Mistakes

Code Example - WRONG:

```
# Multiple assignment with different types
x = y = "5"
x += 5  # ERROR! TypeError: can only concatenate str (not "int") to str

# Undefined variable in compound assignment
total += 10  # ERROR! NameError if total not defined

# Wrong operator order
5 = x  # ERROR! SyntaxError: can't assign to literal

# Modifying immutable
x = 5
x[0] = 3  # ERROR! TypeError: 'int' object does not support item assignment
```

Code Example - CORRECT:

```
# Initialize before compound assignment
total = 0  #   Initialize first
total += 10  #   Now works: total is 10

# Type-consistent operations
x = 5  # Integer
x += 3  #   Still integer: 8

y = "5"  # String
y += "3"  #   String concatenation: "53"

# Correct assignment order
x = 5  #   Variable on left, value on right

# Multiple assignment with same type
x = y = z = 0  #   All set to 0
x += 5  #   x is now 5

# Counter pattern
```

```python
count = 0
count += 1  # Increment
count -= 1  # Decrement

# Accumulator pattern
total = 0
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    total += num  # Accumulate sum
# total is 15

# String building (though join is better for large strings)
message = ""
message += "Hello"
message += " "
message += "World"
# message is "Hello World"
```

3.6 2.5 Operator Precedence

3.6.1 Order of Operations

```python
# Python operator precedence (high to low):
# 1. ** (exponentiation)
# 2. *, /, //, % (multiplication, division, floor division, modulo)
# 3. +, - (addition, subtraction)
# 4. ==, !=, <, >, <=, >= (comparisons)
# 5. not
# 6. and
# 7. or

# Examples
result = 2 + 3 * 4  # 14 (not 20) - multiplication first
result = (2 + 3) * 4  # 20 - parentheses override

result = 10 - 3 - 2  # 5 (left to right: 10-3=7, 7-2=5)
result = 2 ** 3 ** 2  # 512 (right to left: 3**2=9, 2**9=512)

# Comparison chaining
1 < 2 < 3  # True (same as: 1 < 2 and 2 < 3)

# Logical operators
```

```
True or False and False  # True (and before or)
(True or False) and False  # False (parentheses first)
```

3.6.2 Error Type 6: Precedence Confusion

Code Example - WRONG:

```
# Expecting left-to-right for all operators
result = 2 ** 3 ** 2  # 512, not 64 (exponentiation is right-to-left)

# Forgetting multiplication before addition
result = 2 + 3 * 4  # 14, not 20

# Logical operator precedence
if x > 5 or y > 3 and z > 10:  # 'and' has higher precedence
    # This is: x > 5 or (y > 3 and z > 10)
    # Not: (x > 5 or y > 3) and z > 10
    print("Condition met")

# Comparison with calculations
if 5 + 5 == 10:  # Works, but can be confusing
    print("Equal")
```

Code Example - CORRECT:

```
# Use parentheses for clarity
result = 2 ** (3 ** 2)  #   512 (explicit right-to-left)
result = (2 ** 3) ** 2  #   64 (left-to-right)

result = (2 + 3) * 4  #   20 (addition first)

# Clear logical expressions
if (x > 5) or (y > 3 and z > 10):  #   Explicit grouping
    print("Condition met")

# Separate calculation and comparison
sum_value = 5 + 5
if sum_value == 10:  #   More readable
    print("Equal")

# Complex expressions with clear grouping
result = ((10 + 5) * 2) / (3 + 2)  #   Very clear
# = (15 * 2) / 5
# = 30 / 5
# = 6.0
```

```
# Mathematical formula with proper precedence
# Formula: (a + b) / (c - d)
a, b, c, d = 10, 5, 8, 3
result = (a + b) / (c - d)  #   Clear grouping
```

Best Practice:

```
# When in doubt, use parentheses!
# Better to be explicit than to rely on precedence rules

# Unclear
result = 2 + 3 * 4 - 5 / 2

# Clear
result = 2 + (3 * 4) - (5 / 2)
```

3.7 2.6 Practice Problems - Fix These Errors!

3.7.1 Problem 1: Division by Zero

```
x = 10
y = 0
result = x / y
```

Click for Answer

Error: `ZeroDivisionError: division by zero`

Why: Dividing by zero

Fix:

```
x = 10
y = 0

# Check before dividing
if y != 0:
    result = x / y
else:
    result = 0  # Or None, or handle error
    print("Cannot divide by zero")

# Or use try/except
try:
    result = x / y
```

```
except ZeroDivisionError:
    result = 0
    print("Cannot divide by zero")
```

3.7.2 Problem 2: Type Mismatch

```
age = "25"
next_year = age + 1
```

Click for Answer

Error: `TypeError: can only concatenate str (not "int") to str`

Why: Can't add string and integer

Fix:

```
age = "25"
next_year = int(age) + 1  #  Convert to int first
print(next_year)  # 26
```

3.7.3 Problem 3: Wrong Comparison

```
x = 5
if x = 5:
    print("Five")
```

Click for Answer

Error: `SyntaxError: invalid syntax`

Why: Using = (assignment) instead of == (comparison)

Fix:

```
x = 5
if x == 5:  #  Use == for comparison
    print("Five")
```

3.7.4 Problem 4: Operator Precedence

```
result = 10 - 5 - 3
print(result)  # What does this print?
```

Click for Answer

Answer: 2

Explanation: Subtraction is left-to-right - 10 - 5 = 5 - 5 - 3 = 2

Not: 10 - (5 - 3) = 10 - 2 = 8

To get 8:

```
result = 10 - (5 - 3)  #  Use parentheses
```

3.7.5 Problem 5: Logical Operators

```
age = 25
if age > 18 and < 65:
    print("Working age")
```

Click for Answer

Error: `SyntaxError: invalid syntax`

Why: Missing variable in second comparison

Fix:

```
age = 25

# Option 1: Repeat variable
if age > 18 and age < 65:  #
    print("Working age")

# Option 2: Chaining (more Pythonic)
if 18 < age < 65:  #
    print("Working age")
```

3.7.6 Problem 6: String Multiplication

```
result = "hello" * "3"
```

Click for Answer

Error: `TypeError: can't multiply sequence by non-int of type 'str'`

Why: Can't multiply string by string

Fix:

```
result = "hello" * 3  #   Multiply by integer
print(result)  # "hellohellohello"

# If "3" is user input:
count = "3"
result = "hello" * int(count)  #   Convert to int first
```

3.8 2.7 Key Takeaways

3.8.1 What You Learned

1. **Check for zero before division** - Prevent ZeroDivisionError
2. **Match types in operations** - Convert when needed
3. **Use == for comparison** - Not = (assignment)
4. **Understand operator precedence** - Use parentheses when unclear
5. **Use logical operators correctly** - and, or, not
6. **Initialize before compound assignment** - total += 1 needs total to exist
7. **Use is for None/True/False** - Use == for other values

3.8.2 Common Patterns

```
# Pattern 1: Safe division
if denominator != 0:
    result = numerator / denominator

# Pattern 2: Type conversion
value = int(string_value) + 5

# Pattern 3: Range checking
if 0 <= value <= 100:
    # Value in range

# Pattern 4: Null checking with logic
if user is not None and user.is_active():
    # Safe to call method

# Pattern 5: Counter
count = 0
count += 1
```

3.8.3 Error Summary Table

Error Type	Common Cause	Prevention
`ZeroDivisionError`	Dividing by zero	Check denominator != 0
`TypeError` in operations	Mixing incompatible types	Convert types first
`SyntaxError` in if	Using = instead of ==	Use == for comparison
Wrong results	Operator precedence	Use parentheses

3.9 2.8 Moving Forward

You now understand operators and expressions. You can: - Perform arithmetic safely - Compare values correctly - Use logical operators effectively - Avoid common operator errors

In **Chapter 3**, we'll explore **Strings and String Methods** - working with text and avoiding string errors!

Chapter 4

Chapter 3: Strings and String Methods - Text Manipulation Errors

4.1 Introduction

You've mastered variables and operators. Now let's explore **strings** - one of the most commonly used data types in Python. Strings represent text, and Python provides powerful methods to manipulate them.

String-related errors are extremely common: - **IndexError**: Accessing invalid string positions - **AttributeError**: Using wrong methods or typos - **ValueError**: Invalid operations on strings - **TypeError**: Wrong types in string operations

Let's master strings by understanding their errors!

4.2 3.1 String Basics

4.2.1 Creating and Using Strings

```
# Creating strings
name = "Alice"
message = 'Hello, World!'
multiline = """This is
a multiline
string"""
```

```
# String concatenation
greeting = "Hello" + " " + "World"  # "Hello World"

# String repetition
repeated = "Ha" * 3  # "HaHaHa"

# String length
length = len("Python")  # 6

# Accessing characters (0-indexed)
text = "Python"
first = text[0]   # 'P'
last = text[-1]   # 'n'
```

4.2.2 Error Type 1: `IndexError: string index out of range`

Error Message:

```
>>> text = "Hello"
>>> char = text[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

What Happened: You tried to access a character at an index that doesn't exist in the string.

Why It Happens: - Index beyond string length - Empty string - Off-by-one errors (forgetting 0-based indexing) - Negative index too large

Code Example - WRONG:

```
text = "Hello"  # Length 5, indices 0-4

# Index too large
char = text[5]  # ERROR! Valid indices: 0-4

# Much too large
char = text[100]  # ERROR!

# Empty string
empty = ""
char = empty[0]  # ERROR! No characters

# Wrong assumption
```

```python
first = text[1]  # Gets 'e', not 'H' (0-based indexing!)

# Negative index too large
char = text[-10]  # ERROR! Only goes back to -5
```

Code Example - CORRECT:

```python
text = "Hello"

# Use valid indices
first = text[0]   #   'H' (first character)
last = text[4]    #   'o' (last character)
last = text[-1]   #   'o' (last character, better way)

# Check length before accessing
index = 10
if index < len(text):
    char = text[index]
else:
    print(f"Index {index} out of range")

# Safe access with try/except
try:
    char = text[10]
except IndexError:
    char = None  # Or default value
    print("Index out of range")

# Use slicing (doesn't raise error)
substring = text[10:15]  #   Returns empty string, no error

# Get last character safely
if text:  # Check if not empty
    last = text[-1]
else:
    last = None

# Iterate safely with indices
for i in range(len(text)):
    char = text[i]  #   Always valid
    print(char)

# Or iterate directly (better)
for char in text:  #   No indexing needed
    print(char)
```

String Indexing Reference:

```
text = "Python"
#       012345  (positive indices)
#      -654321  (negative indices)

text[0]    # 'P' (first)
text[5]    # 'n' (last)
text[-1]   # 'n' (last, using negative)
text[-6]   # 'P' (first, using negative)

# Valid range: 0 to len(text)-1
# Or: -len(text) to -1
```

4.3 3.2 String Slicing

4.3.1 Understanding Slicing

```
text = "Python"

# Basic slicing [start:stop]
text[0:3]   # 'Pyt' (indices 0, 1, 2)
text[2:5]   # 'tho' (indices 2, 3, 4)

# Omitting start or stop
text[:3]    # 'Pyt' (from beginning)
text[3:]    # 'hon' (to end)
text[:]     # 'Python' (entire string)

# Step parameter [start:stop:step]
text[::2]   # 'Pto' (every 2nd character)
text[1::2]  # 'yhn' (every 2nd, starting at 1)

# Negative step (reverse)
text[::-1]  # 'nohtyP' (reversed)

# Negative indices in slicing
text[-3:]   # 'hon' (last 3 characters)
text[:-2]   # 'Pyth' (all but last 2)
```

4.3.2 Error Type 2: Slicing Mistakes (No Error, But Wrong Results)

What Happened: Slicing doesn't raise errors, but wrong indices give unexpected results.

Code Example - WRONG LOGIC:

```
text = "Hello World"

# Getting empty string unexpectedly
substring = text[5:3]  # '' (start > stop gives empty)

# Wrong order
substring = text[10:0]  # '' (should be text[0:10])

# Confusing positive and negative
substring = text[-1:0]  # '' (wrong direction)

# Off-by-one errors
# Want "Hello" (first 5 chars)
substring = text[0:4]  # 'Hell' (missing last char!)

# Want "World" (last 5 chars)
substring = text[6:10]  # 'Worl' (missing last char!)
```

Code Example - CORRECT:

```
text = "Hello World"

# Get first N characters
first_5 = text[:5]  #   'Hello'

# Get last N characters
last_5 = text[-5:]  #   'World'

# Get middle portion
middle = text[6:11]  #   'World' (or text[6:])

# Remove first N characters
without_first_6 = text[6:]  #   'World'

# Remove last N characters
without_last_6 = text[:-6]  #   'Hello'

# Get every 2nd character
every_2nd = text[::2]  #   'HloWrd'
```

```
# Reverse string
reversed_text = text[::-1]  #   'dlroW olleH'

# Safe slicing (never errors)
substring = text[100:200]  #   '' (empty, no error)

# Get substring between positions
start = 0
end = 5
substring = text[start:end]  #   'Hello'

# Extract file extension
filename = "document.txt"
extension = filename[filename.rfind('.'):]  #   '.txt'
# Or better:
extension = filename.split('.')[-1]  #   'txt'
```

Slicing Patterns:

```
text = "Python Programming"

# First word
first_word = text.split()[0]  # 'Python'
# Or: text[:text.find(' ')]

# Last word
last_word = text.split()[-1]  # 'Programming'
# Or: text[text.rfind(' ')+1:]

# First N characters
text[:5]  # 'Pytho'

# Last N characters
text[-5:]  # 'mming'

# Middle portion
text[7:18]  # 'Programming'

# Remove whitespace from ends
text.strip()  # Removes leading/trailing spaces

# Reverse
text[::-1]  # 'gnimmargorP nohtyP'
```

4.4 3.3 String Methods

4.4.1 Common String Methods

```
text = "Hello World"

# Case conversion
text.upper()        # 'HELLO WORLD'
text.lower()        # 'hello world'
text.capitalize()   # 'Hello world'
text.title()        # 'Hello World'

# Checking content
text.startswith('Hello')  # True
text.endswith('World')    # True
text.isalpha()      # False (has space)
text.isdigit()      # False
text.isalnum()      # False (has space)

# Finding substrings
text.find('World')   # 6 (index where found)
text.find('Python')  # -1 (not found)
text.index('World')  # 6 (raises ValueError if not found)

# Replacing
text.replace('World', 'Python')  # 'Hello Python'

# Splitting and joining
words = text.split()  # ['Hello', 'World']
joined = ' '.join(words)  # 'Hello World'

# Stripping whitespace
"  hello  ".strip()   # 'hello'
"  hello  ".lstrip()  # 'hello  '
"  hello  ".rstrip()  # '  hello'
```

4.4.2 Error Type 3: `AttributeError: 'str' object has no attribute 'X'`

Error Message:

```
>>> text = "hello"
>>> text.append("!")
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
AttributeError: 'str' object has no attribute 'append'
```

What Happened: You tried to use a method that doesn't exist for strings, or you made a typo.

Why It Happens: - Using list methods on strings - Typo in method name - Confusing similar methods - Wrong object type

Code Example - WRONG:

```
text = "hello"

# List methods don't work on strings
text.append("!")  # ERROR! Strings don't have append

# Typo in method name
text.uppper()  # ERROR! Typo: should be upper()

# Wrong method
text.add("world")  # ERROR! No 'add' method

# Trying to modify immutably
text[0] = 'H'  # ERROR! Strings are immutable

# Wrong assumptions
text.remove('l')  # ERROR! Strings don't have remove
```

Code Example - CORRECT:

```
text = "hello"

# Use string concatenation instead of append
text = text + "!"  #   "hello!"
# Or use +=
text += "!"  #

# Correct method names
text.upper()  #   "HELLO"
text.lower()  #   "hello"

# Use replace to "modify" strings
text = text.replace('h', 'H')  #   "Hello"

# Remove characters with replace
text = text.replace('l', '')  #   "heo" (removes all 'l')

# Create new string instead of modifying
```

```python
text = "hello"
new_text = 'H' + text[1:]  #   "Hello"

# Check if method exists
if hasattr(text, 'upper'):
    result = text.upper()  #

# Common string methods (not list methods)
text = "hello world"
text.split()          #   Returns list: ['hello', 'world']
text.replace('o', '0') #   Returns: 'hell0 w0rld'
text.find('world')    #   Returns: 6
text.startswith('h')  #   Returns: True
text.strip()          #   Removes whitespace
text.count('l')       #   Returns: 3
```

String Method Reference:

```python
text = "Hello World"

# Case methods
text.upper()      # HELLO WORLD
text.lower()      # hello world
text.capitalize() # Hello world
text.title()      # Hello World
text.swapcase()   # hELLO wORLD

# Checking methods (return bool)
text.isalpha()    # False (has space)
text.isdigit()    # False
text.isalnum()    # False
text.isspace()    # False
text.isupper()    # False
text.islower()    # False
text.istitle()    # True

# Search methods
text.find('o')        # 4 (first occurrence)
text.rfind('o')       # 7 (last occurrence)
text.index('o')       # 4 (like find, but raises ValueError if not found)
text.count('o')       # 2
text.startswith('He') # True
text.endswith('ld')   # True

# Modification methods (return new string)
text.replace('World', 'Python')  # Hello Python
```

```
text.strip()          # Removes whitespace
text.lstrip()         # Removes left whitespace
text.rstrip()         # Removes right whitespace

# Split and join
text.split()          # ['Hello', 'World']
text.split('o')       # ['Hell', ' W', 'rld']
' '.join(['a', 'b'])  # 'a b'

# Padding and alignment
text.center(20)       # '    Hello World     '
text.ljust(20)        # 'Hello World         '
text.rjust(20)        # '         Hello World'
text.zfill(20)        # '000000000Hello World'
```

4.5 3.4 String Formatting

4.5.1 String Formatting Methods

```
# f-strings (Python 3.6+) - RECOMMENDED
name = "Alice"
age = 25
message = f"My name is {name} and I'm {age} years old"
# "My name is Alice and I'm 25 years old"

# format() method
message = "My name is {} and I'm {} years old".format(name, age)

# %-formatting (old style)
message = "My name is %s and I'm %d years old" % (name, age)

# Formatting numbers
pi = 3.14159
formatted = f"Pi is {pi:.2f}"  # "Pi is 3.14"

# Formatting with width
number = 42
formatted = f"Number: {number:5d}"  # "Number:    42"
```

4.5.2 Error Type 4: `ValueError: invalid format string`

Error Message:

```
>>> name = "Alice"
>>> message = f"Hello {nme}"  # Typo in variable name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'nme' is not defined
```

What Happened: Errors in f-strings or format strings.

Code Example - WRONG:

```
# Typo in variable name
name = "Alice"
message = f"Hello {nme}"  # ERROR! 'nme' not defined

# Wrong number of arguments in format()
template = "Hello {} and {}"
message = template.format("Alice")  # ERROR! Needs 2 arguments

# Wrong format specifier
number = 42
formatted = f"{number:.2f}"  # ERROR! Can't format int as float
# (Actually works in Python 3, but conceptually wrong)

# Unclosed brace
message = f"Hello {name"  # ERROR! SyntaxError: unclosed '{'

# Wrong %-formatting
name = "Alice"
age = 25
message = "Name: %s, Age: %s" % name  # ERROR! Needs tuple with 2 items
```

Code Example - CORRECT:

```
# Correct variable name
name = "Alice"
message = f"Hello {name}"  #

# Correct number of arguments
template = "Hello {} and {}"
message = template.format("Alice", "Bob")  #

# Correct format specifier for float
number = 42.5
formatted = f"{number:.2f}"  #  "42.50"
```

```
# Convert int to float if needed
number = 42
formatted = f"{float(number):.2f}"  #   "42.00"

# Closed braces
message = f"Hello {name}"  #

# Correct %-formatting
name = "Alice"
age = 25
message = "Name: %s, Age: %d" % (name, age)  #   Tuple

# f-string with expressions
x = 10
y = 20
message = f"Sum is {x + y}"  #   "Sum is 30"

# Format with padding
number = 42
formatted = f"{number:05d}"  #   "00042" (5 digits, zero-padded)

# Multiple variables
first = "Alice"
last = "Smith"
age = 25
message = f"{first} {last} is {age} years old"  #

# Format numbers in f-strings
price = 19.99
message = f"Price: ${price:.2f}"  #   "Price: $19.99"

# Use format() safely
message = "Hello {}".format(name)  #
message = "Hello {name}".format(name=name)  #   Named

# Escape braces
message = f"Use {{braces}} in f-strings"  #   "Use {braces} in f-strings"
```

Format Specifier Reference:

```
number = 42
pi = 3.14159
text = "hello"

# Integer formatting
f"{number:d}"     # '42' (decimal)
```

```
f"{number:5d}"    # '   42' (width 5)
f"{number:05d}"   # '00042' (zero-padded)

# Float formatting
f"{pi:f}"         # '3.141590' (default 6 decimals)
f"{pi:.2f}"       # '3.14' (2 decimals)
f"{pi:8.2f}"      # '    3.14' (width 8, 2 decimals)

# String formatting
f"{text:s}"       # 'hello'
f"{text:>10s}"    # '     hello' (right-aligned, width 10)
f"{text:<10s}"    # 'hello     ' (left-aligned)
f"{text:^10s}"    # '  hello   ' (centered)

# Percentage
value = 0.85
f"{value:.1%}"    # '85.0%'

# Scientific notation
large = 1000000
f"{large:e}"      # '1.000000e+06'
f"{large:.2e}"    # '1.00e+06'
```

4.6 3.5 String Immutability

4.6.1 Understanding Immutability

```
# Strings are immutable - cannot be changed
text = "hello"

# This creates a NEW string
text = text.upper()  # "HELLO"

# Original string unchanged (if referenced elsewhere)
original = "hello"
modified = original.upper()
print(original)  # Still "hello"
print(modified)  # "HELLO"
```

4.6.2 Error Type 5: `TypeError: 'str' object does not support item assignment`

Error Message:

```
>>> text = "hello"
>>> text[0] = 'H'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

What Happened: You tried to change a character in a string. Strings are immutable in Python.

Why It Happens: - Trying to modify string directly - Treating string like a list - Not understanding immutability

Code Example - WRONG:

```
text = "hello"

# Can't modify characters
text[0] = 'H'  # ERROR! Strings are immutable

# Can't delete characters
del text[0]  # ERROR! Can't delete from immutable

# Can't use list methods that modify
text.append('!')  # ERROR! No such method
text.remove('l')  # ERROR! No such method
text.insert(0, 'H')  # ERROR! No such method
```

Code Example - CORRECT:

```
text = "hello"

# Create new string with change
text = 'H' + text[1:]  #   "Hello"

# Use string methods that return new strings
text = text.replace('h', 'H')  #   "Hello"

# Build new string
text = "hello"
new_text = ""
for char in text:
    if char == 'h':
        new_text += 'H'
    else:
```

```
        new_text += char
# new_text is "Hello"

# Use list for modifications, then join
text = "hello"
chars = list(text)  # ['h', 'e', 'l', 'l', 'o']
chars[0] = 'H'      #  Lists are mutable
text = ''.join(chars)  #   "Hello"

# Multiple replacements
text = "hello world"
text = text.replace('h', 'H').replace('w', 'W')
# "Hello World"

# Remove characters
text = "hello"
text = text.replace('l', '')  #   "heo"

# Insert characters (create new string)
text = "heo"
text = text[:2] + 'll' + text[2:]  #   "hello"

# Reverse string (creates new)
text = "hello"
reversed_text = text[::-1]  #   "olleh"
```

Why Immutability Matters:

```
# Immutability makes strings safe
def process_text(text):
    text = text.upper()  # Creates NEW string
    return text

original = "hello"
result = process_text(original)
print(original)  # Still "hello" - unchanged
print(result)    # "HELLO" - new string

# Multiple references to same string
text1 = "hello"
text2 = text1
text1 = text1.upper()  # Creates NEW string
print(text1)  # "HELLO"
print(text2)  # "hello" - unchanged

# This is different from lists (mutable)
```

```
list1 = [1, 2, 3]
list2 = list1
list1.append(4)  # Modifies same list
print(list1)  # [1, 2, 3, 4]
print(list2)  # [1, 2, 3, 4] - also changed!
```

4.7 3.6 Common String Operations

4.7.1 Checking String Content

```
text = "Hello123"

# Check if all characters are letters
text.isalpha()  # False (has digits)

# Check if all characters are digits
text.isdigit()  # False (has letters)

# Check if alphanumeric (letters and digits)
text.isalnum()  # True

# Check if all lowercase
"hello".islower()  # True
"Hello".islower()  # False

# Check if all uppercase
"HELLO".isupper()  # True

# Check if empty
text = ""
if text:  # False - empty string is falsy
    print("Has content")
else:
    print("Empty")

# Check if whitespace only
"   ".isspace()  # True
"hello".isspace()  # False
```

4.7.2 Error Type 6: String Comparison Pitfalls

Code Example - WRONG LOGIC:

```
# Case-sensitive comparison
name = "Alice"
if name == "alice":  # False!
    print("Match")

# Leading/trailing whitespace
text = "  hello  "
if text == "hello":  # False!
    print("Match")

# Type mismatch
number = "5"
if number == 5:  # False! String vs int
    print("Match")

# Empty string checks
text = ""
if text == None:  # False! Empty string is not None
    print("None")
```

Code Example - CORRECT:

```
# Case-insensitive comparison
name = "Alice"
if name.lower() == "alice":  #   True
    print("Match")

# Strip whitespace first
text = "  hello  "
if text.strip() == "hello":  #   True
    print("Match")

# Convert types before comparing
number = "5"
if int(number) == 5:  #   True
    print("Match")

# Or convert the other way
if number == str(5):  #   True
    print("Match")

# Check for empty string
text = ""
```

```
if not text:  #   True - empty string is falsy
    print("Empty")

# Or explicitly
if text == "":  #   True
    print("Empty")

# Check for None
value = None
if value is None:  #   True
    print("None")

# Check for empty or None
if not text or text is None:  #
    print("Empty or None")

# Substring checking
text = "Hello World"
if "World" in text:  #   True
    print("Contains 'World'")

# Starts with / ends with
filename = "document.txt"
if filename.endswith('.txt'):  #   True
    print("Text file")
```

4.8 3.7 Practice Problems - Fix These Errors!

4.8.1 Problem 1: Index Out of Range

```
text = "Python"
print(text[6])
```

Click for Answer

Error: `IndexError: string index out of range`

Why: String has indices 0-5, trying to access index 6

Fix:

```
text = "Python"
print(text[5])  #   Last character: 'n'
# Or
print(text[-1])  #   Better: 'n'
```

4.8.2 Problem 2: Wrong Method

```
text = "hello"
text.append(" world")
```

Click for Answer

Error: `AttributeError: 'str' object has no attribute 'append'`

Why: Strings don't have append method (lists do)

Fix:

```
text = "hello"
text = text + " world"  #  Concatenation
# Or
text += " world"  #  Also works
print(text)  # "hello world"
```

4.8.3 Problem 3: String Immutability

```
text = "hello"
text[0] = 'H'
```

Click for Answer

Error: `TypeError: 'str' object does not support item assignment`

Why: Strings are immutable

Fix:

```
text = "hello"
text = 'H' + text[1:]  #  Create new string
print(text)  # "Hello"

# Or use replace
text = "hello"
text = text.replace('h', 'H')  #  "Hello"
```

4.8.4 Problem 4: Format String Error

```
name = "Alice"
age = 25
message = f"Hello {name}, you are {ag} years old"
```

Click for Answer

Error: `NameError: name 'ag' is not defined`

Why: Typo in variable name inside f-string

Fix:

```
name = "Alice"
age = 25
message = f"Hello {name}, you are {age} years old"  #  Correct variable name
print(message)  # "Hello Alice, you are 25 years old"
```

4.8.5 Problem 5: Comparison Error

```
text = "  hello  "
if text == "hello":
    print("Match")
else:
    print("No match")
```

Click for Answer

Issue: Prints "No match" due to whitespace

Fix:

```
text = "  hello  "
if text.strip() == "hello":  #  Strip whitespace first
    print("Match")
else:
    print("No match")
# Prints: "Match"
```

4.9 3.8 Key Takeaways

4.9.1 What You Learned

1. **Check string length before indexing -** Avoid IndexError
2. **Use correct string methods -** Strings don't have list methods
3. **Strings are immutable -** Create new strings, don't modify

4. **Use f-strings for formatting** - Modern and clear
5. **Strip whitespace when comparing** - Avoid comparison issues
6. **Case-insensitive comparison** - Use .lower() or .upper()
7. **Slicing never errors** - But check logic for correct results

4.9.2 Common Patterns

```
# Pattern 1: Safe character access
if len(text) > index:
    char = text[index]

# Pattern 2: Case-insensitive comparison
if text.lower() == "hello":
    print("Match")

# Pattern 3: Clean and compare
if text.strip() == "expected":
    print("Match")

# Pattern 4: Create new string from old
text = 'H' + text[1:]

# Pattern 5: Check empty string
if not text:
    print("Empty")

# Pattern 6: Safe substring check
if "substring" in text:
    print("Found")
```

4.9.3 Error Summary Table

Error Type	Common Cause	Prevention
`IndexError`	Index beyond string length	Check len() first
`AttributeError`	Wrong method or typo	Use correct string methods
`TypeError`	Trying to modify string	Create new string instead
`ValueError`	Wrong format string	Check variable names and format

4.10 3.9 Moving Forward

You now understand strings and string methods. You can: - Access characters safely - Use string methods correctly - Format strings properly - Handle immutability - Compare strings accurately

In **Chapter 4**, we'll explore **Lists and List Methods** - working with collections and avoiding list errors!

Chapter 5

Chapter 4: Lists and List Methods - Collection Errors

5.1 Introduction

You've mastered strings. Now let's explore **lists** - Python's most versatile collection type. Lists store multiple items in a single variable and are **mutable** (unlike strings).

Common list errors: - **IndexError**: Accessing invalid positions - **ValueError**: Item not found - **TypeError**: Wrong operations or types - **AttributeError**: Wrong methods

Lists are everywhere in Python. Let's master them by understanding their errors!

5.2 4.1 List Basics

5.2.1 Creating and Using Lists

```
# Creating lists
numbers = [1, 2, 3, 4, 5]
names = ["Alice", "Bob", "Charlie"]
mixed = [1, "hello", 3.14, True]
empty = []

# List length
length = len(numbers)  # 5
```

```
# Accessing elements (0-indexed)
first = numbers[0]   # 1
last = numbers[-1]   # 5

# Modifying elements (lists are mutable!)
numbers[0] = 10  # [10, 2, 3, 4, 5]

# Adding elements
numbers.append(6)  # [10, 2, 3, 4, 5, 6]

# List concatenation
combined = [1, 2] + [3, 4]  # [1, 2, 3, 4]

# List repetition
repeated = [1, 2] * 3  # [1, 2, 1, 2, 1, 2]
```

5.2.2 Error Type 1: `IndexError: list index out of range`

Error Message:

```
>>> numbers = [1, 2, 3]
>>> print(numbers[5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

What Happened: You tried to access an index that doesn't exist in the list.

Why It Happens: - Index beyond list length - Empty list - Off-by-one errors - Wrong loop bounds

Code Example - WRONG:

```
numbers = [1, 2, 3]  # Length 3, indices 0-2

# Index too large
value = numbers[5]  # ERROR! Valid indices: 0-2

# Empty list
empty = []
value = empty[0]  # ERROR! No elements

# Off-by-one in loop
for i in range(len(numbers) + 1):  # Goes to 4!
    print(numbers[i])  # ERROR when i=3
```

```
# Wrong assumption
first = numbers[1]  # Gets 2, not 1 (0-based!)

# Negative index too large
value = numbers[-10]  # ERROR! Only -3 to -1 valid

# After removing elements
numbers = [1, 2, 3]
numbers.remove(2)  # Now [1, 3]
value = numbers[2]  # ERROR! Only 0-1 valid now
```

Code Example - CORRECT:

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

# Use valid indices
first = numbers[0]   #  1
last = numbers[2]    #  3
last = numbers[-1]   #  3 (better - works for any length)

# Check length before accessing
index = 5
if index < len(numbers):
    value = numbers[index]
else:
    print(f"Index {index} out of range")

# Safe access with try/except
try:
    value = numbers[5]
except IndexError:
    value = None
    print("Index out of range")

# Use proper loop bounds
for i in range(len(numbers)):  #  0, 1, 2
    print(numbers[i])

# Better: iterate directly (no indexing needed)
for num in numbers:  #  No index errors possible
    print(num)

# Safe get with default (custom function)
def safe_get(lst, index, default=None):
    """Safely get list item with default"""
    try:
```

```
        return lst[index]
    except IndexError:
        return default

value = safe_get(numbers, 5, default=0)  #   Returns 0

# Check before accessing after modifications
numbers = [1, 2, 3]
numbers.remove(2)
if len(numbers) > 2:
    value = numbers[2]
else:
    print("Not enough elements")

# Use slicing (doesn't raise errors)
subset = numbers[5:10]  #   Returns [], no error
```

List Indexing Reference:

```
numbers = [10, 20, 30, 40, 50]
#           0   1   2   3   4   (positive indices)
#          -5  -4  -3  -2  -1   (negative indices)

numbers[0]    # 10 (first)
numbers[4]    # 50 (last)
numbers[-1]   # 50 (last, better way)
numbers[-5]   # 10 (first)

# Valid range: 0 to len(numbers)-1
# Or: -len(numbers) to -1
```

5.3 4.2 List Slicing

5.3.1 Understanding List Slicing

```
numbers = [0, 1, 2, 3, 4, 5]

# Basic slicing [start:stop]
numbers[1:4]   # [1, 2, 3] (indices 1, 2, 3)
numbers[0:3]   # [0, 1, 2]

# Omitting start or stop
numbers[:3]    # [0, 1, 2] (from beginning)
```

```
numbers[3:]    # [3, 4, 5] (to end)
numbers[:]     # [0, 1, 2, 3, 4, 5] (copy entire list)

# Step parameter [start:stop:step]
numbers[::2]   # [0, 2, 4] (every 2nd element)
numbers[1::2]  # [1, 3, 5] (every 2nd, starting at 1)

# Negative step (reverse)
numbers[::-1]  # [5, 4, 3, 2, 1, 0] (reversed)

# Negative indices in slicing
numbers[-3:]   # [3, 4, 5] (last 3 elements)
numbers[:-2]   # [0, 1, 2, 3] (all but last 2)

# Modifying with slicing
numbers[1:3] = [10, 20]  # [0, 10, 20, 3, 4, 5]
```

5.4 4.3 List Methods

5.4.1 Common List Methods

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

# Adding elements
numbers.append(4)        # [1, 2, 3, 4] - add to end
numbers.insert(0, 0)     # [0, 1, 2, 3, 4] - add at position
numbers.extend([5, 6])   # [0, 1, 2, 3, 4, 5, 6] - add multiple

# Removing elements
numbers.remove(3)        # [0, 1, 2, 4, 5, 6] - remove first occurrence
popped = numbers.pop()   # [0, 1, 2, 4, 5] - remove and return last
popped = numbers.pop(0)  # [1, 2, 4, 5] - remove at index

# Finding elements
index = numbers.index(2)  # 1 - position of first occurrence
count = numbers.count(2)  # 1 - how many times it appears

# Sorting
numbers.sort()            # [1, 2, 4, 5] - sort in place
numbers.reverse()         # [5, 4, 2, 1] - reverse in place

# Clearing
```

```
numbers.clear()          # [] - remove all elements

# Copying
original = [1, 2, 3]
shallow_copy = original.copy()  # Creates independent copy
```

5.4.2 Error Type 2: `ValueError: X is not in list`

Error Message:

```
>>> numbers = [1, 2, 3]
>>> numbers.remove(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

What Happened: You tried to remove or find an item that doesn't exist in the list.

Why It Happens: - Using remove() on non-existent item - Using index() on non-existent item - Wrong value or type

Code Example - WRONG:

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

# Remove non-existent item
numbers.remove(5)  # ERROR! 5 not in list

# Find non-existent item
index = numbers.index(5)  # ERROR! 5 not in list

# Type mismatch
numbers = [1, 2, 3]
numbers.remove("1")  # ERROR! "1" (string) not same as 1 (int)

# After already removing
numbers = [1, 2, 3]
numbers.remove(2)  # [1, 3]
numbers.remove(2)  # ERROR! 2 already removed

# Case sensitivity with strings
names = ["Alice", "Bob"]
names.remove("alice")  # ERROR! Case doesn't match
```

Code Example - CORRECT:

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

# Check before removing
if 5 in numbers:
    numbers.remove(5)
else:
    print("5 not in list")  #

# Use try/except for remove
try:
    numbers.remove(5)
except ValueError:
    print("Item not found")  #

# Check before finding
if 5 in numbers:
    index = numbers.index(5)
else:
    index = -1  # or None

# Safe find function
def safe_index(lst, item, default=-1):
    """Safely find index with default"""
    try:
        return lst.index(item)
    except ValueError:
        return default

index = safe_index(numbers, 5, default=-1)  #  Returns -1

# Remove by index instead (if you know position)
if len(numbers) > 2:
    numbers.pop(2)  #  Removes item at index 2

# Remove all occurrences
numbers = [1, 2, 3, 2, 2]
while 2 in numbers:  #  Removes all 2s
    numbers.remove(2)
# Result: [1, 3]

# Or use list comprehension
numbers = [1, 2, 3, 2, 2]
numbers = [x for x in numbers if x != 2]  #  [1, 3]

# Case-insensitive removal for strings
```

```
names = ["Alice", "Bob"]
to_remove = "alice"
names = [n for n in names if n.lower() != to_remove.lower()]  #

# Use discard for sets (no error if not found)
number_set = {1, 2, 3}
number_set.discard(5)  #  No error - we'll learn sets in Ch 5
```

5.4.3 Error Type 3: `TypeError: list indices must be integers or slices, not X`

Error Message:

```
>>> numbers = [1, 2, 3]
>>> print(numbers["0"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers or slices, not str
```

What Happened: You tried to use a non-integer as a list index.

Why It Happens: - Using string instead of integer - Using float instead of integer - Forgetting to convert types

Code Example - WRONG:

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

# String index
value = numbers["0"]  # ERROR! Use 0, not "0"

# Float index
value = numbers[1.5]  # ERROR! Use int, not float

# Variable that's wrong type
index = "1"
value = numbers[index]  # ERROR! index is string

# User input (always string)
index = input("Enter index: ")  # "1"
value = numbers[index]  # ERROR! Need to convert
```

Code Example - CORRECT:

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

```python
# Use integer indices
value = numbers[0]  #   1

# Convert string to int
index = "1"
value = numbers[int(index)]  #   2

# Convert user input
index = input("Enter index: ")
try:
    index = int(index)
    value = numbers[index]
except (ValueError, IndexError) as e:
    print(f"Invalid index: {e}")

# Convert float to int (if needed)
index = 1.7
value = numbers[int(index)]  #   Uses 1

# Validate before converting
index_str = "1"
if index_str.isdigit():
    value = numbers[int(index_str)]  #
else:
    print("Invalid index")

# Safe index function
def get_by_index(lst, index):
    """Safely get item by index, handling type conversion"""
    try:
        # Convert to int if needed
        if not isinstance(index, int):
            index = int(index)
        return lst[index]
    except (ValueError, IndexError, TypeError) as e:
        return None

value = get_by_index(numbers, "1")  #   Returns 2
value = get_by_index(numbers, "10")  #   Returns None
```

5.5 4.4 List Mutability

5.5.1 Understanding Mutability

```
# Lists are mutable - can be changed in place
numbers = [1, 2, 3]
numbers[0] = 10  #  Changed: [10, 2, 3]
numbers.append(4)  #  Changed: [10, 2, 3, 4]

# This affects all references
list1 = [1, 2, 3]
list2 = list1  # list2 points to SAME list
list1.append(4)
print(list1)  # [1, 2, 3, 4]
print(list2)  # [1, 2, 3, 4] - also changed!

# To avoid this, create a copy
list1 = [1, 2, 3]
list2 = list1.copy()  # or list1[:]
list1.append(4)
print(list1)  # [1, 2, 3, 4]
print(list2)  # [1, 2, 3] - unchanged
```

5.5.2 Error Type 4: Unintended List Modification

What Happened: Modifying a list affects all references to that list.

Code Example - WRONG:

```
# Shared reference problem
def add_item(lst):
    lst.append(4)
    return lst

original = [1, 2, 3]
new_list = add_item(original)
print(original)  # [1, 2, 3, 4] - Oops! Original changed!

# Default mutable argument (dangerous!)
def add_to_list(item, lst=[]):
    lst.append(item)
    return lst

result1 = add_to_list(1)  # [1]
result2 = add_to_list(2)  # [1, 2] - Unexpected! Same list!
```

```
result3 = add_to_list(3)  # [1, 2, 3] - Still same list!

# Multiple references
list1 = [1, 2, 3]
list2 = list1
list3 = list1
list2.append(4)
print(list1)  # [1, 2, 3, 4]
print(list2)  # [1, 2, 3, 4]
print(list3)  # [1, 2, 3, 4] - All changed!
```

Code Example - CORRECT:

```
# Create a copy in function
def add_item(lst):
    new_list = lst.copy()  #   Create copy
    new_list.append(4)
    return new_list

original = [1, 2, 3]
new_list = add_item(original)
print(original)  # [1, 2, 3] - Unchanged
print(new_list)  # [1, 2, 3, 4]

# Use None as default, create new list inside
def add_to_list(item, lst=None):
    if lst is None:
        lst = []  #   Create new list each time
    lst.append(item)
    return lst

result1 = add_to_list(1)  # [1]
result2 = add_to_list(2)  # [2]   Different list
result3 = add_to_list(3)  # [3]   Different list

# Create independent copies
list1 = [1, 2, 3]
list2 = list1.copy()  #   or list1[:]
list3 = list1.copy()  #
list2.append(4)
print(list1)  # [1, 2, 3] - Unchanged
print(list2)  # [1, 2, 3, 4]
print(list3)  # [1, 2, 3] - Unchanged

# Deep copy for nested lists
import copy
```

```
nested = [[1, 2], [3, 4]]
shallow = nested.copy()  # Inner lists still shared
deep = copy.deepcopy(nested)  #   Complete independent copy

# Explicitly modify or return new
def process_list(lst, modify=False):
    """
    Process list - modify in place or return new
    """
    if modify:
        lst.append(4)
        return lst
    else:
        new_list = lst.copy()
        new_list.append(4)
        return new_list

original = [1, 2, 3]
new = process_list(original, modify=False)  #   Original safe
```

5.6 4.5 List Comprehensions

5.6.1 Understanding List Comprehensions

```
# Traditional loop
squares = []
for x in range(5):
    squares.append(x ** 2)
# [0, 1, 4, 9, 16]

# List comprehension (more Pythonic)
squares = [x ** 2 for x in range(5)]
# [0, 1, 4, 9, 16]

# With condition
evens = [x for x in range(10) if x % 2 == 0]
# [0, 2, 4, 6, 8]

# Transforming list
names = ["alice", "bob", "charlie"]
upper_names = [name.upper() for name in names]
# ["ALICE", "BOB", "CHARLIE"]
```

```python
# Nested comprehensions
matrix = [[i*j for j in range(3)] for i in range(3)]
# [[0, 0, 0], [0, 1, 2], [0, 2, 4]]
```

5.6.2 Error Type 5: List Comprehension Mistakes

Code Example - WRONG:

```python
# Syntax error - wrong order
squares = [for x in range(5) x ** 2]  # ERROR! Wrong order

# Missing brackets
squares = x ** 2 for x in range(5)  # ERROR! This makes generator, not list

# Using wrong variable
numbers = [1, 2, 3]
doubled = [x * 2 for num in numbers]  # ERROR! x not defined

# Complex logic without parentheses
result = [x for x in range(10) if x > 5 and < 8]  # ERROR! Syntax error

# Trying to use statements
result = [print(x) for x in range(5)]  # Works but prints None values
```

Code Example - CORRECT:

```python
# Correct syntax [expression for variable in iterable]
squares = [x ** 2 for x in range(5)]  #

# Add brackets for list
squares = [x ** 2 for x in range(5)]  #  List
# Or omit for generator (advanced)
squares_gen = (x ** 2 for x in range(5))  # Generator

# Use correct variable
numbers = [1, 2, 3]
doubled = [num * 2 for num in numbers]  #

# Complete conditions
result = [x for x in range(10) if x > 5 and x < 8]  #
# Or use chaining
result = [x for x in range(10) if 5 < x < 8]  #

# Use functions for side effects separately
```

```
for x in range(5):  #  Better for printing
    print(x)

# Complex comprehensions
# Filter and transform
numbers = [1, 2, 3, 4, 5, 6]
even_squares = [x ** 2 for x in numbers if x % 2 == 0]  #
# [4, 16, 36]

# Multiple conditions
result = [x for x in range(20) if x % 2 == 0 if x % 3 == 0]  #
# [0, 6, 12, 18]

# Nested comprehension
matrix = [[i+j for j in range(3)] for i in range(3)]  #
# [[0, 1, 2], [1, 2, 3], [2, 3, 4]]

# Flattening nested list
nested = [[1, 2], [3, 4], [5, 6]]
flat = [item for sublist in nested for item in sublist]  #
# [1, 2, 3, 4, 5, 6]
```

5.7 4.6 Common List Patterns

5.7.1 Useful List Operations

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

# Check if list is empty
if numbers:  #  True if not empty
    print("Has items")

if not numbers:  #  True if empty
    print("Empty")

# Check membership
if 3 in numbers:  #  True
    print("Found 3")

# Get min/max
minimum = min(numbers)  # 1
maximum = max(numbers)  # 5
```

```
# Sum all elements
total = sum(numbers)  # 15

# Count occurrences
numbers = [1, 2, 2, 3, 2]
count = numbers.count(2)  # 3

# Find all indices of value
indices = [i for i, x in enumerate(numbers) if x == 2]
# [1, 2, 4]

# Remove duplicates (preserves order)
numbers = [1, 2, 2, 3, 1, 4]
unique = list(dict.fromkeys(numbers))  # [1, 2, 3, 4]
# Or using set (may not preserve order)
unique = list(set(numbers))

# Zip multiple lists
names = ["Alice", "Bob"]
ages = [25, 30]
combined = list(zip(names, ages))
# [("Alice", 25), ("Bob", 30)]

# Sort without modifying original
numbers = [3, 1, 4, 1, 5]
sorted_numbers = sorted(numbers)  # [1, 1, 3, 4, 5]
print(numbers)  # [3, 1, 4, 1, 5] - unchanged
```

5.8 4.7 Practice Problems - Fix These Errors!

5.8.1 Problem 1: Index Out of Range

```
numbers = [10, 20, 30]
for i in range(len(numbers) + 1):
    print(numbers[i])
```

Click for Answer

Error: `IndexError: list index out of range`

Why: Loop goes to index 3, but list only has indices 0-2

Fix:

```
numbers = [10, 20, 30]
for i in range(len(numbers)):  #   Remove +1
    print(numbers[i])

# Or better - iterate directly:
for num in numbers:  #   No indexing needed
    print(num)
```

5.8.2 Problem 2: ValueError

```
numbers = [1, 2, 3]
numbers.remove(5)
```

Click for Answer

Error: `ValueError: list.remove(x): x not in list`

Why: 5 doesn't exist in the list

Fix:

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

# Check first
if 5 in numbers:
    numbers.remove(5)
else:
    print("5 not in list")  #

# Or use try/except
try:
    numbers.remove(5)
except ValueError:
    print("Item not found")  #
```

5.8.3 Problem 3: Type Error

```
numbers = [1, 2, 3]
index = "1"
print(numbers[index])
```

Click for Answer

Error: `TypeError: list indices must be integers or slices, not str`

Why: Index is string, needs to be integer

Fix:

```
numbers = [1, 2, 3]
index = "1"
print(numbers[int(index)])  #  Convert to int
# Prints: 2
```

5.8.4 Problem 4: Unintended Modification

```
def add_item(lst):
    lst.append(99)
    return lst

original = [1, 2, 3]
new_list = add_item(original)
print(original)  # What does this print?
```

Click for Answer

Issue: Prints `[1, 2, 3, 99]` - original was modified!

Why: Function modified the original list, not a copy

Fix:

```
def add_item(lst):
    new = lst.copy()  #  Create copy
    new.append(99)
    return new

original = [1, 2, 3]
new_list = add_item(original)
print(original)  # [1, 2, 3] - unchanged
print(new_list)  # [1, 2, 3, 99]
```

5.8.5 Problem 5: Wrong List Comprehension

```
numbers = [1, 2, 3, 4, 5]
evens = [x for x in numbers if x % 2 = 0]
```

Click for Answer

Error: `SyntaxError: invalid syntax`

Why: Using = (assignment) instead of == (comparison)

Fix:

```
numbers = [1, 2, 3, 4, 5]
evens = [x for x in numbers if x % 2 == 0]  #   Use ==
print(evens)  # [2, 4]
```

5.9 4.8 Key Takeaways

5.9.1 What You Learned

1. **Check length before indexing** - Avoid IndexError
2. **Check membership before removing** - Use `in` operator
3. **Use integer indices** - Convert strings/floats to int
4. **Be aware of mutability** - Lists can be modified
5. **Create copies when needed** - Use .copy() or [:]
6. **Use list comprehensions** - More Pythonic and readable
7. **Iterate directly when possible** - Avoid indexing errors

5.9.2 Common Patterns

```
# Pattern 1: Safe access
if index < len(numbers):
    value = numbers[index]

# Pattern 2: Check before remove
if item in numbers:
    numbers.remove(item)

# Pattern 3: Create copy
new_list = original.copy()

# Pattern 4: Iterate without indices
for item in numbers:
    print(item)

# Pattern 5: List comprehension
result = [x * 2 for x in numbers if x > 0]

# Pattern 6: Empty check
if numbers:  # Has items
    first = numbers[0]
```

5.9.3 Error Summary Table

Error Type	Common Cause	Prevention
`IndexError`	Index out of range	Check len() first, use iteration
`ValueError`	Item not in list	Check with `in` before remove/index
`TypeError`	Non-integer index	Convert to int
Unintended modification	Shared references	Use .copy()

5.10 4.9 Moving Forward

You now understand lists and list methods. You can: - Access list elements safely - Add and remove items correctly - Use list methods properly - Handle mutability - Write list comprehensions

In **Chapter 5**, we'll explore **Dictionaries and Sets** - key-value pairs and unique collections!

Chapter 6

Chapter 5: Dictionaries and Sets - Key-Value and Unique Collection Errors

6.1 Introduction

You've mastered lists. Now let's explore **dictionaries** and **sets** - two powerful collection types that work differently from lists.

Dictionaries store key-value pairs (like a real dictionary: word → definition). **Sets** store unique items with no duplicates.

Common errors: - **KeyError**: Accessing non-existent dictionary keys - **TypeError**: Unhashable types as keys - **AttributeError**: Wrong methods - Set operation errors

Let's master these collections by understanding their errors!

6.2 5.1 Dictionary Basics

6.2.1 Creating and Using Dictionaries

```
# Creating dictionaries
person = {
    "name": "Alice",
    "age": 25,
    "city": "New York"
```

```
}

# Empty dictionary
empty = {}
# Or
empty = dict()

# Accessing values
name = person["name"]  # "Alice"
age = person["age"]    # 25

# Adding/modifying values
person["email"] = "alice@email.com"  # Add new key
person["age"] = 26                    # Modify existing

# Dictionary length
length = len(person)  # Number of key-value pairs

# Check if key exists
if "name" in person:
    print("Name exists")
```

6.2.2 Error Type 1: `KeyError: 'key_name'`

Error Message:

```
>>> person = {"name": "Alice", "age": 25}
>>> print(person["email"])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'email'
```

What Happened: You tried to access a dictionary key that doesn't exist.

Why It Happens: - Key doesn't exist in dictionary - Typo in key name - Case sensitivity - Wrong data type for key

Code Example - WRONG:

```
person = {"name": "Alice", "age": 25}

# Non-existent key
email = person["email"]  # ERROR! Key doesn't exist

# Typo in key
name = person["nane"]  # ERROR! Typo
```

```python
# Case sensitivity
city = person["City"]  # ERROR! Key is "city" not "City"

# Wrong type
data = {1: "one", 2: "two"}
value = data["1"]  # ERROR! Key is int 1, not string "1"

# After deleting
person = {"name": "Alice", "age": 25}
del person["age"]
age = person["age"]  # ERROR! Already deleted

# Nested dictionary access
data = {"user": {"name": "Alice"}}
email = data["user"]["email"]  # ERROR! "email" doesn't exist
```

Code Example - CORRECT:

```python
person = {"name": "Alice", "age": 25}

# Check key exists before accessing
if "email" in person:
    email = person["email"]
else:
    email = None
    print("Email not found")

# Use .get() method (RECOMMENDED)
email = person.get("email")  #   Returns None if not found
email = person.get("email", "no-email@example.com")  #   With default

# Use try/except
try:
    email = person["email"]
except KeyError:
    email = None
    print("Key not found")

# Check spelling carefully
name = person["name"]  #   Correct spelling

# Match case exactly
data = {"city": "NYC", "City": "New York"}
city = data["city"]  #   "NYC"
City = data["City"]  #   "New York" (different key!)
```

```
# Match key type
data = {1: "one", 2: "two"}
value = data[1]  #   Use int, not string

# Safe nested access
data = {"user": {"name": "Alice"}}
email = data.get("user", {}).get("email", "N/A")  #   Safe chain

# Or check each level
if "user" in data and "email" in data["user"]:
    email = data["user"]["email"]
else:
    email = "N/A"

# Use setdefault to get or create
person = {"name": "Alice"}
email = person.setdefault("email", "default@example.com")
# If "email" exists, returns its value
# If not, sets it to default and returns default
```

Dictionary Access Patterns:

```
person = {"name": "Alice", "age": 25}

# Direct access (raises KeyError if missing)
name = person["name"]  # Use when key MUST exist

# .get() method (returns None or default)
email = person.get("email")  # Use when key might not exist
email = person.get("email", "N/A")  # With default value

# Check first
if "email" in person:
    email = person["email"]

# Get with default using setdefault
email = person.setdefault("email", "new@example.com")
# Sets and returns default if key doesn't exist
```

6.3 5.2 Dictionary Methods

6.3.1 Common Dictionary Methods

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

# Get keys, values, items
keys = person.keys()      # dict_keys(['name', 'age', 'city'])
values = person.values()  # dict_values(['Alice', 25, 'NYC'])
items = person.items()    # dict_items([('name', 'Alice'), ...])

# Convert to lists
key_list = list(person.keys())  # ['name', 'age', 'city']

# Get with default
email = person.get("email", "N/A")  # "N/A"

# Set default if missing
person.setdefault("country", "USA")  # Adds if not exists

# Update dictionary
person.update({"email": "alice@example.com", "age": 26})

# Remove items
age = person.pop("age")         # Remove and return value
person.pop("email", None)       # Safe removal with default
del person["city"]              # Remove (raises KeyError if missing)
person.clear()                  # Remove all items

# Copy dictionary
copy = person.copy()  # Shallow copy
```

6.3.2 Error Type 2: `TypeError: unhashable type: 'list'`

Error Message:

```
>>> data = {[1, 2]: "value"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

What Happened: You tried to use a mutable object (like list or dict) as a dictionary key.

Why It Happens: - Using list as key - Using dict as key - Using set as key -

Mutable objects can't be keys

Code Example - WRONG:

```
# List as key
data = {[1, 2]: "value"}  # ERROR! Lists are mutable

# Dictionary as key
data = {{"a": 1}: "value"}  # ERROR! Dicts are mutable

# Set as key
data = {{1, 2}: "value"}  # ERROR! Sets are mutable

# Variable holding mutable
key = [1, 2, 3]
data = {key: "value"}  # ERROR! key is a list
```

Code Example - CORRECT:

```
# Use tuple instead of list (tuples are immutable)
data = {(1, 2): "value"}  #   Tuples work as keys
print(data[(1, 2)])  # "value"

# Use frozenset instead of set
data = {frozenset([1, 2]): "value"}  #   Frozen sets work

# Convert list to tuple
key_list = [1, 2, 3]
data = {tuple(key_list): "value"}  #   Convert to tuple

# Immutable types that work as keys:
data = {
    42: "int",              #   int
    3.14: "float",          #   float
    "key": "string",        #   string
    (1, 2): "tuple",        #   tuple
    True: "bool",           #   bool
    frozenset([1]): "fs"    #   frozenset
}

# Use string representation if you must use mutable
key_list = [1, 2, 3]
key = str(key_list)  # "[1, 2, 3]"
data = {key: "value"}  #   String key

# Or use tuple of sorted items for dict
original = {"b": 2, "a": 1}
```

```
key = tuple(sorted(original.items()))
data = {key: "value"}  #

# Store complex keys as tuples
# Instead of: {[x, y]: value}
coordinates = {(10, 20): "point1", (30, 40): "point2"}  #
```

Hashable vs Unhashable:

```
# HASHABLE (can be dictionary keys):
# - int, float, string, tuple, bool, frozenset
# - Immutable objects

# UNHASHABLE (cannot be dictionary keys):
# - list, dict, set
# - Mutable objects

# Test if something is hashable
try:
    hash([1, 2, 3])
except TypeError:
    print("Unhashable")  # Lists are unhashable

hash((1, 2, 3))  #   Works - tuples are hashable
```

6.4 5.3 Dictionary Iteration

6.4.1 Iterating Over Dictionaries

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

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

# Explicit keys
for key in person.keys():
    print(key)

# Iterate over values
for value in person.values():
    print(value)  # Alice, 25, NYC
```

```
# Iterate over key-value pairs
for key, value in person.items():
    print(f"{key}: {value}")
# name: Alice
# age: 25
# city: NYC
```

6.4.2 Error Type 3: `RuntimeError: dictionary changed size during iteration`

Error Message:

```
>>> person = {"name": "Alice", "age": 25}
>>> for key in person:
...     if key == "age":
...         del person[key]
RuntimeError: dictionary changed size during iteration
```

What Happened: You tried to modify a dictionary while iterating over it.

Why It Happens: - Adding keys during iteration - Removing keys during iteration - Modifying dictionary size while looping

Code Example - WRONG:

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

# Deleting during iteration
for key in person:
    if key == "age":
        del person[key]  # ERROR! Can't modify during iteration

# Adding during iteration
for key in person:
    if key == "name":
        person["email"] = "alice@example.com"  # ERROR!

# Popping during iteration
for key in person:
    person.pop(key)  # ERROR!
```

Code Example - CORRECT:

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

# Create list of keys first
```

```python
for key in list(person.keys()):  #   Convert to list
    if key == "age":
        del person[key]

# Or collect keys to delete
keys_to_delete = []
for key in person:
    if key == "age":
        keys_to_delete.append(key)

for key in keys_to_delete:
    del person[key]  #   Delete after iteration

# Use dictionary comprehension to filter
person = {"name": "Alice", "age": 25, "city": "NYC"}
person = {k: v for k, v in person.items() if k != "age"}  #
# Result: {"name": "Alice", "city": "NYC"}

# Safe modification of values (not keys) is OK
for key in person:
    person[key] = str(person[key]).upper()  #   Modifying values OK
# Result: {"name": "ALICE", "city": "NYC"}

# Create new dictionary with modifications
new_person = {}
for key, value in person.items():
    if key != "age":
        new_person[key] = value  #

# Use copy for safe iteration
for key in person.copy():  #   Iterate over copy
    if key == "age":
        del person[key]
```

6.5 5.4 Set Basics

6.5.1 Creating and Using Sets

```python
# Creating sets
numbers = {1, 2, 3, 4, 5}
names = {"Alice", "Bob", "Charlie"}
```

```
# Empty set (must use set(), not {})
empty = set()  #   Empty set
# empty = {}  # This creates empty dict, not set!

# Set from list (removes duplicates)
numbers = set([1, 2, 2, 3, 3, 3])  # {1, 2, 3}

# Adding elements
numbers.add(6)  # {1, 2, 3, 6}

# Removing elements
numbers.remove(2)    # Raises KeyError if not found
numbers.discard(2)   # No error if not found
popped = numbers.pop()  # Remove and return arbitrary element

# Set length
length = len(numbers)

# Check membership
if 3 in numbers:
    print("Found 3")
```

6.5.2 Error Type 4: `KeyError` in Sets

Error Message:

```
>>> numbers = {1, 2, 3}
>>> numbers.remove(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 5
```

What Happened: You tried to remove an element that doesn't exist in the set.

Why It Happens: - Using remove() on non-existent item - Already removed item - Wrong value or type

Code Example - WRONG:

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

# Remove non-existent item
numbers.remove(5)  # ERROR! 5 not in set

# Type mismatch
```

```
numbers.remove("1")  # ERROR! "1" != 1

# Already removed
numbers.remove(2)  # OK first time
numbers.remove(2)  # ERROR! Already removed
```

Code Example - CORRECT:

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

# Check before removing
if 5 in numbers:
    numbers.remove(5)
else:
    print("5 not in set")

# Use discard() instead (never raises error)
numbers.discard(5)  #   No error if not found
numbers.discard(2)  #   Removes 2
numbers.discard(2)  #   No error, already gone

# Use try/except
try:
    numbers.remove(5)
except KeyError:
    print("Item not found")

# Match type
numbers = {1, 2, 3}
numbers.discard(1)  #   Use int, not string

# Remove all matching items
to_remove = {2, 5, 7}
numbers = numbers - to_remove  #   Creates new set
# Only removes items that exist
```

Set Methods Comparison:

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

# remove() - raises KeyError if not found
numbers.remove(2)  #   OK
numbers.remove(5)  #   KeyError

# discard() - never raises error
numbers.discard(2)  #   OK
```

```
numbers.discard(5)  #   OK (no error)

# pop() - removes and returns arbitrary element
value = numbers.pop()  #   Removes one element
# On empty set:
empty = set()
# empty.pop()  # KeyError: 'pop from an empty set'

# clear() - removes all elements
numbers.clear()  # {}, now empty
```

6.6 5.5 Set Operations

6.6.1 Mathematical Set Operations

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

# Union (all elements from both sets)
union = set1 | set2  # {1, 2, 3, 4, 5, 6}
union = set1.union(set2)  # Same

# Intersection (elements in both sets)
intersection = set1 & set2  # {3, 4}
intersection = set1.intersection(set2)  # Same

# Difference (elements in set1 but not set2)
difference = set1 - set2  # {1, 2}
difference = set1.difference(set2)  # Same

# Symmetric difference (elements in either but not both)
sym_diff = set1 ^ set2  # {1, 2, 5, 6}
sym_diff = set1.symmetric_difference(set2)  # Same

# Subset and superset
is_subset = {1, 2}.issubset({1, 2, 3})  # True
is_superset = {1, 2, 3}.issuperset({1, 2})  # True

# Disjoint (no common elements)
are_disjoint = {1, 2}.isdisjoint({3, 4})  # True
```

6.6.2 Error Type 5: `TypeError: unhashable type` in Sets

Error Message:

```
>>> numbers = {1, 2, [3, 4]}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

What Happened: You tried to add a mutable object to a set.

Why It Happens: - Adding list to set - Adding dict to set - Adding set to set - Set elements must be immutable

Code Example - WRONG:

```
# List in set
numbers = {1, 2, [3, 4]}  # ERROR! Lists are mutable

# Dict in set
data = {1, 2, {"a": 1}}  # ERROR! Dicts are mutable

# Set in set
nested = {1, 2, {3, 4}}  # ERROR! Sets are mutable

# Adding mutable element
numbers = {1, 2, 3}
numbers.add([4, 5])  # ERROR!
```

Code Example - CORRECT:

```
# Use tuple instead of list
numbers = {1, 2, (3, 4)}  #   Tuples are immutable
print(numbers)  # {1, 2, (3, 4)}

# Use frozenset instead of set
nested = {1, 2, frozenset([3, 4])}  #   Frozen sets work
print(nested)  # {1, 2, frozenset({3, 4})}

# Convert before adding
numbers = {1, 2, 3}
to_add = [4, 5]
numbers.add(tuple(to_add))  #
# {1, 2, 3, (4, 5)}

# Only immutable types work
valid_set = {
    42,              #   int
    3.14,            #   float
```

```
    "hello",         #   string
    (1, 2),          #   tuple
    True,            #   bool
    frozenset([1])   #   frozenset
}

# For lists of items, store as tuples
coordinates = {(0, 0), (1, 1), (2, 2)}  #

# Or convert to strings if needed
items = {str([1, 2]), str([3, 4])}  #
# {'[1, 2]', '[3, 4]'}
```

6.7 5.6 Common Dictionary and Set Patterns

6.7.1 Useful Patterns

```
# Counting items
words = ["apple", "banana", "apple", "cherry", "banana", "apple"]
count = {}
for word in words:
    count[word] = count.get(word, 0) + 1
# {'apple': 3, 'banana': 2, 'cherry': 1}

# Or use setdefault
count = {}
for word in words:
    count.setdefault(word, 0)
    count[word] += 1

# Or use Counter (better)
from collections import Counter
count = Counter(words)

# Grouping items
people = [
    {"name": "Alice", "age": 25},
    {"name": "Bob", "age": 30},
    {"name": "Charlie", "age": 25}
]
by_age = {}
for person in people:
    age = person["age"]
```

```
    if age not in by_age:
        by_age[age] = []
    by_age[age].append(person["name"])
# {25: ['Alice', 'Charlie'], 30: ['Bob']}

# Remove duplicates while preserving order
items = [1, 2, 2, 3, 1, 4, 3]
unique = list(dict.fromkeys(items))  # [1, 2, 3, 4]

# Merge dictionaries (Python 3.9+)
dict1 = {"a": 1, "b": 2}
dict2 = {"c": 3, "d": 4}
merged = dict1 | dict2  # {'a': 1, 'b': 2, 'c': 3, 'd': 4}

# Or use update
merged = dict1.copy()
merged.update(dict2)

# Invert dictionary (swap keys and values)
original = {"a": 1, "b": 2, "c": 3}
inverted = {v: k for k, v in original.items()}
# {1: 'a', 2: 'b', 3: 'c'}

# Find common elements
list1 = [1, 2, 3, 4, 5]
list2 = [4, 5, 6, 7, 8]
common = set(list1) & set(list2)  # {4, 5}

# Find unique elements
unique_to_list1 = set(list1) - set(list2)  # {1, 2, 3}
```

6.8 5.7 Practice Problems - Fix These Errors!

6.8.1 Problem 1: KeyError

```
person = {"name": "Alice", "age": 25}
print(person["email"])
```

Click for Answer

Error: `KeyError: 'email'`

Why: “email” key doesn’t exist

Fix:

```
person = {"name": "Alice", "age": 25}

# Use .get() with default
print(person.get("email", "N/A"))  #  Prints: N/A

# Or check first
if "email" in person:
    print(person["email"])
else:
    print("Email not found")  #
```

6.8.2 Problem 2: Unhashable Type

```
data = {[1, 2]: "value"}
```

Click for Answer

Error: `TypeError: unhashable type: 'list'`

Why: Lists can't be dictionary keys (they're mutable)

Fix:

```
# Use tuple instead
data = {(1, 2): "value"}  #  Tuples work as keys
print(data[(1, 2)])  # "value"
```

6.8.3 Problem 3: Modifying During Iteration

```
person = {"name": "Alice", "age": 25, "city": "NYC"}
for key in person:
    if key == "age":
        del person[key]
```

Click for Answer

Error: `RuntimeError: dictionary changed size during iteration`

Why: Can't modify dictionary size while iterating

Fix:

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

```
# Iterate over list of keys
for key in list(person.keys()):  #  Convert to list
    if key == "age":
        del person[key]

# Or use dictionary comprehension
person = {k: v for k, v in person.items() if k != "age"}  #
```

6.8.4 Problem 4: Set Remove Error

```
numbers = {1, 2, 3}
numbers.remove(5)
```

Click for Answer

Error: `KeyError: 5`

Why: 5 doesn't exist in the set

Fix:

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

# Use discard instead (no error if not found)
numbers.discard(5)  #  No error

# Or check first
if 5 in numbers:
    numbers.remove(5)
else:
    print("5 not in set")  #
```

6.8.5 Problem 5: Empty Set Creation

```
empty = {}
empty.add(1)
```

Click for Answer

Error: `AttributeError: 'dict' object has no attribute 'add'`

Why: {} creates empty dict, not empty set

Fix:

```
# Use set() for empty set
empty = set()  #   Empty set
empty.add(1)   #   Works
print(empty)   # {1}

# {} creates empty dictionary
empty_dict = {}  #   Empty dict
empty_dict["key"] = "value"  #   Works for dict
```

6.9 5.8 Key Takeaways

6.9.1 What You Learned

1. **Use .get() for safe dictionary access** - Returns None instead of KeyError
2. **Only immutable types as dict keys** - Use tuples, not lists
3. **Don't modify dict during iteration** - Create list of keys first
4. **Use discard() for safe set removal** - No error if item not found
5. **Empty set needs set()** - {} creates empty dict
6. **Check membership with in** - Before accessing or removing
7. **Sets automatically remove duplicates** - Great for unique items

6.9.2 Common Patterns

```
# Pattern 1: Safe dictionary access
value = data.get("key", default_value)

# Pattern 2: Check before access
if "key" in data:
    value = data["key"]

# Pattern 3: Safe iteration modification
for key in list(data.keys()):
    if condition:
        del data[key]

# Pattern 4: Safe set removal
numbers.discard(item)  # No error

# Pattern 5: Remove duplicates
unique = list(set(items))

# Pattern 6: Count occurrences
```

```
count = {}
for item in items:
    count[item] = count.get(item, 0) + 1
```

6.9.3 Error Summary Table

Error Type	Common Cause	Prevention
`KeyError` (dict)	Non-existent key	Use .get() or check with in
`TypeError` (unhashable)	Mutable key/element	Use immutable types (tuple, frozenset)
`RuntimeError`	Modifying during iteration	Iterate over list(dict.keys())
`KeyError` (set)	Remove non-existent item	Use discard() instead of remove()

6.10 5.9 Moving Forward

You now understand dictionaries and sets. You can: - Access dictionary values safely - Use correct types for keys - Iterate and modify safely - Perform set operations - Handle unique collections

In **Chapter 6**, we'll explore **Tuples and Immutability** - understanding immutable sequences!

Chapter 7

Chapter 6: Tuples and Immutability - Understanding Immutable Sequences

7.1 Introduction

You've mastered lists and dictionaries. Now let's explore **tuples** - immutable sequences that look similar to lists but behave very differently.

Tuples are ordered collections like lists, but once created, they cannot be modified. This immutability makes them useful for data that shouldn't change and as dictionary keys.

Common errors: - **TypeError**: Trying to modify tuples - **AttributeError**: Using list methods on tuples - Unpacking errors - Index errors (similar to lists)

Let's master tuples by understanding their errors!

7.2 6.1 Tuple Basics

7.2.1 Creating and Using Tuples

```
# Creating tuples
coordinates = (10, 20)
person = ("Alice", 25, "NYC")
```

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

# Single element tuple (comma is required!)
single = (5,)    #  Tuple with one element
not_tuple = (5)  # Just the number 5, not a tuple!

# Empty tuple
empty = ()
empty = tuple()

# Tuple without parentheses (tuple packing)
point = 10, 20, 30  # (10, 20, 30)

# Accessing elements (like lists)
first = coordinates[0]   # 10
last = coordinates[-1]   # 20

# Tuple length
length = len(person)  # 3

# Check membership
if "Alice" in person:
    print("Found Alice")

# Slicing (returns new tuple)
subset = colors[1:]  # ('green', 'blue')
```

7.2.2 Error Type 1: `TypeError: 'tuple' object does not support item assignment`

Error Message:

```
>>> coordinates = (10, 20)
>>> coordinates[0] = 15
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

What Happened: You tried to modify a tuple. Tuples are immutable - they cannot be changed after creation.

Why It Happens: - Trying to change tuple elements - Treating tuple like a list - Not understanding immutability - Trying to use mutating methods

Code Example - WRONG:

```
coordinates = (10, 20)

# Can't modify elements
coordinates[0] = 15  # ERROR! Tuples are immutable

# Can't delete elements
del coordinates[0]  # ERROR! Can't delete from tuple

# Can't append
coordinates.append(30)  # ERROR! No append method

# Can't extend
coordinates.extend([30, 40])  # ERROR! No extend method

# Can't remove
coordinates.remove(10)  # ERROR! No remove method

# Can't use list methods
coordinates.insert(0, 5)  # ERROR! No insert method
coordinates.pop()  # ERROR! No pop method
coordinates.reverse()  # ERROR! No reverse method
coordinates.sort()  # ERROR! No sort method
```

Code Example - CORRECT:

```
coordinates = (10, 20)

# Create new tuple with changes
coordinates = (15, 20)  #   New tuple

# Concatenate tuples (creates new tuple)
coordinates = coordinates + (30,)  #   (15, 20, 30)

# Concatenate multiple
coordinates = coordinates + (40, 50)  #   (15, 20, 30, 40, 50)

# Repeat tuples
repeated = (1, 2) * 3  #   (1, 2, 1, 2, 1, 2)

# Convert to list, modify, convert back
coordinates = (10, 20, 30)
temp_list = list(coordinates)  # [10, 20, 30]
temp_list[0] = 15              # Modify list
coordinates = tuple(temp_list)  #   (15, 20, 30)

# Build new tuple with comprehension
```

```
numbers = (1, 2, 3, 4, 5)
doubled = tuple(x * 2 for x in numbers)  #   (2, 4, 6, 8, 10)

# Replace by slicing
coordinates = (10, 20, 30)
coordinates = (15,) + coordinates[1:]  #   (15, 20, 30)

# Filter tuple
numbers = (1, 2, 3, 4, 5)
evens = tuple(x for x in numbers if x % 2 == 0)  #   (2, 4)

# Delete entire tuple (can delete the variable)
coordinates = (10, 20)
del coordinates  #   Deletes the variable, not just contents
```

Why Tuples Are Immutable:

```
# Immutability benefits:
# 1. Can be used as dictionary keys
location_data = {
    (10, 20): "Point A",
    (30, 40): "Point B"
}  #   Tuples work as keys

# 2. Safer - can't be accidentally modified
def process_data(data):
    # If data is a tuple, we know it won't change
    return data[0] + data[1]

# 3. Slightly faster than lists
# 4. Can be used in sets
points = {(0, 0), (1, 1), (2, 2)}  #   Set of tuples

# Lists can't do these:
# {[0, 0], [1, 1]}  # ERROR! Lists can't be in sets
# {[10, 20]: "value"}  # ERROR! Lists can't be dict keys
```

7.3 6.2 Tuple Unpacking

7.3.1 Understanding Unpacking

```
# Basic unpacking
coordinates = (10, 20)
```

```python
x, y = coordinates  # x=10, y=20

# Multiple values
person = ("Alice", 25, "NYC")
name, age, city = person

# Swap values (elegant!)
a, b = 5, 10
a, b = b, a  #   a=10, b=5

# Function returning multiple values
def get_coordinates():
    return 10, 20  # Returns tuple (10, 20)

x, y = get_coordinates()

# Using * to capture remaining
numbers = (1, 2, 3, 4, 5)
first, *rest = numbers  # first=1, rest=[2,3,4,5]
*start, last = numbers  # start=[1,2,3,4], last=5
first, *middle, last = numbers  # first=1, middle=[2,3,4], last=5
```

7.3.2 Error Type 2: `ValueError: too many values to unpack or not enough values to unpack`

Error Message:

```
>>> coordinates = (10, 20, 30)
>>> x, y = coordinates
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack (expected 2)
```

What Happened: The number of variables doesn't match the number of items in the tuple.

Why It Happens: - Too few variables for tuple items - Too many variables for tuple items - Wrong assumptions about tuple size - Function returns different number of values

Code Example - WRONG:

```python
# Too many values in tuple
coordinates = (10, 20, 30)
x, y = coordinates  # ERROR! 3 values, 2 variables
```

```
# Too few values in tuple
coordinates = (10,)
x, y = coordinates  # ERROR! 1 value, 2 variables

# Wrong assumption about function return
def get_data():
    return 1, 2, 3

a, b = get_data()  # ERROR! Returns 3, expecting 2

# Empty tuple
data = ()
x, y = data  # ERROR! No values to unpack
```

Code Example - CORRECT:

```
# Match number of variables to tuple size
coordinates = (10, 20, 30)
x, y, z = coordinates  #   3 values, 3 variables

# Use underscore for unwanted values
x, _, z = coordinates  #   Ignore middle value (y)

# Use * to capture remaining
first, *rest = coordinates  #   first=10, rest=[20,30]

# Check length before unpacking
coordinates = (10, 20, 30)
if len(coordinates) == 2:
    x, y = coordinates
elif len(coordinates) == 3:
    x, y, z = coordinates

# Safe unpacking with try/except
try:
    x, y = coordinates
except ValueError:
    print("Wrong number of values")
    x, y = coordinates[0], coordinates[1]

# Unpack with defaults
def safe_unpack(tup, count, default=None):
    """Safely unpack tuple with defaults"""
    result = list(tup) + [default] * (count - len(tup))
    return tuple(result[:count])
```

```python
coordinates = (10, 20)
x, y, z = safe_unpack(coordinates, 3, default=0)  #  x=10, y=20, z=0

# Use indexing if unsure
coordinates = (10, 20, 30)
x = coordinates[0] if len(coordinates) > 0 else None
y = coordinates[1] if len(coordinates) > 1 else None

# Unpack only what you need
coordinates = (10, 20, 30, 40, 50)
x, y, *_ = coordinates  #  Get first two, ignore rest

# Function with flexible return
def get_data():
    return 1, 2, 3

# Capture all with *
a, *rest = get_data()  #  a=1, rest=[2,3]

# Or match exactly
a, b, c = get_data()  #
```

Unpacking Patterns:

```python
# Basic unpacking
x, y = (10, 20)

# Ignore values
x, _ = (10, 20)  # Ignore second value
_, y = (10, 20)  # Ignore first value

# Extended unpacking (Python 3+)
first, *middle, last = (1, 2, 3, 4, 5)
# first=1, middle=[2,3,4], last=5

# Nested unpacking
point = ((10, 20), (30, 40))
(x1, y1), (x2, y2) = point
# x1=10, y1=20, x2=30, y2=40

# In loops
pairs = [(1, 2), (3, 4), (5, 6)]
for x, y in pairs:
    print(f"x={x}, y={y}")

# Dictionary items
```

```
person = {"name": "Alice", "age": 25}
for key, value in person.items():
    print(f"{key}: {value}")
```

7.4 6.3 Tuple Methods

7.4.1 Available Tuple Methods

```
# Tuples have only 2 methods!
numbers = (1, 2, 3, 2, 4, 2, 5)

# count() - count occurrences
count = numbers.count(2)  # 3

# index() - find first occurrence
index = numbers.index(2)  # 1 (first occurrence)
index = numbers.index(4)  # 4

# That's it! No other methods
```

7.4.2 Error Type 3: `AttributeError: 'tuple' object has no attribute 'X'`

Error Message:

```
>>> numbers = (1, 2, 3)
>>> numbers.append(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

What Happened: You tried to use a list method on a tuple.

Why It Happens: - Confusing tuples with lists - Trying to use mutating methods - Typo in method name

Code Example - WRONG:

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

# List methods don't work on tuples
numbers.append(4)    # ERROR! No append
numbers.extend([4])  # ERROR! No extend
```

```
numbers.insert(0, 0) # ERROR! No insert
numbers.remove(2)    # ERROR! No remove
numbers.pop()        # ERROR! No pop
numbers.sort()       # ERROR! No sort
numbers.reverse()    # ERROR! No reverse
numbers.clear()      # ERROR! No clear
```

Code Example - CORRECT:

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

# Use the 2 available methods
count = numbers.count(2)  #   Returns 1
index = numbers.index(2)  #   Returns 1

# For other operations, convert to list
numbers_list = list(numbers)  # [1, 2, 3]
numbers_list.append(4)        #   Works on list
numbers = tuple(numbers_list) #   Convert back (1, 2, 3, 4)

# Or create new tuple
numbers = numbers + (4,)  #   (1, 2, 3, 4)

# Sort: use sorted() function (returns list)
numbers = (3, 1, 4, 1, 5)
sorted_list = sorted(numbers)    # [1, 1, 3, 4, 5]
sorted_tuple = tuple(sorted(numbers))  #   (1, 1, 3, 4, 5)

# Reverse: use reversed() or slicing
reversed_list = list(reversed(numbers))  # [5, 1, 4, 1, 3]
reversed_tuple = numbers[::-1]  #   (5, 1, 4, 1, 3)

# Filter
numbers = (1, 2, 3, 4, 5)
evens = tuple(x for x in numbers if x % 2 == 0)  #   (2, 4)

# Map
doubled = tuple(x * 2 for x in numbers)  #   (2, 4, 6, 8, 10)

# Check method exists
if hasattr(numbers, 'count'):
    result = numbers.count(2)  #
```

Tuple vs List Methods:

```
# List has many methods:
my_list = [1, 2, 3]
```

```
my_list.append(4)      #
my_list.extend([5, 6]) #
my_list.insert(0, 0)   #
my_list.remove(2)      #
my_list.pop()          #
my_list.sort()         #
my_list.reverse()      #
my_list.clear()        #

# Tuple has only 2:
my_tuple = (1, 2, 3)
my_tuple.count(2)   #
my_tuple.index(2)   #
# That's all!
```

7.5 6.4 When to Use Tuples vs Lists

7.5.1 Choosing the Right Collection

```
# Use TUPLES when:
# 1. Data shouldn't change
coordinates = (10, 20)  # Position shouldn't change
date = (2025, 10, 27)   # Date is fixed

# 2. Need to use as dictionary key
locations = {
    (0, 0): "Origin",
    (10, 20): "Point A"
}

# 3. Need to use in set
points = {(0, 0), (1, 1), (2, 2)}

# 4. Returning multiple values from function
def get_stats():
    return 10, 25, 5  # min, max, avg

# 5. Data has fixed structure
person = ("Alice", 25, "NYC")  # name, age, city

# Use LISTS when:
# 1. Data will change
scores = [85, 90, 92]
```

```
scores.append(88)  # Need to add items

# 2. Need to sort/modify
numbers = [3, 1, 4, 1, 5]
numbers.sort()  # Need to sort

# 3. Collection of same type items
names = ["Alice", "Bob", "Charlie"]
names.remove("Bob")  # May need to remove

# 4. Don't need immutability
temperatures = [72, 75, 68, 70]
temperatures[0] = 73  # May need to update
```

7.6 6.5 Named Tuples

7.6.1 Using Named Tuples for Clarity

```
from collections import namedtuple

# Define named tuple type
Point = namedtuple('Point', ['x', 'y'])

# Create instance
p = Point(10, 20)

# Access by name (more readable)
print(p.x)  # 10
print(p.y)  # 20

# Or by index (still works)
print(p[0])  # 10
print(p[1])  # 20

# Still immutable
# p.x = 15  # ERROR! Can't modify

# More examples
Person = namedtuple('Person', ['name', 'age', 'city'])
person = Person('Alice', 25, 'NYC')
print(person.name)  # Alice
print(person.age)   # 25
```

```
# Unpack like regular tuple
name, age, city = person

# Convert to dict
person_dict = person._asdict()
# {'name': 'Alice', 'age': 25, 'city': 'NYC'}
```

7.7 6.6 Mutable Objects in Tuples

7.7.1 Understanding Nested Mutability

```
# Tuple itself is immutable
# But it can contain mutable objects!

data = ([1, 2, 3], [4, 5, 6])

# Can't change tuple structure
# data[0] = [7, 8, 9]  # ERROR!

# But CAN modify the lists inside
data[0].append(4)  #  Works!
print(data)  # ([1, 2, 3, 4], [4, 5, 6])

# Another example
person = ("Alice", 25, ["Python", "Java"])

# Can't change tuple
# person[0] = "Bob"  # ERROR!

# But can modify the list inside
person[2].append("C++")  #
print(person)  # ("Alice", 25, ["Python", "Java", "C++"])

# This is important for dictionary keys
# Tuple with mutable objects can't be dict key
skills = ["Python", "Java"]
# data = {("Alice", skills): "value"}  # ERROR if you modify skills later

# Use immutable contents for dict keys
data = {("Alice", "Python", "Java"): "value"}  #
```

7.8 6.7 Practice Problems - Fix These Errors!

7.8.1 Problem 1: Tuple Modification

```
colors = ("red", "green", "blue")
colors[0] = "yellow"
```

Click for Answer

Error: `TypeError: 'tuple' object does not support item assignment`

Why: Tuples are immutable

Fix:

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

# Create new tuple
colors = ("yellow", "green", "blue")  #

# Or use concatenation
colors = ("yellow",) + colors[1:]  #
print(colors)  # ("yellow", "green", "blue")
```

7.8.2 Problem 2: Wrong Unpacking

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

Click for Answer

Error: `ValueError: too many values to unpack (expected 2)`

Why: 3 values but only 2 variables

Fix:

```
coordinates = (10, 20, 30)

# Match number of variables
x, y, z = coordinates  #

# Or ignore extra values
x, y, *_ = coordinates  #  x=10, y=20, ignore z

# Or just take what you need
x, y = coordinates[:2]  #  x=10, y=20
```

7.8.3 Problem 3: Using List Method

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

Click for Answer

Error: `AttributeError: 'tuple' object has no attribute 'append'`

Why: Tuples don't have append method

Fix:

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

# Create new tuple with concatenation
numbers = numbers + (4,)  #
print(numbers)  # (1, 2, 3, 4)

# Or convert to list, modify, convert back
temp = list(numbers)
temp.append(4)
numbers = tuple(temp)  #
```

7.8.4 Problem 4: Single Element Tuple

```
single = (5)
print(type(single))
print(len(single))
```

Click for Answer

Issue: This creates an int, not a tuple!

Why: Parentheses alone don't make tuple, need comma

Fix:

```
# Need comma for single element tuple
single = (5,)  #   Notice the comma
print(type(single))  # <class 'tuple'>
print(len(single))   # 1

# Or without parentheses
single = 5,  #   Also works
print(type(single))  # <class 'tuple'>
```

7.8.5 Problem 5: Empty Unpacking

```
data = ()
x, y = data
```

Click for Answer

Error: `ValueError: not enough values to unpack (expected 2, got 0)`

Why: Empty tuple has no values

Fix:

```
data = ()

# Check before unpacking
if len(data) >= 2:
    x, y = data
else:
    x, y = None, None  #  Default values

# Or use try/except
try:
    x, y = data
except ValueError:
    x, y = None, None  #
```

7.9 6.8 Key Takeaways

7.9.1 What You Learned

1. **Tuples are immutable** - Cannot be modified after creation
2. **Only 2 methods** - count() and index()
3. **Comma makes tuple** - (5,) not (5) for single element
4. **Match unpacking variables** - Same number as tuple items
5. **Use as dict keys** - Unlike lists, tuples can be keys
6. **Create new tuples** - Use + or slicing instead of modifying
7. **Named tuples** - More readable than regular tuples

7.9.2 Common Patterns

```
# Pattern 1: Create new instead of modify
old_tuple = (1, 2, 3)
new_tuple = old_tuple + (4,)

# Pattern 2: Convert, modify, convert back
temp = list(my_tuple)
temp.append(new_item)
my_tuple = tuple(temp)

# Pattern 3: Safe unpacking
first, *rest = my_tuple

# Pattern 4: Swap values
a, b = b, a

# Pattern 5: Multiple return values
def get_data():
    return value1, value2, value3
```

7.9.3 Error Summary Table

Error Type	Common Cause	Prevention
`TypeError` (assignment)	Trying to modify tuple	Create new tuple instead
`AttributeError`	Using list methods	Only use count() and index()
`ValueError` (unpacking)	Wrong number of variables	Match count or use *
Single element	Missing comma	Use (value,) not (value)

7.10 6.9 Moving Forward

You now understand tuples and immutability. You can: - Use tuples appropriately - Unpack values safely - Understand when to use tuples vs lists - Work with immutable data - Use tuples as dictionary keys

In **Chapter 7**, we'll explore **Conditional Statements** - if/elif/else and logical flow!

Chapter 8

Chapter 7: Conditional Statements - Logic and Flow Control Errors

8.1 Introduction

You've mastered data structures. Now let's explore **conditional statements** - the foundation of program logic. Conditionals let your code make decisions and execute different code based on conditions.

Common errors: - **IndentationError**: Wrong indentation in Python - **SyntaxError**: Wrong syntax in if/elif/else - **NameError**: Variables not defined - Logic errors (wrong conditions)

Conditionals control program flow. Let's master them!

8.2 7.1 If Statement Basics

8.2.1 Simple If Statements

```
# Basic if statement
age = 18
if age >= 18:
    print("Adult")

# If with else
age = 15
```

```
if age >= 18:
    print("Adult")
else:
    print("Minor")

# If with elif
score = 85
if score >= 90:
    print("A")
elif score >= 80:
    print("B")
elif score >= 70:
    print("C")
else:
    print("F")

# Multiple conditions
age = 25
has_license = True
if age >= 18 and has_license:
    print("Can drive")
```

8.2.2 Error Type 1: `IndentationError: expected an indented block`

Error Message:

```
>>> if True:
... print("Hello")
  File "<stdin>", line 2
    print("Hello")
    ^
IndentationError: expected an indented block
```

What Happened: Python requires indentation after if statements. The code block must be indented.

Why It Happens: - Missing indentation - Inconsistent indentation (tabs vs spaces) - Wrong indentation level - Copy-paste errors

Code Example - WRONG:

```
# No indentation
if True:
print("Hello")  # ERROR! Must be indented
```

```
# Inconsistent indentation
if True:
    print("Line 1")  # 4 spaces
  print("Line 2")    # ERROR! 2 spaces

# Tab and space mixing
if True:
    print("Tab")    # Tab character
    print("Spaces")  # ERROR! Spaces (looks same but different)

# Wrong else indentation
if True:
    print("If block")
  else:  # ERROR! else should align with if
    print("Else block")

# Empty if block
if True:
# ERROR! Need pass or code

# Multiple levels wrong
if True:
    if True:
    print("Wrong")  # ERROR! Should be indented more
```

Code Example - CORRECT:

```
# Proper indentation (4 spaces is standard)
if True:
    print("Hello")  #   Indented 4 spaces

# Consistent indentation
if True:
    print("Line 1")  # 4 spaces
    print("Line 2")  # 4 spaces

# Correct else alignment
if True:
    print("If block")
else:  #   Aligned with if
    print("Else block")

# Empty if block - use pass
if True:
    pass  #   Placeholder
```

```
# Multiple levels
if True:
    print("Level 1")
    if True:
        print("Level 2")  #   Indented further

# Elif alignment
if score >= 90:
    print("A")
elif score >= 80:  #   Aligned with if
    print("B")
else:  #   Aligned with if
    print("F")

# Configure editor:
# - Use spaces, not tabs
# - Set indent to 4 spaces
# - Enable "show whitespace" to see issues
```

Python Indentation Rules:

```
# Standard is 4 spaces per level
if condition:
    # Level 1 (4 spaces)
    if nested_condition:
        # Level 2 (8 spaces)
        print("Nested")
    print("Level 1")

# All lines in same block must have same indentation
if True:
    print("Line 1")  # 4 spaces
    print("Line 2")  # 4 spaces
    # print("Line 3")  # Comment - indentation doesn't matter

# Control structures need indentation
if True:
    print("If")
else:
    print("Else")

for i in range(3):
    print(i)

while True:
    break
```

```python
def function():
    return

class MyClass:
    pass
```

8.3 7.2 Comparison Operators

8.3.1 Using Comparisons Correctly

```python
# Comparison operators
x = 10

# Equal
x == 10  # True

# Not equal
x != 5   # True

# Greater/less than
x > 5    # True
x < 20   # True
x >= 10  # True
x <= 10  # True

# Chaining comparisons
5 < x < 15  # True (same as: 5 < x and x < 15)
```

8.3.2 Error Type 2: `SyntaxError: invalid syntax` (in conditions)

Error Message:

```
>>> if x = 10:
  File "<stdin>", line 1
    if x = 10:
         ^
SyntaxError: invalid syntax
```

What Happened: Using assignment (=) instead of comparison (==) in condition.

Why It Happens: - Confusing = and == - Missing colon after condition - Wrong operator - Incomplete condition

Code Example - WRONG:

```
x = 10

# Assignment instead of comparison
if x = 10:  # ERROR! Use ==, not =
    print("Ten")

# Missing colon
if x == 10  # ERROR! Missing :
    print("Ten")

# Wrong syntax
if x == 10 and:  # ERROR! Incomplete condition
    print("Ten")

# Chaining without variable
if 5 < x < and x < 15:  # ERROR! Incomplete
    print("Range")

# Using 'is' for value comparison
if x is 10:  # Wrong! Use == for values
    print("Ten")  # May not work as expected

# Missing condition entirely
if:  # ERROR! Need condition
    print("Hello")
```

Code Example - CORRECT:

```
x = 10

# Use == for comparison
if x == 10:  #   Comparison
    print("Ten")

# Include colon
if x == 10:  #   Colon required
    print("Ten")

# Complete conditions
if x == 10 and x > 0:  #   Complete
    print("Ten and positive")
```

```
# Proper chaining
if 5 < x < 15:  #
    print("In range")

# Or explicit
if x > 5 and x < 15:  #   Also works
    print("In range")

# Use == for value comparison
if x == 10:  #   Correct for values
    print("Ten")

# Use 'is' only for None, True, False
if x is None:  #   Correct for None
    print("None")

if x is True:  #   Correct for True/False
    print("True")

# Parentheses for clarity (optional but helpful)
if (x > 5) and (x < 15):  #   Very clear
    print("In range")

# Multiple conditions
if x > 0 and x < 100 and x % 2 == 0:  #
    print("Even number between 0 and 100")
```

Comparison Operators Reference:

```
# All comparison operators:
==  # Equal to
!=  # Not equal to
<   # Less than
>   # Greater than
<=  # Less than or equal to
>=  # Greater than or equal to

# Identity operators:
is      # Same object (use for None, True, False)
is not  # Different object

# Membership operators:
in      # Item in collection
not in  # Item not in collection

# Examples:
```

```
x == 5          # Value comparison
x is None       # Identity comparison
"a" in "apple"  # Membership test
x not in [1,2]  # Negative membership
```

8.4 7.3 Logical Operators

8.4.1 Combining Conditions

```
# AND - both must be True
if age >= 18 and has_license:
    print("Can drive")

# OR - at least one must be True
if is_weekend or is_holiday:
    print("Day off")

# NOT - reverses boolean
if not is_raining:
    print("Go outside")

# Complex combinations
if (age >= 18 and has_license) or is_instructor:
    print("Can drive")

# Short-circuit evaluation
if user is not None and user.is_active():
    # user.is_active() only called if user is not None
    print("Active user")
```

8.4.2 Error Type 3: Logic Errors (No Python Error)

What Happened: Code runs but produces wrong results due to incorrect logic.

Code Example - WRONG LOGIC:

```
# Wrong order of checks
score = 85
if score >= 70:
    print("C")  # Prints "C" even though should be "B"
elif score >= 80:
    print("B")  # Never reached!
```

```
elif score >= 90:
    print("A")  # Never reached!

# Wrong operator
age = 25
if age > 18:  # Should be >=
    print("Adult")  # Excludes 18-year-olds

# Missing parentheses
if x > 5 and y > 3 or z > 10:
    # Evaluated as: (x > 5 and y > 3) or z > 10
    # Might not be what you want!
    print("Condition met")

# Using 'is' instead of ==
x = 1000
y = 1000
if x is y:  # False (different objects)
    print("Same")  # Doesn't print (wrong!)

# Comparing strings case-sensitively
name = "Alice"
if name == "alice":  # False due to case
    print("Match")  # Doesn't print
```

Code Example - CORRECT:

```
# Correct order (most specific first)
score = 85
if score >= 90:
    print("A")
elif score >= 80:
    print("B")  #   Prints correctly
elif score >= 70:
    print("C")
else:
    print("F")

# Correct operator (>= includes 18)
age = 18
if age >= 18:  #   Includes 18
    print("Adult")

# Use parentheses for clarity
if (x > 5 and y > 3) or z > 10:  #   Clear intent
    print("Condition met")
```

```
# Use == for value comparison
x = 1000
y = 1000
if x == y:  #   True (same value)
    print("Same")

# Case-insensitive string comparison
name = "Alice"
if name.lower() == "alice":  #   True
    print("Match")

# Handle None safely
value = None
if value is None:  #   Correct
    print("No value")

# Check type and value
if isinstance(x, int) and x > 0:  #   Safe
    print("Positive integer")

# Multiple conditions with clear logic
age = 25
has_license = True
has_insurance = True
if age >= 18:
    if has_license and has_insurance:
        print("Can drive")  #   Clear logic
    else:
        print("Need license or insurance")
else:
    print("Too young")
```

8.5 7.4 Truthiness and Falsiness

8.5.1 Understanding Boolean Context

```
# Falsy values (evaluate to False)
if False:        pass  # False
if None:         pass  # None
if 0:            pass  # Zero
if 0.0:          pass  # Zero float
if "":           pass  # Empty string
```

```python
if []:          pass  # Empty list
if {}:          pass  # Empty dict
if set():       pass  # Empty set
if ():          pass  # Empty tuple

# Truthy values (everything else)
if True:        pass  #   True
if 1:           pass  #   Non-zero number
if "text":      pass  #   Non-empty string
if [1]:         pass  #   Non-empty list
if {"a": 1}:    pass  #   Non-empty dict

# Common patterns
# Check if list has items
items = [1, 2, 3]
if items:  #   True if not empty
    print("Has items")

# Check if string is not empty
text = "hello"
if text:  #   True if not empty
    print("Has text")

# Check if variable is not None
value = 10
if value is not None:  #   Explicit None check
    print("Has value")

# Be careful with zero
count = 0
if count:  # False! 0 is falsy
    print("Has count")  # Won't print

# Better:
if count is not None:  #   True even if count is 0
    print("Has count")
```

8.5.2 Error Type 4: Truthiness Pitfalls

Code Example - WRONG LOGIC:

```python
# Treating zero as "no value"
count = 0
if count:  # False - treats 0 as falsy
```

```python
    print(f"Count: {count}")  # Doesn't print (wrong!)

# Confusing empty string with None
text = ""
if text:  # False - empty string is falsy
    print(text)  # Doesn't print
else:
    text = "Default"  # Sets default even if string was intentionally empty

# Not checking type
value = []
if value:  # False - empty list is falsy
    print("Has value")  # Doesn't print even though list exists

# Using boolean literal comparison
is_valid = True
if is_valid == True:  # Works but verbose
    print("Valid")
```

Code Example - CORRECT:

```python
# Check for None explicitly when zero is valid
count = 0
if count is not None:  #   True even if count is 0
    print(f"Count: {count}")  # Prints "Count: 0"

# Or check the specific range you want
if count >= 0:  #   Explicitly check non-negative
    print(f"Count: {count}")

# Distinguish empty string from None
text = ""
if text is not None:  #   True even if empty string
    print(f"Text: '{text}'")  # Prints "Text: ''"

# Check type and content separately
value = []
if value is not None:  #   List exists
    if value:  #   List has items
        print("Has items")
    else:
        print("Empty list")  #

# Direct boolean check (more Pythonic)
is_valid = True
if is_valid:  #   Cleaner
```

```
    print("Valid")

# Check length explicitly
items = []
if len(items) > 0:  #   Explicit check
    print("Has items")

# Or direct
if items:  #   Also works for non-empty check
    print("Has items")
```

8.6 7.5 Ternary Operator

8.6.1 Conditional Expressions

```
# Traditional if/else
if x > 0:
    result = "positive"
else:
    result = "non-positive"

# Ternary operator (conditional expression)
result = "positive" if x > 0 else "non-positive"

# More examples
status = "adult" if age >= 18 else "minor"

max_value = a if a > b else b

message = "Even" if x % 2 == 0 else "Odd"

# Nested ternary (avoid if too complex)
grade = "A" if score >= 90 else "B" if score >= 80 else "C"
```

8.6.2 Error Type 5: Ternary Syntax Errors

Code Example - WRONG:

```
# Wrong order
result = if x > 0 "positive" else "negative"  # ERROR! Syntax wrong
```

```
# Missing parts
result = "positive" if x > 0  # ERROR! Missing else

# Wrong keyword
result = "positive" when x > 0 else "negative"  # ERROR! Use 'if'

# Too complex (hard to read)
result = "A" if score >= 90 else "B" if score >= 80 else "C" if score >= 70 else "D" i
# Technically works but hard to read!
```

Code Example - CORRECT:

```
# Correct ternary syntax
result = "positive" if x > 0 else "negative"  #

# With else (required)
result = "positive" if x > 0 else "non-positive"  #

# Correct keyword (if, not when)
result = "positive" if x > 0 else "negative"  #

# Complex logic - use regular if/else instead
if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
elif score >= 60:
    grade = "D"
else:
    grade = "F"
#   Much more readable!

# Good ternary uses (simple, readable)
status = "online" if is_connected else "offline"  #
sign = "+" if x >= 0 else "-"  #
color = "red" if temp > 30 else "blue"  #

# Ternary with function calls
result = process_data() if data else default_value()  #
```

8.7 7.6 Common Conditional Patterns

8.7.1 Useful Patterns

```
# Check multiple conditions
if x and y and z:
    print("All true")

# Check any condition
if x or y or z:
    print("At least one true")

# Range checking
if 0 <= score <= 100:
    print("Valid score")

# Membership testing
if item in collection:
    print("Found")

# Type checking
if isinstance(value, int):
    print("Integer")

# None checking
if value is None:
    print("No value")

# Empty checking
if not items:
    print("Empty")

# Combining checks
if value is not None and value > 0:
    print("Positive value")

# Guard clauses (early return)
def process(data):
    if not data:
        return  # Exit early
    if not valid(data):
        return  # Exit early
    # Main logic here
    return result
```

8.8 7.7 Practice Problems - Fix These Errors!

8.8.1 Problem 1: Indentation Error

```
if True:
print("Hello")
```

Click for Answer

Error: `IndentationError: expected an indented block`

Why: Code after if must be indented

Fix:

```
if True:
    print("Hello")  #  Indented 4 spaces
```

8.8.2 Problem 2: Assignment in Condition

```
x = 10
if x = 10:
    print("Ten")
```

Click for Answer

Error: `SyntaxError: invalid syntax`

Why: Using = (assignment) instead of == (comparison)

Fix:

```
x = 10
if x == 10:  #  Use == for comparison
    print("Ten")
```

8.8.3 Problem 3: Wrong Logic Order

```
score = 85
if score >= 70:
    grade = "C"
elif score >= 80:
    grade = "B"
elif score >= 90:
    grade = "A"
print(grade)
```

Click for Answer

Issue: Prints "C" instead of "B" (logic error)

Why: Checks are in wrong order (70 before 80)

Fix:

```
score = 85
# Check highest first
if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"  #   Now this is checked
elif score >= 70:
    grade = "C"
else:
    grade = "F"
print(grade)  # Prints "B"
```

8.8.4 Problem 4: Truthiness Error

```
count = 0
if count:
    print(f"Count is {count}")
else:
    print("No count")
```

Click for Answer

Issue: Prints "No count" even though count is 0 (a valid value)

Why: 0 is falsy in Python

Fix:

```
count = 0
if count is not None:  #   Check for None explicitly
    print(f"Count is {count}")  # Prints "Count is 0"
else:
    print("No count")

# Or check specific range
if count >= 0:  #   Check for non-negative
    print(f"Count is {count}")
```

8.8.5 Problem 5: Missing Colon

```
if x > 5
    print("Greater than 5")
```

Click for Answer

Error: `SyntaxError: invalid syntax`

Why: Missing colon after condition

Fix:

```
if x > 5:  #   Add colon
    print("Greater than 5")
```

8.9 7.8 Key Takeaways

8.9.1 What You Learned

1. **Indent code blocks** - 4 spaces after if/elif/else
2. **Use == for comparison** - Not = (assignment)
3. **Order matters** - Check most specific conditions first
4. **Use 'is' for None** - Not == for None checks
5. **Understand truthiness** - Empty collections are falsy
6. **Include colon** - After every condition
7. **Use parentheses** - For complex conditions

8.9.2 Common Patterns

```
# Pattern 1: Range check
if 0 <= value <= 100:
    pass

# Pattern 2: None check
if value is not None:
    pass

# Pattern 3: Empty check
if not items:
    pass

# Pattern 4: Safe chaining
if user and user.is_active():
    pass
```

```python
# Pattern 5: Multiple conditions
if condition1 and condition2:
    pass
```

8.9.3 Error Summary Table

Error Type	Common Cause	Prevention
`IndentationError`	Missing/wrong indentation	Use 4 spaces consistently
`SyntaxError`	Using = instead of ==	Use == for comparison
`SyntaxError`	Missing colon	Add : after condition
Logic errors	Wrong order/operators	Check most specific first

8.10 7.9 Moving Forward

You now understand conditional statements. You can: - Write if/elif/else correctly - Use proper indentation - Compare values accurately - Handle None and empty values - Write clear logic

In **Chapter 8**, we'll explore **Loops** - for and while loops for iteration!

Chapter 9

Chapter 8: Loops - Iteration Errors

9.1 Introduction

You've mastered conditionals. Now let's explore **loops** - repeating code multiple times. Loops are essential for processing collections, repeating tasks, and iterating over data.

Common errors: - **IndexError**: Invalid indices in loops - **StopIteration**: Iterator exhausted - **KeyError**: Dictionary iteration errors - Infinite loops - Off-by-one errors

Let's master loops!

9.2 8.1 For Loop Basics

9.2.1 Iterating Over Collections

```
# Loop over list
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print(num)

# Loop over string
for char in "hello":
    print(char)
```

```
# Loop over dictionary keys
person = {"name": "Alice", "age": 25}
for key in person:
    print(key)

# Loop over dictionary items
for key, value in person.items():
    print(f"{key}: {value}")

# Loop with range
for i in range(5):  # 0, 1, 2, 3, 4
    print(i)

# Range with start and stop
for i in range(1, 6):  # 1, 2, 3, 4, 5
    print(i)

# Range with step
for i in range(0, 10, 2):  # 0, 2, 4, 6, 8
    print(i)
```

9.2.2 Error Type 1: `IndexError` in Loops

Error Message:

```
>>> numbers = [1, 2, 3]
>>> for i in range(len(numbers) + 1):
...     print(numbers[i])
1
2
3
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: list index out of range
```

What Happened: Loop index goes beyond list length.

Why It Happens: - Using `range(len(list) + 1)` - Off-by-one errors - Modifying list during iteration - Wrong range bounds

Code Example - WRONG:

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

# Loop too far
for i in range(len(numbers) + 1):
```

```
    print(numbers[i])  # ERROR when i=3

# Starting at wrong index
for i in range(1, len(numbers) + 1):
    print(numbers[i])  # ERROR when i=3

# Modifying list during iteration
for i in range(len(numbers)):
    numbers.append(i)  # ERROR! Changes length during loop
    print(numbers[i])

# Wrong understanding of range
for i in range(5):
    print(numbers[i])  # ERROR if list has < 5 items
```

Code Example - CORRECT:

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

# Correct range (no +1)
for i in range(len(numbers)):  #   0, 1, 2
    print(numbers[i])

# Better: iterate directly (no indexing)
for num in numbers:  #   No index errors possible
    print(num)

# With enumerate for index and value
for i, num in enumerate(numbers):
    print(f"Index {i}: {num}")  #

# Correct start index
for i in range(len(numbers)):  #   Starts at 0
    print(f"Index {i}: {numbers[i]}")

# Don't modify during iteration
# Instead, iterate over copy
for num in numbers.copy():  #   Iterate over copy
    numbers.append(num * 2)

# Or collect changes, apply later
to_add = []
for num in numbers:
    to_add.append(num * 2)
numbers.extend(to_add)  #   Add after loop
```

```
# Check bounds
for i in range(min(5, len(numbers))):  #   Safe
    print(numbers[i])
```

Loop Best Practices:

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

# BEST: Direct iteration (no indexing)
for num in numbers:
    print(num)  #   Simplest, safest

# GOOD: enumerate when you need index
for i, num in enumerate(numbers):
    print(f"{i}: {num}")  #

# AVOID: range(len(...)) unless necessary
for i in range(len(numbers)):
    print(numbers[i])  # Works but verbose

# NEVER: range(len(...) + 1)
# for i in range(len(numbers) + 1):  #   Common error
```

9.3 8.2 While Loops

9.3.1 Condition-Based Iteration

```
# Basic while loop
count = 0
while count < 5:
    print(count)
    count += 1

# While with break
while True:
    response = input("Enter 'quit' to exit: ")
    if response == "quit":
        break

# While with continue
count = 0
while count < 10:
    count += 1
```

```
    if count % 2 == 0:
        continue  # Skip even numbers
    print(count)
```

9.3.2 Error Type 2: Infinite Loops

What Happened: Loop never ends because condition stays True.

Why It Happens: - Forgetting to update condition - Wrong condition logic - Never reaching break statement

Code Example - WRONG:

```
# Forgot to update variable
count = 0
while count < 5:
    print(count)  # Prints 0 forever!
    # ERROR! Forgot: count += 1

# Wrong condition
count = 0
while count != 5:
    count += 2  # 0, 2, 4, 6, 8... never equals 5!
    print(count)  # Infinite loop!

# Condition never becomes False
while True:
    print("Forever!")  # ERROR! No break

# Break never reached
count = 0
while count < 10:
    print(count)
    if count > 20:  # Never true!
        break
    # Forgot to increment count!
```

Code Example - CORRECT:

```
# Update variable in loop
count = 0
while count < 5:
    print(count)
    count += 1  #   Updates condition variable

# Use correct condition
```

```
count = 0
while count < 5:  #   Use <, not !=
    print(count)
    count += 2

# Include break for True loops
while True:
    response = input("Enter 'quit': ")
    if response == "quit":
        break  #   Can exit

# Ensure break is reachable
count = 0
while count < 100:  # Safety limit
    print(count)
    if count >= 10:
        break  #   Reachable
    count += 1

# Add safety counter
count = 0
max_iterations = 1000
while condition and count < max_iterations:
    # Loop body
    count += 1  #   Safety limit

# Use for loop when count is known
for i in range(5):  #   Better than while for fixed iterations
    print(i)
```

9.4 8.3 Break and Continue

9.4.1 Controlling Loop Flow

```
# break - exit loop immediately
for i in range(10):
    if i == 5:
        break  # Exit loop when i is 5
    print(i)  # Prints 0, 1, 2, 3, 4

# continue - skip to next iteration
for i in range(5):
    if i == 2:
```

```
        continue  # Skip when i is 2
    print(i)  # Prints 0, 1, 3, 4

# break in nested loops (only exits inner loop)
for i in range(3):
    for j in range(3):
        if j == 1:
            break  # Only exits inner loop
        print(f"{i},{j}")

# Using else with loops (runs if no break)
for i in range(5):
    if i == 10:
        break
else:
    print("Loop completed")  # Prints because no break
```

9.5 8.4 Iterating Over Dictionaries

9.5.1 Dictionary Iteration Patterns

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

# Iterate over keys (default)
for key in person:
    print(key)

# Explicit keys
for key in person.keys():
    print(key)

# Iterate over values
for value in person.values():
    print(value)

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

9.5.2 Error Type 3: RuntimeError: dictionary changed size during iteration

Error Message:

```
>>> person = {"name": "Alice", "age": 25}
>>> for key in person:
...     if key == "age":
...         del person[key]
RuntimeError: dictionary changed size during iteration
```

What Happened: Cannot modify dictionary size while iterating.

Code Example - WRONG:

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

# Deleting during iteration
for key in person:
    if key == "age":
        del person[key]  # ERROR!

# Adding during iteration
for key in person:
    if key == "name":
        person["email"] = "alice@example.com"  # ERROR!
```

Code Example - CORRECT:

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

# Iterate over list of keys
for key in list(person.keys()):  #  Convert to list
    if key == "age":
        del person[key]

# Collect keys to delete
to_delete = []
for key in person:
    if key == "age":
        to_delete.append(key)

for key in to_delete:
    del person[key]  #

# Dictionary comprehension
person = {k: v for k, v in person.items() if k != "age"}  #
```

```python
# Modifying values is OK
for key in person:
    person[key] = str(person[key]).upper()  #   Values OK
```

9.6 8.5 Enumerate and Zip

9.6.1 Advanced Iteration

```python
# enumerate - get index and value
names = ["Alice", "Bob", "Charlie"]
for i, name in enumerate(names):
    print(f"{i}: {name}")
# 0: Alice
# 1: Bob
# 2: Charlie

# enumerate with custom start
for i, name in enumerate(names, start=1):
    print(f"{i}: {name}")
# 1: Alice
# 2: Bob
# 3: Charlie

# zip - iterate over multiple lists
names = ["Alice", "Bob"]
ages = [25, 30]
for name, age in zip(names, ages):
    print(f"{name} is {age}")
# Alice is 25
# Bob is 30

# zip with different lengths (stops at shortest)
list1 = [1, 2, 3]
list2 = ['a', 'b']
for num, letter in zip(list1, list2):
    print(num, letter)
# 1 a
# 2 b
# (3 is ignored)
```

9.7 8.6 List Comprehensions vs Loops

9.7.1 Comparing Approaches

```
# Traditional loop
squares = []
for x in range(5):
    squares.append(x ** 2)
# [0, 1, 4, 9, 16]

# List comprehension (better)
squares = [x ** 2 for x in range(5)]  #

# With condition
evens = []
for x in range(10):
    if x % 2 == 0:
        evens.append(x)

# Better
evens = [x for x in range(10) if x % 2 == 0]  #

# When to use loops vs comprehensions:
# Use comprehension: Simple transformations, filters
# Use loop: Complex logic, multiple statements, side effects
```

9.8 8.7 Common Loop Patterns

9.8.1 Useful Patterns

```
# Sum all numbers
numbers = [1, 2, 3, 4, 5]
total = 0
for num in numbers:
    total += num
# Or: total = sum(numbers)

# Find maximum
max_val = numbers[0]
for num in numbers[1:]:
    if num > max_val:
        max_val = num
# Or: max_val = max(numbers)
```

```
# Count occurrences
items = [1, 2, 2, 3, 2, 4]
count = 0
for item in items:
    if item == 2:
        count += 1
# Or: count = items.count(2)

# Build string
words = ["Hello", "World"]
result = ""
for word in words:
    result += word + " "
# Better: result = " ".join(words)

# Filter items
numbers = [1, 2, 3, 4, 5, 6]
evens = []
for num in numbers:
    if num % 2 == 0:
        evens.append(num)
# Better: evens = [x for x in numbers if x % 2 == 0]
```

9.9 8.8 Practice Problems

9.9.1 Problem 1: Index Error

```
numbers = [1, 2, 3]
for i in range(len(numbers) + 1):
    print(numbers[i])
```

Click for Answer

Error: `IndexError: list index out of range`

Fix:

```
numbers = [1, 2, 3]
for i in range(len(numbers)):  #   Remove +1
    print(numbers[i])

# Or better:
for num in numbers:  #   No indexing
    print(num)
```

9.9.2 Problem 2: Infinite Loop

```
count = 0
while count < 5:
    print(count)
```

Click for Answer

Issue: Infinite loop - count never increments

Fix:

```
count = 0
while count < 5:
    print(count)
    count += 1  #  Update condition variable
```

9.9.3 Problem 3: Dictionary Modification

```
data = {"a": 1, "b": 2, "c": 3}
for key in data:
    if key == "b":
        del data[key]
```

Click for Answer

Error: `RuntimeError: dictionary changed size during iteration`

Fix:

```
data = {"a": 1, "b": 2, "c": 3}

# Convert keys to list
for key in list(data.keys()):  #
    if key == "b":
        del data[key]

# Or use dict comprehension
data = {k: v for k, v in data.items() if k != "b"}  #
```

9.10 8.9 Key Takeaways

9.10.1 What You Learned

1. **Iterate directly** - for item in list (avoid indexing)
2. **Use enumerate** - When you need both index and value
3. **Update while conditions** - Prevent infinite loops
4. **Don't modify during iteration** - Use list(dict.keys())
5. **Use break/continue** - Control loop flow
6. **List comprehensions** - For simple transformations
7. **range(len())** - Usually not needed

9.10.2 Common Patterns

```
# Pattern 1: Direct iteration
for item in collection:
    process(item)

# Pattern 2: With index
for i, item in enumerate(collection):
    print(f"{i}: {item}")

# Pattern 3: Multiple lists
for x, y in zip(list1, list2):
    print(x, y)

# Pattern 4: Dictionary items
for key, value in dict.items():
    print(f"{key}: {value}")

# Pattern 5: Break on condition
for item in items:
    if condition:
        break
```

9.10.3 Error Summary

Error	Cause	Prevention
`IndexError`	range(len()+1)	Use range(len()) or iterate directly
Infinite loop	No update	Update condition variable
`RuntimeError`	Modify dict	Use list(dict.keys())

9.11 8.10 Moving Forward

You now understand loops! In **Chapter 9**, we'll explore **Functions**!

Chapter 10

Chapter 9: Functions - Defining and Calling Functions

10.1 Introduction

You've mastered loops and conditionals. Now let's explore **functions** - reusable blocks of code that perform specific tasks. Functions are fundamental to organizing and structuring programs.

Common errors: - **TypeError**: Wrong number or type of arguments - **NameError**: Function not defined or wrong scope - **UnboundLocalError**: Variable scope issues - **RecursionError**: Infinite recursion - Return value errors

Let's master functions!

10.2 9.1 Function Basics

10.2.1 Defining and Calling Functions

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

# Call the function
greet()  # Prints: Hello!
```

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

greet_person("Alice")  # Prints: Hello, Alice!

# Function with return value
def add(a, b):
    return a + b

result = add(5, 3)  # result = 8

# Function with multiple parameters
def calculate(x, y, operation):
    if operation == "add":
        return x + y
    elif operation == "multiply":
        return x * y

result = calculate(5, 3, "add")  # 8
```

10.2.2 Error Type 1: `TypeError: function() takes X positional arguments but Y were given`

Error Message:

```
>>> def greet(name):
...     print(f"Hello, {name}!")
>>> greet()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: greet() missing 1 required positional argument: 'name'
```

What Happened: Called function with wrong number of arguments.

Why It Happens: - Missing required arguments - Too many arguments - Forgetting self in methods - Wrong parameter count

Code Example - WRONG:

```
# Missing argument
def greet(name):
    print(f"Hello, {name}!")

greet()  # ERROR! Missing 'name'
```

```python
# Too many arguments
def add(a, b):
    return a + b

result = add(1, 2, 3)  # ERROR! Too many arguments

# Wrong number of arguments
def calculate(x, y, z):
    return x + y + z

result = calculate(1, 2)  # ERROR! Missing z

# Mixing positional and keyword wrong
def process(a, b, c):
    return a + b + c

result = process(1, c=3)  # ERROR! Missing b
```

Code Example - CORRECT:

```python
# Provide all required arguments
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")  #   Correct

# Match parameter count
def add(a, b):
    return a + b

result = add(1, 2)  #   Correct

# Use default parameters for optional arguments
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Alice")  #   Uses default: "Hello, Alice!"
greet("Bob", "Hi")  #   Custom: "Hi, Bob!"

# Use *args for variable arguments
def add_all(*numbers):
    return sum(numbers)

result = add_all(1, 2)  #   Works
result = add_all(1, 2, 3, 4, 5)  #   Also works
```

```
# Use **kwargs for keyword arguments
def print_info(**info):
    for key, value in info.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=25)  #

# Mix positional, default, *args, **kwargs
def complex_function(required, optional="default", *args, **kwargs):
    print(f"Required: {required}")
    print(f"Optional: {optional}")
    print(f"Args: {args}")
    print(f"Kwargs: {kwargs}")

complex_function("test")  #
complex_function("test", "custom", 1, 2, key="value")  #
```

10.3 9.2 Return Values

10.3.1 Understanding Returns

```
# Function with return
def add(a, b):
    return a + b

result = add(3, 5)  # result = 8

# Function without return (returns None)
def greet(name):
    print(f"Hello, {name}!")
    # No return statement

result = greet("Alice")  # result = None

# Multiple return values (returns tuple)
def get_stats(numbers):
    return min(numbers), max(numbers), sum(numbers)

min_val, max_val, total = get_stats([1, 2, 3, 4, 5])

# Early return
def check_age(age):
    if age < 0:
```

```
        return "Invalid"
    if age < 18:
        return "Minor"
    return "Adult"
```

10.3.2 Error Type 2: `TypeError: 'NoneType' object is not...`

Error Message:

```
>>> def add(a, b):
...     result = a + b
...     # Forgot return!
>>> total = add(3, 5)
>>> print(total + 10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

What Happened: Function returns None (no return statement), trying to use it.

Why It Happens: - Forgetting return statement - Return in wrong place - Conditional return missing else

Code Example - WRONG:

```
# Forgot return
def add(a, b):
    result = a + b
    # ERROR! No return

total = add(3, 5)  # None
print(total + 10)  # ERROR! None + 10

# Return in wrong place
def calculate(x):
    if x > 0:
        return x * 2
    # ERROR! No return for x <= 0

result = calculate(-5)  # None

# Indentation error
def multiply(a, b):
    result = a * b
```

```
return result  # ERROR! Wrong indentation

# After return is unreachable
def process():
    return "done"
    print("After")  # Never executes
```

Code Example - CORRECT:

```
# Include return statement
def add(a, b):
    result = a + b
    return result  #

total = add(3, 5)  # 8
print(total + 10)  #   18

# Return in all branches
def calculate(x):
    if x > 0:
        return x * 2
    else:
        return 0  #   Return for all cases

# Or single return at end
def calculate(x):
    if x > 0:
        result = x * 2
    else:
        result = 0
    return result  #

# Correct indentation
def multiply(a, b):
    result = a * b
    return result  #   Indented

# Check for None before using
def get_value():
    return None

value = get_value()
if value is not None:  #   Check first
    print(value + 10)
else:
    print("No value returned")
```

```
# Return default value
def safe_divide(a, b):
    if b == 0:
        return None
    return a / b

result = safe_divide(10, 0)  # None
if result is not None:  #   Safe
    print(result)
```

10.4 9.3 Variable Scope

10.4.1 Understanding Scope

```
# Global variable
global_var = "I'm global"

def function():
    # Local variable
    local_var = "I'm local"
    print(global_var)  #   Can read global
    print(local_var)   #   Can read local

function()
# print(local_var)  # ERROR! local_var not accessible

# Modifying global (need 'global' keyword)
counter = 0

def increment():
    global counter  # Declare as global
    counter += 1

increment()
print(counter)  # 1
```

10.4.2 Error Type 3: `UnboundLocalError: local variable 'X' referenced before assignment`

Error Message:

```
>>> count = 0
>>> def increment():
...     count = count + 1
>>> increment()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in increment
UnboundLocalError: local variable 'count' referenced before assignment
```

What Happened: Trying to modify global variable without declaring it as global.

Why It Happens: - Modifying global without 'global' keyword - Variable shadowing - Accessing before assignment in same scope

Code Example - WRONG:

```
# Modifying global without 'global'
count = 0

def increment():
    count = count + 1  # ERROR! UnboundLocalError
    return count

# Reading then modifying
value = 10

def update():
    print(value)  # Would work...
    value = 20    # ERROR! But this makes 'value' local

# Nested scope issue
def outer():
    x = 10
    def inner():
        x = x + 1  # ERROR! UnboundLocalError
        return x
    return inner()
```

Code Example - CORRECT:

```
# Use 'global' keyword
count = 0

def increment():
    global count  #   Declare as global
    count = count + 1
    return count
```

```python
increment()  # count is now 1

# Or pass as parameter (better)
def increment(count):
    return count + 1

count = 0
count = increment(count)  #   Better approach

# Return new value instead
value = 10

def update(val):
    return val + 10

value = update(value)  #   Functional approach

# Use nonlocal for nested functions
def outer():
    x = 10
    def inner():
        nonlocal x  #   For nested scope
        x = x + 1
        return x
    return inner()

# Avoid global when possible
# Instead, use class or pass parameters
class Counter:
    def __init__(self):
        self.count = 0

    def increment(self):
        self.count += 1  #   No global needed

counter = Counter()
counter.increment()
```

Scope Best Practices:

```python
# BEST: Avoid global variables
# Use parameters and return values
def add(x, y):
    return x + y
```

```
# GOOD: Use class for state
class Calculator:
    def __init__(self):
        self.total = 0

    def add(self, value):
        self.total += value

# AVOID: Global variables
# global_count = 0
# def increment():
#     global global_count
#     global_count += 1
```

10.5 9.4 Default Arguments

10.5.1 Using Default Parameters

```
# Simple default
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Alice")  # Uses default: "Hello, Alice!"
greet("Bob", "Hi")  # Custom: "Hi, Bob!"

# Multiple defaults
def create_user(name, age=None, city="Unknown"):
    print(f"{name}, {age}, {city}")

create_user("Alice")  # Alice, None, Unknown
create_user("Bob", 25)  # Bob, 25, Unknown
create_user("Charlie", 30, "NYC")  # Charlie, 30, NYC
```

10.5.2 Error Type 4: Mutable Default Arguments

What Happened: Using mutable objects (list, dict) as default arguments causes unexpected behavior.

Code Example - WRONG:

```
# DANGEROUS: Mutable default
def add_item(item, items=[]):
```

```
    items.append(item)
    return items

list1 = add_item("apple")  # ['apple']
list2 = add_item("banana")  # ['apple', 'banana'] - Unexpected!
list3 = add_item("cherry")  # ['apple', 'banana', 'cherry'] - Wrong!
# All share the same list!

# Same with dictionaries
def add_key(key, value, data={}):
    data[key] = value
    return data

dict1 = add_key("a", 1)  # {'a': 1}
dict2 = add_key("b", 2)  # {'a': 1, 'b': 2} - Unexpected!
```

Code Example - CORRECT:

```
# Use None as default, create new inside
def add_item(item, items=None):
    if items is None:
        items = []  #   Create new list each time
    items.append(item)
    return items

list1 = add_item("apple")  # ['apple']
list2 = add_item("banana")  # ['banana']
list3 = add_item("cherry")  # ['cherry']

# Same pattern for dictionaries
def add_key(key, value, data=None):
    if data is None:
        data = {}  #   Create new dict each time
    data[key] = value
    return data

dict1 = add_key("a", 1)  # {'a': 1}
dict2 = add_key("b", 2)  # {'b': 2}

# Or explicitly pass new object
def add_item(item, items=None):
    items = items if items is not None else []
    items.append(item)
    return items

# Immutable defaults are safe
```

```
def process(value, multiplier=2):  #   int is immutable
    return value * multiplier

def greet(name, prefix="Hello"):  #   string is immutable
    return f"{prefix}, {name}!"
```

10.6 9.5 *args and **kwargs

10.6.1 Variable Arguments

```
# *args - variable positional arguments
def add_all(*numbers):
    return sum(numbers)

add_all(1, 2)  # 3
add_all(1, 2, 3, 4, 5)  # 15

# **kwargs - variable keyword arguments
def print_info(**info):
    for key, value in info.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=25, city="NYC")

# Combining all parameter types
def complex_func(required, *args, optional="default", **kwargs):
    print(f"Required: {required}")
    print(f"Args: {args}")
    print(f"Optional: {optional}")
    print(f"Kwargs: {kwargs}")

complex_func("test", 1, 2, 3, optional="custom", key="value")
# Required: test
# Args: (1, 2, 3)
# Optional: custom
# Kwargs: {'key': 'value'}
```

10.7 9.6 Lambda Functions

10.7.1 Anonymous Functions

```
# Regular function
def square(x):
    return x ** 2

# Lambda equivalent
square = lambda x: x ** 2

# Lambda with multiple parameters
add = lambda x, y: x + y

# Lambda in sorting
pairs = [(1, 'one'), (3, 'three'), (2, 'two')]
pairs.sort(key=lambda pair: pair[0])
# [(1, 'one'), (2, 'two'), (3, 'three')]

# Lambda in map
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
# [1, 4, 9, 16, 25]

# Lambda in filter
evens = list(filter(lambda x: x % 2 == 0, numbers))
# [2, 4]
```

10.8 9.7 Recursion

10.8.1 Recursive Functions

```
# Basic recursion
def countdown(n):
    if n <= 0:  # Base case
        print("Done!")
        return
    print(n)
    countdown(n - 1)  # Recursive call

countdown(5)  # 5, 4, 3, 2, 1, Done!

# Factorial
```

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

factorial(5)  # 120

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

fibonacci(6)  # 8
```

10.8.2 Error Type 5: `RecursionError: maximum recursion depth exceeded`

Error Message:

```
>>> def infinite():
...     return infinite()
>>> infinite()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in infinite
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

What Happened: Recursive function never reaches base case.

Why It Happens: - Missing base case - Base case never reached - Wrong recursive logic

Code Example - WRONG:

```python
# Missing base case
def countdown(n):
    print(n)
    countdown(n - 1)  # ERROR! Never stops

# Base case never reached
def countdown(n):
    if n == 0:  # Only checks equality
        return
    countdown(n - 2)  # ERROR! Skips 0 if n is odd
```

```
# Wrong direction
def countdown(n):
    if n == 0:
        return
    countdown(n + 1)  # ERROR! Goes up, not down
```

Code Example - CORRECT:

```
# Include base case
def countdown(n):
    if n <= 0:  #   Base case
        return
    print(n)
    countdown(n - 1)

# Ensure base case is reachable
def countdown(n):
    if n <= 0:  #   Uses <= not ==
        return
    print(n)
    countdown(n - 1)

# Add safety limit
def countdown(n, depth=0, max_depth=1000):
    if n <= 0 or depth >= max_depth:  #   Safety
        return
    print(n)
    countdown(n - 1, depth + 1, max_depth)

# Use iteration when appropriate
def countdown(n):
    while n > 0:  #   Often better than recursion
        print(n)
        n -= 1
```

10.9 9.8 Practice Problems

10.9.1 Problem 1: Missing Argument

```
def greet(name, age):
    print(f"{name} is {age}")

greet("Alice")
```

Click for Answer

Error: `TypeError: greet() missing 1 required positional argument: 'age'`

Fix:

```
def greet(name, age):
    print(f"{name} is {age}")

greet("Alice", 25)  #  Provide both arguments

# Or use default
def greet(name, age=0):
    print(f"{name} is {age}")

greet("Alice")  #  Uses default age
```

10.9.2 Problem 2: Missing Return

```
def add(a, b):
    result = a + b

total = add(3, 5)
print(total * 2)
```

Click for Answer

Error: `TypeError: unsupported operand type(s) for *: 'NoneType' and 'int'`

Why: Function returns None (no return statement)

Fix:

```
def add(a, b):
    result = a + b
    return result  #  Add return

total = add(3, 5)
print(total * 2)  #  Works: 16
```

10.9.3 Problem 3: Global Variable

```
count = 0

def increment():
    count = count + 1

increment()
```

Click for Answer

Error: `UnboundLocalError: local variable 'count' referenced before assignment`

Fix:

```
count = 0

def increment():
    global count  #   Declare as global
    count = count + 1

increment()
print(count)  # 1

# Or better - use parameter
def increment(n):
    return n + 1

count = 0
count = increment(count)  #   Better
```

10.10 9.9 Key Takeaways

10.10.1 What You Learned

1. **Match argument count** - Provide all required arguments
2. **Always return values** - Don't forget return statement
3. **Use 'global' for global variables** - Or avoid globals
4. **Don't use mutable defaults** - Use None, create inside
5. **Include base case in recursion** - Prevent infinite recursion
6. **Pass parameters instead of globals** - Better design
7. **Check return values for None** - Before using

10.10.2 Common Patterns

```
# Pattern 1: Function with defaults
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

# Pattern 2: Safe mutable default
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items

# Pattern 3: Multiple returns
def get_stats(numbers):
    return min(numbers), max(numbers)

# Pattern 4: Variable arguments
def add_all(*numbers):
    return sum(numbers)
```

10.10.3 Error Summary

Error	Cause	Prevention
`TypeError` (args)	Wrong argument count	Match function signature
`TypeError` (NoneType)	Missing return	Add return statement
`UnboundLocalError`	Global without declaration	Use 'global' or parameters
`RecursionError`	No base case	Include termination condition

10.11 9.10 Moving Forward

You now understand functions! In **Chapter 10**, we'll explore **File I/O -** reading and writing files!

Chapter 11

Chapter 10: File I/O - Reading and Writing Files

11.1 Introduction

You've mastered functions. Now let's explore **File I/O** (Input/Output) - reading from and writing to files. Working with files is essential for data persistence, configuration, logging, and data processing.

Common errors: - **FileNotFoundError**: File doesn't exist - **PermissionError**: No access to file - **IsADirectoryError**: Path is directory, not file - **IOError**: Input/output errors - Encoding issues

Let's master file operations!

11.2 10.1 Reading Files

11.2.1 Basic File Reading

```
# Read entire file
with open('file.txt', 'r') as file:
    content = file.read()
    print(content)

# Read line by line
with open('file.txt', 'r') as file:
    for line in file:
        print(line.strip())  # strip() removes \n
```

```
# Read all lines into list
with open('file.txt', 'r') as file:
    lines = file.readlines()  # List of lines

# Read specific number of characters
with open('file.txt', 'r') as file:
    chunk = file.read(100)  # First 100 characters
```

11.2.2 Error Type 1: `FileNotFoundError: No such file or directory`

Error Message:

```
>>> with open('nonexistent.txt', 'r') as file:
...     content = file.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [Errno 2] No such file or directory: 'nonexistent.txt'
```

What Happened: Trying to open a file that doesn't exist.

Why It Happens: - File doesn't exist at path - Wrong file name or path - Working directory confusion - Typo in filename

Code Example - WRONG:

```
# File doesn't exist
with open('nonexistent.txt', 'r') as file:
    content = file.read()  # ERROR!

# Wrong path
with open('/wrong/path/file.txt', 'r') as file:
    content = file.read()  # ERROR!

# Typo in filename
with open('flie.txt', 'r') as file:  # Typo: flie
    content = file.read()  # ERROR!

# Case sensitivity on Linux/Mac
with open('File.txt', 'r') as file:  # file.txt exists
    content = file.read()  # ERROR on case-sensitive systems
```

Code Example - CORRECT:

```
import os
```

```python
# Check if file exists first
if os.path.exists('file.txt'):
    with open('file.txt', 'r') as file:
        content = file.read()  #
else:
    print("File not found")

# Use try/except
try:
    with open('file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    print("File not found")  #
    content = ""  # Default value

# Use absolute path
file_path = '/home/user/documents/file.txt'
if os.path.exists(file_path):
    with open(file_path, 'r') as file:
        content = file.read()

# Check current directory
print("Current directory:", os.getcwd())
print("Files:", os.listdir())  # List files

# Use os.path.join for cross-platform paths
file_path = os.path.join('data', 'file.txt')
if os.path.exists(file_path):
    with open(file_path, 'r') as file:
        content = file.read()

# Create file if doesn't exist (for writing)
with open('file.txt', 'a') as file:  # 'a' creates if needed
    file.write("Content")
```

11.3 10.2 Writing Files

11.3.1 Basic File Writing

```python
# Write to file (overwrites)
with open('output.txt', 'w') as file:
    file.write("Hello, World!")
```

```
# Append to file
with open('output.txt', 'a') as file:
    file.write("\nNew line")

# Write multiple lines
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
with open('output.txt', 'w') as file:
    file.writelines(lines)

# Write with print (adds newline)
with open('output.txt', 'w') as file:
    print("Hello", file=file)
    print("World", file=file)
```

11.3.2 Error Type 2: `PermissionError: Permission denied`

Error Message:

```
>>> with open('/root/file.txt', 'w') as file:
...     file.write("Content")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
PermissionError: [Errno 13] Permission denied: '/root/file.txt'
```

What Happened: No permission to read/write file.

Why It Happens: - Insufficient permissions - File is read-only - Directory doesn't allow writes - File is locked by another program

Code Example - WRONG:

```
# No permission to write
with open('/root/file.txt', 'w') as file:
    file.write("Content")  # ERROR!

# File is read-only
with open('readonly.txt', 'w') as file:
    file.write("Content")  # ERROR!

# Directory doesn't exist
with open('/nonexistent/dir/file.txt', 'w') as file:
    file.write("Content")  # ERROR!
```

Code Example - CORRECT:

```
import os

```

```
# Check write permission
file_path = 'file.txt'
try:
    with open(file_path, 'w') as file:
        file.write("Content")
except PermissionError:
    print("No permission to write")  #

# Check if directory is writable
dir_path = '/some/directory'
if os.access(dir_path, os.W_OK):
    file_path = os.path.join(dir_path, 'file.txt')
    with open(file_path, 'w') as file:
        file.write("Content")
else:
    print("Directory not writable")

# Create directory if needed
dir_path = 'data'
os.makedirs(dir_path, exist_ok=True)  #   Creates if needed
file_path = os.path.join(dir_path, 'file.txt')
with open(file_path, 'w') as file:
    file.write("Content")

# Write to user's home directory (usually writable)
import os.path
home = os.path.expanduser('~')
file_path = os.path.join(home, 'file.txt')
with open(file_path, 'w') as file:
    file.write("Content")  #

# Write to temp directory
import tempfile
with tempfile.NamedTemporaryFile(mode='w', delete=False) as file:
    file.write("Content")
    temp_path = file.name  #   Always writable
```

11.4 10.3 Context Managers (with statement)

11.4.1 Understanding 'with'

```
# WITHOUT 'with' (manual close)
file = open('file.txt', 'r')
try:
    content = file.read()
finally:
    file.close()  # Must close manually

# WITH 'with' (automatic close)
with open('file.txt', 'r') as file:
    content = file.read()
# File automatically closed

# Multiple files
with open('input.txt', 'r') as infile, \
     open('output.txt', 'w') as outfile:
    content = infile.read()
    outfile.write(content)
# Both files closed automatically
```

11.4.2 Error Type 3: Forgetting to Close Files

What Happened: Not closing files can lead to data loss and resource leaks.

Code Example - WRONG:

```
# Not closing file
file = open('file.txt', 'w')
file.write("Content")
# ERROR! File not closed, data might not be saved

# Exception prevents close
file = open('file.txt', 'r')
content = file.read()
process(content)  # If this raises exception...
file.close()  # ...this never runs

# Closing in wrong place
file = open('file.txt', 'r')
for line in file:
    print(line)
    file.close()  # ERROR! Closes after first line
```

Code Example - CORRECT:

```python
# Use 'with' statement (BEST)
with open('file.txt', 'w') as file:
    file.write("Content")
# File automatically closed

# Use try/finally if not using 'with'
file = open('file.txt', 'r')
try:
    content = file.read()
    process(content)
finally:
    file.close()  #   Always closes

# Correct loop
with open('file.txt', 'r') as file:
    for line in file:
        print(line)
# File closed after loop

# Flush to ensure write
with open('file.txt', 'w') as file:
    file.write("Important data")
    file.flush()  #   Forces write to disk
```

11.5 10.4 File Modes

11.5.1 Understanding File Modes

```python
# Read modes
'r'   # Read (default) - error if doesn't exist
'rb'  # Read binary
'r+'  # Read and write

# Write modes
'w'   # Write - creates new or overwrites
'wb'  # Write binary
'w+'  # Write and read

# Append modes
'a'   # Append - creates if doesn't exist
'ab'  # Append binary
'a+'  # Append and read
```

```
# Exclusive creation
'x'   # Create new - error if exists
```

11.5.2 Error Type 4: FileExistsError: File exists

Error Message:

```
>>> with open('existing.txt', 'x') as file:
...     file.write("Content")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'existing.txt'
```

What Happened: Using 'x' mode on existing file.

Code Example - WRONG:

```
# 'x' mode with existing file
with open('existing.txt', 'x') as file:
    file.write("Content")  # ERROR if file exists

# Wrong mode for operation
with open('file.txt', 'r') as file:
    file.write("Content")  # ERROR! Can't write in 'r' mode

# Binary mode with string
with open('file.txt', 'wb') as file:
    file.write("Hello")  # ERROR! Need bytes, not string
```

Code Example - CORRECT:

```
# Check before using 'x'
if not os.path.exists('file.txt'):
    with open('file.txt', 'x') as file:
        file.write("Content")  #
else:
    print("File already exists")

# Use appropriate mode
with open('file.txt', 'w') as file:  #   For writing
    file.write("Content")

with open('file.txt', 'r') as file:  #   For reading
    content = file.read()

# Binary mode with bytes
```

```python
with open('file.txt', 'wb') as file:
    file.write(b"Hello")  #   Bytes object

# Or encode string
with open('file.txt', 'wb') as file:
    file.write("Hello".encode('utf-8'))  #

# Text mode (default)
with open('file.txt', 'w') as file:
    file.write("Hello")  #   String

# Safe overwrite pattern
import shutil
if os.path.exists('file.txt'):
    shutil.copy('file.txt', 'file.txt.bak')  # Backup
with open('file.txt', 'w') as file:
    file.write("New content")  #
```

11.6 10.5 Working with Paths

11.6.1 Path Operations

```python
import os

# Current directory
current = os.getcwd()

# Change directory
os.chdir('/path/to/directory')

# Join paths (cross-platform)
path = os.path.join('data', 'files', 'document.txt')
# Windows: data\files\document.txt
# Unix: data/files/document.txt

# Split path
directory, filename = os.path.split('/path/to/file.txt')
# directory: '/path/to'
# filename: 'file.txt'

# Get filename and extension
filename, ext = os.path.splitext('document.txt')
# filename: 'document'
```

```
# ext: '.txt'

# Check existence
os.path.exists('file.txt')      # File or directory
os.path.isfile('file.txt')      # File only
os.path.isdir('directory')      # Directory only

# Get absolute path
abs_path = os.path.abspath('file.txt')

# List directory contents
files = os.listdir('directory')

# Create directory
os.makedirs('path/to/directory', exist_ok=True)
```

11.7 10.6 Common File Patterns

11.7.1 Useful Patterns

```
# Read file safely
def read_file(filename):
    try:
        with open(filename, 'r') as file:
            return file.read()
    except FileNotFoundError:
        return None

# Write file safely
def write_file(filename, content):
    try:
        with open(filename, 'w') as file:
            file.write(content)
        return True
    except (PermissionError, IOError):
        return False

# Copy file
def copy_file(source, destination):
    with open(source, 'r') as infile:
        with open(destination, 'w') as outfile:
            outfile.write(infile.read())
```

```
# Process file line by line
def process_large_file(filename):
    with open(filename, 'r') as file:
        for line in file:  # Memory efficient
            process_line(line.strip())

# Read CSV
def read_csv(filename):
    data = []
    with open(filename, 'r') as file:
        for line in file:
            data.append(line.strip().split(','))
    return data

# Write CSV
def write_csv(filename, data):
    with open(filename, 'w') as file:
        for row in data:
            file.write(','.join(str(x) for x in row) + '\n')
```

11.8 10.7 Practice Problems

11.8.1 Problem 1: File Not Found

```
with open('data.txt', 'r') as file:
    content = file.read()
```

Click for Answer

Error: `FileNotFoundError: No such file or directory: 'data.txt'`

Fix:

```
import os

# Check first
if os.path.exists('data.txt'):
    with open('data.txt', 'r') as file:
        content = file.read()
else:
    print("File not found")  #

# Or use try/except
try:
```

```
    with open('data.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    content = ""  #  Default
```

11.8.2 Problem 2: Not Closing File

```
file = open('output.txt', 'w')
file.write("Content")
```

Click for Answer

Issue: File not closed, data might not be saved

Fix:

```
# Use 'with' statement
with open('output.txt', 'w') as file:
    file.write("Content")
# File automatically closed
```

11.8.3 Problem 3: Wrong Mode

```
with open('file.txt', 'r') as file:
    file.write("New content")
```

Click for Answer

Error: `io.UnsupportedOperation: not writable`

Fix:

```
# Use write mode
with open('file.txt', 'w') as file:  #  Use 'w'
    file.write("New content")

# Or append mode
with open('file.txt', 'a') as file:  #  Use 'a'
    file.write("New content")
```

11.9 10.8 Key Takeaways

11.9.1 What You Learned

1. **Check file exists** - Before opening for reading
2. **Use 'with' statement** - Automatically closes files
3. **Handle exceptions** - FileNotFoundError, PermissionError
4. **Use correct mode** - 'r' for read, 'w' for write, 'a' for append
5. **Use os.path.join** - Cross-platform paths
6. **Close files** - Or use 'with' to auto-close
7. **Check permissions** - Before writing

11.9.2 Common Patterns

```
# Pattern 1: Safe read
try:
    with open('file.txt', 'r') as file:
        content = file.read()
except FileNotFoundError:
    content = ""

# Pattern 2: Safe write
with open('file.txt', 'w') as file:
    file.write(content)

# Pattern 3: Check exists
if os.path.exists('file.txt'):
    # Process file

# Pattern 4: Create path
os.makedirs('path/to/dir', exist_ok=True)
```

11.9.3 Error Summary

Error	Cause	Prevention
`FileNotFoundError`	File doesn't exist	Check with os.path.exists()
`PermissionError`	No access rights	Check permissions or use try/except
Not closing	Forgot to close	Use 'with' statement
Wrong mode	Incorrect r/w/a	Choose correct mode

11.10 10.9 Congratulations!

11.10.1 You Completed Part I: Python Fundamentals!

You've mastered: - Variables and Data Types (Chapter 1) - Operators and Expressions (Chapter 2) - Strings and String Methods (Chapter 3) - Lists and List Methods (Chapter 4) - Dictionaries and Sets (Chapter 5) - Tuples and Immutability (Chapter 6) - Conditional Statements (Chapter 7) - Loops (Chapter 8) - Functions (Chapter 9) - File I/O (Chapter 10)

11.11 10.10 Moving Forward

Part I Complete! You now have a solid foundation in Python fundamentals.

What's Next?

11.11.1 Part II: Libraries and Data (Chapters 11-15)

- Chapter 11: Regular Expressions
- Chapter 12: Pandas Basics
- Chapter 13: Pandas Advanced
- Chapter 14: NumPy
- Chapter 15: Matplotlib

11.11.2 Part III: Advanced Topics (Chapters 16-20)

- Chapter 16: Object-Oriented Programming
- Chapter 17: Modules and Imports
- Chapter 18: Exception Handling
- Chapter 19: Debugging Techniques
- Chapter 20: Testing and Code Quality

Chapter 12

Wrong escape

pattern = ‘+̣’ # Should be raw string re.search(pattern, text) # Might not work as expected

~~~
**Code Example - CORRECT:**
```python
import re

# Close all parentheses
pattern = r'(hello)'  #   Closed
match = re.search(pattern, 'hello world')

# Close all brackets
pattern = r'[abc]'  #   Closed
match = re.search(pattern, 'abc')

# Valid quantifier syntax
pattern = r'a{0,5}'  #   Valid: 0 to 5 times
pattern = r'a{5}'    #   Valid: exactly 5 times
pattern = r'a{5,}'   #   Valid: 5 or more times

# Escape special characters
pattern = r'price: \$50'  #   Escaped $
match = re.search(pattern, 'price: $50')

# Use raw strings for regex patterns
pattern = r'\d+'  #   Raw string (r prefix)
match = re.search(pattern, '123')

# Test pattern before using
~~~

```
try:
    re.compile(pattern)
except re.error as e:
    print(f"Invalid pattern: {e}")
```

Regex Special Characters:

```
# Characters that need escaping:
. ^ $ * + ? { } [ ] \ | ( )

# To match literally, escape with \
pattern = r'\.'   # Matches literal .
pattern = r'\$'   # Matches literal $
pattern = r'\('   # Matches literal (
pattern = r'\\'   # Matches literal \

# In character class, different rules
pattern = r'[.]'  # . doesn't need escape in []
pattern = r'[\^]' # ^ needs escape in []
```

12.1 11.2 Common Regex Patterns

12.1.1 Useful Patterns

```
import re

# Digits
pattern = r'\d'     # Any digit [0-9]
pattern = r'\d+'    # One or more digits
pattern = r'\d{3}'  # Exactly 3 digits

# Letters
pattern = r'[a-z]'  # Lowercase letter
pattern = r'[A-Z]'  # Uppercase letter
pattern = r'[a-zA-Z]'  # Any letter

# Whitespace
pattern = r'\s'     # Any whitespace
pattern = r'\s+'    # One or more whitespace

# Word characters
pattern = r'\w'     # Letter, digit, or underscore
pattern = r'\w+'    # One or more word characters
```

```
# Beginning and end
pattern = r'^hello' # Starts with "hello"
pattern = r'world$' # Ends with "world"

# Email pattern
email_pattern = r'[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}'

# Phone pattern (US)
phone_pattern = r'\d{3}-\d{3}-\d{4}'

# URL pattern
url_pattern = r'https?://[^\s]+'
```

12.2 11.3 Match vs Search vs Findall

12.2.1 Understanding Different Methods

```
import re

text = "The cat and the bat sat on the mat"

# match() - checks beginning only
match = re.match(r'cat', text)
print(match)  # None - doesn't start with 'cat'

match = re.match(r'The', text)
print(match)  # <Match object> - starts with 'The'

# search() - finds first occurrence anywhere
match = re.search(r'cat', text)
print(match.group())  # 'cat' - found it

# findall() - finds all occurrences
matches = re.findall(r'at', text)
print(matches)  # ['at', 'at', 'at', 'at']

# finditer() - iterator of match objects
for match in re.finditer(r'at', text):
    print(f"Found at position {match.start()}: {match.group()}")
```

12.2.2 Error Type 2: `AttributeError: 'NoneType' object has no attribute 'group'`

Error Message:

```
>>> import re
>>> match = re.search(r'xyz', 'abc')
>>> print(match.group())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'NoneType' object has no attribute 'group'
```

What Happened: Pattern not found, match is None, trying to call .group().

Why It Happens: - Pattern doesn't exist in text - Wrong pattern - Not checking if match succeeded

Code Example - WRONG:

```
import re

# Not checking if match found
text = "Hello World"
match = re.search(r'xyz', text)
print(match.group())  # ERROR! match is None

# Using wrong method result
matches = re.findall(r'world', text)
print(matches.group())  # ERROR! findall returns list, not match object

# Assuming match always succeeds
email = "not-an-email"
match = re.search(r'[\w.]+@[\w.]+', email)
domain = match.group()  # ERROR if no match
```

Code Example - CORRECT:

```
import re

# Check if match exists
text = "Hello World"
match = re.search(r'xyz', text)
if match:
    print(match.group())
else:
    print("Not found")  #

# Use findall correctly (returns list)
matches = re.findall(r'World', text)
```

```python
if matches:
    print(matches[0])  #   Access list element

# Safe pattern with default
email = "not-an-email"
match = re.search(r'[\w.]+@[\w.]+', email)
domain = match.group() if match else "invalid"  #

# Use try/except
try:
    match = re.search(r'pattern', text)
    result = match.group()
except AttributeError:
    result = None  #

# Helper function
def safe_search(pattern, text, default=""):
    """Safely search with default"""
    match = re.search(pattern, text)
    return match.group() if match else default

result = safe_search(r'xyz', text, default="not found")  #
```

12.3 11.4 Groups and Capturing

12.3.1 Extracting Parts

```python
import re

# Basic grouping
text = "John: 30"
match = re.search(r'(\w+): (\d+)', text)
if match:
    name = match.group(1)  # "John"
    age = match.group(2)   # "30"
    full = match.group(0)  # "John: 30" (entire match)

# Named groups
match = re.search(r'(?P<name>\w+): (?P<age>\d+)', text)
if match:
    name = match.group('name')  # "John"
    age = match.group('age')    # "30"
```

```
# Extract email parts
email = "user@example.com"
match = re.search(r'(?P<user>[\w.]+)@(?P<domain>[\w.]+)', email)
if match:
    user = match.group('user')      # "user"
    domain = match.group('domain')  # "example.com"

# Multiple matches with groups
text = "John:30, Jane:25, Bob:35"
matches = re.findall(r'(\w+):(\d+)', text)
for name, age in matches:
    print(f"{name} is {age}")
```

12.4 11.5 Greedy vs Non-Greedy

12.4.1 Understanding Quantifiers

```
import re

text = "<div>content</div>"

# Greedy (default) - matches as much as possible
match = re.search(r'<.*>', text)
print(match.group())  # "<div>content</div>" - entire string

# Non-greedy - matches as little as possible
match = re.search(r'<.*?>', text)
print(match.group())  # "<div>" - stops at first >

# Examples
text = "aaa"
re.findall(r'a+', text)   # ['aaa'] - greedy
re.findall(r'a+?', text)  # ['a', 'a', 'a'] - non-greedy

text = '123'
re.findall(r'\d{2,4}', text)   # ['123'] - greedy (max 4)
re.findall(r'\d{2,4}?', text)  # ['12'] - non-greedy (min 2)

# Practical example - extract HTML tags
html = "<p>First</p><p>Second</p>"
re.findall(r'<p>.*?</p>', html)  #   ['<p>First</p>', '<p>Second</p>']
re.findall(r'<p>.*</p>', html)   # ['<p>First</p><p>Second</p>'] - greedy
```

12.5 11.6 Substitution

12.5.1 Replacing Patterns

```python
import re

# Simple replacement
text = "Hello World"
result = re.sub(r'World', 'Python', text)
print(result)  # "Hello Python"

# Replace with function
def uppercase(match):
    return match.group().upper()

text = "hello world"
result = re.sub(r'\w+', uppercase, text)
print(result)  # "HELLO WORLD"

# Replace using groups
text = "John Doe"
result = re.sub(r'(\w+) (\w+)', r'\2, \1', text)
print(result)  # "Doe, John"

# Replace with named groups
result = re.sub(r'(?P<first>\w+) (?P<last>\w+)',
                r'\g<last>, \g<first>', text)
print(result)  # "Doe, John"

# Limit replacements
text = "cat bat cat rat"
result = re.sub(r'cat', 'dog', text, count=1)
print(result)  # "dog bat cat rat"

# Case-insensitive replacement
result = re.sub(r'WORLD', 'Python', 'Hello WORLD', flags=re.IGNORECASE)
print(result)  # "Hello Python"
```

12.6 11.7 Flags

12.6.1 Regex Modifiers

```
import re

# Case-insensitive
text = "Hello WORLD"
match = re.search(r'world', text, re.IGNORECASE)
# Or: re.IGNORECASE, re.I

# Multiline - ^ and $ match line boundaries
text = "line1\nline2\nline3"
matches = re.findall(r'^line', text, re.MULTILINE)
# Matches: ['line', 'line', 'line']

# Dotall - . matches newlines
text = "first\nsecond"
match = re.search(r'first.second', text, re.DOTALL)
# Matches across newline

# Verbose - allows comments and whitespace
pattern = r'''
    \d{3}  # Area code
    -      # Separator
    \d{3}  # Prefix
    -      # Separator
    \d{4}  # Line number
'''
match = re.search(pattern, '555-123-4567', re.VERBOSE)

# Combine flags
match = re.search(r'pattern', text, re.IGNORECASE | re.MULTILINE)
```

12.7 11.8 Common Patterns and Use Cases

12.7.1 Practical Examples

```
import re

# Validate email
def is_valid_email(email):
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
```

```
    return re.match(pattern, email) is not None

# Validate phone number (US)
def is_valid_phone(phone):
    pattern = r'^\d{3}-\d{3}-\d{4}$'
    return re.match(pattern, phone) is not None

# Extract URLs from text
def extract_urls(text):
    pattern = r'https?://[^\s]+'
    return re.findall(pattern, text)

# Remove HTML tags
def remove_html_tags(html):
    pattern = r'<[^>]+>'
    return re.sub(pattern, '', html)

# Extract numbers from string
def extract_numbers(text):
    pattern = r'\d+'
    return [int(x) for x in re.findall(pattern, text)]

# Validate password (8+ chars, letter, number)
def is_strong_password(password):
    if len(password) < 8:
        return False
    if not re.search(r'[a-zA-Z]', password):
        return False
    if not re.search(r'\d', password):
        return False
    return True

# Parse log line
def parse_log_line(line):
    pattern = r'(?P<date>\S+) (?P<time>\S+) (?P<level>\w+) (?P<message>.*)'
    match = re.match(pattern, line)
    if match:
        return match.groupdict()
    return None
```

12.8 11.9 Practice Problems

12.8.1 Problem 1: Invalid Pattern

```
import re
pattern = '(hello'
re.search(pattern, 'hello world')
```

Click for Answer

Error: `re.error: missing ), unterminated subpattern`

Fix:

```
import re
pattern = r'(hello)'  #  Close parentheses
match = re.search(pattern, 'hello world')
```

12.8.2 Problem 2: NoneType Error

```
import re
text = "Hello World"
match = re.search(r'xyz', text)
print(match.group())
```

Click for Answer

Error: `AttributeError: 'NoneType' object has no attribute 'group'`

Fix:

```
import re
text = "Hello World"
match = re.search(r'xyz', text)
if match:  #  Check if found
    print(match.group())
else:
    print("Not found")
```

12.8.3 Problem 3: Greedy Match

```
import re
html = "<p>First</p><p>Second</p>"
matches = re.findall(r'<p>.*</p>', html)
print(matches)
```

Click for Answer

Issue: Returns entire string (greedy)

Fix:

```
import re
html = "<p>First</p><p>Second</p>"
matches = re.findall(r'<p>.*?</p>', html)  #  Non-greedy
print(matches)  # ['<p>First</p>', '<p>Second</p>']
```

12.9 11.10 Key Takeaways

12.9.1 What You Learned

1. **Use raw strings** - r' ' for regex patterns
2. **Check match results** - Before calling .group()
3. **Escape special characters** - . $ (etc.
4. **Use non-greedy** - .*? for minimal matching
5. **Validate patterns** - Test with re.compile()
6. **Use named groups** - (?P...) for clarity
7. **Choose right method** - match/search/findall/sub

12.9.2 Common Patterns

```
# Pattern 1: Safe search
match = re.search(pattern, text)
if match:
    result = match.group()

# Pattern 2: Extract all
matches = re.findall(pattern, text)

# Pattern 3: Replace
result = re.sub(pattern, replacement, text)

# Pattern 4: Validate
def is_valid(text):
    return re.match(pattern, text) is not None
```

12.9.3 Error Summary

Error	Cause	Prevention
`re.error`	Invalid pattern syntax	Use raw strings, close brackets
`AttributeError`	match is None	Check if match before .group()
Greedy issues	Using .*	Use .*? for non-greedy

12.10 11.11 Moving Forward

You now understand regular expressions! In **Chapter 12**, we'll explore **Pandas Basics** - data analysis with DataFrames!

Chapter 13

Chapter 12: Pandas Basics - DataFrame Errors

13.1 Introduction

Welcome to **Pandas** - Python's powerful data analysis library! Pandas provides DataFrames for working with structured data (like spreadsheets or SQL tables). It's essential for data science and analysis.

Common errors: - **KeyError**: Column/index doesn't exist - **ValueError**: Wrong shape or values - **AttributeError**: Wrong method for operation - **TypeError**: Wrong data types - Index alignment issues

Let's master Pandas!

13.2 12.1 Creating DataFrames

13.2.1 Basic DataFrame Creation

```
import pandas as pd

# From dictionary
data = {
    'name': ['Alice', 'Bob', 'Charlie'],
    'age': [25, 30, 35],
    'city': ['NYC', 'LA', 'Chicago']
}
df = pd.DataFrame(data)
```

```
# From list of lists
data = [
    ['Alice', 25, 'NYC'],
    ['Bob', 30, 'LA'],
    ['Charlie', 35, 'Chicago']
]
df = pd.DataFrame(data, columns=['name', 'age', 'city'])

# From list of dictionaries
data = [
    {'name': 'Alice', 'age': 25, 'city': 'NYC'},
    {'name': 'Bob', 'age': 30, 'city': 'LA'}
]
df = pd.DataFrame(data)

# Read from CSV
df = pd.read_csv('data.csv')

# Basic info
print(df.head())      # First 5 rows
print(df.tail())      # Last 5 rows
print(df.shape)       # (rows, columns)
print(df.columns)     # Column names
print(df.dtypes)      # Data types
print(df.info())      # Overview
```

13.2.2 Error Type 1: `KeyError: 'column_name'`

Error Message:

```
>>> import pandas as pd
>>> df = pd.DataFrame({'name': ['Alice', 'Bob'], 'age': [25, 30]})
>>> df['salary']
Traceback (most recent call last):
  ...
KeyError: 'salary'
```

What Happened: Trying to access a column that doesn't exist.

Why It Happens: - Column doesn't exist in DataFrame - Typo in column name - Case sensitivity - Using wrong accessor

Code Example - WRONG:

```
import pandas as pd

df = pd.DataFrame({
    'name': ['Alice', 'Bob'],
    'age': [25, 30]
})

# Non-existent column
salary = df['salary']  # ERROR! Column doesn't exist

# Typo
name = df['nane']  # ERROR! Typo

# Case sensitivity
age = df['Age']  # ERROR! Column is 'age' not 'Age'

# Wrong bracket type
name = df('name')  # ERROR! Use [] not ()

# Multiple columns with typo
subset = df[['name', 'salary']]  # ERROR! 'salary' doesn't exist
```

Code Example - CORRECT:

```
import pandas as pd

df = pd.DataFrame({
    'name': ['Alice', 'Bob'],
    'age': [25, 30]
})

# Check if column exists
if 'salary' in df.columns:
    salary = df['salary']
else:
    print("Column doesn't exist")  #

# Use .get() for Series (doesn't work for DataFrame columns)
# But can use try/except
try:
    salary = df['salary']
except KeyError:
    salary = None  #

# Check available columns
print(df.columns.tolist())  # ['name', 'age']
```

```python
# Correct spelling
name = df['name']  #

# Match case exactly
age = df['age']  #

# Use correct brackets
name = df['name']  #

# Safe column selection
columns_to_select = ['name', 'age']
existing_cols = [col for col in columns_to_select if col in df.columns]
subset = df[existing_cols]  #

# Add missing column with default
if 'salary' not in df.columns:
    df['salary'] = 0  #   Add with default value

# Use .loc for safe access
try:
    data = df.loc[:, 'salary']
except KeyError:
    df['salary'] = 0
    data = df.loc[:, 'salary']  #
```

13.3 12.2 Selecting Data

13.3.1 Accessing Rows and Columns

```python
import pandas as pd

df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie'],
    'age': [25, 30, 35],
    'city': ['NYC', 'LA', 'Chicago']
})

# Select column (returns Series)
ages = df['age']

# Select multiple columns (returns DataFrame)
subset = df[['name', 'age']]
```

```
# Select rows by index position (.iloc)
first_row = df.iloc[0]           # First row
first_three = df.iloc[0:3]       # First 3 rows
last_row = df.iloc[-1]           # Last row

# Select rows by label (.loc)
df_indexed = df.set_index('name')
alice = df_indexed.loc['Alice']

# Select specific cells
value = df.loc[0, 'age']         # Row 0, column 'age'
value = df.iloc[0, 1]            # Row 0, column 1

# Boolean indexing
adults = df[df['age'] >= 30]
in_nyc = df[df['city'] == 'NYC']

# Multiple conditions
result = df[(df['age'] >= 30) & (df['city'] == 'LA')]
```

13.3.2 Error Type 2: `ValueError: Location based indexing can only have [integer, integer slice, listlike of integers, boolean array] types`

Error Message:

```
>>> df = pd.DataFrame({'name': ['Alice', 'Bob'], 'age': [25, 30]})
>>> df.iloc['Alice']
Traceback (most recent call last):
  ...
ValueError: Location based indexing can only have [integer, integer slice...
```

What Happened: Using wrong indexing method (.iloc vs .loc).

Why It Happens: - Using labels with .iloc (needs integers) - Using integers with .loc on non-integer index - Confusing .iloc and .loc - Wrong indexing syntax

Code Example - WRONG:

```
import pandas as pd

df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie'],
    'age': [25, 30, 35]
})
```

```
# Using label with .iloc
row = df.iloc['Alice']  # ERROR! .iloc needs integer

# Using column name with .iloc
ages = df.iloc[:, 'age']  # ERROR! Use column index or .loc

# Wrong syntax
row = df.iloc['name' == 'Alice']  # ERROR! Wrong method
```

Code Example - CORRECT:

```
import pandas as pd

df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie'],
    'age': [25, 30, 35]
})

# Use .iloc with integers
first_row = df.iloc[0]  #   Integer index
first_three = df.iloc[0:3]  #   Integer slice

# Use .loc with labels/conditions
# First, set index if you want to use labels
df_indexed = df.set_index('name')
alice = df_indexed.loc['Alice']  #   Label index

# Or use .loc with column names
value = df.loc[0, 'age']  #   Row by position, column by name

# Use .iloc for column by position
ages = df.iloc[:, 1]  #   All rows, second column

# Boolean indexing (use direct or .loc)
adults = df[df['age'] >= 30]  #
# Or
adults = df.loc[df['age'] >= 30]  #

# Remember:
# .loc[row_label, column_label]  - uses labels
# .iloc[row_position, column_position]  - uses integers

# Examples:
df.loc[0, 'age']      #   Row 0, column 'age'
df.iloc[0, 1]         #   Row 0, column 1
```

```
df.loc[0:2, ['name', 'age']]  #  Rows 0-2, specific columns
df.iloc[0:2, 0:2]     #  First 2 rows, first 2 columns
```

13.4 12.3 Data Types

13.4.1 Understanding dtypes

```
import pandas as pd

# Check data types
df = pd.DataFrame({
    'name': ['Alice', 'Bob'],
    'age': [25, 30],
    'salary': [50000.0, 60000.0],
    'hired': ['2020-01-01', '2021-06-15']
})

print(df.dtypes)
# name       object
# age        int64
# salary     float64
# hired      object

# Convert types
df['age'] = df['age'].astype(float)
df['hired'] = pd.to_datetime(df['hired'])

# Check for missing values
print(df.isnull())
print(df.isnull().sum())  # Count per column

# Fill missing values
df['age'].fillna(0, inplace=True)
df['name'].fillna('Unknown', inplace=True)

# Drop missing values
df_clean = df.dropna()  # Drop rows with any NaN
df_clean = df.dropna(subset=['age'])  # Drop rows with NaN in 'age'
```

13.4.2 Error Type 3: `TypeError: cannot concatenate object of type`

Error Message:

```
>>> df = pd.DataFrame({'age': ['25', '30']})
>>> df['age'].mean()
Traceback (most recent call last):
  ...
TypeError: Could not convert 25 30 to numeric
```

What Happened: Trying to perform numeric operations on non-numeric data.

Why It Happens: - Column contains strings not numbers - Mixed types in column - Wrong data type - Not converting before operation

Code Example - WRONG:

```
import pandas as pd

# Numeric operations on strings
df = pd.DataFrame({'age': ['25', '30', '35']})
average = df['age'].mean()  # ERROR! Strings not numbers

# Mixed types
df = pd.DataFrame({'value': [1, 2, '3', 4]})
total = df['value'].sum()  # ERROR! Mixed types

# String operations on numbers
df = pd.DataFrame({'code': [101, 102, 103]})
upper = df['code'].str.upper()  # ERROR! Not strings
```

Code Example - CORRECT:

```
import pandas as pd

# Convert to numeric first
df = pd.DataFrame({'age': ['25', '30', '35']})
df['age'] = pd.to_numeric(df['age'])  #   Convert
average = df['age'].mean()  #   Works now

# Handle errors in conversion
df = pd.DataFrame({'value': ['1', '2', 'invalid', '4']})
df['value'] = pd.to_numeric(df['value'], errors='coerce')  #   NaN for invalid
# value: [1.0, 2.0, NaN, 4.0]

# Check dtype before operations
if pd.api.types.is_numeric_dtype(df['age']):
    average = df['age'].mean()  #
```

```
else:
    print("Not numeric")

# Convert on creation
df = pd.DataFrame({
    'age': [25, 30, 35]  #   Use numbers not strings
})

# Convert to string for string operations
df = pd.DataFrame({'code': [101, 102, 103]})
df['code'] = df['code'].astype(str)  #   Convert to string
upper = df['code'].str.upper()  #   Now works

# Specify dtypes when reading CSV
df = pd.read_csv('data.csv', dtype={'age': int, 'name': str})  #

# Handle mixed types
df = pd.DataFrame({'value': [1, 2, '3', 4]})
df['value'] = df['value'].apply(lambda x: int(x) if isinstance(x, str) else x)  #
```

13.5 12.4 Adding and Modifying Data

13.5.1 Creating and Changing Columns

```
import pandas as pd

df = pd.DataFrame({
    'name': ['Alice', 'Bob'],
    'age': [25, 30]
})

# Add new column
df['city'] = 'NYC'  # Same value for all
df['salary'] = [50000, 60000]  # Different values

# Create from calculation
df['age_in_months'] = df['age'] * 12

# Modify existing column
df['age'] = df['age'] + 1

# Conditional creation
df['is_adult'] = df['age'] >= 18
```

```
# Using .loc for modification
df.loc[df['age'] > 30, 'category'] = 'senior'
df.loc[df['age'] <= 30, 'category'] = 'junior'

# Apply function
df['name_upper'] = df['name'].apply(lambda x: x.upper())

# Rename columns
df.rename(columns={'age': 'years'}, inplace=True)

# Drop columns
df.drop('city', axis=1, inplace=True)
# Or
df = df.drop(columns=['city'])
```

13.5.2 Error Type 4: `ValueError: Length of values does not match length of index`

Error Message:

```
>>> df = pd.DataFrame({'name': ['Alice', 'Bob', 'Charlie']})
>>> df['age'] = [25, 30]
Traceback (most recent call last):
  ...
ValueError: Length of values (2) does not match length of index (3)
```

What Happened: Trying to assign list with wrong length to column.

Why It Happens: - List length doesn't match DataFrame rows - Wrong number of values - Off-by-one error

Code Example - WRONG:

```
import pandas as pd

df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie']
})

# Too few values
df['age'] = [25, 30]  # ERROR! 2 values, 3 rows

# Too many values
df['city'] = ['NYC', 'LA', 'Chicago', 'Boston']  # ERROR! 4 values, 3 rows
```

Code Example - CORRECT:

```
import pandas as pd

df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie']
})

# Match number of rows
df['age'] = [25, 30, 35]  #   3 values for 3 rows

# Use single value (broadcasts)
df['country'] = 'USA'  #   Same value for all rows

# Check length first
ages = [25, 30]
if len(ages) == len(df):
    df['age'] = ages
else:
    print(f"Wrong length: need {len(df)}, got {len(ages)}")

# Pad with default if needed
ages = [25, 30]
while len(ages) < len(df):
    ages.append(0)  # Pad with 0
df['age'] = ages  #

# Use .loc for conditional assignment
df['age'] = 0  # Initialize
df.loc[0, 'age'] = 25
df.loc[1, 'age'] = 30
df.loc[2, 'age'] = 35  #

# Create from Series (index-aligned)
ages = pd.Series([25, 30, 35], index=[0, 1, 2])
df['age'] = ages  #   Aligns by index
```

13.6 12.5 Filtering Data

13.6.1 Boolean Indexing

```
import pandas as pd

df = pd.DataFrame({
```

```
    'name': ['Alice', 'Bob', 'Charlie', 'David'],
    'age': [25, 30, 35, 40],
    'city': ['NYC', 'LA', 'NYC', 'Chicago']
})

# Single condition
adults_30plus = df[df['age'] >= 30]

# Multiple conditions (AND)
result = df[(df['age'] >= 30) & (df['city'] == 'NYC')]

# Multiple conditions (OR)
result = df[(df['age'] < 25) | (df['age'] > 35)]

# NOT condition
not_nyc = df[~(df['city'] == 'NYC')]
# Or
not_nyc = df[df['city'] != 'NYC']

# String contains
in_name = df[df['name'].str.contains('a', case=False)]

# isin() for multiple values
cities = df[df['city'].isin(['NYC', 'LA'])]

# Between
age_range = df[df['age'].between(25, 35)]

# Query method (alternative)
result = df.query('age >= 30 and city == "NYC"')
```

13.7 12.6 Common Operations

13.7.1 Useful DataFrame Operations

```
import pandas as pd

df = pd.DataFrame({
    'name': ['Alice', 'Bob', 'Charlie'],
    'age': [25, 30, 35],
    'salary': [50000, 60000, 70000]
})
```

```
# Sort
df_sorted = df.sort_values('age')
df_sorted = df.sort_values('age', ascending=False)
df_sorted = df.sort_values(['age', 'salary'])

# Group by
grouped = df.groupby('city')['salary'].mean()
grouped = df.groupby('city').agg({
    'age': 'mean',
    'salary': 'sum'
})

# Reset index
df_reset = df.reset_index(drop=True)

# Set index
df_indexed = df.set_index('name')

# Drop duplicates
df_unique = df.drop_duplicates()
df_unique = df.drop_duplicates(subset=['name'])

# Value counts
counts = df['city'].value_counts()

# Unique values
unique = df['city'].unique()

# Replace values
df['city'] = df['city'].replace('NYC', 'New York')

# Map values
city_map = {'NYC': 'New York', 'LA': 'Los Angeles'}
df['city_full'] = df['city'].map(city_map)
```

13.8 12.7 Practice Problems

13.8.1 Problem 1: KeyError

```
import pandas as pd
df = pd.DataFrame({'name': ['Alice'], 'age': [25]})
print(df['salary'])
```

Click for Answer

Error: `KeyError: 'salary'`

Fix:

```
import pandas as pd
df = pd.DataFrame({'name': ['Alice'], 'age': [25]})

# Check first
if 'salary' in df.columns:
    print(df['salary'])
else:
    print("Column doesn't exist")  #

# Or add with default
df['salary'] = 0
print(df['salary'])  #
```

13.8.2 Problem 2: Wrong Indexer

```
import pandas as pd
df = pd.DataFrame({'name': ['Alice', 'Bob'], 'age': [25, 30]})
print(df.iloc[:, 'age'])
```

Click for Answer

Error: `ValueError: Location based indexing can only have...`

Fix:

```
import pandas as pd
df = pd.DataFrame({'name': ['Alice', 'Bob'], 'age': [25, 30]})

# Use .loc for column names
print(df.loc[:, 'age'])  #

# Or use .iloc with column position
print(df.iloc[:, 1])  #

# Or direct column access
print(df['age'])  #
```

13.8.3 Problem 3: Length Mismatch

```
import pandas as pd
df = pd.DataFrame({'name': ['Alice', 'Bob', 'Charlie']})
df['age'] = [25, 30]
```

Click for Answer

Error: `ValueError: Length of values does not match length of index`

Fix:

```
import pandas as pd
df = pd.DataFrame({'name': ['Alice', 'Bob', 'Charlie']})

# Match number of rows
df['age'] = [25, 30, 35]  #   3 values

# Or use single value
df['age'] = 25  #   Same for all
```

13.9 12.8 Key Takeaways

13.9.1 What You Learned

1. **Check columns exist** - Use `in df.columns`
2. **Use .loc for labels** - .iloc for positions
3. **Convert data types** - pd.to_numeric(), .astype()
4. **Match lengths** - Values must match row count
5. **Use boolean indexing** - For filtering
6. **Handle missing values** - .fillna(), .dropna()
7. **Check dtypes** - Before operations

13.9.2 Common Patterns

```
# Pattern 1: Safe column access
if 'column' in df.columns:
    data = df['column']

# Pattern 2: Convert types
df['col'] = pd.to_numeric(df['col'], errors='coerce')

# Pattern 3: Filter data
filtered = df[df['age'] >= 30]
```

```
# Pattern 4: Add column safely
df['new'] = df['old'] * 2
```

13.9.3 Error Summary

Error	Cause	Prevention
`KeyError`	Column doesn't exist	Check with `in df.columns`
`ValueError` (indexing)	Wrong indexer (.iloc vs .loc)	Use .loc for labels, .iloc for positions
`TypeError`	Wrong data type	Convert with pd.to_numeric()
`ValueError` (length)	Wrong number of values	Match DataFrame length

13.10 12.9 Moving Forward

You now understand Pandas basics! In **Chapter 13**, we'll explore **Pandas Advanced -** merging, pivoting, and complex operations!

Chapter 14

Chapter 13: Pandas Advanced - Complex Operations

14.1 Introduction

You've learned Pandas basics. Now let's explore **advanced Pandas** - merging, joining, pivoting, grouping, and complex data transformations. These skills are essential for real-world data analysis.

Common errors: - **MergeError**: Wrong merge keys - **ValueError**: Shape mismatches - **KeyError**: Index/column issues - Memory errors with large datasets

Let's master advanced Pandas!

14.2 13.1 Merging DataFrames

14.2.1 Combining DataFrames

```
import pandas as pd

# Sample data
df1 = pd.DataFrame({
    'id': [1, 2, 3],
    'name': ['Alice', 'Bob', 'Charlie']
})
```

```
df2 = pd.DataFrame({
    'id': [1, 2, 4],
    'salary': [50000, 60000, 70000]
})

# Inner join (default)
merged = pd.merge(df1, df2, on='id')
# Only keeps matching rows (id 1, 2)

# Left join
merged = pd.merge(df1, df2, on='id', how='left')
# Keeps all df1 rows, fills NaN for missing df2

# Right join
merged = pd.merge(df1, df2, on='id', how='right')
# Keeps all df2 rows

# Outer join
merged = pd.merge(df1, df2, on='id', how='outer')
# Keeps all rows from both

# Merge on different column names
df1 = pd.DataFrame({'emp_id': [1, 2], 'name': ['Alice', 'Bob']})
df2 = pd.DataFrame({'employee_id': [1, 2], 'salary': [50000, 60000]})
merged = pd.merge(df1, df2, left_on='emp_id', right_on='employee_id')

# Merge on index
merged = pd.merge(df1, df2, left_index=True, right_index=True)
```

14.2.2 Error Type 1: `MergeError` or Wrong Results

What Happened: Merge produces unexpected results or errors.

Why It Happens: - Wrong merge key - Duplicate keys - Missing values in key columns - Wrong merge type

Code Example - WRONG:

```
import pandas as pd

df1 = pd.DataFrame({
    'id': [1, 2, 3],
    'name': ['Alice', 'Bob', 'Charlie']
})
```

```
df2 = pd.DataFrame({
    'employee_id': [1, 2, 4],  # Different column name!
    'salary': [50000, 60000, 70000]
})

# Wrong: merging on non-existent 'id' in df2
merged = pd.merge(df1, df2, on='id')  # ERROR or empty result

# Duplicate keys without handling
df1 = pd.DataFrame({
    'id': [1, 1, 2],  # Duplicate id=1
    'name': ['Alice', 'Alice2', 'Bob']
})
df2 = pd.DataFrame({
    'id': [1, 1, 2],  # Duplicate id=1
    'salary': [50000, 55000, 60000]
})
merged = pd.merge(df1, df2, on='id')  # Creates cartesian product!
```

Code Example - CORRECT:

```
import pandas as pd

df1 = pd.DataFrame({
    'id': [1, 2, 3],
    'name': ['Alice', 'Bob', 'Charlie']
})

df2 = pd.DataFrame({
    'employee_id': [1, 2, 4],
    'salary': [50000, 60000, 70000]
})

# Use left_on and right_on for different names
merged = pd.merge(df1, df2,
                  left_on='id',
                  right_on='employee_id')  #

# Check for duplicates before merging
print("Duplicates in df1:", df1['id'].duplicated().sum())
print("Duplicates in df2:", df2['employee_id'].duplicated().sum())

# Remove duplicates if needed
df1 = df1.drop_duplicates(subset=['id'])  #
```

```
# Specify how to handle many-to-many
merged = pd.merge(df1, df2, on='id', how='left', validate='1:1')  #
# validate options: '1:1', '1:m', 'm:1', 'm:m'

# Check result
print(f"df1 rows: {len(df1)}, df2 rows: {len(df2)}, merged rows: {len(merged)}")

# Handle missing keys
merged = pd.merge(df1, df2, on='id', how='outer', indicator=True)  #
# indicator shows where each row came from
```

14.3 13.2 Concatenating DataFrames

14.3.1 Stacking DataFrames

```
import pandas as pd

df1 = pd.DataFrame({
    'name': ['Alice', 'Bob'],
    'age': [25, 30]
})

df2 = pd.DataFrame({
    'name': ['Charlie', 'David'],
    'age': [35, 40]
})

# Concatenate vertically (stack rows)
combined = pd.concat([df1, df2], ignore_index=True)

# Concatenate horizontally (side by side)
combined = pd.concat([df1, df2], axis=1)

# With keys to identify source
combined = pd.concat([df1, df2], keys=['first', 'second'])

# Only keep common columns
combined = pd.concat([df1, df2], join='inner')
```

14.3.2 Error Type 2: `ValueError: Shape mismatch in concat`

What Happened: Concatenating DataFrames with incompatible shapes.

Code Example - WRONG:

```
import pandas as pd

df1 = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})

df2 = pd.DataFrame({
    'A': [7, 8],  # Only 2 rows!
    'C': [9, 10]  # Different column!
})

# Horizontal concat with different row counts
combined = pd.concat([df1, df2], axis=1)  # Fills NaN but might be unexpected
```

Code Example - CORRECT:

```
import pandas as pd

df1 = pd.DataFrame({
    'A': [1, 2, 3],
    'B': [4, 5, 6]
})

df2 = pd.DataFrame({
    'A': [7, 8],
    'C': [9, 10]
})

# Vertical concat (rows) - works with different columns
combined = pd.concat([df1, df2], ignore_index=True)  #
# Fills NaN for missing columns

# Check shapes before concat
print(f"df1: {df1.shape}, df2: {df2.shape}")

# Reset index if needed
df1_reset = df1.reset_index(drop=True)
df2_reset = df2.reset_index(drop=True)
combined = pd.concat([df1_reset, df2_reset], ignore_index=True)  #
```

```
# Only keep matching columns for horizontal concat
common_cols = list(set(df1.columns) & set(df2.columns))
if common_cols:
    combined = pd.concat([df1[common_cols], df2[common_cols]], axis=1)  #

# Use merge instead if you have a key
combined = pd.merge(df1, df2, on='A', how='outer')  #
```

14.4 13.3 Pivot Tables

14.4.1 Reshaping Data

```
import pandas as pd

df = pd.DataFrame({
    'date': ['2024-01', '2024-01', '2024-02', '2024-02'],
    'city': ['NYC', 'LA', 'NYC', 'LA'],
    'sales': [100, 150, 200, 175]
})

# Create pivot table
pivot = df.pivot_table(
    values='sales',
    index='date',
    columns='city',
    aggfunc='sum'
)
#         LA   NYC
# 2024-01 150  100
# 2024-02 175  200

# Multiple aggregations
pivot = df.pivot_table(
    values='sales',
    index='date',
    columns='city',
    aggfunc=['sum', 'mean', 'count']
)

# Fill missing values
pivot = df.pivot_table(
    values='sales',
    index='date',
```

```
    columns='city',
    aggfunc='sum',
    fill_value=0
)

# Melt (reverse of pivot)
melted = pivot.reset_index().melt(
    id_vars=['date'],
    value_vars=['NYC', 'LA'],
    var_name='city',
    value_name='sales'
)
```

14.5 13.4 GroupBy Operations

14.5.1 Aggregating Data

```
import pandas as pd

df = pd.DataFrame({
    'city': ['NYC', 'NYC', 'LA', 'LA', 'Chicago'],
    'year': [2023, 2024, 2023, 2024, 2023],
    'sales': [100, 150, 200, 175, 90]
})

# Simple groupby
grouped = df.groupby('city')['sales'].sum()

# Multiple columns
grouped = df.groupby(['city', 'year'])['sales'].sum()

# Multiple aggregations
grouped = df.groupby('city').agg({
    'sales': ['sum', 'mean', 'count']
})

# Custom aggregation
grouped = df.groupby('city')['sales'].agg(
    total='sum',
    average='mean',
    maximum='max'
)
```

```
# Transform (keep original shape)
df['sales_pct'] = df.groupby('city')['sales'].transform(
    lambda x: x / x.sum() * 100
)

# Filter groups
filtered = df.groupby('city').filter(
    lambda x: x['sales'].sum() > 200
)

# Apply custom function
def custom_func(group):
    return group['sales'].max() - group['sales'].min()

result = df.groupby('city').apply(custom_func)
```

14.5.2 Error Type 3: KeyError in GroupBy

What Happened: Column doesn't exist in grouped result.

Code Example - WRONG:

```
import pandas as pd

df = pd.DataFrame({
    'city': ['NYC', 'LA'],
    'sales': [100, 200]
})

# Groupby returns Series not DataFrame
grouped = df.groupby('city')['sales'].sum()
# Now grouped is a Series

# Trying to access like DataFrame
result = grouped['city']  # ERROR! Series doesn't have 'city' column
```

Code Example - CORRECT:

```
import pandas as pd

df = pd.DataFrame({
    'city': ['NYC', 'LA'],
    'sales': [100, 200]
})
```

```
# Keep as DataFrame
grouped = df.groupby('city')[['sales']].sum()  #  Double brackets
# Or
grouped = df.groupby('city').sum()  #

# Reset index to access groupby column
grouped = df.groupby('city')['sales'].sum().reset_index()  #
print(grouped['city'])  #  Now works

# Access index directly
grouped = df.groupby('city')['sales'].sum()
cities = grouped.index  #  Get city names from index

# Use .agg() for DataFrame result
grouped = df.groupby('city').agg({'sales': 'sum'})  #
```

14.6 13.5 Working with Dates

14.6.1 DateTime Operations

```
import pandas as pd

# Create datetime column
df = pd.DataFrame({
    'date_str': ['2024-01-15', '2024-02-20', '2024-03-10']
})

# Convert to datetime
df['date'] = pd.to_datetime(df['date_str'])

# Extract components
df['year'] = df['date'].dt.year
df['month'] = df['date'].dt.month
df['day'] = df['date'].dt.day
df['day_of_week'] = df['date'].dt.day_name()
df['quarter'] = df['date'].dt.quarter

# Date arithmetic
df['next_week'] = df['date'] + pd.Timedelta(days=7)
df['last_month'] = df['date'] - pd.DateOffset(months=1)

# Date range
dates = pd.date_range('2024-01-01', '2024-12-31', freq='D')
```

```
# Resample time series
df = df.set_index('date')
monthly = df.resample('M').sum()

# Rolling windows
df['rolling_mean'] = df['sales'].rolling(window=7).mean()
```

14.7 13.6 Apply and Transform

14.7.1 Custom Functions

```
import pandas as pd

df = pd.DataFrame({
    'name': ['alice', 'bob', 'charlie'],
    'age': [25, 30, 35]
})

# Apply to column (Series)
df['name_upper'] = df['name'].apply(lambda x: x.upper())

# Apply to multiple columns
df['age_category'] = df['age'].apply(
    lambda x: 'young' if x < 30 else 'old'
)

# Apply to DataFrame (row-wise)
def categorize(row):
    if row['age'] < 30:
        return 'junior'
    return 'senior'

df['category'] = df.apply(categorize, axis=1)

# Apply element-wise (applymap) - deprecated, use .map()
df_numeric = df[['age']]
df_numeric = df_numeric.map(lambda x: x * 2)

# Vectorized operations (FASTER)
df['age_doubled'] = df['age'] * 2  #   Better than apply
df['is_adult'] = df['age'] >= 18   #   Vectorized
```

14.8 13.7 Memory Optimization

14.8.1 Handling Large DataFrames

```
import pandas as pd

# Read in chunks
chunk_size = 10000
chunks = []
for chunk in pd.read_csv('large_file.csv', chunksize=chunk_size):
    # Process chunk
    processed = chunk[chunk['age'] > 18]
    chunks.append(processed)

df = pd.concat(chunks, ignore_index=True)

# Optimize dtypes
df['age'] = df['age'].astype('int8')  # Instead of int64
df['category'] = df['category'].astype('category')

# Check memory usage
print(df.memory_usage(deep=True))
print(df.info(memory_usage='deep'))

# Use appropriate dtypes when reading
df = pd.read_csv('data.csv', dtype={
    'age': 'int8',
    'category': 'category'
})

# Select columns
df = pd.read_csv('data.csv', usecols=['name', 'age'])
```

14.9 13.8 Practice Problems

14.9.1 Problem 1: Merge Error

```
import pandas as pd
df1 = pd.DataFrame({'id': [1, 2], 'name': ['Alice', 'Bob']})
df2 = pd.DataFrame({'emp_id': [1, 2], 'salary': [50000, 60000]})
```

```
merged = pd.merge(df1, df2, on='id')
```

Click for Answer

Issue: Column 'id' doesn't exist in df2

Fix:

```
import pandas as pd
df1 = pd.DataFrame({'id': [1, 2], 'name': ['Alice', 'Bob']})
df2 = pd.DataFrame({'emp_id': [1, 2], 'salary': [50000, 60000]})

# Use left_on and right_on
merged = pd.merge(df1, df2, left_on='id', right_on='emp_id')  #
```

14.9.2 Problem 2: GroupBy KeyError

```
import pandas as pd
df = pd.DataFrame({'city': ['NYC', 'LA'], 'sales': [100, 200]})
grouped = df.groupby('city')['sales'].sum()
print(grouped['city'])
```

Click for Answer

Error: `KeyError: 'city'`

Why: GroupBy result is Series, 'city' is index

Fix:

```
import pandas as pd
df = pd.DataFrame({'city': ['NYC', 'LA'], 'sales': [100, 200]})
grouped = df.groupby('city')['sales'].sum()

# Reset index
grouped = grouped.reset_index()  #
print(grouped['city'])  #  Works now

# Or access index
cities = grouped.index  #
```

14.10 13.9 Key Takeaways

14.10.1 What You Learned

1. **Specify merge keys** - Use left_on/right_on
2. **Check for duplicates** - Before merging
3. **Use ignore_index** - When concatenating
4. **Reset index** - After groupby to access columns
5. **Optimize dtypes** - For memory efficiency
6. **Use vectorized ops** - Faster than apply
7. **Process in chunks** - For large files

14.10.2 Common Patterns

```
# Pattern 1: Safe merge
merged = pd.merge(df1, df2,
                  left_on='id1',
                  right_on='id2',
                  how='left')

# Pattern 2: Concat with reset
combined = pd.concat([df1, df2], ignore_index=True)

# Pattern 3: GroupBy with reset
result = df.groupby('col')['val'].sum().reset_index()

# Pattern 4: Vectorized operations
df['new'] = df['old'] * 2  # Better than apply
```

14.11 13.10 Moving Forward

You now understand advanced Pandas! In **Chapter 14**, we'll explore **NumPy** - numerical computing!

Chapter 15

Chapter 14: NumPy - Array Computing Errors

15.1 Introduction

Welcome to **NumPy** - the foundation of scientific computing in Python! NumPy provides powerful array operations and is the backbone of Pandas, SciPy, and most data science libraries.

Common errors: - **ValueError**: Shape mismatches - **IndexError**: Array index out of bounds - **TypeError**: Wrong data types - Broadcasting errors

Let's master NumPy!

15.2 14.1 Creating Arrays

15.2.1 Basic Array Creation

```
import numpy as np

# From list
arr = np.array([1, 2, 3, 4, 5])

# 2D array
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Zeros
zeros = np.zeros(5)          # [0. 0. 0. 0. 0.]
```

```
zeros_2d = np.zeros((3, 4))  # 3x4 matrix of zeros

# Ones
ones = np.ones(3)            # [1. 1. 1.]
ones_2d = np.ones((2, 3))    # 2x3 matrix of ones

# Range
arr = np.arange(0, 10, 2)    # [0 2 4 6 8]

# Linspace
arr = np.linspace(0, 1, 5)   # [0. 0.25 0.5 0.75 1.]

# Random
random = np.random.rand(5)    # 5 random numbers [0, 1)
random = np.random.randint(0, 10, 5)  # 5 random ints

# Identity matrix
identity = np.eye(3)          # 3x3 identity matrix

# Array info
print(arr.shape)    # Dimensions
print(arr.dtype)    # Data type
print(arr.size)     # Total elements
print(arr.ndim)     # Number of dimensions
```

15.2.2 Error Type 1: `ValueError: setting an array element with a sequence`

Error Message:

```
>>> import numpy as np
>>> arr = np.array([1, 2, [3, 4]])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: setting an array element with a sequence.
```

What Happened: Trying to create array with inconsistent dimensions.

Why It Happens: - Nested lists of different lengths - Mixed types in nested structure - Jagged arrays

Code Example - WRONG:

```
import numpy as np

# Inconsistent lengths
```

```
arr = np.array([[1, 2, 3], [4, 5]])  # ERROR! Different lengths

# Mixed nesting
arr = np.array([1, 2, [3, 4]])  # ERROR! Inconsistent depth

# Jagged array
data = [[1, 2], [3, 4, 5], [6]]
arr = np.array(data)  # ERROR or unexpected result
```

Code Example - CORRECT:

```
import numpy as np

# Consistent dimensions
arr = np.array([[1, 2, 3], [4, 5, 6]])  #   2x3 array

# Same nesting level
arr = np.array([1, 2, 3, 4])  #   1D array
arr = np.array([[1, 2], [3, 4]])  #   2D array

# For jagged arrays, use dtype=object
data = [[1, 2], [3, 4, 5], [6]]
arr = np.array(data, dtype=object)  #   Array of lists

# Or pad to same length
max_len = max(len(row) for row in data)
padded = [row + [0] * (max_len - len(row)) for row in data]
arr = np.array(padded)  #

# Check before creating
data = [[1, 2, 3], [4, 5, 6]]
lengths = [len(row) for row in data]
if len(set(lengths)) == 1:
    arr = np.array(data)  #   All same length
else:
    print("Inconsistent lengths")
```

15.3 14.2 Array Indexing and Slicing

15.3.1 Accessing Elements

```
import numpy as np
```

```
arr = np.array([10, 20, 30, 40, 50])

# Single element
print(arr[0])    # 10
print(arr[-1])   # 50

# Slicing
print(arr[1:4])  # [20 30 40]
print(arr[:3])   # [10 20 30]
print(arr[2:])   # [30 40 50]

# 2D array
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Single element
print(arr_2d[0, 0])     # 1
print(arr_2d[1, 2])     # 6

# Row
print(arr_2d[0])        # [1 2 3]
print(arr_2d[0, :])     # [1 2 3]

# Column
print(arr_2d[:, 0])     # [1 4 7]

# Subarray
print(arr_2d[0:2, 1:3]) # [[2 3]
                        #  [5 6]]

# Boolean indexing
arr = np.array([1, 2, 3, 4, 5])
mask = arr > 2
print(arr[mask])        # [3 4 5]
print(arr[arr > 2])     # [3 4 5]

# Fancy indexing
indices = [0, 2, 4]
print(arr[indices])     # [1 3 5]
```

15.3.2 Error Type 2: `IndexError: index out of bounds`

Error Message:

```
>>> import numpy as np
>>> arr = np.array([1, 2, 3])
>>> print(arr[5])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 5 is out of bounds for axis 0 with size 3
```

What Happened: Index exceeds array bounds.

Code Example - WRONG:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Index too large
value = arr[10]  # ERROR! Only indices 0-4 exist

# Wrong dimension count
arr_2d = np.array([[1, 2], [3, 4]])
value = arr_2d[0, 5]  # ERROR! Column 5 doesn't exist

# Negative index too large
value = arr[-10]  # ERROR! Only -1 to -5 valid
```

Code Example - CORRECT:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Check bounds
if index < len(arr):
    value = arr[index]  #
else:
    print("Index out of bounds")

# Use .shape for multi-dimensional
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
rows, cols = arr_2d.shape
if row < rows and col < cols:
    value = arr_2d[row, col]  #

# Use try/except
try:
    value = arr[index]
except IndexError:
```

```
    value = None  #

# Safe indexing with clip
safe_index = np.clip(index, 0, len(arr) - 1)
value = arr[safe_index]  #

# Use take with mode
value = arr.take(index, mode='clip')  #   Clips to valid range
value = arr.take(index, mode='wrap')  #   Wraps around
```

15.4 14.3 Array Operations

15.4.1 Mathematical Operations

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

# Element-wise operations
print(arr + 10)      # [11 12 13 14 15]
print(arr * 2)       # [2 4 6 8 10]
print(arr ** 2)      # [1 4 9 16 25]

# Array operations
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
print(arr1 + arr2)   # [5 7 9]
print(arr1 * arr2)   # [4 10 18]

# Aggregations
print(arr.sum())     # 15
print(arr.mean())    # 3.0
print(arr.std())     # Standard deviation
print(arr.min())     # 1
print(arr.max())     # 5

# Along axis
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr_2d.sum(axis=0))  # [5 7 9] column sums
print(arr_2d.sum(axis=1))  # [6 15] row sums

# Universal functions
print(np.sqrt(arr))    # Square root
```

```
print(np.exp(arr))     # Exponential
print(np.log(arr))     # Natural log
print(np.sin(arr))     # Sine
```

15.4.2 Error Type 3: `ValueError: operands could not be broadcast together`

Error Message:

```
>>> import numpy as np
>>> arr1 = np.array([1, 2, 3])
>>> arr2 = np.array([1, 2])
>>> result = arr1 + arr2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

What Happened: Trying to operate on arrays with incompatible shapes.

Why It Happens: - Different array sizes - Incompatible dimensions - Wrong broadcasting

Code Example - WRONG:

```
import numpy as np

# Different lengths
arr1 = np.array([1, 2, 3])
arr2 = np.array([1, 2])
result = arr1 + arr2  # ERROR! Shapes (3,) and (2,)

# Incompatible 2D shapes
arr1 = np.array([[1, 2, 3], [4, 5, 6]])  # 2x3
arr2 = np.array([[1, 2], [3, 4]])        # 2x2
result = arr1 + arr2  # ERROR! Incompatible

# Wrong dimension operations
arr1 = np.array([[1, 2], [3, 4]])  # 2x2
arr2 = np.array([1, 2, 3])         # 3 elements
result = arr1 + arr2  # ERROR! Can't broadcast
```

Code Example - CORRECT:

```
import numpy as np

# Match array sizes
arr1 = np.array([1, 2, 3])
```

```
arr2 = np.array([1, 2, 3])  #   Same size
result = arr1 + arr2

# Broadcasting with scalar
arr = np.array([1, 2, 3])
result = arr + 10  #   Scalar broadcasts to all elements

# Broadcasting with compatible shapes
arr1 = np.array([[1, 2, 3], [4, 5, 6]])  # 2x3
arr2 = np.array([10, 20, 30])            # 3 elements
result = arr1 + arr2  #   Broadcasts arr2 to each row
# [[11 22 33]
#  [14 25 36]]

# Reshape for broadcasting
arr1 = np.array([[1, 2], [3, 4]])  # 2x2
arr2 = np.array([10, 20])          # 2 elements
result = arr1 + arr2  #   Broadcasts across columns

# Make compatible with reshape
arr1 = np.array([[1, 2], [3, 4]])    # 2x2
arr2 = np.array([10, 20])            # 2 elements
arr2_reshaped = arr2.reshape(2, 1)   # 2x1
result = arr1 + arr2_reshaped  #
# [[11 12]
#  [23 24]]

# Check shapes before operation
if arr1.shape == arr2.shape:
    result = arr1 + arr2  #
else:
    print(f"Incompatible: {arr1.shape} vs {arr2.shape}")

# Pad shorter array
arr1 = np.array([1, 2, 3])
arr2 = np.array([1, 2])
arr2_padded = np.pad(arr2, (0, len(arr1) - len(arr2)))  #
result = arr1 + arr2_padded
```

Broadcasting Rules:

```
# Arrays broadcast when:
# 1. They have same shape
# 2. One dimension is 1
# 3. One array has fewer dimensions
```

```
# Examples:
# (3, 4) + (3, 4)      Same shape
# (3, 4) + (4,)        Broadcasts to (3, 4)
# (3, 4) + (3, 1)      Broadcasts to (3, 4)
# (3, 4) + (1, 4)      Broadcasts to (3, 4)
# (3, 4) + (3, 5)      Incompatible
```

15.5 14.4 Reshaping Arrays

15.5.1 Changing Array Shape

```python
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])

# Reshape
arr_2d = arr.reshape(2, 3)  # 2x3
# [[1 2 3]
#  [4 5 6]]

# Flatten
arr_flat = arr_2d.flatten()  # [1 2 3 4 5 6]
arr_flat = arr_2d.ravel()    # Same but returns view

# Transpose
arr_t = arr_2d.T
# [[1 4]
#  [2 5]
#  [3 6]]

# Add dimension
arr_3d = arr[np.newaxis, :]    # (1, 6)
arr_3d = arr[:, np.newaxis]    # (6, 1)

# Squeeze (remove single dimensions)
arr = np.array([[[1, 2, 3]]])  # (1, 1, 3)
arr_squeezed = arr.squeeze()   # (3,)
```

15.5.2 Error Type 4: `ValueError: cannot reshape array`

Error Message:

```
>>> import numpy as np
>>> arr = np.array([1, 2, 3, 4, 5])
>>> arr.reshape(2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot reshape array of size 5 into shape (2,3)
```

What Happened: New shape doesn't match total number of elements.

Code Example - WRONG:

```
import numpy as np

# Wrong total elements
arr = np.array([1, 2, 3, 4, 5])  # 5 elements
arr_2d = arr.reshape(2, 3)  # ERROR! 2*3=6   5

# Incompatible shape
arr = np.array([1, 2, 3, 4])  # 4 elements
arr_2d = arr.reshape(3, 2)  # ERROR! 3*2=6   4
```

Code Example - CORRECT:

```
import numpy as np

# Matching total elements
arr = np.array([1, 2, 3, 4, 5, 6])  # 6 elements
arr_2d = arr.reshape(2, 3)  #   2*3=6
arr_2d = arr.reshape(3, 2)  #   3*2=6

# Use -1 to infer dimension
arr = np.array([1, 2, 3, 4, 5, 6])
arr_2d = arr.reshape(2, -1)  #   Auto-calculates 3
arr_2d = arr.reshape(-1, 3)  #   Auto-calculates 2

# Check if reshapeable
if arr.size % 3 == 0:
    arr_2d = arr.reshape(-1, 3)  #
else:
    print("Cannot reshape to 3 columns")

# Pad to make reshapeable
arr = np.array([1, 2, 3, 4, 5])  # 5 elements
target_size = 6
arr_padded = np.pad(arr, (0, target_size - arr.size))  #
arr_2d = arr_padded.reshape(2, 3)
```

```
# Use resize (changes size)
arr = np.array([1, 2, 3, 4, 5])
arr.resize((2, 3))  #   Pads with zeros
# [[1 2 3]
#  [4 5 0]]
```

15.6 14.5 Common Patterns

15.6.1 Useful Operations

```
import numpy as np

# Stacking arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
stacked_v = np.vstack([arr1, arr2])  # Vertical
# [[1 2 3]
#  [4 5 6]]
stacked_h = np.hstack([arr1, arr2])  # Horizontal
# [1 2 3 4 5 6]

# Where (conditional selection)
arr = np.array([1, 2, 3, 4, 5])
result = np.where(arr > 2, arr, 0)  # [0 0 3 4 5]

# Unique values
arr = np.array([1, 2, 2, 3, 3, 3])
unique = np.unique(arr)  # [1 2 3]

# Sorting
arr = np.array([3, 1, 4, 1, 5])
sorted_arr = np.sort(arr)  # [1 1 3 4 5]
indices = np.argsort(arr)  # [1 3 0 2 4]

# Finding elements
arr = np.array([1, 2, 3, 4, 5])
indices = np.where(arr > 3)  # (array([3, 4]),)
```

15.7 14.6 Practice Problems

15.7.1 Problem 1: Shape Mismatch

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([1, 2])
result = arr1 + arr2
```

Click for Answer

Error: `ValueError: operands could not be broadcast together`

Fix:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([1, 2, 3])  #  Match sizes
result = arr1 + arr2
```

15.7.2 Problem 2: Reshape Error

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
arr_2d = arr.reshape(2, 3)
```

Click for Answer

Error: `ValueError: cannot reshape array of size 5 into shape (2,3)`

Fix:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])  #  6 elements
arr_2d = arr.reshape(2, 3)  #  2*3=6

# Or use -1
arr = np.array([1, 2, 3, 4, 5, 6])
arr_2d = arr.reshape(2, -1)  #  Auto-calculates 3
```

15.8 14.7 Key Takeaways

15.8.1 What You Learned

1. **Match array dimensions** - For operations

2. **Check shapes** - Before reshaping
3. **Use broadcasting** - For efficient operations
4. **Validate indices** - Before accessing
5. **Consistent nesting** - For array creation
6. **Use -1 in reshape** - To infer dimension
7. **Vectorize operations** - Avoid loops

15.9 14.8 Moving Forward

You now understand NumPy! In **Chapter 15**, we'll explore **Matplotlib** - data visualization!

Chapter 16

Chapter 15: Matplotlib - Data Visualization Errors

16.1 Introduction

Welcome to **Matplotlib** - Python's primary plotting library! Matplotlib creates publication-quality figures and is the foundation for many other visualization libraries.

Common errors: - **ValueError**: Invalid data shapes - **TypeError**: Wrong data types for plots - **AttributeError**: Wrong method or property - Figure/axis confusion

Let's master Matplotlib!

16.2 15.1 Basic Plotting

16.2.1 Creating Simple Plots

```
import matplotlib.pyplot as plt
import numpy as np

# Line plot
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]
plt.plot(x, y)
plt.xlabel('X axis')
plt.ylabel('Y axis')
```

```
plt.title('Simple Line Plot')
plt.show()

# Multiple lines
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)
plt.plot(x, y1, label='sin(x)')
plt.plot(x, y2, label='cos(x)')
plt.legend()
plt.show()

# Scatter plot
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]
plt.scatter(x, y)
plt.show()

# Bar plot
categories = ['A', 'B', 'C', 'D']
values = [4, 7, 2, 9]
plt.bar(categories, values)
plt.show()

# Histogram
data = np.random.randn(1000)
plt.hist(data, bins=30)
plt.show()
```

16.2.2 Error Type 1: `ValueError: x and y must have same first dimension`

Error Message:

```
>>> import matplotlib.pyplot as plt
>>> x = [1, 2, 3]
>>> y = [1, 2]
>>> plt.plot(x, y)
Traceback (most recent call last):
  ...
ValueError: x and y must have same first dimension, but have shapes (3,) and (2,)
```

What Happened: X and Y arrays have different lengths.

Why It Happens: - Different array sizes - Data mismatch - Missing values

Code Example - WRONG:

```
import matplotlib.pyplot as plt

# Different lengths
x = [1, 2, 3, 4, 5]
y = [2, 4, 6]  # Only 3 values
plt.plot(x, y)  # ERROR! 5 vs 3

# Accidental truncation
x = range(10)
y = [i**2 for i in range(5)]  # Only 5 values
plt.plot(x, y)  # ERROR!
```

Code Example - CORRECT:

```
import matplotlib.pyplot as plt
import numpy as np

# Match lengths
x = [1, 2, 3, 4, 5]
y = [2, 4, 6, 8, 10]  #   5 values
plt.plot(x, y)
plt.show()

# Check lengths before plotting
if len(x) == len(y):
    plt.plot(x, y)  #
else:
    print(f"Length mismatch: {len(x)} vs {len(y)}")

# Generate matching arrays
x = np.linspace(0, 10, 100)
y = x ** 2  #   Automatically same length
plt.plot(x, y)

# Truncate to shorter length
min_len = min(len(x), len(y))
plt.plot(x[:min_len], y[:min_len])  #

# Fill missing values
x = [1, 2, 3, 4, 5]
y = [2, 4, 6]
while len(y) < len(x):
    y.append(0)  #   Pad with zeros
plt.plot(x, y)
```

16.3 15.2 Subplots

16.3.1 Multiple Plots

```
import matplotlib.pyplot as plt
import numpy as np

# Create figure with subplots
fig, axes = plt.subplots(2, 2, figsize=(10, 8))

x = np.linspace(0, 10, 100)

# Access subplots
axes[0, 0].plot(x, np.sin(x))
axes[0, 0].set_title('Sine')

axes[0, 1].plot(x, np.cos(x))
axes[0, 1].set_title('Cosine')

axes[1, 0].plot(x, x**2)
axes[1, 0].set_title('Square')

axes[1, 1].plot(x, np.exp(x/10))
axes[1, 1].set_title('Exponential')

plt.tight_layout()
plt.show()

# Single row
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
for i, ax in enumerate(axes):
    ax.plot(x, x**i)
    ax.set_title(f'x^{i}')
plt.show()
```

16.3.2 Error Type 2: `AttributeError: 'numpy.ndarray' object has no attribute 'plot'`

Error Message:

```
>>> fig, axes = plt.subplots(2, 2)
>>> axes.plot([1, 2, 3])
Traceback (most recent call last):
  ...
AttributeError: 'numpy.ndarray' object has no attribute 'plot'
```

What Happened: Calling plot() on axes array instead of individual axis.

Why It Happens: - Confusing axes array with single axis - Wrong indexing - Not understanding subplots return type

Code Example - WRONG:

```
import matplotlib.pyplot as plt

# Multiple subplots - axes is array
fig, axes = plt.subplots(2, 2)
axes.plot([1, 2, 3])  # ERROR! axes is array

# Wrong method on figure
fig = plt.figure()
fig.plot([1, 2, 3])  # ERROR! Use ax, not fig
```

Code Example - CORRECT:

```
import matplotlib.pyplot as plt
import numpy as np

# Single subplot - axes is single axis
fig, ax = plt.subplots()
ax.plot([1, 2, 3])  #   ax is single axis
plt.show()

# Multiple subplots - index into array
fig, axes = plt.subplots(2, 2)
axes[0, 0].plot([1, 2, 3])  #   Index specific axis
axes[0, 1].plot([1, 4, 9])  #
plt.show()

# Flatten for easy iteration
fig, axes = plt.subplots(2, 2)
axes_flat = axes.flatten()
for i, ax in enumerate(axes_flat):
    ax.plot([1, 2, 3])  #
plt.show()

# Use plt.subplot (different approach)
plt.subplot(2, 2, 1)
```

```
plt.plot([1, 2, 3])  #
plt.subplot(2, 2, 2)
plt.plot([1, 4, 9])  #
plt.show()

# Check type
fig, axes = plt.subplots(2, 2)
if isinstance(axes, np.ndarray):
    for ax in axes.flat:
        ax.plot([1, 2, 3])  #
else:
    axes.plot([1, 2, 3])  #
```

16.4 15.3 Customizing Plots

16.4.1 Styling and Formatting

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)
y = np.sin(x)

# Line style and color
plt.plot(x, y,
         color='red',       # or 'r', '#FF0000', (1, 0, 0)
         linestyle='--',    # or ':', '-.', '-'
         linewidth=2,
         marker='o',
         markersize=5,
         label='sin(x)')

# Grid
plt.grid(True, alpha=0.3)

# Limits
plt.xlim(0, 10)
plt.ylim(-1.5, 1.5)

# Labels
plt.xlabel('Time', fontsize=12)
plt.ylabel('Value', fontsize=12)
```

```
plt.title('Sine Wave', fontsize=14, fontweight='bold')

# Legend
plt.legend(loc='upper right')

# Save figure
plt.savefig('plot.png', dpi=300, bbox_inches='tight')

plt.show()
```

16.5 15.4 Different Plot Types

16.5.1 Common Visualizations

```
import matplotlib.pyplot as plt
import numpy as np

# Scatter with colors
x = np.random.rand(50)
y = np.random.rand(50)
colors = np.random.rand(50)
sizes = 1000 * np.random.rand(50)
plt.scatter(x, y, c=colors, s=sizes, alpha=0.5, cmap='viridis')
plt.colorbar()
plt.show()

# Bar plot
categories = ['A', 'B', 'C', 'D', 'E']
values = [23, 45, 56, 78, 32]
plt.bar(categories, values, color='steelblue')
plt.xticks(rotation=45)
plt.show()

# Horizontal bar
plt.barh(categories, values, color='coral')
plt.show()

# Histogram
data = np.random.randn(1000)
plt.hist(data, bins=30, color='skyblue', edgecolor='black')
plt.xlabel('Value')
plt.ylabel('Frequency')
```

```
plt.show()

# Box plot
data = [np.random.normal(0, std, 100) for std in range(1, 4)]
plt.boxplot(data, labels=['Group 1', 'Group 2', 'Group 3'])
plt.show()

# Pie chart
sizes = [30, 25, 20, 25]
labels = ['A', 'B', 'C', 'D']
plt.pie(sizes, labels=labels, autopct='%1.1f%%', startangle=90)
plt.axis('equal')
plt.show()

# Heatmap
data = np.random.rand(10, 10)
plt.imshow(data, cmap='hot', interpolation='nearest')
plt.colorbar()
plt.show()
```

16.5.2 Error Type 3: `TypeError: Image data of dtype object cannot be converted to float`

What Happened: Wrong data type for plot.

Code Example - WRONG:

```
import matplotlib.pyplot as plt

# String data for numerical plot
data = ['a', 'b', 'c']
plt.plot(data)  # ERROR! Can't plot strings

# Mixed types
x = [1, 2, '3', 4]
y = [1, 4, 9, 16]
plt.plot(x, y)  # ERROR! Mixed types
```

Code Example - CORRECT:

```
import matplotlib.pyplot as plt
import numpy as np

# Convert to numbers
data = ['1', '2', '3', '4']
```

```
data_numeric = [float(x) for x in data]  #
plt.plot(data_numeric)

# Use categorical plot for strings
categories = ['A', 'B', 'C', 'D']
values = [1, 3, 2, 4]
plt.bar(categories, values)  #

# Handle missing/invalid data
data = [1, 2, None, 4, 5]
data_clean = [x for x in data if x is not None]  #
plt.plot(data_clean)

# Use pandas for automatic handling
import pandas as pd
df = pd.DataFrame({'x': [1, 2, 3], 'y': [1, 4, 9]})
df.plot(x='x', y='y')  #
```

16.6 15.5 Common Patterns

16.6.1 Useful Techniques

```
import matplotlib.pyplot as plt
import numpy as np

# Multiple y-axes
fig, ax1 = plt.subplots()
ax2 = ax1.twinx()

x = np.linspace(0, 10, 100)
ax1.plot(x, np.sin(x), 'b-')
ax2.plot(x, x**2, 'r-')
ax1.set_ylabel('sin(x)', color='b')
ax2.set_ylabel('x^2', color='r')

# Annotations
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
plt.annotate('Peak', xy=(4, 16), xytext=(3, 12),
             arrowprops=dict(arrowstyle='->'))

# Fill between
x = np.linspace(0, 10, 100)
```

```
y1 = np.sin(x)
y2 = np.cos(x)
plt.plot(x, y1)
plt.plot(x, y2)
plt.fill_between(x, y1, y2, alpha=0.3)

# Log scale
plt.plot(x, np.exp(x))
plt.yscale('log')

# Style sheets
plt.style.use('seaborn')  # or 'ggplot', 'dark_background'
```

16.7 15.6 Practice Problems

16.7.1 Problem 1: Length Mismatch

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [1, 4, 9]
plt.plot(x, y)
```

Click for Answer

Error: `ValueError: x and y must have same first dimension`

Fix:

```
import matplotlib.pyplot as plt
x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]  #  Match length
plt.plot(x, y)
plt.show()
```

16.7.2 Problem 2: Wrong Axes Access

```
import matplotlib.pyplot as plt
fig, axes = plt.subplots(2, 2)
axes.plot([1, 2, 3])
```

Click for Answer

Error: `AttributeError: 'numpy.ndarray' object has no attribute 'plot'`

Fix:

```
import matplotlib.pyplot as plt
fig, axes = plt.subplots(2, 2)
axes[0, 0].plot([1, 2, 3])  #  Index specific axis
plt.show()
```

16.8 15.7 Key Takeaways

16.8.1 What You Learned

1. **Match array lengths** - X and Y must be same size
2. **Index subplot axes** - axes[i, j] for multiple plots
3. **Use ax methods** - Not plt when using subplots
4. **Check data types** - Convert strings to numbers
5. **Use tight_layout()** - Prevent overlapping
6. **Save before show()** - Or figure won't save
7. **Close figures** - plt.close() to free memory

16.8.2 Common Patterns

```
# Pattern 1: Basic plot
plt.plot(x, y)
plt.xlabel('X')
plt.ylabel('Y')
plt.title('Title')
plt.show()

# Pattern 2: Subplots
fig, axes = plt.subplots(2, 2)
axes[0, 0].plot(x, y)
plt.tight_layout()
plt.show()

# Pattern 3: Save figure
plt.plot(x, y)
plt.savefig('plot.png', dpi=300)
plt.show()
```

16.8.3 Error Summary

Error	Cause	Prevention
`ValueError` (dimension)	X and Y different lengths	Match array sizes
`AttributeError`	Wrong axes access	Index axes array properly
`TypeError`	Wrong data type	Convert to numeric

16.9 15.8 Congratulations - Part II Complete!

16.9.1 You Completed Part II: Libraries and Data!

You've mastered: - Regular Expressions (Chapter 11) - Pandas Basics (Chapter 12) - Pandas Advanced (Chapter 13) - NumPy (Chapter 14) - Matplotlib (Chapter 15)

Total Progress: 15/20 chapters (75%) complete!

16.10 15.9 Moving Forward

What's Next: Part III - Advanced Topics (Chapters 16-20) - Chapter 16: Object-Oriented Programming - Chapter 17: Modules and Imports - Chapter 18: Exception Handling - Chapter 19: Debugging Techniques - Chapter 20: Testing and Code Quality

Chapter 17

Chapter 16: Object-Oriented Programming - Class and Object Errors

17.1 Introduction

Welcome to **Object-Oriented Programming (OOP)** - organizing code into classes and objects. OOP is fundamental to Python and most modern programming. Understanding OOP errors is essential for building robust applications.

Common errors: - **AttributeError**: Missing attributes or methods - **TypeError**: Wrong initialization or method calls - **NameError**: Class/method not defined - Inheritance issues

Let's master OOP!

17.2 16.1 Classes and Objects

17.2.1 Basic Class Definition

```
# Define a class
class Dog:
    def __init__(self, name, age):
        self.name = name  # Instance attribute
```

```
        self.age = age

    def bark(self):
        return f"{self.name} says woof!"

    def get_age(self):
        return self.age

# Create objects (instances)
dog1 = Dog("Buddy", 5)
dog2 = Dog("Max", 3)

# Access attributes
print(dog1.name)  # "Buddy"
print(dog2.age)   # 3

# Call methods
print(dog1.bark())  # "Buddy says woof!"

# Class attributes (shared by all instances)
class Cat:
    species = "Felis catus"  # Class attribute

    def __init__(self, name):
        self.name = name  # Instance attribute

cat1 = Cat("Whiskers")
cat2 = Cat("Mittens")
print(cat1.species)  # "Felis catus"
print(cat2.species)  # "Felis catus"
```

17.2.2 Error Type 1: **TypeError: __init__() missing required positional argument**

Error Message:

```
>>> class Dog:
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
>>> dog = Dog("Buddy")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: __init__() missing 1 required positional argument: 'age'
```

What Happened: Creating object without providing all required parameters.

Why It Happens: - Missing arguments in initialization - Wrong number of arguments - Forgetting self parameter

Code Example - WRONG:

```
class Person:
    def __init__(self, name, age, city):
        self.name = name
        self.age = age
        self.city = city

# Missing arguments
person = Person("Alice")  # ERROR! Missing age and city

# Too many arguments
person = Person("Alice", 25, "NYC", "USA")  # ERROR! Too many

# Wrong argument order
person = Person(25, "Alice", "NYC")  # Wrong but no error (logic issue)

# Forgetting to pass arguments
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

car = Car()  # ERROR! Missing make and model
```

Code Example - CORRECT:

```
class Person:
    def __init__(self, name, age, city):
        self.name = name
        self.age = age
        self.city = city

# Provide all arguments
person = Person("Alice", 25, "NYC")  #

# Use default parameters
class Person:
    def __init__(self, name, age=0, city="Unknown"):
        self.name = name
        self.age = age
        self.city = city
```

```
person = Person("Alice")  #   Uses defaults
person = Person("Bob", 30)  #   Partial defaults
person = Person("Charlie", 35, "LA")  #   All specified

# Use keyword arguments
person = Person(name="Alice", age=25, city="NYC")  #   Clear

# Flexible initialization with *args, **kwargs
class FlexiblePerson:
    def __init__(self, name, **kwargs):
        self.name = name
        self.age = kwargs.get('age', 0)
        self.city = kwargs.get('city', 'Unknown')

person = FlexiblePerson("Alice")  #
person = FlexiblePerson("Bob", age=30)  #
person = FlexiblePerson("Charlie", age=35, city="LA")  #

# Validate arguments
class Person:
    def __init__(self, name, age):
        if not name:
            raise ValueError("Name cannot be empty")
        if age < 0:
            raise ValueError("Age cannot be negative")
        self.name = name
        self.age = age  #   Validated
```

17.3 16.2 Instance vs Class Attributes

17.3.1 Understanding Attribute Scope

```
class Counter:
    # Class attribute (shared)
    total_count = 0

    def __init__(self, name):
        # Instance attribute (unique to each object)
        self.name = name
        self.count = 0
        Counter.total_count += 1
```

```
    def increment(self):
        self.count += 1

# Create instances
c1 = Counter("Counter1")
c2 = Counter("Counter2")

print(Counter.total_count)  # 2 (class attribute)
print(c1.count)  # 0 (instance attribute)
print(c2.count)  # 0 (instance attribute)

c1.increment()
print(c1.count)  # 1
print(c2.count)  # 0 (unchanged)
```

17.3.2 Error Type 2: `AttributeError: 'ClassName' object has no attribute 'attribute_name'`

Error Message:

```
>>> class Dog:
...     def __init__(self, name):
...         self.name = name
>>> dog = Dog("Buddy")
>>> print(dog.age)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Dog' object has no attribute 'age'
```

What Happened: Accessing attribute that doesn't exist.

Why It Happens: - Attribute not defined in **init** - Typo in attribute name - Conditional attribute creation - Accessing before assignment

Code Example - WRONG:

```
class Person:
    def __init__(self, name):
        self.name = name
        # age not defined!

person = Person("Alice")
print(person.age)  # ERROR! age doesn't exist

# Typo
class Car:
```

```
    def __init__(self, make):
        self.make = make

car = Car("Toyota")
print(car.maker)  # ERROR! Typo: maker vs make

# Conditional creation
class Student:
    def __init__(self, name, graduated=False):
        self.name = name
        if graduated:
            self.graduation_year = 2024

student = Student("Alice")
print(student.graduation_year)  # ERROR! Not created

# Accessing class attribute on instance incorrectly
class MyClass:
    class_var = "class"

obj = MyClass()
print(MyClass.instance_var)  # ERROR! Doesn't exist
```

Code Example - CORRECT:

```
class Person:
    def __init__(self, name, age=None):
        self.name = name
        self.age = age  #   Always defined

person = Person("Alice")
print(person.age)  # None (but defined)

# Check attribute exists
if hasattr(person, 'age'):
    print(person.age)  #
else:
    print("No age attribute")

# Use getattr with default
age = getattr(person, 'age', 0)  #   Returns 0 if not exists

# Always initialize attributes
class Student:
    def __init__(self, name, graduated=False):
        self.name = name
```

```
        self.graduation_year = 2024 if graduated else None  #

student = Student("Alice")
print(student.graduation_year)  # None (but defined)

# Use property with getter
class Person:
    def __init__(self, name):
        self.name = name
        self._age = None

    @property
    def age(self):
        return self._age if self._age is not None else 0  #

    @age.setter
    def age(self, value):
        if value < 0:
            raise ValueError("Age cannot be negative")
        self._age = value

person = Person("Alice")
print(person.age)  # 0 (property returns default)

# Try/except for optional attributes
try:
    print(person.optional_attr)
except AttributeError:
    print("Attribute doesn't exist")  #
```

17.4 16.3 Methods

17.4.1 Instance, Class, and Static Methods

```
class MyClass:
    class_var = "class variable"

    def __init__(self, value):
        self.value = value

    # Instance method (accesses self)
    def instance_method(self):
        return f"Instance: {self.value}"
```

```
    # Class method (accesses class, not instance)
    @classmethod
    def class_method(cls):
        return f"Class: {cls.class_var}"

    # Static method (no access to class or instance)
    @staticmethod
    def static_method(x, y):
        return x + y

obj = MyClass(10)

# Call methods
print(obj.instance_method())      # "Instance: 10"
print(MyClass.class_method())     # "Class: class variable"
print(MyClass.static_method(5, 3)) # 8
```

17.4.2 Error Type 3: `TypeError: method() takes 1 positional argument but 2 were given`

Error Message:

```
>>> class Dog:
...     def bark():
...         return "Woof!"
>>> dog = Dog()
>>> dog.bark()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: bark() takes 0 positional arguments but 1 was given
```

What Happened: Forgetting `self` parameter in method definition.

Why It Happens: - Missing self parameter - Wrong method type - Calling method incorrectly

Code Example - WRONG:

```
class Dog:
    def __init__(self, name):
        self.name = name

    # Missing self
    def bark():  # ERROR! Missing self
        return "Woof!"
```

```python
dog = Dog("Buddy")
dog.bark()  # ERROR! Python passes self automatically

# Wrong static method
class Calculator:
    @staticmethod
    def add(self, x, y):  # ERROR! Static methods don't use self
        return x + y

# Calling instance method on class
class Cat:
    def meow(self):
        return "Meow!"

Cat.meow()  # ERROR! Need instance
```

Code Example - CORRECT:

```python
class Dog:
    def __init__(self, name):
        self.name = name

    # Include self
    def bark(self):  #   self parameter
        return f"{self.name} says Woof!"

dog = Dog("Buddy")
print(dog.bark())  #

# Static method (no self)
class Calculator:
    @staticmethod
    def add(x, y):  #   No self
        return x + y

print(Calculator.add(5, 3))  #

# Call instance method on instance
class Cat:
    def meow(self):
        return "Meow!"

cat = Cat()
print(cat.meow())  #
```

```
# Or pass instance explicitly
print(Cat.meow(cat))  #   Explicit self

# Class method uses cls
class Counter:
    count = 0

    @classmethod
    def increment(cls):  #   cls parameter
        cls.count += 1

Counter.increment()  #
```

17.5 16.4 Inheritance

17.5.1 Extending Classes

```
# Base class
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return "Some sound"

# Derived class
class Dog(Animal):
    def speak(self):  # Override
        return "Woof!"

class Cat(Animal):
    def speak(self):  # Override
        return "Meow!"

# Use inheritance
dog = Dog("Buddy")
print(dog.name)    # "Buddy" (from Animal)
print(dog.speak()) # "Woof!" (from Dog)

# Call parent method
class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  # Call parent __init__
```

```
        self.breed = breed

    def speak(self):
        parent_sound = super().speak()
        return f"{parent_sound} and Woof!"

# Multiple inheritance
class Flyable:
    def fly(self):
        return "Flying"

class Bird(Animal, Flyable):
    def speak(self):
        return "Tweet!"

bird = Bird("Tweety")
print(bird.speak())  # "Tweet!"
print(bird.fly())    # "Flying"
```

17.5.2 Error Type 4: `TypeError: super() argument 1 must be type, not classobj`

What Happened: Issues with super() or inheritance.

Code Example - WRONG:

```
# Forgetting to call parent __init__
class Animal:
    def __init__(self, name):
        self.name = name

class Dog(Animal):
    def __init__(self, breed):
        # ERROR! Not calling parent __init__
        self.breed = breed

dog = Dog("Labrador")
print(dog.name)  # AttributeError! name not set

# Wrong super() syntax (Python 2 style)
class Dog(Animal):
    def __init__(self, name, breed):
        super(Dog, self).__init__(name)  # Works but verbose
        self.breed = breed
```

```
# Circular inheritance
class A(B):
    pass

class B(A):  # ERROR! Circular
    pass
```

Code Example - CORRECT:

```
# Call parent __init__
class Animal:
    def __init__(self, name):
        self.name = name

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name)  #   Python 3 syntax
        self.breed = breed

dog = Dog("Buddy", "Labrador")
print(dog.name)   #   "Buddy"
print(dog.breed)  #   "Labrador"

# Check inheritance
print(isinstance(dog, Dog))     # True
print(isinstance(dog, Animal))  # True
print(issubclass(Dog, Animal))  # True

# Multiple inheritance - Method Resolution Order (MRO)
class A:
    def method(self):
        return "A"

class B(A):
    def method(self):
        return "B"

class C(A):
    def method(self):
        return "C"

class D(B, C):  #   B before C
    pass

d = D()
```

```
print(d.method())  # "B" (follows MRO)
print(D.__mro__)   # Shows method resolution order

# Use super() in multiple inheritance
class B(A):
    def method(self):
        result = super().method()
        return f"B > {result}"

class C(A):
    def method(self):
        result = super().method()
        return f"C > {result}"

class D(B, C):
    def method(self):
        result = super().method()
        return f"D > {result}"  #   Calls through MRO
```

17.6 16.5 Special Methods (Dunder Methods)

17.6.1 Magic Methods

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # String representation
    def __str__(self):
        return f"Point({self.x}, {self.y})"

    def __repr__(self):
        return f"Point(x={self.x}, y={self.y})"

    # Arithmetic operations
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Point(self.x - other.x, self.y - other.y)
```

```
    # Comparison
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    # Length/bool
    def __len__(self):
        return int((self.x**2 + self.y**2)**0.5)

    def __bool__(self):
        return self.x != 0 or self.y != 0

# Usage
p1 = Point(1, 2)
p2 = Point(3, 4)

print(p1)           # "Point(1, 2)" (uses __str__)
print(p1 + p2)      # "Point(4, 6)" (uses __add__)
print(p1 == p2)     # False (uses __eq__)
print(len(p1))      # Length
print(bool(p1))     # True

# Container methods
class MyList:
    def __init__(self):
        self.items = []

    def __getitem__(self, index):
        return self.items[index]

    def __setitem__(self, index, value):
        self.items[index] = value

    def __len__(self):
        return len(self.items)

    def __contains__(self, item):
        return item in self.items

my_list = MyList()
my_list.items = [1, 2, 3]
print(my_list[0])      # 1 (uses __getitem__)
print(2 in my_list)    # True (uses __contains__)
```

17.7 16.6 Properties

```
class Temperature:
    def __init__(self, celsius):
        self._celsius = celsius

    @property
    def celsius(self):
        """Get temperature in Celsius"""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        """Set temperature in Celsius"""
        if value < -273.15:
            raise ValueError("Temperature below absolute zero")
        self._celsius = value

    @property
    def fahrenheit(self):
        """Get temperature in Fahrenheit"""
        return self._celsius * 9/5 + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        """Set temperature in Fahrenheit"""
        self.celsius = (value - 32) * 5/9

# Usage
temp = Temperature(25)
print(temp.celsius)     # 25
print(temp.fahrenheit)  # 77.0

temp.celsius = 30
print(temp.fahrenheit)  # 86.0

temp.fahrenheit = 100
print(temp.celsius)     # 37.77...
```

17.8 16.7 Practice Problems

17.8.1 Problem 1: Missing init Argument

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

person = Person("Alice")
```

Click for Answer

Error: `TypeError: __init__() missing 1 required positional argument: 'age'`

Fix:

```
class Person:
    def __init__(self, name, age=0):  #   Default value
        self.name = name
        self.age = age

person = Person("Alice")  #
```

17.8.2 Problem 2: Missing self

```
class Dog:
    def bark():
        return "Woof!"

dog = Dog()
dog.bark()
```

Click for Answer

Error: `TypeError: bark() takes 0 positional arguments but 1 was given`

Fix:

```
class Dog:
    def bark(self):  #   Add self
        return "Woof!"

dog = Dog()
print(dog.bark())  #
```

17.9 16.8 Key Takeaways

17.9.1 What You Learned

1. **Include self** - In all instance methods
2. **Call super().___init___()** - In derived classes
3. **Initialize all attributes** - In **init**
4. **Use @property** - For computed attributes
5. **Check with hasattr()** - Before accessing attributes
6. **Provide defaults** - For optional parameters
7. **Use isinstance()** - For type checking

17.9.2 Common Patterns

```
# Pattern 1: Basic class
class MyClass:
    def __init__(self, value):
        self.value = value

    def method(self):
        return self.value

# Pattern 2: Inheritance
class Child(Parent):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

# Pattern 3: Property
class MyClass:
    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, val):
        self._value = val
```

17.10 16.9 Moving Forward

You now understand OOP! In **Chapter 17**, we'll explore **Modules and Imports**!

Chapter 18

Chapter 17: Modules and Imports - Import Errors

18.1 Introduction

Modules organize code into reusable files. Understanding imports is essential for using Python libraries and organizing your own code.

Common errors: - **ModuleNotFoundError**: Module not installed or found - **ImportError**: Can't import specific name - **AttributeError**: Module attribute doesn't exist - Circular imports

Let's master imports!

18.2 17.1 Basic Imports

18.2.1 Import Syntax

```python
# Import entire module
import math
print(math.pi)  # 3.14159...

# Import specific function
from math import sqrt
print(sqrt(16))  # 4.0

# Import multiple
from math import pi, sqrt, ceil
```

```
# Import with alias
import numpy as np
arr = np.array([1, 2, 3])

# Import all (not recommended)
from math import *

# Import submodule
from os.path import join
path = join('folder', 'file.txt')
```

18.2.2 Error Type 1: `ModuleNotFoundError: No module named 'module_name'`

Error Message:

```
>>> import pandas
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'pandas'
```

What Happened: Module not installed or not in Python path.

Why It Happens: - Module not installed - Wrong module name - Wrong Python environment - Module in wrong location

Code Example - WRONG:

```
# Module not installed
import pandas  # ERROR if not installed

# Typo in name
import nump  # ERROR! Should be numpy

# Wrong capitalization
import Pandas  # ERROR! Should be pandas

# Module doesn't exist
import my_nonexistent_module  # ERROR!
```

Code Example - CORRECT:

```
# Install module first
# pip install pandas

import pandas  #   After installation
```

```
# Check if module exists before importing
try:
    import pandas as pd
except ModuleNotFoundError:
    print("pandas not installed")  #
    pd = None

# Use importlib to check
import importlib.util
spec = importlib.util.find_spec("pandas")
if spec is not None:
    import pandas  #   Module exists
else:
    print("pandas not found")

# Correct module name
import numpy  #   Correct spelling

# Check installed packages
# pip list
# pip show pandas

# Use correct Python environment
# python -m pip install pandas
# python3 -m pip install pandas

# Add to path if needed
import sys
sys.path.append('/path/to/module')  #
import my_module
```

18.3 17.2 From Imports

18.3.1 Importing Specific Names

```
# Import specific function
from math import sqrt, pi

# Import class
from datetime import datetime

# Import with alias
```

```
from collections import defaultdict as dd

# Import from subpackage
from os.path import join, exists

# Multiple lines
from mymodule import (
    function1,
    function2,
    MyClass
)
```

18.3.2 Error Type 2: `ImportError: cannot import name 'name' from 'module'`

Error Message:

```
>>> from math import square_root
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: cannot import name 'square_root' from 'math'
```

What Happened: Trying to import name that doesn't exist in module.

Why It Happens: - Function/class doesn't exist - Typo in name - Wrong module - Circular import

Code Example - WRONG:

```
# Function doesn't exist
from math import square_root  # ERROR! It's sqrt

# Class doesn't exist
from datetime import Date  # ERROR! It's date

# Wrong module
from os import listdir  # Actually in os module, this works
from sys import listdir  # ERROR! Not in sys

# Typo
from math import squrt  # ERROR! Typo
```

Code Example - CORRECT:

```
# Correct function name
from math import sqrt  #
```

```
# Correct class name
from datetime import date  #

# Check what's in module
import math
print(dir(math))  #   List all names

# Check if name exists
if hasattr(math, 'sqrt'):
    from math import sqrt  #
else:
    print("sqrt not in math")

# Try/except for import
try:
    from math import sqrt
except ImportError:
    print("Cannot import sqrt")  #
    sqrt = None

# Import module, then access
import math
result = math.sqrt(16)  #   Always works

# Check module documentation
help(math)  #   See available names
```

18.4 17.3 Creating Your Own Modules

18.4.1 Module Structure

```
# mymodule.py
"""
My custom module
"""

# Module-level constant
PI = 3.14159

# Function
def greet(name):
    """Greet someone"""
```

```
    return f"Hello, {name}!"

# Class
class Calculator:
    """Simple calculator"""
    @staticmethod
    def add(x, y):
        return x + y

# Main execution guard
if __name__ == "__main__":
    print("Running as script")
    print(greet("World"))
```

```
# Use the module (in another file)
import mymodule

print(mymodule.PI)
print(mymodule.greet("Alice"))
calc = mymodule.Calculator()
print(calc.add(5, 3))
```

18.4.2 Error Type 3: Circular Import Error

What Happened: Two modules import each other.

Code Example - WRONG:

```
# module_a.py
from module_b import func_b

def func_a():
    return func_b()

# module_b.py
from module_a import func_a  # ERROR! Circular

def func_b():
    return func_a()

# main.py
import module_a  # ERROR! Circular import
```

Code Example - CORRECT:

```python
# Solution 1: Restructure to remove circular dependency
# module_a.py
def func_a():
    from module_b import func_b  #  Import inside function
    return func_b()

# module_b.py
def func_b():
    return "result"

# Solution 2: Create third module
# common.py
def shared_function():
    return "shared"

# module_a.py
from common import shared_function

def func_a():
    return shared_function()

# module_b.py
from common import shared_function

def func_b():
    return shared_function()

# Solution 3: Use lazy import
# module_a.py
def func_a():
    import module_b  #  Import when called
    return module_b.func_b()

# Solution 4: Import at bottom
# module_a.py
def func_a():
    return func_b()

from module_b import func_b  #  After definition
```

18.5 17.4 Package Structure

18.5.1 Creating Packages

```
mypackage/
    __init__.py
    module1.py
    module2.py
    subpackage/
        __init__.py
        module3.py
```

```python
# mypackage/__init__.py
"""Package initialization"""
from .module1 import function1
from .module2 import Class2

__all__ = ['function1', 'Class2']

# mypackage/module1.py
def function1():
    return "Function 1"

# mypackage/module2.py
class Class2:
    pass

# Usage
from mypackage import function1, Class2
from mypackage.subpackage import module3
```

18.6 17.5 Import Best Practices

18.6.1 Guidelines

```python
#  GOOD: Import order (PEP 8)
# 1. Standard library
import os
import sys
from datetime import datetime

# 2. Third-party
import numpy as np
import pandas as pd
```

```
# 3. Local/custom
from mymodule import myfunction

#   GOOD: Specific imports
from math import sqrt, pi

#   AVOID: Import *
from math import *  # Pollutes namespace

#   GOOD: Clear aliases
import numpy as np
import pandas as pd

#   AVOID: Unclear aliases
import numpy as n
import pandas as p

#   GOOD: Group related imports
from os import (
    path,
    listdir,
    makedirs
)

#   AVOID: Multiple statements per line
import sys, os  # Use separate lines

#   GOOD: Absolute imports
from mypackage.subpackage import module

#   GOOD: Relative imports (within package)
from . import sibling_module
from .. import parent_module
from ..sibling import cousin_module
```

18.7 17.6 Common Patterns

18.7.1 Import Patterns

```
# Conditional imports
try:
    import pandas as pd
```

```
    HAS_PANDAS = True
except ImportError:
    HAS_PANDAS = False

if HAS_PANDAS:
    # Use pandas
    df = pd.DataFrame()

# Version checking
import sys
if sys.version_info < (3, 6):
    raise RuntimeError("Python 3.6+ required")

# Dynamic imports
module_name = "math"
module = __import__(module_name)
result = module.sqrt(16)

# Or use importlib
import importlib
module = importlib.import_module("math")
result = module.sqrt(16)

# Lazy imports
class MyClass:
    def method(self):
        import expensive_module  #  Only import when needed
        return expensive_module.function()

# Optional dependencies
try:
    import matplotlib.pyplot as plt
    CAN_PLOT = True
except ImportError:
    CAN_PLOT = False

def plot_data(data):
    if not CAN_PLOT:
        print("matplotlib not available")
        return
    plt.plot(data)
    plt.show()
```

18.8 17.7 Practice Problems

18.8.1 Problem 1: Module Not Found

```
import numpyy
```

Click for Answer

Error: `ModuleNotFoundError: No module named 'numpyy'`

Fix:

```
import numpy  #   Correct spelling

# Or install if needed
# pip install numpy
```

18.8.2 Problem 2: Import Name Error

```
from math import square_root
```

Click for Answer

Error: `ImportError: cannot import name 'square_root'`

Fix:

```
from math import sqrt  #   Correct name

# Check available names
import math
print(dir(math))  #
```

18.9 17.8 Key Takeaways

18.9.1 What You Learned

1. **Install before importing** - pip install package
2. **Check spelling** - Module names are case-sensitive
3. **Avoid circular imports** - Restructure code
4. **Use try/except** - For optional imports
5. **Import at top** - Unless lazy loading
6. **Follow PEP 8** - Import order and style
7. **Check with dir()** - See module contents

18.10 17.9 Moving Forward

You now understand imports! In **Chapter 18**, we'll explore **Exception Handling**!

Chapter 19

Chapter 18: Exception Handling - try/except Patterns

19.1 Introduction

Exception handling lets you gracefully manage errors instead of crashing. Proper exception handling is crucial for robust applications.

Common topics: - try/except/finally blocks - Multiple exception types - Raising exceptions - Custom exceptions - Best practices

Let's master exception handling!

19.2 18.1 Basic Exception Handling

19.2.1 try/except Blocks

```
# Basic try/except
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
    result = None

# Multiple exceptions
try:
```

```
    value = int("abc")
except ValueError:
    print("Invalid number")
except TypeError:
    print("Wrong type")

# Catch any exception
try:
    risky_operation()
except Exception as e:
    print(f"Error: {e}")

# Multiple exceptions in one block
try:
    operation()
except (ValueError, TypeError) as e:
    print(f"Error: {e}")

# Get exception details
try:
    result = 10 / 0
except ZeroDivisionError as e:
    print(f"Error type: {type(e)}")
    print(f"Error message: {e}")
```

19.3 18.2 else and finally

19.3.1 Complete try Block

```
# try/except/else/finally
try:
    file = open('data.txt', 'r')
    data = file.read()
except FileNotFoundError:
    print("File not found")
    data = None
else:
    print("File read successfully")  # Only if no exception
finally:
    print("Cleanup")  # Always executes
    if 'file' in locals():
        file.close()
```

```python
# Common pattern
try:
    result = risky_operation()
except Exception as e:
    print(f"Error: {e}")
    result = default_value
else:
    print("Success")
finally:
    cleanup()

# finally for cleanup
file = None
try:
    file = open('data.txt', 'r')
    data = file.read()
except Exception as e:
    print(f"Error: {e}")
finally:
    if file:
        file.close()  # Always closes
```

19.4 18.3 Raising Exceptions

19.4.1 Creating Exceptions

```python
# Raise exception
def divide(a, b):
    if b == 0:
        raise ValueError("Divisor cannot be zero")
    return a / b

# Re-raise exception
try:
    risky_operation()
except Exception as e:
    print(f"Logging error: {e}")
    raise  # Re-raise same exception

# Raise different exception
try:
    external_api_call()
```

```python
except ExternalAPIError as e:
    raise RuntimeError("API failed") from e

# Raise with context
def process_data(data):
    if not data:
        raise ValueError("Data cannot be empty")
    if not isinstance(data, list):
        raise TypeError(f"Expected list, got {type(data)}")
    return process(data)
```

19.5 18.4 Custom Exceptions

19.5.1 Creating Custom Exceptions

```python
# Basic custom exception
class MyCustomError(Exception):
    pass

raise MyCustomError("Something went wrong")

# With additional data
class ValidationError(Exception):
    def __init__(self, message, field=None):
        super().__init__(message)
        self.field = field

try:
    raise ValidationError("Invalid email", field="email")
except ValidationError as e:
    print(f"Error in {e.field}: {e}")

# Exception hierarchy
class AppError(Exception):
    """Base exception for app"""
    pass

class DatabaseError(AppError):
    """Database related errors"""
    pass

class NetworkError(AppError):
    """Network related errors"""
```

```python
    pass

try:
    raise DatabaseError("Connection failed")
except DatabaseError as e:
    print("Database error")
except AppError as e:
    print("App error")

# With additional methods
class HTTPError(Exception):
    def __init__(self, status_code, message):
        self.status_code = status_code
        self.message = message
        super().__init__(self.message)

    def is_client_error(self):
        return 400 <= self.status_code < 500

    def is_server_error(self):
        return 500 <= self.status_code < 600

try:
    raise HTTPError(404, "Not Found")
except HTTPError as e:
    if e.is_client_error():
        print("Client error")
```

19.6 18.5 Exception Best Practices

19.6.1 Good Patterns

```python
#   GOOD: Specific exceptions
try:
    value = int(user_input)
except ValueError:  # Specific
    print("Invalid number")

#   AVOID: Bare except
try:
    value = int(user_input)
except:  # Catches everything, even KeyboardInterrupt!
    print("Error")
```

```
#   GOOD: Catch specific, then general
try:
    operation()
except ValueError:
    handle_value_error()
except TypeError:
    handle_type_error()
except Exception as e:
    handle_general_error(e)

#   AVOID: Catch Exception first
try:
    operation()
except Exception:  # Too broad, catches everything
    pass
except ValueError:  # Never reached!
    pass

#   GOOD: Don't suppress errors
try:
    important_operation()
except Exception as e:
    logger.error(f"Error: {e}")
    raise  # Re-raise

#   AVOID: Silent failures
try:
    important_operation()
except:
    pass  # Error lost!

#   GOOD: Use finally for cleanup
resource = None
try:
    resource = acquire_resource()
    use_resource(resource)
finally:
    if resource:
        resource.release()

#   GOOD: Use context managers (better than finally)
with open('file.txt', 'r') as file:
    data = file.read()
# File automatically closed
```

```
#  GOOD: Fail fast
def process(data):
    if not data:
        raise ValueError("Data required")
    # Process data

#  AVOID: Catching too much
try:
    # Many operations
    operation1()
    operation2()
    operation3()
except Exception:
    # Which operation failed?
    pass

#  GOOD: Narrow try blocks
try:
    operation1()
except SpecificError:
    handle_error()

try:
    operation2()
except AnotherError:
    handle_error()
```

19.7 18.6 Common Exception Types

19.7.1 Built-in Exceptions

```
# ValueError: Invalid value
try:
    int("abc")
except ValueError:
    print("Cannot convert to int")

# TypeError: Wrong type
try:
    "2" + 2
except TypeError:
    print("Cannot add string and int")
```

```
# KeyError: Key doesn't exist
try:
    d = {'a': 1}
    value = d['b']
except KeyError:
    print("Key not found")

# IndexError: Index out of range
try:
    lst = [1, 2, 3]
    value = lst[10]
except IndexError:
    print("Index out of range")

# FileNotFoundError: File doesn't exist
try:
    with open('nonexistent.txt') as f:
        data = f.read()
except FileNotFoundError:
    print("File not found")

# AttributeError: Attribute doesn't exist
try:
    x = 5
    x.append(1)
except AttributeError:
    print("Attribute doesn't exist")

# ZeroDivisionError: Division by zero
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")

# ImportError: Cannot import
try:
    import nonexistent_module
except ImportError:
    print("Module not found")

# RuntimeError: General runtime error
try:
    raise RuntimeError("Something went wrong")
except RuntimeError:
    print("Runtime error")
```

19.8 18.7 Context Managers

19.8.1 with Statement

```python
# File handling
with open('file.txt', 'r') as file:
    data = file.read()
# File automatically closed

# Multiple resources
with open('input.txt', 'r') as infile, \
     open('output.txt', 'w') as outfile:
    data = infile.read()
    outfile.write(data)

# Custom context manager
class DatabaseConnection:
    def __enter__(self):
        print("Opening connection")
        self.conn = connect_to_database()
        return self.conn

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("Closing connection")
        self.conn.close()
        return False  # Don't suppress exceptions

with DatabaseConnection() as conn:
    conn.execute("SELECT * FROM users")

# Using contextlib
from contextlib import contextmanager

@contextmanager
def managed_resource():
    resource = acquire_resource()
    try:
        yield resource
    finally:
        resource.release()

with managed_resource() as resource:
    use_resource(resource)
```

19.9 18.8 Assertion Errors

19.9.1 Using Assertions

```
# Basic assertion
x = 5
assert x > 0, "x must be positive"

# Development checks
def calculate_average(numbers):
    assert len(numbers) > 0, "List cannot be empty"
    return sum(numbers) / len(numbers)

#   GOOD: Use for development checks
def set_age(age):
    assert isinstance(age, int), "Age must be int"
    assert age >= 0, "Age must be positive"
    self.age = age

#   AVOID: For user input validation
def process_input(user_input):
    assert user_input  # BAD! Use proper validation
    return process(user_input)

#   GOOD: Proper validation
def process_input(user_input):
    if not user_input:
        raise ValueError("Input required")
    return process(user_input)

# Note: Assertions can be disabled with -O flag
# python -O script.py  # Skips assertions
```

19.10 18.9 Logging Errors

19.10.1 Error Logging

```
import logging

# Configure logging
```

```python
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)

logger = logging.getLogger(__name__)

# Log exceptions
try:
    risky_operation()
except Exception as e:
    logger.error(f"Operation failed: {e}")
    logger.exception("Full traceback:")  # Includes traceback

# Different log levels
try:
    operation()
except ValueError as e:
    logger.warning(f"Validation error: {e}")
except Exception as e:
    logger.error(f"Unexpected error: {e}")
    logger.exception("Details:")

# Custom exception logging
class CustomError(Exception):
    def __init__(self, message, code):
        super().__init__(message)
        self.code = code

try:
    raise CustomError("Failed", code=500)
except CustomError as e:
    logger.error(f"Error {e.code}: {e}")
```

19.11 18.10 Practice Problems

19.11.1 Problem 1: Bare Except

```python
try:
    value = int(input())
except:
    print("Error")
```

Click for Answer

Issue: Catches everything, including KeyboardInterrupt

Fix:

```
try:
    value = int(input())
except ValueError:  #  Specific exception
    print("Invalid number")
except KeyboardInterrupt:
    print("Cancelled")
```

19.12 18.11 Key Takeaways

19.12.1 What You Learned

1. **Use specific exceptions** - Not bare except
2. **Clean up in finally** - Or use context managers
3. **Don't suppress errors** - Log and re-raise
4. **Raise for validation** - Don't return None
5. **Create custom exceptions** - For app-specific errors
6. **Log exceptions** - Use logging.exception()
7. **Use with statement** - For resources

19.12.2 Common Patterns

```
# Pattern 1: Specific handling
try:
    operation()
except SpecificError:
    handle()

# Pattern 2: With cleanup
try:
    resource = acquire()
    use(resource)
finally:
    release(resource)

# Pattern 3: Context manager
with resource() as r:
    use(r)

# Pattern 4: Log and re-raise
```

```python
try:
    operation()
except Exception as e:
    logger.error(f"Error: {e}")
    raise
```

19.13 18.12 Moving Forward

You now understand exception handling! In **Chapter 19**, we'll explore **Debugging Techniques**!

Chapter 20

Chapter 19: Debugging Techniques - Finding and Fixing Errors

20.1 Introduction

Debugging is the art of finding and fixing bugs. Mastering debugging techniques dramatically improves your productivity and code quality.

Topics covered: - Print debugging - Using debuggers (pdb) - IDE debugging - Logging - Common debugging strategies

Let's master debugging!

20.2 19.1 Print Debugging

20.2.1 Basic Debugging with Print

```
# Simple print debugging
def calculate_total(items):
    total = 0
    print(f"Starting calculation with {len(items)} items")  # Debug
    for item in items:
        print(f"Processing item: {item}")  # Debug
        total += item['price']
    print(f"Final total: {total}")  # Debug
    return total
```

```
# Print variable types
value = get_value()
print(f"value = {value}, type = {type(value)}")  # Debug

# Print with context
def process(data):
    print(f"[process] Input: {data}")  # Debug with context
    result = transform(data)
    print(f"[process] Result: {result}")  # Debug
    return result

# Use repr() for detailed output
text = "hello\nworld"
print(f"text = {text!r}")  # 'hello\nworld' (shows newline)

# Temporary assertions
def divide(a, b):
    print(f"divide({a}, {b})")  # Debug
    assert b != 0, f"b is {b}"  # Debug assertion
    return a / b
```

20.3 19.2 Python Debugger (pdb)

20.3.1 Using pdb

```
import pdb

# Set breakpoint
def buggy_function(x, y):
    result = x + y
    pdb.set_trace()  # Execution pauses here
    return result * 2

# Python 3.7+: builtin breakpoint()
def buggy_function(x, y):
    result = x + y
    breakpoint()  # Easier syntax
    return result * 2

# Post-mortem debugging
try:
    buggy_operation()
```

```
except Exception:
    import pdb
    pdb.post_mortem()  # Debug at exception

# pdb commands:
# n (next) - Execute current line
# s (step) - Step into function
# c (continue) - Continue execution
# l (list) - Show code
# p variable - Print variable
# pp variable - Pretty print
# w (where) - Show stack trace
# u (up) - Move up stack
# d (down) - Move down stack
# q (quit) - Exit debugger
```

20.4 19.3 Logging for Debugging

20.4.1 Strategic Logging

```
import logging

# Configure logging
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    filename='debug.log'
)

logger = logging.getLogger(__name__)

def process_data(data):
    logger.debug(f"Processing {len(data)} items")

    for i, item in enumerate(data):
        logger.debug(f"Item {i}: {item}")
        try:
            result = transform(item)
            logger.debug(f"Transformed to: {result}")
        except Exception as e:
            logger.error(f"Failed on item {i}: {e}")
            logger.exception("Full traceback:")
```

```
    logger.info("Processing complete")

# Different log levels
logger.debug("Detailed debugging info")
logger.info("General information")
logger.warning("Warning message")
logger.error("Error occurred")
logger.critical("Critical error!")

# Conditional logging
if logger.isEnabledFor(logging.DEBUG):
    expensive_debug_info = compute_debug_info()
    logger.debug(f"Debug info: {expensive_debug_info}")

# Multiple loggers
user_logger = logging.getLogger('user_actions')
system_logger = logging.getLogger('system')

user_logger.info("User logged in")
system_logger.debug("System check passed")
```

20.5 19.4 Common Debugging Strategies

20.5.1 Systematic Approaches

```
# 1. Binary search / Divide and conquer
def complex_function():
    step1()
    step2()
    print("Checkpoint 1")  # Add checkpoints
    step3()
    step4()
    print("Checkpoint 2")
    step5()

# 2. Simplify inputs
# Instead of:
result = complex_function(large_data, many_params, complex_config)

# Try:
result = complex_function([1, 2, 3], simple_params, default_config)
```

```
# 3. Comment out sections
def buggy_function():
    part1()
    part2()
    # part3()  # Comment out to isolate
    # part4()
    part5()

# 4. Add assertions
def process(data):
    assert data is not None, "Data is None"
    assert len(data) > 0, f"Data is empty: {data}"

    result = transform(data)

    assert result is not None, "Result is None"
    assert isinstance(result, list), f"Result type: {type(result)}"

    return result

# 5. Rubber duck debugging
# Explain your code line by line (to rubber duck)
# Often reveals the bug!

# 6. Check assumptions
def divide(a, b):
    # Assumption: b is never zero
    print(f"Dividing {a} by {b}")
    print(f"b == 0? {b == 0}")  # Check assumption
    return a / b

# 7. Read error messages carefully
try:
    result = data['key']['subkey'][0]
except Exception as e:
    print(f"Error type: {type(e)}")
    print(f"Error message: {e}")
    print(f"Error args: {e.args}")
    import traceback
    traceback.print_exc()

# 8. Verify data types
def process(value):
    print(f"value: {value}")
    print(f"type: {type(value)}")
```

```
    print(f"is None: {value is None}")
    print(f"is empty: {not value}")
    print(f"len: {len(value) if hasattr(value, '__len__') else 'N/A'}")
```

20.6 19.5 Common Bug Patterns

20.6.1 Recognizing Patterns

```
# 1. Off-by-one errors
# Wrong:
for i in range(len(items) + 1):  # Goes too far!
    print(items[i])

# Correct:
for i in range(len(items)):
    print(items[i])

# 2. Mutable default arguments
# Wrong:
def add_item(item, items=[]):
    items.append(item)
    return items

# Correct:
def add_item(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items

# 3. Variable scope issues
# Wrong:
def update_global():
    count = count + 1  # UnboundLocalError

# Correct:
def update_global():
    global count
    count = count + 1

# 4. String/bytes confusion
# Wrong:
```

```
text = b"hello"
text.upper()  # Returns bytes, not string!

# Correct:
text = b"hello"
text = text.decode('utf-8')
text.upper()  # Now works

# 5. Integer division
# Python 2 vs 3:
result = 5 / 2  # 2 in Python 2, 2.5 in Python 3

# Explicit:
result = 5 // 2  # 2 (integer division)
result = 5 / 2   # 2.5 (float division)

# 6. Reference vs copy
# Wrong:
list1 = [1, 2, 3]
list2 = list1  # Reference, not copy!
list2.append(4)
print(list1)  # [1, 2, 3, 4] - modified!

# Correct:
list1 = [1, 2, 3]
list2 = list1.copy()  # Or list(list1) or list1[:]
list2.append(4)
print(list1)  # [1, 2, 3] - unchanged
```

20.7 19.6 IDE Debugging Tools

20.7.1 Using IDE Debuggers

```
# Most IDEs (PyCharm, VS Code, etc.) provide:

# 1. Breakpoints
#    - Click left of line number
#    - Code pauses when reached

# 2. Step controls
#    - Step Over: Execute line, don't enter functions
#    - Step Into: Enter function calls
```

```
#    - Step Out: Exit current function
#    - Continue: Run until next breakpoint

# 3. Variable inspection
#    - View all variables
#    - Evaluate expressions
#    - Modify values during debugging

# 4. Watch expressions
#    - Monitor specific variables
#    - Track changes

# 5. Call stack
#    - See function call history
#    - Navigate up/down stack

# 6. Conditional breakpoints
#    - Only pause when condition is True
#    - Example: i == 10

# 7. Exception breakpoints
#    - Pause on any exception
#    - Or specific exception types
```

20.8 19.7 Performance Debugging

20.8.1 Finding Slow Code

```python
import time

# Simple timing
start = time.time()
slow_function()
end = time.time()
print(f"Took {end - start:.2f} seconds")

# Context manager for timing
from contextlib import contextmanager

@contextmanager
def timer(name):
    start = time.time()
    yield
```

```
    end = time.time()
    print(f"{name} took {end - start:.2f} seconds")

with timer("Database query"):
    query_database()

# Profile with timeit
import timeit

# Time a statement
time = timeit.timeit('sum(range(100))', number=10000)
print(f"Time: {time}")

# Compare approaches
time1 = timeit.timeit('[i for i in range(1000)]', number=1000)
time2 = timeit.timeit('list(range(1000))', number=1000)
print(f"List comp: {time1}, list(): {time2}")

# Use cProfile
import cProfile
import pstats

cProfile.run('expensive_function()', 'output.prof')

# Analyze results
stats = pstats.Stats('output.prof')
stats.sort_stats('cumulative')
stats.print_stats(10)  # Top 10 functions

# Line profiler (requires line_profiler package)
# @profile decorator
# kernprof -l script.py
# python -m line_profiler script.py.lprof

# Memory profiling (requires memory_profiler)
# from memory_profiler import profile
# @profile
# def my_function():
#     ...
```

20.9.1 Systematic Approach

20.9 19.8 Debugging Checklist

```
"""
When you encounter a bug:

1.  Reproduce the bug
   - Can you make it happen consistently?
   - What are the exact steps?
   - What inputs cause it?

2.  Isolate the problem
   - Which function/module?
   - Binary search: comment out code
   - Simplify inputs

3.  Examine the error
   - Read error message carefully
   - Note the line number
   - Check the stack trace

4.  Form a hypothesis
   - What do you think is wrong?
   - What should happen vs what does happen?

5.  Test your hypothesis
   - Add print statements
   - Use debugger
   - Add assertions

6.  Fix the bug
   - Make smallest change possible
   - Don't add features while fixing bugs

7.  Verify the fix
   - Run the test case
   - Check edge cases
   - Make sure you didn't break anything else

8.  Prevent regression
   - Add a test
   - Document the bug
   - Review similar code
"""
```

20.10 19.9 Debugging Tools Summary

20.10.1 Tool Comparison

```
# Print debugging
#   Quick and simple
#   Works everywhere
#   Clutters code
#   Easy to forget to remove

# pdb (Python debugger)
#   Interactive
#   Inspect variables
#   Command line only
#   Learning curve

# IDE debuggers
#   Visual interface
#   Easy breakpoints
#   Variable inspection
#   Requires IDE

# Logging
#   Permanent
#   Levels (debug, info, error)
#   Can log to file
#   Overhead

# Assertions
#   Check assumptions
#   Document expectations
#   Can be disabled
#   Crashes on failure
```

20.11 19.10 Key Takeaways

20.11.1 What You Learned

1. **Use print strategically -** Add context
2. **Learn pdb basics -** n, s, c, p commands
3. **Use logging -** For production code

4. **Read errors carefully** - Stack trace has clues
5. **Isolate the problem** - Binary search
6. **Check assumptions** - Add assertions
7. **Use IDE debugger** - Visual and powerful

20.11.2 Debugging Workflow

```
# 1. Reproduce
# 2. Isolate
# 3. Understand
# 4. Fix
# 5. Test
# 6. Prevent
```

20.12 19.11 Moving Forward

You now understand debugging! In **Chapter 20**, we'll explore **Testing and Code Quality** - the final chapter!

Chapter 21

Chapter 20: Testing and Code Quality - Writing Better Code

21.1 Introduction

Testing and code quality ensure your code works correctly and is maintainable. This final chapter covers testing frameworks, best practices, and tools for writing professional Python code.

Topics covered: - Unit testing - Test-driven development - Code quality tools - Best practices

Let's complete your journey!

21.2 20.1 Unit Testing Basics

21.2.1 Using unittest

```
import unittest

# Code to test
def add(a, b):
    return a + b

def divide(a, b):
    if b == 0:
```

```
        raise ValueError("Cannot divide by zero")
    return a / b

# Test class
class TestMath(unittest.TestCase):

    def test_add(self):
        self.assertEqual(add(2, 3), 5)
        self.assertEqual(add(-1, 1), 0)
        self.assertEqual(add(0, 0), 0)

    def test_divide(self):
        self.assertEqual(divide(10, 2), 5)
        self.assertEqual(divide(9, 3), 3)

    def test_divide_by_zero(self):
        with self.assertRaises(ValueError):
            divide(10, 0)

    def test_types(self):
        self.assertIsInstance(add(1, 2), int)
        self.assertTrue(add(1, 1) > 0)
        self.assertFalse(add(0, 0) > 0)

# Run tests
if __name__ == '__main__':
    unittest.main()
```

21.3 20.2 Common Test Assertions

21.3.1 unittest Assertions

```
import unittest

class TestAssertions(unittest.TestCase):

    def test_equality(self):
        self.assertEqual(1, 1)
        self.assertNotEqual(1, 2)

    def test_truth(self):
        self.assertTrue(True)
        self.assertFalse(False)
```

```
    def test_none(self):
        self.assertIsNone(None)
        self.assertIsNotNone("value")

    def test_membership(self):
        self.assertIn(1, [1, 2, 3])
        self.assertNotIn(4, [1, 2, 3])

    def test_types(self):
        self.assertIsInstance(1, int)
        self.assertIsInstance("text", str)

    def test_comparisons(self):
        self.assertGreater(2, 1)
        self.assertLess(1, 2)
        self.assertGreaterEqual(2, 2)
        self.assertLessEqual(1, 2)

    def test_sequences(self):
        self.assertListEqual([1, 2], [1, 2])
        self.assertDictEqual({'a': 1}, {'a': 1})

    def test_exceptions(self):
        with self.assertRaises(ValueError):
            int("abc")

        with self.assertRaises(ZeroDivisionError):
            1 / 0

    def test_almost_equal(self):
        self.assertAlmostEqual(0.1 + 0.2, 0.3)
```

21.4 20.3 pytest Framework

21.4.1 Using pytest

```
# test_math.py
import pytest

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

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b

# Tests (no class needed)
def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0

def test_divide():
    assert divide(10, 2) == 5
    assert divide(9, 3) == 3

def test_divide_by_zero():
    with pytest.raises(ValueError):
        divide(10, 0)

# Parametrized tests
@pytest.mark.parametrize("a,b,expected", [
    (2, 3, 5),
    (-1, 1, 0),
    (0, 0, 0),
    (100, 200, 300),
])
def test_add_parametrized(a, b, expected):
    assert add(a, b) == expected

# Fixtures
@pytest.fixture
def sample_data():
    return [1, 2, 3, 4, 5]

def test_with_fixture(sample_data):
    assert len(sample_data) == 5
    assert sum(sample_data) == 15

# Run: pytest test_math.py
```

21.5 20.4 Test Organization

21.5.1 Structuring Tests

```
# Project structure
"""
myproject/
    mypackage/
        __init__.py
        module1.py
        module2.py
    tests/
        __init__.py
        test_module1.py
        test_module2.py
    setup.py
    README.md
"""

# test_module1.py
import unittest
from mypackage import module1

class TestModule1(unittest.TestCase):

    def setUp(self):
        """Run before each test"""
        self.data = [1, 2, 3]

    def tearDown(self):
        """Run after each test"""
        self.data = None

    def test_function1(self):
        result = module1.function1(self.data)
        self.assertEqual(result, expected)

    def test_function2(self):
        result = module1.function2(self.data)
        self.assertTrue(result)

# Run all tests
# python -m unittest discover tests
# pytest tests/
```

21.6 20.5 Test-Driven Development (TDD)

```python
"""
TDD Cycle:
1. Write test (it fails - Red)
2. Write code (make it pass - Green)
3. Refactor (improve code - Refactor)
4. Repeat
"""

# Example: TDD for a function

# Step 1: Write test first
def test_calculate_discount():
    assert calculate_discount(100, 10) == 90
    assert calculate_discount(50, 20) == 40

# Step 2: Run test (fails - function doesn't exist)

# Step 3: Write minimal code
def calculate_discount(price, discount_percent):
    return price - (price * discount_percent / 100)

# Step 4: Run test (passes)

# Step 5: Add more tests
def test_calculate_discount_edge_cases():
    assert calculate_discount(100, 0) == 100
    assert calculate_discount(100, 100) == 0
    with pytest.raises(ValueError):
        calculate_discount(100, -10)

# Step 6: Update code
def calculate_discount(price, discount_percent):
    if discount_percent < 0 or discount_percent > 100:
        raise ValueError("Discount must be 0-100")
    return price - (price * discount_percent / 100)

# Step 7: Refactor if needed
```

21.7 20.6 Code Coverage

21.7.1 Measuring Test Coverage

```
# Install coverage
# pip install coverage

# Run tests with coverage
# coverage run -m pytest

# View report
# coverage report

# Generate HTML report
# coverage html
# open htmlcov/index.html

# .coveragerc configuration
"""
[run]
source = mypackage
omit =
    */tests/*
    */venv/*

[report]
exclude_lines =
    pragma: no cover
    def __repr__
    raise NotImplementedError
    if __name__ == .__main__.:
"""

# Aim for high coverage
# 80%+ is good
# 100% is ideal but not always necessary
```

21.8 20.7 Code Quality Tools

21.8.1 Linting and Formatting

```
# pylint - Code analysis
# pip install pylint
```

```
# pylint mymodule.py

# flake8 - Style guide enforcement
# pip install flake8
# flake8 mymodule.py

# black - Code formatter
# pip install black
# black mymodule.py

# mypy - Type checking
# pip install mypy
# mypy mymodule.py

# Example with type hints
def add(a: int, b: int) -> int:
    """Add two integers"""
    return a + b

# isort - Import sorting
# pip install isort
# isort mymodule.py

# Example .flake8 config
"""
[flake8]
max-line-length = 88
extend-ignore = E203, W503
exclude =
    .git,
    __pycache__,
    venv
"""

# Example pyproject.toml for black
"""
[tool.black]
line-length = 88
target-version = ['py38']
include = '\.pyi?$'
"""
```

21.9.1 Writing Quality Code

21.9 20.8 Best Practices

```
# 1. Write docstrings
def calculate_total(items: list, tax_rate: float) -> float:
    """
    Calculate total price including tax.

    Args:
        items: List of items with 'price' key
        tax_rate: Tax rate as decimal (0.08 = 8%)

    Returns:
        Total price including tax

    Raises:
        ValueError: If tax_rate is negative

    Example:
        >>> items = [{'price': 10}, {'price': 20}]
        >>> calculate_total(items, 0.08)
        32.4
    """
    if tax_rate < 0:
        raise ValueError("Tax rate cannot be negative")

    subtotal = sum(item['price'] for item in items)
    return subtotal * (1 + tax_rate)

# 2. Use type hints
from typing import List, Dict, Optional

def process_data(
    data: List[Dict[str, any]],
    filter_key: Optional[str] = None
) -> List[Dict[str, any]]:
    """Process data with optional filtering"""
    if filter_key:
        return [d for d in data if filter_key in d]
    return data

# 3. Follow PEP 8
# - 4 spaces for indentation
# - 2 blank lines between functions
```

```
# - Lowercase with underscores for functions
# - CamelCase for classes

#   Good
def calculate_average(numbers):
    return sum(numbers) / len(numbers)

class DataProcessor:
    pass

#   Bad
def calculateAverage(numbers):  # camelCase
    return sum(numbers)/len(numbers)  # no spaces

# 4. Keep functions small
#   Good - One responsibility
def validate_email(email):
    return '@' in email and '.' in email

def send_email(email, message):
    if not validate_email(email):
        raise ValueError("Invalid email")
    # Send email

#   Bad - Too many responsibilities
def process_user_registration(email, password, name):
    # Validate email
    # Validate password
    # Hash password
    # Save to database
    # Send confirmation email
    # Log activity
    pass  # Too much!

# 5. Use meaningful names
#   Good
user_age = 25
total_price = calculate_total(items)
is_valid = validate_input(data)

#   Bad
x = 25
tmp = calc(items)
flag = check(data)
```

```python
# 6. Don't repeat yourself (DRY)
#   Bad
def calculate_area_rectangle(width, height):
    return width * height

def calculate_area_square(side):
    return side * side

#   Good
def calculate_area(width, height=None):
    if height is None:
        height = width
    return width * height

# 7. Handle errors properly
#   Good
def read_file(filename):
    try:
        with open(filename, 'r') as f:
            return f.read()
    except FileNotFoundError:
        logger.error(f"File not found: {filename}")
        raise
    except Exception as e:
        logger.error(f"Error reading {filename}: {e}")
        raise

#   Bad
def read_file(filename):
    try:
        with open(filename, 'r') as f:
            return f.read()
    except:
        pass  # Silent failure!
```

21.10 20.9 Continuous Integration

21.10.1 CI/CD Setup

```python
# GitHub Actions example
# .github/workflows/tests.yml
"""
```

```
name: Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: 3.9

    - name: Install dependencies
      run: |
        pip install -r requirements.txt
        pip install pytest coverage

    - name: Run tests
      run: |
        pytest

    - name: Check coverage
      run: |
        coverage run -m pytest
        coverage report --fail-under=80

    - name: Lint
      run: |
        pip install flake8
        flake8 .
"""

# pre-commit hooks
# .pre-commit-config.yaml
"""
repos:
  - repo: https://github.com/psf/black
    rev: 23.1.0
    hooks:
      - id: black
```

```
  - repo: https://github.com/pycqa/flake8
    rev: 6.0.0
    hooks:
      - id: flake8

  - repo: https://github.com/pycqa/isort
    rev: 5.12.0
    hooks:
      - id: isort
"""
```

21.11 20.10 Final Checklist

21.11.1 Code Quality Checklist

```
"""
Before committing code:

  Tests pass
  - All unit tests pass
  - Coverage > 80%
  - Edge cases covered

  Code quality
  - No linter warnings
  - Code formatted (black)
  - Imports sorted (isort)
  - Type hints added

  Documentation
  - Docstrings added
  - README updated
  - CHANGELOG updated

  Best practices
  - No duplicate code
  - Functions < 20 lines
  - No magic numbers
  - Error handling

  Review
  - Self-review changes
  - Test locally
```

```
  - Check CI passes
"""
```

21.12 20.11 Key Takeaways

21.12.1 What You Learned

1. **Write tests first** - TDD approach
2. **Test edge cases** - Not just happy path
3. **Aim for high coverage** - 80%+ is good
4. **Use quality tools** - Linters, formatters
5. **Follow PEP 8** - Consistent style
6. **Write docstrings** - Document your code
7. **Keep it simple** - KISS principle

21.12.2 Quality Pyramid

```
         /\
        /CI\
       /----\
      /Tests \
     /--------\
    /  Linting \
   /------------\
  / Code Reviews \
 /----------------\
```

21.13 20.12 Congratulations!

21.13.1 You've Completed the Python Error Guide!

You've mastered all 20 chapters:

Part I: Fundamentals - Variables, Operators, Strings - Lists, Dictionaries, Sets, Tuples - Conditionals, Loops, Functions, Files

Part II: Libraries - Regular Expressions - Pandas, NumPy, Matplotlib

Part III: Advanced - OOP, Modules, Exceptions - Debugging, Testing, Quality

---

21.14 20.13 Next Steps

21.14.1 Continue Your Python Journey

1. **Practice regularly**
    - Code every day
    - Build projects
    - Contribute to open source
2. **Read quality code**
    - Study Python standard library
    - Read popular projects
    - Learn from experts
3. **Stay updated**
    - Follow Python PEPs
    - Read blogs and articles
    - Join Python communities
4. **Specialize**
    - Web (Django, Flask)
    - Data Science (Pandas, scikit-learn)
    - DevOps (automation)
    - ML/AI (TensorFlow, PyTorch)
5. **Teach others**
    - Write blog posts
    - Answer questions
    - Mentor beginners

21.15 20.14 Resources

21.15.1 Recommended Learning

Books: - "Fluent Python" by Luciano Ramalho - "Effective Python" by Brett Slatkin - "Python Tricks" by Dan Bader

Websites: - Real Python (realpython.com) - Python Docs (docs.python.org) - PEP 8 Style Guide

Practice: - LeetCode - HackerRank - Project Euler

Communities: - r/Python - Python Discord - Stack Overflow

21.16 Thank You!

You've completed this comprehensive guide to Python errors and best practices. You now have the knowledge to:

- Understand and fix Python errors
- Write clean, maintainable code
- Test your code properly
- Debug effectively
- Follow best practices

Keep coding, keep learning, and keep growing!

End of Python Error Guide Thank you for learning with us!