Complete Guide to Functions in Python

From Basic Definitions to Advanced Functional Programming

Table of Contents

Chapter 1: Introduction to Functions {#introduction}

What are Functions?

Functions are reusable blocks of code that perform specific tasks. They are fundamental building blocks in Python programming that help organize code, avoid repetition, and make programs more modular and maintainable.

Why Use Functions?

1. **Code Reusability**: Write once, use many times
2. **Modularity**: Break complex problems into smaller, manageable pieces
3. **Abstraction**: Hide implementation details
4. **Testing**: Easier to test individual components

5. **Readability**: Self-documenting code with meaningful function names
6. **Namespace Management**: Avoid variable name conflicts

Function Anatomy

python

```python
def function_name(parameters):
    """Docstring describing the function"""
    # Function body
    # Code to execute
    return result  # Optional return statement
```

Chapter 2: Basic Function Concepts {#basic-concepts}

Defining Functions

python

```python
# Simple function without parameters
def greet():
    print("Hello, World!")

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

# Function with return value
def add(a, b):
    return a + b

# Function with multiple statements
def calculate_area_of_circle(radius):
    """Calculate the area of a circle given its radius."""
    pi = 3.14159
    area = pi * radius ** 2
    return area
```

Calling Functions

python

```python
# Calling a function without arguments
greet()  # Output: Hello, World!

# Calling with arguments
greet_person("Alice")  # Output: Hello, Alice!

# Using return values
result = add(5, 3)
print(result)  # Output: 8

# Functions can be called within expressions
total = add(10, 20) + add(5, 5)
print(total)  # Output: 40
```

Function Documentation (Docstrings)

python

```python
def calculate_bmi(weight, height):
    """
    Calculate Body Mass Index (BMI).

    Args:
        weight (float): Weight in kilograms
        height (float): Height in meters

    Returns:
        float: BMI value

    Example:
        >>> calculate_bmi(70, 1.75)
        22.86
    """
    return weight / (height ** 2)

# Access docstring
print(calculate_bmi.__doc__)
help(calculate_bmi)
```

The `pass` Statement

python

```python
def placeholder_function():
    """This function does nothing yet."""
    pass  # Used as a placeholder for future implementation

# Useful for designing code structure
class MyClass:
    def method1(self):
        pass

    def method2(self):
        pass
```

Chapter 3: Parameters and Arguments {#parameters-arguments}

Positional Parameters

python

```python
def describe_pet(animal_type, pet_name):
    """Display information about a pet."""
    print(f"I have a {animal_type} named {pet_name}.")

# Order matters for positional arguments
describe_pet("hamster", "Harry")  # Correct
describe_pet("Harry", "hamster")  # Wrong order!
```

Keyword Arguments

python

```python
# Using keyword arguments
describe_pet(animal_type="hamster", pet_name="Harry")
describe_pet(pet_name="Harry", animal_type="hamster")  # Order doesn't matter

# Mixing positional and keyword arguments
describe_pet("hamster", pet_name="Harry")  # Valid
# describe_pet(animal_type="hamster", "Harry")  # SyntaxError!
```

Default Parameter Values

python

```python
def describe_pet(pet_name, animal_type="dog"):
    """Display information about a pet."""
    print(f"I have a {animal_type} named {pet_name}.")

# Using default value
describe_pet("Willie")  # Output: I have a dog named Willie.
describe_pet("Harry", "hamster")  # Override default

# Default values are evaluated once at function definition
def append_to_list(value, target=[]):  # Problematic!
    target.append(value)
    return target

list1 = append_to_list(1)  # [1]
list2 = append_to_list(2)  # [1, 2] - Unexpected!

# Better approach
def append_to_list(value, target=None):
    if target is None:
        target = []
    target.append(value)
    return target
```

Variable-Length Arguments (*args)

```python
def sum_all(*numbers):
    """Sum any number of arguments."""
    total = 0
    for num in numbers:
        total += num
    return total

print(sum_all(1, 2, 3))  # 6
print(sum_all(1, 2, 3, 4, 5))  # 15

# Combining with regular parameters
def greet_many(greeting, *names):
    """Greet multiple people."""
    for name in names:
        print(f"{greeting}, {name}!")

greet_many("Hello", "Alice", "Bob", "Charlie")

# Unpacking arguments
numbers = [1, 2, 3, 4, 5]
print(sum_all(*numbers))  # Unpack list
```

Keyword Variable-Length Arguments (**kwargs)

python

```python
def build_profile(first, last, **user_info):
    """Build a user profile dictionary."""
    profile = {
        'first_name': first,
        'last_name': last
    }
    profile.update(user_info)
    return profile

user = build_profile('albert', 'einstein',
                    location='princeton',
                    field='physics',
                    born=1879)

print(user)
# {'first_name': 'albert', 'last_name': 'einstein',
#  'location': 'princeton', 'field': 'physics', 'born': 1879}

# Unpacking dictionaries
extra_info = {'location': 'princeton', 'field': 'physics'}
user2 = build_profile('albert', 'einstein', **extra_info)
```

Parameter Order

python

```python
def complex_function(pos1, pos2,                    # Positional parameters
                    *args,                         # Variable positional
                    keyword1="default",            # Keyword parameters
                    keyword2="default",
                    **kwargs):                     # Variable keyword
    """Demonstrate parameter order."""
    print(f"Positional: {pos1}, {pos2}")
    print(f"Args: {args}")
    print(f"Keywords: {keyword1}, {keyword2}")
    print(f"Kwargs: {kwargs}")

complex_function(1, 2, 3, 4, 5,
                keyword1="changed",
                extra1="value1",
                extra2="value2")
```

Keyword-Only Arguments

python

```python
# Force certain arguments to be keyword-only
def calculate(a, b, *, operation='add', verbose=False):
    """Perform calculation with keyword-only operation."""
    if operation == 'add':
        result = a + b
    elif operation == 'multiply':
        result = a * b
    else:
        raise ValueError(f"Unknown operation: {operation}")

    if verbose:
        print(f"{a} {operation} {b} = {result}")

    return result

# Valid calls
calculate(5, 3)  # Uses default operation='add'
calculate(5, 3, operation='multiply')
calculate(5, 3, operation='add', verbose=True)

# Invalid call
# calculate(5, 3, 'multiply')  # TypeError!
```

Positional-Only Arguments (Python 3.8+)

python

```python
def greet(name, /, greeting="Hello"):
    """Name must be positional-only."""
    return f"{greeting}, {name}!"

# Valid calls
print(greet("Alice"))
print(greet("Bob", "Hi"))
print(greet("Charlie", greeting="Hey"))

# Invalid call
# print(greet(name="David"))  # TypeError!

# Combined positional-only and keyword-only
def full_example(pos_only, /, standard, *, kw_only):
    print(f"Positional only: {pos_only}")
    print(f"Standard: {standard}")
    print(f"Keyword only: {kw_only}")

full_example(1, 2, kw_only=3)  # Valid
full_example(1, standard=2, kw_only=3)  # Valid
# full_example(pos_only=1, standard=2, kw_only=3)  # TypeError!
```

Chapter 4: Return Values and Multiple Returns {#return-values}

Basic Return

python

```python
def square(x):
    """Return the square of a number."""
    return x ** 2

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

# Functions without explicit return return None
def no_return():
    print("This function returns None")

result = no_return()
print(result)  # None
```

Multiple Return Values

```python
def get_name_and_age():
    """Return multiple values as a tuple."""
    name = "Alice"
    age = 30
    return name, age  # Returns a tuple

# Unpacking return values
name, age = get_name_and_age()
print(f"{name} is {age} years old")

# Getting as tuple
result = get_name_and_age()
print(result)  # ('Alice', 30)

# More complex example
def analyze_list(numbers):
    """Return multiple statistics about a list."""
    if not numbers:
        return None, None, None, None

    return min(numbers), max(numbers), sum(numbers), len(numbers)

data = [1, 2, 3, 4, 5]
minimum, maximum, total, count = analyze_list(data)
average = total / count if count > 0 else 0
```

Early Returns

python

```python
def validate_age(age):
    """Validate age with early returns."""
    if not isinstance(age, (int, float)):
        return False, "Age must be a number"

    if age < 0:
        return False, "Age cannot be negative"

    if age > 150:
        return False, "Age seems unrealistic"

    return True, "Age is valid"

# Using the function
valid, message = validate_age(25)
if valid:
    print("Age accepted")
else:
    print(f"Error: {message}")
```

Returning Different Types

python

```python
def flexible_return(option):
    """Return different types based on input."""
    if option == 'string':
        return "Hello"
    elif option == 'number':
        return 42
    elif option == 'list':
        return [1, 2, 3]
    elif option == 'dict':
        return {'key': 'value'}
    elif option == 'function':
        return lambda x: x * 2
    else:
        return None

# Different return types
print(flexible_return('string'))    # "Hello"
print(flexible_return('number'))    # 42
func = flexible_return('function')
print(func(5))                      # 10
```

Named Tuples for Clarity

python

```python
from collections import namedtuple

# Define a named tuple
Result = namedtuple('Result', ['success', 'data', 'error'])

def fetch_data(id):
    """Return structured data using named tuple."""
    try:
        # Simulate fetching data
        if id < 0:
            raise ValueError("Invalid ID")

        data = f"Data for ID {id}"
        return Result(success=True, data=data, error=None)

    except Exception as e:
        return Result(success=False, data=None, error=str(e))

# Using the function
result = fetch_data(123)
if result.success:
    print(f"Data: {result.data}")
else:
    print(f"Error: {result.error}")
```

Chapter 5: Scope and Namespaces {#scope-namespaces}

Understanding Scope

```python
# Global scope
global_var = "I'm global"

def demonstrate_scope():
    # Function scope
    function_var = "I'm local to function"

    def inner_function():
        # Nested function scope
        inner_var = "I'm local to inner function"

        # Can access outer scopes
        print(global_var)       # Works
        print(function_var)     # Works
        print(inner_var)        # Works

    inner_function()
    # print(inner_var)  # NameError - not accessible here

demonstrate_scope()
# print(function_var)  # NameError - not accessible here
```

LEGB Rule

Python resolves names using the LEGB rule:

- **L**ocal: Inside the current function
- **E**nclosing: In the enclosing function (for nested functions)
- **G**lobal: At the top level of the module
- **B**uilt-in: In the built-in namespace

python

```python
# Built-in
# print, len, etc. are in built-in namespace

# Global
x = "global x"

def outer():
    # Enclosing
    x = "enclosing x"

    def inner():
        # Local
        x = "local x"
        print(f"Inner: {x}")  # local x

    inner()
    print(f"Outer: {x}")  # enclosing x

outer()
print(f"Global: {x}")  # global x
```

The `global` Keyword

python

```python
counter = 0

def increment():
    global counter
    counter += 1

def increment_wrong():
    # This creates a new local variable!
    counter = counter + 1  # UnboundLocalError

increment()
increment()
print(counter)  # 2

# Multiple global declarations
def modify_globals():
    global x, y, z
    x = 1
    y = 2
    z = 3

modify_globals()
print(x, y, z)  # 1 2 3
```

The `nonlocal` Keyword

python

```python
def outer():
    count = 0

    def inner():
        nonlocal count
        count += 1
        return count

    return inner

# Create closure
counter = outer()
print(counter())  # 1
print(counter())  # 2
print(counter())  # 3

# Without nonlocal
def outer_wrong():
    count = 0

    def inner():
        count = 1  # Creates new local variable
        return count

    inner()
    return count  # Still 0

print(outer_wrong())  # 0
```

Namespace Dictionaries

python

```
def explore_namespaces():
    """Explore namespace dictionaries."""
    local_var = 42

    print("Locals:", locals())
    print("Globals keys:", list(globals().keys())[:5])  # First 5 keys

    # Modifying namespace dictionaries (not recommended)
    locals()['dynamic_var'] = 100  # May not work as expected

    # Better approach for dynamic variables
    namespace = {}
    exec("dynamic_var = 100", namespace)
    print(namespace['dynamic_var'])

explore_namespaces()

# Built-in namespace
import builtins
print(dir(builtins)[:10])  # First 10 built-in names
```

Chapter 6: First-Class Functions {#first-class-functions}

Functions as Objects

python

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

# Functions are objects
print(type(greet))  # <class 'function'>
print(greet.__name__)  # 'greet'

# Assign function to variable
my_func = greet
print(my_func("Alice"))  # Hello, Alice!

# Functions can be stored in data structures
operations = {
    'add': lambda x, y: x + y,
    'subtract': lambda x, y: x - y,
    'multiply': lambda x, y: x * y,
    'divide': lambda x, y: x / y if y != 0 else None
}

result = operations['add'](5, 3)
print(result)  # 8
```

Functions as Arguments

python

```python
def apply_operation(func, x, y):
    """Apply a function to two arguments."""
    return func(x, y)

def add(a, b):
    return a + b

def multiply(a, b):
    return a * b

print(apply_operation(add, 5, 3))       # 8
print(apply_operation(multiply, 5, 3))  # 15

# More complex example
def process_list(items, processor):
    """Process each item in a list with a function."""
    return [processor(item) for item in items]

numbers = [1, 2, 3, 4, 5]
squared = process_list(numbers, lambda x: x ** 2)
print(squared)  # [1, 4, 9, 16, 25]

# Sorting with custom key function
students = [
    {'name': 'Alice', 'grade': 85},
    {'name': 'Bob', 'grade': 92},
    {'name': 'Charlie', 'grade': 78}
]

students.sort(key=lambda student: student['grade'], reverse=True)
print(students)
```

Functions Returning Functions

python

```python
def make_multiplier(n):
    """Create a function that multiplies by n."""
    def multiplier(x):
        return x * n
    return multiplier

# Create specialized functions
double = make_multiplier(2)
triple = make_multiplier(3)

print(double(5))   # 10
print(triple(5))   # 15

# More complex example
def make_formatter(prefix, suffix):
    """Create a custom formatting function."""
    def formatter(value):
        return f"{prefix}{value}{suffix}"
    return formatter

bold = make_formatter("**", "**")
italic = make_formatter("*", "*")
code = make_formatter("`", "`")

print(bold("Important"))    # **Important**
print(italic("emphasis"))   # *emphasis*
print(code("print()"))      # `print()`
```

Function Factories

python

```python
def create_validator(min_val=None, max_val=None, required=True):
    """Create a validation function with specific rules."""
    def validator(value):
        if required and value is None:
            return False, "Value is required"

        if value is None:
            return True, "OK"

        if min_val is not None and value < min_val:
            return False, f"Value must be >= {min_val}"

        if max_val is not None and value > max_val:
            return False, f"Value must be <= {max_val}"

        return True, "OK"

    return validator

# Create specific validators
age_validator = create_validator(min_val=0, max_val=150)
score_validator = create_validator(min_val=0, max_val=100)
optional_validator = create_validator(required=False)

# Use validators
valid, message = age_validator(25)
print(f"Age 25: {message}")  # OK

valid, message = age_validator(-5)
print(f"Age -5: {message}")  # Value must be >= 0
```

Chapter 7: Lambda Functions {#lambda-functions}

Lambda Basics

```python
# Regular function
def square(x):
    return x ** 2

# Equivalent lambda
square_lambda = lambda x: x ** 2

print(square(5))         # 25
print(square_lambda(5))  # 25

# Lambda with multiple parameters
add = lambda x, y: x + y
print(add(3, 4))  # 7

# Lambda with conditional expression
max_value = lambda x, y: x if x > y else y
print(max_value(5, 3))  # 5
```

Common Lambda Use Cases

python

```python
# 1. Sorting
students = [
    ('Alice', 85),
    ('Bob', 92),
    ('Charlie', 78)
]

# Sort by grade (second element)
students.sort(key=lambda x: x[1])
print(students)

# 2. Filtering
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens = list(filter(lambda x: x % 2 == 0, numbers))
print(evens)  # [2, 4, 6, 8, 10]

# 3. Mapping
squares = list(map(lambda x: x ** 2, numbers))
print(squares)  # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# 4. Reduce
from functools import reduce
product = reduce(lambda x, y: x * y, [1, 2, 3, 4, 5])
print(product)  # 120
```

Lambda Limitations and Alternatives

python

```python
# Lambdas are limited to single expressions
# This won't work:
# lambda x:
#     y = x + 1
#     return y * 2

# Use regular function instead
def complex_operation(x):
    y = x + 1
    return y * 2

# Lambdas can't contain statements
# This won't work:
# lambda x: print(x)  # print is fine in Python 3

# But this won't:
# lambda x: x = x + 1  # Assignment is a statement

# Lambda with default arguments
multiply = lambda x, y=2: x * y
print(multiply(5))     # 10
print(multiply(5, 3))  # 15
```

Advanced Lambda Patterns

```python
# Immediately Invoked Lambda
result = (lambda x: x ** 2)(5)
print(result)  # 25

# Lambda in comprehensions
squares = [(lambda x: x ** 2)(x) for x in range(5)]
print(squares)  # [0, 1, 4, 9, 16]

# Nested lambdas
add = lambda x: lambda y: x + y
add_five = add(5)
print(add_five(3))  # 8

# Lambda with *args and **kwargs
flexible = lambda *args, **kwargs: (args, kwargs)
print(flexible(1, 2, 3, name="Alice", age=30))
# ((1, 2, 3), {'name': 'Alice', 'age': 30})

# Using lambdas in dictionaries for dispatch
operations = {
    '+': lambda x, y: x + y,
    '-': lambda x, y: x - y,
    '*': lambda x, y: x * y,
    '/': lambda x, y: x / y if y != 0 else float('inf'),
    '**': lambda x, y: x ** y
}

def calculate(x, op, y):
    return operations.get(op, lambda x, y: None)(x, y)

print(calculate(5, '+', 3))   # 8
print(calculate(5, '**', 2))  # 25
```

Chapter 8: Decorators {#decorators}

Decorator Basics

```python
def my_decorator(func):
    """Basic decorator that adds behavior."""
    def wrapper():
        print("Something before the function")
        result = func()
        print("Something after the function")
        return result
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

# Equivalent to: say_hello = my_decorator(say_hello)

say_hello()
# Output:
# Something before the function
# Hello!
# Something after the function
```

Decorators with Arguments

python

```python
def repeat(times):
    """Decorator that repeats function execution."""
    def decorator(func):
        def wrapper(*args, **kwargs):
            results = []
            for _ in range(times):
                result = func(*args, **kwargs)
                results.append(result)
            return results
        return wrapper
    return decorator

@repeat(times=3)
def greet(name):
    print(f"Hello, {name}!")
    return f"Greeted {name}"

results = greet("Alice")
# Output:
# Hello, Alice!
# Hello, Alice!
# Hello, Alice!
print(results)  # ['Greeted Alice', 'Greeted Alice', 'Greeted Alice']
```

Preserving Function Metadata

```python
import functools

def my_decorator(func):
    """Decorator that preserves function metadata."""
    @functools.wraps(func)  # Preserves metadata
    def wrapper(*args, **kwargs):
        """Wrapper function"""
        return func(*args, **kwargs)
    return wrapper

@my_decorator
def example():
    """Example function."""
    pass

print(example.__name__)  # 'example' (not 'wrapper')
print(example.__doc__)   # 'Example function.'
```

Class Decorators

python

```python
def singleton(cls):
    """Class decorator to implement singleton pattern."""
    instances = {}

    def get_instance(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return get_instance

@singleton
class Database:
    def __init__(self):
        print("Creating database connection")
        self.connection = "Connected"

# Both create the same instance
db1 = Database()  # Creating database connection
db2 = Database()  # No output - returns existing instance
print(db1 is db2)  # True

# Decorator for adding methods to class
def add_debug_method(cls):
    """Add debug method to class."""
    def debug(self):
        attrs = ', '.join(f"{k}={v}" for k, v in self.__dict__.items())
        return f"{cls.__name__}({attrs})"

    cls.debug = debug
    return cls

@add_debug_method
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

p = Point(3, 4)
print(p.debug())  # Point(x=3, y=4)
```

Decorator Classes

python

```python
class CountCalls:
    """Decorator class that counts function calls."""
    def __init__(self, func):
        self.func = func
        self.count = 0

    def __call__(self, *args, **kwargs):
        self.count += 1
        print(f"{self.func.__name__} called {self.count} times")
        return self.func(*args, **kwargs)

@CountCalls
def say_hello():
    print("Hello!")

say_hello()  # say_hello called 1 times
say_hello()  # say_hello called 2 times

# Parameterized decorator class
class MaxRetries:
    """Decorator class with parameters."""
    def __init__(self, max_attempts=3):
        self.max_attempts = max_attempts

    def __call__(self, func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(self.max_attempts):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    if attempt == self.max_attempts - 1:
                        raise
                    print(f"Attempt {attempt + 1} failed: {e}")
            return None
        return wrapper

@MaxRetries(max_attempts=3)
def unreliable_function():
    import random
    if random.random() < 0.7:
        raise ValueError("Random failure")
    return "Success!"
```

Practical Decorators

python

```python
import time
import functools
from typing import Any, Callable
import logging

# 1. Timing decorator
def timeit(func: Callable) -> Callable:
    """Measure function execution time."""
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.perf_counter()
        result = func(*args, **kwargs)
        end = time.perf_counter()
        print(f"{func.__name__} took {end - start:.4f} seconds")
        return result
    return wrapper

# 2. Caching decorator
def memoize(func: Callable) -> Callable:
    """Cache function results."""
    cache = {}

    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        key = str(args) + str(kwargs)
        if key not in cache:
            cache[key] = func(*args, **kwargs)
        return cache[key]

    wrapper.cache_clear = lambda: cache.clear()
    return wrapper

# 3. Validation decorator
def validate_types(**expected_types):
    """Validate function argument types."""
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            # Get function signature
            import inspect
            sig = inspect.signature(func)
            bound_args = sig.bind(*args, **kwargs)

            # Validate types
            for param_name, expected_type in expected_types.items():
                if param_name in bound_args.arguments:
                    value = bound_args.arguments[param_name]
                    if not isinstance(value, expected_type):
                        raise TypeError(
                            f"{param_name} must be {expected_type.__name__}, "
                            f"got {type(value).__name__}"
                        )

            return func(*args, **kwargs)
        return wrapper
    return decorator

@validate_types(x=int, y=int)
def add(x, y):
    return x + y

# 4. Logging decorator
def log_calls(level=logging.INFO):
    """Log function calls."""
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            logger = logging.getLogger(func.__module__)
            logger.log(
                level,
                f"Calling {func.__name__} with args={args}, kwargs={kwargs}"
            )
            try:
                result = func(*args, **kwargs)
                logger.log(level, f"{func.__name__} returned {result}")
                return result
            except Exception as e:
                logger.exception(f"{func.__name__} raised {e}")
                raise
        return wrapper
    return decorator

# 5. Access control decorator
def require_permission(permission):
    """Check user permission before function execution."""
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            # Simulate permission check
            user_permissions = kwargs.get('user_permissions', [])
            if permission not in user_permissions:
                raise PermissionError(f"Permission '{permission}' required")
            return func(*args, **kwargs)
        return wrapper
    return decorator

@require_permission('admin')
def delete_user(user_id, user_permissions=None):
    return f"User {user_id} deleted"
```

Stacking Decorators

```python
# Multiple decorators are applied bottom-up
@timeit
@memoize
@validate_types(n=int)
def fibonacci(n):
    """Calculate nth Fibonacci number."""
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

# Equivalent to:
# fibonacci = timeit(memoize(validate_types(n=int)(fibonacci)))

print(fibonacci(30))  # Fast due to memoization
```

Chapter 9: Closures and Nested Functions {#closures}

Nested Functions

python

```python
def outer_function(x):
    """Outer function with nested function."""

    def inner_function(y):
        """Inner function can access outer function's variables."""
        return x + y

    return inner_function

# Create closures
add_five = outer_function(5)
add_ten = outer_function(10)

print(add_five(3))    # 8
print(add_ten(3))     # 13

# Inner function remembers the environment
print(add_five.__closure__)  # (<cell at 0x...>,)
print(add_five.__closure__[0].cell_contents)  # 5
```

Understanding Closures

python

```python
def make_counter(start=0):
    """Create a counter function with closure."""
    count = start

    def counter():
        nonlocal count
        count += 1
        return count

    # Add methods to the closure
    def reset():
        nonlocal count
        count = start

    def set_count(value):
        nonlocal count
        count = value

    # Return multiple functions sharing the same closure
    counter.reset = reset
    counter.set_count = set_count

    return counter

# Create independent counters
counter1 = make_counter()
counter2 = make_counter(100)

print(counter1())  # 1
print(counter1())  # 2
print(counter2())  # 101
print(counter2())  # 102

counter1.reset()
print(counter1())  # 1
```

Closure Use Cases

python

```python
# 1. Configuration functions
def configure_greeting(greeting):
    """Create customized greeting function."""
    def greet(name):
        return f"{greeting}, {name}!"
    return greet

say_hello = configure_greeting("Hello")
say_hi = configure_greeting("Hi")
say_hey = configure_greeting("Hey")

print(say_hello("Alice"))  # Hello, Alice!
print(say_hi("Bob"))       # Hi, Bob!

# 2. Function factories with state
def make_accumulator():
    """Create accumulator with internal state."""
    total = 0

    def add(value):
        nonlocal total
        total += value
        return total

    return add

acc1 = make_accumulator()
acc2 = make_accumulator()

print(acc1(10))  # 10
print(acc1(20))  # 30
print(acc2(5))   # 5 (independent)

# 3. Callbacks with context
def create_button_handler(button_id):
    """Create event handler with context."""
    click_count = 0

    def handle_click(event):
        nonlocal click_count
        click_count += 1
        print(f"Button {button_id} clicked {click_count} times")
        print(f"Event: {event}")

    return handle_click
```

```python
button1_handler = create_button_handler("Save")
button2_handler = create_button_handler("Cancel")

# Simulate clicks
button1_handler({'x': 100, 'y': 50})
button1_handler({'x': 102, 'y': 51})
button2_handler({'x': 200, 'y': 50})
```

Advanced Closure Patterns

python

```python
# 1. Closure-based private variables
def create_bank_account(initial_balance):
    """Create bank account with private balance."""
    balance = initial_balance
    transaction_history = []

    def deposit(amount):
        nonlocal balance
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")
        balance += amount
        transaction_history.append(f"Deposited: ${amount}")
        return balance

    def withdraw(amount):
        nonlocal balance
        if amount <= 0:
            raise ValueError("Withdrawal amount must be positive")
        if amount > balance:
            raise ValueError("Insufficient funds")
        balance -= amount
        transaction_history.append(f"Withdrew: ${amount}")
        return balance

    def get_balance():
        return balance

    def get_history():
        return transaction_history.copy()

    # Return interface object
    return {
        'deposit': deposit,
        'withdraw': withdraw,
        'get_balance': get_balance,
        'get_history': get_history
    }

account = create_bank_account(1000)
print(account['deposit'](500))      # 1500
print(account['withdraw'](200))     # 1300
print(account['get_balance']())     # 1300
print(account['get_history']())     # ['Deposited: $500', 'Withdrew: $200']

# 2. Decorators using closures
def rate_limiter(max_calls, period):
    """Rate limiting decorator using closure."""
    def decorator(func):
        calls = []

        def wrapper(*args, **kwargs):
            now = time.time()
            # Remove old calls
            nonlocal calls
            calls = [call_time for call_time in calls
                    if now - call_time < period]

            if len(calls) >= max_calls:
                raise Exception(f"Rate limit exceeded: {max_calls} calls per {period}s")

            calls.append(now)
            return func(*args, **kwargs)

        return wrapper
    return decorator

@rate_limiter(max_calls=3, period=10)
def api_call():
    return "API response"

# 3. Partial application using closures
def partial(func, *partial_args, **partial_kwargs):
    """Create partially applied function."""
    def wrapper(*args, **kwargs):
        new_args = partial_args + args
        new_kwargs = {**partial_kwargs, **kwargs}
        return func(*new_args, **new_kwargs)
    return wrapper

def greet(greeting, name, punctuation="!"):
    return f"{greeting}, {name}{punctuation}"

say_hello = partial(greet, "Hello")
say_hello_excited = partial(greet, "Hello", punctuation="!!!")

print(say_hello("Alice"))          # Hello, Alice!
print(say_hello_excited("Bob"))    # Hello, Bob!!!
```

Closure Pitfalls and Solutions

python

```python
# Pitfall 1: Late binding in loops
def create_multipliers():
    """Incorrect: All functions will multiply by 4."""
    multipliers = []
    for i in range(5):
        multipliers.append(lambda x: x * i)
    return multipliers

multipliers = create_multipliers()
print(multipliers[0](10))  # 40 (not 0!)
print(multipliers[1](10))  # 40 (not 10!)

# Solution: Default parameter
def create_multipliers_fixed():
    """Correct: Each function captures its own i."""
    multipliers = []
    for i in range(5):
        multipliers.append(lambda x, i=i: x * i)
    return multipliers

# Alternative solution: Factory function
def create_multipliers_factory():
    """Using factory function."""
    def make_multiplier(n):
        return lambda x: x * n

    return [make_multiplier(i) for i in range(5)]

# Pitfall 2: Mutable default arguments in closures
def create_appender(lst=[]):  # Problematic!
    def append(value):
        lst.append(value)
        return lst
    return append

app1 = create_appender()
app2 = create_appender()
print(app1(1))  # [1]
print(app2(2))  # [1, 2] - Shared list!

# Solution: None default
def create_appender_fixed(lst=None):
    if lst is None:
        lst = []

    def append(value):
        lst.append(value)
        return lst
    return append
```

Chapter 10: Generators and Yield {#generators}

Generator Basics

python

```
# Generator function
def count_up_to(n):
    """Generate numbers from 1 to n."""
    i = 1
    while i <= n:
        yield i
        i += 1

# Create generator object
counter = count_up_to(5)
print(type(counter))  # <class 'generator'>

# Iterate through generator
for num in counter:
    print(num)  # 1, 2, 3, 4, 5

# Generator is exhausted after iteration
print(list(counter))  # [] - Empty!

# Manual iteration
counter2 = count_up_to(3)
print(next(counter2))  # 1
print(next(counter2))  # 2
print(next(counter2))  # 3
# print(next(counter2))  # StopIteration
```

Generator Expressions

```python
# List comprehension - creates list immediately
squares_list = [x**2 for x in range(10)]
print(squares_list)  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# Generator expression - creates generator object
squares_gen = (x**2 for x in range(10))
print(squares_gen)  # <generator object ...>

# Consume generator
print(list(squares_gen))  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

# Memory efficiency
import sys
large_list = [x for x in range(1000000)]
large_gen = (x for x in range(1000000))

print(f"List size: {sys.getsizeof(large_list)} bytes")
print(f"Generator size: {sys.getsizeof(large_gen)} bytes")
```

Advanced Generator Patterns

python

```python
# 1. Infinite generators
def fibonacci():
    """Generate infinite Fibonacci sequence."""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Use with itertools
import itertools
fib = fibonacci()
first_10 = list(itertools.islice(fib, 10))
print(first_10)  # [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

# 2. Generator pipelines
def read_lines(filename):
    """Generate lines from file."""
    with open(filename) as f:
        for line in f:
            yield line.strip()

def filter_comments(lines):
    """Filter out comment lines."""
    for line in lines:
        if not line.startswith('#'):
            yield line

def extract_numbers(lines):
    """Extract numbers from lines."""
    for line in lines:
        parts = line.split()
        for part in parts:
            try:
                yield float(part)
            except ValueError:
                pass

# Chain generators
# lines = read_lines('data.txt')
# filtered = filter_comments(lines)
# numbers = extract_numbers(filtered)
# total = sum(numbers)

# 3. yield from (Python 3.3+)
def flatten(nested_list):
    """Recursively flatten nested lists."""
    for item in nested_list:
        if isinstance(item, list):
            yield from flatten(item)
        else:
            yield item

nested = [1, [2, 3, [4, 5]], 6, [7, [8, [9]]]]
print(list(flatten(nested)))  # [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Coroutines with send()

python

```python
def running_average():
    """Coroutine that maintains running average."""
    total = 0
    count = 0
    average = 0

    while True:
        value = yield average
        if value is not None:
            total += value
            count += 1
            average = total / count

# Initialize coroutine
avg = running_average()
next(avg)  # Prime the coroutine

# Send values
print(avg.send(10))   # 10.0
print(avg.send(20))   # 15.0
print(avg.send(30))   # 20.0
print(avg.send(5))    # 16.25

# Two-way communication
def echo_coroutine():
    """Echo received values with transformation."""
    while True:
        value = yield
        yield value.upper() if isinstance(value, str) else value * 2

echo = echo_coroutine()
next(echo)  # Prime

# Send and receive
echo.send("hello")
print(next(echo))  # HELLO

echo.send(5)
print(next(echo))  # 10
```

Generator Methods

python

```python
def demonstrator():
    """Demonstrate generator methods."""
    try:
        x = 0
        while True:
            value = yield x
            if value is not None:
                x = value
            else:
                x += 1
    except GeneratorExit:
        print("Generator closing")
    finally:
        print("Cleanup complete")

gen = demonstrator()
print(next(gen))      # 0
print(gen.send(10))   # 10
print(next(gen))      # 11

# throw() method
def error_handler():
    """Handle exceptions in generator."""
    try:
        while True:
            try:
                yield "OK"
            except ValueError as e:
                yield f"Handled: {e}"
    except GeneratorExit:
        print("Closing generator")

gen = error_handler()
print(next(gen))  # OK
print(gen.throw(ValueError("Test error")))  # Handled: Test error
print(next(gen))  # OK

# close() method
gen.close()  # Closing generator
```

Practical Generator Examples

python

```python
# 1. Large file processing
def process_large_csv(filename):
    """Process large CSV file row by row."""
    with open(filename) as f:
        headers = next(f).strip().split(',')
        for line in f:
            values = line.strip().split(',')
            yield dict(zip(headers, values))

# Memory efficient processing
# for row in process_large_csv('large_file.csv'):
#     process_row(row)

# 2. Data batching
def batch(iterable, batch_size):
    """Yield batches of items."""
    batch = []
    for item in iterable:
        batch.append(item)
        if len(batch) == batch_size:
            yield batch
            batch = []
    if batch:  # Yield remaining items
        yield batch

data = range(10)
for batch_data in batch(data, 3):
    print(batch_data)
# [0, 1, 2]
# [3, 4, 5]
# [6, 7, 8]
# [9]

# 3. Window sliding
def sliding_window(iterable, window_size):
    """Generate sliding windows over iterable."""
    window = []
    for item in iterable:
        window.append(item)
        if len(window) == window_size:
            yield tuple(window)
            window.pop(0)

data = [1, 2, 3, 4, 5]
for window in sliding_window(data, 3):
    print(window)
```

```python
# (1, 2, 3)
# (2, 3, 4)
# (3, 4, 5)

# 4. Tree traversal
class TreeNode:
    def __init__(self, value, children=None):
        self.value = value
        self.children = children or []

def traverse_tree(node):
    """Generate all values in tree (depth-first)."""
    yield node.value
    for child in node.children:
        yield from traverse_tree(child)

# Example tree
root = TreeNode(1, [
    TreeNode(2, [TreeNode(4), TreeNode(5)]),
    TreeNode(3, [TreeNode(6)])
])

print(list(traverse_tree(root)))  # [1, 2, 4, 5, 3, 6]
```

Chapter 11: Recursive Functions {#recursion}

Recursion Basics

python

```python
def factorial(n):
    """Calculate factorial recursively."""
    # Base case
    if n <= 1:
        return 1
    # Recursive case
    return n * factorial(n - 1)

print(factorial(5))  # 120 (5 * 4 * 3 * 2 * 1)

# Visualizing recursion
def factorial_verbose(n, depth=0):
    """Factorial with call visualization."""
    indent = "  " * depth
    print(f"{indent}factorial({n})")

    if n <= 1:
        print(f"{indent}returning 1")
        return 1

    result = n * factorial_verbose(n - 1, depth + 1)
    print(f"{indent}returning {result}")
    return result

factorial_verbose(4)
```

Common Recursive Patterns

```
python
```

```python
# 1. Tree/nested structure processing
def sum_nested_list(lst):
    """Sum all numbers in nested list structure."""
    total = 0
    for item in lst:
        if isinstance(item, list):
            total += sum_nested_list(item)
        else:
            total += item
    return total

nested = [1, [2, 3], [4, [5, 6]], 7]
print(sum_nested_list(nested))  # 28

# 2. Divide and conquer
def binary_search(arr, target, low=0, high=None):
    """Recursive binary search."""
    if high is None:
        high = len(arr) - 1

    if low > high:
        return -1

    mid = (low + high) // 2

    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        return binary_search(arr, target, mid + 1, high)
    else:
        return binary_search(arr, target, low, mid - 1)

sorted_list = [1, 3, 5, 7, 9, 11, 13, 15]
print(binary_search(sorted_list, 7))   # 3
print(binary_search(sorted_list, 10))  # -1

# 3. Backtracking
def find_path(maze, start, end, path=[]):
    """Find path through maze using backtracking."""
    path = path + [start]

    if start == end:
        return path

    for next_pos in get_valid_moves(maze, start):
        if next_pos not in path:  # Avoid cycles
```

```
            new_path = find_path(maze, next_pos, end, path)
            if new_path:
                return new_path

    return None  # No path found

def get_valid_moves(maze, pos):
    """Get valid moves from position (simplified)."""
    # Implementation depends on maze structure
    pass
```

Tail Recursion and Optimization

python

```python
# Regular recursion (not tail-recursive)
def factorial_regular(n):
    if n <= 1:
        return 1
    return n * factorial_regular(n - 1)  # Operation after recursive call

# Tail-recursive version
def factorial_tail(n, accumulator=1):
    """Tail-recursive factorial."""
    if n <= 1:
        return accumulator
    return factorial_tail(n - 1, n * accumulator)  # Nothing after recursive call

# Python doesn't optimize tail recursion, but we can use a trampoline
def trampoline(func):
    """Trampoline to optimize tail recursion."""
    def wrapper(*args, **kwargs):
        result = func(*args, **kwargs)
        while callable(result):
            result = result()
        return result
    return wrapper

def factorial_trampoline(n, accumulator=1):
    """Factorial using trampoline pattern."""
    if n <= 1:
        return accumulator
    return lambda: factorial_trampoline(n - 1, n * accumulator)

factorial_optimized = trampoline(factorial_trampoline)
print(factorial_optimized(5))  # 120
```

Memoization for Recursive Functions

python

```python
# Manual memoization
def memoize(func):
    """Memoization decorator for recursive functions."""
    cache = {}

    def wrapper(*args):
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result

    wrapper.cache = cache  # Expose cache for inspection
    return wrapper

@memoize
def fibonacci(n):
    """Memoized Fibonacci."""
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(100))  # Fast even for large n
print(fibonacci.cache)  # Inspect cache

# Using functools.lru_cache
from functools import lru_cache

@lru_cache(maxsize=None)
def fibonacci_lru(n):
    """Fibonacci with LRU cache."""
    if n <= 1:
        return n
    return fibonacci_lru(n - 1) + fibonacci_lru(n - 2)

print(fibonacci_lru(100))
print(fibonacci_lru.cache_info())  # CacheInfo(hits=98, misses=101, maxsize=None, currsize=101)
```

Advanced Recursive Examples

python

```python
# 1. Permutations
def permutations(items):
    """Generate all permutations of items."""
    if len(items) <= 1:
        yield items
    else:
        for i in range(len(items)):
            # Remove current item
            remaining = items[:i] + items[i+1:]
            # Generate permutations of remaining items
            for perm in permutations(remaining):
                yield [items[i]] + perm

print(list(permutations([1, 2, 3])))
# [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

# 2. Quick sort
def quicksort(arr):
    """Recursive quicksort implementation."""
    if len(arr) <= 1:
        return arr

    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]

    return quicksort(left) + middle + quicksort(right)

print(quicksort([3, 6, 8, 10, 1, 2, 1]))  # [1, 1, 2, 3, 6, 8, 10]

# 3. Recursive data structure processing
def deep_update(d, updates):
    """Recursively update nested dictionary."""
    for key, value in updates.items():
        if isinstance(value, dict) and key in d and isinstance(d[key], dict):
            deep_update(d[key], value)
        else:
            d[key] = value
    return d

original = {
    'a': 1,
    'b': {'c': 2, 'd': 3},
    'e': {'f': {'g': 4}}
}
```

```python
updates = {
    'b': {'c': 20, 'x': 10},
    'e': {'f': {'g': 40, 'h': 50}}
}

print(deep_update(original, updates))
# {'a': 1, 'b': {'c': 20, 'd': 3, 'x': 10}, 'e': {'f': {'g': 40, 'h': 50}}}

# 4. Expression evaluation
def evaluate(expression):
    """Evaluate mathematical expression recursively."""
    if isinstance(expression, (int, float)):
        return expression

    operator, left, right = expression
    left_val = evaluate(left)
    right_val = evaluate(right)

    if operator == '+':
        return left_val + right_val
    elif operator == '-':
        return left_val - right_val
    elif operator == '*':
        return left_val * right_val
    elif operator == '/':
        return left_val / right_val

# Expression tree: (5 + 3) * (10 - 6)
expr = ('*', ('+', 5, 3), ('-', 10, 6))
print(evaluate(expr))  # 32
```

Recursion Pitfalls and Solutions

python

```python
# Stack overflow prevention
import sys
print(f"Default recursion limit: {sys.getrecursionlimit()}")

# Increase recursion limit (be careful!)
# sys.setrecursionlimit(10000)

# Better solution: Convert to iteration
def factorial_iterative(n):
    """Iterative factorial to avoid stack overflow."""
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result

# Mutual recursion
def is_even(n):
    """Check if number is even using mutual recursion."""
    if n == 0:
        return True
    return is_odd(n - 1)

def is_odd(n):
    """Check if number is odd using mutual recursion."""
    if n == 0:
        return False
    return is_even(n - 1)

print(is_even(4))  # True
print(is_odd(4))   # False

# Debugging recursive functions
def factorial_debug(n, indent=0):
    """Factorial with debug output."""
    spaces = "  " * (indent * 2)
    print(f"{spaces}factorial({n}) called")

    if n <= 1:
        print(f"{spaces}Base case reached, returning 1")
        return 1

    print(f"{spaces}Computing {n} * factorial({n-1})")
    result = n * factorial_debug(n - 1, indent + 1)
    print(f"{spaces}Returning {result}")
    return result
```

```
factorial_debug(4)
```

Chapter 12: Built-in Functions {#builtin-functions}

Essential Built-in Functions

```
python

# Type conversion functions
int("123")       # 123
float("3.14")    # 3.14
str(123)         # "123"
bool(0)          # False
list("abc")      # ['a', 'b', 'c']
tuple([1,2,3])   # (1, 2, 3)
set([1,2,2,3])   # {1, 2, 3}
dict([('a',1), ('b',2)])  # {'a': 1, 'b': 2}

# Mathematical functions
abs(-5)          # 5
round(3.7)       # 4
round(3.14159, 2)  # 3.14
min(5, 3, 8, 1)    # 1
max([1, 5, 3])     # 5
sum([1, 2, 3, 4])  # 10
pow(2, 3)          # 8 (same as 2**3)
divmod(10, 3)      # (3, 1) - quotient and remainder

# Sequence functions
len([1, 2, 3])     # 3
sorted([3, 1, 4])  # [1, 3, 4]
reversed([1, 2, 3]) # <reversed object>
list(reversed([1, 2, 3]))  # [3, 2, 1]

# String-specific
ord('A')     # 65 (ASCII value)
chr(65)      # 'A' (character from ASCII)
```

Input/Output Functions

python

```python
# Basic I/O
name = input("Enter your name: ")
print(f"Hello, {name}!")

# print() parameters
print("A", "B", "C", sep="-")  # A-B-C
print("Hello", end="")
print(" World")  # Hello World (no newline between)

# File operations
with open('file.txt', 'w') as f:
    f.write("Hello, World!")

with open('file.txt', 'r') as f:
    content = f.read()

# format() function
"{} + {} = {}".format(2, 3, 5)  # "2 + 3 = 5"
"{1} {0}".format("world", "hello")  # "hello world"
format(1234.5678, '.2f')  # '1234.57'
```

Iteration and Functional Tools

python

```python
# range() function
list(range(5))         # [0, 1, 2, 3, 4]
list(range(2, 10, 2))  # [2, 4, 6, 8]

# enumerate()
fruits = ['apple', 'banana', 'orange']
for i, fruit in enumerate(fruits):
    print(f"{i}: {fruit}")

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

# map()
numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))  # [1, 4, 9, 16]

# filter()
evens = list(filter(lambda x: x % 2 == 0, numbers))  # [2, 4]

# all() and any()
all([True, True, True])    # True
all([True, False, True])   # False
any([False, False, True]) # True
any([False, False])        # False
```

Object and Attribute Functions

python

```python
# Object inspection
class MyClass:
    def __init__(self):
        self.attribute = "value"

obj = MyClass()

type(obj)           # <class '__main__.MyClass'>
isinstance(obj, MyClass)  # True
hasattr(obj, 'attribute') # True
getattr(obj, 'attribute') # 'value'
setattr(obj, 'new_attr', 'new_value')
delattr(obj, 'attribute')

# dir() - list attributes
dir(obj)  # List of all attributes and methods

# vars() - return __dict__
vars(obj)  # {'new_attr': 'new_value'}

# id() - object identity
id(obj)  # Unique identifier (memory address)

# hash() - hash value
hash("hello")  # Hash value for hashable objects
```

Advanced Built-in Functions

python

```python
# eval() and exec()
result = eval("2 + 3 * 4")  # 14
exec("x = 5; y = 10; z = x + y")

# compile()
code = compile("print('Hello')", "<string>", "exec")
exec(code)  # Hello

# globals() and locals()
x = 10  # Global
def func():
    y = 20  # Local
    print(locals())   # {'y': 20}
    print(globals())  # All global variables

# callable()
callable(print)      # True
callable(lambda x: x)  # True
callable(42)         # False

# iter() and next()
iterator = iter([1, 2, 3])
next(iterator)  # 1
next(iterator)  # 2
next(iterator, "default")  # 3
next(iterator, "default")  # "default"
```

Memory and Reference Functions

python

```python
# Memory view
data = bytearray(b'Hello')
mv = memoryview(data)
mv[0] = ord('h')  # Modify through memory view
print(data)  # bytearray(b'hello')

# Copying
import copy
original = [[1, 2], [3, 4]]
shallow = copy.copy(original)
deep = copy.deepcopy(original)

original[0][0] = 99
print(shallow[0][0])  # 99 (affected)
print(deep[0][0])     # 1 (not affected)

# Reference counting
import sys
a = []
b = a
print(sys.getrefcount(a))  # Reference count
```

Binary and Encoding Functions

python

```python
# Binary operations
bin(10)     # '0b1010'
oct(10)     # '0o12'
hex(10)     # '0xa'

# Bitwise functions
a = 0b1010  # 10
b = 0b1100  # 12
a & b       # 8 (AND)
a | b       # 14 (OR)
a ^ b       # 6 (XOR)
~a          # -11 (NOT)
a << 2      # 40 (left shift)
a >> 1      # 5 (right shift)

# Encoding/Decoding
text = "Hello, 世界"
encoded = text.encode('utf-8')  # b'Hello, \xe4\xb8\x96\xe7\x95\x8c'
decoded = encoded.decode('utf-8')  # "Hello, 世界"

# bytes and bytearray
b = bytes([72, 101, 108, 108, 111])  # b'Hello'
ba = bytearray(b)
ba[0] = 104  # Mutable
print(ba)  # bytearray(b'hello')
```

Functional Programming Built-ins

python

```python
from functools import reduce

# reduce()
numbers = [1, 2, 3, 4, 5]
product = reduce(lambda x, y: x * y, numbers)  # 120

# Complex example with reduce
data = [
    {'name': 'Alice', 'score': 85},
    {'name': 'Bob', 'score': 92},
    {'name': 'Charlie', 'score': 78}
]

total_score = reduce(
    lambda acc, student: acc + student['score'],
    data,
    0  # Initial value
)
print(f"Average: {total_score / len(data)}")

# slice object
s = slice(1, 5, 2)  # start:stop:step
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(numbers[s])  # [1, 3]
print(numbers[1:5:2])  # Same result

# property() as a function
class Temperature:
    def __init__(self):
        self._celsius = 0

    def get_celsius(self):
        return self._celsius

    def set_celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature below absolute zero")
        self._celsius = value

    celsius = property(get_celsius, set_celsius)

# staticmethod() and classmethod()
class MyClass:
    @staticmethod
    def static_method():
        print("Static method")
    @classmethod
    def class_method(cls):
        print(f"Class method of {cls}")

# Using as functions
MyClass.static_method = staticmethod(lambda: print("New static"))
MyClass.class_method = classmethod(lambda cls: print(f"New class method of {cls}"))
```

Special Built-in Functions

python

```python
# super()
class Parent:
    def method(self):
        print("Parent method")

class Child(Parent):
    def method(self):
        super().method()  # Call parent method
        print("Child method")

# __import__() - Low-level import
math = __import__('math')
print(math.pi)  # 3.141592653589793

# help() - interactive help
help(sorted)  # Display help for sorted function

# repr() vs str()
import datetime
now = datetime.datetime.now()
print(str(now))   # Human-readable
print(repr(now))  # Developer-readable, can recreate object

# Complex sorting with key function
students = [
    {'name': 'Alice', 'grade': 85, 'age': 20},
    {'name': 'Bob', 'grade': 92, 'age': 19},
    {'name': 'Charlie', 'grade': 85, 'age': 21}
]

# Sort by grade (descending), then age (ascending)
sorted_students = sorted(
    students,
    key=lambda s: (-s['grade'], s['age'])
)
```

Chapter 13: Functional Programming {#functional-programming}

Functional Programming Principles

python

```python
# Pure functions - no side effects
# Impure function
counter = 0
def impure_increment():
    global counter
    counter += 1
    return counter

# Pure function
def pure_increment(value):
    return value + 1

# Immutability
# Bad - mutating
def add_item_bad(lst, item):
    lst.append(item)
    return lst

# Good - creating new
def add_item_good(lst, item):
    return lst + [item]

# First-class functions
def apply_twice(func, arg):
    return func(func(arg))

result = apply_twice(lambda x: x * 2, 5)  # 20
```

Higher-Order Functions

python

```python
# Function composition
def compose(*functions):
    """Compose multiple functions."""
    def inner(arg):
        result = arg
        for func in reversed(functions):
            result = func(result)
        return result
    return inner

# Example
add_one = lambda x: x + 1
multiply_two = lambda x: x * 2
square = lambda x: x ** 2

f = compose(square, multiply_two, add_one)
print(f(3))  # ((3 + 1) * 2) ** 2 = 64

# Currying
def curry(func):
    """Convert function to curried form."""
    def curried(*args, **kwargs):
        if len(args) + len(kwargs) >= func.__code__.co_argcount:
            return func(*args, **kwargs)
        return lambda *new_args, **new_kwargs: curried(
            *(args + new_args),
            **{**kwargs, **new_kwargs}
        )
    return curried

@curry
def add(a, b, c):
    return a + b + c

print(add(1)(2)(3))  # 6
print(add(1, 2)(3))  # 6
```

Functional Data Processing

python

```python
from functools import reduce, partial
from operator import add, mul, itemgetter

# Using operator module
numbers = [1, 2, 3, 4, 5]
total = reduce(add, numbers)  # 15
product = reduce(mul, numbers)  # 120

# Partial application
def greet(greeting, name):
    return f"{greeting}, {name}!"

say_hello = partial(greet, "Hello")
say_goodbye = partial(greet, "Goodbye")

print(say_hello("Alice"))    # Hello, Alice!
print(say_goodbye("Bob"))    # Goodbye, Bob!

# Function pipelines
def pipeline(*functions):
    return reduce(lambda f, g: lambda x: g(f(x)), functions)

process = pipeline(
    lambda x: x.strip(),
    lambda x: x.lower(),
    lambda x: x.replace(' ', '_')
)

print(process("  Hello World  "))  # hello_world
```

Monadic Patterns

python

```python
# Maybe monad-like pattern
class Maybe:
    def __init__(self, value):
        self.value = value

    def bind(self, func):
        if self.value is None:
            return Maybe(None)
        try:
            return Maybe(func(self.value))
        except:
            return Maybe(None)

    def or_else(self, default):
        return self.value if self.value is not None else default

# Usage
result = (Maybe("123")
    .bind(int)
    .bind(lambda x: x * 2)
    .bind(lambda x: x + 10)
    .or_else(0))

print(result)  # 256

# Result type pattern
class Result:
    def __init__(self, value=None, error=None):
        self.value = value
        self.error = error
        self.is_ok = error is None

    @classmethod
    def ok(cls, value):
        return cls(value=value)

    @classmethod
    def err(cls, error):
        return cls(error=error)

    def map(self, func):
        if self.is_ok:
            try:
                return Result.ok(func(self.value))
            except Exception as e:
                return Result.err(str(e))
        return self

    def and_then(self, func):
        if self.is_ok:
            return func(self.value)
        return self

def safe_divide(a, b):
    if b == 0:
        return Result.err("Division by zero")
    return Result.ok(a / b)

result = (Result.ok(10)
    .and_then(lambda x: safe_divide(x, 2))
    .map(lambda x: x + 1))

if result.is_ok:
    print(f"Success: {result.value}")  # Success: 6.0
else:
    print(f"Error: {result.error}")
```

Lazy Evaluation

python

```python
# Lazy sequences
class LazySequence:
    def __init__(self, generator_func):
        self.generator_func = generator_func
        self._cache = []
        self._generator = None

    def __getitem__(self, index):
        if isinstance(index, slice):
            # Handle slicing
            return [self[i] for i in range(*index.indices(len(self)))]

        # Ensure we have enough cached values
        while len(self._cache) <= index:
            if self._generator is None:
                self._generator = self.generator_func()
            try:
                self._cache.append(next(self._generator))
            except StopIteration:
                raise IndexError("Index out of range")

        return self._cache[index]

    def take(self, n):
        return [self[i] for i in range(n)]

# Infinite sequences
def naturals():
    n = 1
    while True:
        yield n
        n += 1

def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Usage
nats = LazySequence(naturals)
fibs = LazySequence(fibonacci)

print(nats.take(10))  # [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(fibs[10])       # 55
```

```python
# Lazy evaluation with properties
class LazyProperty:
    def __init__(self, func):
        self.func = func
        self.name = f'_lazy_{func.__name__}'

    def __get__(self, obj, type=None):
        if obj is None:
            return self

        if not hasattr(obj, self.name):
            setattr(obj, self.name, self.func(obj))

        return getattr(obj, self.name)

class DataProcessor:
    def __init__(self, data):
        self.data = data

    @LazyProperty
    def processed_data(self):
        print("Processing data...")
        return [x * 2 for x in self.data]

    @LazyProperty
    def statistics(self):
        print("Calculating statistics...")
        return {
            'mean': sum(self.processed_data) / len(self.processed_data),
            'max': max(self.processed_data),
            'min': min(self.processed_data)
        }
```

Functional Design Patterns

python

```python
# Strategy pattern with functions
def process_data(data, strategy):
    return strategy(data)

# Different strategies
strategies = {
    'sum': sum,
    'average': lambda data: sum(data) / len(data),
    'median': lambda data: sorted(data)[len(data) // 2],
    'range': lambda data: max(data) - min(data)
}

data = [1, 2, 3, 4, 5]
for name, strategy in strategies.items():
    result = process_data(data, strategy)
    print(f"{name}: {result}")

# Memoization pattern
def memoize(func):
    cache = {}

    def memoized(*args, **kwargs):
        key = (args, tuple(sorted(kwargs.items())))
        if key not in cache:
            cache[key] = func(*args, **kwargs)
        return cache[key]

    memoized.cache = cache
    memoized.clear_cache = cache.clear
    return memoized

# Functional error handling
def try_except(func, exception_handler, *exceptions):
    """Functional try-except pattern."""
    def wrapper(*args, **kwargs):
        try:
            return func(*args, **kwargs)
        except exceptions as e:
            return exception_handler(e)
    return wrapper

safe_int = try_except(
    int,
    lambda e: 0,
    ValueError
)
```

```
print(safe_int("123"))   # 123
print(safe_int("abc"))   # 0
```

Chapter 14: Async Functions {#async-functions}

Async/Await Basics

python

```
import asyncio
import time

# Basic async function
async def hello_async():
    print("Hello")
    await asyncio.sleep(1)
    print("World")

# Running async function
asyncio.run(hello_async())

# Multiple async tasks
async def task(name, delay):
    print(f"Task {name} starting")
    await asyncio.sleep(delay)
    print(f"Task {name} completed after {delay}s")
    return f"Result from {name}"

async def main():
    # Run tasks concurrently
    results = await asyncio.gather(
        task("A", 2),
        task("B", 1),
        task("C", 3)
    )
    print(f"Results: {results}")

asyncio.run(main())
```

Async Context Managers

python

```python
class AsyncResource:
    async def __aenter__(self):
        print("Acquiring resource")
        await asyncio.sleep(1)
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        print("Releasing resource")
        await asyncio.sleep(0.5)

    async def do_something(self):
        print("Using resource")
        await asyncio.sleep(0.5)

async def use_resource():
    async with AsyncResource() as resource:
        await resource.do_something()

# Async file operations (requires aiofiles)
# async with aiofiles.open('file.txt', 'r') as f:
#     content = await f.read()
```

Async Generators

python

```python
async def async_range(start, stop):
    """Async generator example."""
    for i in range(start, stop):
        await asyncio.sleep(0.1)  # Simulate async work
        yield i

async def consume_async_generator():
    async for num in async_range(0, 5):
        print(f"Got: {num}")

# Async generator with cleanup
async def monitored_generator():
    print("Generator started")
    try:
        for i in range(5):
            await asyncio.sleep(0.1)
            yield i
    finally:
        print("Generator cleanup")

async def consume_monitored():
    async for value in monitored_generator():
        print(f"Value: {value}")
        if value == 2:
            break  # Early exit triggers cleanup
```

Async Patterns and Best Practices

```
python
```

```python
# Timeout handling
async def long_running_task():
    await asyncio.sleep(10)
    return "Completed"

async def with_timeout():
    try:
        result = await asyncio.wait_for(long_running_task(), timeout=2.0)
        print(result)
    except asyncio.TimeoutError:
        print("Task timed out")

# Concurrent execution with different patterns
async def concurrent_patterns():
    # Pattern 1: gather
    results = await asyncio.gather(
        task("A", 1),
        task("B", 2),
        return_exceptions=True
    )

    # Pattern 2: as_completed
    tasks = [
        asyncio.create_task(task("A", 2)),
        asyncio.create_task(task("B", 1)),
        asyncio.create_task(task("C", 3))
    ]

    for coro in asyncio.as_completed(tasks):
        result = await coro
        print(f"Got result: {result}")

    # Pattern 3: wait
    tasks = [
        asyncio.create_task(task("A", 1)),
        asyncio.create_task(task("B", 2))
    ]

    done, pending = await asyncio.wait(tasks, timeout=1.5)
    print(f"Done: {len(done)}, Pending: {len(pending)}")

# Async queue for producer-consumer
async def producer(queue, n):
    for i in range(n):
        await asyncio.sleep(0.5)
        await queue.put(f"Item {i}")
        print(f"Produced: Item {i}")

async def consumer(queue, name):
    while True:
        item = await queue.get()
        if item is None:  # Poison pill
            break
        print(f"Consumer {name} got: {item}")
        await asyncio.sleep(1)  # Process item
        queue.task_done()

async def producer_consumer_example():
    queue = asyncio.Queue(maxsize=3)

    # Create tasks
    producers = [asyncio.create_task(producer(queue, 5))]
    consumers = [
        asyncio.create_task(consumer(queue, "A")),
        asyncio.create_task(consumer(queue, "B"))
    ]

    # Wait for producers
    await asyncio.gather(*producers)

    # Send poison pills
    for _ in consumers:
        await queue.put(None)

    # Wait for consumers
    await asyncio.gather(*consumers)
```

Advanced Async Techniques

python

```python
# Async locks and synchronization
class AsyncCounter:
    def __init__(self):
        self.count = 0
        self.lock = asyncio.Lock()

    async def increment(self):
        async with self.lock:
            current = self.count
            await asyncio.sleep(0.1)  # Simulate work
            self.count = current + 1

async def test_counter():
    counter = AsyncCounter()
    await asyncio.gather(*[counter.increment() for _ in range(10)])
    print(f"Final count: {counter.count}")  # Should be 10

# Async semaphore for rate limiting
async def rate_limited_task(semaphore, task_id):
    async with semaphore:
        print(f"Task {task_id} started")
        await asyncio.sleep(1)
        print(f"Task {task_id} completed")

async def rate_limiting_example():
    semaphore = asyncio.Semaphore(3)  # Max 3 concurrent tasks
    tasks = [rate_limited_task(semaphore, i) for i in range(10)]
    await asyncio.gather(*tasks)

# Async event for coordination
async def waiter(event, name):
    print(f"{name} waiting for event")
    await event.wait()
    print(f"{name} received event")

async def setter(event):
    await asyncio.sleep(2)
    print("Setting event")
    event.set()

async def event_example():
    event = asyncio.Event()
    await asyncio.gather(
        waiter(event, "Waiter 1"),
        waiter(event, "Waiter 2"),
        setter(event)
    )

# Async callbacks and futures
async def async_with_callback():
    future = asyncio.Future()

    def callback(fut):
        print(f"Callback called with result: {fut.result()}")

    future.add_done_callback(callback)

    # Simulate async work
    await asyncio.sleep(1)
    future.set_result("Success!")

    return await future
```

Chapter 15: Function Annotations and Type Hints {#annotations}

Basic Type Hints

```
python

# Function annotations
def greet(name: str) -> str:
    return f"Hello, {name}!"

# Variable annotations
age: int = 25
price: float = 19.99
is_valid: bool = True
names: list[str] = ["Alice", "Bob"]

# Using type hints with default values
def create_user(name: str, age: int = 0, active: bool = True) -> dict:
    return {
        'name': name,
        'age': age,
        'active': active
    }
```

Advanced Type Hints

python

```python
from typing import (
    List, Dict, Tuple, Set, Optional, Union, Any,
    Callable, TypeVar, Generic, Protocol, Literal,
    Final, TypedDict, overload
)

# Complex type hints
Matrix = List[List[float]]
UserDict = Dict[str, Union[str, int, bool]]
Coordinate = Tuple[float, float]

# Optional and Union
def find_user(user_id: int) -> Optional[Dict[str, Any]]:
    # Returns user dict or None
    pass

def process_value(value: Union[int, float, str]) -> str:
    return str(value)

# Callable types
Handler = Callable[[str, int], bool]

def register_handler(handler: Handler) -> None:
    pass

# Type variables
T = TypeVar('T')
K = TypeVar('K')
V = TypeVar('V')

def first(items: List[T]) -> Optional[T]:
    return items[0] if items else None

def get_value(data: Dict[K, V], key: K) -> Optional[V]:
    return data.get(key)
```

Generic Functions and Classes

python

```python
# Generic function
def identity(x: T) -> T:
    return x

# Generic class
class Stack(Generic[T]):
    def __init__(self) -> None:
        self._items: List[T] = []

    def push(self, item: T) -> None:
        self._items.append(item)

    def pop(self) -> Optional[T]:
        return self._items.pop() if self._items else None

    def peek(self) -> Optional[T]:
        return self._items[-1] if self._items else None

# Using generic class
string_stack: Stack[str] = Stack()
string_stack.push("hello")

int_stack: Stack[int] = Stack()
int_stack.push(42)
```

Protocols and Structural Subtyping

```python
from typing import Protocol, runtime_checkable

@runtime_checkable
class Drawable(Protocol):
    def draw(self) -> None: ...
    def get_position(self) -> Tuple[float, float]: ...

class Circle:
    def __init__(self, x: float, y: float, radius: float):
        self.x = x
        self.y = y
        self.radius = radius

    def draw(self) -> None:
        print(f"Drawing circle at ({self.x}, {self.y})")

    def get_position(self) -> Tuple[float, float]:
        return (self.x, self.y)

def render(drawable: Drawable) -> None:
    drawable.draw()
    pos = drawable.get_position()
    print(f"Position: {pos}")

# Circle implements Drawable protocol without inheritance
circle = Circle(10, 20, 5)
render(circle)  # Works!

# Runtime checking
print(isinstance(circle, Drawable))  # True
```

TypedDict and Literal Types

python

```python
from typing import TypedDict, Literal, get_type_hints

# TypedDict for structured dictionaries
class UserInfo(TypedDict):
    name: str
    age: int
    email: str
    active: bool

def process_user(user: UserInfo) -> str:
    return f"{user['name']} ({user['email']})"

# Partial TypedDict
class PartialUserInfo(TypedDict, total=False):
    name: str
    age: int
    email: str

# Literal types
Direction = Literal["north", "south", "east", "west"]

def move(direction: Direction, steps: int) -> None:
    print(f"Moving {direction} for {steps} steps")

# Status with literal
Status = Literal["pending", "approved", "rejected"]

def update_status(status: Status) -> None:
    print(f"Status updated to: {status}")
```

Function Overloading

```python
from typing import overload

@overload
def process(x: int) -> str: ...

@overload
def process(x: str) -> int: ...

@overload
def process(x: List[int]) -> int: ...

def process(x: Union[int, str, List[int]]) -> Union[str, int]:
    if isinstance(x, int):
        return str(x)
    elif isinstance(x, str):
        return len(x)
    elif isinstance(x, list):
        return sum(x)
    else:
        raise TypeError(f"Unsupported type: {type(x)}")

# Type checker understands different return types
result1: str = process(42)       # OK
result2: int = process("hello")  # OK
result3: int = process([1,2,3])  # OK
```

Advanced Type Hint Patterns

python

```python
# Constrained type variables
from typing import TypeVar, Callable
import numbers

T = TypeVar('T', bound=numbers.Number)

def clamp(value: T, min_val: T, max_val: T) -> T:
    return max(min_val, min(value, max_val))

# Protocol with type variable
K = TypeVar('K')
V = TypeVar('V')

class Mapping(Protocol[K, V]):
    def __getitem__(self, key: K) -> V: ...
    def __setitem__(self, key: K, value: V) -> None: ...

# Recursive types
from __future__ import annotations

JSONValue = Union[str, int, float, bool, None, Dict[str, 'JSONValue'], List['JSONValue']]

def parse_json(data: str) -> JSONValue:
    import json
    return json.loads(data)

# Type aliases
Vector2D = Tuple[float, float]
Vector3D = Tuple[float, float, float]
Point = Union[Vector2D, Vector3D]

def distance(p1: Point, p2: Point) -> float:
    # Calculate distance
    pass

# Decorated function types
F = TypeVar('F', bound=Callable[..., Any])

def timer(func: F) -> F:
    @functools.wraps(func)
    def wrapper(*args: Any, **kwargs: Any) -> Any:
        start = time.time()
        result = func(*args, **kwargs)
        print(f"{func.__name__} took {time.time() - start}s")
        return result
    return wrapper  # type: ignore
```

```python
# Using Final
from typing import Final

MAX_RETRIES: Final = 3
API_KEY: Final[str] = "secret-key"

# MAX_RETRIES = 4  # Type checker error!
```

Runtime Type Checking

python

```python
from typing import get_type_hints, get_args, get_origin
import inspect

def validate_types(func: Callable) -> Callable:
    """Runtime type validation decorator."""
    hints = get_type_hints(func)
    sig = inspect.signature(func)

    @functools.wraps(func)
    def wrapper(*args: Any, **kwargs: Any) -> Any:
        # Bind arguments
        bound = sig.bind(*args, **kwargs)
        bound.apply_defaults()

        # Validate each argument
        for param_name, value in bound.arguments.items():
            if param_name in hints:
                expected_type = hints[param_name]
                if not check_type(value, expected_type):
                    raise TypeError(
                        f"{param_name} must be {expected_type}, "
                        f"got {type(value)}"
                    )

        result = func(*args, **kwargs)

        # Validate return value
        if 'return' in hints and hints['return'] != type(None):
            if not check_type(result, hints['return']):
                raise TypeError(
                    f"Return value must be {hints['return']}, "
                    f"got {type(result)}"
                )

        return result

    return wrapper

def check_type(value: Any, expected_type: type) -> bool:
    """Check if value matches expected type."""
    origin = get_origin(expected_type)

    if origin is Union:
        return any(check_type(value, t) for t in get_args(expected_type))
    elif origin is list:
        return isinstance(value, list) and all(
            check_type(item, get_args(expected_type)[0])
            for item in value
        )
    elif origin is dict:
        key_type, value_type = get_args(expected_type)
        return isinstance(value, dict) and all(
            check_type(k, key_type) and check_type(v, value_type)
            for k, v in value.items()
        )
    elif expected_type is Any:
        return True
    else:
        return isinstance(value, expected_type)

@validate_types
def add_numbers(a: int, b: int) -> int:
    return a + b

# This works
result = add_numbers(5, 3)  # 8

# This raises TypeError
# result = add_numbers("5", 3)  # TypeError: a must be <class 'int'>
```

Chapter 16: Advanced Patterns and Best Practices {#advanced-patterns}

Function Design Principles

python

```python
# Single Responsibility Principle
# Bad - doing too many things
def process_user_data_bad(user_data):
    # Validate data
    if not user_data.get('email'):
        raise ValueError("Email required")

    # Transform data
    user_data['email'] = user_data['email'].lower()

    # Save to database
    db.save(user_data)

    # Send email
    send_welcome_email(user_data['email'])

    # Log action
    logger.info(f"User {user_data['email']} created")

# Good - separate concerns
def validate_user_data(user_data):
    if not user_data.get('email'):
        raise ValueError("Email required")

def normalize_user_data(user_data):
    return {
        **user_data,
        'email': user_data['email'].lower()
    }

def create_user(user_data):
    validate_user_data(user_data)
    normalized = normalize_user_data(user_data)
    user = db.save(normalized)
    send_welcome_email(user.email)
    logger.info(f"User {user.email} created")
    return user
```

Error Handling Best Practices

python

```python
# Custom exceptions
class ApplicationError(Exception):
    """Base application exception."""
    pass

class ValidationError(ApplicationError):
    """Validation error."""
    def __init__(self, field, message):
        self.field = field
        self.message = message
        super().__init__(f"{field}: {message}")

class NotFoundError(ApplicationError):
    """Resource not found error."""
    def __init__(self, resource_type, resource_id):
        self.resource_type = resource_type
        self.resource_id = resource_id
        super().__init__(f"{resource_type} with id {resource_id} not found")

# Error handling patterns
def safe_divide(a: float, b: float) -> Union[float, None]:
    """Safe division with None on error."""
    try:
        return a / b
    except ZeroDivisionError:
        return None

# Result type pattern
from typing import Union, Generic, TypeVar

T = TypeVar('T')
E = TypeVar('E')

class Result(Generic[T, E]):
    def __init__(self, value: Union[T, None] = None, error: Union[E, None] = None):
        self._value = value
        self._error = error

    @property
    def is_ok(self) -> bool:
        return self._error is None

    @property
    def is_err(self) -> bool:
        return self._error is not None
    def unwrap(self) -> T:
        if self.is_err:
            raise ValueError(f"Called unwrap on error: {self._error}")
        return self._value

    def unwrap_or(self, default: T) -> T:
        return self._value if self.is_ok else default

    @classmethod
    def ok(cls, value: T) -> 'Result[T, E]':
        return cls(value=value)

    @classmethod
    def err(cls, error: E) -> 'Result[T, E]':
        return cls(error=error)

def parse_int(s: str) -> Result[int, str]:
    try:
        return Result.ok(int(s))
    except ValueError:
        return Result.err(f"'{s}' is not a valid integer")

# Usage
result = parse_int("42")
if result.is_ok:
    print(f"Parsed: {result.unwrap()}")
else:
    print(f"Error: {result._error}")
```

Documentation Best Practices

python

```python
def calculate_compound_interest(
    principal: float,
    rate: float,
    time: float,
    n: int = 12
) -> float:
    """
    Calculate compound interest.

    The compound interest formula is:
    A = P(1 + r/n)^(nt)

    Where:
    - A is the final amount
    - P is the principal amount
    - r is the annual interest rate (as a decimal)
    - n is the number of times interest is compounded per year
    - t is the time in years

    Args:
        principal: The initial amount of money
        rate: The annual interest rate (e.g., 0.05 for 5%)
        time: The time period in years
        n: The number of times interest is compounded per year (default: 12)

    Returns:
        The final amount after compound interest

    Raises:
        ValueError: If principal, rate, or time is negative
        ValueError: If n is less than 1

    Examples:
        >>> calculate_compound_interest(1000, 0.05, 2)
        1104.94...

        >>> calculate_compound_interest(5000, 0.08, 10, n=4)
        10955.61...

    Note:
        This function assumes that the rate is given as a decimal.
        For a percentage rate, divide by 100 before passing.
    """
    if principal < 0:
        raise ValueError("Principal must be non-negative")
    if rate < 0:
        raise ValueError("Rate must be non-negative")
    if time < 0:
        raise ValueError("Time must be non-negative")
    if n < 1:
        raise ValueError("Compounding frequency must be at least 1")

    return principal * (1 + rate / n) ** (n * time)
```

Performance Optimization Patterns

python

```python
import functools
import time
from typing import Callable, Any
import sys

# Memoization with size limit
def memoize_with_limit(maxsize: int = 128):
    """Memoization decorator with size limit."""
    def decorator(func: Callable) -> Callable:
        cache = {}
        cache_order = []

        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            key = (args, tuple(sorted(kwargs.items())))

            if key in cache:
                # Move to end (LRU)
                cache_order.remove(key)
                cache_order.append(key)
                return cache[key]

            result = func(*args, **kwargs)
            cache[key] = result
            cache_order.append(key)

            # Remove oldest if over limit
            if len(cache) > maxsize:
                oldest = cache_order.pop(0)
                del cache[oldest]

            return result

        wrapper.cache_info = lambda: {
            'size': len(cache),
            'maxsize': maxsize,
            'hits': sum(1 for k in cache_order if k in cache),
            'misses': len(cache_order) - len(cache)
        }

        return wrapper
    return decorator

# Lazy evaluation
class lazy_property:
    """Lazy property evaluation."""
    def __init__(self, func):
        self.func = func
        self.name = func.__name__

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self

        value = self.func(obj)
        # Replace the property with the computed value
        setattr(obj, self.name, value)
        return value

class DataAnalyzer:
    def __init__(self, data):
        self.data = data

    @lazy_property
    def mean(self):
        print("Computing mean...")
        return sum(self.data) / len(self.data)

    @lazy_property
    def variance(self):
        print("Computing variance...")
        mean = self.mean  # Uses cached value if already computed
        return sum((x - mean) ** 2 for x in self.data) / len(self.data)

    @lazy_property
    def std_dev(self):
        print("Computing standard deviation...")
        return self.variance ** 0.5

# Chunking for memory efficiency
def process_in_chunks(data: list, chunk_size: int, processor: Callable):
    """Process large data in chunks to save memory."""
    for i in range(0, len(data), chunk_size):
        chunk = data[i:i + chunk_size]
        yield processor(chunk)

# Example usage
def sum_chunk(chunk):
    return sum(chunk)

large_data = list(range(1000000))
```

```
chunk_sums = list(process_in_chunks(large_data, 10000, sum_chunk))
total = sum(chunk_sums)
```

Testing Functions

python

```python
import unittest
from unittest.mock import Mock, patch, MagicMock
import pytest

# Function to test
def fetch_user_data(user_id: int, api_client) -> dict:
    """Fetch user data from API."""
    try:
        response = api_client.get(f"/users/{user_id}")
        if response.status_code == 200:
            return response.json()
        elif response.status_code == 404:
            raise NotFoundError("User", user_id)
        else:
            raise Exception(f"API error: {response.status_code}")
    except ConnectionError:
        raise Exception("Network error")

# Unit tests with mocking
class TestFetchUserData(unittest.TestCase):
    def setUp(self):
        self.mock_client = Mock()

    def test_successful_fetch(self):
        # Arrange
        mock_response = Mock()
        mock_response.status_code = 200
        mock_response.json.return_value = {'id': 1, 'name': 'Alice'}
        self.mock_client.get.return_value = mock_response

        # Act
        result = fetch_user_data(1, self.mock_client)

        # Assert
        self.assertEqual(result, {'id': 1, 'name': 'Alice'})
        self.mock_client.get.assert_called_once_with('/users/1')

    def test_user_not_found(self):
        # Arrange
        mock_response = Mock()
        mock_response.status_code = 404
        self.mock_client.get.return_value = mock_response

        # Act & Assert
        with self.assertRaises(NotFoundError) as context:
            fetch_user_data(1, self.mock_client)
        self.assertEqual(context.exception.resource_type, "User")
        self.assertEqual(context.exception.resource_id, 1)

    def test_network_error(self):
        # Arrange
        self.mock_client.get.side_effect = ConnectionError()

        # Act & Assert
        with self.assertRaises(Exception) as context:
            fetch_user_data(1, self.mock_client)

        self.assertEqual(str(context.exception), "Network error")

# Pytest fixtures
@pytest.fixture
def sample_data():
    return [1, 2, 3, 4, 5]

@pytest.fixture
def analyzer(sample_data):
    return DataAnalyzer(sample_data)

def test_data_analyzer_mean(analyzer):
    assert analyzer.mean == 3.0

def test_data_analyzer_caching(analyzer):
    # First access computes
    mean1 = analyzer.mean
    # Second access uses cached value
    mean2 = analyzer.mean
    assert mean1 == mean2
    # Should only print "Computing mean..." once

# Parameterized tests
@pytest.mark.parametrize("input,expected", [
    ("123", 123),
    ("-45", -45),
    ("0", 0),
])
def test_parse_int_valid(input, expected):
    result = parse_int(input)
    assert result.is_ok
    assert result.unwrap() == expected

@pytest.mark.parametrize("input", ["abc", "12.3", "", "1e10"])
def test_parse_int_invalid(input):
    result = parse_int(input)
    assert result.is_err
```

Function Composition Patterns

python

```python
# Function composition utilities
def compose(*functions):
    """Compose functions right to left."""
    def inner(arg):
        result = arg
        for func in reversed(functions):
            result = func(result)
        return result
    return inner

def pipe(*functions):
    """Pipe functions left to right."""
    def inner(arg):
        result = arg
        for func in functions:
            result = func(result)
        return result
    return inner

# Example usage
def add_one(x): return x + 1
def multiply_two(x): return x * 2
def square(x): return x ** 2

# Composition: square(multiply_two(add_one(x)))
f = compose(square, multiply_two, add_one)
print(f(3))  # 64: (3+1)*2 = 8, 8^2 = 64

# Piping: square(multiply_two(add_one(x)))
g = pipe(add_one, multiply_two, square)
print(g(3))  # 64

# Partial application and currying
from functools import partial

def log_message(level, timestamp, message):
    return f"[{timestamp}] {level}: {message}"

# Create specialized functions
log_error = partial(log_message, "ERROR")
log_info = partial(log_message, "INFO")

# Use with current timestamp
from datetime import datetime
now = datetime.now().isoformat()
print(log_error(now, "Something went wrong"))
print(log_info(now, "Process started"))

# Currying decorator
def curry(func):
    """Convert function to curried form."""
    @functools.wraps(func)
    def curried(*args, **kwargs):
        if len(args) + len(kwargs) >= func.__code__.co_argcount:
            return func(*args, **kwargs)
        return partial(func, *args, **kwargs)
    return curried

@curry
def add3(a, b, c):
    return a + b + c

# Can be called in multiple ways
print(add3(1)(2)(3))     # 6
print(add3(1, 2)(3))     # 6
print(add3(1)(2, 3))     # 6
print(add3(1, 2, 3))     # 6

# Method chaining pattern
class QueryBuilder:
    def __init__(self):
        self._query = {}

    def select(self, *fields):
        self._query['select'] = fields
        return self

    def where(self, **conditions):
        self._query['where'] = conditions
        return self

    def order_by(self, field, direction='ASC'):
        self._query['order_by'] = (field, direction)
        return self

    def limit(self, count):
        self._query['limit'] = count
        return self

    def build(self):
        return self._query

# Fluent interface
```

```python
query = (QueryBuilder()
    .select('id', 'name', 'email')
    .where(active=True, role='admin')
    .order_by('created_at', 'DESC')
    .limit(10)
    .build())

print(query)
```

Context-Aware Functions

python

```python
import contextvars
from contextlib import contextmanager

# Context variables for request handling
request_id = contextvars.ContextVar('request_id', default=None)
user_context = contextvars.ContextVar('user_context', default=None)

@contextmanager
def request_context(req_id, user):
    """Set request context."""
    token1 = request_id.set(req_id)
    token2 = user_context.set(user)
    try:
        yield
    finally:
        request_id.reset(token1)
        user_context.reset(token2)

def log_with_context(message):
    """Log with request context."""
    req_id = request_id.get()
    user = user_context.get()

    context_str = ""
    if req_id:
        context_str += f"[{req_id}]"
    if user:
        context_str += f"[{user}]"

    print(f"{context_str} {message}")

# Usage
with request_context("req-123", "alice@example.com"):
    log_with_context("Processing request")  # [req-123][alice@example.com] Processing request

    # Context is maintained across function calls
    def nested_function():
        log_with_context("In nested function")

    nested_function()  # [req-123][alice@example.com] In nested function

# Outside context
log_with_context("No context")  # No context
```

Best Practices Summary

python

```python
# 1. Use descriptive names
def calc(x, y):  # Bad
    return x * y * 0.1

def calculate_discount_amount(price, quantity):  # Good
    return price * quantity * 0.1

# 2. Avoid side effects in functions that return values
# Bad
total = 0
def add_to_total(value):
    global total
    total += value
    return total

# Good
def add_to_total(current_total, value):
    return current_total + value

# 3. Use type hints
def process_data(data: List[Dict[str, Any]]) -> pd.DataFrame:
    """Process raw data into DataFrame."""
    pass

# 4. Handle errors explicitly
def safe_operation(value: str) -> Result[int, str]:
    """Safely convert string to int."""
    try:
        return Result.ok(int(value))
    except ValueError as e:
        return Result.err(str(e))

# 5. Keep functions small and focused
# Each function should do one thing well

# 6. Use default arguments carefully
def create_list(items=None):  # Good
    if items is None:
        items = []
    return items

# 7. Document edge cases
def divide(a: float, b: float) -> float:
    """
    Divide a by b.
    Raises:
        ZeroDivisionError: If b is zero
    """
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero")
    return a / b

# 8. Make functions testable
def get_user_age(user_id: int, database) -> int:
    """Get user age from database (injectable dependency)."""
    user = database.get_user(user_id)
    return user.age

# 9. Use functional programming concepts where appropriate
# Pure functions, immutability, function composition

# 10. Profile and optimize when necessary
# Don't optimize prematurely, but profile when performance matters
```

Conclusion

This comprehensive guide has covered every aspect of functions in Python, from basic definitions to advanced functional programming concepts. Functions are the building blocks of well-structured Python programs, and mastering them is essential for writing clean, maintainable, and efficient code.

Key takeaways:

- Functions promote code reuse and modularity
- Python supports multiple programming paradigms through functions
- Advanced features like decorators, generators, and async functions provide powerful abstractions
- Type hints improve code clarity and catch errors early
- Following best practices leads to more maintainable code

Continue practicing and exploring these concepts to become proficient in Python function programming!