Python Programming Bootcamp
Week 3: Functions & Modules

Hilal Khan

Center for Artificial Intelligence & Emerging Technologies

Learning Objectives

By the end of this week, you will be able to:

- Define and call functions with various parameter types
- Design reusable functions with proper scope
- Understand local vs global scope and the LEGB rule
- Understand the module system and code organization
- Import and use modules effectively
- Write proper documentation with docstrings
- Use common built-in functions
- Implement recursive functions with base and recursive cases

Week Overview

Topics we'll cover this week:

- Function Basics - Definition, calling, and return values
- Parameters and Arguments - Positional, keyword, default, *args, **kwargs
- Variable Scope - Local, global, and the LEGB rule
- Docstrings - Documenting your code properly
- Modules and Packages - Organizing and reusing code
- Import Statements - Different ways to import
- Built-in Functions - Common Python functions
- Recursion - Functions that call themselves

Introduction to Functions

What is a Function?

A function is a reusable block of code that performs a specific task.

Why use functions?

- **Code Reusability:** Write once, use many times
- **Organization:** Break complex problems into smaller pieces
- **Readability:** Make code easier to understand
- **Maintainability:** Fix bugs in one place
- **Testing:** Test individual components

Think of functions as mini-programs within your program!

Functions We've Already Used

You've been using functions all along!

Built-in Functions:

- `print()` - Displays output
- `input()` - Gets user input
- `len()` - Returns length
- `type()` - Returns data type
- `range()` - Generates sequences
- `int()`, `float()`, `str()` - Type conversion

Now you'll learn to create your own functions!

Function Basics

Defining a Function i

Use the `def` keyword to define a function:

```
# Basic function syntax
def function_name():
    # Function body (indented)
    # Code to execute
    pass
```

Simple example:

```
def greet():
    print("Hello!")
    print("Welcome to Python functions!")
```

Key points:

Defining a Function ii

- Function names follow variable naming rules (lowercase, underscores)
- Parentheses `()` are required
- Colon `:` at the end
- Body must be indented

To use a function, you need to call it:

```
# Define the function
def greet():
    print("Hello!")
    print("Welcome to Python!")

# Call the function
greet()
# Output:
# Hello!
# Welcome to Python!

# Call it again
greet()
# Output:
```

Calling a Function ii

```
# Hello!
# Welcome to Python!
```

Note: Defining a function doesn't execute it - you must call it!

Functions with Return Values i

Functions can return values using the `return` keyword:

```python
def add_numbers():
    result = 5 + 3
    return result

# Capture the return value
answer = add_numbers()
print(answer)  # Output: 8

# Use directly in expressions
total = add_numbers() + 10
print(total)   # Output: 18
```

Key points:

Functions with Return Values ii

- `return` sends a value back to the caller
- Function execution stops at `return`
- Functions without `return` return `None`

Return vs Print i

Common beginner confusion - return and print are different!

```
# Using print - displays but doesn't return
def greet_print():
    print("Hello!")
    # Returns None by default

result1 = greet_print()  # Displays: Hello!
print(result1)            # Output: None

# Using return - sends value back
def greet_return():
    return "Hello!"

result2 = greet_return()  # Doesn't display anything
print(result2)            # Output: Hello!
```

Return vs Print ii

Rule of thumb: Use `return` to give values back, `print` to display.

Multiple Return Statements i

Functions can have multiple return statements:

```python
def check_age(age):
    if age >= 18:
        return "Adult"
    else:
        return "Minor"

status1 = check_age(25)
print(status1)  # Output: Adult

status2 = check_age(15)
print(status2)  # Output: Minor
```

Only ONE return statement executes - the function exits immediately!

Returning Multiple Values i

Python can return multiple values as a tuple:

```
def get_min_max(numbers):
    minimum = min(numbers)
    maximum = max(numbers)
    return minimum, maximum

# Unpack the returned values
min_val, max_val = get_min_max([5, 2, 9, 1, 7])
print(f"Min: {min_val}, Max: {max_val}")
# Output: Min: 1, Max: 9

# Or capture as tuple
result = get_min_max([5, 2, 9, 1, 7])
print(result)  # Output: (1, 9)
```

Returning Multiple Values ii

This is actually returning a tuple, which we can unpack!

Parameters and Arguments

Parameters vs Arguments

Important terminology:

Parameters: Variables listed in the function definition

- The "placeholder" names

Arguments: Actual values passed when calling the function

- The "real" data

Example:

- `def greet(name):` - `name` is a parameter
- `greet("Alice")` - `"Alice"` is an argument

Positional Parameters i

Parameters are matched by position:

```python
def introduce(name, age, city):
    print(f"My name is {name}")
    print(f"I am {age} years old")
    print(f"I live in {city}")

# Arguments match parameters by position
introduce("Alice", 25, "New York")
# Output:
# My name is Alice
# I am 25 years old
# I live in New York

# Order matters!
introduce("New York", "Alice", 25)
```

Positional Parameters ii

```
# Output:
# My name is New York
# I am Alice years old
# I live in 25
```

Keyword Arguments i

Specify arguments by parameter name:

```python
def introduce(name, age, city):
    print(f"My name is {name}")
    print(f"I am {age} years old")
    print(f"I live in {city}")

# Use parameter names - order doesn't matter
introduce(city="Paris", name="Bob", age=30)
# Output:
# My name is Bob
# I am 30 years old
# I live in Paris

# Mix positional and keyword (positional first!)
introduce("Charlie", age=28, city="London")
```

Keyword Arguments ii

More readable and less error-prone!

Default Parameters i

Provide default values for parameters:

```python
def greet(name, greeting="Hello"):
    print(f"{greeting}, {name}!")

# Use default greeting
greet("Alice")
# Output: Hello, Alice!

# Override default
greet("Bob", "Hi")
# Output: Hi, Bob!

# Override with keyword
greet("Charlie", greeting="Hey")
# Output: Hey, Charlie!
```

Default Parameters ii

Default parameters must come after non-default parameters!

Common Default Parameter Patterns i

Useful examples with default parameters:

```python
def power(base, exponent=2):
    return base ** exponent

print(power(5))       # Output: 25 (5^2)
print(power(5, 3))    # Output: 125 (5^3)

def create_user(username, role="user", active=True):
    return {
        "username": username,
        "role": role,
        "active": active
    }

user1 = create_user("alice")
```

Common Default Parameter Patterns ii

```
print(user1)
# Output: {'username': 'alice', 'role': 'user', '
    active': True}

admin = create_user("bob", role="admin")
print(admin)
# Output: {'username': 'bob', 'role': 'admin', 'active
    ': True}
```

Variable-Length Arguments: *args i

Accept any number of positional arguments:

```
def sum_all(*numbers):
    total = 0
    for num in numbers:
        total += num
    return total

print(sum_all(1, 2, 3))              # Output: 6
print(sum_all(10, 20, 30, 40))       # Output: 100
print(sum_all(5))                     # Output: 5

# *args creates a tuple
def show_args(*args):
    print(type(args))  # <class 'tuple'>
    print(args)
```

Variable-Length Arguments: *args ii

```
show_args(1, 2, 3, 4)
# Output:
# <class 'tuple'>
# (1, 2, 3, 4)
```

Variable-Length Arguments: **kwargs i

Accept any number of keyword arguments:

```python
def print_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

print_info(name="Alice", age=25, city="NYC")
# Output:
# name: Alice
# age: 25
# city: NYC

# **kwargs creates a dictionary
def show_kwargs(**kwargs):
    print(type(kwargs))  # <class 'dict'>
    print(kwargs)
```

Variable-Length Arguments: **kwargs ii

```
show_kwargs(x=1, y=2, z=3)
# Output:
# <class 'dict'>
# {'x': 1, 'y': 2, 'z': 3}
```

You can combine different parameter types:

```
def complex_function(pos1, pos2, default="test", *args
    , **kwargs):
    print(f"Positional 1: {pos1}")
    print(f"Positional 2: {pos2}")
    print(f"Default: {default}")
    print(f"Extra positional: {args}")
    print(f"Extra keyword: {kwargs}")

complex_function(1, 2, "hello", 3, 4, 5, x=10, y=20)
# Output:
# Positional 1: 1
# Positional 2: 2
# Default: hello
# Extra positional: (3, 4, 5)
```

Combining All Parameter Types ii

```
# Extra keyword: {'x': 10, 'y': 20}
```

Order matters: positional, default, *args, **kwargs

Recursion

What is Recursion?

Recursion
A function that calls itself to solve a problem by breaking it into smaller subproblems

Key Components:

- **Base Case:** Condition where recursion stops
- **Recursive Case:** Function calls itself with modified parameters

When to use recursion:

- Problems that can be divided into similar subproblems
- Tree or graph traversal
- Mathematical sequences (Fibonacci, factorial)
- Divide and conquer algorithms

Simple Recursion: Countdown i

Example: Countdown from n to 0

```python
def countdown(n):
    """Recursively count down from n to 0"""
    if n <= 0:
        print("Blastoff!")
    else:
        print(n)
        countdown(n - 1)  # Recursive call

# Call the function
countdown(5)
```

Output:

Simple Recursion: Countdown ii

```
5
4
3
2
1
Blastoff!
```

How it works:

- Base case: `n <= 0` stops recursion
- Recursive case: Print `n`, call `countdown(n-1)`

Factorial Using Recursion i

Mathematical definition:

- $n! = n \times (n-1) \times (n-2) \times \ldots \times 1$
- $0! = 1$ and $1! = 1$

```python
def factorial(n):
    """
    Calculate factorial recursively
    Base case: 0! = 1, 1! = 1
    Recursive case: n! = n * (n-1)!
    """
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Factorial Using Recursion ii

```python
# Examples
print(factorial(5))  # 120
print(factorial(7))  # 5040
```

Recursion trace for factorial(5):

- `factorial(5) = 5 * factorial(4)`
- `factorial(4) = 4 * factorial(3)`
- `factorial(3) = 3 * factorial(2)`
- `factorial(2) = 2 * factorial(1)`
- `factorial(1) = 1` (base case)
- Result: $5 \times 4 \times 3 \times 2 \times 1 = 120$

Fibonacci Sequence i

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Definition:

- $F(0) = 0$
- $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$ for $n \geq 2$

```python
def fibonacci(n):
    """Calculate nth Fibonacci number recursively"""
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Fibonacci Sequence ii

```python
# First 10 Fibonacci numbers
for i in range(10):
    print(f"F({i}) = {fibonacci(i)}")
```

Output:

```
F(0) = 0
F(1) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
F(7) = 13
F(8) = 21
F(9) = 34
```

Fibonacci Sequence iii

Sum of List Using Recursion i

```python
def sum_list(numbers):
    """Calculate sum of list recursively"""
    if len(numbers) == 0:
        return 0  # Base case: empty list
    else:
        # Recursive case: first element + sum of rest
        return numbers[0] + sum_list(numbers[1:])

# Example
numbers = [1, 2, 3, 4, 5]
print(f"Sum of {numbers}: {sum_list(numbers)}")  # 15
```

How it works:

- `sum_list([1,2,3,4,5]) = 1 + sum_list([2,3,4,5])`

Sum of List Using Recursion ii

- `sum_list([2,3,4,5]) = 2 + sum_list([3,4,5])`
- `sum_list([3,4,5]) = 3 + sum_list([4,5])`
- `sum_list([4,5]) = 4 + sum_list([5])`
- `sum_list([5]) = 5 + sum_list([])`
- `sum_list([]) = 0` (base case)
- Result: $1 + 2 + 3 + 4 + 5 + 0 = 15$

Power Function Using Recursion i

```python
def power(base, exponent):
    """Calculate base^exponent recursively"""
    if exponent == 0:
        return 1  # Base case: any number^0 = 1
    else:
        return base * power(base, exponent - 1)

# Examples
print(f"2^5 = {power(2, 5)}")  # 32
print(f"3^4 = {power(3, 4)}")  # 81
print(f"5^0 = {power(5, 0)}")  # 1
```

Euclidean Algorithm using recursion:

```python
def gcd(a, b):
    """
    Calculate GCD using Euclidean algorithm
    GCD(a, b) = GCD(b, a mod b)
    Base case: GCD(a, 0) = a
    """
    if b == 0:
        return a  # Base case
    else:
        return gcd(b, a % b)  # Recursive case

# Examples
print(f"GCD(48, 18) = {gcd(48, 18)}")  # 6
print(f"GCD(100, 35) = {gcd(100, 35)}")  # 5
```

```python
print(f"GCD(17, 19) = {gcd(17, 19)}")  # 1
```

Trace for GCD(48, 18):

- `gcd(48, 18) = gcd(18, 48 % 18) = gcd(18, 12)`
- `gcd(18, 12) = gcd(12, 18 % 12) = gcd(12, 6)`
- `gcd(12, 6) = gcd(6, 12 % 6) = gcd(6, 0)`
- `gcd(6, 0) = 6` (base case)

Recursion vs Iteration

Recursion:

- + Elegant and clean code
- + Natural for tree/graph problems
- + Easier to understand for some problems
- – Uses more memory (call stack)
- – Can be slower
- – Risk of stack overflow

Iteration:

- + More memory efficient
- + Generally faster
- + No stack overflow risk
- – Can be more complex
- – Less intuitive for some problems

Best Practice

Use recursion when the problem is naturally recursive (trees, graphs).
Use iteration for simple loops and when performance is critical.

1. **Always define a base case:**

```python
# Bad - No base case, infinite recursion!
def bad_countdown(n):
    print(n)
    bad_countdown(n - 1)

# Good - Has base case
def good_countdown(n):
    if n <= 0:  # Base case
        print("Done!")
    else:
        print(n)
        good_countdown(n - 1)
```

Recursion Best Practices ii

2. **Make sure recursive case moves towards base case:**
    - Each recursive call should get closer to the base case
    - Otherwise, you'll have infinite recursion
3. **Be aware of performance:**
    - Recursive solutions can be inefficient (e.g., Fibonacci)
    - Consider memoization or iteration for better performance

Variable Scope

What is Scope?

Scope determines where a variable can be accessed.

Python has four scopes (LEGB Rule):

1. **Local (L):** Inside the current function
2. **Enclosing (E):** Inside enclosing functions (nested)
3. **Global (G):** At the top level of the module
4. **Built-in (B):** Python's built-in names

Python searches for variables in LEGB order!

Local Scope i

Variables created inside a function are local:

```python
def my_function():
    local_var = "I'm local"
    print(local_var)  # Works fine

my_function()
# Output: I'm local

# Try to access outside
print(local_var)  # Error!
# NameError: name 'local_var' is not defined
```

Local variables:

- Exist only inside the function

Local Scope ii

- Created when function is called
- Destroyed when function ends

Global Scope i

Variables created outside functions are global:

```python
global_var = "I'm global"

def my_function():
    print(global_var)  # Can access global

my_function()
# Output: I'm global

print(global_var)  # Also works here
# Output: I'm global

# Function can't modify global by default
def try_modify():
```

Global Scope ii

```
    global_var = "Modified"  # Creates NEW local
    variable!
    print(global_var)

try_modify()    # Output: Modified
print(global_var)  # Output: I'm global (unchanged!)
```

The global Keyword i

Use `global` to modify global variables:

```python
counter = 0

def increment():
    global counter  # Declare we're using global counter
    counter += 1
    print(f"Counter: {counter}")

increment()  # Output: Counter: 1
increment()  # Output: Counter: 2
increment()  # Output: Counter: 3

print(counter)  # Output: 3
```

The global Keyword ii

Best practice: Avoid global variables when possible!

- Use function parameters and return values instead
- Makes code more predictable and testable

Local vs Global Example i

Understanding the difference:

```python
x = "global"

def test():
    x = "local"  # New local variable
    print(f"Inside function: {x}")

test()
# Output: Inside function: local

print(f"Outside function: {x}")
# Output: Outside function: global

# Different variables with same name!
```

Local vs Global Example ii

Local variable shadows (hides) the global one inside the function.

Enclosing Scope (Nested Functions) i

Functions can be defined inside other functions:

```
def outer():
    outer_var = "I'm from outer"

    def inner():
        print(outer_var)  # Can access outer's
    variables

    inner()

outer()
# Output: I'm from outer

# Can't call inner from outside
inner()  # Error! NameError
```

Enclosing Scope (Nested Functions) ii

Inner functions have access to variables from enclosing functions.

LEGB Rule in Action i

Python searches for variables in LEGB order:

```python
x = "global x"

def outer():
    x = "outer x"

    def inner():
        x = "inner x"
        print(x)  # Which x?

    inner()
    print(x)

outer()
print(x)
```

LEGB Rule in Action ii

```
# Output:
# inner x   (Local wins)
# outer x   (Enclosing scope)
# global x  (Global scope)
```

Docstrings and Documentation

What are Docstrings?

Docstrings document your functions, classes, and modules.

Why use docstrings?

- Explain what your function does
- Describe parameters and return values
- Provide usage examples
- Help other developers (including future you!)
- Used by help() function
- Used by documentation generators

Good documentation is as important as good code!

Writing Docstrings i

Use triple quotes immediately after function definition:

```python
def calculate_area(radius):
    """Calculate the area of a circle.

    Args:
        radius: The radius of the circle

    Returns:
        The area of the circle
    """
    return 3.14159 * radius ** 2

# Access docstring with help()
help(calculate_area)
```

Writing Docstrings ii

```python
# Or with __doc__
print(calculate_area.__doc__)
```

Comprehensive Docstring Example i

Best practice format:

```
def divide(numerator, denominator):
    """Divide two numbers.

    This function divides the numerator by the
    denominator
    and returns the result. It handles division by
    zero.

    Args:
        numerator (float): The number to be divided
        denominator (float): The number to divide by

    Returns:
        float: The result of division
```

Comprehensive Docstring Example ii

```python
    Raises:
        ValueError: If denominator is zero

    Examples:
        >>> divide(10, 2)
        5.0
        >>> divide(7, 3)
        2.3333333333333335
    """
    if denominator == 0:
        raise ValueError("Cannot divide by zero")
    return numerator / denominator
```

Modules and Packages

What are Modules?

A module is a file containing Python code.

Benefits of modules:

- **Organization:** Group related code together
- **Reusability:** Use same code in multiple programs
- **Namespace:** Avoid naming conflicts
- **Maintainability:** Easier to find and fix bugs

Types of modules:

- **Built-in:** Come with Python (math, random, datetime)
- **Third-party:** Installed separately (requests, numpy, pandas)
- **Your own:** Python files you create

Creating Your Own Module i

Any Python file is a module!

Create a file named `mymath.py`:

```
# mymath.py
"""A simple math module."""

def add(x, y):
    """Add two numbers."""
    return x + y

def multiply(x, y):
    """Multiply two numbers."""
    return x * y

PI = 3.14159
```

Creating Your Own Module ii

```python
def circle_area(radius):
    """Calculate circle area."""
    return PI * radius ** 2
```

Now you can import and use it in other files!

The `if __name__ == "__main__":` Pattern i

Special variable `__name__` tells you how the file is being used:

- When you **run a file directly**: `__name__` = `"__main__"`
- When you **import a file as module**: `__name__` = module name

Why is this useful?

- Write code that only runs when file is executed directly
- Prevent code from running when module is imported
- Add test code to modules without affecting imports

Example: Module with Tests i

Create `calculator.py`:

```python
# calculator.py
"""A calculator module with built-in tests."""

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

def subtract(a, b):
    """Subtract b from a."""
    return a - b

def multiply(a, b):
    """Multiply two numbers."""
    return a * b
```

Example: Module with Tests ii

```python
# This code only runs when file is executed directly
if __name__ == "__main__":
    print("Testing calculator module...")
    print(f"5 + 3 = {add(5, 3)}")
    print(f"10 - 4 = {subtract(10, 4)}")
    print(f"6 * 7 = {multiply(6, 7)}")
    print("All tests passed!")
```

Scenario 1: Run file directly

```
$ python calculator.py
Testing calculator module...
5 + 3 = 8
10 - 4 = 6
6 * 7 = 42
All tests passed!
```

Using the Module ii

Scenario 2: Import as module

```python
# main.py
import calculator

result = calculator.add(10, 20)
print(result)  # Output: 30

# The test code does NOT run!
# No "Testing calculator module..." message
```

Key Point
When imported, only the functions are available. The test code under `if __name__ == "__main__":` doesn't execute!

Checking __name__ Value i

See it in action:

```python
# demo.py
print(f"__name__ is: {__name__}")

if __name__ == "__main__":
    print("This file is being run directly!")
else:
    print("This file is being imported as a module!")
```

Run directly:

```
$ python demo.py
__name__ is: __main__
This file is being run directly!
```

Checking __name__ Value ii

Import it:

```python
# another_file.py
import demo

# Output:
# __name__ is: demo
# This file is being imported as a module!
```

Best Practices

When to use `if __name__ == "__main__":`

- **Test code:** Add quick tests for your functions
- **Demo code:** Show how to use your module
- **Command-line scripts:** Code that should run when executed
- **Main program logic:** Keep module definitions separate from execution

Common Pattern
Put function/class definitions at the top of the file, then add `if __name__ == "__main__":` at the bottom with test or demo code.

Import Statements

Basic Import i

Import an entire module:

```python
# Import entire module
import math

# Use module.name syntax
print(math.pi)          # Output: 3.141592653589793
print(math.sqrt(16))    # Output: 4.0
print(math.pow(2, 3))   # Output: 8.0

# Import your own module
import mymath

print(mymath.add(5, 3))           # Output: 8
print(mymath.circle_area(5))      # Output: 78.53975
print(mymath.PI)                  # Output: 3.14159
```

Basic Import ii

Always use module name as prefix.

Import with Alias i

Give modules shorter names:

```python
# Import with alias
import math as m

print(m.pi)        # Output: 3.141592653589793
print(m.sqrt(25)) # Output: 5.0

# Common convention for popular libraries
import numpy as np       # Standard alias
import pandas as pd      # Standard alias
import matplotlib.pyplot as plt  # Standard alias

# Your own modules
import mymath as mm
print(mm.add(10, 20))  # Output: 30
```

Import with Alias ii

Use standard aliases for popular libraries!

Import Specific Items i

Import only what you need:

```
# Import specific functions
from math import pi, sqrt, pow

# Use directly without module prefix
print(pi)          # Output: 3.141592653589793
print(sqrt(16))    # Output: 4.0
print(pow(2, 3))   # Output: 8.0

# Import multiple items
from mymath import add, multiply, PI

print(add(5, 3))       # Output: 8
print(multiply(4, 7)) # Output: 28
print(PI)              # Output: 3.14159
```

Import Specific Items ii

```python
# Import with alias
from math import sqrt as square_root
print(square_root(25))  # Output: 5.0
```

Import All (Avoid This!) i

Import everything from a module:

```
# Import all (NOT recommended!)
from math import *

print(pi)     # Works
print(sqrt(9)) # Works

# Why avoid this?
# 1. Unclear where names come from
# 2. Can overwrite existing names
# 3. Pollutes namespace
# 4. Makes code harder to understand

# Example of problem:
from math import *
```

Import All (Avoid This!) ii

```python
from mymath import *

# Which PI? math.pi or mymath.PI?
print(PI)  # Confusing!
```

Best practice: Import specific names or use module prefix!

Useful Built-in Modules i

Python comes with many useful modules:

```
# math - Mathematical functions
import math
print(math.sqrt(16))    # 4.0
print(math.factorial(5)) # 120

# random - Random number generation
import random
print(random.randint(1, 10))     # Random int between
    1-10
print(random.choice(['a', 'b', 'c']))  # Random choice

# datetime - Date and time
from datetime import datetime
now = datetime.now()
```

```
print(now)  # Current date and time

# os - Operating system interface
import os
print(os.getcwd())  # Current directory
```

The random Module i

Generating random numbers and choices:

```python
import random

# Random integer
dice = random.randint(1, 6)
print(f"Dice roll: {dice}")

# Random float between 0 and 1
rand_num = random.random()
print(f"Random: {rand_num}")

# Random choice from list
colors = ['red', 'blue', 'green', 'yellow']
color = random.choice(colors)
print(f"Color: {color}")
```

The random Module ii

```python
# Shuffle a list
numbers = [1, 2, 3, 4, 5]
random.shuffle(numbers)
print(f"Shuffled: {numbers}")

# Random sample (without replacement)
sample = random.sample(range(1, 50), 6)
print(f"Lottery numbers: {sample}")
```

The math Module i

Common mathematical operations:

```python
import math

# Constants
print(math.pi)    # 3.141592653589793
print(math.e)     # 2.718281828459045

# Power and logarithm
print(math.pow(2, 10))   # 1024.0
print(math.log(100, 10)) # 2.0 (log base 10)

# Rounding
print(math.ceil(4.3))    # 5 (round up)
print(math.floor(4.7))   # 4 (round down)
```

The math Module ii

```
# Trigonometry
print(math.sin(math.pi/2))  # 1.0
print(math.cos(0))          # 1.0

# Other
print(math.factorial(5))  # 120
print(math.gcd(48, 18))   # 6 (greatest common divisor
    )
```

The datetime Module i

Working with dates and times:

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

# Current date and time
now = datetime.now()
print(now)  # 2024-01-15 14:30:45.123456

# Create specific date
birthday = date(2000, 5, 15)
print(birthday)  # 2000-05-15

# Extract components
print(now.year)    # 2024
print(now.month)   # 1
print(now.day)     # 15
```

The datetime Module ii

```python
print(now.hour)    # 14

# Format dates
formatted = now.strftime("%Y-%m-%d %H:%M")
print(formatted)  # 2024-01-15 14:30

# Date arithmetic
tomorrow = now + timedelta(days=1)
print(tomorrow)
```

Built-in Functions

Common Built-in Functions i

Python provides many useful built-in functions:

```python
# Type conversion
int("42")       # 42
float("3.14")  # 3.14
str(100)        # "100"
bool(1)         # True

# Math functions
abs(-5)         # 5
round(3.7)      # 4
pow(2, 3)       # 8
min(1, 2, 3)    # 1
max(1, 2, 3)    # 3
sum([1, 2, 3]) # 6
```

Common Built-in Functions ii

```python
# Sequence functions
len([1, 2, 3])       # 3
sorted([3, 1, 2])    # [1, 2, 3]
reversed([1, 2, 3]) # iterator
```

Functions for user interaction:

```
# Output
print("Hello")
print("Name:", "Alice", sep=" - ")
# Output: Name - Alice

print("Line 1", end=" ")
print("Line 2")
# Output: Line 1 Line 2

# Input
name = input("Enter your name: ")
print(f"Hello, {name}!")

# Reading numbers
```

Input/Output Functions ii

```python
age = int(input("Enter age: "))
price = float(input("Enter price: "))
```

Type Checking Functions i

Check types of objects:

```
# type() returns the type
print(type(42))          # <class 'int'>
print(type(3.14))        # <class 'float'>
print(type("hello"))     # <class 'str'>
print(type([1, 2]))      # <class 'list'>

# isinstance() checks if object is of a type
print(isinstance(42, int))          # True
print(isinstance(3.14, float))      # True
print(isinstance("hi", str))        # True
print(isinstance([1], list))        # True

# Check multiple types
print(isinstance(42, (int, float)))  # True
```

Type Checking Functions ii

Check conditions across sequences:

```
# all() - True if ALL elements are True
numbers = [2, 4, 6, 8]
print(all(n % 2 == 0 for n in numbers))  # True (all
    even)

numbers = [2, 4, 5, 8]
print(all(n % 2 == 0 for n in numbers))  # False (5 is
     odd)

# any() - True if ANY element is True
numbers = [1, 3, 4, 7]
print(any(n % 2 == 0 for n in numbers))  # True (4 is
    even)
```

```
numbers = [1, 3, 5, 7]
print(any(n % 2 == 0 for n in numbers))  # False (all
    odd)

# Empty checks
print(all([]))  # True (vacuous truth)
print(any([]))  # False
```

enumerate() and zip() i

Useful for iteration:

```python
# enumerate() - get index and value
fruits = ['apple', 'banana', 'cherry']
for index, fruit in enumerate(fruits):
    print(f"{index}: {fruit}")
# Output:
# 0: apple
# 1: banana
# 2: cherry

# Start from different index
for index, fruit in enumerate(fruits, start=1):
    print(f"{index}: {fruit}")

# zip() - combine multiple iterables
```

enumerate() and zip() ii

```python
names = ['Alice', 'Bob', 'Charlie']
ages = [25, 30, 35]
for name, age in zip(names, ages):
    print(f"{name} is {age} years old")
```

Practice Problems

Practice Problem 1: Temperature Converter i

Create a temperature conversion module:

```python
# temperature.py
def celsius_to_fahrenheit(celsius):
    """Convert Celsius to Fahrenheit."""
    return (celsius * 9/5) + 32

def fahrenheit_to_celsius(fahrenheit):
    """Convert Fahrenheit to Celsius."""
    return (fahrenheit - 32) * 5/9

def celsius_to_kelvin(celsius):
    """Convert Celsius to Kelvin."""
    return celsius + 273.15

# Test the module
```

Practice Problem 1: Temperature Converter ii

```python
if __name__ == "__main__":
    print(celsius_to_fahrenheit(0))    # 32.0
    print(fahrenheit_to_celsius(32))   # 0.0
    print(celsius_to_kelvin(0))        # 273.15
```

Practice Problem 2: Grade Calculator i

Function with error handling:

```python
def calculate_grade(scores):
    """Calculate average and letter grade.

    Args:
        scores: List of numeric scores

    Returns:
        Tuple of (average, letter_grade)
    """
    try:
        if not scores:
            return None, "No scores provided"

        average = sum(scores) / len(scores)
```

```
        if average >= 90:
            grade = 'A'
        elif average >= 80:
            grade = 'B'
        elif average >= 70:
            grade = 'C'
        elif average >= 60:
            grade = 'D'
        else:
            grade = 'F'

        return round(average, 2), grade

    except TypeError:
        return None, "Invalid score format"
```

Using lambda with map and filter:

```python
def process_numbers(numbers):
    """Process a list of numbers."""

    # Remove negative numbers
    positive = list(filter(lambda x: x >= 0, numbers))

    # Square all numbers
    squared = list(map(lambda x: x ** 2, positive))

    # Get even squares only
    even_squares = list(filter(lambda x: x % 2 == 0,
   squared))

    return even_squares
```

Practice Problem 3: List Processing ii

```python
# Test
numbers = [-2, -1, 0, 1, 2, 3, 4, 5]
result = process_numbers(numbers)
print(result)
# Output: [0, 4, 16]
# Explanation: 0^2=0, 2^2=4, 4^2=16 (all even)
```

Using *args and **kwargs:

```
def calculate(*args, operation="sum", **kwargs):
    """Flexible calculator function.

    Args:
        *args: Numbers to calculate
        operation: Type of operation (sum, product,
    average)
        **kwargs: Additional options
    """
    if not args:
        return 0

    if operation == "sum":
        result = sum(args)
```

Practice Problem 4: Flexible Calculator ii

```
    elif operation == "product":
        result = 1
        for num in args:
            result *= num
    elif operation == "average":
        result = sum(args) / len(args)
    else:
        return "Invalid operation"

    # Check if rounding requested
    if kwargs.get("round_result"):
        result = round(result, kwargs.get("decimals",
2))

    return result
```

Summary and Best Practices

Function Best Practices

Writing good functions:

- **Single Responsibility:** Each function should do ONE thing well
- **Descriptive Names:** Use clear, verb-based names
- **Keep it Short:** Aim for functions under 20 lines
- **Document:** Always write docstrings
- **Avoid Side Effects:** Don't modify global state
- **Return, Don't Print:** Let caller decide what to do
- **Use Type Hints:** Help others understand expectations
- **Handle Errors:** Use try-except appropriately
- **Test:** Write tests for your functions

Good vs Bad Function Examples i

```python
# BAD: Does too much, modifies global, no docstring
total = 0
def bad_function(x):
    global total
    print(x)
    total += x
    print(total)
    return x * 2

# GOOD: Single purpose, documented, pure function
def double(number):
    """Double the given number.

    Args:
        number: A numeric value
```

Good vs Bad Function Examples ii

```
    Returns:
        The number multiplied by 2
    """
    return number * 2
```

Module Organization Best Practices

Organizing your code:

- **One module = One purpose:** Group related functions
- **Descriptive filenames:** math_utils.py, not utils.py
- **Module docstring:** Explain what the module does
- **Organize imports:**
  1. Standard library imports
  2. Third-party imports
  3. Local imports
- **Use `__name__ == "__main__"`:** For testing code
- **Avoid circular imports:** A imports B, B imports A

Week 3 Summary

What we learned:

- **Functions:** Define with `def`, call with `()`
- **Parameters:** Positional, keyword, default, *args, **kwargs
- **Return:** Send values back with `return`
- **Scope:** Local vs global, LEGB rule
- **Docstrings:** Document your code with `"""`
- **Modules:** Organize code in separate files
- **Import:** Use others' code with `import`
- **Built-ins:** Many useful functions already available

Skills Checklist

You should now be able to:

- Create functions with various parameter types
- Use return statements effectively
- Understand variable scope and the LEGB rule
- Document functions with docstrings
- Create and import custom modules
- Use common built-in modules (math, random, datetime)
- Apply built-in functions appropriately

Next Week Preview

Week 4: File I/O & Error Handling

- Understanding exceptions and error types
- Try-except-else-finally blocks
- Raising exceptions with informative messages
- Reading and writing files
- Working with different file modes
- Context managers (with statement)
- Working with CSV and JSON files
- Combining error handling with file operations

Homework

- Complete all practice exercises
- Create utility functions for your own projects
- Practice writing clear docstrings
- Experiment with different modules

Thank You!

Thank you for your attention!

Keep Practicing!

Access Course Materials:

Download Course Materials

Center for Artificial Intelligence & Emerging Technologies