Programming Techniques for Scientific Simulations I: A Comprehensive Introduction to Python

Based on lecture slides

December 4, 2025

Contents

1 Introduction to Python 2
1.1 Why Python for Scientific Computing? 2

2 Understanding Execution Models: Interpreted vs. Compiled Languages 3
2.1 Compiled Languages: The C++ Model 3
2.2 Interpreted Languages: The Python Model 4

3 Type Systems: Static vs. Dynamic Typing 6
3.1 Static Typing: The C++ Approach 6
3.2 Dynamic Typing: The Python Approach 7

4 Python Versions: Python 2 vs. Python 3 8
4.1 Python 2.x: The Legacy Version 8
4.2 Python 3.x: The Modern Standard 9

5 Installing and Setting Up Python 10
5.1 Installation on Linux 10
5.2 Installation on macOS 11
5.3 Installation on Windows 11
5.4 Python Distribution Recommendations 12

6 Python's Library Ecosystem 13
6.1 The Python Standard Library 13
6.2 The Python Package Index (PyPI) 13
6.3 Comparing Python and C++ Library Ecosystems 15

7 Essential Scientific Computing Packages 15
7.1 NumPy: Numerical Python 15
7.2 SciPy: Scientific Python 16
7.3 Matplotlib: Visualization 17
7.4 H5py: HDF5 Interface 18

7.5 mpi4py: Parallel Computing 18
7.6 SymPy: Symbolic Mathematics 18

8 Python Execution Environments 19
8.1 The Standard Python Interactive Shell 19
8.2 Running Python Scripts 20
8.3 IPython: Enhanced Interactive Shell 21
8.4 Jupyter Notebooks: Interactive Computing Environment 22
8.5 Note on Python vs. Python3 Command 25

9 Getting Help in Python 25
9.1 The help() Function 25
9.2 IPython's Enhanced Help 26
9.3 Online Documentation 26

10 Python Syntax Fundamentals 27
10.1 A First Complete Example 27
10.2 Comments 28
10.3 Line Continuation 29

11 Basic Built-In Data Types 29
11.1 Boolean Type 29
11.2 Integer Type 30
11.3 Floating-Point Type 31
11.4 Complex Numbers 32
11.5 String Type 32

12 Operators and Expressions 34
12.1 Arithmetic Operators 34
12.2 Using the Math Module 35
12.3 Random Number Generation 35

13 Compound Data Types: Lists 36
13.1 Creating and Accessing Lists 36
13.2 Modifying Lists 37
13.3 List Concatenation and Repetition 37
13.4 List Slicing 38
13.5 Common List Operations 39

14 Tuples: Immutable Sequences 40
14.1 Creating and Using Tuples 40
14.2 Tuple Immutability 41
14.3 When to Use Tuples vs. Lists 41
14.4 Tuple Packing and Unpacking 42

15 Dictionaries: Key-Value Mappings **42**
15.1 Creating Dictionaries . . . 42
15.2 Accessing and Modifying Dictionaries . . . 43
15.3 Iterating Over Dictionaries . . . 44
15.4 Dictionary Constraints . . . 44
15.5 Dictionary Ordering . . . 45

16 Mutability: A Fundamental Concept **45**
16.1 Defining Mutability . . . 45
16.2 Demonstrating Mutability . . . 46
16.3 Why Mutability Matters . . . 46
16.4 The Identity vs Equality Distinction . . . 46

17 Control Flow: Conditional Statements **47**
17.1 The if Statement . . . 47
17.2 The if-else Statement . . . 48
17.3 The if-elif-else Statement . . . 48
17.4 Comparison Operators . . . 49
17.5 Logical Operators . . . 49
17.6 Chained Comparisons . . . 50
17.7 Indentation: Python's Block Structure . . . 50

18 Control Flow: Loops **51**
18.1 The while Loop . . . 51
18.2 Loop Control: break and continue . . . 52
18.3 The for Loop: Iteration Over Sequences . . . 53
18.4 The range() Function . . . 54
18.5 Iterating with Indices . . . 54
18.6 Nested Loops . . . 55
18.7 Loop else Clause . . . 55

19 List and Dictionary Comprehensions **56**
19.1 List Comprehensions . . . 56
19.2 Dictionary Comprehensions . . . 57
19.3 Set Comprehensions . . . 58
19.4 Generator Expressions . . . 58
19.5 When to Use Comprehensions . . . 59

20 Functions **59**
20.1 Defining Functions . . . 59
20.2 Function Characteristics . . . 60
20.3 Default Parameter Values . . . 62
20.4 Keyword Arguments . . . 63
20.5 Variable-Length Arguments . . . 63
20.6 Unpacking Arguments . . . 65
20.7 Lambda Functions . . . 65

21 Coding Style and Conventions **67**
21.1 PEP 8: The Python Style Guide 67
21.2 Enforcing Style with Tools 69
21.3 Editor Configuration 70
21.4 Interesting Style-Related Facts 70

22 How Function Arguments Work: Pass by Assignment **70**
22.1 The Mystery 70
22.2 Understanding Python's Object Model 71
22.3 Step-by-Step: The Integer Example 71
22.4 Step-by-Step: The List Example 72
22.5 The Critical Distinction 73
22.6 Assignment and Aliasing 76
22.7 Copying Objects 77
22.8 Practical Implications 78

23 Object-Oriented Programming: Classes **78**
23.1 Defining Classes 78
23.2 The self Parameter 79
23.3 The Constructor: __init__ 80
23.4 Instance Methods 81
23.5 Python vs. C++ Classes 82

24 Privacy in Python Classes **82**
24.1 Convention-Based Privacy 83
24.2 Python vs. C++ Privacy 83
24.3 Name Mangling 84
24.4 Why No True Privacy? 84

25 Class Variables and Static Data **85**
25.1 Defining Class Variables 85
25.2 Class Variables vs. Instance Variables 85
25.3 Comparing with C++ 86

26 Magic Methods: Operator Overloading **87**
26.1 Understanding Magic Methods 87
26.2 Common Magic Methods 88
26.3 String Representation Methods 91
26.4 Complete Magic Methods Reference 91

27 Class Documentation with Docstrings **91**
27.1 Class and Method Docstrings 91
27.2 Docstring Conventions 93

28 Inheritance **94**
28.1 Basic Inheritance 94
28.2 The super() Function 96
28.3 Multiple Inheritance 97

29 Decorators **98**
29.1 Understanding Decorators 98
29.2 Decorators with Arguments 98
29.3 Practical Decorator Examples 99

30 Built-In Decorators for Classes **100**
30.1 @staticmethod 100
30.2 @classmethod 100
30.3 @property 101

31 Modules and Code Organization **103**
31.1 What is a Module? 103
31.2 Importing Modules 104
31.3 Module Search Path 105
31.4 Packages 105
31.5 The __init__.py File 105

32 Executing Modules as Scripts **106**
32.1 The __name__ Variable 106
32.2 Command-Line Arguments 107

33 String Formatting **108**
33.1 F-Strings (Format String Literals) 108
33.2 Format String Method 109
33.3 Old-Style Formatting (printf-style) 110
33.4 Format Specification Mini-Language 110

34 File Input/Output **111**
34.1 Opening Files 111
34.2 Writing to Files 112
34.3 Reading from Files 112
34.4 Context Managers: The with Statement 112
34.5 Complete File I/O Examples 113
34.6 File Path Operations 114

35 Exception Handling **115**
35.1 Understanding Exceptions 115
35.2 Basic Exception Handling 115
35.3 Catching Multiple Exceptions 115
35.4 Complete Exception Handling Structure 116
35.5 Accessing Exception Information 117
35.6 Raising Exceptions 117

35.7 Common Built-In Exceptions . . . 117
35.8 Custom Exceptions . . . 118

36 Python Conventions and Best Practices **118**
36.1 Naming Conventions . . . 119
36.2 Important Naming Conventions . . . 119
36.3 The Zen of Python . . . 120
36.4 PEP 8 Checklist . . . 120
36.5 Docstring Conventions - PEP 257 . . . 121
36.6 Style Checking with pylint . . . 122
36.7 Complementary Style Guides . . . 122

37 Additional Resources and Further Learning **122**
37.1 Official Python Documentation . . . 123
37.2 Third-Party Packages . . . 123
37.3 Recommended Learning Path . . . 124
37.4 Community Resources . . . 124

38 Summary and Conclusion **125**
38.1 Key Concepts Covered . . . 125
38.2 Python's Philosophy . . . 126
38.3 Next Steps . . . 127
38.4 Final Thoughts . . . 127

1 Introduction to Python

Python is a high-level, interpreted programming language that has become one of the most popular choices for scientific computing, data analysis, web development, and general-purpose programming. Created by Guido van Rossum and first released in 1991, Python emphasizes code readability and simplicity, allowing programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java.

The language's design philosophy, often summarized in the *Zen of Python* (accessible by typing `import this` in any Python interpreter), emphasizes principles such as "Beautiful is better than ugly," "Explicit is better than implicit," and "Readability counts." These principles have guided Python's development and contributed to its widespread adoption across diverse domains.

This comprehensive guide introduces Python programming from the ground up, with a particular focus on techniques applicable to scientific simulations and computational research. We will explore Python's fundamental concepts, syntax, data structures, object-oriented programming features, and the ecosystem of scientific computing libraries that make Python an excellent choice for research and simulation work.

1.1 Why Python for Scientific Computing?

Python has emerged as a dominant force in scientific computing for several compelling reasons:

- **Ease of Learning:** Python's clean syntax and readable code make it accessible to researchers and scientists who may not have extensive programming backgrounds.
- **Rich Ecosystem:** The availability of powerful libraries such as NumPy, SciPy, Matplotlib, and pandas provides ready-made solutions for numerical computing, data visualization, and analysis.
- **Rapid Prototyping:** Python's interpreted nature allows for quick testing of ideas without lengthy compilation cycles.
- **Integration Capabilities:** Python can easily interface with code written in other languages (C, C++, Fortran), allowing performance-critical sections to be optimized while maintaining Python's ease of use for the rest of the application.
- **Community Support:** A large, active community provides extensive documentation, tutorials, and packages for virtually any computational task.

2 Understanding Execution Models: Interpreted vs. Compiled Languages

One of the fundamental distinctions in programming languages is how they transform human-readable source code into machine-executable instructions. Understanding this distinction is crucial for comprehending Python's behavior, performance characteristics, and development workflow.

2.1 Compiled Languages: The C++ Model

A **compiled language** like C++ follows a multi-stage process before code execution:

1. **Preprocessing:** The preprocessor handles directives (lines beginning with `#`), such as `#include` statements, expanding them and preparing the code for compilation.
2. **Compilation:** The compiler translates the preprocessed source code into assembly language or directly into machine code, creating object files.
3. **Linking:** The linker combines multiple object files and libraries into a single executable binary file.
4. **Execution:** The resulting binary file can be run directly by the operating system, executing machine instructions native to the processor.

Analogy: Think of compiled languages like translating an entire book from English into Japanese before anyone reads it. Once translated, Japanese readers can read the book directly without needing the English original or a translator present.

Advantages of Compiled Languages:

- **Performance:** Direct machine code execution is typically faster than interpreted execution.
- **Early Error Detection:** Many errors are caught at compile time, before the program runs.
- **Optimization:** Compilers can perform sophisticated optimizations during the translation process.

Disadvantages:

- **Compilation Time:** Every code change requires recompilation, which can be time-consuming for large projects.
- **Platform Dependence:** Binaries are typically platform-specific; code must be recompiled for different operating systems or architectures.
- **Development Cycle:** The edit-compile-run cycle can slow down development and debugging.

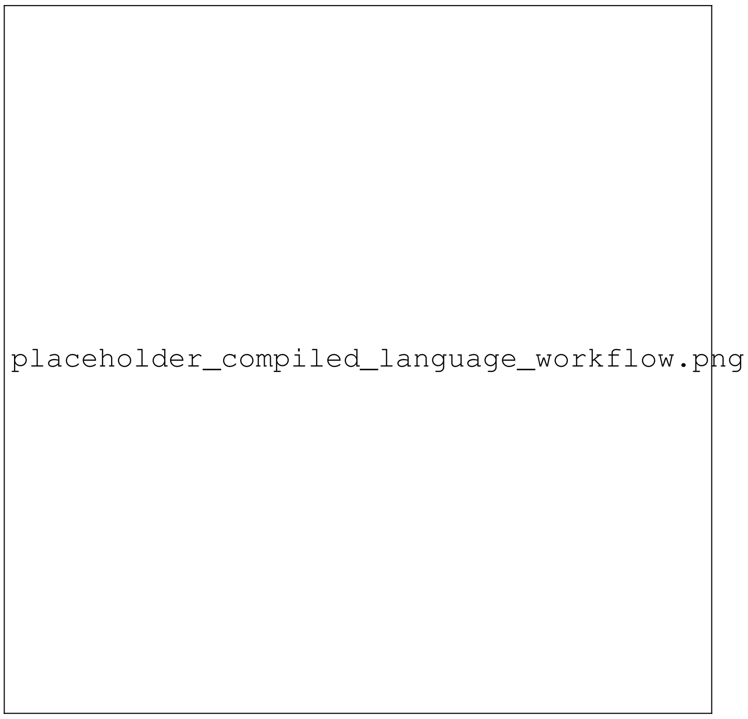

Figure 1: The compilation workflow for C++: source code passes through preprocessor, compiler, and linker stages to produce a binary executable that is then run to generate results.

2.2 Interpreted Languages: The Python Model

An **interpreted language** like Python follows a different approach:

1. **Source Code:** The programmer writes Python code in plain text files.
2. **Interpretation:** The Python interpreter reads the source code line by line (or statement by statement), translating it into bytecode and executing it on the fly.
3. **Results:** Output is generated directly during interpretation without creating a separate executable file.

Analogy: Think of interpreted languages like having a simultaneous translator at a conference. As the English speaker talks, the translator immediately translates each sentence into Japanese. There's no pre-translated document; translation happens in real-time during execution.

Advantages of Interpreted Languages:

- **Rapid Development:** No compilation step means immediate testing of code changes.
- **Portability:** The same source code runs on any platform with a compatible interpreter.

Figure 2: The interpretation workflow for Python: source code is directly processed by the interpreter, which executes it immediately to produce results, without an intermediate binary compilation stage.

- **Dynamic Features:** Runtime interpretation enables powerful dynamic features like reflection and dynamic typing.
- **Interactive Mode:** Interpreters often provide interactive shells (REPLs - Read-Eval-Print Loops) for experimentation.

Disadvantages:

- **Performance:** Interpreted code typically runs slower than compiled code due to the overhead of runtime interpretation.
- **Runtime Errors:** Some errors only appear during execution, which might not happen until specific code paths are triggered.
- **Interpreter Dependency:** The program requires an interpreter to be installed on any system where it runs.

Note: In practice, the distinction is more nuanced. Python actually compiles source code to bytecode (stored in `.pyc` files), which is then interpreted by the Python Virtual Machine. This hybrid approach balances some benefits of both models. Additionally, tools like PyPy use Just-In-Time (JIT) compilation to improve performance, while projects like Cython allow compilation of Python-like code to C extensions.

3 Type Systems: Static vs. Dynamic Typing

Type systems represent another fundamental difference between programming languages, affecting how variables are declared, used, and checked for correctness.

3.1 Static Typing: The C++ Approach

C++ employs **static typing**, which means:

1. **Explicit Type Declaration:** Variable types must be declared explicitly when variables are created.
2. **Compile-Time Type Checking:** The compiler verifies that all operations are type-safe before the program runs.
3. **Type Immutability:** Once a variable is declared with a specific type, it cannot change to a different type.

Example in C++:

```cpp
int x = 5;          // x is declared as an integer
x = 10;             // Valid: assigning another integer
x = "hello";        // COMPILATION ERROR: cannot assign string to int
double y = 3.14;    // y is declared as a double
y = 42;             // Valid: 42 is converted to 42.0
```

Listing 1: Static typing in C++: explicit type declarations and compile-time enforcement

Analogy: Think of statically typed variables like labeled storage boxes in a warehouse. A box labeled "Integers Only" can only contain integers. If you try to put a string in that box, the warehouse management system (compiler) will reject it before the warehouse even opens for business.

Advantages of Static Typing:

- **Early Error Detection:** Type mismatches are caught at compile time, preventing many runtime errors.
- **Performance:** The compiler can optimize code based on known types, often resulting in faster execution.
- **Documentation:** Type declarations serve as inline documentation, making code intent clearer.
- **Tooling Support:** IDEs can provide better autocomplete, refactoring, and error detection based on type information.

3.2 Dynamic Typing: The Python Approach

Python uses **dynamic typing**, characterized by:

1. **No Explicit Type Declaration:** Variables are created simply by assignment; no type specification is required.
2. **Runtime Type Checking:** Type compatibility is verified during program execution.
3. **Type Flexibility:** Variables can be reassigned to values of completely different types.

Example in Python:

```python
x = 5           # x refers to an integer object
print(type(x)) # <class 'int'>
x = "hello"     # Perfectly valid: x now refers to a string object
print(type(x)) # <class 'str'>
x = [1, 2, 3]   # Also valid: x now refers to a list object
print(type(x)) # <class 'list'>
```

Listing 2: Dynamic typing in Python: no type declarations and flexible type reassignment

Analogy: Think of dynamically typed variables like flexible containers. The container (variable name) doesn't care what's inside it—integers, strings, lists, or any other object. You can empty it and fill it with something completely different at any time.

Advantages of Dynamic Typing:

- **Flexibility:** Code can work with different types without modification (duck typing).
- **Conciseness:** Less boilerplate code; no need for verbose type declarations.
- **Rapid Prototyping:** Quick experimentation without worrying about type systems.
- **Generic Programming:** Easier to write code that works with multiple types.

Disadvantages:

- **Runtime Errors:** Type errors only appear during execution, potentially in production.
- **Performance:** Runtime type checking and flexibility can slow execution.
- **Less IDE Support:** Harder for tools to provide accurate autocomplete without explicit types.
- **Implicit Documentation:** Type information isn't immediately visible from variable declarations.

Modern Python: Python 3.5+ introduced optional type hints, allowing developers to annotate types without enforcing them at runtime. Tools like `mypy` can perform static type checking on annotated code, combining benefits of both approaches:

```python
def add_numbers(a: int, b: int) -> int:
    """Add two integers and return the result."""
    return a + b

result: int = add_numbers(5, 10)  # Type hints are optional
    documentation
```

Listing 3: Python type hints provide optional static type checking

4 Python Versions: Python 2 vs. Python 3

Understanding Python's version history is important for navigating the ecosystem and making informed decisions about which version to use.

4.1 Python 2.x: The Legacy Version

Python 2.x was the dominant version from approximately 2000 to 2010, with Python 2.7 being the final release in this series.

Key Facts:

- **Timeline:** Released around 2000; Python 2.7 (the last 2.x version) was released in 2010.
- **Extended Support:** Python 2.7 received extended support until 2020, with the final release (2.7.18) in April 2020.
- **End of Life:** Official support ended on January 1, 2020 (see `https://pythonclock.org/`).
- **Legacy Codebases:** Many older projects and libraries were written in Python 2.
- **Current Status:** No longer recommended for new projects; considered obsolete.

Why Python 2 Still Matters: While Python 2 is officially deprecated, you may encounter it in:

- Legacy systems and older scientific code
- Older tutorials and documentation
- Some embedded systems that haven't been updated
- Systems where updating would break critical dependencies

4.2 Python 3.x: The Modern Standard

Python 3.x represents the present and future of the language, with the first release in 2008.

Key Facts:

- **Timeline:** First released in December 2008 (Python 3.0).
- **Breaking Changes:** Not backward compatible with Python 2, requiring code migration.
- **Improvements:** Cleaner design, better Unicode support, improved consistency.
- **Ecosystem:** Virtually all major libraries now support Python 3.
- **Current Dominance:** According to the 2025 Stack Overflow Developer Survey, Python 3 is overwhelmingly the standard.

Major Differences Between Python 2 and 3:

1. **Print Statement vs. Function:**

```
# Python 2
print "Hello, World!"  # Statement

# Python 3
print("Hello, World!")  # Function

```

Listing 4: Print differences between Python 2 and 3

2. **Integer Division:**

```
# Python 2
5 / 2   # Result: 2 (integer division)

# Python 3
5 / 2   # Result: 2.5 (true division)
5 // 2  # Result: 2 (floor division)

```

Listing 5: Division behavior differences

3. **Unicode Strings:**
   - Python 2: Strings are bytes by default; Unicode requires `u"string"` prefix.
   - Python 3: Strings are Unicode by default; bytes require `b"string"` prefix.

4. **Range Function:**
   - Python 2: `range()` returns a list; `xrange()` returns an iterator.

- Python 3: `range()` returns an iterator; `xrange()` doesn't exist.

Recommendation: Always use Python 3.x for new projects. As of 2025, Python 3.10+ is widely adopted, with Python 3.11 and 3.12 offering significant performance improvements and new features.

5 Installing and Setting Up Python

Python's availability across different operating systems makes it accessible to virtually any developer. However, installation methods vary by platform.

5.1 Installation on Linux

Most modern Linux distributions come with Python pre-installed, though it may be necessary to install additional packages or ensure Python 3 is the default version.

Installation Steps:

1. **Check Existing Installation:**

```
python3 --version  # Check Python 3 version
which python3      # Find Python 3 location

```

Listing 6: Checking Python installation on Linux

2. **Install via Package Manager:** Different distributions use different package managers:

```
# Debian/Ubuntu
sudo apt update
sudo apt install python3 python3-pip python3-dev

# Fedora/RHEL/CentOS
sudo yum install python3 python3-pip

# OpenSUSE
sudo zypper install python3 python3-pip

```

Listing 7: Installing Python on various Linux distributions

3. **Install Additional Packages:** You may need to install additional packages for a "full" Python environment:

```
sudo apt install python3-numpy python3-scipy python3-matplotlib

```

Listing 8: Installing additional Python packages

5.2 Installation on macOS

macOS typically includes a basic Python installation, but you should verify it's Python 3 and consider using a package manager for better control.

Installation Methods:

1. **Built-in Python:**

```
python3 --version  # Verify Python 3 is available

```

Listing 9: Checking macOS Python installation

2. **Homebrew (Recommended):**

```
# Install Homebrew if not already installed
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/
    Homebrew/install/HEAD/install.sh)"

# Install Python 3
brew install python3

```

Listing 10: Installing Python via Homebrew

3. **MacPorts:**

```
sudo port install python310  # Install Python 3.10
sudo port select --set python python310  # Set as default

```

Listing 11: Installing Python via MacPorts

4. **Python Distributions:**

   - **Anaconda:** (https://www.anaconda.com/download/) - Comprehensive distribution with scientific packages
   - **Canopy:** (https://assets.enthought.com/downloads/edm/) - Enthought's scientific distribution

5.3 Installation on Windows

Windows doesn't include Python by default, so it must be explicitly installed.

Installation Methods:

1. **Official Python Installer:**

   - Download from https://www.python.org
   - Run installer and ensure "Add Python to PATH" is checked
   - Verify installation in Command Prompt or PowerShell

2. **Microsoft Store:**

   - Search for "Python" in Microsoft Store

    - Install directly with automatic PATH configuration

3. **Windows Subsystem for Linux (WSL):**

```
# Install WSL2 first, then:
sudo apt update
sudo apt install python3 python3-pip

```

Listing 12: Using Python in WSL

   For graphical applications, you may need an X11 server:

    - Windows 11+ with WSL2 includes built-in X11 support
    - Older versions can use VcXsrv or similar X servers

4. **Python Distributions:** Same as macOS: Anaconda or Canopy provide comprehensive scientific computing environments.

Important Note: Some older operating systems may still have Python 2.x as the default. Always verify you're using Python 3 by running `python -version` or `python3 -version`.

5.4 Python Distribution Recommendations

For scientific computing, specialized Python distributions offer significant advantages over standard installations:

Anaconda/Miniconda:

- Includes 250+ pre-installed scientific packages
- Conda package manager simplifies dependency management
- Environment management for project isolation
- Cross-platform consistency
- Free and open source

Enthought Canopy:

- Curated collection of scientific packages
- Integrated development environment
- Package management tools
- Educational resources and documentation

For most scientific computing purposes, Anaconda is the recommended distribution due to its comprehensive package collection and excellent environment management capabilities.

6 Python's Library Ecosystem

One of Python's greatest strengths is its extensive ecosystem of libraries and packages, which extend the language's capabilities far beyond what's built into the core.

6.1 The Python Standard Library

The **Python Standard Library** consists of over 100 modules that ship with Python, providing functionality for common programming tasks without requiring external installation.

Categories of Standard Library Modules:

- **Text Processing:** `string`, `re` (regular expressions), `textwrap`
- **Data Types:** `collections`, `array`, `copy`, `enum`
- **Mathematics:** `math`, `random`, `statistics`, `decimal`
- **File and Directory Access:** `os`, `pathlib`, `glob`, `shutil`
- **Data Persistence:** `pickle`, `json`, `sqlite3`
- **Networking:** `socket`, `http`, `urllib`, `ftplib`
- **Internet Protocols:** `email`, `xml`, `html`
- **Concurrency:** `threading`, `multiprocessing`, `asyncio`
- **Debugging and Testing:** `unittest`, `pdb`, `timeit`

Documentation: Complete documentation available at `https://docs.python.org/3/library/index.html`

Installation Note: While the standard library comes with Python, some Linux distributions may split it into separate packages. You may need to use your OS package manager to install the complete library.

Exploring the Standard Library: The Python Module of the Week (PyMOTW-3) provides excellent tutorials and examples for standard library modules: `https://pymotw.com/3/index.html`

6.2 The Python Package Index (PyPI)

The **Python Package Index** (PyPI, pronounced "pie-pee-eye") is a repository of software packages for Python, containing over 500,000 projects as of 2025.

Key Facts:

- **URL:** `https://pypi.org/`
- **Scope:** Covers virtually every domain: web development, data science, machine learning, automation, games, and more

- **Installation Tool:** `pip` (Python Package Installer)
- **Open Source:** Most packages are open source and free to use

Installing Packages with pip:

```
# Install a package
pip install numpy

# Install a specific version
pip install numpy==1.24.0

# Upgrade a package
pip install --upgrade numpy

# Uninstall a package
pip uninstall numpy

# List installed packages
pip list

# Show information about a package
pip show numpy
```

Listing 13: Basic pip usage for installing packages

Virtual Environments: Best practice involves using virtual environments to isolate project dependencies:

```
# Create a virtual environment
python3 -m venv myproject_env

# Activate the environment (Linux/macOS)
source myproject_env/bin/activate

# Activate the environment (Windows)
myproject_env\Scripts\activate

# Install packages in the virtual environment
pip install numpy scipy matplotlib

# Deactivate when done
deactivate
```

Listing 14: Creating and using virtual environments

Why Use Virtual Environments?

- **Dependency Isolation:** Different projects can use different versions of the same package
- **Reproducibility:** Easily recreate exact environments using `requirements.txt`
- **System Protection:** Avoid conflicts with system Python packages
- **Clean Testing:** Start with a clean slate for each project

6.3 Comparing Python and C++ Library Ecosystems

While Python's package ecosystem is centralized through PyPI, C++ has a more fragmented landscape:

- **C++ Libraries:** Often distributed through various channels (system package managers, source repositories, dedicated sites)
- **Reference:** https://en.cppreference.com/w/cpp/links/libs
- **Installation:** Typically more complex, often requiring manual compilation
- **Header-Only Libraries:** Many C++ libraries are header-only, simplifying distribution

Python's centralized package management through PyPI and pip provides a significantly more streamlined experience for installing and managing dependencies compared to C++'s more fragmented ecosystem.

7 Essential Scientific Computing Packages

Python's dominance in scientific computing is largely due to a core set of mature, well-designed packages that provide efficient numerical computing capabilities.

7.1 NumPy: Numerical Python

NumPy is the foundational package for numerical computing in Python, providing:

- **Multi-dimensional Arrays:** Efficient storage and manipulation of arrays and matrices
- **Mathematical Functions:** Comprehensive collection of mathematical operations
- **Linear Algebra:** Basic linear algebra operations
- **Random Number Generation:** Statistical distributions and random sampling
- **Fourier Transforms:** Fast Fourier Transform (FFT) operations

Installation:

```
pip install numpy
```

Basic Example:

```python
import numpy as np

# Create arrays
a = np.array([1, 2, 3, 4, 5])
b = np.array([[1, 2, 3], [4, 5, 6]])

# Mathematical operations (vectorized)
c = a * 2           # Multiply all elements by 2
d = np.sin(a)       # Apply sine to all elements
e = a + a           # Element-wise addition

# Statistical operations
mean = np.mean(a)
std = np.std(a)

print(f"Array a: {a}")
print(f"Doubled: {c}")
print(f"Mean: {mean}, Std: {std}")
```

Listing 15: NumPy array operations

Why NumPy Matters: NumPy arrays are implemented in C, making operations much faster than pure Python lists. This forms the foundation for virtually all scientific computing in Python.

7.2 SciPy: Scientific Python

SciPy builds on NumPy to provide higher-level scientific computing functionality:

- **Optimization:** Function minimization, curve fitting, root finding
- **Integration:** Numerical integration and differential equation solvers
- **Interpolation:** Data interpolation and smoothing
- **Linear Algebra:** Advanced linear algebra operations beyond NumPy
- **Statistics:** Statistical distributions, tests, and functions
- **Signal Processing:** Filtering, convolution, Fourier analysis
- **Image Processing:** Basic image manipulation and filtering
- **Sparse Matrices:** Efficient handling of sparse matrix operations

Installation:

```
pip install scipy
```

Example:

```python
import numpy as np
from scipy import optimize

# Define a function to minimize
```

```python
def f(x):
    return (x - 3)**2 + 5

# Find the minimum
result = optimize.minimize(f, x0=0)
print(f"Minimum at x = {result.x[0]:.4f}")
print(f"Minimum value = {result.fun:.4f}")
```

Listing 16: SciPy optimization example

7.3 Matplotlib: Visualization

Matplotlib is Python's primary plotting library, providing publication-quality figures:

- **2D Plots:** Line plots, scatter plots, bar charts, histograms
- **3D Plots:** Surface plots, wireframes, 3D scatter plots
- **Customization:** Fine control over every element of a figure
- **Multiple Formats:** Export to PNG, PDF, SVG, and more
- **Interactive Plotting:** Zooming, panning, and updating plots dynamically

Installation:

```
pip install matplotlib
```

Example:

```python
import numpy as np
import matplotlib.pyplot as plt

# Generate data
x = np.linspace(0, 2*np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# Create plot
plt.figure(figsize=(10, 6))
plt.plot(x, y1, label='sin(x)', linewidth=2)
plt.plot(x, y2, label='cos(x)', linewidth=2)
plt.xlabel('x')
plt.ylabel('y')
plt.title('Trigonometric Functions')
plt.legend()
plt.grid(True)
plt.savefig('trig_functions.png', dpi=300)
plt.show()
```

Listing 17: Creating plots with Matplotlib

7.4 H5py: HDF5 Interface

H5py provides a Pythonic interface to the HDF5 binary data format:

- **Hierarchical Data Format:** Organize data in tree-like structures
- **Large Datasets:** Efficiently store and access datasets larger than RAM
- **Metadata:** Attach arbitrary metadata to datasets
- **Compression:** Built-in data compression support
- **Cross-Platform:** HDF5 files are portable across systems

Installation:

```
pip install h5py
```

Use Cases:

- Storing simulation results
- Managing large scientific datasets
- Sharing data between different programming languages

7.5 mpi4py: Parallel Computing

mpi4py provides Python bindings for the Message Passing Interface (MPI):

- **Distributed Computing:** Run Python programs across multiple processors or nodes
- **High-Performance Computing:** Essential for large-scale scientific simulations
- **Standard Interface:** Uses the widely-adopted MPI standard

Installation:

```
pip install mpi4py
```

Note: Requires an MPI implementation (OpenMPI, MPICH, etc.) to be installed on the system. This topic is covered extensively in High-Performance Computing courses.

7.6 SymPy: Symbolic Mathematics

SymPy is a computer algebra system written entirely in Python:

- **Symbolic Computation:** Algebraic manipulation of mathematical expressions
- **Equation Solving:** Solve equations symbolically

- **Calculus:** Symbolic differentiation and integration
- **Linear Algebra:** Matrix operations with symbolic entries
- **Pretty Printing:** Beautiful mathematical output

Installation:

```
pip install sympy
```

Example:

```python
from sympy import symbols, diff, integrate, sin, cos, exp

# Define symbolic variables
x, y = symbols('x y')

# Define an expression
expr = x**2 + 2*x*y + y**2

# Differentiate
dx = diff(expr, x)  # Result: 2*x + 2*y
dy = diff(expr, y)  # Result: 2*x + 2*y

# Integrate
integral = integrate(sin(x)*exp(x), x)

print(f"Expression: {expr}")
print(f"d/dx: {dx}")
print(f"Integral: {integral}")
```

Listing 18: Symbolic mathematics with SymPy

These packages form the core of Python's scientific computing ecosystem. Together, they provide functionality comparable to commercial systems like MATLAB while remaining free and open source.

8 Python Execution Environments

Python offers multiple ways to write and execute code, each suited to different workflows and purposes.

8.1 The Standard Python Interactive Shell

The basic Python interpreter provides an interactive Read-Eval-Print Loop (REPL):

Starting the Shell:

```
$ python
Python 3.10.8 (main, Oct 12 2022, 09:46:29) [Clang 14.0.0] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Listing 19: Launching the Python interactive shell

Features:

- **Interactive Execution:** Enter code line by line and see immediate results
- **Experimentation:** Quick testing of ideas and syntax
- **Learning:** Explore language features interactively
- **Prompt:** The »> prompt indicates Python is ready for input

Example Session:

```
>>> x = 5
>>> y = 10
>>> x + y
15
>>> print("Hello, World!")
Hello, World!
>>> import math
>>> math.sqrt(16)
4.0
```

Listing 20: Using the Python interactive shell

Limitations:

- No syntax highlighting
- Limited command history
- No tab completion
- Code not easily saved

8.2 Running Python Scripts

For programs longer than a few lines, code should be saved in files with the `.py` extension and executed as scripts:

Creating a Script: Create a file named `script.py`:

```
#!/usr/bin/env python3
"""
A simple demonstration script.
"""

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

def main():
    """Main function."""
    message = greet("World")
    print(message)

    # Perform calculations
    result = sum(range(1, 11))
    print(f"Sum of 1-10: {result}")

if __name__ == "__main__":
    main()
```

Listing 21: A simple Python script (script.py)

Running the Script:

```
$ python script.py
Hello, World!
Sum of 1-10: 55
```

Listing 22: Executing a Python script

The Shebang Line: The first line (`#!/usr/bin/env python3`) allows the script to be executed directly on Unix-like systems:

```
$ chmod +x script.py  # Make executable
$ ./script.py         # Run directly
```

8.3 IPython: Enhanced Interactive Shell

IPython is an enhanced interactive Python shell with powerful features for productive development and exploration:

Starting IPython:

```
$ ipython
Python 3.13.5 (main, Jun 25 2025, 18:55:22) [GCC 14.2.0]
Type 'copyright', 'credits' or 'license' for more information
IPython 9.8.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Listing 23: Launching IPython

Key Features:

- **Syntax Highlighting:** Color-coded code for readability
- **Tab Completion:** Press Tab to autocomplete variable names, functions, and module members
- **Magic Commands:** Special commands prefixed with `%` or `%%`
- **History:** Advanced command history with search and replay
- **Help System:** Type `?object` for documentation, `??object` for source code
- **Shell Commands:** Execute system commands with `!command`
- **Numbered Prompts:** Input and output are numbered for easy reference

Example IPython Session:

```
In [1]: import numpy as np

In [2]: arr = np.array([1, 2, 3, 4, 5])

In [3]: arr.
```

```
arr.T           arr.all         arr.argmin      arr.astype
arr.any         arr.argmax      arr.argsort     ...

In [4]: %timeit sum(range(1000))
5.23  s     123 ns per loop (mean    std. dev. of 7 runs, 100000 loops each)

In [5]: np.sqrt?
Signature: np.sqrt(x, /, out=None, *, where=True, casting='same_kind', ...)
Docstring:
Return the non-negative square-root of an array, element-wise.
...

In [6]: %hist
1: import numpy as np
2: arr = np.array([1, 2, 3, 4, 5])
...
```

Listing 24: Using IPython's enhanced features

Useful Magic Commands:

- `%timeit`: Measure execution time of statements
- `%run`: Execute a Python script
- `%hist`: Display command history
- `%pwd`: Print working directory
- `%cd`: Change directory
- `%load`: Load code from file into cell
- `%save`: Save input history to file
- `%%writefile`: Write cell contents to file

Installation:

```
pip install ipython
```

8.4 Jupyter Notebooks: Interactive Computing Environment

Jupyter Notebooks provide a web-based interactive computing environment that combines code execution, rich text, mathematics, plots, and media in a single document.

Starting Jupyter:

```
# Classic Notebook interface
$ jupyter notebook

# JupyterLab (next-generation interface)
$ jupyter lab
```

Listing 25: Launching Jupyter Notebook or JupyterLab

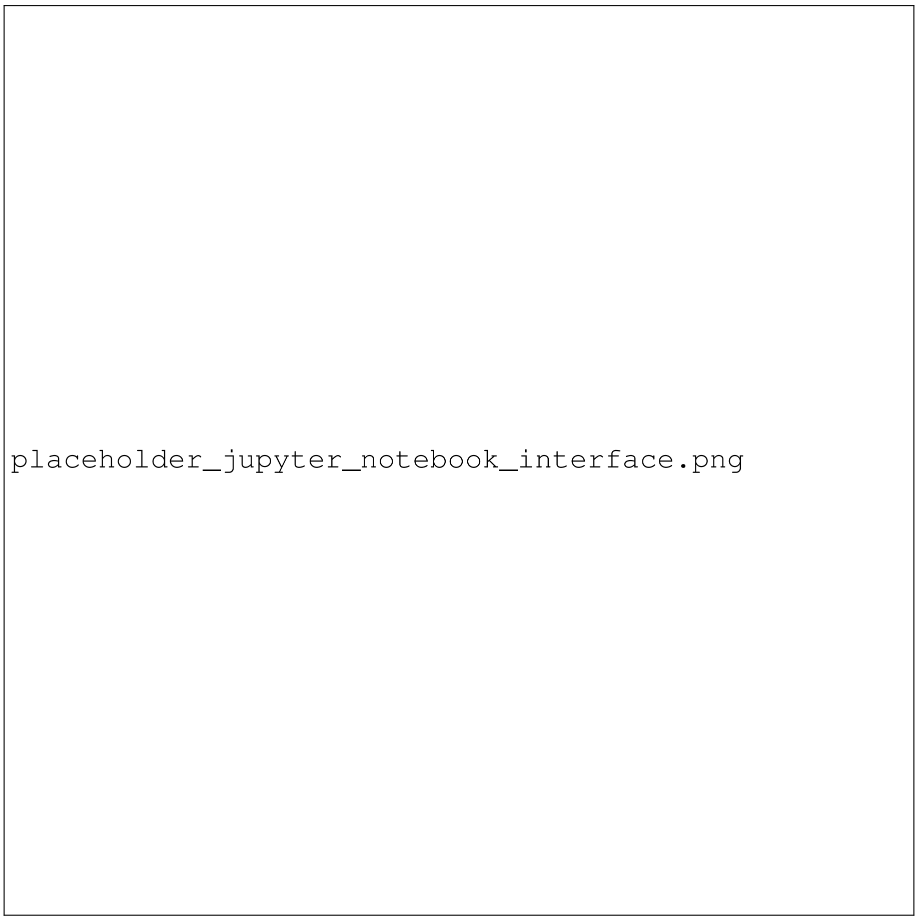

Figure 3: The Jupyter Notebook interface showing code cells, output, and markdown text combined in a single document. The browser-based interface allows for interactive computing with immediate visual feedback.

Both commands start a local web server and open a browser window with the Jupyter interface.

Key Features:

- **Notebook Format:** Documents contain code cells, markdown cells, and output
- **Interactive Execution:** Run cells individually, in any order
- **Rich Output:** Plots, images, videos, and formatted text display inline
- **Markdown Support:** Write formatted text, equations (LaTeX), and documentation
- **Sharing:** Notebooks are JSON files that can be easily shared and version-controlled
- **Export:** Convert notebooks to HTML, PDF, slides, or Python scripts

- **Kernels:** Support for multiple languages beyond Python (R, Julia, etc.)

Cell Types:

1. **Code Cells:** Contain executable Python code
2. **Markdown Cells:** Contain formatted text, equations, and documentation
3. **Raw Cells:** Contain unformatted text (rarely used)

Example Notebook Content:

```
# Markdown Cell
# Analysis of Sine Function
This notebook explores properties of the sine function.

# Code Cell 1
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)

# Code Cell 2
plt.figure(figsize=(10, 6))
plt.plot(x, y)
plt.title('Sine Function')
plt.xlabel('x')
plt.ylabel('sin(x)')
plt.grid(True)
plt.show()

# Markdown Cell
## Observations
- The sine function is periodic with period $2\pi$
- Maximum value is 1, minimum is -1
```

Listing 26: Example Jupyter Notebook cells

Use Cases:

- Data exploration and analysis
- Scientific research documentation
- Teaching and tutorials
- Reproducible research
- Presentation of results with code

Installation:

```
pip install jupyter notebook jupyterlab
```

JupyterLab vs. Notebook:

- **Notebook:** Classic, simpler interface
- **JupyterLab:** Modern, extensible interface with multiple panes, terminal, text editor, and more

8.5 Note on Python vs. Python3 Command

The command used to invoke Python (`python` vs. `python3`) varies across systems:

System-Dependent Behavior:

- **Modern Systems:** `python` typically refers to Python 3
- **Legacy Systems:** `python` might still refer to Python 2
- **Safe Approach:** Use `python3` to explicitly invoke Python 3

PEP 394: Python Enhancement Proposal 394 provides guidelines on this topic: https://peps.python.org/pep-0394/

Checking Your System:

```
python --version   # Check what 'python' refers to
python3 --version  # Check Python 3 version
```

9 Getting Help in Python

Python includes a comprehensive built-in help system, making it easy to learn about functions, classes, and modules without leaving the interpreter.

9.1 The help() Function

The `help()` function provides interactive access to Python's documentation:

General Help:

```
>>> help()
Welcome to Python 3.10's help utility!
...
help> quit
```

Listing 27: Starting the interactive help system

Help on Specific Objects:

```
>>> help(int)
Help on class int in module builtins:

class int(object)
 |  int([x]) -> integer
 |  int(x, base=10) -> integer
 |
 |  Convert a number or string to an integer...
 ...

>>> help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    ...

>>> import math
>>> help(math.sqrt)
Help on built-in function sqrt in module math:

sqrt(x, /)
    Return the square root of x.
```
Listing 28: Getting help on specific objects

What help() Shows:

- Function signatures (parameters and return types)
- Docstrings (documentation written by developers)
- Available methods and attributes
- Usage examples (if provided)

9.2 IPython's Enhanced Help

IPython provides an even more convenient help system:
Quick Help:

```
In [1]: import numpy as np

# Append ? for quick help
In [2]: np.sqrt?
Signature: np.sqrt(x, /, out=None, *, where=True, ...)
Docstring:
Return the non-negative square-root of an array, element-wise.
...

# Append ?? for source code
In [3]: np.sqrt??
Signature: np.sqrt(x, /, out=None, *, where=True, ...)
Source:
# C function - source not available

# Tab completion for methods
In [4]: np.arr<Tab>
np.array        np.array_equal   np.array_repr
np.array_api    np.array_equiv   np.array_split
```
Listing 29: IPython's help shortcuts

9.3 Online Documentation

While built-in help is convenient, online documentation often provides more comprehensive information:

- **Official Python Docs:** https://docs.python.org/3/

- **Library References:** Each major package has detailed documentation
- **Stack Overflow:** Community Q&A for specific problems
- **Real Python:** Tutorials and guides

10 Python Syntax Fundamentals

Now that we understand Python's ecosystem and execution environments, let's dive into the language's syntax and fundamental programming constructs.

10.1 A First Complete Example

Let's examine a complete Python program that demonstrates several key syntactic features:

```python
x = 11/3 - 9//2  # this is a comment
y = "Hola"
z = 3.14         # another comment

if y == "Hola" or z >= 3:
    x = x + 2
    y = y + " mundo!"  # string concatenation

print(y)
print(f"{x = :5.3f}")  # f-string formatting

year, month, day = 1943, 6, 15
hour, minute, second = 23, 6, 54

if 1900 < year < 2100 and 1 <= month <= 12 \
        and 1 <= day <= 31 and 0 <= hour < 24 \
        and 0 <= minute < 60 and 0 <= second < 60:
    print("Looks like a valid date!",
          "Indeed!")
```

Listing 30: A comprehensive Python syntax example

Syntactic Elements Demonstrated:

1. **Comments:** Lines beginning with `#` are ignored by Python
2. **Dynamic Typing:** Variables don't require type declarations
3. **Operators:**
   - / performs floating-point division (11/3 = 3.666...)
   - // performs floor division (9//2 = 4)
4. **Conditional Statements:** `if` blocks with Boolean expressions
5. **Logical Operators:** `or`, `and`, comparison operators
6. **Indentation:** Code blocks are defined by indentation (4 spaces is standard)

7. **String Operations:** Concatenation with +
8. **Formatted Output:** f-strings for flexible string formatting
9. **Multiple Assignment:** Assign several variables in one statement
10. **Chained Comparisons:** `1900 < year < 2100` is valid Python
11. **Line Continuation:** Backslash (\) continues long lines
12. **Implicit Line Joining:** Lines can continue inside parentheses without backslash

Output:

```
Hola mundo!
x = 1.667
Looks like a valid date! Indeed!
```

10.2 Comments

Comments are text that Python ignores, used to document code for human readers.

Single-Line Comments:

```python
# This is a comment
x = 5  # This is also a comment (following code)
```

Listing 31: Single-line comments in Python

Multi-Line Comments: Python doesn't have a specific multi-line comment syntax, but multi-line strings can serve this purpose:

```python
"""
This is a multi-line comment.
It can span multiple lines.
It's technically a string, but if not assigned,
it's ignored by Python.
"""

'''
Single quotes work too for multi-line strings.
'''
```

Listing 32: Multi-line comments using triple-quoted strings

Best Practices:

- Use comments to explain *why*, not *what*
- Keep comments up-to-date with code changes
- Avoid obvious comments (`# increment x` for `x += 1`)
- Use docstrings (covered later) for function/class documentation

10.3 Line Continuation

Python statements typically end at the line's end, but long statements can be continued across lines:

Explicit Continuation (Backslash):

```python
total = 1 + 2 + 3 + \
        4 + 5 + 6 + \
        7 + 8 + 9

# Conditional spanning multiple lines
if condition1 and condition2 \
        and condition3:
    print("All conditions met")
```

Listing 33: Explicit line continuation with backslash

Implicit Continuation (Inside Delimiters): Lines automatically continue inside parentheses, brackets, and braces:

```python
# Inside parentheses (preferred method)
total = (1 + 2 + 3 +
         4 + 5 + 6 +
         7 + 8 + 9)

# Function call with many arguments
result = some_function(arg1, arg2,
                       arg3, arg4,
                       arg5, arg6)

# List spanning multiple lines
numbers = [1, 2, 3,
           4, 5, 6,
           7, 8, 9]

# Print with multiple arguments
print("This is a long message",
      "that spans multiple lines",
      "without needing backslashes")
```

Listing 34: Implicit line continuation inside delimiters

Best Practice: Prefer implicit continuation (using parentheses) over explicit continuation (backslashes), as it's cleaner and less error-prone.

11 Basic Built-In Data Types

Python provides several built-in data types that form the foundation of all Python programs.

11.1 Boolean Type

Booleans represent truth values:

```python
x = True   # Note: Capital T
y = False  # Note: Capital F
```

```python

# Boolean operations
a = True and False   # False
b = True or False    # True
c = not True         # False

# Comparison operations return booleans
d = 5 > 3           # True
e = 10 == 10        # True
f = "hello" != "world"  # True
```

Listing 35: Boolean values in Python

Key Points:

- Must be capitalized: `True` and `False`, not `true` and `false`
- Different from C++, where `true` and `false` are lowercase
- Many values are "truthy" or "falsy" when used in Boolean contexts

Truthiness in Python:

```python
# Falsy values (evaluate to False in boolean context)
bool(0)          # False
bool(0.0)        # False
bool("")         # False (empty string)
bool([])         # False (empty list)
bool({})         # False (empty dictionary)
bool(None)       # False

# Truthy values (evaluate to True)
bool(1)          # True
bool(-1)         # True
bool("hello")    # True (non-empty string)
bool([1, 2])     # True (non-empty list)
```

Listing 36: Truthy and falsy values

11.2 Integer Type

Integers are whole numbers without size limits in Python 3:

```python
x = 1
y = 42
z = -17

# Python integers have arbitrary precision
big = 12345678901234567890123456789012345678901234567890
bigger = big ** 2  # No overflow!

# Common operations
addition = 5 + 3       # 8
subtraction = 5 - 3    # 2
multiplication = 5 * 3  # 15
division = 5 / 3       # 1.6666... (float result)
floor_division = 5 // 3  # 1 (integer result, rounded down)
```

```python
modulo = 5 % 3          # 2 (remainder)
power = 2 ** 10         # 1024

# Different number bases
binary = 0b1010         # 10 in decimal
octal = 0o12            # 10 in decimal
hexadecimal = 0xA       # 10 in decimal
```
Listing 37: Integer operations in Python

Important Differences from C++:

- **No Size Limits:** Python 3 integers grow as needed (no overflow)
- **Division Behavior:** `/` always returns float; use `//` for integer division
- **No Type Variants:** No separate `short`, `long`, `long long`, etc.

11.3 Floating-Point Type

Floats represent decimal numbers:

```python
x = 1.0         # Float with decimal point
y = 1.          # Also valid
z = .5          # Also valid (0.5)

# Scientific notation
a = 1e0         # 1.0
b = 1.5e3       # 1500.0
c = 2e-4        # 0.0002
d = 3.14E2      # 314.0 (capital E also works)

# Float operations
result = 0.1 + 0.2  # 0.30000000000000004 (floating-point imprecision
    !)

# Mathematical functions (requires import math)
import math
root = math.sqrt(2.0)      # 1.414...
sine = math.sin(math.pi)   # ~0 (numerical approximation)
```
Listing 38: Floating-point numbers in Python

Key Points:

- Python floats typically map to C's `double` (64-bit, double precision)
- Subject to floating-point imprecision (cannot exactly represent all decimals)
- Scientific notation uses `e` or `E` for powers of 10

Floating-Point Caution:

```python
# Dangerous: direct equality comparison
if 0.1 + 0.2 == 0.3:  # This may be False!
    print("Equal")

# Better: compare with tolerance
```

```python
tolerance = 1e-9
if abs((0.1 + 0.2) - 0.3) < tolerance:
    print("Approximately equal")

# Or use math.isclose() (Python 3.5+)
import math
if math.isclose(0.1 + 0.2, 0.3):
    print("Close enough")
```

Listing 39: Floating-point comparison pitfalls

11.4 Complex Numbers

Python has built-in support for complex numbers:

```python
# Creating complex numbers (j or J for imaginary unit)
z1 = 1 + 1j
z2 = 2 + 3J
z3 = complex(4, 5)  # Alternative: 4 + 5j

# Operations
sum = z1 + z2        # (3+4j)
product = z1 * z2    # (-1+5j)
power = 1j ** 2      # (-1+0j) = -1

# Accessing components
real_part = z1.real      # 1.0
imag_part = z1.imag      # 1.0
conjugate = z1.conjugate()  # (1-1j)

# Magnitude
import math
magnitude = abs(z1)  # sqrt(1^2 + 1^2) = sqrt(2)
```

Listing 40: Complex number operations

Mathematical Notation: Python uses `j` (or `J`) for $\sqrt{-1}$, following electrical engineering convention, rather than the mathematical `i`.

11.5 String Type

Strings represent text and are immutable sequences of characters:

Creating Strings:

```python
# Single quotes
s1 = 'string'

# Double quotes (equivalent to single quotes)
s2 = "string"

# Choosing quotes based on content
s3 = "Hello! I'm a string!"  # Contains single quote
s4 = 'Hi! I\'m "another" string!'  # Escaping quotes

# Triple quotes for multi-line strings
s5 = """String spanning
```

```python
multiple lines
with preserved formatting"""

s6 = '''Single quotes
also work for
multi-line strings'''
```

Listing 41: Various ways to create strings

String Operations:

```python
# Concatenation
greeting = "Hello" + " " + "World"  # "Hello World"

# Repetition
repeated = "Ha" * 3  # "HaHaHa"

# Length
length = len("hello")  # 5

# Indexing (zero-based)
first = "Python"[0]    # 'P'
last = "Python"[-1]    # 'n'

# Slicing
substring = "Python"[0:4]  # "Pyth"

# Methods
upper = "hello".upper()         # "HELLO"
lower = "HELLO".lower()         # "hello"
stripped = "  text  ".strip()   # "text"
replaced = "hello".replace("l", "L")  # "heLLo"
split = "a,b,c".split(",")      # ['a', 'b', 'c']
```

Listing 42: Common string operations

Escape Sequences:

```python
newline = "Line 1\nLine 2"      # Newline character
tab = "Column1\tColumn2"        # Tab character
backslash = "Path\\to\\file"    # Literal backslash
quote = "She said \"Hello\""    # Escaped quote
unicode = "\u03B1"              # Unicode character (  )

# Raw strings (no escape processing)
path = r"C:\new\folder"  # Backslashes treated literally
```

Listing 43: Common escape sequences in strings

String Immutability:

```python
s = "hello"
s[0] = "H"  # TypeError: 'str' object does not support item assignment

# Instead, create a new string
s = "H" + s[1:]  # "Hello"
```

Listing 44: Strings cannot be modified in-place

String Encoding: Python 3 strings are Unicode by default, supporting characters from any language:

```python
chinese = "      "
arabic = "          "
emoji = "          "
mixed = "Hello,       "  # Mix languages freely
```
Listing 45: Unicode support in Python 3 strings

12 Operators and Expressions

Python provides a rich set of operators for performing calculations and comparisons.

12.1 Arithmetic Operators

```python
# Basic operations
addition = 5 + 7            # 12
subtraction = 5 - 2         # 3
multiplication = 5 * 7      # 35

# Division always returns float
true_division = 5 / 2       # 2.5

# Floor division returns integer (rounds down)
floor_division = 5 // 2     # 2
negative_floor = -5 // 2    # -3 (rounds toward negative infinity)

# Modulo (remainder)
modulo = 5 % 2              # 1
modulo2 = 17 % 5            # 2

# Exponentiation (power)
power = 3 ** 4              # 81 (3^4)
square_root = 9 ** 0.5      # 3.0 (sqrt using fractional exponent)

# Complex number operations
complex_power = (1j) ** 2   # (-1+0j)
```
Listing 46: Arithmetic operations in Python

Operator Precedence: (highest to lowest)

1. Parentheses: `()`
2. Exponentiation: `**`
3. Unary plus/minus: `+x`, `-x`
4. Multiplication, division: `*`, `/`, `//`, `%`
5. Addition, subtraction: `+`, `-`

```python
result1 = 2 + 3 * 4      # 14 (not 20)
result2 = (2 + 3) * 4    # 20
result3 = 2 ** 3 ** 2    # 512 (2^(3^2), right-associative)
result4 = (2 ** 3) ** 2  # 64
```

Listing 47: Operator precedence examples

12.2 Using the Math Module

For more advanced mathematical operations, import the `math` module:

```python
import math

# Constants
pi = math.pi      # 3.141592653589793
e = math.e        # 2.718281828459045

# Roots and powers
sqrt_2 = math.sqrt(2.0)         # 1.4142135623730951
cube_root = 8 ** (1/3)          # 2.0 (or math.pow(8, 1/3))

# Trigonometric functions (angles in radians)
sine = math.sin(math.pi)        # ~0 (numerical approximation)
cosine = math.cos(0)            # 1.0
tangent = math.tan(math.pi/4)   # ~1.0

# Logarithms
natural_log = math.log(math.e)    # 1.0 (ln)
log_base_10 = math.log10(100)     # 2.0
log_base_2 = math.log2(8)         # 3.0

# Rounding
ceiling = math.ceil(4.3)    # 5
floor = math.floor(4.7)     # 4

# Absolute value and sign
absolute = math.fabs(-5.5)  # 5.5
absolute2 = abs(-5)         # 5 (built-in, works with integers too)
```

Listing 48: Mathematical functions from the math module

12.3 Random Number Generation

The `random` module provides functions for generating random numbers:

```python
import random

# Random float in [0.0, 1.0)
r1 = random.random()

# Random integer in range [a, b] (inclusive)
r2 = random.randint(1, 10)  # Integer from 1 to 10

# Random float in range [a, b)
r3 = random.uniform(0.0, 10.0)

```

```python
# Random choice from a sequence
color = random.choice(['red', 'green', 'blue'])

# Shuffle a list in-place
deck = list(range(52))
random.shuffle(deck)

# Random sample without replacement
sample = random.sample(range(100), 5)  # 5 unique numbers

# Set seed for reproducibility
random.seed(42)  # Same seed produces same sequence
```
Listing 49: Random number generation

Scope and Modules: When importing modules, access their contents using the dot operator:

```python
import math
import random

# Access with module.member syntax
result1 = math.sqrt(16)      # math is the module, sqrt is the
    function
result2 = random.random()    # random is the module, random is the
    function

# Alternative: import specific items
from math import sqrt, pi
result3 = sqrt(16)  # Can use directly without module prefix

# Import with alias
import numpy as np
arr = np.array([1, 2, 3])  # np is the alias for numpy
```
Listing 50: Module scope and the dot operator

13 Compound Data Types: Lists

Lists are one of Python's most versatile and commonly used data structures. They are ordered, mutable sequences that can contain elements of any type.

13.1 Creating and Accessing Lists

```python
# Creating lists with square brackets
x = [0, 1, 2, 3, 3]

# Zero-based indexing
first = x[0]    # 0
second = x[1]   # 1
third = x[2]    # 2

# Negative indexing (from the end)
last = x[-1]     # 3 (last element)
```

```python
second_last = x[-2]  # 3 (second-to-last)

# Length of list
size = len(x)    # 5

# Lists can contain mixed types
mixed = [0, 1, 2, 'three']  # integers and string
nested = [1, [2, 3], 4]     # list within list
```

Listing 51: Creating and accessing Python lists

Why Negative Indexing? Negative indices provide a convenient way to access elements from the end without knowing the list's length:

```python
numbers = [10, 20, 30, 40, 50]

# Without negative indexing
last = numbers[len(numbers) - 1]  # 50 (verbose)

# With negative indexing
last = numbers[-1]  # 50 (concise)
```

Listing 52: Comparing positive and negative indexing

13.2 Modifying Lists

Lists are mutable, meaning their contents can be changed:

```python
x = [0, 1, 2, 3, 4]

# Modify element
x[2] = 99  # x = [0, 1, 99, 3, 4]

# Insert at specific position
x.insert(0, -1)  # x = [-1, 0, 1, 99, 3, 4]

# Append to end
x.append(5)  # x = [-1, 0, 1, 99, 3, 4, 5]

# Remove by index (returns removed value)
removed = x.pop(3)  # removes 99, returns 99
# x = [-1, 0, 1, 3, 4, 5]

# Remove by value (first occurrence)
x.remove(3)  # x = [-1, 0, 1, 4, 5]

# Extend list with another list
x.extend([6, 7])  # x = [-1, 0, 1, 4, 5, 6, 7]

# Clear all elements
x.clear()  # x = []
```

Listing 53: Modifying list contents

13.3 List Concatenation and Repetition

```python
# Concatenation with +
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined = list1 + list2  # [1, 2, 3, 4, 5, 6]

# In-place concatenation with +=
x = [0, 1, 2, 'three']
x += [4, 5, 6]  # x = [0, 1, 2, 'three', 4, 5, 6]

# Repetition with *
repeated = [0] * 5  # [0, 0, 0, 0, 0]
pattern = [1, 2] * 3  # [1, 2, 1, 2, 1, 2]
```

Listing 54: List concatenation and repetition

13.4 List Slicing

Slicing extracts portions of a list, creating a new list:

Syntax: `list[start:stop:step]`

- `start`: Starting index (inclusive)
- `stop`: Ending index (exclusive - not included!)
- `step`: Step size (optional, default 1)

```python
x = [0, 1, 2, 'three', 4, 5, 6]

# Basic slicing [start:stop]
slice1 = x[1:4]    # [1, 2, 'three'] (indices 1, 2, 3)

# From start to position
slice2 = x[:3]     # [0, 1, 2] (omit start = begin from 0)

# From position to end
slice3 = x[1:]     # [1, 2, 'three', 4, 5, 6] (omit stop = until end)

# Slice from end
slice4 = x[:-1]    # [0, 1, 2, 'three', 4, 5] (all but last)
slice5 = x[-3:]    # [4, 5, 6] (last three)

# Slicing with step [start:stop:step]
slice6 = x[::2]    # [0, 2, 4, 6] (every second element)
slice7 = x[1::2]   # [1, 'three', 5] (every second, starting at 1)

# Negative step (reverse)
slice8 = x[::-1]   # [6, 5, 4, 'three', 2, 1, 0] (reversed)
slice9 = x[::-2]   # [6, 4, 2, 0] (reversed, every second)

# Complex slicing
slice10 = x[1:6:2]  # [1, 'three', 5] (from 1 to 5, step 2)
```

Listing 55: List slicing operations

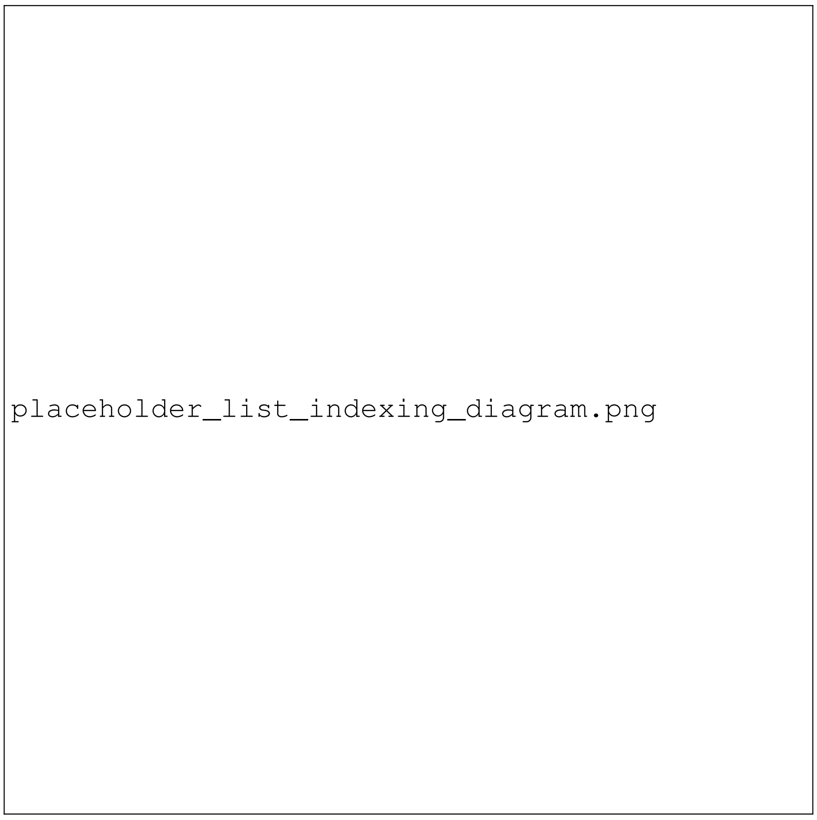

Figure 4: Visualization of list indexing: indices point between elements. For a list [0, 1, 2, 3], index positions are at 0, 1, 2, 3, 4, with the slice [1:3] capturing elements between positions 1 and 3 (elements 1 and 2).

Key Insight: The stop index is always exclusive. Think of slice indices as pointing between elements:

Guido van Rossum's Explanation: Python's creator explained the rationale for slice semantics on Stack Overflow (https://stackoverflow.com/a/21481885):

- Makes `len(s[a:b]) == b-a` when valid
- Easy to split at position: `s[:n] + s[n:] == s`
- Makes `s[:n]` and `s[n:]` complementary

13.5 Common List Operations

```python
numbers = [3, 1, 4, 1, 5, 9, 2, 6]

# Check membership
contains = 4 in numbers  # True
not_contains = 7 not in numbers  # True

```

```python
# Count occurrences
count = numbers.count(1)  # 2

# Find index of first occurrence
index = numbers.index(4)  # 2

# Sort list in-place
numbers.sort()  # numbers = [1, 1, 2, 3, 4, 5, 6, 9]

# Sort in reverse
numbers.sort(reverse=True)  # [9, 6, 5, 4, 3, 2, 1, 1]

# Return sorted copy (original unchanged)
original = [3, 1, 4]
sorted_copy = sorted(original)  # [1, 3, 4]
# original is still [3, 1, 4]

# Reverse list in-place
numbers.reverse()  # Reverses order

# Min, max, sum
minimum = min(numbers)  # Smallest value
maximum = max(numbers)  # Largest value
total = sum(numbers)    # Sum of all elements

# Copy a list
shallow_copy = numbers.copy()  # or numbers[:]
```

Listing 56: Additional list operations and methods

14 Tuples: Immutable Sequences

Tuples are similar to lists but with one crucial difference: they are immutable.

14.1 Creating and Using Tuples

```python
# Create tuple with parentheses
t = (1, 2, 3)

# Access elements (same as lists)
first = t[0]    # 1
last = t[-1]    # 3

# Slicing works too
slice_t = t[1:]  # (2, 3)

# Tuples can contain mixed types
mixed = (1, "two", 3.0, [4, 5])

# Single element tuple (note the comma!)
single = (1,)    # Tuple with one element
not_tuple = (1)  # This is just an integer with parentheses

# Tuple without parentheses (tuple packing)
```

```python
coords = 3, 4, 5  # Creates tuple (3, 4, 5)

# Empty tuple
empty = ()
empty2 = tuple()
```

Listing 57: Creating and using tuples

14.2 Tuple Immutability

The defining characteristic of tuples is that they cannot be modified after creation:

```python
t = (1, 2, 3)

# Attempting modification causes error
t[1] = 99  # TypeError: 'tuple' object does not support item
    assignment

# Also cannot append, insert, remove, etc.
t.append(4)  # AttributeError: 'tuple' object has no attribute 'append
    '

# Can create new tuple by concatenation
new_t = t + (4, 5)  # (1, 2, 3, 4, 5) (new tuple, original unchanged)
```

Listing 58: Tuples are immutable

Important Subtlety: While tuples themselves are immutable, they can contain mutable objects:

```python
t = (1, 2, [3, 4])

# Cannot replace the list
t[2] = [5, 6]  # TypeError

# But can modify the list in-place
t[2].append(5)  # Works! t = (1, 2, [3, 4, 5])
```

Listing 59: Tuples containing mutable objects

14.3 When to Use Tuples vs. Lists

Use Tuples When:

- Data shouldn't change (coordinates, RGB colors, database records)
- Need to use as dictionary keys (lists can't be keys)
- Want to convey intent that data is fixed
- Slight performance advantage (tuples are faster)
- Unpacking multiple values from functions

Use Lists When:

- Data will change over time
- Need to add/remove elements
- Need mutability for algorithms (sorting, filtering)

14.4 Tuple Packing and Unpacking

One of tuples' most powerful features is automatic packing and unpacking:

```python
# Tuple packing (creating tuple without parentheses)
coords = 3, 4, 5  # Creates (3, 4, 5)

# Tuple unpacking (multiple assignment)
x, y, z = coords  # x=3, y=4, z=5

# Swap variables elegantly
a = 1
b = 2
a, b = b, a  # Swaps values (no temp variable needed!)

# Function returning multiple values (actually returns tuple)
def get_dimensions():
    return 1920, 1080  # Returns tuple

width, height = get_dimensions()  # Unpack return value

# Unpacking with * (Python 3.0+)
first, *middle, last = [1, 2, 3, 4, 5]
# first = 1, middle = [2, 3, 4], last = 5
```

Listing 60: Tuple packing and unpacking

15 Dictionaries: Key-Value Mappings

Dictionaries are unordered collections of key-value pairs, similar to hash maps or associative arrays in other languages.

15.1 Creating Dictionaries

```python
# Method 1: dict() constructor with keyword arguments
d1 = dict(a=1, b=2, c='three')

# Method 2: Dictionary literal with braces
d2 = {'a': 1, 'b': 2, 'c': 'three'}

# Method 3: From list of tuples
d3 = dict([('a', 1), ('b', 2), ('c', 'three')])

# Method 4: Dictionary comprehension (covered later)
d4 = {x: x**2 for x in range(5)}  # {0:0, 1:1, 2:4, 3:9, 4:16}

# Empty dictionary
```

```python
empty = {}
empty2 = dict()

# Keys can be any hashable type
mixed_keys = {
    'string_key': 1,
    42: 'integer key',
    (1, 2): 'tuple key',
    frozenset([1, 2]): 'frozen set key'
}
```

Listing 61: Creating dictionaries

15.2 Accessing and Modifying Dictionaries

```python
person = {'name': 'Alice', 'age': 30, 'city': 'New York'}

# Access values by key
name = person['name']  # 'Alice'

# Safe access with get() (returns None if key missing)
country = person.get('country')  # None (no KeyError)
country = person.get('country', 'USA')  # 'USA' (default value)

# Check if key exists
has_name = 'name' in person  # True
has_country = 'country' in person  # False

# Add new key-value pair
person['email'] = 'alice@example.com'

# Modify existing value
person['age'] = 31

# Remove key-value pair
removed = person.pop('city')  # Returns 'New York', removes from dict
# Or use del
del person['email']

# Remove with default if key doesn't exist
value = person.pop('nonexistent', 'default')  # Returns 'default'

# Get keys, values, items
keys = person.keys()      # dict_keys(['name', 'age'])
values = person.values()  # dict_values(['Alice', 31])
items = person.items()    # dict_items([('name', 'Alice'), ('age', 31)
    ])

# Update dictionary with another dictionary
person.update({'city': 'Boston', 'country': 'USA'})

# Clear all entries
person.clear()
```

Listing 62: Dictionary operations

15.3 Iterating Over Dictionaries

```python
grades = {'Alice': 90, 'Bob': 85, 'Charlie': 92}

# Iterate over keys (default)
for student in grades:
    print(student)  # Alice, Bob, Charlie

# Explicitly iterate over keys
for student in grades.keys():
    print(student)

# Iterate over values
for grade in grades.values():
    print(grade)  # 90, 85, 92

# Iterate over key-value pairs (most common)
for student, grade in grades.items():
    print(f"{student}: {grade}")
    # Alice: 90
    # Bob: 85
    # Charlie: 92
```

Listing 63: Iterating over dictionaries

15.4 Dictionary Constraints

Key Requirements:

- **Hashable:** Keys must be immutable types (strings, numbers, tuples)
- **Unique:** Each key can appear only once
- **No Lists:** Lists cannot be keys (they're mutable)

```python
# Valid keys
valid = {
    'string': 1,
    42: 2,
    (1, 2): 3,
    3.14: 4
}

# Invalid keys
invalid = {
    [1, 2]: 'value'  # TypeError: unhashable type: 'list'
}

# But lists can be values
valid_values = {
    'key1': [1, 2, 3],
    'key2': {'nested': 'dict'}
}
```

Listing 64: Valid and invalid dictionary keys

15.5 Dictionary Ordering

Historical Note:

- **Python < 3.7:** Dictionaries had no guaranteed order
- **Python 3.7+:** Dictionaries maintain insertion order as a language feature

```python
# Python 3.7+: Order is preserved
d = {}
d['first'] = 1
d['second'] = 2
d['third'] = 3

for key in d:
    print(key)  # Always prints: first, second, third (in order)
```

Listing 65: Dictionary ordering in modern Python

16 Mutability: A Fundamental Concept

Understanding mutability is crucial for mastering Python's behavior, especially when working with functions and data structures.

16.1 Defining Mutability

Mutable Objects: Objects whose contents can be changed after creation:

- Lists: `[1, 2, 3]`
- Dictionaries: `{'a': 1}`
- Sets: `{1, 2, 3}`
- User-defined classes (usually)

Immutable Objects: Objects whose contents cannot be changed after creation:

- Numbers: `int`, `float`, `complex`
- Strings: `"hello"`
- Tuples: `(1, 2, 3)`
- Frozen sets: `frozenset({1, 2})`
- Booleans: `True`, `False`
- `None`

16.2 Demonstrating Mutability

```python
# Mutable: Lists
lst = [1, 2, 3]
lst[0] = 99        # Modifies the list in-place
lst.append(4)      # Adds to existing list
print(lst)         # [99, 2, 3, 4]

# Immutable: Strings
s = "hello"
s[0] = "H"         # TypeError: 'str' object does not support item
    assignment

# With strings, you must create new objects
s = "H" + s[1:]    # Creates new string "Hello"

# Immutable: Tuples
t = (1, 2, 3)
t[0] = 99          # TypeError: 'tuple' object does not support item
    assignment

# Immutable: Numbers
x = 5
x += 1             # Creates new integer object 6, rebinds x to it
```

Listing 66: Mutable vs immutable behavior

16.3 Why Mutability Matters

Performance:

- Mutable objects can be modified efficiently without creating copies
- Immutable objects require creating new objects for any "change"

Safety:

- Immutable objects are safer in concurrent programs (no race conditions)
- Immutable objects can be dictionary keys and set members

Semantics:

- Mutability affects how objects behave when passed to functions (covered in detail later)
- Mutability affects aliasing and copying behavior

16.4 The Identity vs Equality Distinction

Python distinguishes between object identity and value equality:

```python
# Equality: same value
a = [1, 2, 3]
b = [1, 2, 3]
print(a == b)     # True (same contents)
print(a is b)     # False (different objects)

# Identity: same object
c = a
print(a == c)     # True (same contents)
print(a is c)     # True (same object!)

# Modifying c affects a (they're the same object)
c.append(4)
print(a)          # [1, 2, 3, 4]

# Immutable objects: Python optimizes by reusing
x = 100
y = 100
print(x is y)     # True (Python reuses small integer objects)

# Strings too
s1 = "hello"
s2 = "hello"
print(s1 is s2)  # Often True (string interning)
```

Listing 67: Identity (is) vs equality (==)

Guidelines:

- Use == to compare values
- Use `is` to check if two names refer to the same object
- Use `is None` to check for None (by convention)

17 Control Flow: Conditional Statements

Control flow structures allow programs to make decisions and execute different code based on conditions.

17.1 The if Statement

Syntax:

```python
if condition:
    # Code executes if condition is True
    statement1
    statement2
```

Listing 68: Basic if statement syntax

Example:

```python
temperature = 25

if temperature > 30:
    print("It's hot today!")
    print("Drink plenty of water.")
```
Listing 69: Simple if statement

17.2 The if-else Statement

```python
x = -5

if x < 0:
    print("x is negative")
else:
    print("x is non-negative")
```
Listing 70: if-else statement

17.3 The if-elif-else Statement

For multiple conditions, use `elif` (else-if):

```python
score = 85

if score >= 90:
    grade = 'A'
    print("Excellent!")
elif score >= 80:
    grade = 'B'
    print("Good job!")
elif score >= 70:
    grade = 'C'
    print("Satisfactory")
elif score >= 60:
    grade = 'D'
    print("Need improvement")
else:
    grade = 'F'
    print("Failed")

print(f"Your grade: {grade}")
```
Listing 71: if-elif-else chain

Key Points:

- Only the first True condition's block executes
- `elif` and `else` are optional
- Can have multiple `elif` blocks
- No limit on number of conditions

17.4 Comparison Operators

```python
x = 10
y = 20

# Equality and inequality
x == y    # False (equal to)
x != y    # True (not equal to)

# Ordering
x < y     # True (less than)
x > y     # False (greater than)
x <= y    # True (less than or equal to)
x >= y    # False (greater than or equal to)

# Identity
x is y    # False (same object)
x is not y  # True (different objects)

# Membership (for sequences)
'a' in 'abc'        # True
'x' not in 'abc'    # True
2 in [1, 2, 3]      # True
```

Listing 72: Comparison operators

17.5 Logical Operators

Combine multiple conditions:

```python
age = 25
has_license = True

# and: Both must be True
if age >= 18 and has_license:
    print("Can drive")

# or: At least one must be True
is_weekend = True
is_holiday = False

if is_weekend or is_holiday:
    print("No work today!")

# not: Negates boolean
is_raining = False

if not is_raining:
    print("Can go outside")

# Complex conditions
temperature = 28
humidity = 70

if (temperature > 25 and humidity > 60) or temperature > 35:
    print("Uncomfortable weather")
```

Listing 73: Logical operators: and, or, not

17.6 Chained Comparisons

Python allows elegant chained comparisons:

```python
x = 15

# Instead of:
if x > 10 and x < 20:
    print("x is between 10 and 20")

# Python allows:
if 10 < x < 20:
    print("x is between 10 and 20")

# Multiple chains
year = 2024
month = 3
day = 15

if 1900 < year < 2100 and 1 <= month <= 12 and 1 <= day <= 31:
    print("Valid date range")

# Even works with different operators
a = 1
b = 2
c = 3

if a < b < c:    # True
    print("Ascending order")

if a < b == b < c:  # Can mix operators
    print("Complex chain")
```

Listing 74: Chained comparison operators

17.7 Indentation: Python's Block Structure

Critical Rule: Python uses indentation to define code blocks, not braces.

```python
if condition:
    # This is inside the if block (indented)
    statement1
    statement2
# This is outside the if block (not indented)
statement3
```

Listing 75: Indentation defines code structure

Standard Convention:

- Use 4 spaces per indentation level
- Never mix tabs and spaces

- Configure your editor to insert spaces when Tab is pressed

Nested Blocks:

```python
x = 15
y = 20

if x > 10:
    print("x is greater than 10")
    if y > 15:
        print("y is also greater than 15")
        if x < y:
            print("x is less than y")
    print("Still in outer if block")
print("Outside all if blocks")
```

Listing 76: Nested indentation

Why Indentation?

- **Enforces Readability:** Code is naturally formatted correctly
- **Reduces Clutter:** No need for braces
- **Prevents Errors:** Indentation errors are caught immediately
- **Consistency:** All Python code looks similar

18 Control Flow: Loops

Loops allow code to be executed repeatedly, either a specific number of times or while a condition remains true.

18.1 The while Loop

The `while` loop repeats as long as its condition is True:
Syntax:

```python
while condition:
    # Code block executes while condition is True
    statements
```

Listing 77: while loop syntax

Example:

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

print("Loop finished")
# Output:
# x = 0
# x = 1
# x = 2
# x = 3
# x = 4
# Loop finished
```

Listing 78: Basic while loop

18.2 Loop Control: break and continue

break: Immediately exits the loop **continue:** Skips the rest of the current iteration and continues with the next

```python
x = 0
while x < 10:
    if x == 2:
        x += 1
        continue  # Skip when x is 2

    if x == 7:
        break  # Exit loop when x is 7

    print(f"x = {x}")
    x += 1

# Output:
# x = 0
# x = 1
# (skips 2)
# x = 3
# x = 4
# x = 5
# x = 6
# (breaks at 7)
```

Listing 79: break and continue in loops

Common Use Cases:

```python
# User input validation
while True:
    password = input("Enter password: ")
    if len(password) >= 8:
        break
    print("Password too short. Try again.")

# Process until sentinel value
while True:
    value = input("Enter number (or 'quit'): ")
    if value == 'quit':
        break
    process(value)

# Countdown
count = 10
while count > 0:
    print(count)
    count -= 1
```

```python
print("Liftoff!")
```

Listing 80: Practical while loop examples

18.3 The for Loop: Iteration Over Sequences

Python's `for` loop is fundamentally different from C++'s. It's always a "for-each" or "range-based" loop that iterates over elements of a sequence.

Syntax:

```python
for item in sequence:
    # Code executes once for each item in sequence
    statements
```

Listing 81: for loop syntax

Iterating Over Lists:

```python
fruits = ['apple', 'banana', 'cherry']

for fruit in fruits:
    print(f"I like {fruit}")

# Output:
# I like apple
# I like banana
# I like cherry
```

Listing 82: Iterating over list elements

Iterating Over Strings:

```python
for letter in "Python":
    print(letter)

# Output:
# P
# y
# t
# h
# o
# n
```

Listing 83: Iterating over string characters

Iterating Over Mixed-Type Lists:

```python
mixed = [0, "a", 7, 1j]

for item in mixed:
    print(f"Item: {item}, Type: {type(item).__name__}")

# Output:
# Item: 0, Type: int
# Item: a, Type: str
# Item: 7, Type: int
# Item: 1j, Type: complex
```

Listing 84: Iterating over mixed-type sequences

18.4 The range() Function

For iterating over numbers, Python provides the `range()` function:

Syntax:

- `range(stop)`: Numbers from 0 to stop-1
- `range(start, stop)`: Numbers from start to stop-1
- `range(start, stop, step)`: Numbers from start to stop-1, incrementing by step

```python
# range(stop)
for i in range(5):
    print(i)  # 0, 1, 2, 3, 4

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

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

# Negative step (counting down)
for i in range(10, 0, -1):
    print(i)  # 10, 9, 8, ..., 1

# range() is memory efficient (doesn't create list)
big_range = range(1000000)  # Uses constant memory
```

Listing 85: Using range() for numeric iteration

18.5 Iterating with Indices

When you need both the index and the element:

Method 1: Using range() and len():

```python
fruits = ['apple', 'banana', 'cherry']

for i in range(len(fruits)):
    print(f"Index {i}: {fruits[i]}")

# Output:
# Index 0: apple
# Index 1: banana
# Index 2: cherry
```

Listing 86: Iterating with indices using range()

Method 2: Using enumerate() (Preferred):

```python
fruits = ['apple', 'banana', 'cherry']

for i, fruit in enumerate(fruits):
    print(f"Index {i}: {fruit}")
```

```python

# Start enumeration at different number
for i, fruit in enumerate(fruits, start=1):
    print(f"Item {i}: {fruit}")

# Output:
# Item 1: apple
# Item 2: banana
# Item 3: cherry
```

Listing 87: Iterating with enumerate() - Pythonic approach

Why enumerate() is Better:

- More Pythonic and readable
- Slightly more efficient
- Works with any iterable, not just sequences with `len()`
- Can specify starting index

18.6 Nested Loops

Loops can be nested to iterate over multi-dimensional data:

```python
# Multiplication table
for i in range(1, 6):
    for j in range(1, 6):
        product = i * j
        print(f"{i}    {j} = {product:2d}", end="  ")
    print()  # New line after each row

# Nested iteration over 2D list
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

for row in matrix:
    for element in row:
        print(element, end=" ")
    print()
```

Listing 88: Nested loop example

18.7 Loop else Clause

Python loops can have an `else` clause that executes if the loop completes normally (not via `break`):

```python
# Search for element
numbers = [1, 3, 5, 7, 9]
target = 6

```

```python
for num in numbers:
    if num == target:
        print(f"Found {target}!")
        break
else:
    # Executes only if loop didn't break
    print(f"{target} not found in list")

# Practical example: checking primality
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False  # Found divisor
    else:
        return True  # No divisor found
```

Listing 89: Loop else clause

19 List and Dictionary Comprehensions

Comprehensions provide a concise way to create lists and dictionaries from existing iterables.

19.1 List Comprehensions

Syntax:

```python
[expression for variable in iterable if condition]
```

Listing 90: List comprehension syntax

Basic Examples:

```python
# Square numbers
numbers = [1, 2, 3, 4, 5]
squares = [x**2 for x in numbers]
# [1, 4, 9, 16, 25]

# Add 1 to each element
incremented = [x + 1 for x in numbers]
# [2, 3, 4, 5, 6]

# Convert to strings
str_numbers = [str(x) for x in numbers]
# ['1', '2', '3', '4', '5']
```

Listing 91: Basic list comprehensions

With Conditions:

```python
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Only even numbers
evens = [x for x in numbers if x % 2 == 0]
```

```python
# [2, 4, 6, 8, 10]

# Squares of odd numbers
odd_squares = [x**2 for x in numbers if x % 2 != 0]
# [1, 9, 25, 49, 81]

# Skip specific value
x = [1, 2, 3]
result = [val + 1 for val in x if val != 2]
# [2, 4]
```

Listing 92: List comprehensions with filtering

Comparison with Traditional Approach:

```python
# Traditional approach
result = []
for x in numbers:
    if x % 2 == 0:
        result.append(x**2)

# List comprehension (equivalent, more concise)
result = [x**2 for x in numbers if x % 2 == 0]
```

Listing 93: Comprehension vs traditional loop

Complex Comprehensions:

```python
# Nested loops
matrix = [[i*j for j in range(1, 4)] for i in range(1, 4)]
# [[1, 2, 3], [2, 4, 6], [3, 6, 9]]

# Flattening nested lists
nested = [[1, 2], [3, 4], [5, 6]]
flat = [item for sublist in nested for item in sublist]
# [1, 2, 3, 4, 5, 6]

# With function calls
words = ['hello', 'world', 'python']
uppercase = [word.upper() for word in words]
# ['HELLO', 'WORLD', 'PYTHON']

# Conditional expression (ternary operator)
numbers = [1, 2, 3, 4, 5]
labels = ['even' if x % 2 == 0 else 'odd' for x in numbers]
# ['odd', 'even', 'odd', 'even', 'odd']
```

Listing 94: Advanced list comprehensions

19.2 Dictionary Comprehensions

Syntax:

```python
{key_expression: value_expression for variable in iterable if
    condition}
```

Listing 95: Dictionary comprehension syntax

Examples:

```python
# Square mapping
squares_dict = {x: x**2 for x in range(1, 6)}
# {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# From existing dictionary
prices = {'apple': 0.50, 'banana': 0.30, 'cherry': 0.75}

# Double all prices
doubled = {fruit: price * 2 for fruit, price in prices.items()}
# {'apple': 1.0, 'banana': 0.6, 'cherry': 1.5}

# Filter dictionary
expensive = {fruit: price for fruit, price in prices.items() if price
    > 0.40}
# {'apple': 0.5, 'cherry': 0.75}

# Transform keys and values
y = dict(x=1, y=2, z=3)
transformed = {key + '_a': val + 1 for key, val in y.items() if val !=
     3}
# {'x_a': 2, 'y_a': 3}

# Swap keys and values
inverted = {value: key for key, value in prices.items()}
# {0.5: 'apple', 0.3: 'banana', 0.75: 'cherry'}
```

Listing 96: Dictionary comprehensions

19.3 Set Comprehensions

Similar syntax for creating sets:

```python
# Set of squares (duplicates automatically removed)
numbers = [1, 2, 2, 3, 3, 3, 4, 5]
unique_squares = {x**2 for x in numbers}
# {1, 4, 9, 16, 25}

# Characters in string (no duplicates)
letters = {char for char in "hello"}
# {'h', 'e', 'l', 'o'}
```

Listing 97: Set comprehensions

19.4 Generator Expressions

Use parentheses instead of brackets for memory-efficient iteration:

```python
# List comprehension (creates entire list in memory)
squares_list = [x**2 for x in range(1000000)]  # Uses lots of memory

# Generator expression (computes on-the-fly)
squares_gen = (x**2 for x in range(1000000))   # Uses constant memory

# Use in iteration
for square in squares_gen:
    if square > 100:
        break

# Use with functions that accept iterables
total = sum(x**2 for x in range(100))  # No brackets needed
```
Listing 98: Generator expressions

19.5 When to Use Comprehensions

Use Comprehensions When:

- Creating a new list/dict from an existing iterable
- Logic is simple and fits on one line
- Filtering and transforming data
- Makes code more readable

Use Traditional Loops When:

- Logic is complex or multi-step
- Need to handle exceptions
- Performing actions (not creating new collections)
- Comprehension would be too long or nested

20 Functions

Functions are reusable blocks of code that perform specific tasks. They are fundamental to organizing code and avoiding repetition.

20.1 Defining Functions

Syntax:

```python
def function_name(parameters):
    """Docstring describing the function."""
    # Function body
    statements
    return value
```
Listing 99: Function definition syntax

Components:

1. **def keyword:** Begins function definition
2. **function_name:** Identifier following Python naming conventions
3. **parameters:** Comma-separated list of inputs (can be empty)

4. **Docstring:** First line after definition, documenting the function
5. **Function body:** Indented code block
6. **return statement:** Optional, sends value back to caller

Simple Example:

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

# Call the function
result = greet("Alice")
print(result)  # "Hello, Alice!"
```

Listing 100: A simple function example

Fibonacci Example:

```python
def fib(n):
    """Return the nth Fibonacci number.

    The Fibonacci sequence is: 0, 1, 1, 2, 3, 5, 8, 13, ...
    where each number is the sum of the two preceding ones.

    Args:
        n: The index of the Fibonacci number to compute (0-based)

    Returns:
        The nth Fibonacci number
    """
    a = 0
    b = 1
    for i in range(n):
        tmp = a
        a = a + b
        b = tmp
    return a

# Test the function
print(fib(0))   # 0
print(fib(1))   # 1
print(fib(5))   # 5
print(fib(10))  # 55
```

Listing 101: Fibonacci sequence function

20.2 Function Characteristics

No Type Declarations:

```python
# Python (no types)
def add(a, b):
    return a + b

# C++ (with types)
```

```
// int add(int a, int b) {
//     return a + b;
// }

# Python function works with any types that support +
print(add(5, 3))           # 8
print(add(2.5, 1.5))       # 4.0
print(add("Hello", "!"))   # "Hello!"
print(add([1], [2]))       # [1, 2]
```

Listing 102: Python functions don't declare types

Always Have Return Value:

```
def no_explicit_return():
    print("No return statement")

result = no_explicit_return()
print(result)  # None

# Explicit return of None
def explicit_none():
    return None

# Early return
def absolute(x):
    if x < 0:
        return -x
    return x
```

Listing 103: Functions always return something

Functions are Objects:

```
def square(x):
    return x ** 2

def cube(x):
    return x ** 3

# Store functions in list
operations = [square, cube]

# Call functions from list
for op in operations:
    print(op(3))  # 9, then 27

# Pass function as argument
def apply_twice(func, value):
    return func(func(value))

result = apply_twice(square, 2)  # square(square(2)) = 16

# Return function from function
def make_multiplier(n):
    def multiplier(x):
        return x * n
    return multiplier

```

```python
times_three = make_multiplier(3)
print(times_three(5))  # 15
```
Listing 104: Functions as first-class objects

No Function Overloading:

```python
# In C++, you can have:
// int add(int a, int b);
// double add(double a, double b);
// int add(int a, int b, int c);

# In Python, later definition replaces earlier
def add(a, b):
    return a + b

def add(a, b, c):  # This REPLACES the previous add()
    return a + b + c

result = add(1, 2)  # TypeError: missing required argument 'c'
```
Listing 105: Python doesn't support function overloading

Note: Operator overloading for classes works differently and is supported through magic methods (covered later).

20.3 Default Parameter Values

Functions can specify default values for parameters:

```python
def greet(name, greeting="Hello", punctuation="!"):
    """Greet someone with customizable message.

    Args:
        name: Person's name (required)
        greeting: Greeting word (default: "Hello")
        punctuation: Ending punctuation (default: "!")
    """
    return f"{greeting}, {name}{punctuation}"

# Use all defaults
print(greet("Alice"))  # "Hello, Alice!"

# Override greeting
print(greet("Bob", "Hi"))  # "Hi, Bob!"

# Override all
print(greet("Charlie", "Hey", "."))  # "Hey, Charlie."
```
Listing 106: Functions with default parameters

Rules for Default Parameters:

1. Parameters with defaults must come after those without defaults
2. Default values are evaluated once at function definition time
3. Mutable default arguments can cause surprising behavior (discussed later)

```python
# Valid
def func1(a, b, c=0, d=1):
    pass

# Invalid: non-default after default
def func2(a, b=0, c):  # SyntaxError
    pass

# Valid: all have defaults
def func3(a=1, b=2, c=3):
    pass
```

Listing 107: Valid and invalid parameter ordering

20.4 Keyword Arguments

Parameters can be passed by name (keyword arguments) rather than position:

```python
def describe_pet(animal, name, age=1):
    """Describe a pet."""
    return f"{name} is a {age}-year-old {animal}"

# Positional arguments
print(describe_pet("dog", "Rex", 3))

# Keyword arguments (any order)
print(describe_pet(name="Whiskers", animal="cat", age=2))

# Mix positional and keyword (positional first)
print(describe_pet("bird", name="Tweety"))

# Skip optional parameters
print(describe_pet("hamster", "Fuzzy"))  # Uses default age=1

# Can skip middle parameters with keywords
def config(host, port=8080, timeout=30, retry=3):
    pass

config("localhost", retry=5)  # Skips port and timeout
```

Listing 108: Using keyword arguments

Benefits of Keyword Arguments:

- Makes function calls self-documenting
- Allows skipping optional parameters
- Order-independent
- Reduces errors when functions have many parameters

20.5 Variable-Length Arguments

Functions can accept variable numbers of arguments:

***args (Variadic Positional Arguments):**

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

print(sum_all(1, 2, 3))         # 6
print(sum_all(1, 2, 3, 4, 5))   # 15
print(sum_all())                 # 0

# args is a tuple
def print_args(*args):
    print(f"Received {len(args)} arguments:")
    print(f"Type: {type(args)}")  # <class 'tuple'>
    for i, arg in enumerate(args):
        print(f"  arg[{i}] = {arg}")

print_args(1, "two", 3.0)
```

Listing 109: Using *args for variable positional arguments

****kwargs (Variadic Keyword Arguments):**

```python
def print_config(**kwargs):
    """Print configuration options."""
    print("Configuration:")
    for key, value in kwargs.items():
        print(f"  {key} = {value}")

print_config(host="localhost", port=8080, debug=True)

# kwargs is a dictionary
def collect_kwargs(**kwargs):
    print(f"Type: {type(kwargs)}")  # <class 'dict'>
    return kwargs

config = collect_kwargs(a=1, b=2, c=3)
print(config)  # {'a': 1, 'b': 2, 'c': 3}
```

Listing 110: Using **kwargs for variable keyword arguments

Complete Parameter Syntax:

```python
def complex_function(a, b=2, *args, c, d=4, e, **kwargs):
    """Demonstrate all parameter types.

    Args:
        a: Required positional argument
        b: Positional argument with default
        *args: Variable positional arguments
        c: Required keyword-only argument
        d: Keyword-only argument with default
        e: Required keyword-only argument
        **kwargs: Variable keyword arguments
    """
    print(f"a={a}, b={b}")
    print(f"args={args}")
    print(f"c={c}, d={d}, e={e}")
    print(f"kwargs={kwargs}")

# Example call
complex_function(1, 3, 5, 6, c=7, e=8, x=9, y=10)
# Output:
# a=1, b=3
# args=(5, 6)
# c=7, d=4, e=8
# kwargs={'x': 9, 'y': 10}
```

Listing 111: All parameter types together

Parameter Order Rules:

1. Regular positional parameters
2. Positional parameters with defaults
3. `*args` (variadic positional)
4. Keyword-only parameters (with or without defaults)
5. `**kwargs` (variadic keyword)

20.6 Unpacking Arguments

Use `*` and `**` to unpack sequences and dictionaries:

```python
def add_three(a, b, c):
    return a + b + c

# Unpack list with *
numbers = [1, 2, 3]
result = add_three(*numbers)  # Same as add_three(1, 2, 3)

# Unpack dictionary with **
config = {'a': 1, 'b': 2, 'c': 3}
result = add_three(**config)  # Same as add_three(a=1, b=2, c=3)

# Practical example: combining lists
list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined = [*list1, *list2]  # [1, 2, 3, 4, 5, 6]

# Combining dictionaries
dict1 = {'a': 1, 'b': 2}
dict2 = {'c': 3, 'd': 4}
combined = {**dict1, **dict2}  # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Listing 112: Unpacking arguments with * and **

20.7 Lambda Functions

Lambda functions are small anonymous functions:

Syntax:

```
lambda parameters: expression
```

Listing 113: Lambda function syntax

Examples:

```
# Simple lambda
square = lambda x: x ** 2
print(square(5))  # 25

# Equivalent to:
def square(x):
    return x ** 2

# Multiple parameters
add = lambda a, b: a + b
print(add(3, 5))  # 8

# Use in sorting
students = [('Alice', 85), ('Bob', 92), ('Charlie', 78)]
students.sort(key=lambda student: student[1])  # Sort by grade
print(students)  # [('Charlie', 78), ('Alice', 85), ('Bob', 92)]

# Use with map()
numbers = [1, 2, 3, 4, 5]
doubled = list(map(lambda x: x * 2, numbers))
print(doubled)  # [2, 4, 6, 8, 10]

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

# Passing function as argument
def apply(func, x):
    return func(x)

result = apply(lambda z: z ** 2, 5)  # 25
```

Listing 114: Lambda function examples

When to Use Lambda:

- Single, simple expressions
- As arguments to higher-order functions
- Short-lived functions not reused elsewhere

When NOT to Use Lambda:

- Complex logic requiring multiple statements
- Functions that will be reused (use `def` instead)
- When debugging (named functions are clearer)
- When documentation is important (lambdas can't have docstrings)

21 Coding Style and Conventions

Python places exceptional emphasis on code readability and consistency. Following established conventions makes code easier to read, maintain, and collaborate on.

21.1 PEP 8: The Python Style Guide

PEP 8 is the official style guide for Python code, and adherence to it is considered essential in the Python community.

URL: `https://www.python.org/dev/peps/pep-0008/`

Core Rules:

1. **Indentation:**
   - Use 4 spaces per indentation level
   - NEVER use tabs
   - NEVER mix tabs and spaces
2. **Line Length:**
   - Maximum 79 characters for code
   - Maximum 72 characters for docstrings/comments
3. **Blank Lines:**
   - Two blank lines between top-level functions and classes
   - One blank line between methods inside a class
   - Use blank lines sparingly to separate logical sections
4. **Imports:**
   - Each import on separate line
   - Imports at top of file
   - Group imports: standard library, third-party, local
   - Use absolute imports when possible
5. **Whitespace:**
   - No spaces inside parentheses, brackets, or braces
   - Space around binary operators
   - No space before function call parentheses
6. **Naming Conventions:**
   - `lowercase_with_underscores` for functions and variables
   - `UPPERCASE_WITH_UNDERSCORES` for constants

- `CapitalizedWords` for class names
- `_leading_underscore` for internal/private

7. **Documentation:**

- Use docstrings for all public modules, functions, classes, and methods
- Use inline comments sparingly
- Keep comments up-to-date

Example of Good Style:

```python
"""Module for geometric calculations.

This module provides functions for calculating areas and volumes
of basic geometric shapes.
"""

import math
import sys

# Constants
PI = 3.14159265359
GOLDEN_RATIO = 1.618


class Circle:
    """Represent a circle with radius and center coordinates."""

    def __init__(self, radius, center_x=0, center_y=0):
        """Initialize circle with radius and optional center.

        Args:
            radius: Circle radius (must be positive)
            center_x: X coordinate of center (default: 0)
            center_y: Y coordinate of center (default: 0)
        """
        self.radius = radius
        self.center_x = center_x
        self.center_y = center_y

    def area(self):
        """Calculate and return the circle's area."""
        return math.pi * self.radius ** 2


def calculate_rectangle_area(width, height):
    """Calculate the area of a rectangle.

    Args:
        width: Rectangle width
        height: Rectangle height

    Returns:
        The area as a float

    Raises:
        ValueError: If width or height is negative
    """
    if width < 0 or height < 0:
        raise ValueError("Dimensions must be non-negative")

    return width * height


def main():
    """Main program entry point."""
    circle = Circle(5.0)
    area = circle.area()
    print(f"Circle area: {area:.2f}")


if __name__ == "__main__":
    main()
```

Listing 115: Well-styled Python code following PEP 8

21.2 Enforcing Style with Tools

Rather than manually checking style, use automated tools:

Code Formatters (Automatic Formatting):

- **Black:** Opinionated formatter, very popular

```bash
pip install black
black myfile.py  # Formats file in-place

```

- **autopep8:** Formats to conform to PEP 8

```bash
pip install autopep8
autopep8 --in-place --aggressive myfile.py

```

- **yapf:** Google's formatter

Linters (Style Checking):

- **pylint:** Comprehensive style and error checker

```bash
pip install pylint
pylint myfile.py

```

- **flake8:** Combines pycodestyle, pyflakes, and McCabe complexity

```bash
pip install flake8
flake8 myfile.py

```

- **mypy:** Optional static type checker

```
pip install mypy
mypy myfile.py

```

Complete List: https://github.com/life4/awesome-python-code-formatters

21.3 Editor Configuration

Configure your editor to help maintain style:

- Set tab key to insert 4 spaces
- Enable visible whitespace characters
- Install Python syntax highlighting
- Enable automatic PEP 8 checking
- Configure automatic formatting on save
- Set line length marker at 79 characters

21.4 Interesting Style-Related Facts

Spaces vs. Tabs:

- Stack Overflow 2017: Developers using spaces earn more (https://stackoverflow.blog/2017/06/15/developers-use-spaces-make-money-use-tabs/)
- Correlation, not causation (likely reflects attention to detail)
- Python 3 explicitly disallows mixing

Coding Horror: "Death to the Space Infidels" - https://blog.codinghorror.com/death-to-the-space-infidels/

22 How Function Arguments Work: Pass by Assignment

Understanding how Python passes arguments to functions is crucial for avoiding common pitfalls and understanding unexpected behavior.

22.1 The Mystery

Consider these two functions:

```python
# Function 1: Incrementing an integer
def incr(x):
    x += 1

x = 0
incr(x)
```

```python
print(x)  # Prints: 0 (unchanged!)

# Function 2: Incrementing first element of list
def incr_first(x):
    x[0] += 1

x = [0, 1, 2]
incr_first(x)
print(x)  # Prints: [1, 1, 2] (changed!)
```

Listing 116: A puzzling behavior

The first looks like "pass by value" (value unchanged), the second like "pass by reference" (value changed). What's really happening?

22.2 Understanding Python's Object Model

Python uses a unique approach called **"pass by assignment"** or **"call by object reference"**:

Key Concepts:

1. **Variables are names, not containers**
   - Variables don't "contain" values; they're labels that "refer to" objects
   - Like name tags stuck on objects
2. **Objects live in a separate space**
   - All objects exist in memory (the "object space")
   - Variables are just names that point to these objects
3. **Assignment doesn't copy data**
   - Assignment makes a name refer to an object
   - Multiple names can refer to the same object

22.3 Step-by-Step: The Integer Example

Step 1: Initial assignment

```python
x = 0
```

Step 2: Function called

```python
incr(x)  # Call the function
```

Step 3: Inside function - increment

```python
def incr(x):
    x += 1  # This line executes
```

Because integers are IMMUTABLE, this creates a NEW object:

Step 4: Function returns

```python
print(x)  # Back in global scope
```

Result: Global x is unchanged because integers are immutable, and the local reassignment created a new object that the global name never saw.

Figure 5: Python's object model: Variables are labels (names) that bind to objects in memory. The same object can have multiple names referring to it. When you write x = 5, you're creating a label 'x' that points to an integer object 5 in memory.

22.4 Step-by-Step: The List Example

Step 1: Initial assignment

```python
x = [0, 1, 2]
```

Step 2: Function called

```python
incr_first(x)
```

Step 3: Inside function - modify

```python
def incr_first(x):
    x[0] += 1  # This line executes
```

Because lists are MUTABLE, this modifies the existing object:
Step 4: Function returns

```python
print(x)  # Back in global scope
```

Result: Global `x` reflects the change because lists are mutable, and both names pointed to the same list object that was modified in-place.

Figure 6: Global scope contains name ′x′ pointing to integer object 0 in object space.

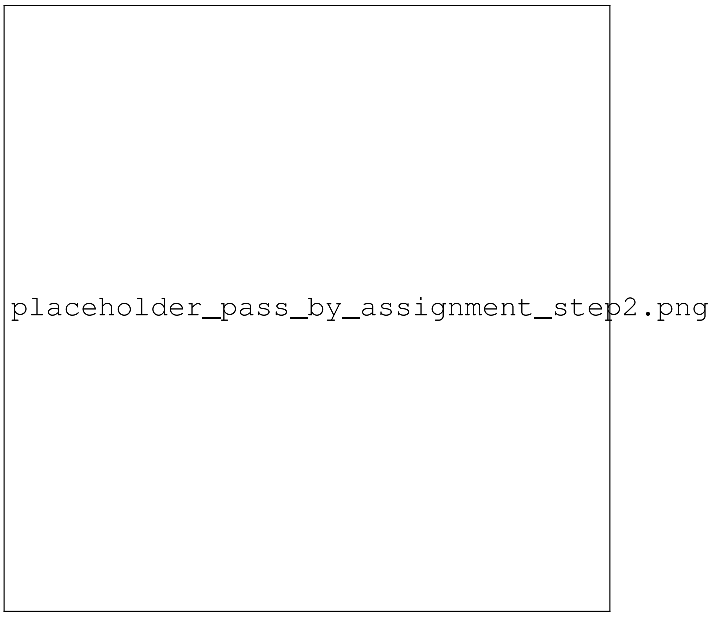

Figure 7: Local scope created with parameter ′x′ pointing to the same integer object 0. Both global and local ′x′ refer to the same object.

22.5 The Critical Distinction

The key is **mutability**:

Immutable Objects (int, float, str, tuple):

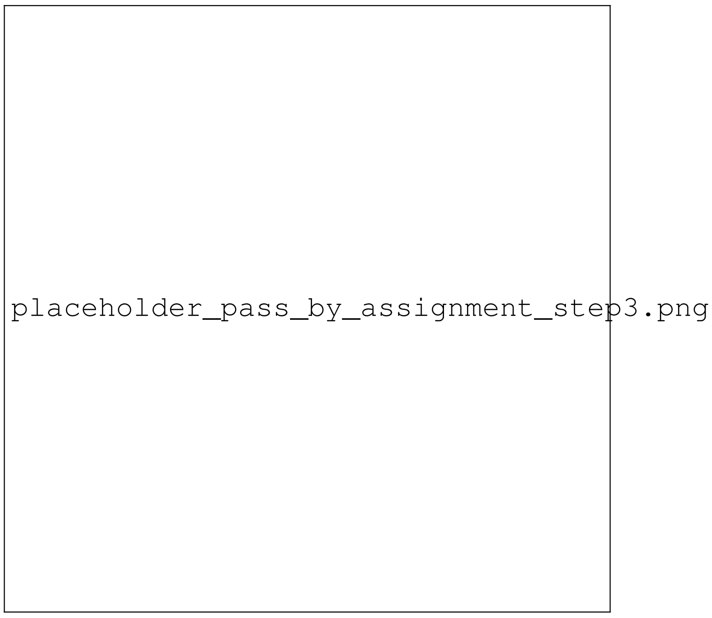

Figure 8: Integer 0 cannot be modified. Python creates new integer object 1, and local 'x' now points to it. Global 'x' still points to 0.

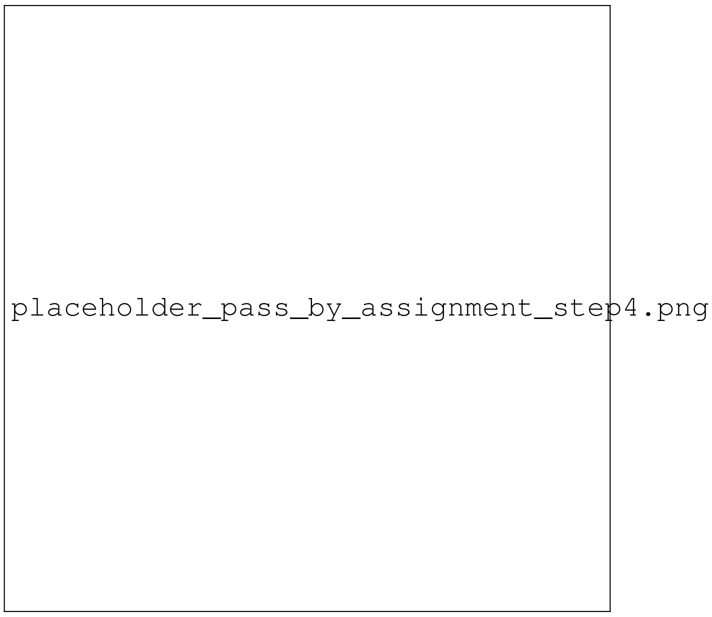

Figure 9: Local scope destroyed. Global 'x' still points to 0 (unchanged). The integer object 1 has no references and will be garbage collected.

- Any "modification" creates a new object
- Local reassignment doesn't affect global variables
- Behaves like "pass by value"

Figure 10: Global ′x′ points to a list object containing three elements.

Figure 11: Local ′x′ created, pointing to the SAME list object as global ′x′.

Mutable Objects (list, dict, set):

- Can be modified in-place
- Modifications are visible to all names referencing the object
- Behaves like "pass by reference"

Figure 12: The list object itself is modified in-place. The first element changes from 0 to 1. Both local and global 'x' point to this same modified list.

Figure 13: Local scope destroyed. Global 'x' points to the list, which was modified. Changes are visible!

22.6 Assignment and Aliasing

This same behavior applies to simple assignment:

```
# Lists (mutable)
x = [1, 2, 3]
```

```python
y = x  # y refers to the SAME list as x

y[0] = 99
print(x)  # [99, 2, 3] - x is affected!
print(y)  # [99, 2, 3]

# Integers (immutable)
a = 5
b = a  # b refers to the SAME integer as a

b += 1  # Creates NEW integer, rebinds b
print(a)  # 5 - a is unchanged
print(b)  # 6
```

Listing 117: Assignment creates references, not copies

22.7 Copying Objects

To create independent copies, use the `copy` module:

Shallow Copy:

```python
import copy

x = [1, 2, 3]
y = copy.copy(x)  # or y = x.copy() or y = x[:]

y[0] = 99
print(x)  # [1, 2, 3] - x unchanged
print(y)  # [99, 2, 3]
```

Listing 118: Shallow copying

Shallow Copy Limitation: Shallow copy only copies the top level:

```python
import copy

x = [1, 2, [3, 4]]
y = copy.copy(x)

y[2][0] = 99
print(x)  # [1, 2, [99, 4]] - inner list still shared!
print(y)  # [1, 2, [99, 4]]
```

Listing 119: Shallow copy limitation with nested lists

Deep Copy: Deep copy recursively copies all nested objects:

```python
import copy

x = [1, 2, [3, 4]]
y = copy.deepcopy(x)

y[2][0] = 99
print(x)  # [1, 2, [3, 4]] - completely independent!
print(y)  # [1, 2, [99, 4]]
```

Listing 120: Deep copying

The copy module documentation states: "The difference between shallow and deep copying is only relevant for compound objects (objects that contain other objects, like lists or class instances)."

22.8 Practical Implications

Avoiding Unintended Modifications:

```python
def process_list(data):
    """Process list without modifying original."""
    # Make a copy to avoid side effects
    data = data.copy()
    data.sort()
    # ... other operations
    return data

original = [3, 1, 4, 1, 5]
result = process_list(original)
print(original)  # [3, 1, 4, 1, 5] - unchanged
print(result)    # [1, 1, 3, 4, 5] - sorted
```

Listing 121: Defensive copying to avoid side effects

Mutable Default Arguments: A common pitfall:

```python
# WRONG: Mutable default argument
def add_item(item, items=[]):
    items.append(item)
    return items

print(add_item(1))  # [1]
print(add_item(2))  # [1, 2] - Unexpected!
print(add_item(3))  # [1, 2, 3] - Shared list!

# RIGHT: Use None as default
def add_item_correct(item, items=None):
    if items is None:
        items = []
    items.append(item)
    return items

print(add_item_correct(1))  # [1]
print(add_item_correct(2))  # [2] - Fresh list
print(add_item_correct(3))  # [3] - Each call independent
```

Listing 122: Mutable default arguments pitfall

23 Object-Oriented Programming: Classes

Python supports object-oriented programming (OOP), allowing you to define custom types with their own data and behavior.

23.1 Defining Classes

Syntax:

```python
class ClassName:
    """Class docstring."""

    def __init__(self, parameters):
        """Constructor/initializer."""
        self.attribute = value

    def method(self, parameters):
        """Instance method."""
        # Method body
```

Listing 123: Basic class definition syntax

Simple Example:

```python
class Point:
    """Represent a 2D point with x and y coordinates."""

    def __init__(self, x, y):
        """Initialize point with coordinates.

        Args:
            x: X coordinate
            y: Y coordinate
        """
        self.x = x
        self.y = y

# Create instances
p1 = Point(1, 2)
p2 = Point(3, 4)

# Access attributes
print(p1.x)  # 1
print(p1.y)  # 2

# Can add attributes dynamically
p1.z = 5  # Adds new attribute to p1 only
print(p1.z)  # 5
# print(p2.z)  # AttributeError - p2 doesn't have z
```

Listing 124: A simple Point class

23.2 The self Parameter

Critical Convention: The first parameter of instance methods is always `self`:

- Refers to the instance the method is called on
- Equivalent to C++'s implicit `this` pointer
- Name `self` is convention (could technically be anything, but ALWAYS use `self`)
- Must be explicit in Python (unlike C++)

```python
class Counter:
    def __init__(self):
        self.count = 0

    def increment(self):
        self.count += 1  # self refers to the instance

    def get_count(self):
        return self.count

c1 = Counter()
c2 = Counter()

c1.increment()
c1.increment()
c2.increment()

print(c1.get_count())  # 2
print(c2.get_count())  # 1 (separate instance)

# When you call c1.increment():
# Python actually calls Counter.increment(c1)
# self parameter receives c1
```

Listing 125: Understanding self

23.3 The Constructor: __init__

The `__init__` method is the constructor, called when creating new instances:

```python
class Rectangle:
    """Represent a rectangle."""

    def __init__(self, width, height):
        """Initialize rectangle with dimensions.

        Args:
            width: Rectangle width
            height: Rectangle height
        """
        self.width = width
        self.height = height

    def area(self):
        """Calculate and return area."""
        return self.width * self.height

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

# Create instance
rect = Rectangle(10, 5)

# Call methods
print(f"Area: {rect.area()}")          # 50
print(f"Perimeter: {rect.perimeter()}")  # 30
```

Listing 126: Constructor examples

Key Points:

- Name must be exactly `__init__` (double underscores)
- Called automatically when creating instance
- Doesn't return anything (implicitly returns `None`)
- Initializes instance attributes with `self.attribute = value`

23.4 Instance Methods

Methods are functions defined inside a class:

```python
class BankAccount:
    """Represent a simple bank account."""

    def __init__(self, owner, balance=0):
        """Initialize account.

        Args:
            owner: Account owner's name
            balance: Initial balance (default: 0)
        """
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        """Deposit money into account.

        Args:
            amount: Amount to deposit

        Raises:
            ValueError: If amount is negative
        """
        if amount < 0:
            raise ValueError("Cannot deposit negative amount")
        self.balance += amount

    def withdraw(self, amount):
        """Withdraw money from account.

        Args:
            amount: Amount to withdraw

        Returns:
            True if successful, False if insufficient funds
        """
        if amount > self.balance:
            return False
        self.balance -= amount
        return True

    def get_balance(self):
        """Return current balance."""
        return self.balance

# Usage
account = BankAccount("Alice", 1000)
account.deposit(500)
account.withdraw(200)
print(account.get_balance())  # 1300
```
Listing 127: Instance methods

23.5 Python vs. C++ Classes

Key Differences:

```python
# Python
class Point:
    def __init__(self, x, y):
        self.x = x  # Attributes created in __init__
        self.y = y

p = Point(1, 2)
p.z = 3  # Can add attributes dynamically!

# C++
// struct Point {
//     Point(double x, double y) : x(x), y(y) {}
//     double x, y;  // Members declared in class
// };
//
// Point p(1, 2);
// p.z = 3;  // ERROR: z not declared
```
Listing 128: Comparing Python and C++ classes

Comparison Table:

Feature	**Python**	**C++**
Member declaration	Implicitly in `__init__`	Explicitly in class body
self/this parameter	Explicit `self`	Implicit `this`
Adding attributes	Can add dynamically	Fixed at compile time
Privacy	Convention-based	Enforced by compiler
Constructor name	`__init__`	Class name
Type checking	Runtime	Compile time

Table 1: Python vs C++ class features

24 Privacy in Python Classes

Python's approach to privacy is fundamentally different from C++.

24.1 Convention-Based Privacy

Python has no enforced privacy mechanism. Instead, it relies on conventions:
The Convention:

- Names with leading underscore (`_name`) are considered "internal" or "private"
- This is a signal to other programmers, not a restriction
- Python programmers follow the philosophy: "We're all consenting adults here"

```python
class BankAccount:
    """A bank account with conventional privacy."""

    def __init__(self, owner, balance):
        """Initialize account."""
        self._owner = owner      # "Private" by convention
        self._balance = balance  # "Private" by convention

    def deposit(self, amount):
        """Public method to deposit money."""
        self._balance += amount

    def get_balance(self):
        """Public method to access balance."""
        return self._balance

# Usage
account = BankAccount("Alice", 1000)

# Proper usage (through public methods)
account.deposit(500)
print(account.get_balance())  # 1500

# Can still access "private" attributes (not recommended!)
print(account._balance)  # 1500 - works, but shouldn't do this
account._balance = 0     # Can modify directly (bad practice!)
```

Listing 129: Privacy by convention in Python

24.2 Python vs. C++ Privacy

```python
# Python
class Point:
    def __init__(self, x, y):
        self._x = x  # Convention: private
        self._y = y

    def get_x(self):
        return self._x

p = Point(1, 2)
print(p._x)  # Works, but violates convention
```

```

# C++
// class Point {
// public:
//     Point(double x, double y) : x(x), y(y) {}
//     double get_x() const { return x; }
// private:
//     double x, y;
// };
//
// Point p(1, 2);
// p.x;  // COMPILATION ERROR - truly private
```
Listing 130: Privacy comparison: Python vs C++

24.3 Name Mangling

Python provides a stronger privacy mechanism for class-internal names:

```python
class MyClass:
    def __init__(self):
        self.__private = "truly private"  # Double underscore
        self._internal = "semi-private"   # Single underscore

    def __private_method(self):
        """Private method."""
        return "secret"

obj = MyClass()

# Single underscore (accessible)
print(obj._internal)  # Works

# Double underscore (name mangled)
# print(obj.__private)  # AttributeError

# But can still access via mangled name (if you really want to)
print(obj._MyClass__private)  # Works (mangled to _ClassName__name)
```
Listing 131: Name mangling with double underscores

Name Mangling Rules:

- Names starting with __ (but not ending with __) are mangled
- Mangled to `_ClassName__name`
- Helps avoid name conflicts in inheritance
- Not true privacy, just harder to access accidentally

24.4 Why No True Privacy?

Python's philosophy emphasizes:

- **Trust:** Programmers are expected to respect conventions

- **Flexibility:** Can access internals for debugging or testing
- **Simplicity:** Less language complexity
- **Pragmatism:** Sometimes you need to bend the rules

Official Documentation: https://docs.python.org/3/tutorial/classes.html#private-variables

25 Class Variables and Static Data

Class variables are shared by all instances of a class:

25.1 Defining Class Variables

```python
class Point:
    """A 2D point class."""

    # Class variable (shared by all instances)
    dimensions = 2

    def __init__(self, x, y):
        """Initialize point."""
        # Instance variables (unique to each instance)
        self.x = x
        self.y = y

# Access class variable through class
print(Point.dimensions)  # 2

# Access through instances
p1 = Point(1, 2)
p2 = Point(3, 4)
print(p1.dimensions)  # 2
print(p2.dimensions)  # 2

# Modify class variable
Point.dimensions = 3
print(p1.dimensions)  # 3 (all instances see change)
print(p2.dimensions)  # 3
```

Listing 132: Class variables (static data)

25.2 Class Variables vs. Instance Variables

```python
class Counter:
    """Track instances and individual counts."""

    # Class variable - shared by all instances
    total_instances = 0

    def __init__(self):
        """Initialize counter."""
        # Increment class variable
        Counter.total_instances += 1

        # Instance variable - unique to this instance
        self.count = 0

    def increment(self):
        """Increment this counter."""
        self.count += 1

# Create instances
c1 = Counter()
c2 = Counter()
c3 = Counter()

print(Counter.total_instances)  # 3

c1.increment()
c1.increment()
c2.increment()

print(c1.count)  # 2 (instance-specific)
print(c2.count)  # 1 (instance-specific)
print(c3.count)  # 0 (instance-specific)
```

Listing 133: Distinguishing class and instance variables

Warning - Shadowing:

```python
class Point:
    dimensions = 2

p1 = Point()
p2 = Point()

# Modifying through instance creates instance variable!
p1.dimensions = 3  # Creates NEW instance variable

print(p1.dimensions)      # 3 (instance variable)
print(p2.dimensions)      # 2 (class variable)
print(Point.dimensions)   # 2 (class variable)

# To modify class variable, use class name
Point.dimensions = 3
print(p2.dimensions)  # 3 (now sees updated class variable)
```

Listing 134: Accidental shadowing of class variables

25.3 Comparing with C++

```python
# Python
class Point:
    dim = 2  # Class variable

    def __init__(self, x, y):
        self._x = x
        self._y = y
```

```

Point.dim  # Access class variable

# C++
// struct Point {
//     Point(double x, double y) : x(x), y(y) {}
//     double x, y;
//     static const int dim = 2;  // Static member
// };
//
// Point::dim;  // Access static member
```
Listing 135: Static members: Python vs C++

26 Magic Methods: Operator Overloading

Magic methods (also called "dunder methods" for double underscore) allow you to define how objects interact with Python's operators and built-in functions.

26.1 Understanding Magic Methods

Magic methods are special methods with names surrounded by double underscores that Python calls automatically in response to certain operations.

```python
class Point:
    """2D point with operator overloading."""

    def __init__(self, x, y):
        """Initialize point."""
        self._x = x
        self._y = y

    def __add__(self, other):
        """Define point + point."""
        return Point(self._x + other._x, self._y + other._y)

    def __str__(self):
        """Define string representation for print()."""
        return f"({self._x}, {self._y})"

    def __repr__(self):
        """Define official string representation."""
        return f"Point({self._x}, {self._y})"

# Usage
p1 = Point(1, 2)
p2 = Point(3, 4)

# Calls p1.__add__(p2)
p3 = p1 + p2
print(p3)  # Calls p3.__str__() -> "(4, 6)"

# In interpreter
# >>> p3
# Point(4, 6)  # Calls p3.__repr__()
```

Listing 136: Basic magic methods example

26.2 Common Magic Methods

Object Lifecycle:

```
class Resource:
    """Demonstrate lifecycle methods."""

    def __init__(self, name):
        """Constructor."""
        self.name = name
        print(f"Creating {self.name}")

    def __del__(self):
        """Destructor (finalizer)."""
        print(f"Destroying {self.name}")

# Create and destroy
r = Resource("MyResource")
# ... use resource ...
del r  # Explicitly delete (or wait for garbage collection)
```

Listing 137: Lifecycle magic methods

Arithmetic Operators:

```
class Vector:
    """Simple vector class."""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        """Vector addition: v1 + v2"""
        return Vector(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        """Vector subtraction: v1 - v2"""
        return Vector(self.x - other.x, self.y - other.y)

    def __mul__(self, scalar):
        """Scalar multiplication: v * scalar"""
        return Vector(self.x * scalar, self.y * scalar)

    def __truediv__(self, scalar):
        """Scalar division: v / scalar"""
        return Vector(self.x / scalar, self.y / scalar)

    def __floordiv__(self, scalar):
        """Floor division: v // scalar"""
        return Vector(self.x // scalar, self.y // scalar)

    def __neg__(self):
```

```python
        """Negation: -v"""
        return Vector(-self.x, -self.y)

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

# Usage
v1 = Vector(3, 4)
v2 = Vector(1, 2)

print(v1 + v2)    # Vector(4, 6)
print(v1 - v2)    # Vector(2, 2)
print(v1 * 2)     # Vector(6, 8)
print(v1 / 2)     # Vector(1.5, 2.0)
print(-v1)        # Vector(-3, -4)
```

Listing 138: Arithmetic operator overloading

Comparison Operators:

```python
class Point:
    """Point with comparison based on distance from origin."""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        """Calculate distance from origin."""
        return (self.x**2 + self.y**2)**0.5

    def __eq__(self, other):
        """Equality: p1 == p2"""
        return self.x == other.x and self.y == other.y

    def __ne__(self, other):
        """Inequality: p1 != p2"""
        return not self.__eq__(other)

    def __lt__(self, other):
        """Less than: p1 < p2 (by distance)"""
        return self.distance_from_origin() < other.
    distance_from_origin()

    def __le__(self, other):
        """Less than or equal: p1 <= p2"""
        return self.distance_from_origin() <= other.
    distance_from_origin()

    def __gt__(self, other):
        """Greater than: p1 > p2"""
        return self.distance_from_origin() > other.
    distance_from_origin()

    def __ge__(self, other):
        """Greater than or equal: p1 >= p2"""
        return self.distance_from_origin() >= other.
    distance_from_origin()
```

```python

# Usage
p1 = Point(1, 1)
p2 = Point(2, 2)

print(p1 == p2)  # False
print(p1 < p2)   # True (p1 closer to origin)
print(p1 <= p2)  # True
```
Listing 139: Comparison operator overloading

Container Emulation:

```python
class MyList:
    """Simple list-like container."""

    def __init__(self, items):
        self._items = list(items)

    def __len__(self):
        """Return length: len(obj)"""
        return len(self._items)

    def __getitem__(self, index):
        """Get item: obj[index]"""
        return self._items[index]

    def __setitem__(self, index, value):
        """Set item: obj[index] = value"""
        self._items[index] = value

    def __delitem__(self, index):
        """Delete item: del obj[index]"""
        del self._items[index]

    def __contains__(self, item):
        """Membership test: item in obj"""
        return item in self._items

    def __iter__(self):
        """Make iterable: for item in obj"""
        return iter(self._items)

# Usage
ml = MyList([1, 2, 3, 4, 5])

print(len(ml))      # 5 - calls __len__
print(ml[2])        # 3 - calls __getitem__
ml[2] = 99          # calls __setitem__
print(3 in ml)      # False - calls __contains__
print(99 in ml)     # True

for item in ml:     # calls __iter__
    print(item)
```
Listing 140: Container protocol magic methods

26.3 String Representation Methods

```python
class Point:
    """Point with both string representations."""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        """Informal string for end users (print, str)."""
        return f"Point at ({self.x}, {self.y})"

    def __repr__(self):
        """Official string for developers (repr, interactive)."""
        return f"Point({self.x}, {self.y})"

p = Point(3, 4)

print(str(p))    # "Point at (3, 4)" - uses __str__
print(repr(p))   # "Point(3, 4)" - uses __repr__
print(p)         # "Point at (3, 4)" - print uses __str__

# In interactive interpreter:
# >>> p
# Point(3, 4)  # Uses __repr__
```

Listing 141: String representation magic methods

Guidelines:

- `__str__`: Human-readable representation for end users
- `__repr__`: Unambiguous representation for developers (ideally, code that recreates the object)
- If only one is defined, `__repr__` is preferred as it's used as fallback

26.4 Complete Magic Methods Reference

Table of Common Magic Methods:
Complete Reference: https://docs.python.org/3/reference/datamodel.html

27 Class Documentation with Docstrings

Proper documentation is crucial for maintainable code. Python uses docstrings for documentation.

27.1 Class and Method Docstrings

Python Operation	Magic Method	C++ Equivalent
`obj = Class()`	`__init__`	Constructor
`del obj`	`__del__`	Destructor
`str(obj)`	`__str__`	N/A
`repr(obj)`	`__repr__`	N/A
`a + b`	`__add__`	`operator+`
`a - b`	`__sub__`	`operator-`
`a * b`	`__mul__`	`operator*`
`a / b`	`__truediv__`	`operator/`
`a // b`	`__floordiv__`	N/A
`a % b`	`__mod__`	`operator%`
`a ** b`	`__pow__`	N/A
`a == b`	`__eq__`	`operator==`
`a != b`	`__ne__`	`operator!=`
`a < b`	`__lt__`	`operator<`
`a <= b`	`__le__`	`operator<=`
`a > b`	`__gt__`	`operator>`
`a >= b`	`__ge__`	`operator>=`
`len(obj)`	`__len__`	N/A
`obj[key]`	`__getitem__`	`operator[]`
`obj[key] = val`	`__setitem__`	`operator[]`
`del obj[key]`	`__delitem__`	N/A
`item in obj`	`__contains__`	N/A
`for x in obj`	`__iter__`	N/A

Table 2: Common magic methods and their uses

```python
class Point:
    """A simple two-dimensional Cartesian coordinate point.

    This class represents a point in 2D space with x and y coordinates.
    It provides methods for common operations like scaling and computing
    distance.

    Attributes:
        _x: The x coordinate (private by convention)
        _y: The y coordinate (private by convention)

    Examples:
        >>> p1 = Point(3, 4)
        >>> p2 = Point(0, 0)
        >>> p1.distance_to(p2)
        5.0
    """

    # Class variable
    dimensions = 2

    def __init__(self, x, y):
        """Create a point from two Cartesian coordinates.

        Args:
            x (float): The x coordinate
            y (float): The y coordinate

        Raises:
            TypeError: If x or y are not numeric
        """
        self._x = float(x)
        self._y = float(y)

    def scale(self, sx=1, sy=1):
        """Scale the point by factors in x and y directions.

        Multiplies the x coordinate by sx and the y coordinate by sy.

        Args:
            sx (float): Scale factor for x direction (default: 1)
            sy (float): Scale factor for y direction (default: 1)

        Examples:
            >>> p = Point(2, 3)
            >>> p.scale(2, 3)
            >>> print(p)
            Point(4.0, 9.0)
        """
        self._x *= sx
        self._y *= sy

    def distance_to(self, other):
        """Calculate Euclidean distance to another point.

        Args:
            other (Point): The point to measure distance to

        Returns:
            float: The Euclidean distance between the two points

        Examples:
            >>> p1 = Point(0, 0)
            >>> p2 = Point(3, 4)
            >>> p1.distance_to(p2)
            5.0
        """
        dx = self._x - other._x
        dy = self._y - other._y
        return (dx**2 + dy**2)**0.5

    def __str__(self):
        """Return informal string representation."""
        return f"Point({self._x}, {self._y})"
```

Listing 142: Comprehensive class documentation

27.2 Docstring Conventions

PEP 257 - Docstring Conventions: https://www.python.org/dev/peps/pep-0257/

Key Rules:

- Use triple double quotes: `"""docstring"""`
- First line is a brief summary
- Blank line separates summary from detailed description
- Document all public modules, functions, classes, and methods
- Use imperative mood ("Return the..." not "Returns the...")

Popular Docstring Styles:

1. **NumPy Style:** Used extensively in scientific Python
2. **Google Style:** Clear and readable
3. **Sphinx/reStructuredText:** For generating documentation

Accessing Docstrings:

```python
# Get class docstring
print(Point.__doc__)

# Get method docstring
print(Point.scale.__doc__)

# Use help() function
help(Point)
help(Point.scale)

# In IPython
# Point?
# Point.scale?
```

Listing 143: Accessing documentation

28 Inheritance

Inheritance allows classes to derive behavior from parent classes while adding or overriding functionality.

28.1 Basic Inheritance

```python
class Animal:
    """Base class for animals."""

    def __init__(self, name, age):
        """Initialize animal.

        Args:
            name: Animal's name
            age: Animal's age
        """
        self.name = name
        self.age = age

    def speak(self):
        """Make the animal speak."""
        return "Some sound"

    def info(self):
        """Return animal information."""
        return f"{self.name} is {self.age} years old"


class Dog(Animal):
    """Dog class inheriting from Animal."""

    def __init__(self, name, age, breed):
        """Initialize dog.

        Args:
            name: Dog’s name
            age: Dog’s age
            breed: Dog’s breed
        """
        # Call parent constructor
        super().__init__(name, age)
        self.breed = breed

    def speak(self):
        """Override speak method."""
        return "Woof!"

    def fetch(self):
        """Dog-specific method."""
        return f"{self.name} is fetching the ball!"


class Cat(Animal):
    """Cat class inheriting from Animal."""

    def __init__(self, name, age, color):
        """Initialize cat."""
        super().__init__(name, age)
        self.color = color

    def speak(self):
        """Override speak method."""
        return "Meow!"

# Usage
dog = Dog("Rex", 3, "Labrador")
cat = Cat("Whiskers", 2, "Orange")

print(dog.name)        # Inherited attribute
print(dog.info())      # Inherited method
print(dog.speak())     # Overridden method: "Woof!"
print(dog.fetch())     # Dog-specific method

print(cat.speak())      # "Meow!"
print(cat.info())       # Inherited method
```
Listing 144: Basic class inheritance

28.2 The super() Function

Critical: In Python, the parent class constructor is NOT called automatically. You must explicitly call it using `super()`.

```python
class A:
    """Base class."""

    def __init__(self):
        self.a = 1
        print("A.__init__ called")

    def print_A(self):
        print(f"a = {self.a}")


class B(A):
    """Derived class."""

    def __init__(self):
        # MUST call parent constructor explicitly
        super().__init__()
        # Now self.a exists
        self.b = 2
        print("B.__init__ called")

    def print_B(self):
        print(f"b = {self.b}")


# Create instance
b = B()
# Output:
# A.__init__ called
# B.__init__ called

b.print_A()  # 1 (inherited method)
b.print_B()  # 2 (own method)
print(b.a)   # 1 (inherited attribute)
print(b.b)   # 2 (own attribute)
```
Listing 145: Using super() to call parent methods

What happens if you forget super()?

```python
class B_Wrong(A):
    def __init__(self):
        # Forgot super().__init__()!
        self.b = 2

b = B_Wrong()
```

```python
# b.print_A()  # AttributeError: 'B_Wrong' object has no attribute 'a'
print(b.b)     # 2 (own attribute exists)
```
Listing 146: Forgetting to call super()

28.3 Multiple Inheritance

Python supports multiple inheritance:

```python
class Flyer:
    """Mixin for flying ability."""

    def fly(self):
        return f"{self.name} is flying!"


class Swimmer:
    """Mixin for swimming ability."""

    def swim(self):
        return f"{self.name} is swimming!"


class Duck(Animal, Flyer, Swimmer):
    """Duck can fly and swim."""

    def __init__(self, name, age):
        super().__init__(name, age)

    def speak(self):
        return "Quack!"

# Usage
duck = Duck("Donald", 5)
print(duck.speak())  # "Quack!"
print(duck.fly())    # "Donald is flying!"
print(duck.swim())   # "Donald is swimming!"
print(duck.info())   # Inherited from Animal
```
Listing 147: Multiple inheritance

Method Resolution Order (MRO): Python uses the C3 linearization algorithm to determine method lookup order:

```python
# View MRO
print(Duck.__mro__)
# (<class 'Duck'>, <class 'Animal'>, <class 'Flyer'>,
#  <class 'Swimmer'>, <class 'object'>)

# Or use mro() method
print(Duck.mro())
```
Listing 148: Viewing method resolution order

29 Decorators

Decorators are a powerful Python feature that allow you to modify the behavior of functions or classes.

29.1 Understanding Decorators

A decorator is a function that takes another function as input and returns a modified version of it.

```python
def my_decorator(func):
    """A simple decorator."""
    def wrapper():
        print("Something before the function")
        func()
        print("Something after the function")
    return wrapper

# Apply decorator manually
def say_hello():
    print("Hello!")

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

# Or use @ syntax (preferred)
@my_decorator
def say_goodbye():
    print("Goodbye!")

say_goodbye()
# Output:
# Something before the function
# Goodbye!
# Something after the function
```

Listing 149: Basic decorator example

29.2 Decorators with Arguments

```python
def smart_decorator(func):
    """Decorator that preserves function arguments."""
    def wrapper(*args, **kwargs):
        print(f"Calling {func.__name__}")
        result = func(*args, **kwargs)
        print(f"{func.__name__} returned {result}")
        return result
    return wrapper

@smart_decorator
```

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

result = add(3, 5)
# Output:
# Calling add
# add returned 8
print(result)  # 8
```

Listing 150: Decorators handling function arguments

29.3 Practical Decorator Examples

Timing Decorator:

```python
import time

def timer(func):
    """Measure execution time of function."""
    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"

result = slow_function()
# Output: slow_function took 1.0001 seconds
```

Listing 151: Timing function execution

Memoization Decorator:

```python
def memoize(func):
    """Cache function results."""
    cache = {}
    def wrapper(*args):
        if args not in cache:
            cache[args] = func(*args)
        return cache[args]
    return wrapper

@memoize
def fibonacci(n):
    """Compute nth Fibonacci number."""
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

# Much faster with memoization
```

```python
print(fibonacci(100))  # Completes quickly!
```

Listing 152: Caching function results

30 Built-In Decorators for Classes

Python provides several built-in decorators specifically for use in classes.

30.1 @staticmethod

Static methods don't receive the instance (`self`) as the first argument:

```python
class MathUtils:
    """Utility class for math operations."""

    @staticmethod
    def add(a, b):
        """Add two numbers.

        No access to instance or class.
        """
        return a + b

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

# Call without creating instance
result = MathUtils.add(5, 3)  # 8

# Can also call on instance (but unusual)
utils = MathUtils()
result = utils.multiply(4, 7)  # 28
```

Listing 153: Using @staticmethod

Use Cases for @staticmethod:

- Utility functions related to the class
- Functions that don't need instance or class data
- Grouping related functions in a namespace

30.2 @classmethod

Class methods receive the class (not instance) as first argument:

```python
class Point:
    """Point with alternative constructors."""

    def __init__(self, x, y):
        self.x = x
        self.y = y

    @classmethod
    def from_tuple(cls, coords):
        """Create point from tuple.

        Args:
            coords: Tuple of (x, y)

        Returns:
            New Point instance
        """
        return cls(coords[0], coords[1])

    @classmethod
    def origin(cls):
        """Create point at origin."""
        return cls(0, 0)

    @classmethod
    def name(cls):
        """Return class name."""
        return cls.__name__

# Use alternative constructors
p1 = Point.from_tuple((3, 4))
p2 = Point.origin()

print(Point.name())  # "Point"
```

Listing 154: Using @classmethod

Convention: Use `cls` as the first parameter name for class methods (analogous to `self` for instance methods).

Use Cases for @classmethod:

- Alternative constructors (factory methods)
- Methods that work with class variables
- Methods that need access to the class itself

30.3 @property

Properties allow you to define methods that are accessed like attributes:

```python
class Temperature:
    """Temperature with Celsius and Fahrenheit conversions."""

    def __init__(self, celsius):
        self._celsius = celsius

    @property
    def celsius(self):
        """Get temperature in Celsius."""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        """Set temperature in Celsius with validation."""
        if value < -273.15:
            raise ValueError("Temperature below absolute zero!")
        self._celsius = value

    @property
    def fahrenheit(self):
        """Get temperature in Fahrenheit (computed)."""
        return self._celsius * 9/5 + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        """Set temperature via Fahrenheit."""
        self.celsius = (value - 32) * 5/9

# Usage - looks like attribute access
temp = Temperature(25)
print(temp.celsius)       # 25 (calls getter)
print(temp.fahrenheit)    # 77.0 (computed)

temp.celsius = 30         # Calls setter
print(temp.fahrenheit)    # 86.0

temp.fahrenheit = 32      # Sets via Fahrenheit
print(temp.celsius)       # 0.0

# temp.celsius = -300  # Raises ValueError
```

Listing 155: Using @property for computed attributes

More Complex Example:

```python
class Point:
    """Point with properties."""

    def __init__(self, x, y):
        self._x = x
        self._y = y

    @property
    def x(self):
        """The point's x coordinate."""
        return self._x

    @x.setter
    def x(self, value):
        """Set x coordinate."""
        self._x = value

    @property
    def y(self):
        """The point's y coordinate."""
        return self._y

    @y.setter
    def y(self, value):
        """Set y coordinate."""
        self._y = value

    @property
    def magnitude(self):
        """Distance from origin (read-only)."""
        return (self._x**2 + self._y**2)**0.5

p = Point(3, 4)
print(p.x)          # 3
print(p.magnitude)  # 5.0

p.x = 5
print(p.magnitude)  # 6.4031...

# p.magnitude = 10  # AttributeError - no setter defined
```

Listing 156: Property with validation and computation

Benefits of @property:

- Provides computed attributes
- Adds validation to attribute setting
- Can make read-only attributes
- Allows changing implementation without breaking interface
- Cleaner syntax than getter/setter methods

31 Modules and Code Organization

As programs grow, organizing code into modules becomes essential.

31.1 What is a Module?

A **module** is simply a Python file containing definitions (functions, classes, variables). The filename becomes the module name.

Example Module (mymath.py):

```python
"""Mathematical utility functions.

This module provides basic mathematical operations.
"""

PI = 3.14159265359

def square(x):
    """Return the square of x."""
    return x ** 2

def cube(x):
    """Return the cube of x."""
    return x ** 3

```

```python
class Circle:
    """Represent a circle."""

    def __init__(self, radius):
        self.radius = radius

    def area(self):
        """Calculate area."""
        return PI * self.radius ** 2

if __name__ == "__main__":
    # This code only runs if module is executed directly
    print("Testing mymath module")
    print(f"square(5) = {square(5)}")
    print(f"cube(3) = {cube(3)}")
```

Listing 157: A simple module - mymath.py

31.2 Importing Modules

Method 1: Import Entire Module

```python
import mymath

result = mymath.square(5)
circle = mymath.Circle(10)
print(mymath.PI)
```

Listing 158: Importing entire module

Method 2: Import with Alias

```python
import mymath as mm

result = mm.square(5)
print(mm.PI)
```

Listing 159: Importing with alias

Method 3: Import Specific Items

```python
from mymath import square, Circle

result = square(5)  # Use directly without module name
circle = Circle(10)
# print(PI)  # NameError - PI not imported
```

Listing 160: Importing specific items

Method 4: Import Everything (Not Recommended)

```python
from mymath import *

result = square(5)
circle = Circle(10)
print(PI)

# Warning: Can cause name conflicts and makes code less clear
```

Listing 161: Importing everything - use with caution

31.3 Module Search Path

Python searches for modules in this order:

1. Current directory
2. Directories in `PYTHONPATH` environment variable
3. Standard library directories
4. Site-packages directories (third-party packages)

```python
import sys
print(sys.path)
# ['', '/usr/lib/python3.10', '/usr/lib/python3.10/site-packages',
    ...]
```

Listing 162: Viewing the module search path

31.4 Packages

A **package** is a directory containing Python modules and a special `__init__.py` file:

Package Structure:

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

Importing from Packages:

```python
# Import module from package
import mypackage.module1

# Import specific item from module in package
from mypackage.module1 import my_function

# Import from subpackage
from mypackage.subpackage.module3 import MyClass
```

Listing 163: Importing from packages

31.5 The __init__.py File

The `__init__.py` file:

- Marks a directory as a Python package
- Can be empty

- Can contain package initialization code
- Can define `__all__` to control `from package import *`

Example __init__.py:

```python
"""MyPackage - A collection of utilities.

This package provides mathematical and string utilities.
"""

# Import commonly used items to package level
from .module1 import important_function
from .module2 import ImportantClass

# Define what gets imported with 'from mypackage import *'
__all__ = ['important_function', 'ImportantClass']

# Package version
__version__ = '1.0.0'
```

Listing 164: Package initialization file

32 Executing Modules as Scripts

Python modules can be both imported and executed as standalone scripts.

32.1 The __name__ Variable

Python sets the `__name__` variable differently depending on how the file is used:

- When **executed directly**: `__name__ == "__main__"`
- When **imported**: `__name__ == module_name`

```python
"""whats_the_point.py - Demonstrate Point class usage."""

import point  # Import our point module
import sys

def demonstrate_points():
    """Demonstrate point operations."""
    p1 = point.Point(1., 2.)
    p2 = point.Point(0., 7.)
    p3 = p1 + p2
    print(f"p1 + p2 = {p3}")

if __name__ == "__main__":
    # This code only runs when executed directly
    print("Running whats_the_point.py as script")

    # Can access command-line arguments
    if len(sys.argv) > 1:
        y_coord = int(sys.argv[1])
        p1 = point.Point(1., y_coord)
    else:
        p1 = point.Point(1., 2.)

    p2 = point.Point(0., 7.)
    p3 = p1 + p2
    print(f"Result: {p3}")
else:
    # This code runs when imported
    print("whats_the_point module imported")
```

Listing 165: Dual-use module with $_{n}ame_{check}$

Running as Script:

```
$ python whats_the_point.py
Running whats_the_point.py as script
Result: (1.0, 9.0)

$ python whats_the_point.py 5
Running whats_the_point.py as script
Result: (1.0, 12.0)
```

Importing as Module:

```
import whats_the_point
# Output: whats_the_point module imported

whats_the_point.demonstrate_points()
# Uses the functions, but __main__ block didn't run
```

Listing 166: When imported, main block doesn't run

32.2 Command-Line Arguments

Access command-line arguments via `sys.argv`:

```
import sys

def main():
    """Main program function."""
    print(f"Script name: {sys.argv[0]}")
    print(f"Number of arguments: {len(sys.argv) - 1}")

    if len(sys.argv) < 2:
        print("Usage: python script.py <name> [age]")
        sys.exit(1)

    name = sys.argv[1]
    age = int(sys.argv[2]) if len(sys.argv) > 2 else None

    print(f"Hello, {name}!")
    if age:
        print(f"You are {age} years old.")

if __name__ == "__main__":
```

```python
    main()
```
Listing 167: Handling command-line arguments

Better Approach: argparse Module
For complex command-line interfaces, use the `argparse` module:

```python
import argparse

def main():
    """Main program with argument parsing."""
    parser = argparse.ArgumentParser(
        description='Process some integers.'
    )

    parser.add_argument('name', help='Your name')
    parser.add_argument('-a', '--age', type=int, help='Your age')
    parser.add_argument('-v', '--verbose', action='store_true',
                        help='Verbose output')

    args = parser.parse_args()

    print(f"Hello, {args.name}!")
    if args.age:
        print(f"You are {args.age} years old.")
    if args.verbose:
        print("Verbose mode enabled")

if __name__ == "__main__":
    main()
```
Listing 168: Using argparse for command-line arguments

Usage:

```
$ python script.py Alice --age 30 --verbose
Hello, Alice!
You are 30 years old.
Verbose mode enabled

$ python script.py --help
usage: script.py [-h] [-a AGE] [-v] name
...
```

Reference: https://docs.python.org/3/library/argparse.html

33 String Formatting

Python provides multiple methods for formatting strings, with f-strings being the modern recommended approach.

33.1 F-Strings (Format String Literals)

Introduced in Python 3.6, f-strings are the most readable and performant method:

```python
import math

# Basic usage
name = "Alice"
age = 30
print(f"My name is {name} and I am {age} years old")

# Expressions inside braces
x = 10
y = 20
print(f"{x} + {y} = {x + y}")  # "10 + 20 = 30"

# Format specifications
pi = math.pi
print(f"Pi = {pi:.2f}")       # "Pi = 3.14"
print(f"Pi = {pi:10.4f}")     # "Pi =     3.1416" (10 chars wide)

# Padding with zeros
number = 42
print(f"{number:05d}")        # "00042"

# Alignment
text = "hello"
print(f"{text:<10}")          # "hello     " (left-aligned)
print(f"{text:>10}")          # "     hello" (right-aligned)
print(f"{text:^10}")          # "  hello   " (centered)

# Number formatting
value = 1234567.89
print(f"{value:,.2f}")        # "1,234,567.89" (thousands separator)
print(f"{value:e}")           # "1.234568e+06" (scientific notation)

# Percentage
fraction = 0.85
print(f"{fraction:.1%}")      # "85.0%"

# Debug feature (Python 3.8+)
x = 42
print(f"{x = }")              # "x = 42"
print(f"{x = :5.2f}")         # "x = 42.00"
```

Listing 169: F-string formatting examples

33.2 Format String Method

The `.format()` method (Python 2.6+):

```python
import math

# Positional arguments
print("{} + {} = {}".format(10, 20, 30))  # "10 + 20 = 30"

# Named arguments
print("{name} is {age} years old".format(name="Bob", age=25))

# Indexed arguments
```

```python
print("{0} {1} {0}".format("hello", "world"))  # "hello world hello"

# Format specifications
print("{:.2f}".format(math.pi))    # "3.14"
print("{:05d}".format(42))         # "00042"

# Number formatting
print("{:,.2f}".format(1234567.89))  # "1,234,567.89"
```

Listing 170: String format() method

33.3 Old-Style Formatting (printf-style)

C-style formatting (legacy, still works):
Basic usage print("
Format specifications print("
Multiple values print("Name:
Padding print("

33.4 Format Specification Mini-Language

General Format: `{value:fill align sign width .precision type}`
Components:

- **fill:** Character to use for padding (default: space)
- **align:** < (left), > (right), ^ (center), = (after sign)
- **sign:** + (always), − (negatives only), `(space)` (space for positive)
- **width:** Minimum field width
- **precision:** Number of decimal places (for floats) or max length (for strings)
- **type:** `d` (int), `f` (float), `e` (scientific), `s` (string), etc.

```python
# Different types
print(f"{42:d}")        # Integer: "42"
print(f"{42:b}")        # Binary: "101010"
print(f"{42:x}")        # Hexadecimal: "2a"
print(f"{42:o}")        # Octal: "52"

# Floating point
print(f"{3.14159:f}")    # Fixed point: "3.141590"
print(f"{3.14159:.2f}") # Two decimals: "3.14"
print(f"{3.14159:e}")    # Scientific: "3.141590e+00"

# Alignment and padding
print(f"{'test':*<10}") # "test******"
print(f"{'test':*>10}") # "******test"
print(f"{'test':*^10}") # "***test***"

# Numbers with signs
```

```python
print(f"{42:+d}")        # "+42"
print(f"{-42:+d}")       # "-42"
print(f"{42: d}")        # " 42" (space for positive)
```

Listing 171: Format specification examples

Complete Reference:

- https://docs.python.org/3/tutorial/inputoutput.html#fancier-output-formatting
- https://docs.python.org/3/library/string.html#formatspec

34 File Input/Output

Working with files is essential for reading data, saving results, and persisting program state.

34.1 Opening Files

Basic Syntax:

```python
# Open file
f = open("filename.txt", "r")  # 'r' for read mode
# ... use file ...
f.close()  # Always close when done!
```

Listing 172: Opening files with open()

File Modes:

- `'r'`: Read (default) - file must exist
- `'w'`: Write - creates new file or overwrites existing
- `'a'`: Append - adds to end of file
- `'x'`: Exclusive creation - fails if file exists
- `'r+'`: Read and write
- `'rb'`: Read binary
- `'wb'`: Write binary
- `'ab'`: Append binary

34.2 Writing to Files

```python
# Write mode (overwrites existing file)
f = open("output.txt", "w")
f.write("Hello, World!\n")
f.write("Second line\n")
f.close()

# Append mode (adds to existing file)
f = open("output.txt", "a")
f.write("Third line\n")
f.close()

# Writing multiple lines
lines = ["Line 1\n", "Line 2\n", "Line 3\n"]
f = open("output.txt", "w")
f.writelines(lines)
f.close()
```

Listing 173: Writing to files

34.3 Reading from Files

```python
# Read entire file
f = open("input.txt", "r")
content = f.read()
f.close()
print(content)

# Read line by line
f = open("input.txt", "r")
for line in f:
    print(line.strip())  # strip() removes trailing newline
f.close()

# Read one line
f = open("input.txt", "r")
first_line = f.readline()
second_line = f.readline()
f.close()

# Read all lines into list
f = open("input.txt", "r")
lines = f.readlines()
f.close()
print(lines)  # List of strings, each ending with \n
```

Listing 174: Reading from files

34.4 Context Managers: The with Statement

Best Practice: Always use the `with` statement for file operations:

```python
# File automatically closed, even if exception occurs
with open("data.txt", "r") as f:
    content = f.read()
    # Process content
# File is automatically closed here

# Multiple files
with open("input.txt", "r") as infile, \
     open("output.txt", "w") as outfile:
    for line in infile:
        outfile.write(line.upper())
# Both files automatically closed

# Writing
with open("numbers.txt", "w") as f:
    for i in range(10):
        f.write(f"{i}\n")
```

Listing 175: Using with statement for file handling

Why Use with?

- Automatically closes file, even if exception occurs
- Cleaner code (no explicit close needed)
- Prevents resource leaks
- Pythonic and recommended approach

34.5 Complete File I/O Examples

Reading and Processing CSV:

```python
# Simple CSV processing
with open("data.csv", "r") as f:
    for line in f:
        fields = line.strip().split(",")
        # Process fields
        print(fields)

# Better: use csv module
import csv

with open("data.csv", "r") as f:
    reader = csv.reader(f)
    headers = next(reader)  # First row
    for row in reader:
        print(dict(zip(headers, row)))
```

Listing 176: Processing CSV files

Reading and Writing JSON:

```python
import json

# Write JSON
data = {
    "name": "Alice",
    "age": 30,
    "scores": [85, 90, 95]
}

with open("data.json", "w") as f:
    json.dump(data, f, indent=2)

# Read JSON
with open("data.json", "r") as f:
    loaded_data = json.load(f)
    print(loaded_data["name"])
```
Listing 177: JSON file operations

Binary Files:

```python
# Write binary
data = b"\x00\x01\x02\x03"
with open("data.bin", "wb") as f:
    f.write(data)

# Read binary
with open("data.bin", "rb") as f:
    data = f.read()
    print(data)  # b'\x00\x01\x02\x03'
```
Listing 178: Binary file operations

34.6 File Path Operations

Use `pathlib` for modern path handling:

```python
from pathlib import Path

# Create path object
path = Path("data") / "files" / "input.txt"

# Check if file exists
if path.exists():
    print("File exists")

# Get file information
print(path.name)        # "input.txt"
print(path.stem)        # "input"
print(path.suffix)      # ".txt"
print(path.parent)      # Path("data/files")

# Read/write with pathlib
content = path.read_text()
path.write_text("New content")

# Iterate over directory
data_dir = Path("data")
for file in data_dir.glob("*.txt"):
    print(file)
```
Listing 179: Using pathlib for file paths

35 Exception Handling

Exceptions provide a mechanism for handling errors gracefully rather than crashing the program.

35.1 Understanding Exceptions

The Concept:

1. Function encounters error or exceptional condition
2. Cannot handle it locally
3. **Raises** (throws) an exception
4. Calling code can **catch** (except) the exception
5. If not caught, program terminates with traceback

35.2 Basic Exception Handling

```python
try:
    # Code that might raise exception
    result = 10 / 0
except ZeroDivisionError:
    # Handle specific exception
    print("Cannot divide by zero!")
    result = None

print(f"Result: {result}")
```

Listing 180: Basic try-except structure

35.3 Catching Multiple Exceptions

```python
def safe_divide(a, b):
    """Divide a by b with error handling."""
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        print("Error: Division by zero")
        return None
    except TypeError:
        print("Error: Invalid types for division")
        return None

# Test
print(safe_divide(10, 2))       # 5.0
print(safe_divide(10, 0))       # Error message, returns None
print(safe_divide(10, "2"))     # Error message, returns None

# Multiple exceptions in one except clause
```

```python
def process_value(x):
    try:
        return int(x) * 2
    except (ValueError, TypeError):
        print("Error: Cannot convert to integer")
        return None
```
Listing 181: Handling multiple exception types

35.4 Complete Exception Handling Structure

```python
def read_number_from_file(filename):
    """Read number from file with complete error handling."""
    result = None

    try:
        # Code that might raise exception
        with open(filename, "r") as f:
            content = f.read().strip()
            result = int(content)

    except FileNotFoundError:
        # Specific exception
        print(f"File {filename} not found")

    except ValueError:
        # Another specific exception
        print("File doesn't contain a valid number")

    except Exception as e:
        # Catch any other exception
        print(f"Unexpected error: {e}")

    else:
        # Executes if NO exception occurred
        print("Successfully read number")

    finally:
        # ALWAYS executes (cleanup code)
        print("Finished processing file")

    return result

# Test
number = read_number_from_file("data.txt")
```
Listing 182: Full try-except-else-finally structure

Execution Flow:

- `try`: Code that might raise exception
- `except`: Handles specific exception types
- `else`: Runs only if NO exception occurred
- `finally`: ALWAYS runs (for cleanup), even if exception occurs

35.5 Accessing Exception Information

```python
try:
    x = int("not a number")
except ValueError as e:
    # Access exception object
    print(f"Error occurred: {e}")
    print(f"Exception type: {type(e).__name__}")

# Getting full traceback
import traceback

try:
    result = 1 / 0
except ZeroDivisionError:
    # Print full traceback
    traceback.print_exc()
```

Listing 183: Getting exception details

35.6 Raising Exceptions

Functions can raise exceptions:

```python
def validate_age(age):
    """Validate age is positive."""
    if age < 0:
        raise ValueError("Age cannot be negative")
    if age > 150:
        raise ValueError("Age is unrealistically high")
    return True

try:
    validate_age(-5)
except ValueError as e:
    print(f"Validation error: {e}")

# Re-raising exceptions
def process_file(filename):
    """Process file with logging."""
    try:
        with open(filename) as f:
            return f.read()
    except FileNotFoundError:
        print(f"Logging: {filename} not found")
        raise  # Re-raise the same exception
```

Listing 184: Raising exceptions

35.7 Common Built-In Exceptions

- `Exception`: Base class for all exceptions
- `ValueError`: Invalid value (e.g., `int("abc")`)

- `TypeError`: Wrong type (e.g., `"text" + 5`)
- `KeyError`: Dictionary key not found
- `IndexError`: List index out of range
- `FileNotFoundError`: File doesn't exist
- `ZeroDivisionError`: Division by zero
- `AttributeError`: Attribute doesn't exist
- `ImportError`: Module not found
- `RuntimeError`: Generic runtime error

35.8 Custom Exceptions

```python
class InsufficientFundsError(Exception):
    """Raised when withdrawal exceeds balance."""
    pass

class BankAccount:
    """Bank account with custom exceptions."""

    def __init__(self, balance=0):
        self.balance = balance

    def withdraw(self, amount):
        """Withdraw money, raise exception if insufficient funds."""
        if amount > self.balance:
            raise InsufficientFundsError(
                f"Cannot withdraw {amount}, balance is {self.balance}"
            )
        self.balance -= amount

# Usage
account = BankAccount(100)

try:
    account.withdraw(150)
except InsufficientFundsError as e:
    print(f"Transaction failed: {e}")
```

Listing 185: Creating custom exception classes

Complete Reference: https://docs.python.org/3/library/exceptions.html

36 Python Conventions and Best Practices

Following Python conventions ensures code is readable, maintainable, and Pythonic.

36.1 Naming Conventions

Variables and Functions:

```python
# Variables: lowercase with underscores
user_name = "Alice"
total_count = 42
max_value = 100

# Functions: lowercase with underscores
def calculate_average(numbers):
    return sum(numbers) / len(numbers)

def get_user_input():
    return input("Enter value: ")
```

Listing 186: Variable and function naming

Constants:

```python
# Constants: UPPERCASE with underscores
PI = 3.14159265359
MAX_CONNECTIONS = 100
DEFAULT_TIMEOUT = 30
```

Listing 187: Constant naming

Classes:

```python
# Classes: CapitalizedWords (PascalCase)
class BankAccount:
    pass

class UserProfile:
    pass

class HTTPConnection:
    pass
```

Listing 188: Class naming

Private/Internal:

```python
class MyClass:
    def __init__(self):
        self._internal = "semi-private"    # Single underscore
        self.__private = "name mangled"     # Double underscore

    def _internal_method(self):
        """Internal method, not part of public API."""
        pass
```

Listing 189: Private naming convention

36.2 Important Naming Conventions

Standard Parameter Names:

- `self`: First parameter of instance methods

- `cls`: First parameter of class methods
- `args`: Variable positional arguments
- `kwargs`: Variable keyword arguments

```python
class Example:
    def instance_method(self, value):
        """self refers to instance."""
        pass

    @classmethod
    def class_method(cls, value):
        """cls refers to class."""
        pass

    def flexible_method(self, *args, **kwargs):
        """Accept variable arguments."""
        pass
```

Listing 190: Standard parameter naming

36.3 The Zen of Python

Python's design philosophy, accessible via `import this`:

```python
import this
# Output:
# Beautiful is better than ugly.
# Explicit is better than implicit.
# Simple is better than complex.
# Complex is better than complicated.
# Flat is better than nested.
# Sparse is better than dense.
# Readability counts.
# Special cases aren't special enough to break the rules.
# Although practicality beats purity.
# Errors should never pass silently.
# Unless explicitly silenced.
# In the face of ambiguity, refuse the temptation to guess.
# There should be one-- and preferably only one --obvious way to do it
    .
# Although that way may not be obvious at first unless you're Dutch.
# Now is better than never.
# Although never is often better than *right* now.
# If the implementation is hard to explain, it's a bad idea.
# If the implementation is easy to explain, it may be a good idea.
# Namespaces are one honking great idea -- let's do more of those!
```

Listing 191: The Zen of Python

36.4 PEP 8 Checklist

Critical Rules:

1. Use 4 spaces per indentation level
2. NEVER use tabs
3. Maximum line length: 79 characters
4. Use blank lines to separate functions and classes
5. Use docstrings for all public modules, functions, classes, methods
6. Use spaces around operators and after commas
7. Name classes with CapitalizedWords
8. Name functions and variables with lowercase_with_underscores
9. Always use `self` for first method argument
10. Always use `cls` for first class method argument
11. Use UTF-8 encoding (default in Python 3)

36.5 Docstring Conventions - PEP 257

PEP 257: https://www.python.org/dev/peps/pep-0257/

```python
def complex_function(arg1, arg2, option=None):
    """Do something complex with arguments.

    This is a more detailed explanation of what the function does.
    It can span multiple paragraphs if needed.

    Args:
        arg1 (int): First argument description
        arg2 (str): Second argument description
        option (bool, optional): Optional parameter. Defaults to None.

    Returns:
        dict: Description of return value

    Raises:
        ValueError: If arg1 is negative
        TypeError: If arg2 is not a string

    Examples:
        >>> complex_function(5, "test")
        {'result': 'processed'}
    """
    if arg1 < 0:
        raise ValueError("arg1 must be non-negative")
    # Implementation...
```

Listing 192: Proper docstring formatting

36.6 Style Checking with pylint

pylint is a comprehensive code quality checker:

```
# Install
pip install pylint

# Check a file
pylint myfile.py

# Output includes:
# - Code style violations
# - Potential errors
# - Code smell detection
# - Complexity metrics
# - Overall score

# Generate config file
pylint --generate-rcfile > .pylintrc

# Disable specific warnings
# pylint: disable=line-too-long
```

Listing 193: Using pylint

36.7 Complementary Style Guides

Beyond PEP 8, organizations maintain their own guides:

Google Python Style Guide:

- https://google.github.io/styleguide/pyguide.html
- Used across Google projects
- Extends PEP 8 with additional conventions

NumPy Style Guide:

- https://numpydoc.readthedocs.io/
- Docstring conventions for scientific computing
- Used by NumPy, SciPy, and many scientific packages

37 Additional Resources and Further Learning

Python has extensive documentation and community resources for continued learning.

37.1 Official Python Documentation

Python Homepage:

- https://www.python.org/
- Downloads, news, community information

Python Tutorial:

- https://docs.python.org/3/tutorial/index.html
- Official beginner's guide
- Many examples in this document come from this tutorial
- Comprehensive introduction to Python concepts

Python Library Reference:

- https://docs.python.org/3/library/index.html
- Complete documentation for standard library
- Essential reference for built-in modules
- Includes usage examples

Python Language Reference:

- https://docs.python.org/3/reference/index.html
- Detailed language specification
- Explains how Python works internally
- For advanced users wanting deep understanding

37.2 Third-Party Packages

Python Package Index (PyPI):

- https://pypi.org/
- Over 500,000 packages
- Searchable repository
- Installation via pip

Key Scientific Packages:

- NumPy: https://numpy.org/doc/
- SciPy: https://docs.scipy.org/
- Matplotlib: https://matplotlib.org/
- pandas: https://pandas.pydata.org/docs/
- scikit-learn: https://scikit-learn.org/

37.3 Recommended Learning Path

For Beginners:

1. Start with official Python tutorial
2. Practice with simple programs
3. Learn one module at a time from standard library
4. Read and understand others' code
5. Contribute to open source projects

For Scientific Computing:

1. Master Python fundamentals
2. Learn NumPy thoroughly
3. Explore SciPy for specific needs
4. Master Matplotlib for visualization
5. Learn pandas for data analysis
6. Explore domain-specific packages

37.4 Community Resources

Online Communities:

- Stack Overflow: `https://stackoverflow.com/questions/tagged/python`
- Reddit: `https://www.reddit.com/r/Python/`
- Python Discord servers
- Local Python user groups

Learning Platforms:

- Real Python: `https://realpython.com/`
- Python Tutor (visualizes code execution): `https://pythontutor.com/`
- Codecademy, Coursera, edX courses

Books:

- "Python Crash Course" by Eric Matthes
- "Fluent Python" by Luciano Ramalho
- "Effective Python" by Brett Slatkin
- "Python Cookbook" by David Beazley

38 Summary and Conclusion

This comprehensive guide has covered the fundamental concepts of Python programming, from basic syntax to advanced object-oriented programming features.

38.1 Key Concepts Covered

Language Fundamentals:

- Interpreted execution model
- Dynamic typing system
- Python 2 vs Python 3
- Installation across platforms

Basic Syntax:

- Data types: integers, floats, strings, booleans, complex numbers
- Operators and expressions
- Comments and documentation
- Indentation-based structure

Data Structures:

- Lists: mutable sequences
- Tuples: immutable sequences
- Dictionaries: key-value mappings
- Sets: unordered unique collections
- List and dictionary comprehensions

Control Flow:

- Conditional statements (if-elif-else)
- Loops (while, for)
- Loop control (break, continue)
- Chained comparisons

Functions:

- Function definition and calling
- Parameters: positional, keyword, default, variable-length

- Return values
- Lambda functions
- First-class functions

Object-Oriented Programming:

- Class definition and instantiation
- Instance and class variables
- Methods and properties
- Inheritance and super()
- Magic methods
- Decorators

Advanced Topics:

- Pass by assignment
- Mutability vs immutability
- Shallow and deep copying
- Module system
- Exception handling
- File I/O
- String formatting

Best Practices:

- PEP 8 style guide
- Docstring conventions
- Code organization
- Use of tools (formatters, linters)

38.2 Python's Philosophy

Python emphasizes:

- **Readability:** Code should be easy to read and understand
- **Simplicity:** Simple solutions are preferred over complex ones
- **Explicitness:** Explicit is better than implicit
- **Practicality:** Pragmatism over purity
- **Community:** Strong emphasis on conventions and consistency

38.3 Next Steps

To continue your Python journey:

1. **Practice regularly:** Write code every day
2. **Read others' code:** Learn from open source projects
3. **Build projects:** Apply concepts to real problems
4. **Contribute:** Join open source communities
5. **Specialize:** Dive deep into areas of interest (web, data science, ML, etc.)
6. **Stay updated:** Follow Python Enhancement Proposals (PEPs)
7. **Use the documentation:** Make the official docs your primary reference

38.4 Final Thoughts

Python is a powerful, versatile language that has become indispensable in scientific computing, data analysis, web development, automation, and many other domains. Its clear syntax, extensive libraries, and supportive community make it an excellent choice for both beginners and experienced programmers.

The key to mastering Python is consistent practice and engagement with the community. Start with small programs, gradually tackle more complex projects, and don't hesitate to consult the excellent documentation and community resources available.

Remember: Python's strength lies not just in its syntax, but in its philosophy of writing clear, readable code that others (including your future self) can understand and maintain. Follow the conventions, embrace the Zen of Python, and you'll find yourself writing not just functional code, but truly Pythonic code.

Happy coding!