

Python Course

Advanced Python Course

Samuele Carli

CNR-ISTC, Entersys

Bio: Computer scientist with broad interests in physics, mathematics, electronics, engineering, neurosciences, and business management.

Research @ CNR-ISTC: - Computational brain models of neurological diseases - Numerical simulation of monoamine interactions - ODE-based modeling of neural systems

Business @ Entersys s.r.l.: - Founder & CTO - Software development, automation & AI solutions - Product: Zelo ERP

Publication: - *"Simulating the Brain: A Four-Step Method using ODEs and Python"* (Springer, 2025)

Contacts: carlisamuele@csspace.net - www.csspace.net

Section 1

Introduction to Python

Subsection 1

Python Philosophy

Python Philosophy - The Zen of Python

Pythonic Rationale:

- PEP 20: 19 guiding principles for Python's design
- "Readability counts" - code is read more often than written
- "Simple is better than complex" - clarity over cleverness
- "There should be one- and preferably only one -obvious way to do it"

Python Code:

```
import this # Displays the Zen of Python
```

Historical Context:

- Written by Tim Peters in 1999, incorporated into Python 2.1
- Embodies Python's design philosophy: practicality beats purity
- Influences all Python style guides (PEP 8, Google Style Guide)

Python Philosophy - Readability First

Pythonic Rationale:

- Code is read far more often than it is written
- Clean syntax reduces cognitive load
- Whitespace as structure enforces consistent formatting
- Explicit is better than implicit: clear code over clever tricks

Python Code:

```
# Python: Readable and explicit
def calculate_average(numbers):
    if not numbers:
        return 0.0
    return sum(numbers) / len(numbers)

result = calculate_average([1, 2, 3, 4, 5]) # 3.0
```

Python Philosophy - Readability First (continued)

C Comparison:

```
// C: More verbose, manual memory concerns
double calculate_average(int* numbers, int count) {
    if (count == 0) return 0.0;
    double sum = 0.0;
    for (int i = 0; i < count; i++) {
        sum += numbers[i];
    }
    return sum / count;
}
```

Rust Comparison:

```
// Rust: Safety at cost of explicit types
fn calculate_average(numbers: &[i32]) -> f64 {
    if numbers.is_empty() { return 0.0; }
    let sum: i32 = numbers.iter().sum();
    sum as f64 / numbers.len() as f64
}
```

Computer Science Theory:

- Cognitive load: readable code reduces mental burden
- Whitespace significance: enforces consistent style
- Expressiveness vs verbosity trade-off

Python Philosophy - Batteries Included

Pythonic Rationale:

- “Batteries included” - rich standard library for common tasks
- File I/O, networking, data structures, text processing built-in
- No external dependencies for basic functionality
- Consistent API design across modules

Python Code:

```
import json, os, itertools, collections
```

```
# File I/O
```

```
with open('data.json') as f:  
    data = json.load(f)
```

```
# Iteration tools
```

```
for a, b in itertools.combinations([1, 2, 3], 2):  
    print(a, b)
```

```
# Advanced data structures
```

```
counter = collections.Counter('mississippi')  
# Counter({'i': 4, 's': 4, 'p': 2})
```

Python Philosophy - Batteries Included (continued)

C Comparison:

```
// C: No built-in JSON, must use external library  
#include <cjson/cJSON.h> // External dependency  
// Manual parsing, memory management required  
cJSON* root = cJSON_Parse(json_string);  
// ... cleanup required  
cJSON_Delete(root);
```

Rust Comparison:

```
// Rust: Most functionality requires external crates  
use serde_json; // Must add to Cargo.toml  
use std::collections::HashMap; // HashMap in std, but limited  
// Counter requires external crate: use indexmap or custom impl
```

Computer Science Theory:

- Standard library design: breadth vs depth trade-off
- Dependency management: batteries-included vs micro-libraries
- API consistency across modules

Subsection 2

Python for Science

Python for Science - Why Python?

Pythonic Rationale:

- Clean syntax lets scientists focus on algorithms, not language details
- Interactive development with Jupyter notebooks: explore, visualize, iterate
- Rich ecosystem: NumPy, SciPy, Pandas, Matplotlib cover 90% of scientific needs
- Bridge to high-performance code: C, Fortran, CUDA integration

Python Code:

```
# Solve differential equation in 5 lines
```

```
from scipy.integrate import solve_ivp  
import numpy as np
```

```
def pendulum(t, y): return [y[1], -9.81 * np.sin(y[0])]  
solution = solve_ivp(pendulum, [0, 10], [0.1, 0])
```

Historical Context:

- NumPy (2006) unified Numeric and NumArray ecosystems
- SciPy (2001) brought scientific algorithms to Python
- Jupyter (2015) revolutionized interactive scientific computing
- Now: NASA, CERN, LIGO use Python for data analysis

Python for Science - Numerical Computing

Pythonic Rationale:

- NumPy provides C-speed arrays with Python convenience
- Vectorized operations eliminate slow Python loops
- Broadcasting handles arrays of different shapes automatically
- SciPy wraps decades of Fortran numerical libraries

Python Code:

```
import numpy as np
```

```
# Vectorized operations - 100x faster than Python loops
```

```
a = np.array([1, 2, 3, 4, 5])
```

```
b = np.array([10, 20, 30, 40, 50])
```

```
c = a + b # Element-wise: [11, 22, 33, 44, 55]
```

```
# Broadcasting: (3,1) array + (1,4) array = (3,4) array
```

```
x = np.arange(3).reshape(3, 1)
```

```
y = np.arange(4).reshape(1, 4)
```

```
z = x + y # Automatic shape expansion
```

Python for Science - Numerical Computing (continued)

C Comparison:

```
// C: Manual loops, no broadcasting
double a[5] = {1, 2, 3, 4, 5};
double b[5] = {10, 20, 30, 40, 50};
double c[5];
for (int i = 0; i < 5; i++) {
    c[i] = a[i] + b[i]; // Explicit loop required
}
```

Fortran Comparison:

```
! Fortran: Array syntax (inspired NumPy)
real :: a(5), b(5), c(5)
c = a + b ! Element-wise, like NumPy
! But: limited ecosystem, harder to learn
```

Computer Science Theory:

- Vectorization: SIMD instructions (SSE, AVX) for parallelism
- Broadcasting: shape inference and stride manipulation
- Memory layout: contiguous arrays for cache efficiency

Subsection 3

Python Ecosystem

Python Ecosystem - Core Libraries

NumPy: Foundation for numerical computing - N-dimensional arrays: ndarray with broadcasting - Linear algebra: `np.linalg`, matrix operations - FFT, random numbers, polynomial fitting

SciPy: Scientific algorithms - Optimization: `scipy.optimize` (minimize, root finding) - Integration: `scipy.integrate` (ODE solvers, quadrature) - Signal processing: `scipy.signal` (filtering, spectral analysis)

Pandas: Data manipulation - DataFrame: labeled 2D arrays with mixed types - Time series: resampling, rolling windows - I/O: read/write CSV, Excel, SQL, Parquet

Matplotlib: Visualization - Publication-quality figures: lines, scatter, contours - Customizable: fonts, colors, LaTeX integration - Multiple backends: interactive, PDF, PNG

Python Ecosystem - Machine Learning

scikit-learn: Classical machine learning - Regression: Linear, Ridge, Lasso, ElasticNet - Classification: LogisticRegression, SVM, RandomForest - Clustering: KMeans, DBSCAN, hierarchical - Model selection: cross-validation, hyperparameter tuning

TensorFlow/PyTorch: Deep learning - Neural networks: CNN, RNN, Transformer architectures - GPU acceleration: CUDA, automatic differentiation - Production deployment: TensorFlow Serving, TorchScript

Python Code:

```
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier

X_train, X_test, y_train, y_test = train_test_split(X, y)
model = RandomForestClassifier(n_estimators=100)
model.fit(X_train, y_train)
accuracy = model.score(X_test, y_test)
```

Python Ecosystem - Domain Applications

Neuroscience: - Brian2: Spiking neural network simulator - NEST: Large-scale neural network simulation - Neo: Electrophysiology data handling - MNE: EEG/MEG analysis

Physics: - Astropy: Astronomy data analysis, coordinates, units - QuTiP: Quantum mechanics simulations - h5py: HDF5 data format for large datasets

Biology: - Biopython: Sequence analysis, GenBank access - scikit-image: Image processing for microscopy - Scanpy: Single-cell RNA sequencing analysis

Engineering: - FEniCS: Finite element simulations - OpenFOAM (Python bindings): CFD simulations - SimPy: Discrete-event simulation

Section 2

Part I: Python Syntax Fundamentals

Subsection 1

Basic Syntax and Data Types

Theory - Types and Values

Conceptual Overview:

- Python is dynamically typed: values have types, variables don't
- Types are checked at runtime, not compile time
- Same variable can hold different types at different times

Mathematical Foundations:

- Type: a set of values with defined operations
- Dynamic typing: $\text{type}(v)$ determined at runtime
- Type checking: verify operation valid for operand types

Historical Context:

- Dynamic typing from Lisp (1958), the first dynamically typed language
- Contrast with static typing in C, Java, Rust
- Type hints (PEP 484, 2014) add optional static checking

Types and Values

Python is dynamically typed: values have types, variables don't

Type	Description	Example
int	Whole numbers	42
float	Decimals	3.14
str	Text	'Hello'
list	Mutable sequence	[1, 2, 3]
dict	Key-value mapping	{'a': 1}

Key behaviors: - `type('2')` is `str`, not `int` - Division always returns float: `84 / 2`
→ `42.0` - Commas create tuples: `1,000,000` is `(1, 0, 0)`

Python Philosophy - Dynamic Typing

Pythonic Rationale:

- “Easier to Ask Forgiveness than Permission” (EAFP)
- Readability over compile-time safety
- Rapid prototyping and experimentation

Python Code:

```
# Python: Dynamic typing - types checked at runtime  
x = 42           # x is an int  
x = "hello"      # x is now a str - perfectly valid!  
type(x)          # <class 'str'>
```

Python Philosophy - Dynamic Typing (continued)

C Comparison:

```
// C: Static typing - errors at compile time  
int x = 42;  
x = "hello"; // Error: incompatible types
```

Rust Comparison:

```
// Rust: Type inference with compile-time safety  
let x = 42; // infers i32  
let y: &str = "hello"; // explicit type  
// x = "hello"; // Error: mismatched types
```

Theory - Type Systems

Conceptual Overview:

- Static typing checks types at compile time, catching errors before runtime
- Dynamic typing checks types at runtime, offering flexibility but deferring error detection
- Type inference automatically deduces types without explicit annotations

Mathematical Foundations:

- Type judgment: $\Gamma \vdash e : \tau$ (expression e has type τ in context Γ)
- Hindley-Milner algorithm: computes principal (most general) type for expressions
- Type inference: $\text{infer}(e) = \tau$ without explicit annotations
- Gradual typing: consistency relation allows mixing typed and untyped code

Historical Context:

- Type theory from Russell (1908) to avoid paradoxes in set theory
- Hindley-Milner algorithm discovered independently (Hindley 1969, Milner 1978)
- Gradual typing formalized by Siek and Taha (2006)
- Python type hints (PEP 484, 2014) enable optional static checking with mypy

Theory - Arithmetic Operators

Conceptual Overview:

- Arithmetic operators perform mathematical calculations
- Python supports standard operations plus floor division and modulus
- Division always returns float, floor division returns int

Mathematical Foundations:

- Floor division: $\lfloor a/b \rfloor$ rounds toward negative infinity
- Modulus: $a \bmod b = a - b \cdot \lfloor a/b \rfloor$
- Exponentiation: a^b for integer exponents, a^b for any real

Historical Context:

- Arithmetic operators from mathematical notation
- Floor division `//` distinguishes from true division `/`
- Python 3 made `/` always return float (breaking change from Python 2)

Arithmetic Operators

Operator	Math	Python	Example
Addition	$a + b$	<code>a + b</code>	<code>40 + 2 → 42</code>
Subtraction	$a - b$	<code>a - b</code>	<code>43 - 1 → 42</code>
Multiplication	$a \times b$	<code>a * b</code>	<code>6 * 7 → 42</code>
Division	$a \div b$	<code>a / b</code>	<code>84 / 2 → 42.0</code>
Exponentiation	a^b	<code>a ** b</code>	<code>6**2 → 36</code>
Floor division	$\lfloor a/b \rfloor$	<code>a // b</code>	<code>7 // 3 → 2</code>
Modulus	$a \bmod b$	<code>a % b</code>	<code>7 % 3 → 1</code>

Warning: `^` is XOR, not exponentiation!

Theory - Order of Operations (PEMDAS)

Conceptual Overview:

- Operations have defined precedence (order of evaluation)
- Parentheses override default precedence
- Same precedence operators evaluate left-to-right

Mathematical Foundations:

- Precedence: $\text{priority}(\text{op})$ determines order
- Associativity: left-to-right for most operators
- Parentheses: (e) evaluated before surrounding expression

Historical Context:

- PEMDAS/BODMAS from mathematical conventions
- Programming languages formalize precedence tables
- Exponentiation right-associative: $2^{3^4} = 2^{81}$

Order of Operations (PEMDAS)

- 1 **P**arentheses
- 2 **E**xponentiation
- 3 **M**ultiplication/**D**ivision (left to right)
- 4 **A**ddition/**S**ubtraction (left to right)

Example:

```
>>> 2 + 3 * 4
14          # Not 20!
>>> (2 + 3) * 4
20
```


Subsection 2

Variables and Expressions

Theory - Assignment

Conceptual Overview:

- Assignment binds a name to a value
- Multiple assignment and tuple unpacking are idiomatic
- Assignment creates references, not copies of objects

Mathematical Foundations:

- Assignment: $\text{env}' = \text{env}[x \mapsto v]$ (environment update)
- Reference semantics: $x = y$ makes x point to same object as y
- Tuple unpacking: $(a, b) = (b, a)$ swaps without temp variable

Historical Context:

- Assignment operator = from FORTRAN and ALGOL
- Reference semantics from Lisp and Smalltalk
- Tuple unpacking unique to Python

Assignment

```
n = 17           # Simple assignment
x = y = 1        # Multiple assignment
a, b = b, a       # Tuple swap (no temp variable)
```

Key concept: Assignment creates references, not copies

Python Philosophy - The Reference Model

Pythonic Rationale:

- Variables are labels, not boxes
- “Names bound to objects” mental model
- Everything is an object, everything is a reference

Python Code:

```
# Python: Variables are labels on objects  
x = 42      # x labels the int object 42  
y = x      # y labels the same object  
x = 99      # x now labels 99, y still labels 42  
# No ownership transfer!
```

Python Philosophy - The Reference Model (continued)

C Comparison:

```
// C: Variables as memory boxes  
int x = 42;           // x is a box containing 42  
int *p = &x;         // p points to x's box  
*p = 99;             // x is now 99
```

Rust Comparison:

```
// Rust: Ownership model  
let x = 42;           // x owns the value  
let y = x;            // y takes ownership (move)  
// x is invalid now!  
let z = y.clone();    // explicit copy
```

Theory - Reference vs Value Semantics

Conceptual Overview:

- Reference semantics: variables hold references to objects; assignment copies the reference
- Value semantics: variables hold values directly; assignment copies the value
- Aliasing occurs when multiple variables refer to the same object, affecting mutation behavior

Mathematical Foundations:

- Reference semantics: $\text{assign}(x, y) \Rightarrow \text{ref}(x) = \text{ref}(y)$; mutation visible through both
- Value semantics: $\text{assign}(x, y) \Rightarrow \text{val}(x) = \text{copy}(\text{val}(y))$; independent copies
- Aliasing: $\text{alias}(x, y) \Leftrightarrow \text{id}(x) = \text{id}(y)$
- Memory model: stack for local variables (automatic lifetime), heap for objects (manual/GC lifetime)

Historical Context:

- Reference semantics from Lisp (1958) and Smalltalk (1972)
- Value semantics from C (1972) where structs are copied on assignment
- Java uses reference semantics for objects, value semantics for primitives
- Rust (2010) introduced ownership to make aliasing explicit and safe

Theory - Naming Rules

Conceptual Overview:

- Identifiers name variables, functions, classes
- Names must follow lexical rules
- Good names improve code readability

Mathematical Foundations:

- Identifier: $id \in \Sigma^+$ where Σ is allowed characters
- Scope: region where identifier is visible
- Namespace: mapping from names to objects

Historical Context:

- Naming conventions evolved with each language
- snake_case from C and Unix tradition
- PascalCase from Pascal and Smalltalk

Naming Rules

- Letters, numbers, underscore only
- Cannot start with number
- Keywords (`if`, `def`, `while`, `class`) cannot be used
- **Style:** `snake_case` for variables/functions, `PascalCase` for classes

Theory - Expressions vs Statements

Conceptual Overview:

- Expressions evaluate to a value
- Statements perform an action (side effect)
- Understanding the difference helps write correct code

Mathematical Foundations:

- Expression: $e \rightarrow v$ (evaluates to value)
- Statement: $s \rightarrow \text{effect}$ (produces side effect)
- Composition: expressions can nest, statements execute sequentially

Historical Context:

- Expression/statement distinction from ALGOL 60
- Functional languages treat everything as expressions
- Imperative languages distinguish statements with side effects

Expressions vs Statements

- **Expression:** Evaluates to a value ($n + 25$, $\text{len}(s)$)
- **Statement:** Performs an action ($n = 17$, $\text{print}(n)$, $\text{if } x:$)

Example:

```
n = 17          # Statement (assignment)
print(n + 5)    # Statement (print), contains expression  $n + 5$ 
```

Subsection 3

Functions

Theory - Definition and Key Concepts

Conceptual Overview:

- Functions encapsulate reusable code blocks
- Parameters are variables in the function definition
- Arguments are values passed when calling the function

Mathematical Foundations:

- Function: $f : X \rightarrow Y$ maps inputs to outputs
- Parameters: $f(x, y)$ - formal variables
- Arguments: $f(a, b)$ - actual values substituted

Historical Context:

- Functions from mathematical functions (Euler, 18th century)
- Subroutines invented for assembly programming (1940s)
- Modern function syntax from ALGOL and CPL

Definition and Key Concepts

```
def function_name(parameters):  
    """Docstring"""  
    body  
    return value
```

Key distinctions: - **Parameters** (in definition) vs **Arguments** (at call) - **Fruitful** (returns value) vs **Void** (returns None)

Example:

```
def is_divisible(x, y):      # parameters: x, y  
    return x % y == 0  
  
is_divisible(10, 2)         # arguments: 10, 2 → True
```

Theory - Type Checking (Guardian Pattern)

Conceptual Overview:

- Guardian pattern checks preconditions early
- Prevents invalid inputs from causing errors later
- Returns None or raises exception for invalid types

Mathematical Foundations:

- Precondition: $P(x) \Rightarrow f(x)$ produces valid result
- Type guard: $\text{isinstance}(x, T)$ checks type membership
- Defensive programming: validate before processing

Historical Context:

- Defensive programming from reliability engineering (1970s)
- Design by Contract (Meyer, 1986) formalized preconditions
- Python's `isinstance` for runtime type checking

Type Checking (Guardian Pattern)

```
def factorial(n):  
    if not isinstance(n, int):  
        return None  
    if n < 0:  
        return None  
    if n == 0:  
        return 1  
    return n * factorial(n - 1)
```

Mathematical definition:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$

Subsection 4

Conditionals and Recursion

Theory - Conditional Statements

Conceptual Overview:

- Conditionals execute code based on boolean conditions
- if-elif-else chain tests conditions in order
- Python allows chained comparisons for readability

Mathematical Foundations:

- Conditional: if P then A else B
- Chained: $a < b < c \equiv a < b \wedge b < c$
- Short-circuit: and/or evaluate left-to-right, stop when result known

Historical Context:

- Conditional expressions from ALGOL 60 (1960)
- Chained comparisons unique to Python
- Short-circuit evaluation from Lisp

Conditional Statements

```
# Chained
if x < y:
    print('less')
elif x > y:
    print('greater')
else:
    print('equal')

# Pythonic: chained comparison
if 0 < x < 10:
    print('single digit')
```

Logical operators: and, or, not

Theory - Recursion Requirements

Conceptual Overview:

- Recursion is a function calling itself
- Must have base case to terminate
- Each call should progress toward base case

Mathematical Foundations:

- Recursive definition: $f(n) = g(f(n-1))$ with $f(0) = base$
- Mathematical induction: prove $P(0)$, prove $P(n) \Rightarrow P(n+1)$
- Termination: decreasing measure ensures eventual base case

Historical Context:

- Recursion formalized in mathematical logic (Gödel, 1931)
- Recursive functions in programming from Lisp (1958)
- McCarthy's recursive style influenced all functional languages

Recursion Requirements

```
def factorial(n):  
    if n == 0:                # Base case  
        return 1  
    else:                     # Recursive case  
        return n * factorial(n - 1)
```

Must have: 1. Base case (no recursion) 2. Progress toward base case 3. Avoid infinite recursion

Subsection 5

Loops and Iteration

Theory - while Loop

Conceptual Overview:

- while loop repeats as long as condition is true
- Condition checked before each iteration
- Use when iteration count is unknown beforehand

Mathematical Foundations:

- Loop invariant: condition I holds before and after each iteration
- Termination: \exists variant V that decreases and reaches 0
- while C : body \Rightarrow loop until $\neg C$

Historical Context:

- while loop from ALGOL 60 (1960)
- Fundamental to structured programming
- Dijkstra proved loops with invariants are correct

while Loop

```
n = 5
while n > 0:
    print(n)
    n -= 1
print('Blastoff!')
```

Use when: Number of iterations is unknown (condition-based)

Theory - for Loop

Conceptual Overview:

- for loop iterates over any iterable sequence
- More concise than while when iteration count is known
- Works with strings, lists, ranges, files, and custom iterables

Mathematical Foundations:

- Iterator: $\text{next} : S \rightarrow (\text{element}, S')$ or StopIteration
- For loop: for each $x \in S$: execute body
- Range: $\text{range}(n) = \{0, 1, 2, \dots, n - 1\}$

Historical Context:

- for-each pattern from CLU (1974) and ABC
- Python's for loop is iterator-based, not counter-based
- Iterators unified the iteration protocol (Python 2.2)

for Loop

```
for letter in 'banana':  
    print(letter)  
  
for i in range(5):  
    print(i)  # 0, 1, 2, 3, 4
```

Use when: Iterating over a sequence or known range

Theory - break and continue

Conceptual Overview:

- break exits the loop immediately
- continue skips to the next iteration
- Both alter normal loop control flow

Mathematical Foundations:

- break: $\text{loop}(S) \rightarrow \text{exit on condition}$
- continue: $\text{loop}(S) \rightarrow \text{next_iteration on condition}$
- Control flow: early exit vs skip current

Historical Context:

- break and continue from BCPL and C
- Structured programming debated goto alternatives
- Now standard in most programming languages

break and continue

```
while True:
    line = input('> ')
    if line == 'done':
        break          # Exit loop immediately
    print(line)

for i in range(10):
    if i % 2 == 0:
        continue      # Skip to next iteration
    print(i)          # Prints odd numbers only
```

Theory - Loop Patterns

Conceptual Overview:

- Common loop patterns solve recurring problems
- Counter pattern accumulates a total
- Search pattern finds first match and returns early

Mathematical Foundations:

- Counter: $\text{count}(f, S) = |\{x \in S : f(x)\}|$
- Search: $\text{find}(f, S) = \min\{i : f(s_i)\}$ or -1
- Accumulator pattern: $\text{fold}(op, S, init)$

Historical Context:

- Loop patterns documented in structured programming (Dijkstra, 1968)
- Search patterns fundamental to algorithm design
- Early return pattern improves efficiency

Loop Patterns

Counter Pattern:

```
count = 0
for letter in word:
    if letter == 'a':
        count += 1
```

Search Pattern:

```
def find(word, letter):
    for i, char in enumerate(word):
        if char == letter:
            return i
    return -1
```

Subsection 6

Data Structures

Theory - Strings: Indexing and Slicing

Conceptual Overview:

- Strings are immutable sequences of characters
- Indexing accesses individual characters (0-based)
- Slicing extracts substrings with start:stop:step syntax

Mathematical Foundations:

- Indexing: $s[i]$ returns character at position i
- Negative index: $s[-i] = s[\text{len}(s) - i]$
- Slice: $s[a : b] = s[a], s[a + 1], \dots, s[b - 1]$

Historical Context:

- 0-based indexing from C and BCPL languages
- Python slicing syntax inspired by ABC language
- Immutability enables safe sharing and hashing

Strings: Indexing and Slicing

```
s = 'Monty Python'
s[0]      # 'M' (first char)
s[-1]     # 'n' (last char)
s[0:5]    # 'Monty'
s[::-1]   # 'nohtyP ytnoM' (reversed)
```

Immutability: `s[0] = 'J'` raises `TypeError`

Theory - String Methods

Conceptual Overview:

- Strings provide rich set of methods for manipulation
- Methods return new strings (strings are immutable)
- Common operations: case conversion, search, split, join

Mathematical Foundations:

- Immutability: $\forall s, m : m(s) = s' \wedge s \neq s'$
- Search: $\text{find}(s, \text{sub}) = \min\{i : s[i : i + \text{len}(\text{sub})] = \text{sub}\}$
- Split: $\text{split}(s, \text{sep}) = [s_1, s_2, \dots, s_n]$ where $s = s_1 + \text{sep} + s_2 + \dots$

Historical Context:

- String methods standardized across programming languages
- Python's string methods inspired by Perl and shell utilities
- Regular expressions provide more powerful pattern matching

String Methods

```
s.upper()           # 'MONTY PYTHON'  
s.lower()           # 'monty python'  
s.find('Python')    # 6 (index or -1)  
s.count('a')        # count occurrences  
' '.join(['a', 'b']) # 'a b'  
'a b c'.split()     # ['a', 'b', 'c']  
s.strip()           # remove whitespace  
s.replace('a', 'b') # replace all
```

Python Philosophy - Unicode First

Pythonic Rationale:

- Strings are Unicode by default (Python 3)
- Text processing without encoding headaches
- One string type, not bytes vs text confusion

Python Code:

```
# Python: Unicode strings by default  
s = "hello"           # str type, Unicode  
s2 = "café"           # Also valid Unicode string  
len(s2)               # 2 (code points, not bytes)
```

Python Philosophy - Unicode First (continued)

C Comparison:

```
// C: Strings are null-terminated byte arrays  
char *s = "hello";      // ASCII only  
wchar_t *ws = L"unicode string"; // Wide chars, platform-dependent  
// No built-in Unicode support
```

Rust Comparison:

```
// Rust: UTF-8 enforced at type level  
let s: &str = "hello";    // UTF-8 slice  
let s2: String = String::from("unicode string"); // Owned UTF-8  
// Invalid UTF-8 is a compile or runtime error  
let bytes: &[u8] = s.as_bytes(); // Explicit conversion
```

Theory - Character Encoding and Unicode

Conceptual Overview:

- Character encoding maps characters to byte sequences
- ASCII uses 7 bits (128 characters); UTF-8 uses 1-4 bytes per character
- Unicode code points identify characters; grapheme clusters are user-perceived characters

Mathematical Foundations:

- ASCII: $\text{encode}(c) = b$ where $b \in [0, 127]$, fixed 1 byte per character
- UTF-8: variable-length encoding, n bytes where n depends on code point range
- Code point: integer $U + XXXX$ identifying a Unicode character
- Grapheme cluster: sequence of code points forming a single user-perceived character (e.g., $\acute{e} = e + \text{combining accent}$)

Historical Context:

- ASCII standardized in 1963 (7-bit, American characters only)
- Unicode started in 1987 to unify all character sets; version 1.0 in 1991
- UTF-8 designed by Ken Thompson (1992) for Plan 9; now dominant web encoding
- Python 3 (2008) made strings Unicode by default, fixing Python 2's bytes/str confusion

Theory - String Comparison

Conceptual Overview:

- Strings are compared lexicographically (dictionary order)
- Comparison uses Unicode code point values
- Case matters: uppercase letters sort before lowercase

Mathematical Foundations:

- Lexicographic: $s < t \Leftrightarrow \exists i : s_i < t_i \wedge \forall j < i : s_j = t_j$
- Unicode ordering: $\text{ord}('A') < \text{ord}('a')$ ($65 < 97$)
- Normalization: $\text{lower}(s) = \text{lower}(t)$ for case-insensitive

Historical Context:

- ASCII ordering from telegraph codes (1963)
- Unicode standardized character encoding (1991)
- Locale-aware sorting (collation) for internationalization

String Comparison

```
# Uppercase < lowercase (ASCII order)  
'Pineapple' < 'banana' # True  
  
# Best practice: normalize before comparing  
word.lower() == 'banana'
```

Theory - Lists: Operations and Aliasing

Conceptual Overview:

- Lists are mutable, ordered sequences
- Aliasing occurs when multiple variables reference the same object
- Understanding references is critical for avoiding bugs

Mathematical Foundations:

- List as sequence: $L = [l_0, l_1, \dots, l_{n-1}]$
- Aliasing: $\text{alias}(x, y) \Leftrightarrow \text{id}(x) = \text{id}(y)$
- Copy: $L' = L$ creates alias, $L' = L[:]$ creates new list

Historical Context:

- Lists fundamental to Lisp (1958), the first list-processing language
- Python lists implemented as dynamic arrays (not linked lists)
- Reference semantics shared with Java, JavaScript, Ruby

Lists: Operations and Aliasing

```
# Methods (modify in place)
t.append('x')      # add to end
t.sort()          # sort in place
t.pop()           # remove last

# Aliasing trap
a = [1, 2, 3]
b = a              # SAME list
b[0] = 99          # a also changes!

# Safe copy
b = a[:]           # or list(a), a.copy()
```

Python Philosophy - Dynamic Arrays

Pythonic Rationale:

- Lists grow automatically - no manual sizing
- Amortized $O(1)$ append
- Trade memory for convenience

Python Code:

```
# Python: Lists grow automatically  
arr = []  
arr.append(42)      # Automatic resize  
arr.extend([1, 2, 3]) # Add multiple  
# No manual memory management
```

Python Philosophy - Dynamic Arrays (continued)

C Comparison:

```
// C: Manual array management  
int *arr = malloc(10 * sizeof(int));  
int capacity = 10, size = 0;  
// Must track capacity, handle reallocation  
arr = realloc(arr, capacity * 2 * sizeof(int));
```

Rust Comparison:

```
// Rust: Vec with safety guarantees  
let mut v: Vec<i32> = Vec::new();  
v.push(42); // Automatic growth  
// Bounds checking at runtime  
let x = v[0]; // Panics if empty  
let y = v.get(0); // Returns Option
```

Theory - Dynamic Array Amortized Analysis

Conceptual Overview:

- Dynamic arrays grow automatically when capacity is exceeded
- Amortized analysis shows that occasional expensive resizes are offset by many cheap appends
- The growth factor determines the trade-off between memory overhead and resize frequency

Mathematical Foundations:

- Amortized cost: $\text{cost}_{\text{amortized}} = \frac{\sum_{i=1}^n \text{cost}(op_i)}{n}$
- Geometric growth: when full, allocate array of size $k \cdot n$ where $k > 1$
- Append analysis: n appends cost $O(n)$ for copies + $O(n)$ for assignments = $O(n)$ total, $O(1)$ amortized
- Growth factor k : Python uses ≈ 1.125 , trade-off between memory and speed

Historical Context:

- Dynamic arrays described in early algorithm textbooks (1970s)
- Amortized analysis formalized by Tarjan (1985)
- Different languages use different growth factors: Python $\sim 1.125x$, Java ArrayList $1.5x$, C++ vector $2x$
- Larger growth factor means fewer resizes but more wasted memory

Theory - Dictionaries

Conceptual Overview:

- Dictionaries are mutable mappings from keys to values
- Keys must be hashable (immutable types)
- Provide $O(1)$ average-case lookup, insertion, and deletion

Mathematical Foundations:

- Dictionary as function: $D : K \rightarrow V$ (partial function)
- Hash function: $h : K \rightarrow \mathbb{N}$ maps keys to indices
- Collision resolution: chaining or open addressing

Historical Context:

- Hash tables invented in 1953 (IBM research)
- Python dictionaries use open addressing with probing
- Python 3.7+ guarantees insertion order (implementation detail became spec)

Dictionaries

```
d = {'a': 1, 'b': 2}
d['a']          # 1
d['x']          # KeyError
d.get('x', 0)   # 0 (default)

# Histogram pattern
def histogram(s):
    d = {}
    for c in s:
        d[c] = d.get(c, 0) + 1
    return d
```

Keys must be hashable (immutable)

Theory - Tuples and Unpacking

Conceptual Overview:

- Tuples are immutable, ordered sequences
- Unpacking assigns tuple elements to multiple variables
- Common pattern for returning multiple values from functions

Mathematical Foundations:

- Tuple as Cartesian product: $T = A \times B \times C$
- Immutability: once created, elements cannot change
- Unpacking: $(a, b, c) = t$ where $t = (t_1, t_2, t_3)$

Historical Context:

- Tuples from mathematical tuples (ordered n-tuples)
- Python tuples inspired by ABC language records
- Unpacking pattern common in functional languages

Tuples and Unpacking

```
t = (1, 2, 3)          # or 1, 2, 3
single = (1,)          # comma required!

# Tuple assignment (swap)
a, b = b, a

# zip and enumerate
for i, char in enumerate('abc'):
    print(i, char)  # 0 a, 1 b, 2 c
```

Immutable - can be dictionary keys

Subsection 7

Advanced Python Features

Theory - Comprehensions

Conceptual Overview:

- Comprehensions provide concise syntax for creating sequences
- Combine mapping and filtering in a single expression
- More readable and often faster than equivalent loops

Mathematical Foundations:

- Set builder notation: $S = \{f(x) \mid P(x), x \in X\}$
- Map-filter pattern: $\text{map}(f, \text{filter}(P, X))$
- Nested: $\{f(x, y) \mid x \in X, y \in Y\}$

Historical Context:

- List comprehensions from SETL (late 1960s) and Haskell
- Python added list comprehensions in 2.0 (2000)
- Inspired by ABC language and functional programming

Comprehensions

List Comprehensions:

```
squares = [x**2 for x in range(10)]  
evens = [x for x in range(20) if x % 2 == 0]
```

Nested:

```
matrix = [[i*j for j in range(5)] for i in range(5)]
```

Python Philosophy - Declarative Style

Pythonic Rationale:

- Say WHAT, not HOW
- Comprehensions > loops for simple transformations
- Readable one-liners for common patterns

Python Code:

```
# Python: Comprehensions for declarative style
squares = [x**2 for x in range(10)]
evens = [x for x in range(20) if x % 2 == 0]
# Say WHAT, not HOW
```

Python Philosophy - Declarative Style (continued)

C Comparison:

```
// C: Imperative style only  
int squares[10];  
for (int i = 0; i < 10; i++) {  
    squares[i] = i * i;  
}  
// No built-in filter/map operations
```

Rust Comparison:

```
// Rust: Iterator chains (functional style)  
let squares: Vec<i32> = (0..10)  
    .map(|x| x * x)  
    .collect();  
let evens: Vec<i32> = (0..20)  
    .filter(|x| x % 2 == 0)  
    .collect();
```

Theory - Comprehensions and Set Builder Notation

Conceptual Overview:

- Comprehensions are derived from mathematical set builder notation
- They combine mapping (transformation) and filtering in a single expression
- List comprehensions are eager; generator expressions are lazy

Mathematical Foundations:

- Set builder notation: $S = \{f(x) \mid P(x), x \in X\}$ = elements $f(x)$ where $x \in X$ satisfies P
- List comprehension: $[f(x) \text{ for } x \text{ in } X \text{ if } P(x)] \equiv \text{list}(\text{filter}(P, \text{map}(f, X)))$
- Generator expression: $(f(x) \text{ for } x \text{ in } X \text{ if } P(x))$ yields values lazily
- Complexity: eager $O(n)$ space, lazy $O(1)$ space

Historical Context:

- Set builder notation from set theory (Cantor, late 1800s)
- List comprehensions in SETL (late 1960s), Miranda (1985), Haskell (1990)
- Python added list comprehensions in 2.0 (2000), generator expressions in 2.4 (2004)
- Inspired by ABC language and functional programming; now in JavaScript, Rust, etc.

Theory - Generator Expressions

Conceptual Overview:

- Generator expressions produce values lazily, one at a time
- Use parentheses instead of square brackets
- Memory efficient for large sequences and pipelines

Mathematical Foundations:

- Lazy evaluation: $\text{yield}(x_i)$ on demand, not precomputed
- Memory complexity: $O(1)$ vs $O(n)$ for lists
- Single-pass: generator state consumed after iteration

Historical Context:

- Generator expressions introduced in Python 2.4 (2004)
- Inspired by lazy evaluation in Haskell and generator functions
- Enable streaming data processing without loading all into memory

Generator Expressions

Lazy evaluation - no memory allocation:

```
g = (x**2 for x in range(1000000))  
total = sum(x**2 for x in range(10))
```

Note: Generators can only be iterated once!

Theory - Dictionary and Set Comprehensions

Conceptual Overview:

- Dictionary comprehensions create dictionaries from iterables
- Set comprehensions create sets using similar syntax
- Both follow the same pattern as list comprehensions

Mathematical Foundations:

- Dict comprehension: $\{k : v \mid (k, v) \in S\}$ where S is source
- Set comprehension: $\{x \mid P(x), x \in S\}$ with uniqueness constraint
- Uniqueness: sets enforce $|S| = |\{x \mid x \in S\}|$ (no duplicates)

Historical Context:

- Dictionary comprehensions added in Python 2.7 (2010)
- Set comprehensions added simultaneously
- Unified syntax across all comprehension types

Dictionary and Set Comprehensions

```
# Dictionary  
d = {x: x**2 for x in range(5)} # {0: 0, 1: 1, 2: 4, ...}  
  
# Set  
s = {x for x in 'abracadabra' if x not in 'abc'} # {'r', 'd'}
```

Theory - Collections Module

Conceptual Overview:

- Collections module provides specialized container data types
- Extends built-in types with additional functionality
- Counter, defaultdict, and namedtuple are most commonly used

Mathematical Foundations:

- Counter: multiset (bag) - $M : V \rightarrow \mathbb{N}$ maps values to counts
- defaultdict: auto-initialization with default factory function
- namedtuple: tuple with named fields, immutable like regular tuples

Historical Context:

- Collections module added in Python 2.4 (2004)
- namedtuple added in Python 2.6 (2008)
- Inspired by similar data structures in functional programming

Collections Module

```
from collections import Counter, defaultdict, namedtuple

# Counter - word frequency
count = Counter('parrot')
count.most_common(3) # [('r', 2), ('p', 1), ...]

# defaultdict - auto-create keys
d = defaultdict(list)
d['new'].append('value') # No KeyError

# namedtuple
Point = namedtuple('Point', ['x', 'y'])
p = Point(1, 2)
p.x, p.y # 1, 2
```


Section 3

Part II: Object-Oriented Design

Subsection 1

Classes and Objects

Theory - Basic Class Definition

Conceptual Overview:

- Classes define blueprints for creating objects
- Objects are instances with attributes (data) and methods (behavior)
- Class attributes are shared; instance attributes are unique per object

Mathematical Foundations:

- Class as type: C defines set of objects $\{o_1, o_2, \dots\}$ with same structure
- Instance: $o \in C$ has attributes $o.a_1, o.a_2, \dots$
- Class attribute: $C.a$ shared across all instances $o \in C$

Historical Context:

- Object-oriented programming from Simula (1967) and Smalltalk (1972)
- Classes combine data and behavior in a single abstraction
- Python classes are dynamic - attributes can be added at runtime

Basic Class Definition

```
class Point:
    """Represents a point in 2-D space."""

blank = Point()
blank.x = 3.0
blank.y = 4.0
```

Class attributes (shared by all instances):

```
class Card:
    suit_names = ['Clubs', 'Diamonds', ...]
```

Theory - Methods

Conceptual Overview:

- Methods are functions defined within a class
- The first parameter `self` refers to the instance calling the method
- Special methods (`__init__`, `__str__`) define object behavior

Mathematical Foundations:

- Method as function: $m : C \times P \rightarrow R$ where C is class, P parameters, R return
- self binding: $o.m(x) \equiv C.m(o, x)$ (method is called on instance)
- Constructor: `__init__` initializes instance state

Historical Context:

- Methods evolved from message passing in Smalltalk
- Python's explicit `self` follows the “explicit is better than implicit” principle
- Special methods (dunder methods) enable operator overloading

Methods

```
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        return f"{self.hour:02d}:{self.minute:02d}:{self.second:02d}"
```

Best practice: Always write `__init__` and `__str__` first!

Python Philosophy - Why self?

Pythonic Rationale:

- Explicit self makes instance binding clear
- “Explicit is better than implicit” (PEP 20)
- No hidden this pointer - method is just a function

Python Code:

Python: Explicit self parameter

```
class Point:
```

```
    def __init__(self, x, y):
```

```
        self.x = x # Must use self to access instance
```

```
        self.y = y
```

```
    def move(self, dx, dy):
```

```
        self.x += dx
```

```
        self.y += dy
```

Python Philosophy - Why self? (continued)

C Comparison:

```
// C: No methods, just structs and functions  
struct Point {  
    double x, y;  
};  
void point_move(struct Point *self, double dx, double dy) {  
    self->x += dx;  
    self->y += dy;  
}  
// Must pass 'self' explicitly
```

Rust Comparison:

```
// Rust: Methods with explicit self parameter  
struct Point { x: f64, y: f64 }  
impl Point {  
    fn move_self(&mut self, dx: f64, dy: f64) {  
        self.x += dx;  
        self.y += dy;  
    }  
}  
// &mut self is explicit borrowing
```

Theory - Method Dispatch and Binding

Conceptual Overview:

- Method dispatch determines which method implementation executes for a given call
- Static binding (compile-time) resolves method calls based on declared type
- Dynamic binding (runtime) resolves based on actual object type, enabling polymorphism

Mathematical Foundations:

- Static binding: $\text{dispatch}(o, m) = \text{method}(\text{declared_type}(o), m)$ at compile time
- Dynamic binding: $\text{dispatch}(o, m) = \text{method}(\text{actual_type}(o), m)$ at runtime
- Bound method: $\text{bind}(m, o) = \lambda \text{args}. m(o, \text{args})$ captures receiver
- Closure: $\lambda x. e$ where e references variables from enclosing scope

Historical Context:

- Virtual methods in Simula (1967) and Smalltalk (1972) pioneered dynamic dispatch
- C++ virtual functions (1983) use vtables for efficient dynamic dispatch
- Python always uses dynamic dispatch via attribute lookup on `self`
- The `self/this` parameter is implicit in many languages; Python makes it explicit

Theory - Operator Overloading

Conceptual Overview:

- Operator overloading lets classes define behavior for built-in operators
- Special methods (`__add__`, `__lt__`, etc.) are called when operators are used
- Enables natural, readable syntax for custom types

Mathematical Foundations:

- Operator as method: $a + b \equiv a.\text{__add__}(b)$
- Binary operator: defines behavior for two operands
- Comparison operators define ordering relationships

Historical Context:

- Operator overloading from C++ (1983) and Ada
- Python uses special methods (dunder methods) for operator overloading
- Enables mathematical types to behave like built-in types

Operator Overloading

Method	Operator
<code>__add__</code>	<code>+</code>
<code>__sub__</code>	<code>-</code>
<code>__mul__</code>	<code>*</code>
<code>__lt__</code>	<code><</code>
<code>__len__</code>	<code>len()</code>
<code>__getitem__</code>	<code>[]</code>

Example:

```
def __add__(self, other):  
    return Time(self.time_to_int() + other.time_to_int())
```


Subsection 2

Inheritance and Design

Theory - Inheritance Basics

Conceptual Overview:

- Inheritance allows a class to derive attributes and methods from another class
- Child class (subclass) inherits from parent class (superclass)
- Enables code reuse and hierarchical organization

Mathematical Foundations:

- Inheritance as subset: $Child \subseteq Parent$ (all parent members available to child)
- Method resolution: search order determines which method is called
- IS-A relationship: child “is a kind of” parent

Historical Context:

- Inheritance from Simula (1967) and Smalltalk (1972)
- Single inheritance in most OOP languages; Python supports multiple inheritance
- Method Resolution Order (MRO) uses C3 linearization algorithm

Inheritance Basics

```
class Hand(Deck): # Hand IS-A Deck
    def __init__(self, label=''):
        self.cards = []
        self.label = label
```

Benefits: Code reuse, customization without modifying parent

Method Resolution Order (MRO):

```
type(obj).mro() # List of classes searched
```

Python Philosophy - Duck Typing

Pythonic Rationale:

- “If it walks like a duck and quacks like a duck...”
- Behavior over inheritance
- No interfaces needed - just implement the methods

Python Code:

```
# Python: Duck typing - no interfaces needed
class Duck:
    def quack(self):
        print("Quack!")

class Person:
    def quack(self):
        print("I'm quacking like a duck!")

def make_it_quack(thing):
    thing.quack() # Works on any object with quack()
```

Python Philosophy - Duck Typing (continued)

C Comparison:

```
// C: No polymorphism without function pointers
typedef struct {
    void (*quack)(void*);
} Duck;
// Must explicitly set up vtable
```

Rust Comparison:

```
// Rust: Traits (structural typing)
trait Quack {
    fn quack(&self);
}
struct Duck;
impl Quack for Duck {
    fn quack(&self) { println!("Quack!"); }
}
fn make_it_quack<T: Quack>(thing: T) {
    thing.quack();
}
```

Theory - Structural Typing and Duck Typing

Conceptual Overview:

- Duck typing: “If it walks like a duck and quacks like a duck, it’s a duck”
- Structural typing determines compatibility based on the structure (methods/properties) of a type
- Nominal typing determines compatibility based on explicit declarations (inheritance, implements)

Mathematical Foundations:

- Structural typing: $T_1 \leq T_2$ if T_1 has all members of T_2 with compatible types
- Nominal typing: $T_1 \leq T_2$ only if declared (class T_1 extends T_2)
- Protocol: $P = \{m_1 : \sigma_1, m_2 : \sigma_2, \dots\}$ defines required method signatures
- Interface segregation: prefer small, focused interfaces over large, monolithic ones

Historical Context:

- Duck typing term from “duck test” adage, popularized in Python community
- Structural typing in ML (1970s), Go interfaces (2009), TypeScript (2012)
- Python protocols (PEP 544, 2018) add structural subtyping for type hints
- Interface Segregation Principle (ISP) is one of the SOLID principles (Robert Martin, 2000)

Theory - Design Principles

Conceptual Overview:

- Good class design follows established principles
- HAS-A (composition) is often preferable to IS-A (inheritance)
- Interface and implementation should be separated

Mathematical Foundations:

- Composition: A contains B means A has reference to B
- Inheritance: A extends B means A is subtype of B
- Dependency: A uses B means A calls methods on B

Historical Context:

- Design principles from “Design Patterns” (Gang of Four, 1994)
- “Favor composition over inheritance” is a key guideline
- SOLID principles formalized by Robert Martin

Design Principles

HAS-A vs IS-A: - **HAS-A:** Composition (Rectangle has Point) - **IS-A:** Inheritance (Hand is Deck)

Class Relationships:

Relationship	Description	Example
HAS-A	Contains reference	Rectangle has Point
IS-A	Inheritance	Hand is Deck
Dependency	Uses but doesn't store	Parameters

Python Philosophy - MRO and C3 Linearization

Pythonic Rationale:

- Multiple inheritance is powerful but dangerous
- C3 linearization provides deterministic method resolution
- `super()` follows the MRO chain

Python Code:

```
# Python: Multiple inheritance with MRO
class A:
    def method(self): print("A")
class B(A):
    def method(self): print("B"); super().method()
class C(A):
    def method(self): print("C"); super().method()
class D(B, C):
    pass
D().method() # B -> C -> A (C3 linearization)
D.__mro__    # Shows resolution order
```

Python Philosophy - MRO and C3 Linearization (continued)

C Comparison:

```
// C: No inheritance - composition only  
struct A { int x; };  
struct B { struct A a; int y; };  
// Must manually delegate: b.a.x
```

Rust Comparison:

```
// Rust: No inheritance - traits only  
trait A { fn method_a(&self); }  
trait B: A { fn method_b(&self); }  
// Traits can require other traits (inherit-like)  
// But no data inheritance
```

Theory - Method Resolution Order (MRO)

Conceptual Overview:

- MRO determines the order in which base classes are searched for methods
- The C3 linearization algorithm produces a consistent, monotonic ordering
- Multiple inheritance creates the “diamond problem”: which path to take when a class inherits from two classes with a common ancestor?

Mathematical Foundations:

- C3 linearization: $L[C] = [C] + \text{merge}(L[B_1], L[B_2], \dots, [B_1, B_2, \dots])$
- Merge rule: take first head that appears nowhere in any tail
- Monotonicity: if A precedes B in $L[X]$, then A precedes B in $L[Y]$ where Y inherits from X
- Diamond problem: $D \rightarrow B, C \rightarrow A$; C3 ensures A appears only once

Historical Context:

- C3 linearization developed by Barrett et al. (1996) for Dylan language
- Python adopted C3 in Python 2.3 (2003) to resolve MRO inconsistencies
- Before C3: depth-first search caused subtle bugs in complex hierarchies
- Other languages: Java uses single inheritance (avoids problem); C++ has complex rules

Theory - Interface vs Implementation

Conceptual Overview:

- Interface defines what methods a class provides
- Implementation defines how those methods work internally
- Good design hides implementation details from users

Mathematical Foundations:

- Interface as contract: $I = \{m_1, m_2, \dots, m_n\}$ set of method signatures
- Implementation: concrete code satisfying interface contract
- Abstraction: hide details, expose only interface

Historical Context:

- Interface/implementation separation from abstract data types (1970s)
- Java popularized explicit interface keyword
- Python uses duck typing - interfaces are implicit

Interface vs Implementation

- **Interface:** Methods a class provides (what it does)
- **Implementation:** Internal representation (how it does it)
- **Goal:** Design interfaces that allow implementation changes

Theory - Pure Functions vs Modifiers

Conceptual Overview:

- Pure functions have no side effects and return new values
- Modifiers change object state in place
- Pure functions are easier to test, debug, and reason about

Mathematical Foundations:

- Pure function: $f : X \rightarrow Y$ with no side effects
- Referential transparency: $f(x) = f(x)$ always (same input, same output)
- Modifier: $f : X \times S \rightarrow S'$ modifies state S

Historical Context:

- Pure functions from functional programming (Lisp, Haskell)
- Side-effect-free code is easier to reason about mathematically
- Modern best practice favors immutability and pure functions

Pure Functions vs Modifiers

```
# Pure (preferred)
def add_time(t1, t2):
    return int_to_time(t1.time_to_int() + t2.time_to_int())

# Modifier (avoid)
def increment(time, seconds):
    time.second += seconds # Modifies argument
```

Rule: Prefer pure functions - easier to test and debug

Theory - Liskov Substitution Principle

Conceptual Overview:

- Subtypes must be substitutable for their base types
- Child class should extend parent behavior, not contradict it
- One of the SOLID principles of object-oriented design

Mathematical Foundations:

- Subtyping: if $S \subseteq T$, then S can be used anywhere T is expected
- Behavioral subtyping: S must preserve all invariants of T
- Precondition weakening, postcondition strengthening

Historical Context:

- Barbara Liskov introduced the principle (1987)
- Part of SOLID principles formalized by Robert Martin
- Ensures polymorphism works correctly with inheritance

Liskov Substitution Principle

Override methods with matching interfaces so child objects can replace parent objects.

Theory - Copying Objects

Conceptual Overview:

- Shallow copy creates new object but shares references to embedded objects
- Deep copy recursively copies all nested objects
- Understanding copy semantics prevents unintended sharing bugs

Mathematical Foundations:

- Shallow copy: $copy(o) = o'$ where $o'.a = o.a$ (same references)
- Deep copy: $deepcopy(o)$ recursively copies all nested objects
- Object graph: tree of references from root object

Historical Context:

- Copy semantics important since reference-based languages (Lisp, Java)
- Prototype pattern uses cloning to create new objects
- Python's copy module provides both shallow and deep copy

Copying Objects

```
import copy

p2 = copy.copy(p1)           # Shallow copy
box3 = copy.deepcopy(box)    # Deep copy
```

Type	Copies	Embedded objects
Shallow	Object	Shared (same ref)
Deep	Object + embedded	Separate copies

Python Philosophy - Object Model Internals

Pythonic Rationale:

- Everything is an object with `__dict__`
- Attribute lookup is dynamic dictionary access
- Flexibility over memory efficiency

Python Code:

```
# Python: Dynamic object model
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

p = Point(1, 2)
p.z = 3           # Add attribute at runtime!
p.__dict__        # {'x': 1, 'y': 2, 'z': 3}
```

Python Philosophy - Object Model Internals (continued)

C Comparison:

```
// C: Structs have fixed layout at compile time  
struct Point {  
    double x, y; // Fixed offset in memory  
};  
// Cannot add fields at runtime  
// sizeof(Point) is constant
```

Rust Comparison:

```
// Rust: Structs with compile-time layout  
struct Point { x: f64, y: f64 }  
// Memory layout determined at compile time  
// No runtime attribute addition  
// Use HashMap for dynamic fields
```

Theory - Object Memory Layout

Conceptual Overview:

- Python objects store attributes in a dictionary (`__dict__`) by default
- `__slots__` declares fixed attributes, eliminating the dictionary and saving memory
- Trade-off: slots reduce flexibility but improve memory efficiency and attribute access speed

Mathematical Foundations:

- Dict-based: $\text{memory}(o) = \text{overhead} + \text{dict_overhead} + n \cdot \text{entry_size}$
- Slots-based: $\text{memory}(o) = \text{overhead} + n \cdot \text{slot_size}$ (no dict overhead)
- Attribute access: `dict[key]` is $O(1)$ average, $O(n)$ worst case (hash collision)
- Slots access: direct offset, $O(1)$ guaranteed, no hash computation

Historical Context:

- Python objects have used dict-based attributes since the beginning (1991)
- `__slots__` added in Python 2.2 (2001) for memory-constrained applications
- Trade-off mirrors static vs dynamic typing: flexibility vs efficiency
- Modern alternatives: `@dataclass`, `namedtuple` offer different trade-offs

Subsection 3

Polymorphism and Type-Based Dispatch

Theory - Polymorphism

Conceptual Overview:

- Polymorphism allows the same operation to work on different types
- Duck typing: if it has the right methods, it works
- Enables flexible, reusable code without explicit interfaces

Mathematical Foundations:

- Polymorphism: $f : \forall T \in \mathcal{T}. T \rightarrow T'$ works for multiple types
- Duck typing: $f(x)$ works if x has required methods
- Ad-hoc polymorphism: different implementations for different types

Historical Context:

- Polymorphism from Greek “many forms”
- Duck typing term from “duck test” adage
- Python’s polymorphism is implicit through duck typing

Polymorphism

Definition: Same operation works correctly on different types

```
# Works with any objects that define __add__  
total = sum([t1, t2, t3])
```

Duck Typing: “If it walks like a duck and quacks like a duck...”

Theory - Type-Based Dispatch

Conceptual Overview:

- Type-based dispatch handles different types in the same method
- Uses `isinstance()` to check argument types
- Enables methods to behave differently based on argument types

Mathematical Foundations:

- Dispatch: $f(x) = \begin{cases} f_1(x) & \text{if } x \in T_1 \\ f_2(x) & \text{if } x \in T_2 \end{cases}$
- Type predicate: `isinstance(x, T)` checks $x \in T$
- Multiple dispatch: behavior depends on multiple argument types

Historical Context:

- Type-based dispatch from method overloading in OOP
- Python uses single dispatch (based on first argument type)
- `functools.singledispatch` provides cleaner dispatch mechanism

Type-Based Dispatch

Handle different types in same method:

```
def __add__(self, other):  
    if isinstance(other, Time):  
        return self.add_time(other)  
    else:  
        return self.increment(other)
```

Use when: Method needs to handle multiple argument types

Section 4

Part III: Dynamic Typing and Debugging

Subsection 1

Dynamic Typing

Theory - Dynamic Typing

Conceptual Overview:

- Dynamic typing means variables have no type; values have types
- A variable is just a name bound to an object in memory
- The same variable can reference objects of different types over time

Mathematical Foundations:

- Type as property of value: $\text{type}(v) \in \mathcal{T}$ where v is a value
- Variable as binding: $\text{env}(x) \rightarrow v$ (environment maps names to values)
- Type checking at runtime: $\text{valid}(op, \text{type}(v_1), \text{type}(v_2))$ evaluated during execution

Historical Context:

- Dynamic typing originated in Lisp (1958)
- Python adopted dynamic typing for flexibility and ease of use
- Contrast with static typing (C, Java, Rust) where types are checked at compile time

Core Concepts

Values have types, variables don't:

```
type(42)          # <class 'int'>
type(42.0)        # <class 'float'>
```

Duck Typing: “If it walks like a duck...”

```
def print_length(obj):
    print(len(obj))  # Works with str, list, dict, etc.
```

isinstance() for type checking:

```
isinstance(x, (int, float))  # int or float
```


Subsection 2

Error Types

Theory - Error Types

Conceptual Overview:

- Errors are classified by when they are detected in the program lifecycle
- Syntax errors occur before execution during parsing
- Runtime errors occur during execution and raise exceptions
- Semantic errors produce incorrect results without raising errors

Mathematical Foundations:

- Syntax: $\text{parse}(\text{code}) \rightarrow \text{AST or SyntaxError}$
- Runtime: $\text{eval}(\text{AST}, \text{env}) \rightarrow \text{value or Exception}$
- Semantic: $\text{eval}(\text{AST}, \text{env}) = v$ but $v \neq v_{\text{expected}}$

Historical Context:

- Error classification formalized in early compiler theory (1960s)
- Exception handling mechanisms evolved from PL/I and CLU
- Semantic errors remain the hardest to detect and debug

Three Types of Errors

Type	When	Example
Syntax	Before run	Missing colon, unmatched parens
Runtime	During run	<code>1/0</code> , <code>int('abc')</code>
Semantic	Wrong output	Wrong formula, logic error

Theory - Syntax Errors

Conceptual Overview:

- Syntax errors violate the grammatical rules of the programming language
- Detected by the parser before any code executes
- Common causes: missing punctuation, mismatched brackets, invalid keywords

Mathematical Foundations:

- Grammar: $G = (N, \Sigma, P, S)$ defines valid program structure
- Parsing: $\text{parse}(\text{tokens}) \rightarrow \text{AST or error}$
- A program is syntactically valid if $\text{code} \in L(G)$ (language generated by grammar)

Historical Context:

- Parser error detection from early compiler design (1950s)
- Python's parser provides helpful error messages with line numbers
- Modern IDEs highlight syntax errors in real-time

Syntax Errors

- Violations of program structure rules
- Detected at parse time, before program runs
- Program won't start
- **Examples:** unmatched parentheses, invalid tokens, missing colons

Theory - Runtime Errors

Conceptual Overview:

- Runtime errors occur during program execution
- Python raises exceptions when operations cannot be completed
- Exceptions can be caught and handled with try/except blocks

Mathematical Foundations:

- Exception as signal: $\text{eval}(e) \rightarrow \text{raise}(E)$ where E is exception type
- Exception propagation: call stack unwinds until handler found
- Exception handler: $\text{try } e \text{ catch } E : h$ evaluates h if e raises E

Historical Context:

- Exception handling introduced in PL/I (1964) and CLU (1974)
- Modern languages use try/catch or try/except patterns
- Python's exception hierarchy: BaseException \rightarrow Exception \rightarrow specific types

Runtime Errors (Exceptions)

Exception	Cause
NameError	Using undefined variable
TypeError	Wrong type for operation
ValueError	Correct type, invalid value
KeyError	Dictionary key doesn't exist
IndexError	Index out of range
AttributeError	Non-existent attribute/method

Theory - Semantic Errors

Conceptual Overview:

- Semantic errors occur when code is syntactically correct but logically wrong
- The program runs without errors but produces incorrect results
- These are the hardest bugs to find because there is no error message

Mathematical Foundations:

- Correctness specification: $\forall x : f(x) = f_{\text{spec}}(x)$
- Semantic error: $\exists x : f(x) \neq f_{\text{spec}}(x)$ but no exception raised
- Formal verification can prove correctness but is rarely used in practice

Historical Context:

- Semantic errors are as old as programming itself
- Testing and debugging techniques evolved to find these errors
- Formal methods (Hoare logic, model checking) can prove correctness

Semantic Errors

- Program runs but produces wrong results
- No error messages
- **Hardest to debug** - incorrect mental model

Theory - Exception Handling

Conceptual Overview:

- Exception handling separates error detection from error recovery
- try block contains code that might raise exceptions
- except block handles specific exception types
- finally block always executes, used for cleanup

Mathematical Foundations:

- Exception semantics: $\text{try } e_1 \text{ except } E : e_2 = \begin{cases} v & \text{if } e_1 \rightarrow v \\ e_2 & \text{if } e_1 \rightarrow \text{raise}(E) \end{cases}$
- Exception propagation: unhandled exceptions bubble up the call stack
- Resource cleanup: finally ensures cleanup runs regardless of outcome

Historical Context:

- Exception handling from PL/I (1964) and CLU (1974)
- try/except/finally pattern standardized across languages
- Context managers (with statement) provide cleaner resource management

Exception Handling

```
try:
    fin = open('file.txt')
    data = fin.read()
except FileNotFoundError:
    print('File not found')
except Exception as e:
    print(f'Error: {e}')
finally:
    fin.close() # Always runs
```

Python Philosophy - EAFP vs LBYL

Pythonic Rationale:

- EAFP: “Easier to Ask Forgiveness than Permission”
- LBYL: “Look Before You Leap” (avoid in Python)
- Exceptions are cheap, checking is verbose

Python Code:

```
# Python: EAFP style (preferred)
```

```
try:
```

```
    with open("file.txt") as f:  
        data = f.read()
```

```
except FileNotFoundError:
```

```
    data = None
```

```
# LBYL style (avoid in Python)
```

```
import os
```

```
if os.path.exists("file.txt"):
```

```
    with open("file.txt") as f:  
        data = f.read()
```

```
else:
```

```
    data = None
```

Python Philosophy - EAFP vs LBYL (continued)

C Comparison:

```
// C: LBYL style with error codes
int result = open_file("data.txt");
if (result == -1) {
    // Handle error
    return ERROR_CODE;
}
// Must check every operation
```

Rust Comparison:

```
// Rust: Result type (no exceptions)
match file.open("data.txt") {
    Ok(mut f) => { /* use file */ },
    Err(e) => { /* handle error */ },
}
// Errors are values, not control flow
```

Theory - Exception Implementation Strategies

Conceptual Overview:

- Exceptions provide structured error handling separate from normal control flow
- Zero-cost exception model: no overhead when exceptions are not thrown
- Exceptions vs error codes: exceptions propagate automatically through call stack

Mathematical Foundations:

- Exception semantics: $\text{try } e_1 \text{ catch } E : e_2 = \begin{cases} v & \text{if } e_1 \rightarrow v \\ e_2 & \text{if } e_1 \rightarrow \text{raise}(E) \end{cases}$
- Zero-cost: $\text{cost}_{no_exception} = O(0)$ table lookup only;
 $\text{cost}_{exception} = O(\text{stack_depth})$
- Unwinding: stack frames popped until matching handler found

Historical Context:

- Exceptions introduced in PL/I (1964) and CLU (1974)
- C++ exceptions (1989) use zero-cost model with table-based unwinding
- Java (1995) introduced checked exceptions (controversial feature)
- Python uses unchecked exceptions; EAFP style encourages exception use

Subsection 3

Debugging Strategies

Theory - Debugging Strategies

Conceptual Overview:

- Debugging is the process of finding and fixing errors in programs
- Systematic approaches reduce debugging time significantly
- Different error types require different debugging strategies

Mathematical Foundations:

- Bug localization: find smallest code segment S' such that $\text{bug}(S')$
- Binary search: $\log_2(n)$ probes to isolate bug in n -line program
- Hypothesis testing: propose cause, design experiment, verify or refute

Historical Context:

- “Debugging” term attributed to Grace Hopper finding a moth in a relay (1947)
- Systematic debugging methods developed alongside software engineering
- Modern debuggers (gdb, pdb) allow step-by-step execution inspection

The Five R's

- ➊ **Reading** - Examine code carefully
- ➋ **Running** - Experiment, display values
- ➌ **Ruminating** - Think about error type
- ➍ **Rubberducking** - Explain problem aloud
- ➎ **Retreating** - Undo to working code

Theory - Bisection Method

Conceptual Overview:

- Bisection method uses binary search to isolate bugs
- Test the middle of the code to determine which half contains the bug
- Repeat until bug is found in a small enough section

Mathematical Foundations:

- Binary search: $\log_2(n)$ tests to find bug in n -line program
- Correctness: if bug at line k , check at m narrows to $[1, m]$ or $[m + 1, n]$
- Optimal strategy for finding a single bug with yes/no tests

Historical Context:

- Binary search algorithm from computer science fundamentals
- Applied to debugging in “The Practice of Programming” (Kernighan, Pike)
- Effective when you can test intermediate states

Check middle of code → isolate to first or second half → repeat

Theory - Assertions

Conceptual Overview:

- Assertions are statements that check conditions at runtime
- Used to verify assumptions and catch bugs early
- Failed assertions raise `AssertionError` and stop the program

Mathematical Foundations:

- Assertion: `assert(P)` checks predicate P at program point
- Precondition: $P(x) \Rightarrow f(x)$ produces correct result
- Postcondition: $f(x) \Rightarrow Q(f(x))$ ensures output satisfies constraint
- Invariant: condition that holds before and after each loop iteration

Historical Context:

- Assertions formalized in Hoare logic (1969) for program verification
- Design by Contract methodology from Eiffel language (Bertrand Meyer, 1986)
- Python assertions can be disabled with `-O` flag for production

```
assert valid_time(t1) and valid_time(t2)
```

Theory - Print Debugging

Conceptual Overview:

- Print debugging displays variable values at key program points
- Simple but effective technique for understanding program flow
- Helps identify where values diverge from expectations

Mathematical Foundations:

- Trace: sequence of states (s_0, s_1, \dots, s_n) through execution
- Observation: $\text{print}(x)$ reveals $\text{env}(x)$ at current point
- Comparison: expected vs actual values identifies error location

Historical Context:

- Oldest debugging technique, available in all languages
- `printf` debugging named after C's `printf` function
- Still widely used despite availability of interactive debuggers

Print Debugging

```
def foo():  
    print("entering function foo")  
    print(f"x = {x}, type = {type(x)}")  
    # function body
```

Tip: Display values at key points

Theory - IPython Debugging Tools

Conceptual Overview:

- IPython provides powerful debugging and profiling tools
- Postmortem debugging allows inspecting state after an exception
- Profiling identifies performance bottlenecks

Mathematical Foundations:

- Debugger: inspect call stack and variable state at each frame
- Profiling: measure time $T(f)$ spent in each function f
- Benchmarking: statistical timing with multiple runs for accuracy

Historical Context:

- Interactive debuggers since the 1970s (gdb, dbx)
- IPython's magic commands extend Python with debugging features
- Jupyter notebooks integrate debugging with interactive computing

IPython Debugging Tools

```
%debug      # Postmortem debugger after exception  
%prun       # Profiler for performance analysis  
%timeit     # Benchmark code execution
```

Subsection 4

Algorithm Complexity

Theory - Big-O Notation

Conceptual Overview:

- Big-O notation describes how algorithm runtime grows with input size
- Focus on worst-case or average-case asymptotic behavior
- Allows comparing algorithms without implementation details

Mathematical Foundations:

- Definition: $f(n) = O(g(n))$ if $\exists c > 0, n_0 : \forall n \geq n_0, f(n) \leq c \cdot g(n)$
- Common classes: $O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(2^n)$
- Hierarchy: polynomial algorithms are tractable; exponential are not

Historical Context:

- Big-O notation from number theory (Bachmann, 1894; Landau, 1909)
- Applied to algorithm analysis by Donald Knuth (1970s)
- Fundamental to computer science curriculum and algorithm design

Big-O Notation

Definition: $f(n) = O(g(n))$ if $f(n) \leq c \cdot g(n)$ for $n \geq n_0$

Order	Name	Example
$O(1)$	Constant	Indexing
$O(\log n)$	Logarithmic	Binary search
$O(n)$	Linear	Simple loop
$O(n^2)$	Quadratic	Nested loops

Search comparison: - Linear: $O(n)$ - Binary: $O(\log n)$ (requires sorted data) - Hashtable: $O(1)$ (dict lookup)

Section 5

Part IV: Numerical Computing

Subsection 1

NumPy Arrays

Theory - NumPy Arrays

Conceptual Overview:

- NumPy provides efficient n-dimensional array objects for numerical computing
- Arrays are homogeneous (all elements same type) and stored contiguously in memory
- Vectorized operations eliminate Python loops for dramatic performance gains

Mathematical Foundations:

- Array as tensor: $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \dots \times n_d}$
- Memory layout: row-major (C) vs column-major (Fortran)
- Broadcasting rules: dimensions align from right, size 1 expands

Historical Context:

- NumPy created by Travis Oliphant in 2006, building on Numeric (1995) and NumArray
- Foundation of the Python scientific computing ecosystem
- Enables Python to compete with MATLAB and R for numerical work

Creation and Attributes

```
import numpy as np

np.array([1, 2, 3])      # From list
np.zeros((2, 3))         # Zeros
np.ones(4)               # Ones
np.arange(0, 10, 0.5)    # Like range
np.linspace(0, 10, 100)  # 100 points
```

Key attributes:

```
data.ndim    # Dimensions
data.shape   # (3, 2)
data.size    # Total elements
data.dtype   # int64, float64
```

Python Philosophy - Why NumPy?

Pythonic Rationale:

- “Premature optimization is the root of all evil”
- But when you need speed, vectorize!
- NumPy: C-speed with Python convenience

Python Code:

```
# Python: NumPy vectorization
import numpy as np
a = np.array([1, 2, 3, 4, 5])
b = np.array([10, 20, 30, 40, 50])
c = a + b # Element-wise, no loop!
# 100x faster than Python loop
```

Python Philosophy - Why NumPy? (continued)

C Comparison:

```
// C: Manual loops for array operations
double a[1000], b[1000], c[1000];
for (int i = 0; i < 1000; i++) {
    c[i] = a[i] + b[i]; // Loop overhead
}
// Fast but verbose, no bounds checking
```

Rust Comparison:

```
// Rust: ndarray crate for numerical computing
use ndarray::Array1;
let a = Array1::<f64>::zeros(1000);
let b = Array1::<f64>::zeros(1000);
let c = &a + &b; // Element-wise, no explicit loop
// Bounds checking can be disabled with unsafe
```

Theory - Vectorization and SIMD

Conceptual Overview:

- Vectorization applies the same operation to multiple data elements simultaneously
- SIMD (Single Instruction, Multiple Data) instructions process 4-8 elements per CPU cycle
- Contiguous memory layout enables efficient cache utilization and vectorized access

Mathematical Foundations:

- SIMD: $\text{vadd}(\mathbf{a}, \mathbf{b}) = (a_1 + b_1, a_2 + b_2, \dots, a_n + b_n)$ in parallel
- Speedup: $\frac{T_{\text{scalar}}}{T_{\text{vector}}} \approx \text{vector width} \times \text{cache efficiency}$
- Loop unrolling: execute k iterations per branch to reduce overhead
- Cache line: 64 bytes typically; contiguous access maximizes hits

Historical Context:

- Vector processors emerged in supercomputers (Cray-1, 1976)
- SIMD extensions to CPUs: MMX (1997), SSE (1999), AVX (2011), AVX-512 (2016)
- NumPy leverages BLAS/LAPACK which use SIMD internally
- GPU computing (CUDA, 2007) provides massive parallelism for vectorized operations

Indexing and Broadcasting

```
# Slicing  
A[:, 1]      # Second column  
A[1, :]     # Second row  
A[A > 0.5]  # Boolean indexing  
  
# Broadcasting  
x = np.array([[1, 2], [3, 4]]) # (2, 2)  
z = np.array([[2, 4]])         # (1, 2) → broadcasts to (2, 2)  
x / z # Works!
```

Python Philosophy - Broadcasting Design

Pythonic Rationale:

- Implicit array expansion for element-wise ops
- “Shape compatibility” - dimensions align right-to-left
- Reduces memory allocation and code complexity

Python Code:

Python: Broadcasting automatically expands arrays

```
import numpy as np
```

```
a = np.array([[1, 2], [3, 4]]) # (2, 2)
```

```
b = np.array([10, 20]) # (1, 2) → broadcasts to (2, 2)
```

```
c = a + b # Works!
```

Python Philosophy - Broadcasting Design (continued)

C Comparison:

```
// C: Must manually handle different shapes  
double a[3][4], b[4]; // b broadcast across rows  
for (int i = 0; i < 3; i++) {  
    for (int j = 0; j < 4; j++) {  
        a[i][j] = a[i][j] + b[j]; // Explicit loops  
    }  
}
```

Rust Comparison:

```
// Rust: ndarray supports broadcasting  
use ndarray::{Array2, Array1, s};  
let a = Array2::::zeros((3, 4));  
let b = Array1::::zeros(4);  
let c = &a + &b.broadcast((3, 4)).unwrap();  
// Explicit broadcasting method
```


Theory - Broadcasting Semantics

Conceptual Overview:

- Broadcasting allows operations on arrays of different shapes by virtually expanding smaller arrays
- Dimensions are compared right-to-left; dimensions of size 1 can expand to any size
- No actual data is copied during broadcasting - it creates views with adjusted strides

Mathematical Foundations:

- Broadcasting rule: dimensions align from right, size 1 expands to match larger dimension
- Shape inference: $(n_1, \dots, n_k) \odot (m_1, \dots, m_l) \rightarrow (\max(n_i, m_i))$
- Stride manipulation: $\text{stride}_i = 0$ for broadcast dimensions (no memory movement)
- Memory efficiency: $O(1)$ additional memory regardless of expanded size

Historical Context:

- Broadcasting concept from APL (1964) and MATLAB (1984)
- NumPy popularized broadcasting for scientific Python (2006)
- Enables concise code without explicit loops or replication
- Modern frameworks (TensorFlow, PyTorch) adopted NumPy's broadcasting rules

Subsection 2

Vectorized Operations

Theory - Vectorized Operations

Conceptual Overview:

- Vectorization applies operations to entire arrays without explicit loops
- Implemented in compiled C code for orders of magnitude speedup
- Universal functions (ufuncs) operate elementwise on arrays

Mathematical Foundations:

- Elementwise: $(f \circ \mathbf{x})_i = f(x_i)$
- Hadamard product: $(\mathbf{a} \odot \mathbf{b})_i = a_i \cdot b_i$
- Aggregation: $\text{reduce}(f, \mathbf{x}) = f(x_1, f(x_2, \dots f(x_{n-1}, x_n)))$

Historical Context:

- Vectorization concept from APL (1964) and MATLAB (1984)
- NumPy's ufuncs inspired by NumPy's predecessors Numeric and NumArray
- Key insight: move loops from Python (slow) to C (fast)

Elementwise and Aggregates

```
x + y, x - y, x * y, x / y # Elementwise
```

```
# Universal functions (ufuncs)
```

```
np.cos(x), np.exp(x), np.sqrt(x)
```

```
# Aggregates
```

```
np.mean(data), np.std(data)
```

```
data.sum(axis=0) # Along axis
```

Matrix Operations - Matrix Multiplication

Python Code:

```
C = A @ B          # Matrix multiplication (Python 3.5+)
C = np.dot(A, B)    # Equivalent
```

Mathematical Definition:

$$C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$$

Where $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$, result $C \in \mathbb{R}^{m \times p}$

Matrix Operations - Dot Product (Inner Product)

Python Code:

```
result = np.inner(x, y)      # Dot product
```

Mathematical Definition:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{i=1}^n x_i y_i$$

Where $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$, result is a scalar

Matrix Operations - Outer Product

Python Code:

```
M = np.outer(x, y)      # Outer product
```

Mathematical Definition:

$$(\mathbf{x} \otimes \mathbf{y})_{ij} = x_i \cdot y_j$$

Where $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^n$, result $M \in \mathbb{R}^{m \times n}$

Matrix Operations - Cross Product

Python Code:

```
c = np.cross(a, b)      # Cross product (3D vectors)
```

Mathematical Definition:

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = (a_2b_3 - a_3b_2, a_3b_1 - a_1b_3, a_1b_2 - a_2b_1)$$

Where $\mathbf{a}, \mathbf{b} \in \mathbb{R}^3$, result $\mathbf{c} \in \mathbb{R}^3$ is perpendicular to both \mathbf{a} and \mathbf{b}

Python Philosophy - Memory Layout

Pythonic Rationale:

- Row-major by default (C-order)
- Matches how we think about 2D arrays
- But can choose column-major (Fortran-order)

Python Code:

```
# Python: Choose memory layout for performance  
import numpy as np  
row_major = np.zeros((3, 4))          # C-order (default)  
col_major = np.zeros((3, 4), order='F') # Fortran-order  
# Row-major: faster row access  
# Column-major: faster column access
```

Python Philosophy - Memory Layout (continued)

C Comparison:

```
// C: Row-major layout  
int arr[2][3] = {{1,2,3}, {4,5,6}};  
// Memory: [1,2,3,4,5,6]  
// arr[i][j] at offset: i*3 + j
```

Rust Comparison:

```
// Rust: ndarray supports both layouts  
use ndarray::{Array2, Order};  
let row_major = Array2::::zeros((3, 4));  
// Row-major by default  
let col_major = Array2::::zeros((3, 4).f());  
// Fortran (column-major) order
```

Theory - Array Memory Layout

Conceptual Overview:

- Row-major (C order) stores elements row-by-row; column-major (Fortran order) stores column-by-column
- Memory layout affects cache performance: contiguous access is faster than strided access
- Choosing the right layout can dramatically impact numerical computation speed

Mathematical Foundations:

- Row-major: $\text{offset}(i, j) = i \cdot n_{\text{cols}} + j$
- Column-major: $\text{offset}(i, j) = j \cdot n_{\text{rows}} + i$
- Cache hit rate: higher when accessing contiguous memory locations
- Stride: step size in memory between consecutive elements in each dimension

Historical Context:

- Row-major from C language (1972), matches how humans visualize 2D arrays
- Column-major from Fortran (1957), adopted by BLAS/LAPACK libraries
- NumPy supports both layouts; default is row-major (C order)
- Modern CPUs have cache lines (~64 bytes); contiguous access maximizes cache utilization

Subsection 3

SciPy for Numerical Methods

Theory - Numerical Methods

Conceptual Overview:

- Numerical methods solve mathematical problems that lack analytical solutions
- SciPy provides optimized implementations of classical algorithms
- Key areas: linear algebra, optimization, integration, differential equations

Mathematical Foundations:

- Linear system: $A\mathbf{x} = \mathbf{b}$ where $A \in \mathbb{R}^{n \times n}$
- Optimization: $\min_{\mathbf{x}} f(\mathbf{x})$ or find $\mathbf{x}^* : f(\mathbf{x}^*) = 0$
- Quadrature: $I = \int_a^b f(x)dx \approx \sum_{i=1}^n w_i f(x_i)$

Historical Context:

- Roots in numerical analysis from 1940s-50s (von Neumann, Wilkinson)
- LAPACK (1992) provides core linear algebra routines
- SciPy wraps Fortran libraries (LAPACK, MINPACK, QUADPACK) for Python

Linear Systems

Solve: $A\mathbf{x} = \mathbf{b}$

```
from scipy import linalg as la
```

```
x = la.solve(A, b)
```

```
P, L, U = la.lu(A) # LU decomposition
```

```
sol = la.lstsq(A, b) # Least squares:  $\min \|A\mathbf{x} - \mathbf{b}\|^2$ 
```

Eigenvalues: $A\mathbf{v} = \lambda\mathbf{v}$

```
evals, evecs = la.eig(A)
```

Optimization

```
from scipy import optimize
```

Root finding

```
optimize.brentq(f, a, b)
```

Find $f(x) = 0$ in $[a, b]$

```
optimize.newton(f, x0)
```

Newton: $x_{n+1} = x_n - f(x_n)/f'(x_n)$

Minimization

```
optimize.minimize(f, x0, method='BFGS')
```

```
optimize.minimize(f, x0, bounds=[(lb, ub)]) # Constrained
```

Interpolation and Integration

Interpolation: Find $p(x)$ such that $p(x_i) = y_i$

```
from scipy import interpolate
```

```
f = interpolate.interpld(x, y, kind='cubic')
```

Integration: $I = \int_a^b f(x)dx$

```
from scipy import integrate
```

```
val, err = integrate.quad(f, a, b)  # Gaussian quadrature  
integrate.trapz(y, x)              # Trapezoid rule  
integrate.simps(y, x)              # Simpson's rule
```


ODE Solving

First-order system: $\frac{d\mathbf{y}}{dt} = f(t, \mathbf{y})$

```
from scipy import integrate
```

```
def f(y, t):  
    return [dy1_dt, dy2_dt]
```

```
y = integrate.odeint(f, y0, t)
```

Euler method: $\mathbf{y}_{n+1} = \mathbf{y}_n + h \cdot f(t_n, \mathbf{y}_n)$

Subsection 4

SymPy Symbolic Computation

Theory - Symbolic Computation

Conceptual Overview:

- Symbolic computation manipulates mathematical expressions exactly
- Unlike numerical methods, symbols remain unevaluated until needed
- Enables exact differentiation, integration, and equation solving

Mathematical Foundations:

- Symbolic differentiation: $\frac{d}{dx}f(x)$ via chain rule application
- Symbolic integration: $\int f(x)dx$ using pattern matching and rules
- Algebraic simplification: canonical form reduction

Historical Context:

- Symbolic math systems: Macsyma (1968), Mathematica (1988), Maple (1988)
- SymPy started in 2007 by Ondřej Čertík
- Pure Python implementation makes it portable and embeddable

Calculus and Equations

```
import sympy

x, y = sympy.symbols('x y')

# Calculus
sympy.diff(expr, x)                # Derivative
sympy.integrate(expr, (x, 0, pi))  # Definite integral
sympy.series(sympy.sin(x), x)      # Taylor series
sympy.limit(sympy.sin(x)/x, x, 0)  # Limit

# Equation solving
sympy.solve(x**2 + 2*x - 3)         # Roots: [-3, 1]
sympy.solve([x + 2*y - 1, x - y + 1], [x, y]) # System
```

Subsection 5

Matplotlib Plotting

Theory - Data Visualization

Conceptual Overview:

- Visualization transforms data into graphical representations
- Effective plots reveal patterns, trends, and outliers invisible in raw data
- Matplotlib provides publication-quality figures with fine-grained control

Mathematical Foundations:

- Coordinate transformation: data coordinates \rightarrow pixel coordinates
- Rendering pipeline: data \rightarrow aesthetic mapping \rightarrow geometric objects \rightarrow pixels
- Color mapping: $c : [v_{min}, v_{max}] \rightarrow [\text{colorspace}]$

Historical Context:

- Matplotlib created by John Hunter in 2003, inspired by MATLAB plotting
- Became the foundation of Python's visualization ecosystem
- Seaborn, pandas plotting, and many others build on matplotlib

Basic Plots

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot(x, y, label="curve")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.legend()
plt.show()
```

Multiple subplots:

```
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(8, 6))
```

Customization

```
ax.set_xlim(-5, 35)
ax.set_xscale('log')
ax.axis('equal')
ax.grid(color="grey", linestyle=':')
```

Specialized plots

```
ax.scatter(x, y, c=colors)
ax.hist(data, bins=20)
ax.contour(X, Y, Z, levels=10)
```

LaTeX support

```
ax.set_title(r"$E = mc^2$")
```


Section 6

Part V: Data Analysis and Statistics

Subsection 1

Pandas

Theory - Series and DataFrame

Conceptual Overview:

- Series is a one-dimensional labeled array capable of holding any data type
- DataFrame is a two-dimensional labeled data structure with columns of potentially different types
- Both provide powerful indexing, alignment, and manipulation capabilities

Mathematical Foundations:

- Series: $S : I \rightarrow V$ where I is index set and V is value space
- DataFrame: $DF : I \times C \rightarrow V$ where C is column set
- Alignment: operations match on index labels, not positions

Historical Context:

- Pandas created by Wes McKinney in 2008 while at AQR Capital Management
- Named from “panel data” - econometrics term for multidimensional data
- Became essential tool for data science, inspired by R’s `data.frame`

Series and DataFrame

```
import pandas as pd

# Series (1D)
s = pd.Series([909976, 8615246],
              name="Population",
              index=["Stockholm", "London"])

# DataFrame (2D)
df = pd.DataFrame({"Population": [909976, 8615246],
                  "State": ["Sweden", "UK"]},
                  index=["Stockholm", "London"])
```

Python Philosophy - Pandas Design

Pythonic Rationale:

- Labeled arrays - index matters as much as data
- “Align first, compute second” - automatic alignment
- DataFrame = spreadsheet in code

Python Code:

```
# Python: Labeled arrays with automatic alignment
import pandas as pd
df = pd.DataFrame({'a': [1, 2, 3]}, index=['x', 'y', 'z'])
s = pd.Series([10, 20, 30], index=['y', 'z', 'x'])
result = df['a'] + s # Aligns by index!
# x: 1+30=31, y: 2+10=12, z: 3+20=23
```

Python Philosophy - Pandas Design (continued)

C Comparison:

```
// C: Unlabeled arrays, manual bookkeeping
struct DataFrame {
    char **index;           // Row labels
    char **columns;        // Column labels
    double **data;         // 2D array
    int nrows, ncols;
};
// No automatic alignment
// Must manually match indices
```

Rust Comparison:

```
// Rust: Polars for dataframes (faster than Pandas)
use polars::prelude::*;
let df = DataFrame::new(vec![
    Series::new("a", &[1, 2, 3]),
    Series::new("b", &[4, 5, 6]),
]);
// Lazy evaluation for optimization
```

Theory - Pandas Design and Relational Algebra

Conceptual Overview:

- Pandas DataFrames implement relational algebra operations from database theory
- Selection filters rows, projection selects columns, join combines tables
- Columnar storage enables efficient operations on individual columns

Mathematical Foundations:

- Selection: $\sigma_{\phi}(R) = \{t \in R : \phi(t)\}$ filters rows satisfying predicate ϕ
- Projection: $\pi_{A_1, \dots, A_n}(R) = \{t[A_1, \dots, A_n] : t \in R\}$ extracts columns
- Join: $R \bowtie_{\theta} S = \{(r, s) : r \in R, s \in S, \theta(r, s)\}$ combines related rows
- Query optimization: reorder operations to minimize intermediate results

Historical Context:

- Relational model introduced by Codd (1970) at IBM
- SQL standardized relational operations for databases (1986)
- Pandas brought relational operations to Python (2008)
- Modern systems (Polars, DuckDB) use columnar storage and lazy evaluation

Theory - Data Manipulation

Conceptual Overview:

- Data manipulation transforms raw data into analysis-ready formats
- Key operations: filtering, aggregation, transformation, and merging
- Pandas provides vectorized operations that avoid explicit loops

Mathematical Foundations:

- Filter: $DF' = \{r \in DF : P(r)\}$ where P is predicate
- Aggregation: $g(G) = f(\{r \in G\})$ for groups G
- Join: $DF_1 \bowtie_{\theta} DF_2 = \{(r_1, r_2) : \theta(r_1, r_2)\}$

Historical Context:

- SQL (1974) established relational algebra operations
- Pandas brought SQL-like operations to Python
- split-apply-combine pattern formalized by Hadley Wickham (2011)

Data Manipulation

```
# Access
df["Population"]           # Column
df.loc["Stockholm"]       # Row by label
df.iloc[0]                 # Row by position

# Transform
df["new"] = df.col.apply(lambda x: x * 2)
df.groupby("State").sum()
df["col"].value_counts()

# Time series
df.resample("H").mean()
```

Subsection 2

Statistics with SciPy

Theory - Random Variables

Conceptual Overview:

- A random variable maps outcomes from a probability space to real numbers
- Discrete RVs take countable values; continuous RVs take values in intervals
- SciPy's stats module provides 100+ distributions with consistent interface

Mathematical Foundations:

- CDF: $F(x) = P(X \leq x)$
- PDF (continuous): $f(x) = \frac{d}{dx}F(x)$, $\int_{-\infty}^{\infty} f(x)dx = 1$
- PMF (discrete): $p(x) = P(X = x)$
- Quantile: $F^{-1}(p) = \inf\{x : F(x) \geq p\}$

Historical Context:

- Probability theory formalized by Kolmogorov (1933)
- SciPy.stats built on legacy of statistical Fortran libraries
- Unified interface inspired by R's distribution functions

Random Variables

```
from scipy import stats

X = stats.norm(1, 0.5)      # Normal( $\mu=1$ ,  $\sigma=0.5$ )
X.pdf(x)                   # Probability density  $f(x)$ 
X.cdf(x)                   # Cumulative  $F(x) = P(X \leq x)$ 
X.ppf(0.95)                # Quantile  $F^{-1}(p)$ 
X.interval(0.95)           # 95% confidence interval
X.rvs(100)                 # Random samples
```

Normal PDF: $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

Theory - Hypothesis Testing

Conceptual Overview:

- Hypothesis testing evaluates evidence against a null hypothesis H_0
- The p-value is the probability of observing results as extreme as data if H_0 is true
- Small p-values provide evidence against H_0

Mathematical Foundations:

- Test statistic: $T = t(X_1, \dots, X_n)$ under H_0
- p-value: $p = P(T \geq t_{obs} | H_0)$ for one-tailed, $p = P(|T| \geq |t_{obs}| | H_0)$ for two-tailed
- t-test: $t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}} \sim t_{n-1}$

Historical Context:

- Developed by Fisher (1925), Neyman and Pearson (1933)
- Significance testing (Fisher) vs decision theory (Neyman-Pearson)
- Controversy over p-value interpretation persists in modern statistics

Hypothesis Testing

```
stats.ttest_1samp(data, mu0)      #  $t = \frac{\bar{x} - \mu_0}{s/\sqrt{n}}$   
stats.ttest_ind(s1, s2)          # Two-sample  
stats.pearsonr(x, y)             # Correlation
```

p-value: $P(\text{observing result} | H_0 \text{ true})$

Reject H_0 if $p < \alpha$ (typically 0.05)

Subsection 3

Statistical Modeling

Theory - Linear Regression

Conceptual Overview:

- Linear regression models the relationship between a response and predictors
- Assumes a linear relationship with additive Gaussian noise
- Ordinary Least Squares (OLS) minimizes sum of squared residuals

Mathematical Foundations:

- Model: $y_i = \beta_0 + \sum_{j=1}^p \beta_j x_{ij} + \epsilon_i$ where $\epsilon_i \sim N(0, \sigma^2)$
- OLS estimator: $\hat{\beta} = (X^T X)^{-1} X^T \mathbf{y}$
- $R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$ measures proportion of variance explained

Historical Context:

- Method of least squares published by Legendre (1805), Gauss claimed prior use
- Gauss-Markov theorem proves OLS is BLUE (best linear unbiased estimator)
- Foundation of modern statistical modeling and econometrics

Linear Regression

Model: $y = \beta_0 + \beta_1 x_1 + \dots + \beta_p x_p + \epsilon$

Matrix form: $\mathbf{y} = X\beta + \epsilon$

OLS: $\hat{\beta} = (X^T X)^{-1} X^T \mathbf{y}$

```
import statsmodels.formula.api as smf
```

```
model = smf.ols("y ~ x1 + x2", data=df)
```

```
result = model.fit()
```

```
result.params           # Coefficients
```

```
result.rsquared         #  $R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$ 
```

Python Philosophy - The DataFrame Model

Pythonic Rationale:

- DataFrames encode relational algebra operations
- SQL-like operations in Python syntax
- Composition over inheritance for data pipelines

Python Code:

```
# Python: Relational algebra operations
import pandas as pd
df = pd.DataFrame({
    'category': ['A', 'B', 'A', 'B'],
    'value': [1, 2, 3, 4]
})
result = (df
    .query('value > 1')           # Selection (sigma)
    .groupby('category')         # Grouping
    .agg({'value': 'sum'})        # Aggregation
)
```

Python Philosophy - The DataFrame Model (continued)

C Comparison:

```
// C: No built-in dataframe concept  
// Use SQLite for relational operations  
sqlite3_exec(db,  
    "SELECT col1, SUM(col2) FROM table GROUP BY col1",  
    callback, NULL, NULL);  
// Must switch between C and SQL
```

Rust Comparison:

```
// Rust: Polars with lazy API  
use polars::prelude::*;  
let result = df.lazy()  
    .filter(col("value").gt(0))  
    .groupby([col("category")])  
    .agg([col("value").sum()])  
    .collect()?; // Execute optimized plan
```

Theory - Logistic Regression

Conceptual Overview:

- Logistic regression models binary outcomes using the logistic function
- Predicts probability of class membership rather than direct values
- Coefficients represent log-odds changes per unit increase in predictor

Mathematical Foundations:

- Logistic function: $\sigma(z) = \frac{1}{1+e^{-z}}$
- Model: $P(Y = 1|X) = \sigma(\beta^T \mathbf{x})$
- Log-odds: $\log \frac{p}{1-p} = \beta_0 + \sum_j \beta_j x_j$
- Maximum likelihood estimation via iterative methods (Newton-Raphson)

Historical Context:

- Logistic function introduced by Verhulst (1838) for population growth
- Applied to binary outcomes by Berkson (1944), coined “logit”
- Now standard for binary classification in medicine, social sciences, ML

Logistic Regression

Logistic function: $\sigma(z) = \frac{1}{1+e^{-z}}$

Model: $P(Y = 1|X) = \sigma(\beta_0 + \beta_1 x_1 + \dots)$

```
model = smf.logit("Species ~ Petal_Length", data=df)
result = model.fit()
```

Theory - Formula Syntax (Patsy)

Conceptual Overview:

- Formula syntax provides a concise way to specify statistical models
- Originated in R and S languages for statistical computing
- Patsy brings R-style formulas to Python's statsmodels

Mathematical Foundations:

- Formula $y \sim x_1 + x_2$ generates design matrix X for model fitting
- Interaction $x_1 : x_2$ creates product term $x_1 \times x_2$
- Categorical variables expand to dummy/indicator variables automatically

Historical Context:

- Wilkinson-Rogers notation introduced in S language (1970s)
- R adopted and extended the formula syntax
- Patsy library (2011) implements formulas for Python ecosystem

Formula Syntax (Patsy)

<code>"y ~ x1 + x2"</code>	<code># Linear with intercept</code>
<code>"y ~ x1 * x2"</code>	<code># Includes interaction x1:x2</code>
<code>"y ~ -1 + x1"</code>	<code># No intercept</code>
<code>"y ~ I(x**2)"</code>	<code># Arithmetic operations</code>
<code>"y ~ C(category)"</code>	<code># Categorical variable</code>

Theory - Poisson Regression

Conceptual Overview:

- Poisson regression models count data where outcomes are non-negative integers
- Uses a log link function to ensure predicted values are positive
- Part of the generalized linear model (GLM) family

Mathematical Foundations:

- Poisson distribution: $P(Y = k) = \frac{\lambda^k e^{-\lambda}}{k!}$
- Model: $\log(\lambda) = \beta_0 + \sum_j \beta_j x_j$, or $\lambda = e^{\beta^T \mathbf{x}}$
- Mean equals variance: $E[Y] = \text{Var}(Y) = \lambda$

Historical Context:

- Poisson distribution introduced by Siméon Denis Poisson (1837)
- Generalized Linear Models unified by Nelder and Wedderburn (1972)
- Widely used for rare event counts: accidents, diseases, website visits

Poisson Regression

For count data: $Y \sim \text{Poisson}(\lambda)$

Model: $\log(\lambda) = \beta_0 + \beta_1 x_1 + \dots$

```
model = smf.poisson("count ~ x1 + x2", data=df)
result = model.fit()
```

Subsection 4

Machine Learning with scikit-learn

Theory - Workflow (ML)

Conceptual Overview:

- Machine learning workflow standardizes the process from data to predictions
- Key steps: data preparation, model selection, training, and evaluation
- Train-test split prevents overfitting by evaluating on unseen data

Mathematical Foundations:

- Training: find $\hat{f} = \arg \min_{f \in \mathcal{F}} \sum_{i \in \text{train}} L(y_i, f(x_i))$
- Generalization error: $E_{out} = E_{(x,y)}[L(y, \hat{f}(x))]$
- Cross-validation: k -fold estimates E_{out} with lower variance

Historical Context:

- Train-test split methodology from early statistical learning theory
- Cross-validation introduced by Stone (1974) and Geisser (1975)
- scikit-learn (2007) standardized the fit-predict API for Python

Workflow

```
from sklearn import model_selection, linear_model, metrics

# Split data
X_train, X_test, y_train, y_test = model_selection.train_test_split(
    X, y, train_size=0.7)

# Train
model = linear_model.LinearRegression()
model.fit(X_train, y_train)

# Evaluate
y_pred = model.predict(X_test)
metrics.accuracy_score(y_test, y_pred)
```

Theory - Models

Conceptual Overview:

- ML models learn patterns from data to make predictions on new instances
- Supervised learning: regression (continuous) and classification (categorical)
- Unsupervised learning: clustering, dimensionality reduction without labels

Mathematical Foundations:

- Linear model: $\hat{y} = \mathbf{w}^T \mathbf{x} + b$
- Regularization: $\min_{\mathbf{w}} L(\mathbf{y}, \hat{\mathbf{y}}) + \lambda R(\mathbf{w})$
- Ridge: $R(\mathbf{w}) = \|\mathbf{w}\|_2^2$; Lasso: $R(\mathbf{w}) = \|\mathbf{w}\|_1$

Historical Context:

- Linear regression oldest statistical method (Legendre, 1805)
- Ridge regression (Hoerl and Kennard, 1970) for multicollinearity
- Lasso (Tibshirani, 1996) introduced sparsity-inducing L1 penalty

Models

Task	Model
Regression	LinearRegression, Ridge (L2), Lasso (L1)
Classification	LogisticRegression, SVC, RandomForestClassifier
Clustering	KMeans

Regularization: - Ridge (L2): $\min \|y - Xw\|^2 + \alpha \|w\|^2$ - Lasso (L1):
 $\min \|y - Xw\|^2 + \alpha \|w\|_1$

Theory - Model Evaluation

Conceptual Overview:

- Model evaluation measures how well a model generalizes to unseen data
- Classification metrics derived from the confusion matrix
- Trade-offs exist between different metrics (precision vs recall)

Mathematical Foundations:

- Confusion matrix: TP, FP, TN, FN counts
- Accuracy: $\frac{TP+TN}{TP+TN+FP+FN}$
- Precision: $\frac{TP}{TP+FP}$, Recall: $\frac{TP}{TP+FN}$
- F1: harmonic mean of precision and recall: $2 \cdot \frac{P \cdot R}{P+R}$

Historical Context:

- Confusion matrix from early signal detection theory (WWII radar)
- ROC curves developed for radar operators (Peterson and Birdsall, 1953)
- F1 score became standard in information retrieval (van Rijsbergen, 1979)

Model Evaluation

Confusion Matrix:

	Predicted +	Predicted -
Actual +	TP	FN
Actual -	FP	TN

Metrics: - **Accuracy:** $(TP + TN)/Total$ - **Precision:** $TP/(TP + FP)$ - **Recall:** $TP/(TP + FN)$ - **F1:** $2 \cdot (Precision \cdot Recall)/(Precision + Recall)$

```
metrics.confusion_matrix(y_test, y_pred)
metrics.classification_report(y_test, y_pred)
```

Theory - Clustering

Conceptual Overview:

- Clustering groups similar data points without using labels
- Unsupervised learning: discovers structure in data automatically
- K-means is the most widely used clustering algorithm

Mathematical Foundations:

- K-means objective: $\min_{C_1, \dots, C_k} \sum_{j=1}^k \sum_{x \in C_j} \|x - \mu_j\|^2$
- Algorithm alternates: assign points to nearest centroid, update centroids
- Converges to local minimum; sensitive to initialization

Historical Context:

- K-means algorithm independently discovered by multiple researchers (1950s-60s)
- Lloyd's algorithm (1957, published 1982) is the standard implementation
- Hierarchical and density-based methods (DBSCAN, 1996) handle non-convex clusters

Clustering

```
from sklearn import cluster

kmeans = cluster.KMeans(n_clusters=3)
kmeans.fit(X)
y_pred = kmeans.predict(X)
```


Section 7

Part VI: Advanced Topics

Subsection 1

Concurrency and the GIL

Python Philosophy - The GIL Tradeoff

Pythonic Rationale:

- Global Interpreter Lock simplifies memory management
- Reference counting is not thread-safe without GIL
- “Simple is better than complex” - single-threaded semantics

Python Code:

```
# Python: Threading limited by GIL
```

```
import threading
```

```
def worker():
```

```
    for _ in range(1000000):
```

```
        pass # CPU-bound work
```

```
threads = [threading.Thread(target=worker) for _ in range(4)]
```

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

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

```
# Runs on ONE core due to GIL
```

Python Philosophy - The GIL Tradeoff (continued)

C Comparison:

```
// C: True parallelism with pthreads
#include <pthread.h>
void* thread_func(void* arg) {
    // Runs in parallel on different cores
    return NULL;
}
pthread_t t1, t2;
pthread_create(&t1, NULL, thread_func, NULL);
pthread_create(&t2, NULL, thread_func, NULL);
// Must manually synchronize shared data
```

Rust Comparison:

```
// Rust: Fearless concurrency
use std::thread;
let handle = thread::spawn(|| {
    // Ownership prevents data races at compile time
});
// Mutex<T> guarantees exclusive access
```


Theory - Global Interpreter Lock (GIL)

Conceptual Overview:

- The GIL is a mutex that prevents multiple threads from executing Python bytecode simultaneously
- Simplifies memory management by protecting reference counts from race conditions
- Trade-off: thread safety for CPU-bound tasks but limits parallelism

Mathematical Foundations:

- Lock granularity: $\text{lock}(GIL) \rightarrow \text{execute}() \rightarrow \text{unlock}(GIL)$
- Speedup for I/O-bound: $T_{\text{threaded}} \approx T_{\text{sequential}}/n$ (parallel I/O waits)
- Speedup for CPU-bound: $T_{\text{threaded}} \approx T_{\text{sequential}}$ (no parallelism)
- Alternative: multiprocessing uses separate processes with separate GILs

Historical Context:

- GIL introduced in Python 1.5 (1997) for thread-safe reference counting
- Removal attempts (GILectomy) have failed due to performance regressions on single-threaded code
- Free-threaded Python (PEP 703, Python 3.13+) optional without GIL
- Alternatives: multiprocessing, asyncio, or extension modules that release GIL

Subsection 2

Memory Management

Python Philosophy - Memory Management

Pythonic Rationale:

- Automatic memory management - no malloc/free
- Reference counting + cycle detection
- Focus on algorithms, not memory

Python Code:

```
# Python: Automatic memory management
```

```
def create_list():  
    arr = [1, 2, 3, 4, 5]  
    return arr[0] # arr garbage-collected
```

```
x = create_list()  
# No free() needed - reference counting + GC
```

Python Philosophy - Memory Management (continued)

C Comparison:

```
// C: Manual memory management  
int *arr = malloc(100 * sizeof(int));  
// ... use arr ...  
free(arr); // Must not forget!  
// Use after free: undefined behavior
```

Rust Comparison:

```
// Rust: Ownership-based memory management  
{  
    let arr = vec  
    ![0; 100];  
    // ... use arr ...  
} // arr dropped here automatically  
// No garbage collector, no manual free  
// Compile-time guarantees: no use-after-free
```

Theory - Memory Management Strategies

Conceptual Overview:

- Reference counting tracks how many references point to each object; deallocates when count reaches zero
- Tracing garbage collection periodically scans memory to find unreachable objects
- Ownership systems (Rust) enforce memory safety at compile time without runtime overhead

Mathematical Foundations:

- Reference counting: $\text{refcount}(o) = |\{r : r \rightarrow o\}|$; free when $\text{refcount}(o) = 0$
- Cycle problem: $\text{refcount}(a) > 0, \text{refcount}(b) > 0$ but a, b unreachable
- Mark-and-sweep: $\text{mark}(r)$ for all roots r ; $\text{sweep}(\{o : \neg \text{marked}(o)\})$
- Ownership: $\text{owner}(o)$ is unique; $\text{borrow}(o)$ is temporary reference

Historical Context:

- Reference counting used in Lisp (1958), Python (1991), Swift (2014)
- Tracing GC invented for Lisp (McCarthy, 1960); generational GC (1983)
- Rust ownership model (2010) provides compile-time memory safety
- Trade-offs: reference counting has predictable deallocation but cycles; GC handles cycles but has pause times

Subsection 3

Sparse Matrices

Theory - Sparse Matrices

Conceptual Overview:

- Sparse matrices contain mostly zero elements, making dense storage inefficient
- Specialized formats store only non-zero elements, dramatically reducing memory
- Critical for large-scale scientific computing, graph algorithms, and machine learning

Mathematical Foundations:

- Sparsity: $\text{sparsity}(A) = \frac{\#\{a_{ij}=0\}}{n \times m}$
- Storage: $O(\text{nnz})$ instead of $O(nm)$ where nnz = number of non-zeros
- Complexity: many operations scale with nnz rather than matrix dimensions

Historical Context:

- Sparse matrix methods developed in 1960s for finite element analysis
- Compressed formats (CSR/CSC) standardized in sparse BLAS (1989)
- Now essential in PageRank, recommendation systems, neural networks

Formats

Format	Best For	Description
COO	Construction	Coordinate list (row, col, value)
CSR/CSC	Computation	Compressed sparse row/column
DIA	Diagonal	Diagonal storage

```
import scipy.sparse as sp

A = sp.coo_matrix((values, (rows, cols)))
A = sp.diags([1, -2, 1], [1, 0, -1])
x = sp.linalg.spsolve(A, b)
```

Subsection 4

Signal Processing

Theory - Fourier Transforms

Conceptual Overview:

- Fourier transform decomposes signals into constituent frequencies
- Time domain signal \leftrightarrow Frequency domain representation
- Foundation for filtering, compression, spectral analysis, and communications

Mathematical Foundations:

- DFT: $X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}$
- Inverse: $x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{i2\pi kn/N}$
- FFT: $O(N \log N)$ algorithm (Cooley-Tukey, 1965)

Historical Context:

- Fourier series introduced by Joseph Fourier (1822) for heat equation
- FFT algorithm rediscovered by Cooley and Tukey (1965), originally by Gauss (1805)
- Revolutionized signal processing, enabling real-time spectral analysis

Fourier Transforms

DFT:
$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N}$$

```
from scipy import fftpack
```

```
F = fftpack.fft(signal)
```

```
f = fftpack.fftfreq(N, 1.0/sample_rate)
```

Filtering:

```
b = signal.firwin(numtaps, cutoff) # FIR
```

```
b, a = signal.butter(order, cutoff) # IIR (Butterworth)
```

```
filtered = signal.filtfilt(b, a, signal)
```

Subsection 5

File I/O and Optimization

Theory - File Formats

Conceptual Overview:

- Different file formats optimize for different use cases: size, speed, interoperability
- Text formats (CSV, JSON) are human-readable but slower and larger
- Binary formats (HDF5, Pickle) are compact and fast but less portable

Mathematical Foundations:

- Compression ratio: $\frac{\text{compressed size}}{\text{original size}}$
- I/O bandwidth: $\frac{\text{data size}}{\text{read/write time}}$
- Serialization: encode : Object \rightarrow bytes, decode : bytes \rightarrow Object

Historical Context:

- CSV dates to early mainframe computing (1960s-70s)
- HDF developed at NCSA (1988), HDF5 released (1998)
- JSON specified by Douglas Crockford (early 2000s), standardized as ECMA-404

File Formats

Format	Use Case	Library
CSV	Simple data	<code>np.loadtxt</code> , <code>pd.read_csv</code>
HDF5	Large numerical	<code>h5py</code>
JSON	Cross-language	<code>json</code>
Pickle	Python objects	<code>pickle</code>

Warning: `pickle` can execute arbitrary code!

Theory - CSV Files

Conceptual Overview:

- CSV (Comma-Separated Values) is a simple text format for tabular data
- Each row is a record, columns separated by delimiters (usually comma)
- Widely supported but lacks type information and schema enforcement

Mathematical Foundations:

- Tabular model: $T \subseteq V_1 \times V_2 \times \dots \times V_n$
- Parse complexity: $O(n \times m)$ for n rows, m columns
- Memory mapping enables efficient partial reads for large files

Historical Context:

- CSV format predates personal computers, used in early mainframes
- RFC 4180 (2005) provided first formal specification
- Remains dominant for data exchange due to simplicity and universality

CSV Files

```
# NumPy (numerical data)
np.savetxt("data.csv", array, delimiter=",")
data = np.loadtxt("data.csv", skiprows=1, delimiter=",")

# Pandas (mixed data types)
df = pd.read_csv("data.csv")
df.to_csv("output.csv")
```

Theory - HDF5

Conceptual Overview:

- HDF5 is a hierarchical data format for scientific and large-scale data
- Supports datasets (arrays), groups (folders), attributes (metadata)
- Enables partial I/O, compression, and parallel access

Mathematical Foundations:

- Hierarchical structure: tree of groups and datasets
- Chunked storage: $O(1)$ access to arbitrary slices
- Compression: $\text{store}(\text{chunk}) = \text{compress}(\text{chunk})$ reduces I/O

Historical Context:

- HDF developed at NCSA, University of Illinois (1988)
- HDF5 (1998) addressed limitations of HDF4
- Standard format in astronomy, genomics, climate science, deep learning

HDF5 (Large Datasets)

```
import h5py

f = h5py.File("data.h5", "w")
f.create_dataset("array", data=np.arange(100), compression="gzip")
f.attrs["description"] = "Experiment results"
data_slice = f["large_array"][100:200, :] # Partial reads
```

Theory - Serialization

Conceptual Overview:

- Serialization converts objects to bytes for storage or transmission
- JSON is text-based, human-readable, and cross-language compatible
- Pickle is Python-specific, handles arbitrary objects, but has security risks

Mathematical Foundations:

- Serialization: $\text{encode} : O \rightarrow B$ where B is byte sequence
- Deserialization: $\text{decode} : B \rightarrow O$
- JSON types: string, number, boolean, null, array, object

Historical Context:

- JSON emerged from JavaScript (Douglas Crockford, early 2000s)
- Pickle protocol in Python since version 0.9 (1991)
- JSON standardized as ECMA-404 (2013) and RFC 8259 (2017)

JSON and Pickle

```
import json, pickle

# JSON (human-readable, cross-language)
json_str = json.dumps(data, indent=2)
data = json.loads(json_str)

# pickle (any Python object)
pickle.dump(obj, open("data.pkl", "wb"))
obj = pickle.load(open("data.pkl", "rb"))
```

Theory - Code Optimization

Conceptual Overview:

- Code optimization improves performance without changing behavior
- Python's interpreted nature makes it slower than compiled languages
- Key strategies: vectorization, JIT compilation, ahead-of-time compilation

Mathematical Foundations:

- Runtime: $T_{total} = \sum_i T_i$ where T_i is time in function i
- Amdahl's Law: $speedup = \frac{1}{(1-p)+p/s}$ where p is parallelizable fraction
- Big-O: focus optimization on $O(n^2)$ and higher complexity parts

Historical Context:

- Profiling tools since early computing (gprof, 1982)
- Psyco (2003) first Python JIT, succeeded by PyPy (2007)
- Numba (2012) provides LLVM-based JIT for numerical Python

Code Optimization

Performance hierarchy: Cython > Numba > NumPy > Python

```
import numba

@numba.jit(nopython=True)
def fast_function(x):
    ...
```

Key rule: Profile first, optimize bottlenecks only

Python Philosophy - Performance Model

Pythonic Rationale:

- “Premature optimization is the root of all evil” - Knuth
- Python optimizes for developer time, not CPU time
- Profile first, optimize only bottlenecks

Python Code:

Python: Bytecode interpretation overhead

```
def sum_list(arr):  
    total = 0  
    for x in arr:  
        total += x  
    return total
```

~50-100x slower than C

Use NumPy or Cython for speed

Python Philosophy - Performance Model (continued)

C Comparison:

```
// C: Compiled to machine code
int sum(int *arr, int n) {
    int total = 0;
    for (int i = 0; i < n; i++) {
        total += arr[i]; // Direct CPU instructions
    }
    return total; // ~1ns per iteration
}
```

Rust Comparison:

```
// Rust: Zero-cost abstractions
fn sum(arr: &[i32]) -> i32 {
    arr.iter().sum() // Compiles to same code as C loop
}
// Abstractions have no runtime cost
```

Theory - Bytecode Interpretation Performance

Conceptual Overview:

- Python compiles source code to bytecode, then interprets it in a virtual machine
- Bytecode interpretation adds overhead: each instruction requires dispatch, type checking, and memory allocation
- The interpreter loop is a performance bottleneck compared to native machine code

Mathematical Foundations:

- Bytecode execution: $\text{exec}(bc) = \sum_i \text{cost}(op_i)$ where each op_i has interpreter overhead
- Overhead factor: $T_{\text{Python}} \approx 50 \times T_C$ for tight loops due to boxing/unboxing
- JIT compilation: $\text{compile}(bc) \rightarrow \text{machine_code}$ eliminates interpreter dispatch

Historical Context:

- Virtual machines date to Pascal P-code (1970s) and Smalltalk (1972)
- Python bytecode designed for simplicity, not speed (Python 1.0, 1994)
- JIT approaches: Psyco (2003), PyPy (2007), Numba (2012) for numerical code

Theory - Cython

Conceptual Overview:

- Cython compiles Python to C, enabling C-level performance
- Type annotations allow direct C operations without Python overhead
- Can compile existing Python code with minimal modifications

Mathematical Foundations:

- Overhead reduction: $T_{\text{cython}} \approx T_C + T_{\text{python_calls}}$
- Type inference: static types eliminate runtime type checking
- Memory layout: contiguous arrays enable cache-efficient access

Historical Context:

- Cython forked from Pyrex (2007) by Stefan Behnel and Greg Ewing
- Widely used in NumPy, SciPy, and high-performance Python libraries
- Enables Python to match C/Fortran for numerical computing

Cython (Ahead-of-time Compilation)

```
cimport numpy
cimport cython

@cython.boundscheck(False)
@cython.wraparound(False)
def cy_sum(numpy.ndarray[numpy.float64_t, ndim=1] data):
    cdef numpy.float64_t s = 0.0
    cdef int n, N = len(data)
    for n in range(N):
        s += data[n]
    return s
```

Features: cdef for C-types, bounds checking disabled

Python Philosophy - Extension Philosophy

Pythonic Rationale:

- Write in Python, optimize in C/Rust when needed
- “80/20 rule”: 80% of code runs infrequently
- Seamless FFI (Foreign Function Interface)

Python Code:

```
# Python: Seamlessly call C extensions  
import numpy as np # C extension  
arr = np.array([1, 2, 3])  
result = arr.sum() # Runs in C, fast!
```

```
# Or write your own extension (Cython)  
# mymodule.pyx:  
# def fast_sum(long[:] arr):  
#     cdef long total = 0  
#     for i in range(len(arr)):  
#         total += arr[i]  
#     return total
```

Python Philosophy - Extension Philosophy (continued)

C Comparison:

```
// C: No FFI needed - you're already in C
// Python extension module:
static PyObject* my_func(PyObject *self, PyObject *args) {
    int x;
    if (!PyArg_ParseTuple(args, "i", &x))
        return NULL;
    return PyLong_FromLong(x * 2);
}
```

Rust Comparison:

```
// Rust: Py03 for Python extensions
use pyo3::prelude::*;
#[pyfunction]
fn my_func(x: i64) -> i64 {
    x * 2 // Compiled to native code
}
#[pymodule]
fn my_module(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(my_func, m)?)
}
```

Theory - Foreign Function Interface

Conceptual Overview:

- Foreign Function Interface (FFI) enables calling code written in other languages
- Calling conventions define how functions receive arguments and return values
- Type marshaling converts data between language representations

Mathematical Foundations:

- Calling convention: $\text{call}(f, args) \rightarrow \text{stack/registers} \rightarrow \text{return}$
- Type marshaling: $\text{marshal} : T_{\text{Python}} \rightarrow T_C$ with potential precision loss
- ABI (Application Binary Interface): specification for binary compatibility

Historical Context:

- FFI mechanisms since early inter-language communication (1970s)
- Python C API designed for extension modules (Python 1.0, 1994)
- Modern alternatives: ctypes (Python 2.5), cffi (2012), PyO3 for Rust (2018)

Subsection 6

Bayesian Statistics

Theory - Bayesian Statistics

Conceptual Overview:

- Bayesian statistics treats parameters as random variables with distributions
- Prior beliefs are updated with data to form posterior distributions
- Provides full uncertainty quantification, not just point estimates

Mathematical Foundations:

- Bayes' theorem: $P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$
- Prior: $P(\theta)$ encodes beliefs before seeing data
- Posterior: $P(\theta|D)$ combines prior and likelihood
- Evidence: $P(D) = \int P(D|\theta)P(\theta)d\theta$ (normalizing constant)

Historical Context:

- Bayes' theorem published posthumously by Thomas Bayes (1763)
- Laplace independently developed Bayesian methods (early 1800s)
- MCMC methods (Metropolis-Hastings, 1953) enabled practical Bayesian computation
- Modern revival with PyMC, Stan, and probabilistic programming

Bayes' Theorem

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)}$$

- $P(\theta)$ = Prior (belief before data)
- $P(D|\theta)$ = Likelihood (data given parameters)
- $P(\theta|D)$ = Posterior (belief after data)

```
import pymc3 as mc
```

```
with mc.Model() as model:
```

```
    mu = mc.Normal('mu', 0, sd=10) # Prior
```

```
    mc.Normal('X', mu, sd=1, observed=data) # Likelihood
```

```
    trace = mc.sample(10000) # MCMC
```

Subsection 7

Graphs with NetworkX

Theory - Graph Theory

Conceptual Overview:

- Graphs model relationships between entities: nodes (vertices) and edges (links)
- Can be directed or undirected, weighted or unweighted
- Applications: social networks, transportation, biology, recommendation systems

Mathematical Foundations:

- Graph: $G = (V, E)$ where V is vertex set, $E \subseteq V \times V$ is edge set
- Adjacency matrix: $A_{ij} = 1$ if $(i, j) \in E$, else 0
- Degree: $d(v) = |\{u : (v, u) \in E\}|$

Historical Context:

- Graph theory originated with Euler's Königsberg bridge problem (1736)
- NetworkX created by Aric Hagberg (2004)
- Now standard for network analysis in Python, with 10M+ downloads/year

Graph Creation

```
import networkx as nx

g = nx.Graph()           # Undirected
g = nx.DiGraph()         # Directed
g.add_nodes_from([1, 2, 3])
g.add_weighted_edges_from([(1, 2, 1.5)])
```

Theory - Graph Algorithms

Conceptual Overview:

- Graph algorithms solve problems on network structures
- Key problems: shortest path, centrality, community detection, connectivity
- Efficiency depends on graph representation (adjacency list vs matrix)

Mathematical Foundations:

- Shortest path: $\min_{p \in P(s,t)} \sum_{e \in p} w(e)$
- Centrality measures identify important nodes
- Degree centrality: $C_D(v) = d(v)/(n-1)$
- Betweenness: $C_B(v) = \sum_{s \neq t \neq v} \sigma_{st}(v) / \sigma_{st}$

Historical Context:

- Dijkstra's shortest path algorithm (1956)
- Betweenness centrality introduced by Freeman (1977)
- PageRank algorithm (Brin and Page, 1998) revolutionized web search

Graph Algorithms

```
# Shortest path
path = nx.shortest_path(g, source, target)

# Centrality measures
nx.degree_centrality(g)
nx.betweenness_centrality(g)

# Convert to matrix
A = nx.to_scipy_sparse_matrix(g)
```


Section 8

Part VII: AI-Powered Development Tools

Subsection 1

The Rise of AI Coding Assistants

Theory - AI-Assisted Software Development

Conceptual Overview:

- AI coding assistants use Large Language Models (LLMs) to help write, understand, and debug code
- Evolved from simple autocomplete to full agentic systems that can plan, edit, and test
- Open source tools provide transparency, customization, and provider independence

Mathematical Foundations:

- Code generation: $P(\text{code}|\text{prompt}, \text{context}) = \prod_t P(\text{token}_t|\text{tokens}_{<t})$
- Context window: maximum tokens the model can process (typically 128K - 1M+)
- Temperature: τ controls randomness; lower τ = more deterministic output

Historical Context:

- GitHub Copilot launched 2021, popularized AI coding assistance
- Open-source alternatives emerged 2023-2024: Aider, Continue, OpenCode
- 2024-2025: Agentic tools that can plan, execute, and iterate autonomously

The AI Coding Revolution

Evolution of Developer Tools:

Era	Tool Type	Capability
2010s	Linters/IDE	Syntax highlighting, basic autocomplete
2021	AI Autocomplete	Line-by-line suggestions (Copilot)
2023	Chat Assistants	Explain, refactor, Q&A (ChatGPT)
2024+	AI Agents	Plan, edit, test, iterate autonomously

Key Insight: Modern AI agents don't just suggest—they can read codebases, plan changes, edit files, run tests, and iterate.

Subsection 2

Open Source AI Coding Tools

Tool Comparison Overview

Major Open Source AI Coding Tools (2025):

Tool	Interface	Stars	Key Feature
OpenCode	TUI, Desktop, IDE	131K	Multi-provider, LSP support, plan/build modes
Aider	Terminal	25K+	Git-aware, pair programming focus
Continue	IDE Extension	20K+	VSCode/JetBrains, custom rules
Cursor	Forked VSCode	50K+	Native IDE integration (source-available)

Proprietary Alternatives: - GitHub Copilot: Industry standard, deep IDE integration - Claude Code: Anthropic's official CLI tool - Amazon Q Developer: AWS ecosystem integration

OpenCode: Open Source AI Agent

Why OpenCode?

- **100% Open Source:** MIT license, full transparency
- **Provider Agnostic:** Works with Claude, GPT, Gemini, local models
- **Multiple Interfaces:** Terminal (TUI), Desktop app, IDE extension
- **LSP Integration:** Automatic language server loading for better context
- **Plan/Build Modes:** Read-only planning, full-access building

Installation:

```
# Quick install  
curl -fsSL https://opencode.ai/install | bash
```

```
# Package managers  
npm install -g opencode-ai  
brew install anomalyco/tap/opencode
```

```
# Desktop app available for macOS, Windows, Linux
```

OpenCode: Core Features

Key Commands:

<code>opencode</code>	<i># Start interactive session</i>
<code>/init</code>	<i># Analyze project, create AGENTS.md</i>
<code>/share</code>	<i># Share conversation link</i>
<code>/undo</code>	<i># Revert last changes</i>
<code>/redo</code>	<i># Redo undone changes</i>

Plan vs Build Mode:

Mode	Capability	Use Case
Plan	Read-only, suggests approach	Architecture decisions, unfamiliar code
Build	Full file edit access	Implementation, refactoring

Tip: Use @ to fuzzy search files, drag images into terminal for visual context.

Other Major Players: Aider

Aider: Terminal-Based Pair Programmer

Install

```
pip install aider-chat
```

Usage

```
aider file1.py file2.py      # Start with specific files  
aider --watch                # Auto-add files you edit
```

Key Features: - Git-centric: commits each change with descriptive messages - Works directly in terminal with your editor - Best for: Developers who want tight git integration

Other Major Players: Continue and Cursor

Continue.dev: IDE Extension

```
# Install in VSCode, JetBrains  
# Configure ~/.continue/config.json with your API keys
```

- Customizable rules and prompts
- Best for: Teams with existing IDE workflows

Cursor: AI-Native IDE

- Forked VSCode with built-in AI integration
- Source-available (not fully open source)
- Best for: Users wanting seamless IDE experience

Subsection 3

Best LLMs for Python Development

Model Selection for Coding Tasks

Current Best Models (2025):

Model	Provider	Best For	Context
Claude 3.5/4 Sonnet	Anthropic	Reasoning, code quality	200K
GPT-4o	OpenAI	General purpose, speed	128K
Gemini 2.0 Pro	Google	Large codebases	1M+
Claude 3.5 Haiku	Anthropic	Fast, cheap	200K
Llama 3.1 70B	Ollama (local)	Privacy, offline	128K
Qwen 2.5 Coder	Ollama (local)	Code-specialized	128K

Recommendation: Claude 3.5/4 Sonnet for complex Python work; local models for sensitive codebases.

Local Models for Privacy

Running Local LLMs with Ollama:

Install Ollama

```
curl -fsSL https://ollama.ai/install | sh
```

Pull coding-specialized models

```
ollama pull qwen2.5-coder:7b      # Fast, code-focused
```

```
ollama pull llama3.1:70b        # Best quality local
```

```
ollama pull deepseek-coder:6b   # Lightweight option
```

Configure OpenCode for Local:

```
// ~/.config/opencode/config.json
```

```
{
```

```
  "provider": "ollama",
```

```
  "model": "qwen2.5-coder:7b"
```

```
}
```


Cloud vs Local Models: Trade-offs

Aspect	Cloud Models	Local Models
Quality	Highest	Good (smaller gap)
Privacy	Data sent externally	Fully private
Cost	Pay per token	Hardware cost only
Speed	Network dependent	GPU dependent

When to Use Local: - Proprietary or sensitive codebases - Offline development environments - Compliance requirements (GDPR, HIPAA)

Subsection 4

Usage Recommendations

Effective AI-Assisted Workflow

Recommended Workflow:

1. PLAN: Use plan mode to understand codebase
↓
2. ARCHITECT: Ask AI to propose solution, review critically
↓
3. BUILD: Switch to build mode, implement incrementally
↓
4. VERIFY: Run tests, review changes, check edge cases
↓
5. ITERATE: Refine based on errors or feedback

Best Practices:

- Start with `/init` to create project context (`AGENTS.md`)
- Use plan mode for unfamiliar code or complex changes
- Provide specific file paths with `@filename`
- Review every change before committing
- Always run tests after AI modifications

Practical Usage Examples: Understanding & Features

Example 1: Understanding Code

How is authentication handled in `@src/api/auth.py`?
What design patterns are used?

Example 2: Adding Features

Add a rate limiter to the API endpoints.
Use the decorator pattern similar to `@src/utils/cache.py`.
Include unit tests.

Practical Usage Examples: Debugging & Refactoring

Example 3: Debugging

The function `@calculate_metrics` is returning NaN for some inputs. Find the bug and fix it. Add input validation.

Example 4: Refactoring

Refactor `@src/data/processor.py` to use pandas vectorization instead of loops. Maintain the same interface.

Code Review with AI

Using AI for Code Review:

```
# Ask AI to review your code:  
# "Review this pull request for:  
# 1. Potential bugs  
# 2. Security vulnerabilities  
# 3. Performance issues  
# 4. Code style and best practices"
```

What AI Excels At: - Spotting edge cases and boundary conditions - Identifying security vulnerabilities (SQL injection, XSS) - Suggesting performance optimizations - Explaining complex code sections

AI Limitations in Code Review

What AI Struggles With: - Understanding business logic nuances - Architectural decisions requiring domain knowledge - Novel algorithms not in training data - Code requiring deep project context

Best Practice: Use AI as a first-pass reviewer, but always have human review for final approval.

Subsection 5

Common Pitfalls

What Can Go Wrong: Over-Trust and Hallucinations

1. Over-Trusting Generated Code

AI might generate plausible but WRONG code:

```
def calculate_average(numbers):  
    return sum(numbers) / len(numbers)  # Crashes on empty list!
```

Always verify edge cases yourself

2. Hallucinated APIs

AI may invent non-existent methods:

```
df.calculate_statistics()  # Doesn't exist!  
df.describe()             # Correct method
```

Always check documentation

What Can Go Wrong: Security Issues

3. Security Vulnerabilities

AI might suggest unsafe patterns:

```
query = f"SELECT * FROM users WHERE id = {user_input}" # SQL injection!
```

Verify security-sensitive code carefully

Mitigation: - Never paste API keys or secrets into AI prompts - Review all security-sensitive code manually - Run security scanners (bandit, semgrep) on AI-generated code - Validate inputs and sanitize outputs

Context Window Limitations

Context Window Sizes:

Model	Context	Practical Limit
GPT-4o	128K	~100 files
Claude 3.5	200K	~150 files
Gemini 2.0	1M+	~1000 files

Strategies for Large Codebases:

- Use `@file` to include only relevant files
- Break tasks into smaller, focused requests
- Create `AGENTS.md` to provide project context
- Use plan mode first to identify relevant files

When NOT to Use AI

Avoid AI Assistance For:

- High-stakes production code without review
- Code with strict compliance requirements
- Novel algorithms requiring original thinking
- Learning fundamentals (use AI to understand, not replace learning)

Why? - AI can't guarantee correctness - Regulatory compliance may require human authorship - Learning requires struggling through problems yourself

Maintaining Developer Skills

The Risk: Over-Reliance

- AI can become a crutch, not a tool
- Skills atrophy when not practiced
- Debugging ability degrades
- Understanding of fundamentals weakens

Mitigation Strategies:

Strategy	Description
Understand Before Accept Practice Without AI Learn the “Why”	Read and comprehend every AI suggestion Regularly solve problems independently Ask AI to explain reasoning, not just produce code
Review Critically Learn Fundamentals	Treat AI output as a junior developer’s PR AI can’t replace deep understanding of algorithms, data structures

Subsection 6

Best Practices Summary

Effective AI Collaboration

DO:

- Review every line of generated code
- Run tests after modifications
- Use plan mode for complex changes
- Provide clear, specific instructions
- Maintain AGENTS.md for project context
- Keep API keys secure (use environment variables)
- Learn from AI explanations

DON'T:

- Blindly accept generated code
- Skip testing AI-written code
- Paste sensitive data (API keys, passwords) into AI prompts
- Use AI for code you don't understand
- Let AI replace learning fundamentals
- Ignore security implications

Configuration Best Practices

Secure Configuration:

```
# Store API keys in environment variables  
export ANTHROPIC_API_KEY="your-key-here"  
export OPENAI_API_KEY="your-key-here"  
  
# Or use a secrets manager  
# Never commit API keys to git!
```

Project Context: AGENTS.md

Create AGENTS.md for Project Context:

```
# Project: Data Processing Pipeline
```

```
## Architecture
```

- src/ingest/: Data ingestion modules
- src/transform/: Transformation logic
- src/output/: Export formats

```
## Conventions
```

- Use type hints everywhere
- Test with pytest
- Follow PEP 8 style guide

```
## Key Dependencies
```

- pandas for data manipulation
- pytest for testing

Tip: Run `/init` to auto-generate this file for your project.

Summary: AI as a Force Multiplier

Key Takeaways:

- ➊ **AI tools are assistants, not replacements** — Review, verify, understand
- ➋ **Open source provides flexibility** — OpenCode, Aider, Continue offer transparency
- ➌ **Model choice matters** — Claude 3.5/4 for quality, local for privacy
- ➍ **Context is critical** — Use AGENTS.md, provide specific files
- ➎ **Skills must be maintained** — Practice without AI, learn fundamentals

The Future of AI-Assisted Development

Trends:

- AI tools will continue improving rapidly
- Open source ensures community control and transparency
- Best developers will be those who leverage AI effectively while maintaining deep expertise

Resources:

- OpenCode: <https://opencode.ai>
- Aider: <https://aider.chat>
- Continue: <https://continue.dev>
- Ollama (local models): <https://ollama.ai>

Section 9

Appendices

Subsection 1

A: Quick Reference

Essential Imports

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from scipy import linalg, optimize, integrate
import sympy
from sklearn import model_selection, linear_model
```

Common Patterns

```
# List comprehension  
[x for x in items if condition(x)]  
  
# Safe defaults  
d.get(key, default)  
  
# Enumeration  
for i, item in enumerate(items):
```

Subsection 2

B: Best Practices

Code Style

- Use `snake_case` for functions/variables, `PascalCase` for classes
- Write docstrings for public functions
- Single responsibility principle

Debugging

- Test obvious cases first, then edge cases
- Use bisection to isolate bugs
- The Five R's: Read, Run, Ruminare, Rubberduck, Retreat

Performance

- Vectorize with NumPy (eliminate Python loops)
- Use `@numba.jit` for unavoidable loops
- Profile before optimizing

Subsection 3

C: Mathematical Reference

Linear Algebra

- **Norms:** $\|\mathbf{x}\|_2 = \sqrt{\sum x_i^2}$
- **SVD:** $A = U\Sigma V^T$
- **Eigen:** $A\mathbf{v} = \lambda\mathbf{v}$

Probability

- **Expected value:** $E[X] = \sum xP(x)$
- **Variance:** $\text{Var}(X) = E[(X - \mu)^2]$
- **Normal:** $f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

Information Theory

- **Entropy:** $H(X) = -\sum_x P(x) \log_2 P(x)$
- **KL Divergence:** $D_{KL}(P\|Q) = \sum_x P(x) \log \frac{P(x)}{Q(x)}$

Section 10

Course Summary

Subsection 1

What We've Covered

What We've Covered

- ➊ **Python Syntax** - Types, functions, data structures
- ➋ **Object-Oriented Design** - Classes, inheritance, design principles
- ➌ **Typing & Debugging** - Dynamic typing, error handling, debugging strategies
- ➍ **Numerical Computing** - NumPy, SciPy, Matplotlib, SymPy
- ➎ **Data Analysis** - Pandas, statistics, machine learning
- ➏ **Advanced Topics** - Sparse matrices, optimization, Bayesian statistics

Subsection 2

Key Takeaways

Key Takeaways

- Python's dynamic typing requires careful attention
- Vectorization eliminates slow Python loops
- Mathematical foundations power all data science tools
- Debugging is systematic, not random
- Profile first, optimize only bottlenecks

Section 11

Literature

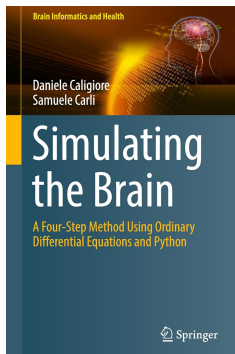


Figure 1: Book Cover

Authors:

Daniele Caligiore &
Samuele Carli

Springer 2025

ISBN:

978-981-96-2717-2

Simulating the Brain

A Four-Step Method Using Ordinary Differential Equations and Python

Comprehensive guide to developing brain models using ODEs and Python

Four-Step Method:

- ① Model Design
- ② ODE Implementation
- ③ Python Simulation
- ④ Analysis & Visualization

Key Features:

- Self-consistent textbook
- Hands-on Python examples
- Applications to healthy & damaged brains
- Suitable for beginners

For: Neuroscience students, Computational modelers, AI practitioners

Applications: Healthcare, Research, Education

Available: Amazon, Springer, Barnes & Noble

Author:

Allen B. Downey

O'Reilly Media**Free Online Access****Think Python: How to Think Like a Computer Scientist***An Introduction to Software Design*

Teaches Python programming with focus on computational thinking and problem-solving skills

Key Topics:

- ① Variables & Expressions
- ② Functions & Conditionals
- ③ Iteration & Strings
- ④ Lists & Dictionaries

Key Features:

- Beginner-friendly approach
- Emphasis on debugging
- Practical exercises
- Free PDF available

For: Programming beginners, Computer science students, Self-learners**Available:** greenteapress.com, O'Reilly, Amazon

Author:

Robert Johansson

Apress**Second Edition****Numerical Python: Scientific Computing and Data Science Applications***with NumPy, SciPy, Matplotlib*

Comprehensive guide to numerical computing with Python for science and engineering

Key Libraries:

- ① NumPy arrays
- ② SciPy algorithms
- ③ Matplotlib visualization
- ④ SymPy symbolic math

Key Features:

- Practical examples
- Performance optimization
- Data analysis techniques
- Machine learning basics

For: Scientists, Engineers, Data analysts, Researchers**Applications:** Physics, Engineering, Finance, Data Science**Available:** Apress, Amazon, Springer

Official Resource
docs.python.org
Free Online

Python Official Documentation

The Authoritative Reference for Python

Official documentation covering all aspects of Python language and standard library

Key Sections:

- ➊ Tutorial (Beginner)
- ➋ Language Reference
- ➌ Library Reference
- ➍ Python HOWTOs

Key Features:

- Complete API reference
- Code examples
- Version-specific docs
- PEP standards

For: All Python users, from beginners to experts

Languages: English, Japanese, French, Spanish, more

Available: docs.python.org (free)

Scientific Python
scipy.org
Free Online

NumPy and SciPy Documentation

Scientific Computing with Python

Documentation for the core scientific computing stack in Python

NumPy Features:

- ① N-dimensional arrays
- ② Broadcasting
- ③ Linear algebra
- ④ Random sampling

SciPy Features:

- Optimization
- Integration
- Interpolation
- Signal processing

For: Scientists, Engineers, Data scientists, Researchers

Related: Matplotlib, pandas, scikit-learn docs

Available: numpy.org, scipy.org (free)

Section 12

Thank You!

Subsection 1

Questions?

Questions?

Remember: The best way to learn is by doing. Practice with real datasets!