Python Fundamentals Documentation

Table of Contents

Introduction

Python is a high-level, interpreted programming language known for its simplicity and readability. It's used for web development, data science, artificial intelligence, automation, and more. This documentation covers Python fundamentals to advanced concepts.

Key Characteristics

- **Interpreted**: No compilation step required
- **Dynamic typing**: Variables don't need explicit type declarations
- **Indentation-based**: Code blocks are defined by indentation
- **Multi-paradigm**: Supports procedural, object-oriented, and functional programming
- **Rich ecosystem**: Extensive standard library and third-party packages

---

Variables and Data Types

Variable Assignment

```
# Simple assignment
name = "John"
age = 30
is_student = True

# Multiple assignment
x, y, z = 1, 2, 3
a = b = c = 0

# Swapping variables
x, y = y, x
```

Numeric Types

```
# Integer
count = 42
large_number = 1_000_000  # Underscore for readability

# Float
temperature = 98.6
scientific = 1.5e-4  # Scientific notation

# Complex
complex_num = 3 + 4j

# Type conversion
int_val = int(3.14)      # 3
float_val = float(42)    # 42.0
str_val = str(123)       # "123"
```

String Operations

```
# String creation
single_quote = 'Hello'
double_quote = "World"
triple_quote = """Multi-line
string content"""

# String formatting
name = "Alice"
age = 25

# f-strings (Python 3.6+)
message = f"Hello, {name}! You are {age} years old."
```

```
# format() method
message = "Hello, {}! You are {} years old.".format(name, age)
message = "Hello, {name}! You are {age} years old.".format(name=name, age=age)

# % formatting (legacy)
message = "Hello, %s! You are %d years old." % (name, age)

# String methods
text = "  Hello World  "
print(text.strip())        # "Hello World"
print(text.upper())        # "  HELLO WORLD  "
print(text.lower())        # "  hello world  "
print(text.replace("World", "Python"))  # "  Hello Python  "
print(text.split())        # ["Hello", "World"]
```

Boolean and None

```
# Boolean values
is_active = True
is_complete = False

# None type
result = None

# Boolean operations
print(bool(0))         # False
print(bool(1))         # True
print(bool(""))        # False
print(bool("hello"))   # True
print(bool([]))        # False
print(bool([1, 2]))    # True
```

Type Checking

```
# type() function
print(type(42))        # <class 'int'>
print(type("hello"))   # <class 'str'>
print(type([1, 2, 3])) # <class 'list'>

# isinstance() function (preferred)
print(isinstance(42, int))     # True
print(isinstance("hi", str))   # True
print(isinstance([1, 2], list)) # True
```

Operators

Arithmetic Operators

```
a, b = 10, 3

print(a + b)   # 13 (addition)
print(a - b)   # 7 (subtraction)
print(a * b)   # 30 (multiplication)
print(a / b)   # 3.333... (division)
print(a // b)  # 3 (floor division)
print(a % b)   # 1 (modulo)
print(a ** b)  # 1000 (exponentiation)

# Augmented assignment
a += 5   # a = a + 5
a -= 2   # a = a - 2
a *= 3   # a = a * 3
a /= 2   # a = a / 2
```

Comparison Operators

```
# Equality and inequality
5 == 5    # True
5 != 3    # True
5 is 5    # True (identity)
5 is not 3 # True

# Relational operators
5 > 3     # True
5 < 3     # False
5 >= 5    # True
5 <= 3    # False

# Chained comparisons
1 < 2 < 3  # True
```

Logical Operators

```
# Logical operators
True and False  # False
True or False   # True
not True        # False

# Short-circuit evaluation
result = condition and expensive_function()
result = condition or default_value
```

Membership and Identity

```
# Membership operators
'a' in 'apple'     # True
'z' not in 'apple' # True
1 in [1, 2, 3]     # True

# Identity operators
a = [1, 2, 3]
b = a
c = [1, 2, 3]

a is b     # True (same object)
a is c     # False (different objects)
a == c     # True (same content)
```

Control Flow

Conditional Statements

```
# if-elif-else
score = 85

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

# Ternary operator
status = "pass" if score >= 60 else "fail"

# Multiple conditions
if age >= 18 and has_license:
    print("Can drive")

if weather == "sunny" or weather == "cloudy":
    print("Good day for outdoor activities")
```

Loops

For Loops

```
# Iterate over sequence
fruits = ["apple", "banana", "orange"]
for fruit in fruits:
    print(fruit)

# Iterate with index
for i, fruit in enumerate(fruits):
    print(f"{i}: {fruit}")

# Range function
for i in range(5):          # 0, 1, 2, 3, 4
    print(i)

for i in range(1, 6):       # 1, 2, 3, 4, 5
    print(i)

for i in range(0, 10, 2):   # 0, 2, 4, 6, 8
    print(i)

# Dictionary iteration
person = {"name": "John", "age": 30, "city": "New York"}

for key in person:
    print(key, person[key])

for key, value in person.items():
    print(f"{key}: {value}")

for value in person.values():
    print(value)
```

While Loops

```
count = 0
while count < 5:
    print(count)
    count += 1

# While with else
count = 0
while count < 3:
    print(count)
    count += 1
```

```
else:
    print("Loop completed normally")
```

Loop Control

```
# Break and continue
for i in range(10):
    if i == 3:
        continue  # Skip iteration
    if i == 7:
        break     # Exit loop
    print(i)

# Nested loops with labels (using functions)
def nested_loop_example():
    for i in range(3):
        for j in range(3):
            if i == 1 and j == 1:
                return  # Exit both loops
            print(f"i={i}, j={j}")

nested_loop_example()
```

Functions

Function Definition

```
# Basic function
def greet(name):
    return f"Hello, {name}!"

# Function with default parameters
def greet(name="World"):
    return f"Hello, {name}!"

# Function with multiple parameters
def calculate_area(length, width):
    return length * width

# Function with keyword arguments
def create_profile(name, age, city="Unknown", country="Unknown"):
    return {
        "name": name,
        "age": age,
        "city": city,
```

```
        "country": country
    }

profile = create_profile("John", 30, country="USA")
```

Variable Arguments

```
# *args - variable positional arguments
def sum_all(*args):
    return sum(args)

result = sum_all(1, 2, 3, 4, 5)  # 15

# **kwargs - variable keyword arguments
def create_person(**kwargs):
    return kwargs

person = create_person(name="John", age=30, city="NYC")

# Combining all parameter types
def complex_function(required, default="default", *args, **kwargs):
    print(f"Required: {required}")
    print(f"Default: {default}")
    print(f"Args: {args}")
    print(f"Kwargs: {kwargs}")

complex_function("test", "custom", 1, 2, 3, key1="value1", key2="value2")
```

Advanced Function Features

```
# Function as first-class objects
def multiply(x, y):
    return x * y

def apply_operation(func, a, b):
    return func(a, b)

result = apply_operation(multiply, 5, 3)  # 15

# Lambda functions
square = lambda x: x ** 2
add = lambda x, y: x + y

numbers = [1, 2, 3, 4, 5]
squared = list(map(square, numbers))  # [1, 4, 9, 16, 25]

# Closure
```

```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

add_10 = outer_function(10)
result = add_10(5)  # 15

# Recursive functions
def factorial(n):
    if n <= 1:
        return 1
    return n * factorial(n - 1)

print(factorial(5))  # 120
```

Function Annotations

```
# Type hints (Python 3.5+)
def add_numbers(a: int, b: int) -> int:
    return a + b

def greet_user(name: str, age: int = 0) -> str:
    return f"Hello {name}, you are {age} years old"

# More complex type hints
from typing import List, Dict, Optional, Union

def process_data(data: List[int]) -> Dict[str, int]:
    return {
        "count": len(data),
        "sum": sum(data),
        "max": max(data) if data else 0
    }

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

Data Structures

Lists

```
# List creation
numbers = [1, 2, 3, 4, 5]
```

```
mixed = [1, "hello", 3.14, True]
empty = []

# List methods
fruits = ["apple", "banana"]
fruits.append("orange")          # Add to end
fruits.insert(1, "grape")        # Insert at index
fruits.remove("banana")          # Remove first occurrence
popped = fruits.pop()            # Remove and return last item
fruits.extend(["mango", "kiwi"]) # Add multiple items

# List operations
numbers = [1, 2, 3, 4, 5]
print(len(numbers))              # 5
print(numbers[0])                # 1 (first element)
print(numbers[-1])               # 5 (last element)
print(numbers[1:4])              # [2, 3, 4] (slicing)
print(numbers[:3])               # [1, 2, 3] (from start)
print(numbers[2:])               # [3, 4, 5] (to end)
print(numbers[::2])              # [1, 3, 5] (step)

# List comprehensions
squares = [x**2 for x in range(10)]
even_squares = [x**2 for x in range(10) if x % 2 == 0]
matrix = [[i*j for j in range(3)] for i in range(3)]
```

Tuples

```
# Tuple creation
point = (3, 4)
colors = ("red", "green", "blue")
single_item = (42,)  # Comma needed for single item

# Tuple operations
x, y = point  # Unpacking
print(point[0])      # 3
print(len(colors))   # 3

# Named tuples
from collections import namedtuple

Person = namedtuple('Person', ['name', 'age', 'city'])
john = Person("John", 30, "New York")
print(john.name)     # "John"
print(john.age)      # 30
```

Sets

```
# Set creation
numbers = {1, 2, 3, 4, 5}
colors = set(["red", "green", "blue", "red"])  # Duplicates removed

# Set operations
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

print(set1 | set2)   # {1, 2, 3, 4, 5, 6} (union)
print(set1 & set2)   # {3, 4} (intersection)
print(set1 - set2)   # {1, 2} (difference)
print(set1 ^ set2)   # {1, 2, 5, 6} (symmetric difference)

# Set methods
numbers = {1, 2, 3}
numbers.add(4)
numbers.remove(2)  # Raises KeyError if not found
numbers.discard(5)  # No error if not found
```

Dictionaries

```
# Dictionary creation
person = {"name": "John", "age": 30, "city": "New York"}
empty_dict = {}

# Dictionary operations
print(person["name"])           # "John"
print(person.get("age"))        # 30
print(person.get("country", "Unknown"))  # "Unknown" (default)

person["email"] = "john@example.com"  # Add new key-value pair
del person["city"]              # Delete key-value pair

# Dictionary methods
keys = person.keys()
values = person.values()
items = person.items()

# Dictionary comprehensions
squares = {x: x**2 for x in range(5)}
filtered = {k: v for k, v in person.items() if len(str(v)) > 3}

# Nested dictionaries
employees = {
    "john": {"age": 30, "department": "IT"},
    "jane": {"age": 25, "department": "HR"}
```

```
}
```

Collections Module

```
from collections import defaultdict, Counter, deque, OrderedDict

# defaultdict
dd = defaultdict(list)
dd['key'].append('value')  # No KeyError

# Counter
text = "hello world"
counter = Counter(text)
print(counter)  # Counter({'l': 3, 'o': 2, 'h': 1, 'e': 1, ' ': 1, 'w': 1, 'r': 1, 'd': 1})

# deque (double-ended queue)
dq = deque([1, 2, 3])
dq.appendleft(0)  # Add to left
dq.append(4)      # Add to right
```

Object-Oriented Programming

Classes and Objects

```
class Person:
    # Class variable
    species = "Homo sapiens"

    def __init__(self, name, age):
        # Instance variables
        self.name = name
        self.age = age

    def greet(self):
        return f"Hello, I'm {self.name}"

    def have_birthday(self):
        self.age += 1

    def __str__(self):
        return f"Person(name='{self.name}', age={self.age})"

    def __repr__(self):
        return f"Person('{self.name}', {self.age})"
```

```
# Creating objects
john = Person("John", 30)
jane = Person("Jane", 25)

print(john.greet())  # "Hello, I'm John"
john.have_birthday()
print(john.age)      # 31
```

Inheritance

```
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def make_sound(self):
        return "Some generic sound"

    def info(self):
        return f"{self.name} is a {self.species}"

class Dog(Animal):
    def __init__(self, name, breed):
        super().__init__(name, "Canine")
        self.breed = breed

    def make_sound(self):
        return "Woof!"

    def fetch(self):
        return f"{self.name} is fetching the ball"

class Cat(Animal):
    def __init__(self, name, color):
        super().__init__(name, "Feline")
        self.color = color

    def make_sound(self):
        return "Meow!"

    def climb(self):
        return f"{self.name} is climbing"

# Using inheritance
rex = Dog("Rex", "Golden Retriever")
whiskers = Cat("Whiskers", "Orange")
```

```
print(rex.make_sound())  # "Woof!"
print(rex.fetch())       # "Rex is fetching the ball"
print(whiskers.make_sound())  # "Meow!"
```

Encapsulation

```
class BankAccount:
    def __init__(self, account_number, initial_balance=0):
        self.account_number = account_number
        self._balance = initial_balance  # Protected attribute
        self.__transaction_history = []  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            self.__transaction_history.append(f"Deposited ${amount}")
            return True
        return False

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            self.__transaction_history.append(f"Withdrew ${amount}")
            return True
        return False

    def get_balance(self):
        return self._balance

    def _get_transaction_history(self):  # Protected method
        return self.__transaction_history.copy()

account = BankAccount("123456", 1000)
account.deposit(500)
print(account.get_balance())  # 1500
```

Properties and Decorators

```
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
```

```
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature below absolute zero is not possible")
        self._celsius = value

    @property
    def fahrenheit(self):
        return (self._celsius * 9/5) + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) * 5/9

    @property
    def kelvin(self):
        return self._celsius + 273.15

temp = Temperature(25)
print(temp.fahrenheit)  # 77.0
temp.fahrenheit = 86
print(temp.celsius)     # 30.0
```

Special Methods (Magic Methods)

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        return Vector(self.x * scalar, self.y * scalar)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
```

```
    def __len__(self):
        return int((self.x**2 + self.y**2)**0.5)

    def __getitem__(self, index):
        if index == 0:
            return self.x
        elif index == 1:
            return self.y
        else:
            raise IndexError("Vector index out of range")

v1 = Vector(3, 4)
v2 = Vector(1, 2)
v3 = v1 + v2  # Uses __add__
print(v3)     # Vector(4, 6)
```

Abstract Base Classes

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2
```

```
    def perimeter(self):
        return 2 * 3.14159 * self.radius

# Cannot instantiate abstract class
# shape = Shape()  # TypeError

rect = Rectangle(5, 3)
circle = Circle(2)
```

File Handling

Reading Files

```
# Basic file reading
with open('example.txt', 'r') as file:
    content = file.read()
    print(content)

# Reading line by line
with open('example.txt', 'r') as file:
    for line in file:
        print(line.strip())

# Reading all lines into a list
with open('example.txt', 'r') as file:
    lines = file.readlines()

# Reading with encoding
with open('example.txt', 'r', encoding='utf-8') as file:
    content = file.read()
```

Writing Files

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

# Writing multiple lines
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
with open('output.txt', 'w') as file:
    file.writelines(lines)

# Appending to file
with open('output.txt', 'a') as file:
```

```
    file.write("This line is appended.\n")
```

Working with CSV Files

```
import csv

# Reading CSV
with open('data.csv', 'r') as file:
    csv_reader = csv.reader(file)
    for row in csv_reader:
        print(row)

# Reading CSV with headers
with open('data.csv', 'r') as file:
    csv_reader = csv.DictReader(file)
    for row in csv_reader:
        print(row['name'], row['age'])

# Writing CSV
data = [
    ['Name', 'Age', 'City'],
    ['John', 30, 'New York'],
    ['Jane', 25, 'Los Angeles']
]

with open('output.csv', 'w', newline='') as file:
    csv_writer = csv.writer(file)
    csv_writer.writerows(data)
```

Working with JSON

```
import json

# Reading JSON
with open('data.json', 'r') as file:
    data = json.load(file)

# Writing JSON
data = {
    'name': 'John',
    'age': 30,
    'city': 'New York'
}

with open('output.json', 'w') as file:
    json.dump(data, file, indent=2)
```

```
# JSON string operations
json_string = json.dumps(data, indent=2)
parsed_data = json.loads(json_string)
```

Error Handling

Basic Exception Handling

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero!")
except ValueError:
    print("Invalid value!")
except Exception as e:
    print(f"An error occurred: {e}")
else:
    print("No exception occurred")
finally:
    print("This always executes")
```

Custom Exceptions

```
class CustomError(Exception):
    pass

class ValidationError(Exception):
    def __init__(self, message, code=None):
        super().__init__(message)
        self.code = code

def validate_age(age):
    if age < 0:
        raise ValidationError("Age cannot be negative", code="NEGATIVE_AGE")
    if age > 150:
        raise ValidationError("Age cannot exceed 150", code="INVALID_AGE")
    return True

try:
    validate_age(-5)
except ValidationError as e:
    print(f"Validation error: {e}")
    print(f"Error code: {e.code}")
```

Exception Handling Best Practices

```
import logging

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def safe_divide(a, b):
    try:
        result = a / b
        logger.info(f"Division successful: {a} / {b} = {result}")
        return result
    except ZeroDivisionError:
        logger.error("Division by zero attempted")
        raise
    except TypeError:
        logger.error("Invalid types for division")
        raise
    except Exception as e:
        logger.error(f"Unexpected error: {e}")
        raise

# Context manager for resource management
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()
        if exc_type is not None:
            print(f"Exception occurred: {exc_val}")
        return False  # Don't suppress exceptions

# Using custom context manager
with FileManager('test.txt', 'w') as f:
    f.write('Hello, World!')
```

Modules and Packages

Importing Modules

```
# Different ways to import
import math
import math as m
from math import sqrt, pi
from math import *  # Not recommended

# Using imported modules
print(math.sqrt(16))  # 4.0
print(m.pi)          # 3.14159...
print(sqrt(25))      # 5.0
```

Creating Modules

```
# mathutils.py
def add(a, b):
    return a + b

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

PI = 3.14159

class Calculator:
    def __init__(self):
        self.history = []

    def add(self, a, b):
        result = a + b
        self.history.append(f"{a} + {b} = {result}")
        return result

# Using the module
# main.py
import mathutils
from mathutils import Calculator

result = mathutils.add(5, 3)
calc = Calculator()
calc.add(10, 20)
```

Packages

```
# Package structure:
```

```
# mypackage/
#   __init__.py
#   module1.py
#   module2.py
#   subpackage/
#     __init__.py
#     module3.py

# mypackage/__init__.py
from .module1 import function1
from .module2 import function2

__all__ = ['function1', 'function2']

# Using the package
from mypackage import function1
import mypackage.subpackage.module3
```

Standard Library Modules

```
# datetime
from datetime import datetime, date, timedelta

now = datetime.now()
today = date.today()
tomorrow = today + timedelta(days=1)

# os and os.path
import os
import os.path

current_dir = os.getcwd()
files = os.listdir('.')
if os.path.exists('file.txt'):
    print("File exists")

# random
import random

random_num = random.random()  # 0.0 to 1.0
random_int = random.randint(1, 10)
choice = random.choice(['apple', 'banana', 'orange'])

# urllib for HTTP requests
import urllib.request
import urllib.parse
```

```
response = urllib.request.urlopen('https://api.example.com/data')
data = response.read()
```

Decorators

Function Decorators

```
def my_decorator(func):
    def wrapper(*args, **kwargs):
        print("Something is happening before the function is called.")
        result = func(*args, **kwargs)
        print("Something is happening after the function is called.")
        return result
    return wrapper

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

say_hello()
```

Decorators with Arguments

```
def repeat(times):
    def decorator(func):
        def wrapper(*args, **kwargs):
            for _ in range(times):
                result = func(*args, **kwargs)
            return result
        return wrapper
    return decorator

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

greet("Alice")  # Prints "Hello, Alice!" three times
```

Practical Decorators

```
import time
import functools

# Timing decorator
def timer(func):
```

```
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.4f} seconds")
        return result
    return wrapper

# Caching decorator
def cache(func):
    cached_results = {}

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

# Authentication decorator
def requires_auth(func):
    @functools.wraps(func)
    def wrapper(*args, **kwargs):
        # In real application, check authentication
        authenticated = True  # Simplified
        if not authenticated:
            raise PermissionError("Authentication required")
        return func(*args, **kwargs)
    return wrapper

@timer
@cache
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Class decorators
def singleton(cls):
    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("Database instance created")

# Property decorators
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        return self._radius

    @radius.setter
    def radius(self, value):
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

    @property
    def area(self):
        return 3.14159 * self._radius ** 2

    @staticmethod
    def from_diameter(diameter):
        return Circle(diameter / 2)

    @classmethod
    def unit_circle(cls):
        return cls(1)
```

Generators and Iterators

Generators

```
# Generator function
def countdown(n):
    while n > 0:
        yield n
        n -= 1

# Using generator
for i in countdown(5):
    print(i)  # 5, 4, 3, 2, 1
```

```
# Generator expression
squares = (x**2 for x in range(10))
print(list(squares))  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

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

# Using infinite generator
fib = fibonacci()
for _ in range(10):
    print(next(fib))

# Generator with send()
def receiver():
    while True:
        value = yield
        print(f"Received: {value}")

gen = receiver()
next(gen)  # Prime the generator
gen.send("Hello")
gen.send("World")
```

Custom Iterators

```
class NumberSequence:
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def __iter__(self):
        return self

    def __next__(self):
        if self.start >= self.end:
            raise StopIteration
        current = self.start
        self.start += 1
        return current

# Using custom iterator
seq = NumberSequence(1, 5)
```

```
for num in seq:
    print(num)  # 1, 2, 3, 4

# Iterator tools
from itertools import count, cycle, repeat, chain, combinations

# Infinite iterators
counter = count(10, 2)  # 10, 12, 14, 16, ...
cycler = cycle(['A', 'B', 'C'])  # A, B, C, A, B, C, ...
repeater = repeat('hello', 3)  # hello, hello, hello

# Finite iterators
list1 = [1, 2, 3]
list2 = [4, 5, 6]
chained = chain(list1, list2)  # [1, 2, 3, 4, 5, 6]

# Combinations
items = ['A', 'B', 'C']
combos = combinations(items, 2)  # [('A', 'B'), ('A', 'C'), ('B', 'C')]
```

Asynchronous Programming

Basic Async/Await

```
import asyncio
import aiohttp
import time

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

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

# Async with multiple tasks
async def task1():
    print("Task 1 starting")
    await asyncio.sleep(2)
    print("Task 1 completed")
    return "Task 1 result"

async def task2():
```

```
    print("Task 2 starting")
    await asyncio.sleep(1)
    print("Task 2 completed")
    return "Task 2 result"

async def main():
    # Run tasks concurrently
    result1, result2 = await asyncio.gather(task1(), task2())
    print(f"Results: {result1}, {result2}")

# asyncio.run(main())
```

Async HTTP Requests

```
import aiohttp
import asyncio
import time

async def fetch_url(session, url):
    try:
        async with session.get(url) as response:
            return await response.text()
    except Exception as e:
        print(f"Error fetching {url}: {e}")
        return None

async def fetch_multiple_urls(urls):
    async with aiohttp.ClientSession() as session:
        tasks = [fetch_url(session, url) for url in urls]
        results = await asyncio.gather(*tasks)
        return results

# Example usage
urls = [
    'https://httpbin.org/delay/1',
    'https://httpbin.org/delay/2',
    'https://httpbin.org/delay/3'
]

# Synchronous version for comparison
def sync_fetch(url):
    import requests
    try:
        response = requests.get(url)
        return response.text
    except Exception as e:
        print(f"Error: {e}")
```

```
        return None

# Timing comparison
async def async_timing_test():
    start = time.time()
    results = await fetch_multiple_urls(urls)
    end = time.time()
    print(f"Async version took: {end - start:.2f} seconds")

def sync_timing_test():
    start = time.time()
    results = [sync_fetch(url) for url in urls]
    end = time.time()
    print(f"Sync version took: {end - start:.2f} seconds")
```

Async Context Managers

```
class AsyncDatabaseConnection:
    def __init__(self, db_url):
        self.db_url = db_url
        self.connection = None

    async def __aenter__(self):
        print(f"Connecting to {self.db_url}")
        await asyncio.sleep(0.1)  # Simulate connection time
        self.connection = f"Connected to {self.db_url}"
        return self

    async def __aexit__(self, exc_type, exc_val, exc_tb):
        print("Closing database connection")
        await asyncio.sleep(0.1)  # Simulate cleanup time
        self.connection = None

    async def query(self, sql):
        if not self.connection:
            raise Exception("Not connected to database")
        print(f"Executing query: {sql}")
        await asyncio.sleep(0.1)  # Simulate query time
        return f"Result for: {sql}"

async def database_example():
    async with AsyncDatabaseConnection("postgresql://localhost/mydb") as db:
        result = await db.query("SELECT * FROM users")
        print(result)

# asyncio.run(database_example())
```

Async Generators

```
async def async_range(n):
    for i in range(n):
        await asyncio.sleep(0.1)
        yield i

async def async_generator_example():
    async for value in async_range(5):
        print(f"Got value: {value}")

# asyncio.run(async_generator_example())

# Async list comprehension
async def async_squares(n):
    return [i**2 async for i in async_range(n)]

# Queue for async producer/consumer
async def producer(queue):
    for i in range(5):
        await asyncio.sleep(0.1)
        await queue.put(f"item_{i}")
    await queue.put(None)  # Signal end

async def consumer(queue):
    while True:
        item = await queue.get()
        if item is None:
            break
        print(f"Consumed: {item}")
        await asyncio.sleep(0.2)

async def producer_consumer_example():
    queue = asyncio.Queue()
    await asyncio.gather(
        producer(queue),
        consumer(queue)
    )

# asyncio.run(producer_consumer_example())
```

Multithreading and Multiprocessing

Threading

```
import threading
```

```
import time
import queue

# Basic threading
def worker(name, delay):
    print(f"Worker {name} starting")
    time.sleep(delay)
    print(f"Worker {name} finished")

# Create and start threads
threads = []
for i in range(3):
    t = threading.Thread(target=worker, args=(f"Thread-{i}", i+1))
    threads.append(t)
    t.start()

# Wait for all threads to complete
for t in threads:
    t.join()

print("All threads completed")
```

Thread Safety

```
import threading
import time

# Thread-safe counter
class ThreadSafeCounter:
    def __init__(self):
        self._value = 0
        self._lock = threading.Lock()

    def increment(self):
        with self._lock:
            self._value += 1

    def get_value(self):
        with self._lock:
            return self._value

counter = ThreadSafeCounter()

def increment_counter(n):
    for _ in range(n):
        counter.increment()
```

```
# Create multiple threads
threads = []
for i in range(5):
    t = threading.Thread(target=increment_counter, args=(1000,))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print(f"Final counter value: {counter.get_value()}")

# Using threading.local for thread-local storage
thread_local_data = threading.local()

def process_data():
    thread_local_data.value = threading.current_thread().name
    time.sleep(1)
    print(f"Thread {thread_local_data.value} processed data")

threads = []
for i in range(3):
    t = threading.Thread(target=process_data, name=f"Worker-{i}")
    threads.append(t)
    t.start()

for t in threads:
    t.join()
```

Producer-Consumer Pattern

```
import threading
import queue
import time
import random

def producer(q, producer_id):
    for i in range(5):
        item = f"Item-{producer_id}-{i}"
        q.put(item)
        print(f"Producer {producer_id} produced {item}")
        time.sleep(random.uniform(0.1, 0.5))
    q.put(None)  # Signal end

def consumer(q, consumer_id):
    while True:
        item = q.get()
```

```
        if item is None:
            q.put(None)  # Re-add sentinel for other consumers
            break
        print(f"Consumer {consumer_id} consumed {item}")
        time.sleep(random.uniform(0.1, 0.5))
        q.task_done()

# Create queue
q = queue.Queue()

# Create and start threads
producers = []
consumers = []

for i in range(2):
    p = threading.Thread(target=producer, args=(q, i))
    producers.append(p)
    p.start()

for i in range(3):
    c = threading.Thread(target=consumer, args=(q, i))
    consumers.append(c)
    c.start()

# Wait for producers
for p in producers:
    p.join()

# Wait for consumers
for c in consumers:
    c.join()
```

Multiprocessing

```
import multiprocessing
import time
import os

def worker_process(name, delay):
    print(f"Process {name} (PID: {os.getpid()}) starting")
    time.sleep(delay)
    print(f"Process {name} (PID: {os.getpid()}) finished")
    return f"Result from {name}"

# Basic multiprocessing
if __name__ == "__main__":
    processes = []
```

```
    for i in range(3):
        p = multiprocessing.Process(target=worker_process, args=(f"Process-{i}", i+1))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()

# Process pool
def cpu_intensive_task(n):
    total = 0
    for i in range(n):
        total += i ** 2
    return total

if __name__ == "__main__":
    with multiprocessing.Pool() as pool:
        tasks = [1000000, 2000000, 3000000, 4000000]
        results = pool.map(cpu_intensive_task, tasks)
        print(f"Results: {results}")

# Shared memory
def increment_shared_value(shared_value, lock):
    for _ in range(1000):
        with lock:
            shared_value.value += 1

if __name__ == "__main__":
    shared_value = multiprocessing.Value('i', 0)
    lock = multiprocessing.Lock()

    processes = []
    for i in range(4):
        p = multiprocessing.Process(target=increment_shared_value, args=(shared_value,
lock))
        processes.append(p)
        p.start()

    for p in processes:
        p.join()

    print(f"Final value: {shared_value.value}")
```

Concurrent.futures

```
from concurrent.futures import ThreadPoolExecutor, ProcessPoolExecutor, as_completed
import time
```

```
import requests

def fetch_url(url):
    try:
        response = requests.get(url, timeout=5)
        return f"{url}: {response.status_code}"
    except Exception as e:
        return f"{url}: Error - {e}"

urls = [
    'https://httpbin.org/delay/1',
    'https://httpbin.org/delay/2',
    'https://httpbin.org/delay/3',
    'https://httpbin.org/status/200',
    'https://httpbin.org/status/404'
]

# Thread pool executor
with ThreadPoolExecutor(max_workers=3) as executor:
    futures = [executor.submit(fetch_url, url) for url in urls]

    for future in as_completed(futures):
        result = future.result()
        print(result)

# Process pool executor for CPU-intensive tasks
def cpu_bound_task(n):
    return sum(i * i for i in range(n))

if __name__ == "__main__":
    with ProcessPoolExecutor(max_workers=4) as executor:
        futures = [executor.submit(cpu_bound_task, n) for n in [100000, 200000, 300000]]

        for future in as_completed(futures):
            result = future.result()
            print(f"Result: {result}")
```

Regular Expressions

Basic Regex Operations

```
import re

# Basic pattern matching
text = "The quick brown fox jumps over the lazy dog"
```

```
pattern = r"fox"

# Search for pattern
match = re.search(pattern, text)
if match:
    print(f"Found '{match.group()}' at position {match.start()}")

# Find all matches
pattern = r"\w+"  # Match all words
matches = re.findall(pattern, text)
print(matches)

# Replace patterns
new_text = re.sub(r"fox", "cat", text)
print(new_text)
```

Advanced Regex Patterns

```
# Email validation
email_pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}

def validate_email(email):
    return re.match(email_pattern, email) is not None

print(validate_email("user@example.com"))  # True
print(validate_email("invalid-email"))     # False

# Phone number extraction
text = "Call me at 123-456-7890 or (555) 123-4567"
phone_pattern = r'(\(?\d{3}\)?[-.\s]?\d{3}[-.\s]?\d{4})'
phones = re.findall(phone_pattern, text)
print(phones)  # ['123-456-7890', '(555) 123-4567']

# URL extraction
url_pattern =
r'https?://(?:[-\w.])+(?:\:[0-9]+)?(?:/(?:[\w/_.])*(?:\?(?:[\w&=%.])*)?(?:\#(?:[\w.])*)?)?'
text = "Visit https://example.com or http://test.org/path?param=value"
urls = re.findall(url_pattern, text)
print(urls)

# Groups and named groups
log_pattern = r'(?P<ip>\d+\.\d+\.\d+\.\d+) - - \[(?P<datetime>[^\]]+)\] "(?P<method>\w+)
(?P<path>[^\s]+) HTTP/1\.[01]" (?P<status>\d+) (?P<size>\d+)'
log_line = '192.168.1.1 - - [25/Dec/2023:10:00:00 +0000] "GET /index.html HTTP/1.1" 200
1234'

match = re.match(log_pattern, log_line)
```

```
if match:
    print(match.groupdict())
```

Regex Compilation and Flags

```
# Compile regex for better performance
compiled_pattern = re.compile(r'\d+')
numbers = compiled_pattern.findall("There are 123 apples and 456 oranges")
print(numbers)  # ['123', '456']

# Regex flags
text = "Hello WORLD"
# Case insensitive
matches = re.findall(r'world', text, re.IGNORECASE)
print(matches)  # ['WORLD']

# Multiline and dotall
multiline_text = """Line 1
Line 2
Line 3"""

# Multiline flag
pattern = re.compile(r'^Line', re.MULTILINE)
matches = pattern.findall(multiline_text)
print(matches)  # ['Line', 'Line', 'Line']

# Verbose flag for readable regex
verbose_pattern = re.compile(r'''
    ^                   # Start of string
    ([a-zA-Z0-9._%+-]+)  # Username
    @                   # @ symbol
    ([a-zA-Z0-9.-]+)     # Domain
    \.                  # Dot
    ([a-zA-Z]{2,})       # Top-level domain
    $                   # End of string
''', re.VERBOSE)

email = "user@example.com"
match = verbose_pattern.match(email)
if match:
    print(f"Username: {match.group(1)}")
    print(f"Domain: {match.group(2)}")
    print(f"TLD: {match.group(3)}")
```

Working with APIs

Making HTTP Requests

```
import requests
import json

# GET request
response = requests.get('https://jsonplaceholder.typicode.com/posts/1')
print(f"Status Code: {response.status_code}")
print(f"Headers: {response.headers}")
print(f"Content: {response.json()}")

# POST request
data = {
    'title': 'My New Post',
    'body': 'This is the content of my post',
    'userId': 1
}

response = requests.post('https://jsonplaceholder.typicode.com/posts', json=data)
print(f"Created post: {response.json()}")

# PUT request
updated_data = {
    'id': 1,
    'title': 'Updated Post',
    'body': 'This is the updated content',
    'userId': 1
}

response = requests.put('https://jsonplaceholder.typicode.com/posts/1', json=updated_data)
print(f"Updated post: {response.json()}")

# DELETE request
response = requests.delete('https://jsonplaceholder.typicode.com/posts/1')
print(f"Delete status: {response.status_code}")
```

Handling Authentication

```
# Basic authentication
from requests.auth import HTTPBasicAuth

response = requests.get('https://api.example.com/data',
                        auth=HTTPBasicAuth('username', 'password'))

# Bearer token authentication
```

```
headers = {'Authorization': 'Bearer your-token-here'}
response = requests.get('https://api.example.com/data', headers=headers)

# API key authentication
params = {'api_key': 'your-api-key'}
response = requests.get('https://api.example.com/data', params=params)
```

Error Handling and Best Practices

```
import requests
from requests.exceptions import RequestException, Timeout, ConnectionError
import time

def make_api_request(url, max_retries=3, backoff_factor=1):
    for attempt in range(max_retries):
        try:
            response = requests.get(url, timeout=10)
            response.raise_for_status()  # Raise exception for bad status codes
            return response.json()

        except Timeout:
            print(f"Timeout on attempt {attempt + 1}")
        except ConnectionError:
            print(f"Connection error on attempt {attempt + 1}")
        except requests.exceptions.HTTPError as e:
            print(f"HTTP error: {e}")
            if response.status_code == 429:  # Rate limited
                time.sleep(backoff_factor * (2 ** attempt))
                continue
            else:
                break
        except RequestException as e:
            print(f"Request exception: {e}")

        if attempt < max_retries - 1:
            time.sleep(backoff_factor * (2 ** attempt))

    return None

# Usage
data = make_api_request('https://api.example.com/data')
if data:
    print(data)
else:
    print("Failed to get data after retries")
```

Creating REST API Client

```
class APIClient:
    def __init__(self, base_url, api_key=None):
        self.base_url = base_url.rstrip('/')
        self.session = requests.Session()
        if api_key:
            self.session.headers.update({'Authorization': f'Bearer {api_key}'})

    def _make_request(self, method, endpoint, **kwargs):
        url = f"{self.base_url}/{endpoint.lstrip('/')}"
        try:
            response = self.session.request(method, url, **kwargs)
            response.raise_for_status()
            return response.json()
        except requests.exceptions.RequestException as e:
            print(f"API request failed: {e}")
            return None

    def get(self, endpoint, params=None):
        return self._make_request('GET', endpoint, params=params)

    def post(self, endpoint, data=None, json=None):
        return self._make_request('POST', endpoint, data=data, json=json)

    def put(self, endpoint, data=None, json=None):
        return self._make_request('PUT', endpoint, data=data, json=json)

    def delete(self, endpoint):
        return self._make_request('DELETE', endpoint)

# Usage
client = APIClient('https://jsonplaceholder.typicode.com')
posts = client.get('posts')
new_post = client.post('posts', json={'title': 'New Post', 'body': 'Content'})
```

Database Operations

SQLite Database

```
import sqlite3
from contextlib import contextmanager

# Database connection context manager
@contextmanager
def get_db_connection(db_name):
```

```
    conn = sqlite3.connect(db_name)
    try:
        yield conn
    finally:
        conn.close()

# Database operations
def create_table():
    with get_db_connection('example.db') as conn:
        cursor = conn.cursor()
        cursor.execute('''
            CREATE TABLE IF NOT EXISTS users (
                id INTEGER PRIMARY KEY AUTOINCREMENT,
                name TEXT NOT NULL,
                email TEXT UNIQUE NOT NULL,
                age INTEGER
            )
        ''')
        conn.commit()

def insert_user(name, email, age):
    with get_db_connection('example.db') as conn:
        cursor = conn.cursor()
        cursor.execute(
            'INSERT INTO users (name, email, age) VALUES (?, ?, ?)',
            (name, email, age)
        )
        conn.commit()
        return cursor.lastrowid

def get_all_users():
    with get_db_connection('example.db') as conn:
        cursor = conn.cursor()
        cursor.execute('SELECT * FROM users')
        return cursor.fetchall()

def get_user_by_id(user_id):
    with get_db_connection('example.db') as conn:
        cursor = conn.cursor()
        cursor.execute('SELECT * FROM users WHERE id = ?', (user_id,))
        return cursor.fetchone()

def update_user(user_id, name=None, email=None, age=None):
    with get_db_connection('example.db') as conn:
        cursor = conn.cursor()
        updates = []
        params = []
```

```
        if name:
            updates.append('name = ?')
            params.append(name)
        if email:
            updates.append('email = ?')
            params.append(email)
        if age:
            updates.append('age = ?')
            params.append(age)

        if updates:
            query = f'UPDATE users SET {", ".join(updates)} WHERE id = ?'
            params.append(user_id)
            cursor.execute(query, params)
            conn.commit()

def delete_user(user_id):
    with get_db_connection('example.db') as conn:
        cursor = conn.cursor()
        cursor.execute('DELETE FROM users WHERE id = ?', (user_id,))
        conn.commit()

# Usage
create_table()
user_id = insert_user('John Doe', 'john@example.com', 30)
users = get_all_users()
print(users)
```

Database Class

```
class Database:
    def __init__(self, db_name):
        self.db_name = db_name
        self.init_database()

    def init_database(self):
        with sqlite3.connect(self.db_name) as conn:
            conn.execute('''
                CREATE TABLE IF NOT EXISTS users (
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    name TEXT NOT NULL,
                    email TEXT UNIQUE NOT NULL,
                    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                )
            ''')
            conn.commit()
```

```
    def execute_query(self, query, params=None):
        with sqlite3.connect(self.db_name) as conn:
            cursor = conn.cursor()
            if params:
                cursor.execute(query, params)
            else:
                cursor.execute(query)
            conn.commit()
            return cursor.fetchall()

    def create_user(self, name, email):
        query = 'INSERT INTO users (name, email) VALUES (?, ?)'
        self.execute_query(query, (name, email))

    def get_users(self):
        query = 'SELECT * FROM users'
        return self.execute_query(query)

    def find_user(self, email):
        query = 'SELECT * FROM users WHERE email = ?'
        result = self.execute_query(query, (email,))
        return result[0] if result else None

# Usage
db = Database('app.db')
db.create_user('Alice', 'alice@example.com')
users = db.get_users()
print(users)
```

Testing

Unit Testing with unittest

```
import unittest
from unittest.mock import patch, MagicMock

# Code to test
class Calculator:
    def add(self, a, b):
        return a + b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Cannot divide by zero")
        return a / b
```

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

class TestCalculator(unittest.TestCase):
    def setUp(self):
        self.calc = Calculator()

    def test_add(self):
        result = self.calc.add(2, 3)
        self.assertEqual(result, 5)

    def test_divide(self):
        result = self.calc.divide(10, 2)
        self.assertEqual(result, 5)

    def test_divide_by_zero(self):
        with self.assertRaises(ValueError):
            self.calc.divide(10, 0)

    def test_multiply(self):
        result = self.calc.multiply(3, 4)
        self.assertEqual(result, 12)

# Test with mocking
class APIClient:
    def __init__(self):
        self.base_url = "https://api.example.com"

    def get_user(self, user_id):
        import requests
        response = requests.get(f"{self.base_url}/users/{user_id}")
        return response.json()

class TestAPIClient(unittest.TestCase):
    def setUp(self):
        self.client = APIClient()

    @patch('requests.get')
    def test_get_user(self, mock_get):
        # Mock the response
        mock_response = MagicMock()
        mock_response.json.return_value = {'id': 1, 'name': 'John'}
        mock_get.return_value = mock_response

        # Test the method
        result = self.client.get_user(1)
```

```
        # Assertions
        self.assertEqual(result['id'], 1)
        self.assertEqual(result['name'], 'John')
        mock_get.assert_called_once_with('https://api.example.com/users/1')

if __name__ == '__main__':
    unittest.main()
```