Complete Python Fundamentals: Theory and Concepts

Table of Contents

1. Python Language Fundamentals
2. Data Types and Type System
3. Memory Management and Object Model
4. Control Flow and Program Structure
5. Functions and Functional Programming
6. Object-Oriented Programming
7. Modules, Packages, and Namespaces
8. Exception Handling and Error Management
9. Iterators, Generators, and Iterables
10. Decorators and Metaprogramming
11. Context Managers and Resource Management
12. Concurrency and Parallelism
13. Python's Execution Model
14. Advanced Concepts

---

1. Python Language Fundamentals

1.1 What is Python?

Definition: Python is a high-level, interpreted, dynamically-typed, and garbage-collected programming language that emphasizes code readability and simplicity.

Key Characteristics:

- **Interpreted**: Code is executed line by line by the Python interpreter
- **Dynamically Typed**: Variable types are determined at runtime
- **Strongly Typed**: No implicit type conversions that might lose data
- **Multi-paradigm**: Supports procedural, object-oriented, and functional programming
- **Garbage Collected**: Automatic memory management

1.2 Python Philosophy (The Zen of Python)

```python
import this  # Try this to see the Zen of Python
```

Core Principles:

1. **Beautiful is better than ugly**: Code should be aesthetically pleasing
2. **Explicit is better than implicit**: Clear, obvious code is preferred
3. **Simple is better than complex**: Choose simple solutions when possible
4. **Complex is better than complicated**: If complexity is needed, avoid complication
5. **Flat is better than nested**: Avoid deep nesting
6. **Sparse is better than dense**: Don't pack too much into one line
7. **Readability counts**: Code is read more often than written
8. **Special cases aren't special enough**: Consistency is important
9. **Errors should never pass silently**: Handle exceptions explicitly
10. **There should be one obvious way to do it**: Avoid multiple ways for same task

1.3 Python Syntax Fundamentals

Indentation: Python uses indentation to define code blocks

- Standard: 4 spaces per indentation level
- Mixing tabs and spaces is not allowed
- Indentation is syntactically significant

Comments:

- Single-line: `# This is a comment`
- Multi-line: Using triple quotes `"""This is a multi-line comment"""`
- Docstrings: First statement in modules, functions, classes

Naming Conventions (PEP 8):

- Variables and functions: `snake_case`
- Constants: `UPPER_CASE`
- Classes: `PascalCase`
- Protected members: `_single_leading_underscore`
- Private members: `__double_leading_underscore`
- Magic methods: `__double_underscore__`

1.4 Keywords and Built-in Functions

Reserved Keywords: Words that have special meaning in Python

- Control flow: `if`, `elif`, `else`, `for`, `while`, `break`, `continue`, `pass`
- Functions: `def`, `return`, `yield`, `lambda`
- Classes: `class`, `self`, `super`
- Exceptions: `try`, `except`, `finally`, `raise`, `assert`
- Logic: `and`, `or`, `not`, `is`, `in`
- Others: `import`, `from`, `as`, `global`, `nonlocal`, `del`, `with`

Built-in Functions: Functions available without imports

- Type conversion: `int()`, `float()`, `str()`, `list()`, `tuple()`, `dict()`, `set()`
- I/O: `print()`, `input()`, `open()`
- Iteration: `range()`, `enumerate()`, `zip()`, `map()`, `filter()`
- Object inspection: `type()`, `isinstance()`, `hasattr()`, `getattr()`
- Others: `len()`, `sum()`, `min()`, `max()`, `sorted()`, `reversed()`

---

2. Data Types and Type System

2.1 Type System Overview

Dynamic Typing: Types are associated with values, not variables

```python
x = 5       # x refers to an integer object
x = "hello" # Now x refers to a string object
```

Strong Typing: No implicit type coercion that loses information

```python
# This will raise TypeError
result = "5" + 5  # Can't add string and integer

# Explicit conversion required
result = int("5") + 5  # Works: 10
```

2.2 Primitive Data Types

2.2.1 Numeric Types

int (Integer):

- Arbitrary precision (no overflow)

- Immutable
- Operations: `+`, `-`, `*`, `/`, `//`, `%`, `**`
- Bitwise: `&`, `|`, `^`, `~`, `<<`, `>>`

float (Floating Point):

- IEEE 754 double-precision
- Limited precision (~15-17 decimal digits)
- Special values: `float('inf')`, `float('-inf')`, `float('nan')`

complex:

- Format: `a + bj` where a and b are floats
- Access parts: `z.real`, `z.imag`

bool (Boolean):

- Subclass of int: `True == 1`, `False == 0`
- Result of comparison operations

2.2.2 Sequence Types

str (String):

- Immutable sequence of Unicode characters
- Indexed and sliceable
- Methods for manipulation and searching
- Raw strings: `r"raw\string"`
- F-strings: `f"value = {x}"`

list:

- Mutable ordered sequence
- Can contain mixed types
- Dynamic sizing
- O(1) append and pop from end
- O(n) insert and delete

tuple:

- Immutable ordered sequence
- Hashable (can be dictionary keys)

- Memory efficient
- Used for fixed collections

range:

- Immutable sequence of numbers
- Memory efficient (lazy evaluation)
- Commonly used in loops

2.2.3 Mapping Type

dict (Dictionary):

- Mutable mapping of key-value pairs
- Keys must be hashable
- O(1) average case lookup
- Ordered (Python 3.7+)
- Implementation: Hash table

2.2.4 Set Types

set:

- Mutable unordered collection of unique elements
- Elements must be hashable
- O(1) average case membership testing
- Set operations: union, intersection, difference

frozenset:

- Immutable version of set
- Hashable (can be in sets or dict keys)

2.2.5 Binary Types

bytes:

- Immutable sequence of bytes (0-255)
- Used for binary data

bytearray:

- Mutable version of bytes

memoryview:

- Memory-efficient slicing of binary data

2.3 Type Hierarchy

```
object
├── type
├── int
│   └── bool
├── float
├── complex
├── str
├── list
├── tuple
├── range
├── dict
├── set
├── frozenset
├── bytes
├── bytearray
└── ...
```

2.4 Type Checking and Conversion

Type Checking:

```python
type(x)            # Get exact type
isinstance(x, T)   # Check if x is instance of T or subclass
issubclass(A, B)   # Check if A is subclass of B
```

Type Conversion:

- Explicit: Using constructor functions
- No implicit conversion that loses data
- `__int__()`, `__float__()`, `__str__()` methods for custom conversion

---

3. Memory Management and Object Model

3.1 Everything is an Object

Object Model:

- Every value is an object
- Objects have: identity, type, and value

- Identity: Unique identifier (memory address)
- Type: Defines operations and attributes
- Value: The data stored

Object Properties:

```python
id(obj)    # Identity (memory address)
type(obj)  # Type of object
vars(obj)  # Namespace as dictionary
dir(obj)   # All attributes and methods
```

3.2 Reference Counting

How It Works:

- Each object maintains a count of references
- When count reaches 0, object is deallocated
- Immediate deallocation for most objects

Reference Count Management:

```python
import sys
sys.getrefcount(obj)  # Get reference count (returns count + 1)
```

3.3 Garbage Collection

Cyclic References:

- Reference counting can't handle cycles
- Generational garbage collector handles cycles
- Three generations: 0 (young), 1 (middle), 2 (old)

GC Control:

```python
import gc
gc.collect()      # Force garbage collection
gc.disable()      # Disable automatic GC
gc.enable()       # Enable automatic GC
gc.get_count()    # Get collection counts
```

3.4 Object Mutability

Immutable Objects:

- Cannot be changed after creation
- Types: int, float, str, tuple, frozenset, bytes
- Operations create new objects
- Can be used as dictionary keys

Mutable Objects:

- Can be modified in place
- Types: list, dict, set, bytearray
- Cannot be dictionary keys
- Changes affect all references

Identity vs Equality:

python

```python
a is b    # Identity: same object in memory
a == b    # Equality: same value
```

3.5 Object Interning

Small Integer Caching:

- Integers from -5 to 256 are cached
- Same object reused for efficiency

String Interning:

- Some strings are automatically interned
- Short strings, identifiers
- Can manually intern: `sys.intern()`

---

4. Control Flow and Program Structure

4.1 Conditional Statements

if-elif-else:

- Sequential checking of conditions
- First true condition executes

- Short-circuit evaluation

Conditional Expression (Ternary):

```python
value = x if condition else y
```

Pattern Matching (Python 3.10+):

```python
match value:
    case pattern1:
        # action1
    case pattern2:
        # action2
    case _:
        # default
```

4.2 Loops

for Loop:

- Iterates over iterables
- Uses iterator protocol
- Can use `else` clause (executes if no break)

while Loop:

- Continues while condition is true
- Also supports `else` clause

Loop Control:

- `break`: Exit loop immediately
- `continue`: Skip to next iteration
- `pass`: No-op placeholder

4.3 Comprehensions

List Comprehension:

```python
[expr for item in iterable if condition]
```

Dictionary Comprehension:

```python
{key_expr: value_expr for item in iterable if condition}
```

Set Comprehension:

```python
{expr for item in iterable if condition}
```

Generator Expression:

```python
(expr for item in iterable if condition)
```

Theory: Comprehensions are syntactic sugar for loops with optional filtering, providing:

- More readable code
- Often better performance
- Functional programming style

---

5. Functions and Functional Programming

5.1 Function Definition and Calling

Function Object:

- Functions are first-class objects
- Can be assigned, passed, and returned
- Have attributes and methods

Parameter Types:

1. **Positional parameters**: Required, order matters
2. **Default parameters**: Optional with default values
3. **Variable positional (*args)**: Tuple of extra positional arguments
4. **Keyword-only parameters**: Must be passed by name
5. **Variable keyword (**kwargs)**: Dictionary of extra keyword arguments

5.2 Scope and Namespace

LEGB Rule (Scope Resolution Order):

1. **Local**: Inside current function
2. **Enclosing**: In enclosing function
3. **Global**: At module level
4. **Built-in**: Pre-defined names

Namespace:

- Mapping from names to objects
- Implemented as dictionaries
- Created at different times (built-in, module, function call)

5.3 Closures

Definition: Function that retains access to variables from enclosing scope

Requirements:

1. Nested function
2. References variable from enclosing scope
3. Enclosing function returns nested function

Use Cases:

- Data hiding
- Function factories
- Decorators

5.4 Lambda Functions

Anonymous Functions:

- Single expression only
- No statements allowed
- Automatically returns expression result
- Same scope rules as regular functions

5.5 Functional Programming Concepts

Pure Functions:

- No side effects
- Same input always produces same output

- Don't modify external state

Higher-Order Functions:

- Functions that take functions as arguments
- Functions that return functions
- Examples: `map()`, `filter()`, `reduce()`

Immutability:

- Prefer creating new objects over modifying
- Leads to more predictable code
- Easier to reason about and debug

---

6. Object-Oriented Programming

6.1 Classes and Objects

Class Definition:

- Blueprint for creating objects
- Defines attributes and methods
- Creates new type

Object Creation:

- Instantiation calls `__new__()` then `__init__()`
- `__new__()`: Creates instance
- `__init__()`: Initializes instance

6.2 Attributes and Methods

Instance Attributes:

- Unique to each instance
- Stored in instance's `__dict__`
- Defined in `__init__()` typically

Class Attributes:

- Shared among all instances
- Stored in class's `__dict__`
- Can be overridden per instance

Methods:

- **Instance methods**: First parameter is `self`
- **Class methods**: First parameter is `cls`, use `@classmethod`
- **Static methods**: No special first parameter, use `@staticmethod`

6.3 Inheritance

Single Inheritance:

- Class derives from one parent
- Inherits attributes and methods
- Can override parent methods

Multiple Inheritance:

- Class derives from multiple parents
- Method Resolution Order (MRO) determines lookup
- C3 linearization algorithm

super():

- Delegates method calls to parent/sibling classes
- Follows MRO
- Cooperative inheritance

6.4 Encapsulation

Access Modifiers (Conventions):

- **Public**: No underscore prefix
- **Protected**: Single underscore `_name`
- **Private**: Double underscore `__name` (name mangling)

Property Decorator:

- Computed attributes
- Getters, setters, deleters
- Maintains interface while changing implementation

6.5 Polymorphism

Duck Typing:

- "If it walks like a duck and quacks like a duck..."

- Focus on object capabilities, not type
- Enables flexible interfaces

Method Overriding:

- Child class provides different implementation
- Same method signature
- Dynamic dispatch at runtime

Operator Overloading:

- Define behavior for built-in operators
- Magic methods like `__add__()`, `__str__()`

6.6 Abstract Base Classes

ABC Module:

- Define interfaces
- Cannot instantiate abstract classes
- Enforce method implementation in subclasses

Protocol Classes (Python 3.8+):

- Structural subtyping
- Define expected interface
- No explicit inheritance needed

---

7. Modules, Packages, and Namespaces

7.1 Modules

Definition: File containing Python code

- Can define functions, classes, variables
- Provides namespace
- Promotes code reusability

Import Mechanism:

1. Check `sys.modules` cache
2. Find module (search `sys.path`)
3. Create module object

4. Execute module code
5. Add to `sys.modules`

Import Statements:

```python
import module
from module import name
from module import *  # Avoid
import module as alias
```

7.2 Packages

Definition: Directory containing modules

- Must contain `__init__.py` (can be empty)
- Can have subpackages
- Provides hierarchical namespace

Package Initialization:

- `__init__.py` executes on import
- Defines package's public interface
- Can control what's imported with `*`

7.3 Namespace Packages

PEP 420: Packages without `__init__.py`

- Span multiple directories
- Used for plugins/extensions
- Automatic namespace package detection

7.4 Module Search Path

sys.path Order:

1. Current directory
2. PYTHONPATH directories
3. Standard library directories
4. Site-packages directory

Path Manipulation:

- Modify `sys.path` at runtime
- Use `.pth` files
- Virtual environments modify path

---

8. Exception Handling and Error Management

8.1 Exception Hierarchy

```
BaseException
├── SystemExit
├── KeyboardInterrupt
├── GeneratorExit
└── Exception
    ├── StopIteration
    ├── ArithmeticError
    │   ├── ZeroDivisionError
    │   ├── OverflowError
    │   └── FloatingPointError
    ├── LookupError
    │   ├── IndexError
    │   └── KeyError
    ├── TypeError
    ├── ValueError
    └── ...
```

8.2 Exception Handling

try-except-else-finally:

- `try`: Code that might raise exception
- `except`: Handle specific exceptions
- `else`: Executes if no exception
- `finally`: Always executes (cleanup)

Exception Matching:

- Matches by inheritance
- First matching except wins
- Can catch multiple exceptions

8.3 Raising Exceptions

raise Statement:

- Raise new exception
- Re-raise current exception
- Raise with different type

Exception Chaining:

- `raise new from original`
- Preserves traceback
- Shows cause relationship

8.4 Custom Exceptions

Best Practices:

- Inherit from Exception
- Provide useful error messages
- Add custom attributes if needed
- Document when raised

8.5 Assertions

assert Statement:

- Development/debugging tool
- Disabled with -O flag
- Not for user input validation
- Documents assumptions

---

9. Iterators, Generators, and Iterables

9.1 Iteration Protocol

Iterable:

- Object with `__iter__()` method
- Returns iterator
- Can be used in for loops

Iterator:

- Object with `__iter__()` and `__next__()`
- Maintains iteration state
- Raises StopIteration when exhausted

9.2 Generators

Generator Functions:

- Use `yield` keyword
- Return generator object
- Lazy evaluation
- Memory efficient

Generator Expressions:

- Like list comprehensions but lazy
- Use parentheses instead of brackets
- Create generator objects

9.3 yield Statement

Behavior:

- Suspends function execution
- Saves local state
- Returns value
- Resumes on next()

yield from:

- Delegate to subgenerator
- Bidirectional communication
- Exception propagation

9.4 Coroutines

Generator-based Coroutines:

- Use `yield` to receive values
- `send()` method
- Deprecated in favor of async/await

---

10. Decorators and Metaprogramming

10.1 Decorator Basics

Definition: Function that modifies another function

- Takes function as argument
- Returns modified function
- Applied with @ syntax

Execution Time:

- Decorators execute at definition time
- Not at call time
- Can modify function object

10.2 Decorator Patterns

Simple Decorator:

```python
def decorator(func):
    def wrapper(*args, **kwargs):
        # Before function
        result = func(*args, **kwargs)
        # After function
        return result
    return wrapper
```

Parameterized Decorator:

```python
def decorator_factory(param):
    def decorator(func):
        def wrapper(*args, **kwargs):
            # Use param
            return func(*args, **kwargs)
        return wrapper
    return decorator
```

10.3 Class Decorators

Decorating Classes:

- Modify class after definition
- Add/modify attributes
- Implement patterns (singleton, etc.)

10.4 Descriptor Protocol

Descriptors: Objects that define attribute access

- `__get__()`: Attribute access
- `__set__()`: Attribute assignment
- `__delete__()`: Attribute deletion

Use Cases:

- Properties
- Methods
- Class methods
- Static methods

10.5 Metaclasses

Definition: Class whose instances are classes

- Control class creation
- Modify class namespace
- Implement class-level patterns

type: Default metaclass

- Can create classes dynamically
- `type(name, bases, namespace)`

---

11. Context Managers and Resource Management

11.1 Context Manager Protocol

Required Methods:

- `__enter__()`: Setup/acquire resource
- `__exit__()`: Cleanup/release resource

with Statement:

- Ensures cleanup even with exceptions
- Automatic resource management
- Can suppress exceptions

11.2 contextlib Module

contextmanager Decorator:

- Create context managers with generators
- Yield separates setup/cleanup

Other Utilities:

- `closing()`: Adds context management to objects with close()
- `suppress()`: Suppress specific exceptions
- `ExitStack`: Manage dynamic number of contexts

11.3 Use Cases

Resource Management:

- File handling
- Network connections
- Database transactions
- Thread locks

State Management:

- Temporary state changes
- Decimal precision
- Warning filters

---

12. Concurrency and Parallelism

12.1 GIL (Global Interpreter Lock)

What It Is:

- Mutex that protects Python objects
- Only one thread executes Python bytecode
- Simplifies memory management

Implications:

- CPU-bound tasks don't benefit from threads
- I/O-bound tasks can benefit
- True parallelism needs multiprocessing

12.2 Threading

Thread Objects:

- Concurrent execution within process
- Share memory space
- Good for I/O-bound tasks

Synchronization:

- Lock: Mutual exclusion
- RLock: Reentrant lock
- Semaphore: Limited resource access
- Event: Thread communication

12.3 Multiprocessing

Process Objects:

- Separate memory space
- True parallelism
- Inter-process communication needed

Communication:

- Queue: Thread-safe queue
- Pipe: Two-way communication
- Manager: Shared objects

12.4 Asyncio

Coroutines:

- Single-threaded concurrency
- async/await syntax
- Event loop driven

Benefits:

- High concurrency for I/O
- Lower overhead than threads
- Explicit concurrency points

---

13. Python's Execution Model

13.1 Compilation Process

Source to Bytecode:

1. Lexical analysis (tokenization)
2. Parsing (AST creation)
3. Compilation (bytecode generation)
4. Optimization
5. `.pyc` file creation

13.2 Python Virtual Machine

Stack-based VM:

- Executes bytecode instructions
- Uses evaluation stack
- Frame objects for function calls

Bytecode:

- Platform-independent
- Can inspect with `dis` module
- Cached in `__pycache__`

13.3 Name Resolution

Loading Names:

- LOAD_FAST: Local variables
- LOAD_GLOBAL: Global/builtin names
- LOAD_DEREF: Closure variables

Storing Names:

- STORE_FAST: Local variables
- STORE_GLOBAL: Global variables
- STORE_DEREF: Closure variables

13.4 Import System

Import Hooks:

- Finders: Locate modules
- Loaders: Load modules
- Can customize import behavior

Module Caching:

- `sys.modules` dictionary
- Prevents reimporting
- Can be cleared manually

---

14. Advanced Concepts

14.1 Slots

slots:

- Restricts instance attributes
- Saves memory
- Faster attribute access
- No `__dict__` per instance

14.2 Weak References

weakref Module:

- References that don't prevent GC
- Useful for caches
- Callbacks on object deletion

14.3 Abstract Syntax Trees

ast Module:

- Parse Python code
- Analyze code structure
- Code transformation
- Static analysis

14.4 Type Hints

PEP 484:

- Optional static typing
- Better IDE support
- Documentation
- Type checkers (mypy)

Common Types:

```python
from typing import List, Dict, Optional, Union, Callable
```

14.5 Data Classes

@dataclass:

- Automatic `__init__()`, `__repr__()`, etc.
- Less boilerplate
- Type hints integration
- Frozen/mutable options

14.6 Async/Await

Asynchronous Programming:

- Coroutine functions: `async def`
- Await expressions: `await`
- Async context managers
- Async iterators

14.7 Memory Views

Buffer Protocol:

- Zero-copy slicing
- Efficient binary data handling
- Works with bytes, bytearray, array

14.8 Python C API

Extension Modules:

- Write Python modules in C
- Performance critical code
- Wrap existing C libraries

Summary: Key Theoretical Concepts to Master

1. Core Language Mechanics

- How Python executes code (compilation, bytecode, VM)
- Object model (everything is an object)

- Memory management (reference counting, GC)
- Name binding and scope resolution

2. **Type System**

- Dynamic vs static typing
- Strong typing (no implicit conversions)
- Duck typing and protocols
- Type hierarchy and inheritance

3. **Programming Paradigms**

- Procedural programming
- Object-oriented programming
- Functional programming
- Aspect-oriented (decorators)

4. **Concurrency Models**

- GIL and its implications
- Threading for I/O
- Multiprocessing for CPU
- Async/await for concurrency

5. **Advanced Features**

- Metaclasses and descriptors
- Context managers
- Generators and iterators
- Decorators and closures

Learning Path Recommendations

1. **Start with**: Basic syntax, data types, control flow
2. **Then learn**: Functions, modules, basic OOP
3. **Deep dive into**: Memory model, iterators, generators
4. **Advanced topics**: Decorators, metaclasses, async
5. **Specialize in**: Your area of interest (web, data science, etc.)

Remember: Understanding the "why" behind Python's design decisions will make you a better programmer than just knowing the "how".