WEEK 1

Advanced Python for AI

Master Python fundamentals essential for machine learning and AI development

Day 1	**Python Review**	Core concepts, list comprehensions, modular code
Day 2	**Functions**	Parameters, scope, decorators, utility libraries
Day 3	**Lambda + .apply()**	Functional programming, pandas transformations
Day 4	**Dictionaries**	Nested structures, comprehensions, configs
Day 5	**JSON + File Handling**	Data persistence, Git version control

Focus: Writing clean, modular, production-ready Python for ML pipelines

al

DAY 1

Python Review

Refreshing Core Concepts Essential for AI

OBJECTIVES

- Review core Python concepts essential for AI
- Write modular and reusable functions
- Understand lists, loops, and comprehensions
- Apply clean coding principles

ACTIVITY

Refactor a script with repeated code into clean, modular functions.

ASSESSMENT

Code submission: Python script demonstrating modular design.

al

DAY 1 • PYTHON REVIEW

Why Python for AI?

Python dominates AI development due to its extensive ecosystem

57%
Data Scientists' Primary Language

40%
Reduced Development Time

#1
ML Library Ecosystem

10M+
Active Python Developers

1. **Rich Library Ecosystem** — NumPy, pandas, TensorFlow, PyTorch, scikit-learn
2. **Clean, Readable Syntax** — Focus on algorithms, not implementation details
3. **Strong Community Support** — Extensive documentation, tutorials, and forums
4. **Interactive Development** — Jupyter notebooks for rapid prototyping and sharing

al

DAY 1 · PYTHON REVIEW

Variables & Data Types

The building blocks of every Python program

INTEGER

int

```
age = 25
```

FLOAT

float

```
price = 19.99
```

STRING

str

```
name = "AI"
```

BOOLEAN

bool

```
active = True
```

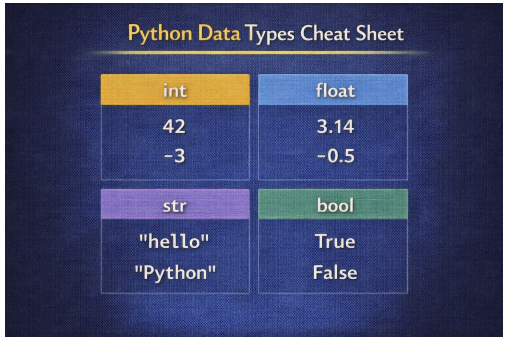
VARIABLE ASSIGNMENT

```
# Python variable assignment
learning_rate = 0.001
epochs = 100
model_name = "neural_net"
is_trained = False

# Multiple assignment
x, y, z = 1, 2, 3
```

AI Tip

Use descriptive variable names like learning_rate instead of lr for better readability.

Consistent naming makes code self-documenting and easier to maintain in team environments.



al

DAY 1 · PYTHON REVIEW

Lists & Collections

Storing and manipulating sequences of data

CREATING LISTS

```
features = [0.5, 1.2, 3.4, 2.1]
labels = ["cat", "dog", "bird"]

# Common operations
features.append(4.5)   # Add element
features.pop()         # Remove last
len(features)          # Get length
```

SLICING - ESSENTIAL FOR AI

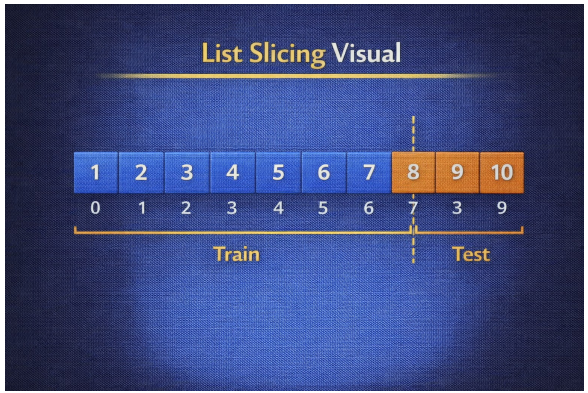
```
data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

train = data[:8]   # First 8 elements
test = data[8:]    # Last 2 elements

# 80/20 split pattern
train = data[:int(len(data)*0.8)]
```

AI Use Cases

- Store feature vectors and embeddings
- Hold training/test data splits
- Batch data samples for training
- Store model predictions and labels

Key Insight

List slicing is fundamental to creating train/test splits, batching data, and extracting subsets for cross-validation.



al

DAY 1 · PYTHON REVIEW

Control Flow

Making decisions and repeating actions in code

CONDITIONALS (if/elif/else)

```
accuracy = 0.85

if accuracy >= 0.9:
    print("Excellent model!")
elif accuracy >= 0.8:
    print("Good model")
else:
    print("Needs improvement")
```

FOR LOOP

```
# Iterate over epochs
for epoch in range(10):
    train_model()
    print(f"Epoch {epoch} complete")

# Enumerate for index + item
for i, item in enumerate(data):
    process(i, item)
```

WHILE LOOP

```
while loss > 0.01:
    loss = update_weights()
    if converged: break
```

AI Pattern

Training loops typically use for loops with range(epochs), while convergence checks use while loops.

al

DAY 1 · PYTHON REVIEW

List Comprehensions

Concise syntax for creating and transforming lists

Syntax

[expression for item in iterable]

[expression for item in iterable if condition]

```
squares = [x**2 for x in range(10)]
```

TRADITIONAL (5 lines)

```
squares = []
for x in range(10):
    squares.append(x**2)

# Result: [0, 1, 4, 9, 16, ...]
```

LIST COMPREHENSION (1 line)

```
squares = [x**2 for x in range(10)]

# Same result, 60% less code
# Standard in professional AI codebases
```

AI EXAMPLES

```
normalized = [(x - min_val) / range_val for x in features]    # Feature scaling
valid = [s for s in samples if s["label"] is not None]        # Filter invalid
```

al

DAY 1 · PYTHON REVIEW

String Operations

Essential for text preprocessing in NLP tasks

COMMON STRING METHODS

```
text = "  Hello World  "

text.lower()     # "  hello world  "
text.upper()     # "  HELLO WORLD  "
text.strip()     # "Hello World"
text.split()     # ["Hello", "World"]
text.replace("o", "0")  # "Hell0 W0rld"
```

F-STRINGS (Python 3.6+)

```
model = "ResNet"
acc = 0.956

print(f"{model}: {acc:.2%}")
# Output: ResNet: 95.60%

print(f"Epoch {epoch:03d}/100")
# Output: Epoch 005/100
```

NLP PREPROCESSING PIPELINE

```
def preprocess(text):
    text = text.lower()          # Lowercase
    text = text.strip()          # Remove whitespace
    tokens = text.split()        # Tokenize
    return tokens
```

NLP Importance

String operations are fundamental for cleaning text data before feeding to models.

al

DAY 1 · PYTHON REVIEW

Writing Modular Code

Breaking down complex workflows into reusable components

BEFORE: Repeated Code ❌

```
# Process dataset 1
data1 = load_csv("train.csv")
data1 = data1.dropna()
data1["price"] *= 1.1
save_csv(data1, "train_clean.csv")

# Process dataset 2 (same code!)
# 8+ lines repeated - high bug risk
```

AFTER: Modular Function ✓

```
def process_dataset(in_path, out_path):
    """Clean and transform dataset."""
    data = load_csv(in_path)
    data = data.dropna()
    data["price"] *= 1.1
    save_csv(data, out_path)

# Now just 2 calls! Easy to test.
```

Problem

- Code duplication increases bug risk
- Changes needed in multiple places
- Hard to test and maintain

Solution

- Single function to update
- Easy to write unit tests
- Consistent behavior guaranteed

DAY 1 · PYTHON REVIEW

From Scripts to Reusable AI Utilities

Turn one-off scripts into testable, reusable ML components

EXAMPLE: METRIC UTILITIES

```
from typing import Sequence, Dict

def accuracy(preds: Sequence[int], labels: Sequence[int]) ->
float:
    """Compute classification accuracy."""
    correct = sum(p == y for p, y in zip(preds, labels))
    return correct / len(labels) if labels else 0.0

def classification_report(preds, labels) -> Dict[str, float]:
    """Return a metrics dict (extend later)."""
    return {"accuracy": accuracy(preds, labels)}
```

USAGE

```
# Import and use across projects
from metrics import accuracy, classification_report
acc = accuracy(predictions, ground_truth)
```

Engineering Notes

- Keep functions small - single concept
- Use type hints to document expectations
- Place in metrics.py module

ML Impact

Shared metric utilities reduce bugs when comparing different models.

al

DAY 1 · ACTIVITY

Refactoring Challenge

Transform repeated code into clean, modular functions

Your Task

1. Identify repeated code patterns
2. Create functions with clear parameters
3. Add docstrings for documentation
4. Test that output matches original
5. Use meaningful variable names

SAMPLE INPUT

```
cat_a = [1, 2, 3, 4, 5]
mean_a = sum(cat_a) / len(cat_a)
max_a = max(cat_a)
```

EXPECTED OUTPUT

```
def calculate_stats(data):
    return {"mean": ..., "max": ...}
```

Requirements

• At least 3 reusable functions • Docstrings for each • Descriptive variable names • No repeated code blocks | Time: 20-30 minutes

al

DAY 1 • ASSESSMENT

Code Submission

A Python script demonstrating modular design

Submission Checklist

- ✓ At least 3 reusable functions
- ✓ Docstrings for each function
- ✓ Descriptive variable names
- ✓ No repeated code blocks
- ✓ Working output matches original

Grading Criteria

40% - Functionality (code works correctly)

30% - Code Quality (clean, readable)

30% - Documentation (docstrings, comments)

EXAMPLE STRUCTURE

```
def calculate_stats(data):
    """Calculate statistics for a list of numbers.
    Args: data: List of numerical values
    Returns: Dict with mean, max, min values
    """
    return {"mean": sum(data)/len(data), "max": max(data), "min": min(data)}
```

al

DAY 2

Functions

Building Modular AI Code

OBJECTIVES

- Master advanced function concepts in depth
- Understand scope and return values
- Learn *args, **kwargs for flexibility
- Create reusable utility libraries

ACTIVITY

Create a library of utility functions for common data tasks.

ASSESSMENT

Mini-quiz on function scope and arguments.

al

DAY 2 • FUNCTIONS

Function Fundamentals

The building blocks of maintainable AI systems

BASIC SYNTAX

```
def function_name(parameters):
    """Docstring describing function."""
    # Function body
    return result
```

EXAMPLE

```
def calculate_accuracy(correct, total):
    """Calculate accuracy as percentage."""
    return (correct / total) * 100

acc = calculate_accuracy(85, 100)  # 85.0
```

70%

Reduction in code duplication

Key Concepts

• def keyword for definition • Parameters and arguments • Return values • Scope and variable lifetime • Docstrings for documentation

al

DAY 2 · FUNCTIONS

Parameters and Arguments

Flexible ways to pass data into functions

POSITIONAL ARGS

```
def train(model, epochs, lr):
    pass

train("ResNet", 100, 0.001)
```

KEYWORD ARGS

```
train(
    model="ResNet",
    lr=0.001,
    epochs=100
)
```

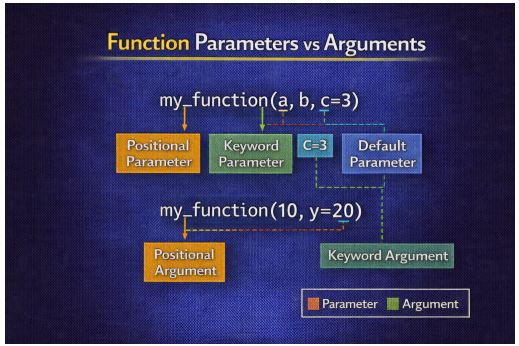
DEFAULT VALUES

```
def train(model,
          epochs=100,
          lr=0.001):
    pass
```

Best Practice

Put required parameters first, optional parameters with defaults last. This makes function calls cleaner and intentions clearer.

```
train("ResNet")                  # Uses all defaults
train("ResNet", epochs=200)      # Override just epochs
```



al

DAY 2 • FUNCTIONS

*args and **kwargs

Handling flexible function signatures

*args - Variable Positional

```
def sum_all(*numbers):
    """Accept any number of arguments."""
    return sum(numbers)

sum_all(1, 2, 3, 4, 5)  # 15
```

**kwargs - Variable Keyword

```
def create_config(**settings):
    """Accept any keyword arguments."""
    return settings

config = create_config(lr=0.01, epochs=50)
```

AI USE CASE: FLEXIBLE MODEL CONFIGURATION

```
def build_model(architecture, **hyperparams):
    """Build model with flexible hyperparameters."""
    model = load_architecture(architecture)
    model.configure(**hyperparams)    # Pass all settings to model
    return model

model = build_model("ResNet", lr=0.001, dropout=0.5, batch_size=32)
```

al

DAY 2 · FUNCTIONS

Return Values

Getting results back from functions

SINGLE RETURN

```
def get_accuracy(pred, labels):
    correct = sum(
        p == l for p, l
        in zip(pred, labels)
    )
    return correct / len(labels)
```

MULTIPLE RETURNS

```
def train_test_split(data):
    split =
int(len(data)*0.8)
    return data[:split],
data[split:]

train, test =
train_test_split(data)
```

RETURN DICT

```
def evaluate(model):
    return {
        "accuracy": 0.95,
        "precision": 0.93,
        "recall": 0.94
    }
```

Pro Tip

Use tuple unpacking for multiple returns, or return a dictionary for named results that are self-documenting.

al

DAY 2 • FUNCTIONS

Variable Scope

Understanding where variables live

LOCAL SCOPE

```
def calculate():
    x = 10          # Local variable
    return x

print(x)            # Error! x not defined
# x only exists inside the function
```

GLOBAL SCOPE

```
learning_rate = 0.001   # Global

def train():
    print(learning_rate)  # Can READ global

def update_lr():
    global learning_rate  # Declare global
    learning_rate = 0.0001
```

LEGB Rule

Local → Enclosing → Global → Built-in

Python searches for variables in this order.

Best Practice

Avoid global variables. Pass values as parameters instead for clearer, testable code.

al

Docstrings

Documenting functions for collaboration

GOOGLE STYLE DOCSTRING

```
def normalize(data, method="minmax"):
    """Normalize data using specified method.

    Args:
        data: List of numerical values
        method: Normalization method ("minmax" or "zscore")

    Returns:
        List of normalized values

    Raises:
        ValueError: If method not recognized
    """
    pass
```

Why Document?

- IDE shows docs on hover
- Auto-generates API docs
- Helps team collaboration
- Essential for open-source
- Self-documenting code
- Easier maintenance

al

DAY 2 · FUNCTIONS

Higher-Order Functions

Functions that work with other functions

PASSING FUNCTIONS AS ARGS

```
def apply_to_all(func, data):
    return [func(x) for x in data]

numbers = [1, 2, 3, 4]
squared = apply_to_all(
    lambda x: x**2, numbers
)  # [1, 4, 9, 16]
```

BUILT-IN HIGHER-ORDER

```
# map - apply to each
list(map(str.upper, ["a", "b"]))

# filter - keep if True
list(filter(lambda x: x > 0, nums))

# sorted with key function
sorted(data, key=lambda x: x["val"])
```

AI Use Case

Building preprocessing pipelines by composing transformation functions - each step is a function passed to the next.

al

DAY 2 · FUNCTIONS

Decorators

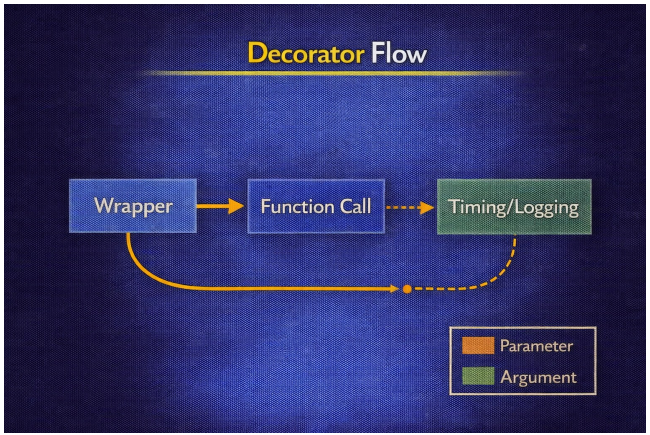
Adding functionality to existing functions

TIMING DECORATOR

```
import time

def timer(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"{func.__name__} took {time.time()-start:.2f}s")
        return result
    return wrapper

@timer
def train_model(epochs):
    # Training code...
    pass
```

Common Use Cases

- Timing/profiling functions
- Logging function calls
- Caching results (memoization)
- Input validation
- Authentication checks
- Retry logic



al

DAY 2 • FUNCTIONS

Building a Utility Library

Common functions for AI data tasks

data_utils.py

```
def load_data(filepath):
    """Load data from CSV file."""
    pass

def clean_missing(df, strategy="drop"):
    """Handle missing values."""
    pass

def normalize(data, method="minmax"):
    """Normalize numerical data."""
    pass

def split_data(X, y, test_size=0.2):
    """Split into train/test sets."""
    pass
```

Usage

```
from data_utils import (
    load_data,
    clean_missing,
    normalize
)
```

Benefits

- Reuse across projects
- Consistent behavior
- Easy to test
- Team standardization

al

DAY 2 • FUNCTIONS

Designing a Reusable data_utils Module

Turn common operations into a stable utility module

EXAMPLE STRUCTURE (data_utils.py)

```
from pathlib import Path
import pandas as pd

def load_csv(path: Path) -> pd.DataFrame:
    """Load CSV with minimal assumptions."""
    return pd.read_csv(path)

def drop_missing(df: pd.DataFrame, cols: list[str]) ->
pd.DataFrame:
    """Drop rows with missing values in required columns."""
    return df.dropna(subset=cols)

def add_ratio(df: pd.DataFrame, num: str, denom: str, out: str):
    """Add a ratio feature like price_per_sqft."""
    df = df.copy()
    df[out] = df[num] / df[denom]
    return df
```

Engineering Notes

- Keep I/O separate from transforms
- Return new DataFrames
- Use explicit column names

ML Impact

A stable data_utils module plugs into pipelines without rewrites. Consistency reduces bugs.

al

DAY 2 • ACTIVITY

Create Utility Functions

Build a library of reusable data functions

Required Functions

1. calculate_statistics(data)
2. normalize_data(data, method)
3. remove_outliers(data, threshold)
4. train_test_split(data, ratio)
5. encode_labels(labels)

Bonus Points

- Add type hints for all parameters
- Include comprehensive docstrings
- Write test cases for each function
- Handle edge cases (empty lists, etc.)

Time: 30-45 minutes

Create data_utils.py with at least 5 utility functions. Each function should have a docstring and handle edge cases.

al

DAY 2 • ASSESSMENT

Mini-Quiz: Functions

Test your understanding of function concepts

Topics Covered

- Function definition and calling
- Parameters vs arguments
- *args and **kwargs usage
- Return values (single, multiple, dict)
- Variable scope (LEGB rule)
- Docstrings and documentation

Sample Questions

1. Difference between parameter and argument?
2. When would you use **kwargs?
3. What does LEGB stand for?
4. Write a function returning multiple values.

Format

10 multiple choice + 2 coding questions | Time: 15 minutes

al

DAY 3

Lambda + .apply()

Functional Programming for Data Transformation

OBJECTIVES

- Write and use lambda functions for concise operations
- Apply functions to Pandas DataFrames using .apply()
- Understand performance considerations
- Build data cleaning pipelines

ACTIVITY

Use .apply() with lambda to clean a column (e.g., remove currency symbols).

ASSESSMENT

Notebook showing use of lambda and apply.

al

DAY 3 • LAMBDA + .APPLY()

Lambda Functions Basics

Anonymous inline functions

Syntax

lambda arguments: expression → Returns result of expression automatically

REGULAR FUNCTION

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

result = square(5)  # 25
```

LAMBDA EQUIVALENT

```
square = lambda x: x ** 2

result = square(5)  # 25
```

Key Characteristics

• Single expression only - no multiple statements • Implicit return - no return keyword needed • No if/for blocks inside (use ternary for conditions) • Best for simple, one-time operations in functional contexts

al

DAY 3 • LAMBDA + .APPLY()

Lambda with Built-in Functions

Power combinations for data processing

WITH map()

```
numbers = [1, 2, 3, 4, 5]

squared = list(map(
    lambda x: x**2,
    numbers
))  # [1, 4, 9, 16, 25]
```

WITH filter()

```
numbers = [-2, -1, 0, 1, 2]

positive = list(filter(
    lambda x: x > 0,
    numbers
))  # [1, 2]
```

WITH sorted()

```
students = [
  {"name": "A", "grade":
85},
  {"name": "B", "grade":
92}
]
sorted(students,
  key=lambda s: s["grade"])
```

When to Use

Lambda functions shine when used with map(), filter(), sorted(), and pandas .apply() - anywhere you need a quick, inline function.

al

DAY 3 • LAMBDA + .APPLY()

The .apply() Method

Pandas' most powerful transformation tool

What is .apply()?

Applies a function to each element (Series) or row/column (DataFrame) in pandas.

```
prices_tax = prices.apply(lambda x: x * 1.1)
```

SERIES.APPLY()

```
import pandas as pd

prices = pd.Series([100, 200, 150, 300])
prices_tax = prices.apply(lambda x: x * 1.1)
# 110.0, 220.0, 165.0, 330.0
```

Why Use .apply()?

- Apply custom logic to data
- Bridge pandas and Python functions
- Handle complex transformations
- Row-wise or column-wise operations

al

DAY 3 • LAMBDA + .APPLY()

.apply() on DataFrames

Column-wise and row-wise operations

COLUMN-WISE (axis=0)

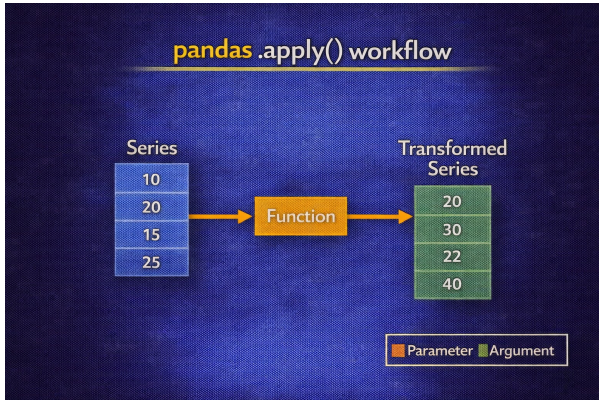
```
df = pd.DataFrame({
    "A": [1, 2, 3],
    "B": [4, 5, 6]
})

df.apply(lambda col: col.max())  # A:3, B:6
```

ROW-WISE (axis=1)

```
df["sum"] = df.apply(
    lambda row: row["A"] + row["B"],
    axis=1
)
# Adds A and B for each row
```

APPLYING TO SPECIFIC COLUMN

```
df['A_squared'] = df['A'].apply(lambda x: x**2)      # Square values in column A
df['B_log'] = df['B'].apply(lambda x: np.log(x))     # Log transform column B
```



DAY 3 • LAMBDA + .APPLY()

Lambda + apply() Examples

Real-world data transformations

TEXT CLEANING (NLP)

```
df["clean"] = df["text"].apply(
    lambda x: x.lower().strip()
)
```

FEATURE SCALING

```
df["norm"] = df["value"].apply(
    lambda x: (x - mean) / std
)
```

CATEGORICAL ENCODING

```
category_map = {"low": 0, "med": 1, "high": 2}
df["enc"] = df["cat"].apply(
    lambda x: category_map.get(x, -1))
```

HANDLING MISSING

```
df["filled"] = df["val"].apply(
    lambda x: 0 if pd.isna(x) else x
)
```

Data Cleaning Pipeline

These patterns are fundamental for NLP preprocessing and feature engineering in AI workflows.

al

DAY 3 • LAMBDA + .APPLY()

Performance Considerations

When to use each approach

VECTORIZED (PREFERRED) ⚡

```
# FAST - Use pandas/numpy operations
df["double"] = df["value"] * 2
df["sum"] = df["A"] + df["B"]
```

USING .apply() 🐢

```
# SLOWER - For complex logic only
df["result"] = df["text"].apply(
    complex_nlp_function
)
```

Performance Comparison

Vectorized: ~1x (baseline)

.apply() with lambda: ~10-100x slower

Python loop: ~100-1000x slower

Rule of Thumb

1. Try vectorized operations first
2. Use .apply() for complex logic
3. Never use Python loops on DataFrames

al

DAY 3 • LAMBDA + .APPLY()

Towards Vectorized ML Data Prep

Replace slow loops with vectorized operations

ANTI-PATTERN: ROW LOOP ❌

```
# Slow: explicit Python loop
def add_length_column(df):
    lengths = []
    for _, row in df.iterrows():
        text = row["text"]
        lengths.append(len(text.split()))
    df["text_length"] = lengths
    return df
```

BETTER: VECTORIZED ✓

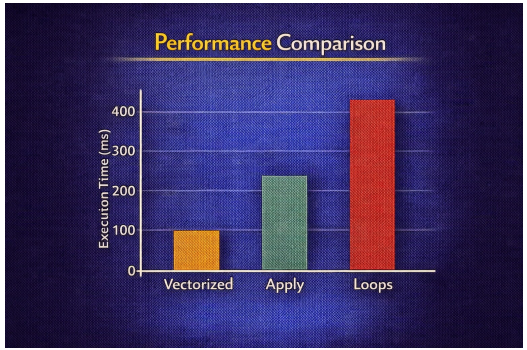
```
# Fast: use Series methods
def add_length_column_vectorized(df):
    df = df.copy()
    df["text_length"] =
df["text"].str.split().apply(len)
    return df

# Even better with pure vectorization:
# df["text_length"] =
df["text"].str.split().str.len()
```

ML Impact

Faster preprocessing = faster experiment iteration

Engineering Note

Prefer .str methods and .apply() over manual loops



al

DAY 3 • ACTIVITY

Clean a Dataset

Use .apply() with lambda to transform real data

Tasks

1. Clean price column (remove $ and ,)
2. Fill missing quantity with 0
3. Create total = price × quantity
4. Categorize price as low/med/high

SAMPLE DATA

```
product   | price      | quantity
Widget A  | $1,234.50  | 10
Widget B  | $567.89    | 5
Widget C  | $2,345.00  | null
```

Requirements

Use .apply() with lambda for at least 2 transformations • Include comments • Time: 25-35 minutes

al

DAY 3 • ASSESSMENT

Notebook Submission

Demonstrating lambda and .apply()

Required Sections

1. Data Loading - Load provided CSV
2. Data Exploration - shape, dtypes, sample
3. Cleaning with .apply() - 3+ transforms
4. Verification - Prove it worked
5. Reflection - When is vectorized better?

Grading Rubric

25% - Correct lambda syntax

25% - Proper .apply() usage

20% - Code runs without errors

15% - Clean, commented code

15% - Insightful reflection

al

DAY 4

Dictionaries

Python's Data Powerhouse

OBJECTIVES

- Manipulate nested dictionaries and complex structures
- Understand dictionary comprehensions
- Store and access ML configurations
- Master dict unpacking and merging

ACTIVITY

Parse a complex nested dictionary representing a student database.

ASSESSMENT

Code challenge: Extract specific data from nested structure.

al

DAY 4 • DICTIONARIES

Dictionary Fundamentals

Key-value data storage with O(1) lookup

CREATING DICTIONARIES

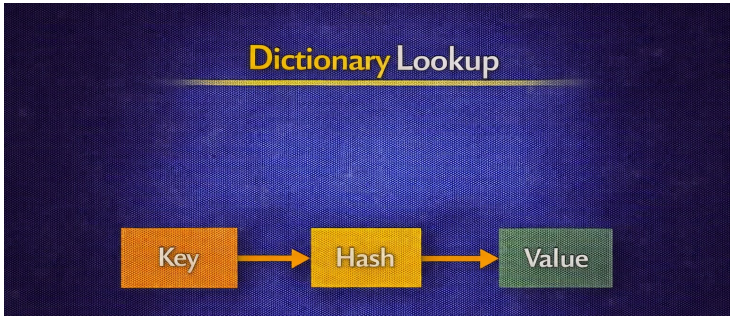
```
# Empty dictionary
config = {}

# With values
model_config = {
    "learning_rate": 0.001,
    "epochs": 100,
    "batch_size": 32
}
```

ACCESSING VALUES

```
# Direct access
lr = model_config["learning_rate"]

# Safe access with default
lr = model_config.get(
    "learning_rate", 0.001
)
```

Why O(1)?

Dictionaries use hash tables for instant key lookup regardless of size - essential for caching, config management, and fast data access.



al

DAY 4 • DICTIONARIES

Common Dictionary Operations

Essential methods for working with dicts

ADDING / UPDATING

```
config["dropout"] = 0.5
config["epochs"] = 200

config.update({
    "lr": 0.01,
    "momentum": 0.9
})
```

REMOVING

```
# Delete key
del config["dropout"]

# Remove & return
val = config.pop("epochs")

# Clear all
config.clear()
```

ITERATING

```
config.keys()
config.values()
config.items()

for k, v in config.items():
    print(f"{k}: {v}")
```

Concise dict creation

Syntax

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

SQUARE NUMBERS

```
squares = {
    x: x**2
    for x in range(5)
}  # {0:0, 1:1, 2:4, 3:9, 4:16}
```

FROM TWO LISTS

```
keys = ["a", "b", "c"]
values = [1, 2, 3]
combined = {
    k: v for k, v in
zip(keys, values)
}
```

LABEL ENCODING

```
labels = ["cat", "dog"]
label_to_int = {
    label: idx
    for idx, label
    in enumerate(labels)
}
```

AI Use Case

Label encoding, vocabulary mapping, feature indexing - all use dictionary comprehensions.

al

Nested Dictionaries

Complex hierarchical data structures

MODEL RESULTS STRUCTURE

```
model_results = {
    "experiment_1": {
        "model": "ResNet50",
        "metrics": {
            "accuracy": 0.92,
            "precision": 0.91,
            "recall": 0.93
        },
        "hyperparams": {"lr": 0.001, "epochs": 100}
    }
}
```

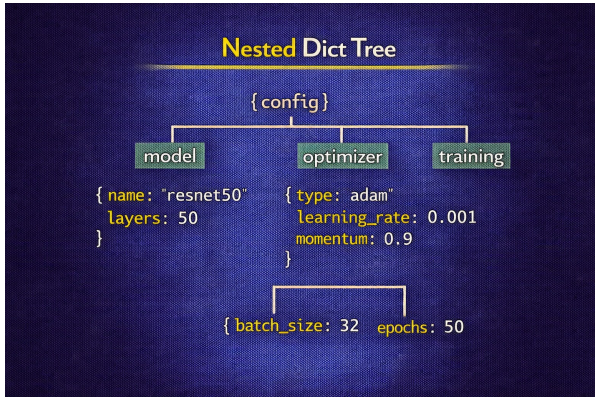
ACCESSING NESTED

```
acc = model_results\
    ["experiment_1"]\
    ["metrics"]\
    ["accuracy"]
```

SAFE ACCESS

```
acc = model_results\
    .get("exp_1", {})\
    .get("metrics", {})\
    .get("accuracy", 0)
```

Use .get() for safe nested access - avoids KeyError when keys might not exist.



al

DAY 4 • DICTIONARIES

Unpacking & Merging

Combining dictionaries efficiently

DICTIONARY UNPACKING (**)

```
default = {"lr": 0.001, "epochs": 100}
user = {"lr": 0.01, "momentum": 0.9}

# Merge - user overrides default
final = {**default, **user}
# {"lr": 0.01, "epochs": 100, "momentum": 0.9}
```

UNPACKING IN FUNCTION CALLS

```
def train(lr, epochs, batch):
    pass

config = {"lr": 0.001, "epochs": 100, "batch":
32}
train(**config)  # Unpacks as keyword args
```

Python 3.9+ Merge Operator

final_config = default_config | user_config # Cleaner syntax for merging

al

DAY 4 • DICTIONARIES

Model Configuration Patterns

Use nested dicts for training configurations

EXAMPLE: MODEL CONFIG DICT

```
model_config = {
    "architecture": "resnet18",
    "input_dim": 224,
    "num_classes": 10,
    "optimizer": {
        "type": "adam",
        "lr": 1e-3,
        "weight_decay": 1e-4,
    },
    "training": {
        "batch_size": 64,
        "epochs": 20,
    },
}
```

Engineering Notes

- Maps to JSON/YAML configs
- Group related settings
- Avoid flat configs with dozens of keys

ML Impact

Dict-based configs plug into JSON handling and training pipelines directly.

al

DAY 4 • ACTIVITY

Parse Student Database

Extract information from nested dictionaries

STUDENT DATABASE

```
students = {
    "S001": {
        "name": "Alice Chen",
        "courses": {
            "CS101": {"grade": 92, "credits": 3},
            "MATH201": {"grade": 88, "credits": 4},
            "AI301": {"grade": 95, "credits": 3}
        },
        "advisor": "Dr. Smith"
    },
    "S002": {...}
}
```

Tasks

1. Get Alice's AI301 grade
2. Calculate Bob's GPA
3. Find all students in CS101
4. Get average grade across all
5. Find student with highest GPA

Time: 25-30 minutes

al

DAY 4 • ASSESSMENT

Code Challenge: Data Extraction

Extract specific data from nested structures

Challenge Tasks

Given a nested dict of AI experiments:

1. Find best performing model name
2. List all learning rates used
3. Calculate average accuracy per model
4. Find experiments with >90% accuracy

Evaluation

50% - Correctness (outputs are correct)

25% - Code efficiency (no unnecessary loops)

25% - Code readability (clear variable names)

Time: 20 minutes

al

DAY 5

JSON + File Handling

Data Persistence for AI Workflows

OBJECTIVES

- Parse and manipulate JSON files for API data
- Perform file handling operations for data I/O
- Learn Git version control basics
- Create config-driven training scripts

ACTIVITY

Git & GitHub Mini-Workshop: Create repo, upload notebook, learn version control.

ASSESSMENT

GitHub repository with uploaded Jupyter notebooks.

al

DAY 5 · JSON + FILE HANDLING

JSON: The Universal Data Format

Standard for AI data exchange

JSON STRUCTURE

```
{
    "model": "GPT-4",
    "accuracy": 0.95,
    "tags": ["nlp", "transformer"],
    "config": {
        "layers": 96,
        "attention_heads": 96
    }
```

JSON ↔ Python Mapping

JSON object { } → Python dict

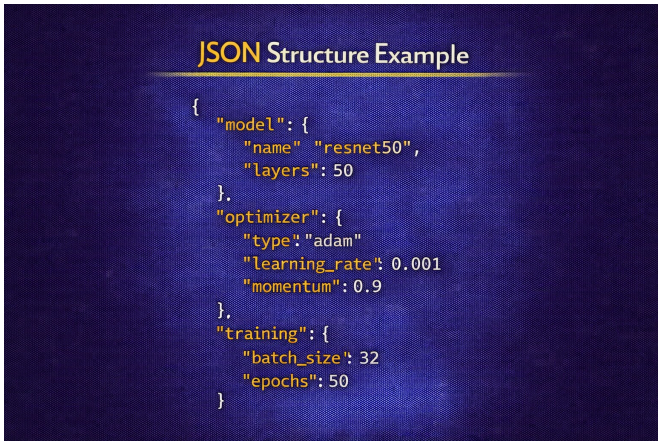
JSON array [ ] → Python list

JSON string → Python str

JSON number → Python int/float

true/false → True/False

Used in 90% of AI APIs and model serving systems. Human-readable and machine-parseable.



al

DAY 5 · JSON + FILE HANDLING

Python's json Module

Reading and writing JSON data

PARSE JSON STRING → PYTHON

```
import json
data = json.loads('{"name": "AI"}')
# Returns: {"name": "AI"}
```

PYTHON → JSON STRING

```
json_str = json.dumps({"name": "AI"})
# Returns: '{"name": "AI"}'
```

READ JSON FILE → PYTHON

```
with open("config.json", "r") as f:
    config = json.load(f)
```

PYTHON → WRITE JSON FILE

```
with open("config.json", "w") as f:
    json.dump(config, f, indent=4)
```

PRETTY PRINTING

```
print(json.dumps(data, indent=4, sort_keys=True))
```

al

DAY 5 • JSON + FILE HANDLING

File Handling in Python

Reading and writing files safely

THE with STATEMENT

```
with open("data.txt", "r") as f:
    content = f.read()

# File automatically closed after block
# Even if an exception occurs
```

File Modes

"r" - Read (default)

"w" - Write (overwrites existing)

"a" - Append to end

"r+" - Read and write

Why Use with?

• Automatically closes file when done • Handles exceptions properly • Prevents resource leaks • Cleaner than try/finally

al

DAY 5 · JSON + FILE HANDLING

Reading File Content

Different methods for different needs

READ ENTIRE FILE

```
with open("data.txt") as f:
    content = f.read()
    # Returns entire file
    # as single string
```

READ AS LIST

```
with open("data.txt") as f:
    lines = f.readlines()
    # Returns list of
    # lines with \n
```

LINE BY LINE

```
with open("big.txt") as f:
    for line in f:
        process(line)
# Memory efficient
```

Best Practice

For large files, iterate line-by-line to avoid loading entire file into memory. Use .strip() to remove newlines.

Creating and modifying files

WRITE STRING

```
with open("out.txt", "w") as f:
    f.write("Hello, AI!\n")
    f.write("Second line\n")
```

APPEND TO FILE

```
with open("log.txt", "a") as
f:
    f.write(f"[{time}]
Started\n")
```

WRITE JSON

```
results = {
    "accuracy": 0.95,
    "loss": 0.05
}
json.dump(results, f)
```

Note

'w' mode overwrites existing files. Use 'a' mode to append. Always use with statement for safety.

al

Complete AI Workflow

Config → Train → Save Results

COMPLETE WORKFLOW PATTERN

```
# 1. Load Configuration
with open("config.json", "r") as f:
    config = json.load(f)
print(f"Training with {config['epochs']} epochs...")

# 2. Training Process (simplified)
results = {"accuracy": 0.95, "loss": 0.04}

# 3. Save Results
try:
    with open("results.json", "w") as f:
        json.dump(results, f, indent=4)
    print("Results saved!")
except IOError as e:
    print(f"Failed to save: {e}")
```

Steps

1. Load hyperparams
2. Initialize model
3. Execute training
4. Save results

This pattern appears in 95% of production AI systems.

al

DAY 5 · JSON + FILE HANDLING

Config-Driven Training Scripts

Load hyperparameters from JSON

config.json

```
{
    "model": "resnet18",
    "epochs": 20,
    "batch_size": 64,
    "learning_rate": 0.001
}
```

PYTHON LOADER

```
def load_training_config(path: str):
    with open(path, "r") as f:
        return json.load(f)

cfg = load_training_config("config.json")
print(f"Training {cfg['model']}...")
```

Engineering Notes

- Easy to track and change hyperparameters
- Save config alongside results for tracking

ML Impact

Config files are the primary interface between researchers and training infrastructure.

al

DAY 5 · GIT WORKSHOP

Git & GitHub Workshop

Version control for AI projects

Workshop Objectives

1. Create and configure GitHub account
2. Understand Git fundamentals
3. Create repository for AI course
4. Upload Jupyter notebooks
5. Learn basic version control
6. Understand branching basics

Why Version Control?

- 98% of AI teams use Git
- Track experiment history
- Collaborate with team members
- Reproduce any previous state
- Professional development practice

BASIC GIT WORKFLOW

```
git add .  →  git commit -m "Add training script"  →  git push origin main
```

al

DAY 5 · ACTIVITY

Git & GitHub Workshop Tasks

Hands-on version control

Part 1: Setup (10 min)

1. Create GitHub account
2. Install Git locally
3. Configure name and email

Part 2: First Repo (10 min)

1. Create repo: ai-course-week1
2. Add README.md
3. Clone to local machine

Part 3: Upload (10 min)

1. Add Day 1-4 notebooks
2. Write commit messages
3. Push to GitHub

COMMANDS TO KNOW

```
git add .                          # Stage all changes
git commit -m "Add Day 1 notebook"  # Commit with message
git push origin main                # Push to GitHub
```

al

DAY 5 · ASSESSMENT

GitHub Repository Submission

Your AI course portfolio

Required Contents

- README.md with project description
- Day 1-4 Jupyter notebooks
- At least 5 commits with messages
- Clean file organization

REPOSITORY STRUCTURE

```
ai-course-week1/
├── README.md
├── day1/
│   └── python_review.ipynb
├── day2/
│   └── functions.ipynb
├── day3/
│   └── lambda_apply.ipynb
└── day4/
    └── dictionaries.ipynb
```

Grading

25% Repo structure • 25% Commit quality • 25% README • 25% All notebooks present

al

Week 1 Complete!

Advanced Python for AI - Key Takeaways

Day 1	Python fundamentals, list comprehensions, modular code design
Day 2	Functions, scope, decorators, building utility libraries
Day 3	Lambda functions, .apply() method, vectorized operations
Day 4	Dictionaries, nested structures, comprehensions, configs
Day 5	JSON handling, file I/O, Git version control

Key Takeaway

Clean, modular Python code is the foundation of professional ML development. Master these fundamentals before diving into ML libraries.

Next Week: NumPy, Pandas, and Data Visualization

al