Python Programming Language

Mathematical Notes

October 27, 2025

Contents

1 Introduction to Python

1.1 What is Python?

Definition 1.1 (Python Programming Language)**.** *Python is a high-level, interpreted, general-purpose programming language with dynamic typing and automatic memory management. It emphasizes code readability and simplicity.*

Python was created by Guido van Rossum and first released in 1991. The language design philosophy emphasizes:

- **Readability** - Code should be easy to read and understand
- **Simplicity** - Simple is better than complex
- **Explicit** - Explicit is better than implicit
- **Beautiful** - Beautiful is better than ugly

1.2 Python's Design Principles

The Zen of Python (PEP 20) outlines the guiding principles:

1. Beautiful is better than ugly
2. Explicit is better than implicit
3. Simple is better than complex
4. Complex is better than complicated
5. Flat is better than nested
6. Sparse is better than dense
7. Readability counts
8. Special cases aren't special enough to break the rules
9. Although practicality beats purity
10. Errors should never pass silently
11. Unless explicitly silenced
12. In the face of ambiguity, refuse the temptation to guess
13. There should be one obvious way to do it
14. Although that way may not be obvious at first
15. Now is better than never
16. Although never is often better than right now
17. If the implementation is hard to explain, it's a bad idea
18. If the implementation is easy to explain, it may be a good idea
19. Namespaces are one honking great idea

2 Python Syntax and Basic Concepts

2.1 Indentation and Code Blocks

Definition 2.1 (Indentation). *Python uses indentation to define code blocks instead of braces or keywords. The standard is 4 spaces per indentation level.*

Example 2.1 (Indentation Example).

```python
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

for i in range(10):
    print(f"F({i}) = {fibonacci(i)}")
```

2.2 Variables and Data Types

2.2.1 Dynamic Typing

Definition 2.2 (Dynamic Typing). *Python uses dynamic typing, meaning variable types are determined at runtime rather than compile time.*

Example 2.2 (Variable Assignment).

```python
# Variables can change type
x = 42          # Integer
x = "Hello"     # String
x = [1, 2, 3]   # List
x = 3.14        # Float
```

2.2.2 Basic Data Types

1. **Integers** - Whole numbers (unlimited precision)
2. **Floats** - Decimal numbers
3. **Strings** - Text data
4. **Booleans** - True/False values
5. **None** - Represents absence of value

Example 2.3 (Data Type Examples).

```python
# Integer
age = 25

# Float
pi = 3.14159

# String
name = "Alice"

# Boolean
```

```
is_student = True

# None
result = None
```

2.3 Operators

2.3.1 Arithmetic Operators

- `+` - Addition
- `-` - Subtraction
- `*` - Multiplication
- `/` - Division (returns float)
- `//` - Floor division
- `%` - Modulo
- `**` - Exponentiation

2.3.2 Comparison Operators

- `==` - Equal to
- `!=` - Not equal to
- `<` - Less than
- `>` - Greater than
- `<=` - Less than or equal to
- `>=` - Greater than or equal to

2.3.3 Logical Operators

- `and` - Logical AND
- `or` - Logical OR
- `not` - Logical NOT

3 Data Structures

3.1 Lists

Definition 3.1 (List)**.** *A list is an ordered, mutable collection of items. Lists can contain items of different types.*

Example 3.1 (List Operations).

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

# Accessing elements
first = numbers[0]       # 1
last = numbers[-1]       # 5

# Slicing
subset = numbers[1:4]     # [2, 3, 4]

# List methods
numbers.append(6)         # Add to end
numbers.insert(0, 0)     # Insert at index
numbers.remove(3)        # Remove first occurrence
numbers.pop()            # Remove and return last element
numbers.sort()           # Sort in place
```

3.2 Tuples

Definition 3.2 (Tuple). *A tuple is an ordered, immutable collection of items. Once created, tuples cannot be modified.*

Example 3.2 (Tuple Usage).

```python
# Creating tuples
coordinates = (10, 20)
point = (x, y, z)

# Unpacking
x, y = coordinates

# Single element tuple
single = (42,)  # Note the comma

# Tuple methods
count = coordinates.count(10)
index = coordinates.index(20)
```

3.3 Dictionaries

Definition 3.3 (Dictionary). *A dictionary is an unordered collection of key-value pairs. Keys must be immutable types.*

Example 3.3 (Dictionary Operations).

```python
# Creating dictionaries
person = {
    "name": "Alice",
    "age": 30,
    "city": "New York"
}

# Accessing values
```

```python
name = person["name"]
age = person.get("age", 0)  # With default value

# Modifying dictionaries
person["email"] = "alice@example.com"
person.update({"phone": "123-456-7890"})

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

3.4 Sets

Definition 3.4 (Set)**.** *A set is an unordered collection of unique elements. Sets support mathematical set operations.*

Example 3.4 (Set Operations)**.**

```python
# Creating sets
numbers = {1, 2, 3, 4, 5}
evens = {2, 4, 6, 8}

# Set operations
union = numbers | evens            # Union
intersection = numbers & evens      # Intersection
difference = numbers - evens        # Difference
symmetric_diff = numbers ^ evens    # Symmetric difference

# Set methods
numbers.add(6)
numbers.remove(1)
numbers.discard(7)  # Safe remove
```

4 Control Flow

4.1 Conditional Statements

Example 4.1 (If-Elif-Else)**.**

```python
score = 85

if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
elif score >= 70:
    grade = "C"
else:
    grade = "F"

print(f"Grade: {grade}")
```

4.2 Loops

4.2.1 For Loops

Example 4.2 (For Loop Examples).

```python
# Iterating over a list
fruits = ["apple", "banana", "orange"]
for fruit in fruits:
    print(fruit)

# Using range
for i in range(5):
    print(i)

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

# Dictionary iteration
person = {"name": "Alice", "age": 30}
for key, value in person.items():
    print(f"{key}: {value}")
```

4.2.2 While Loops

Example 4.3 (While Loop).

```python
count = 0
while count < 5:
    print(f"Count: {count}")
    count += 1

# Break and continue
while True:
    user_input = input("Enter 'quit' to exit: ")
    if user_input == "quit":
        break
    elif user_input == "skip":
        continue
    print(f"You entered: {user_input}")
```

5 Functions

5.1 Function Definition

Definition 5.1 (Function). *A function is a reusable block of code that performs a specific task. Functions can take parameters and return values.*

Example 5.1 (Basic Function).

```python
def greet(name):
    """Return a greeting message."""
    return f"Hello, {name}!"

# Function call
```

```python
message = greet("Alice")
print(message)
```

5.2 Function Parameters

5.2.1 Default Parameters

Example 5.2 (Default Parameters).

```python
def power(base, exponent=2):
    """Calculate base raised to the power of exponent."""
    return base ** exponent

# Using default parameter
result1 = power(5)       # 25
result2 = power(5, 3)    # 125
```

5.2.2 Keyword Arguments

Example 5.3 (Keyword Arguments).

```python
def create_person(name, age, city="Unknown"):
    return {"name": name, "age": age, "city": city}

# Using keyword arguments
person1 = create_person("Alice", 30, "New York")
person2 = create_person(age=25, name="Bob", city="Boston")
```

5.2.3 Variable Arguments

Example 5.4 (Variable Arguments).

```python
def sum_all(*args):
    """Sum all arguments."""
    return sum(args)

def print_info(**kwargs):
    """Print keyword arguments."""
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Usage
total = sum_all(1, 2, 3, 4, 5)
print_info(name="Alice", age=30, city="New York")
```

5.3 Lambda Functions

Definition 5.2 (Lambda Function). *A lambda function is an anonymous function defined using the lambda keyword. It can have any number of arguments but only one expression.*

Example 5.5 (Lambda Functions).

```python
# Basic lambda
square = lambda x: x ** 2

# Using with built-in functions
```

```python
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
evens = list(filter(lambda x: x % 2 == 0, numbers))

# Sorting with lambda
students = [("Alice", 85), ("Bob", 92), ("Charlie", 78)]
students.sort(key=lambda student: student[1])  # Sort by grade
```

6 Object-Oriented Programming

6.1 Classes and Objects

Definition 6.1 (Class)**.** *A class is a blueprint for creating objects. It defines attributes and methods that the objects will have.*

Example 6.1 (Basic Class)**.**

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

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

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

# Creating objects
person1 = Person("Alice", 30)
person2 = Person("Bob", 25)

# Using methods
print(person1.greet())
person1.have_birthday()
print(person1.age)  # 31
```

6.2 Inheritance

Definition 6.2 (Inheritance)**.** *Inheritance allows a class to inherit attributes and methods from another class.*

Example 6.2 (Inheritance)**.**

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

    def speak(self):
        return "Some generic animal sound"

class Dog(Animal):
    def speak(self):
        return "Woof!"
```

```python

class Cat(Animal):
    def speak(self):
        return "Meow!"

# Using inheritance
dog = Dog("Buddy")
cat = Cat("Whiskers")

print(dog.speak())  # Woof!
print(cat.speak())  # Meow!
```

6.3 Encapsulation

Example 6.3 (Encapsulation with Properties).

```python
class BankAccount:
    def __init__(self, initial_balance=0):
        self._balance = initial_balance

    @property
    def balance(self):
        return self._balance

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            return True
        return False

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            return True
        return False

# Using encapsulation
account = BankAccount(100)
print(account.balance)  # 100
account.deposit(50)
account.withdraw(25)
print(account.balance)  # 125
```

7 Error Handling

7.1 Exceptions

Definition 7.1 (Exception). *An exception is an event that occurs during program execution that disrupts the normal flow of instructions.*

Example 7.1 (Basic Exception Handling).

```python
try:
    number = int(input("Enter a number: "))
```

```python
    result = 10 / number
    print(f"Result: {result}")
except ValueError:
    print("Invalid input. Please enter a number.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
except Exception as e:
    print(f"An error occurred: {e}")
finally:
    print("This always executes.")
```

7.2 Custom Exceptions

Example 7.2 (Custom Exception)**.**

```python
class CustomError(Exception):
    def __init__(self, message):
        self.message = message
        super().__init__(self.message)

def validate_age(age):
    if age < 0:
        raise CustomError("Age cannot be negative")
    if age > 150:
        raise CustomError("Age seems unrealistic")
    return True

try:
    validate_age(-5)
except CustomError as e:
    print(f"Validation error: {e.message}")
```

8 Modules and Packages

8.1 Importing Modules

Example 8.1 (Module Import)**.**

```python
# Import entire module
import math
print(math.pi)

# Import specific functions
from math import sqrt, sin, cos
print(sqrt(16))

# Import with alias
import numpy as np
array = np.array([1, 2, 3])

# Import all (not recommended)
from math import *
print(pi)
```

8.2 Creating Modules

Example 8.2 (Custom Module - calculator.py).

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

def subtract(a, b):
    return a - b

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

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

# Using the module
import calculator
result = calculator.add(5, 3)
```

8.3 Packages

Definition 8.1 (Package). *A package is a collection of modules organized in a directory structure with an __init__.py file.*

Example 8.3 (Package Structure).

```python
# Package structure:
# mypackage/
#     __init__.py
#     module1.py
#     module2.py
#     subpackage/
#         __init__.py
#         module3.py

# __init__.py content:
from .module1 import function1
from .module2 import function2

# Using the package
from mypackage import function1, function2
from mypackage.subpackage.module3 import function3
```

9 File Handling

9.1 Reading Files

Example 9.1 (File Reading).

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

```

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

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

9.2 Writing Files

Example 9.2 (File Writing).

```
# Writing text
with open("output.txt", "w") as file:
    file.write("Hello, World!")

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

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

10 List Comprehensions and Generators

10.1 List Comprehensions

Definition 10.1 (List Comprehension). *A list comprehension is a concise way to create lists based on existing lists or other iterables.*

Example 10.1 (List Comprehensions).

```
# Basic list comprehension
squares = [x**2 for x in range(10)]

# With condition
evens = [x for x in range(20) if x % 2 == 0]

# Nested list comprehension
matrix = [[i*j for j in range(3)] for i in range(3)]

# Dictionary comprehension
squares_dict = {x: x**2 for x in range(5)}

# Set comprehension
unique_lengths = {len(word) for word in ["hello", "world", "python"]}
```

10.2 Generators

Definition 10.2 (Generator). *A generator is a function that returns an iterator. It uses the yield keyword instead of return.*

Example 10.2 (Generators).

```python
def fibonacci_generator(n):
    a, b = 0, 1
    for _ in range(n):
        yield a
        a, b = b, a + b

# Using generator
for num in fibonacci_generator(10):
    print(num)

# Generator expression
squares_gen = (x**2 for x in range(10))
print(list(squares_gen))
```

11 Decorators

Definition 11.1 (Decorator). *A decorator is a function that takes another function as input and returns a modified version of that function.*

Example 11.1 (Basic Decorator).

```python
def timer(func):
    import time
    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

@timer
def slow_function():
    time.sleep(1)
    return "Done"

# Using decorator
result = slow_function()
```

11.1 Built-in Decorators

Example 11.2 (Built-in Decorators).

```python
class Circle:
    def __init__(self, radius):
        self._radius = radius

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

    @radius.setter
    def radius(self, value):
```

```python
        if value < 0:
            raise ValueError("Radius cannot be negative")
        self._radius = value

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

# Using properties
circle = Circle(5)
print(circle.area)  # 78.54
circle.radius = 10
print(circle.area)  # 314.16
```

12 Advanced Topics

12.1 Context Managers

Definition 12.1 (Context Manager)**.** *A context manager is an object that defines the methods __enter__ and __exit__ for use with the with statement.*

Example 12.1 (Custom Context Manager)**.**

```python
class DatabaseConnection:
    def __init__(self, database):
        self.database = database

    def __enter__(self):
        print(f"Connecting to {self.database}")
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        print(f"Closing connection to {self.database}")
        if exc_type:
            print(f"Exception occurred: {exc_val}")

# Using context manager
with DatabaseConnection("mydb") as db:
    print("Performing database operations")
    # raise Exception("Something went wrong")
```

12.2 Metaclasses

Definition 12.2 (Metaclass)**.** *A metaclass is a class whose instances are classes. It defines how classes are created.*

Example 12.2 (Simple Metaclass)**.**

```python
class SingletonMeta(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super().__call__(*args, **kwargs)
```

```python
        return cls._instances[cls]

class Singleton(metaclass=SingletonMeta):
    def __init__(self):
        self.value = None

# Testing singleton
s1 = Singleton()
s2 = Singleton()
print(s1 is s2)  # True
```

12.3 Concurrency

12.3.1 Threading

Example 12.3 (Threading).

```python
import threading
import time

def worker(name, delay):
    for i in range(5):
        print(f"Worker {name}: {i}")
        time.sleep(delay)

# Creating threads
thread1 = threading.Thread(target=worker, args=("A", 1))
thread2 = threading.Thread(target=worker, args=("B", 0.5))

# Starting threads
thread1.start()
thread2.start()

# Waiting for threads to complete
thread1.join()
thread2.join()
```

12.3.2 Asyncio

Example 12.4 (Asyncio).

```python
import asyncio

async def async_worker(name, delay):
    for i in range(5):
        print(f"Async Worker {name}: {i}")
        await asyncio.sleep(delay)

async def main():
    # Running coroutines concurrently
    await asyncio.gather(
        async_worker("A", 1),
        async_worker("B", 0.5)
    )

```

```python
# Running the async main function
asyncio.run(main())
```

13 Python Standard Library

13.1 Common Modules

1. **os** - Operating system interface
2. **sys** - System-specific parameters
3. **json** - JSON data handling
4. **csv** - CSV file handling
5. **datetime** - Date and time handling
6. **collections** - Specialized container types
7. **itertools** - Iterator functions
8. **functools** - Higher-order functions
9. **re** - Regular expressions
10. **urllib** - URL handling

Example 13.1 (Using Standard Library)**.**

```python
import os
import json
import datetime
from collections import Counter

# File system operations
files = os.listdir('.')
current_dir = os.getcwd()

# JSON handling
data = {"name": "Alice", "age": 30}
json_string = json.dumps(data)
parsed_data = json.loads(json_string)

# Date and time
now = datetime.datetime.now()
formatted_date = now.strftime("%Y-%m-%d %H:%M:%S")

# Collections
words = ["apple", "banana", "apple", "cherry", "banana", "apple"]
word_count = Counter(words)
print(word_count)  # Counter({'apple': 3, 'banana': 2, 'cherry': 1})
```

14 Best Practices and Style

14.1 PEP 8 Style Guide

Definition 14.1 (PEP 8)**.** *PEP 8 is the official style guide for Python code, covering naming conventions, code layout, and formatting.*

Key PEP 8 guidelines:

- Use 4 spaces for indentation
- Limit lines to 79 characters
- Use descriptive variable names
- Use snake_case for functions and variables
- Use PascalCase for classes
- Use UPPER_CASE for constants

14.2 Documentation

Example 14.1 (Docstrings)**.**

```python
def calculate_fibonacci(n):
    """
    Calculate the nth Fibonacci number.

    Args:
        n (int): The position in the Fibonacci sequence

    Returns:
        int: The nth Fibonacci number

    Raises:
        ValueError: If n is negative

    Example:
        >>> calculate_fibonacci(10)
        55
    """
    if n < 0:
        raise ValueError("n must be non-negative")

    if n <= 1:
        return n

    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b

    return b
```

14.3 Testing

Example 14.2 (Unit Testing).

```python
import unittest

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

class TestMath(unittest.TestCase):
    def test_add_positive_numbers(self):
        self.assertEqual(add(2, 3), 5)

    def test_add_negative_numbers(self):
        self.assertEqual(add(-2, -3), -5)

    def test_add_mixed_numbers(self):
        self.assertEqual(add(2, -3), -1)

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

15 Python Ecosystem

15.1 Popular Libraries

1. **NumPy** - Numerical computing
2. **Pandas** - Data manipulation and analysis
3. **Matplotlib** - Plotting and visualization
4. **Scikit-learn** - Machine learning
5. **Django/Flask** - Web frameworks
6. **Requests** - HTTP library
7. **Pillow** - Image processing
8. **SymPy** - Symbolic mathematics

15.2 Virtual Environments

Definition 15.1 (Virtual Environment). *A virtual environment is an isolated Python environment that allows you to manage dependencies for different projects.*

Example 15.1 (Virtual Environment Usage).

```
# Creating virtual environment
python -m venv myenv

# Activating (Windows)
myenv\Scripts\activate

# Activating (Unix/MacOS)
```

```
source myenv/bin/activate

# Installing packages
pip install numpy pandas matplotlib

# Freezing requirements
pip freeze > requirements.txt

# Installing from requirements
pip install -r requirements.txt
```

16 Performance and Optimization

16.1 Profiling

Example 16.1 (Basic Profiling).

```python
import cProfile
import pstats

def slow_function():
    total = 0
    for i in range(1000000):
        total += i ** 2
    return total

# Profiling
profiler = cProfile.Profile()
profiler.enable()
result = slow_function()
profiler.disable()

# Analyzing results
stats = pstats.Stats(profiler)
stats.sort_stats('cumulative')
stats.print_stats(10)
```

16.2 Optimization Techniques

1. Use list comprehensions instead of loops
2. Use generators for large datasets
3. Use appropriate data structures
4. Avoid global variables
5. Use local variables in loops
6. Profile before optimizing

17 Conclusion

Python is a versatile, powerful programming language that emphasizes readability and simplicity. Its key strengths include:

- **Readability** - Clean, intuitive syntax
- **Versatility** - Suitable for many domains
- **Large Ecosystem** - Extensive standard library and third-party packages
- **Community** - Active, supportive community
- **Cross-platform** - Runs on multiple operating systems

Python's design philosophy of "batteries included" and its focus on developer productivity make it an excellent choice for beginners and experienced programmers alike. The language continues to evolve with regular updates and improvements, ensuring its relevance in modern software development.