Python Programming

A Comprehensive Journey from Basics to Modules

Master the fundamentals with detailed explanations and real-world examples

📚 What We'll Cover

- **Variables & Data Types** - The building blocks
- **Operators** - Performing operations
- **Control Flow** - Making decisions
- **Loops** - Repetition and iteration
- **Functions** - Reusable code blocks
- **Data Structures** - Lists, Tuples, Dictionaries, Sets
- **Strings** - Text manipulation
- **List Comprehensions** - Elegant list creation
- **File Handling** - Reading and writing files
- **Exception Handling** - Error management
- **Modules** - Organizing and reusing code

🎯 Variables and Data Types

What are Variables?

Variables are **containers** that store data values. Think of them as labeled boxes where you can put information and retrieve it later.

> **Key Point:** In Python, you don't need to declare the type explicitly. Python figures it out automatically (dynamic typing).

```
# Python automatically knows what type each variable is
name = "Alice"      # String
age = 25            # Integer
height = 5.6        # Float
is_student = True   # Boolean
```

📊 Core Data Types

1. Integer (int)

Whole numbers, positive or negative, without decimals.

```
count = 42
negative = -17
big_number = 1_000_000  # Underscores for readability
```

2. Float (float)

Numbers with decimal points.

```
price = 19.99
temperature = -3.5
scientific = 2.5e-4  # Scientific notation: 0.00025
```

📊 Core Data Types (continued)

3. String (str)

Text data enclosed in quotes (single or double).

```
single = 'Hello'
double = "World"
multi_line = """This is a
multi-line string"""
```

4. Boolean (bool)

True or False values (used for logic).

```
is_active = True
is_complete = False
result = (5 > 3)  # True
```

🔄 Type Conversion

You can convert between different data types when needed.

```
# String to Integer
age_str = "25"
age_int = int(age_str)  # 25

# Integer to String
count = 42
count_str = str(count)  # "42"

# String to Float
price_str = "19.99"
price_float = float(price_str)  # 19.99

# Checking types
print(type(age_int))     # <class 'int'>
print(isinstance(age_int, int))  # True
```

+ Arithmetic Operators

Operators perform operations on values and variables.

```
a = 10
b = 3

# Basic arithmetic
addition = a + b        # 13
subtraction = a - b     # 7
multiplication = a * b  # 30
division = a / b        # 3.333... (always returns float)
floor_division = a // b # 3 (rounds down to integer)
modulo = a % b          # 1 (remainder)
exponentiation = a ** b # 1000 (10 to the power of 3)
```

Pro Tip: Use `//` for integer division and `%` to check if numbers are divisible.

🔢 Compound Assignment Operators

Shortcuts for common operations.

```
counter = 10

counter += 5   # Same as: counter = counter + 5  → 15
counter -= 3   # Same as: counter = counter - 3  → 12
counter *= 2   # Same as: counter = counter * 2  → 24
counter /= 4   # Same as: counter = counter / 4  → 6.0
counter //= 2  # Same as: counter = counter // 2 → 3.0
counter %= 2   # Same as: counter = counter % 2  → 1.0
counter **= 3  # Same as: counter = counter ** 3 → 1.0
```

These make your code more concise and readable!

⚖️ Comparison Operators

Compare values and return Boolean results.

```
a = 10
b = 5

# Comparisons
print(a > b)   # True  (greater than)
print(a < b)   # False (less than)
print(a >= 10) # True  (greater than or equal)
print(a <= 5)  # False (less than or equal)
print(a == b)  # False (equal to)
print(a != b)  # True  (not equal to)

# Works with strings too!
print("apple" < "banana")  # True (alphabetical order)
print("Hello" == "hello")  # False (case-sensitive)
```

🔗 Logical Operators

Combine multiple conditions.

```
age = 25
has_license = True
is_student = False

# AND - both conditions must be True
can_rent_car = age >= 21 and has_license  # True

# OR - at least one condition must be True
gets_discount = age < 18 or is_student    # False

# NOT - reverses the boolean value
is_adult = not (age < 18)                 # True

# Complex conditions
can_vote_abroad = (age >= 18) and (has_license or has_passport)
```

🛤️ Control Flow: If Statements

Make decisions in your code based on conditions.

Basic If Statement

```
temperature = 25

if temperature > 30:
    print("It's hot outside!")
    print("Drink water!")
```

The indented code only runs if the condition is **True**. Python uses indentation (4 spaces) to define code blocks.

 If-Elif-Else Chain

Handle multiple conditions.

```
score = 85

if score >= 90:
    grade = "A"
    print("Excellent work!")
elif score >= 80:
    grade = "B"
    print("Great job!")
elif score >= 70:
    grade = "C"
    print("Good effort!")
elif score >= 60:
    grade = "D"
    print("You passed!")
else:
    grade = "F"
    print("Keep trying!")
```

🎯 Nested If Statements

If statements inside other if statements.

```
age = 25
has_ticket = True

if age >= 18:
    print("You are an adult.")
    if has_ticket:
        print("Welcome to the concert!")
    else:
        print("Please purchase a ticket.")
else:
    print("Sorry, you must be 18 or older.")
```

Ternary Operator (One-line If)

```
age = 20
```

🔁 For Loops

Repeat code for each item in a sequence.

Looping Through Lists

```
fruits = ["apple", "banana", "cherry", "date"]

for fruit in fruits:
    print(f"I love {fruit}!")

# Output:
# I love apple!
# I love banana!
# I love cherry!
# I love date!
```

Key Concept: The loop variable (`fruit`) takes on each value in the list, one at a

🔢 For Loops with Range

The `range()` function generates a sequence of numbers.

```
# range(stop) - from 0 to stop-1
for i in range(5):
    print(i)  # 0, 1, 2, 3, 4

# range(start, stop) - from start to stop-1
for i in range(2, 7):
    print(i)  # 2, 3, 4, 5, 6

# range(start, stop, step) - with custom increment
for i in range(0, 10, 2):
    print(i)  # 0, 2, 4, 6, 8

# Counting backwards
for i in range(10, 0, -1):
    print(i)  # 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
```

📊 Enumerate for Index and Value

Get both the index and value while looping.

```
fruits = ["apple", "banana", "cherry"]

# Without enumerate
for i in range(len(fruits)):
    print(f"{i}: {fruits[i]}")

# With enumerate (much cleaner!)
for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")

# Start counting from 1 instead of 0
for index, fruit in enumerate(fruits, start=1):
    print(f"{index}. {fruit}")

# Output:
# 1. apple
```

🔄 While Loops

Repeat code as long as a condition is True.

```
# Count to 5
count = 1
while count <= 5:
    print(f"Count: {count}")
    count += 1  # Don't forget to update the condition!

# User input validation
password = ""
while password != "secret":
    password = input("Enter password: ")
print("Access granted!")
```

Warning: Make sure your condition eventually becomes False, or you'll create an infinite loop!

Break and Continue

Control loop execution flow.

Break - Exit the loop immediately

```
# Find first even number
numbers = [1, 3, 5, 8, 9, 11]
for num in numbers:
    if num % 2 == 0:
        print(f"Found even number: {num}")
        break  # Stop searching
    print(f"Checking {num}...")
```

Continue - Skip to next iteration

```
# Print odd numbers only
for num in range(10):
```

🔁 Loop Else Clause

The `else` block runs if the loop completes without a `break`.

```
# Search for a value
search_value = 7
numbers = [1, 3, 5, 7, 9]

for num in numbers:
    if num == search_value:
        print(f"Found {search_value}!")
        break
else:
    # This runs only if break was NOT called
    print(f"{search_value} not found in list")

# Try with search_value = 4 to see the else block execute
```

Use Case: Useful for search operations where you need to know if the item was

🎯 Functions: The Basics

Functions are reusable blocks of code that perform specific tasks.

Why Use Functions?

- **Reusability:** Write once, use many times
- **Organization:** Break complex problems into smaller pieces
- **Maintainability:** Easier to update and debug

```
# Defining a function
def greet():
    print("Hello, World!")

# Calling the function
greet()  # Output: Hello, World!
greet()  # Can call it multiple times!
```

Function Parameters

Pass information to functions.

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

greet("Alice")  # Hello, Alice!
greet("Bob")    # Hello, Bob!

# Multiple parameters
def add_numbers(a, b):
    result = a + b
    print(f"{a} + {b} = {result}")

add_numbers(5, 3)   # 5 + 3 = 8
add_numbers(10, 20) # 10 + 20 = 30
```

📤 Return Values

Functions can send data back to the caller.

```
# Return a value
def square(number):
    return number ** 2

result = square(5)
print(result)  # 25

# Use directly in expressions
area = square(4) * 3.14
print(area)  # 50.24

# Return multiple values (as a tuple)
def divide_with_remainder(dividend, divisor):
    quotient = dividend // divisor
    remainder = dividend % divisor
    return quotient, remainder
```

Default Parameters

Provide default values for parameters.

```
# Default parameter value
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

greet("Alice")              # Hello, Alice!
greet("Bob", "Hi")          # Hi, Bob!
greet("Charlie", "Hey")     # Hey, Charlie!

# Multiple defaults
def power(base, exponent=2):
    return base ** exponent

print(power(5))      # 25 (uses default exponent=2)
print(power(5, 3))   # 125 (overrides default)
print(power(2, 10))  # 1024
```

🏷️ Keyword Arguments

Specify arguments by parameter name.

```
def create_profile(name, age, city, occupation):
    return f"{name}, {age} years old, {occupation} from {city}"

# Positional arguments
profile1 = create_profile("Alice", 30, "NYC", "Engineer")

# Keyword arguments (order doesn't matter!)
profile2 = create_profile(
    occupation="Doctor",
    name="Bob",
    city="LA",
    age=35
)

# Mix of both (positional must come first)
profile3 = create_profile("Charlie", 28, city="Chicago", occupation="Teacher")
```

📝 Docstrings

Document your functions with docstrings.

```
def calculate_area(length, width):
    """
    Calculate the area of a rectangle.

    Parameters:
        length (float): The length of the rectangle
        width (float): The width of the rectangle

    Returns:
        float: The area of the rectangle

    Example:
        >>> calculate_area(5, 3)
        15
    """
    return length * width

# Access the docstring
```

📋 Lists: Dynamic Arrays

Lists are **ordered**, **mutable** collections that can hold any type of data.

```
# Creating lists
empty = []
numbers = [1, 2, 3, 4, 5]
mixed = [1, "hello", 3.14, True]
nested = [[1, 2], [3, 4], [5, 6]]

# Accessing elements (0-indexed)
fruits = ["apple", "banana", "cherry"]
print(fruits[0])   # apple (first element)
print(fruits[2])   # cherry (third element)
print(fruits[-1])  # cherry (last element)
print(fruits[-2])  # banana (second from end)
```

Remember: Lists are mutable - you can change them after creation!

List Operations

Common ways to modify lists.

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

# Adding elements
fruits.append("cherry")         # Add to end: ['apple', 'banana', 'cherry']
fruits.insert(1, "blueberry")  # Insert at index 1
fruits.extend(["date", "fig"]) # Add multiple items

# Removing elements
fruits.remove("banana")         # Remove by value
popped = fruits.pop()           # Remove and return last item
first = fruits.pop(0)           # Remove and return at index
fruits.clear()                  # Remove all items

# Other operations
numbers = [3, 1, 4, 1, 5]
numbers.sort()                  # Sort in place: [1, 1, 3, 4, 5]
```

🔪 List Slicing

Extract portions of lists.

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Basic slicing: list[start:stop]
print(numbers[2:5])    # [2, 3, 4] (from index 2 to 4)
print(numbers[:4])     # [0, 1, 2, 3] (from start to index 3)
print(numbers[6:])     # [6, 7, 8, 9] (from index 6 to end)
print(numbers[:])      # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] (copy)

# With step: list[start:stop:step]
print(numbers[::2])    # [0, 2, 4, 6, 8] (every 2nd element)
print(numbers[1::2])   # [1, 3, 5, 7, 9] (odd numbers)
print(numbers[::-1])   # [9, 8, 7, 6, 5, 4, 3, 2, 1, 0] (reverse)

# Negative indices
print(numbers[-3:])    # [7, 8, 9] (last 3 elements)
print(numbers[:-2])    # [0, 1, 2, 3, 4, 5, 6, 7] (all but last 2)
```

🔍 List Methods and Functions

Useful operations on lists.

```
numbers = [3, 1, 4, 1, 5, 9, 2]

# Finding elements
print(numbers.index(4))        # 2 (index of first occurrence)
print(5 in numbers)            # True (membership test)
print(7 in numbers)            # False

# List information
print(len(numbers))            # 7 (length)
print(min(numbers))            # 1 (minimum value)
print(max(numbers))            # 9 (maximum value)
print(sum(numbers))            # 25 (sum of all elements)

# Sorting without modifying original
sorted_nums = sorted(numbers)          # Returns new sorted list
reverse_sorted = sorted(numbers, reverse=True)

# Copying lists
shallow_copy = numbers.copy()          # or numbers[:]
```

🔐 Tuples: Immutable Sequences

Tuples are like lists but **cannot be changed** after creation.

```
# Creating tuples
empty = ()
single = (42,)           # Note the comma!
coordinates = (10, 20)
rgb = (255, 128, 0)
mixed = (1, "hello", 3.14)

# Accessing elements (same as lists)
point = (100, 200, 300)
x = point[0]  # 100
y = point[1]  # 200
z = point[2]  # 300

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

🎯 Why Use Tuples?

When to use tuples vs lists:

```
# Use tuples for data that shouldn't change
DAYS_OF_WEEK = ("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun")
RGB_RED = (255, 0, 0)
DATABASE_CONFIG = ("localhost", 5432, "mydb")

# Tuples as dictionary keys (lists can't do this!)
locations = {
    (0, 0): "origin",
    (1, 0): "east",
    (0, 1): "north"
}

# Tuples are faster than lists
coordinates_list = [10, 20]
coordinates_tuple = (10, 20)  # More memory efficient

# Multiple return values (actually returns a tuple)
def get_user():
    return "Alice", 30, "Engineer"
```

🗂️ Dictionaries: Key-Value Pairs

Dictionaries store data as key-value pairs for fast lookups.

```
# Creating dictionaries
empty = {}
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York",
    "email": "alice@email.com"
}

# Accessing values
print(person["name"])          # Alice
print(person.get("age"))       # 30
print(person.get("phone"))     # None (key doesn't exist)
print(person.get("phone", "N/A"))  # N/A (default value)

# Adding/modifying
person["phone"] = "555-1234"   # Add new key
```

🔑 Dictionary Methods

Essential operations for working with dictionaries.

```
student = {
    "name": "Bob",
    "age": 20,
    "major": "Computer Science",
    "gpa": 3.8
}

# Getting keys, values, and items
keys = student.keys()           # dict_keys(['name', 'age', 'major', 'gpa'])
values = student.values()       # dict_values(['Bob', 20, 'Computer Science', 3.8])
items = student.items()         # dict_items([('name', 'Bob'), ...])

# Removing items
removed = student.pop("gpa")    # Remove and return value
last = student.popitem()        # Remove and return last item
student.clear()                 # Remove all items

# Checking existence
```

🔄 Iterating Through Dictionaries

Different ways to loop through dictionary data.

```
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

# Iterate over keys
for key in person:
    print(key)

# Iterate over values
for value in person.values():
    print(value)

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

# Output:
# name: Alice
```

🎨 Sets: Unique Collections

Sets are **unordered** collections of **unique** elements.

```
# Creating sets
empty = set()  # Note: {} creates an empty dict, not set!
numbers = {1, 2, 3, 4, 5}
fruits = {"apple", "banana", "cherry"}

# Duplicates are automatically removed
numbers = {1, 2, 2, 3, 3, 3, 4}
print(numbers)  # {1, 2, 3, 4}

# Adding and removing
fruits.add("orange")
fruits.add("apple")    # No effect (already exists)
fruits.remove("banana") # Raises error if not found
fruits.discard("grape") # No error if not found
fruits.pop()           # Remove arbitrary element
```

🔢 Set Operations

Mathematical set operations.

```
a = {1, 2, 3, 4, 5}
b = {4, 5, 6, 7, 8}

# Union (all elements from both sets)
print(a | b)            # {1, 2, 3, 4, 5, 6, 7, 8}
print(a.union(b))       # Same as above

# Intersection (common elements)
print(a & b)            # {4, 5}
print(a.intersection(b)) # Same as above

# Difference (in a but not in b)
print(a - b)            # {1, 2, 3}
print(a.difference(b))  # Same as above

# Symmetric difference (in either, but not both)
```

📝 Strings: Text Processing

Strings are sequences of characters.

```
# Creating strings
single = 'Hello'
double = "World"
multi = """This is a
multi-line
string"""

# String concatenation
greeting = "Hello" + " " + "World"  # Hello World
repeated = "Ha" * 3                  # HaHaHa

# String formatting
name = "Alice"
age = 30

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

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

String Methods

Common string operations.

```
text = "  Hello, World!  "

# Case conversion
print(text.lower())          # "  hello, world!  "
print(text.upper())          # "  HELLO, WORLD!  "
print(text.title())          # "  Hello, World!  "
print(text.capitalize())     # "  hello, world!  "
print(text.swapcase())       # "  hELLO, wORLD!  "

# Whitespace
print(text.strip())          # "Hello, World!" (both ends)
print(text.lstrip())         # "Hello, World!  " (left)
print(text.rstrip())         # "  Hello, World!" (right)

# Replacement
print(text.replace("World", "Python"))  # "  Hello, Python!  "
```

🔍 String Searching and Checking

Find and validate string content.

```
text = "Python Programming"

# Searching
print(text.find("Programming"))      # 7 (index of first occurrence)
print(text.find("Java"))             # -1 (not found)
print(text.index("Python"))          # 0 (raises error if not found)
print(text.count("m"))               # 2 (number of occurrences)

# Checking content
print(text.startswith("Python"))     # True
print(text.endswith("ing"))          # True
print("Hello" in text)               # False
print("gram" in text)                # True

# Character type checking
print("123".isdigit())               # True
print("abc".isalpha())               # True
```

✂️ String Slicing and Splitting

Extract and break apart strings.

```
text = "Python Programming"

# Slicing (same as lists)
print(text[0:6])      # "Python"
print(text[7:])       # "Programming"
print(text[:6])       # "Python"
print(text[::2])      # "Pto rgamn" (every 2nd char)
print(text[::-1])     # "gnimmargorP nohtyP" (reverse)

# Splitting
sentence = "The quick brown fox"
words = sentence.split()          # ['The', 'quick', 'brown', 'fox']
words = sentence.split(" ")       # Same as above

csv_data = "apple,banana,cherry"
fruits = csv_data.split(",")      # ['apple', 'banana', 'cherry']

# Joining
words = ["Hello", "World"]
```

✨ List Comprehensions

Create lists in a concise, readable way.

Basic Syntax

```
# Traditional way
squares = []
for x in range(10):
    squares.append(x**2)

# List comprehension (better!)
squares = [x**2 for x in range(10)]
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# General form: [expression for item in iterable]
```

More Examples

🎯 Conditional List Comprehensions

Add filtering to list comprehensions.

```
# Basic filtering
# [expression for item in iterable if condition]

# Even numbers only
evens = [x for x in range(20) if x % 2 == 0]
# [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

# Positive numbers only
numbers = [-2, -1, 0, 1, 2, 3]
positives = [x for x in numbers if x > 0]
# [1, 2, 3]

# Long words only
words = ["hi", "hello", "hey", "greetings"]
long_words = [word for word in words if len(word) > 3]
# ['hello', 'greetings']
```

Advanced List Comprehensions

More complex patterns and transformations.

```
# If-else in comprehension
# [true_expr if condition else false_expr for item in iterable]
labels = ["even" if x % 2 == 0 else "odd" for x in range(6)]
# ['even', 'odd', 'even', 'odd', 'even', 'odd']

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

# Flatten nested list
nested = [[1, 2], [3, 4], [5, 6]]
flat = [num for sublist in nested for num in sublist]
# [1, 2, 3, 4, 5, 6]

# Dictionary comprehension
squares_dict = {x: x**2 for x in range(5)}
# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}

# Set comprehension
```

File Handling: Reading Files

Work with external files to read data.

```
# Reading entire file
with open("data.txt", "r") as file:
    content = file.read()
    print(content)

# Reading line by line
with open("data.txt", "r") as file:
    for line in file:
        print(line.strip())  # strip() removes newline

# Reading all lines into a list
with open("data.txt", "r") as file:
    lines = file.readlines()  # ['line1\n', 'line2\n', ...]

# Reading first few lines
with open("data.txt", "r") as file:
```

📝 File Handling: Writing Files

Create and modify file content.

```
# Writing to a file (overwrites existing content)
with open("output.txt", "w") as file:
    file.write("Hello, World!\n")
    file.write("This is a new line.")

# Appending to a file
with open("output.txt", "a") as file:
    file.write("\nAppended line.")

# Writing multiple lines
lines = ["First line\n", "Second line\n", "Third line\n"]
with open("output.txt", "w") as file:
    file.writelines(lines)

# Writing formatted data
data = {"name": "Alice", "age": 30}
with open("data.txt", "w") as file:
```

🔄 File Modes

Different ways to open files.

```
# "r" - Read (default)
# Opens for reading, error if file doesn't exist
file = open("data.txt", "r")

# "w" - Write
# Creates new file or overwrites existing
file = open("data.txt", "w")

# "a" - Append
# Adds to end of file, creates if doesn't exist
file = open("data.txt", "a")

# "r+" - Read and Write
# Opens for both, error if doesn't exist
file = open("data.txt", "r+")

# "b" - Binary mode (for images, etc.)
```

📊 Working with CSV Files

Read and write comma-separated values.

```
# Writing CSV
with open("data.csv", "w") as file:
    file.write("Name,Age,City\n")
    file.write("Alice,30,New York\n")
    file.write("Bob,25,Los Angeles\n")

# Reading CSV manually
with open("data.csv", "r") as file:
    for line in file:
        values = line.strip().split(",")
        print(values)

# Using csv module (better!)
import csv

# Writing with csv module
with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
```

⚠ Exception Handling: Try-Except

Handle errors gracefully without crashing.

Why Handle Exceptions?

Programs encounter errors: file not found, invalid input, network issues. Exception handling lets your program respond to errors instead of crashing.

```
# Without exception handling (crashes!)
number = int("abc")  # ValueError: invalid literal

# With exception handling
try:
    number = int("abc")
    print("Conversion successful")
except ValueError:
    print("Invalid input! Please enter a number.")
    number = 0
```

🎯 Catching Specific Exceptions

Handle different types of errors differently.

```
try:
    file = open("nonexistent.txt", "r")
    content = file.read()
    number = int(content)
    result = 10 / number
except FileNotFoundError:
    print("Error: File not found!")
except ValueError:
    print("Error: File content is not a valid number!")
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
except Exception as e:
    # Catch any other exception
    print(f"Unexpected error: {e}")

print("Program continues...")
```

🔄 Try-Except-Else-Finally

Complete exception handling structure.

```
filename = "data.txt"

try:
    file = open(filename, "r")
    content = file.read()
    number = int(content)
except FileNotFoundError:
    print(f"Error: {filename} not found")
except ValueError:
    print("Error: Content is not a valid number")
else:
    # Runs only if NO exception occurred
    print(f"Successfully read number: {number}")
    result = number * 2
    print(f"Result: {result}")
finally:
    # ALWAYS runs, whether exception occurred or not
    try:
        file.close()
        print("File closed")
```

Raising Exceptions

Create your own exceptions when needed.

```
def divide(a, b):
    if b == 0:
        raise ValueError("Divisor cannot be zero!")
    return a / b

try:
    result = divide(10, 0)
except ValueError as e:
    print(f"Error: {e}")

# Validating input
def set_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative!")
    if age > 150:
        raise ValueError("Age must be realistic!")
    return age

try:
    user_age = set_age(-5)
```

📦 Modules: Code Organization

Modules are files containing Python code that you can reuse.

Why Use Modules?

- **Organization:** Group related functions together
- **Reusability:** Use code across multiple programs
- **Namespace:** Avoid naming conflicts
- **Maintainability:** Easier to update and debug

Built-in Modules

Python comes with many useful modules:

Importing Modules

Different ways to import and use modules.

```
# Import entire module
import math
print(math.sqrt(16))        # 4.0
print(math.pi)              # 3.141592653589793
print(math.pow(2, 3))       # 8.0

# Import specific functions
from math import sqrt, pi
print(sqrt(16))             # No need for math. prefix
print(pi)

# Import with alias
import math as m
print(m.sqrt(25))           # 5.0

# Import multiple items
from math import sqrt, pow, pi, e
```

The Random Module

Generate random numbers and make random choices.

```
import random

# Random float between 0 and 1
print(random.random())          # 0.7436582691...

# Random integer in range
print(random.randint(1, 10))    # Random number from 1 to 10

# Random float in range
print(random.uniform(1.5, 6.5)) # Random float from 1.5 to 6.5

# Choose from a list
colors = ["red", "green", "blue", "yellow"]
print(random.choice(colors))    # Random color

# Shuffle a list
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print(numbers)                  # [3, 1, 5, 2, 4] (random order)
```

The Datetime Module

Work with dates and times.

```python
from datetime import datetime, date, time, timedelta

# Current date and time
now = datetime.now()
print(now)                    # 2025-10-09 14:30:45.123456

today = date.today()
print(today)                  # 2025-10-09

# Creating specific dates
birthday = date(1995, 5, 15)
print(birthday)               # 1995-05-15

# Formatting dates
formatted = now.strftime("%Y-%m-%d %H:%M:%S")
print(formatted)              # "2025-10-09 14:30:45"

# Date arithmetic
tomorrow = today + timedelta(days=1)
next_week = today + timedelta(weeks=1)
```

📁 The OS Module

Interact with the operating system.

```
import os

# Current working directory
print(os.getcwd())              # /home/user/projects

# Change directory
os.chdir("/home/user/documents")

# List files in directory
files = os.listdir(".")
print(files)

# Check if path exists
if os.path.exists("data.txt"):
    print("File exists!")

# Create directory
os.mkdir("new_folder")
os.makedirs("path/to/nested/folder")  # Create nested directories

# File operations
os.rename("old.txt", "new.txt")
os.remove("file.txt")
os.rmdir("empty_folder")

# Path operations
```

📝 Creating Your Own Modules

Organize your code into reusable modules.

mymodule.py

```
# mymodule.py
def greet(name):
    """Greet someone by name."""
    return f"Hello, {name}!"

def add(a, b):
    """Add two numbers."""
    return a + b

PI = 3.14159

class Calculator:
    def multiply(self, a, b):
```

📦 Module Best Practices

Tips for working with modules effectively.

```
# 1. Use descriptive module names
# Good: calculator.py, user_auth.py
# Bad: util.py, stuff.py, functions.py

# 2. Import at the top of the file
import math
import random
from datetime import datetime

# 3. Use specific imports when possible
from math import sqrt, pi  # Better than import *

# 4. Group imports
# Standard library
import os
import sys

# Third-party
import numpy as np
import pandas as pd

# Local modules
import mymodule
from my_package import helper

# 5. Add a docstring to your module
"""
```

🎓 Congratulations!

You've learned the fundamental concepts of Python programming:

- ✅ **Variables & Data Types** - Storing information
- ✅ **Operators** - Performing calculations
- ✅ **Control Flow** - Making decisions
- ✅ **Loops** - Repeating actions
- ✅ **Functions** - Reusable code blocks
- ✅ **Data Structures** - Organizing data effectively
- ✅ **String Manipulation** - Working with text
- ✅ **List Comprehensions** - Elegant list creation
- ✅ **File Handling** - Reading and writing files
- ✅ **Exception Handling** - Graceful error management
- ✅ **Modules** - Code organization and reusability

🚀 Next Steps

Keep Learning:

- **Object-Oriented Programming** (Classes, Inheritance)
- **Advanced Data Structures** (Deque, Counter, defaultdict)
- **Decorators and Generators**
- **Regular Expressions**
- **Working with APIs**
- **Web Development** (Django, Flask)
- **Data Science** (NumPy, Pandas, Matplotlib)

Practice Resources: