Python Complete Course

Comprehensive Guide - Mobile Optimized

Python Complete Course - Comprehensive Guide

From Basics to Advanced (Django & Machine Learning)

• --

Table of Contents

• Load, explore, clean, and manipulate data using Pandas DataFrames*

32. [Model Training](#32-model-training)

• Prepare data, split train/test sets, train models, and evaluate accuracy*

33. [Decision Trees](#33-decision-trees)

• Understand decision tree algorithms, visualize trees, and improve model performance*

• --

Part 1: Python Fundamentals

1. Getting Started with Python

What is Python?

Python is a high-level, interpreted programming language known for its:

• **Simplicity**: Easy to read and write

• **Versatility**: Web development, data science, ML, automation

• **Large Ecosystem**: Millions of packages available

Setting Up Development Environment

1. Download from python.org

2. Install Python 3.7+

3. Verify installation:

```
python --version
```

1. Download from code.visualstudio.com

2. Install Python extension

3. Install Pylint (linter)

4. Install autopep8 (formatter)

Your First Python Program

```
# app.py print("Hello World")
```

• *Run the program:**

```
python app.py
```

• *Output:**

```
Hello World
```

VS Code Features for Python

| Feature | Purpose | Shortcut |

|---------|---------|----------|

| **Linting** | Find errors in code | Auto (Pylint) |

| **Formatting** | Auto-format code | Shift+Alt+F |

| **Debugging** | Step through code | F5 |

| **IntelliSense** | Auto-completion | Ctrl+Space |

| **Code Snippets** | Quick templates | Type + Tab |

Code Formatting (PEP 8)

PEP 8 is the style guide for Python code.

• *Bad Code:**

```
x=1 y=2 unit_price=3
```

• *Good Code (PEP 8 Compliant):**

```
x = 1 y = 2 unit_price = 3
```

• *Auto-format:** Save file with autopep8 enabled.

• --

2. Variables

What is a Variable?

A variable is a label for a memory location that stores data.

```
# Declaring variables students_count = 1000 rating = 4.99 is_published = True course_name = "Python Programming"
```

Variable Naming Rules

■ **DO:**

```
student_count = 100 # Use lowercase first_name = "John" # Use underscores age2 = 25 # Can contain numbers
```

■ **DON'T:**

```
2age = 25 # Can't start with number first-name = "John" # Can't use hyphens class = "Math" # Can't use keywords
```

Variable Naming Conventions

```
# Snake case (Python convention) student_count = 1000 first_name = "John" is_published = True # camelCase (Not typical in Python) studentCount = 1000 firstName = "John"
```

Multiple Assignment

```
# Assign same value to multiple variables x = y = z = 0 # Assign different values x, y, z = 1, 2, 3 print(f"x={x}, y={y}, z={z}") # Output: x=1, y=2, z=3
```

• --

3. Primitive Data Types

Python Built-in Primitive Types

Python has three main primitive types:

1. **Numbers** (int, float, complex)

2. **Strings** (str)

3. **Booleans** (bool)

Integers

```
# Whole numbers age = 25 students_count = 1000 temperature = -10 print(type(age)) # Output: <class 'int'>
```

Floats

```
# Numbers with decimal points rating = 4.99 temperature = 98.6 pi = 3.14159 print(type(rating)) # Output: <class 'float'>
```

Booleans

```
# True or False values is_published = True is_active = False print(type(is_published)) # Output: <class 'bool'>
```

Strings

```
# Text data course = "Python Programming" name = 'John Doe' message = """Multi-line string using triple quotes""" print(type(course)) # Output: <class 'str'>
```

Checking Type

```
x = 5 print(type(x)) # <class 'int'> y = 5.0 print(type(y)) # <class 'float'> name = "John" print(type(name)) # <class 'str'>
```

• --

4. Strings

Creating Strings

```
# Single quotes course = 'Python Programming' # Double quotes name = "John Doe" # Triple quotes (multi-line) message = """ Hi John, This is a multi-line string. Best regards, Admin """
```

Escape Sequences

```
# Newline message = "Line 1\nLine 2" print(message) # Output: # Line 1 # Line 2 # Tab message = "Name:\tJohn" print(message) # Name: John # Backslash path = "C:\\Users\\John" print(path) # C:\Users\John # Quote inside string message = "He said, \"Hello!\"" print(message) # He said, "Hello!"
```

String Length

```
course = "Python Programming" length = len(course) print(length) # Output: 18
```

String Indexing

```
course = "Python" # Access individual characters (0-indexed) print(course[0]) # P print(course[1]) # y print(course[-1]) # n (last character) print(course[-2]) # o (second-to-last)
```

String Slicing

```
course = "Python Programming" # [start:end] - end is exclusive print(course[0:6]) # Python print(course[:6]) # Python (start omitted = 0) print(course[7:]) # Programming (end omitted = till end) print(course[:]) # Python Programming (entire string) # Negative indices print(course[0:-1]) # Python Programmin (excludes last char)
```

String Methods

```
course = " Python Programming " # Convert to uppercase print(course.upper()) # " PYTHON PROGRAMMING " # Convert to lowercase print(course.lower()) # " python programming " # Title case print(course.title()) # " Python Programming " # Strip whitespace print(course.strip()) # "Python Programming" print(course.lstrip()) # "Python Programming " print(course.rstrip()) # " Python Programming" # Find substring print(course.find("Pro")) # 9 (index where "Pro" starts) print(course.find("xyz")) # -1 (not found) # Replace substring print(course.replace("Python", "Java")) # " Java Programming " # Check if substring exists print("Python" in course) # True print("Java" in course) # False print("Python" not in course) # False
```

• *Important:** String methods return NEW strings. Original string is unchanged.

```
course = "python" print(course.upper()) # PYTHON print(course) # python (original unchanged)
```

String Concatenation

```
# Using + operator first = "John" last = "Doe" full = first + " " + last print(full) # John Doe # Using f-strings (recommended) full = f"{first} {last}" print(full) # John Doe
```

Formatted Strings (f-strings)

```
name = "John" age = 25 # Old way (format method) message = "Hi, I'm {} and I'm {} years old".format(name, age) # New way (f-strings) - Python 3.6+ message = f"Hi, I'm {name} and I'm {age} years old" print(message) # Hi, I'm John and I'm 25 years old # Expressions in f-strings x = 10 y = 20 print(f"Sum: {x + y}") # Sum: 30 print(f"Length: {len(name)}") # Length: 4
```

• --

5. Numbers

Number Types

```
# Integer x = 10 print(type(x)) # <class 'int'> # Float y = 10.5 print(type(y)) # <class 'float'> # Complex (rarely used) z = 1 + 2j print(type(z)) # <class 'complex'>
```

Built-in Math Functions

```
# Round print(round(2.9)) # 3 print(round(2.5)) # 2 # Absolute value print(abs(-10)) # 10 print(abs(10)) # 10
```

Math Module

```
import math # Ceiling (round up) print(math.ceil(2.2)) # 3 # Floor (round down) print(math.floor(2.9)) # 2 # Square root print(math.sqrt(16)) # 4.0 # Power print(math.pow(2, 3)) # 8.0 # Constants print(math.pi) # 3.141592653589793 print(math.e) # 2.718281828459045
```

Getting User Input

```
# input() always returns a string name = input("What is your name? ") print(f"Hello {name}") # Get number (need to convert) age = input("How old are you? ") age = int(age) # Convert string to int print(f"You are {age} years old")
```

• --

6. Type Conversion

Type Conversion Functions

| Function | Purpose | Example |

|---|---|---|

| int() | Convert to integer | int("10") → 10 |

| float() | Convert to float | float("10.5") → 10.5 |

| bool() | Convert to boolean | bool(1) → True |

| str() | Convert to string | str(10) → "10" |

Examples

```
# String to int x = input("Enter a number: ") # Returns string x = int(x) # Convert to
integer y = x + 1 print(f"x={x}, y={y}") # String to float price = input("Enter price: ")
price = float(price) total = price * 1.1 # Add 10% tax print(f"Total: {total}") # Number to
string age = 25 message = "I am " + str(age) + " years old" print(message) # Or use
f-string (auto-converts) message = f"I am {age} years old" print(message)
```

Falsy and Truthy Values

• *Falsy Values** (evaluate to False):

```
print(bool(0)) # False print(bool("")) # False (empty string) print(bool(None)) # False
print(bool([])) # False (empty list) print(bool({})) # False (empty dict)
```

• *Truthy Values** (evaluate to True):

```
print(bool(1)) # True print(bool(-1)) # True print(bool("hello")) # True print(bool([1,
2])) # True print(bool("False")) # True (non-empty string!)
```

• --

7. Arithmetic Operations

Basic Operators

```
x = 10 y = 3 # Addition print(x + y) # 13 # Subtraction print(x - y) # 7 # Multiplication
print(x * y) # 30 # Division (always returns float) print(x / y) # 3.3333333333333335 #
Floor division (returns int) print(x // y) # 3 # Modulus (remainder) print(x % y) # 1 #
Exponentiation (power) print(x ** y) # 1000 (10^3)
```

Augmented Assignment Operators

```
x = 10 # Instead of: x = x + 3 x += 3 # x is now 13 x -= 3 # x = x - 3 x *= 3 # x = x * 3 x
/= 3 # x = x / 3 x //= 3 # x = x // 3 x %= 3 # x = x % 3 x **= 3 # x = x ** 3
```

Operator Precedence

```
# PEMDAS: Parentheses, Exponents, Multiplication/Division, Addition/Subtraction x = 10 + 3
* 2 # 16 (not 26) x = (10 + 3) * 2 # 26 x = 10 ** 2 * 3 # 300 (exponent first) x = (10 **
2) * 3 # 300 (same)
```

• --

8. Comparison Operators

Comparison Operators

```
x = 10 y = 20 # Greater than print(x > y) # False # Greater than or equal print(x >= y) #
False # Less than print(x < y) # True # Less than or equal print(x <= y) # True # Equal to
print(x == y) # False # Not equal to print(x != y) # True
```

Comparing Strings

```
# Lexicographical comparison (dictionary order) print("apple" < "banana") # True
print("apple" == "Apple") # False (case-sensitive) # Compare lengths name1 = "John" name2
= "Alexander" print(len(name1) < len(name2)) # True
```

• --

9. Logical Operators

Logical Operators

```
# AND - both must be True print(True and True) # True print(True and False) # False print(False and False) # False # OR - at least one must be True print(True or False) # True print(False or False) # False # NOT - inverts the value print(not True) # False print(not False) # True
```

Practical Examples

```
age = 25 has_license = True # Check if can drive can_drive = age >= 18 and has_license print(f"Can drive: {can_drive}") # Can drive: True # Check eligibility is_student = True is_senior = False discount = is_student or is_senior print(f"Eligible for discount: {discount}") # True # Negate condition is_premium = False is_free_user = not is_premium print(f"Free user: {is_free_user}") # True
```

Short-circuit Evaluation

```
# AND stops at first False result = False and print("This won't execute") # OR stops at first True result = True or print("This won't execute")
```

• --

10. If Statements

Basic If Statement

```
age = 18 if age >= 18: print("You are an adult") print("You can vote")
```

If-Else Statement

```
age = 16 if age >= 18: print("You are an adult") else: print("You are a minor")
```

If-Elif-Else Statement

```
age = 15 if age >= 18: print("Adult") elif age >= 13: print("Teenager") else: print("Child")
```

Nested If Statements

```
age = 25 has_license = True if age >= 18: if has_license: print("You can drive") else: print("You need a license") else: print("You're too young to drive")
```

Multiple Conditions

```
age = 25 is_student = True # Using AND if age >= 18 and is_student: print("Student discount available") # Using OR if age < 18 or age > 65: print("Special pricing") # Complex conditions temperature = 30 is_hot = temperature > 30 is_cold = temperature < 10 if is_hot: print("It's hot") elif is_cold: print("It's cold") else: print("It's moderate")
```

Ternary Operator

```
age = 20 # Long form if age >= 18: message = "Adult" else: message = "Minor" # Short form (ternary) message = "Adult" if age >= 18 else "Minor" print(message)
```

• --

11. While Loops

Basic While Loop

```
i = 1 while i <= 5: print(i) i += 1 # Output: 1 2 3 4 5
```

Infinite Loop (with break)

```
i = 1 while True: print(i) if i >= 5: break # Exit loop i += 1
```

While Loop with Else

```
i = 1 while i <= 3: print(i) i += 1 else: print("Loop completed") # Output: # 1 # 2 # 3 # Loop completed
```

Practical Example: Guessing Game

```
secret = 5 guess_count = 0 max_tries = 3 while guess_count < max_tries: guess = int(input("Guess: ")) guess_count += 1 if guess == secret: print("You won!") break else: print("Try again") else: print("You lost!")
```

• --

12. For Loops

Basic For Loop

```
# Iterate over a string for char in "Python": print(char) # Output: P y t h o n (each on new line)
```

For Loop with Lists

```
names = ["John", "Mary", "Bob"] for name in names: print(f"Hello {name}") # Output: # Hello John # Hello Mary # Hello Bob
```

Range Function

```
# range(stop) for i in range(5): print(i) # Output: 0 1 2 3 4 # range(start, stop) for i in range(2, 7): print(i) # Output: 2 3 4 5 6 # range(start, stop, step) for i in range(0, 10, 2): print(i) # Output: 0 2 4 6 8
```

For-Else Statement

```
for i in range(3): print(i) else: print("Loop completed") # Output: # 0 # 1 # 2 # Loop completed
```

Nested For Loops

```
for i in range(3): for j in range(2): print(f"({i}, {j})") # Output: # (0, 0) # (0, 1) # (1, 0) # (1, 1) # (2, 0) # (2, 1)
```

Break and Continue

```
# Break - exit loop for i in range(10): if i == 5: break print(i) # Output: 0 1 2 3 4 # Continue - skip to next iteration for i in range(5): if i == 2: continue print(i) # Output: 0 1 3 4
```

• --

13. Functions

Why Functions?

Functions help you:

• **Organize code** into reusable blocks

• **Avoid repetition** (DRY principle)

• **Improve readability**

• **Easier to test and maintain**

Defining a Function

```
def greet(): print("Hello") print("Welcome!") # Call the function greet() # Output: # Hello # Welcome!
```

Function Naming Conventions

```
# Good names (snake_case) def calculate_total(): pass def get_user_name(): pass def is_valid(): pass # Bad names def CalculateTotal(): # PascalCase (used for classes) pass def a(): # Not descriptive pass
```

• --

14. Parameters & Arguments

Parameters

```
# Function with parameters def greet(name): print(f"Hello {name}") # Call with argument greet("John") # Hello John greet("Mary") # Hello Mary
```

Multiple Parameters

```
def greet(first_name, last_name): print(f"Hello {first_name} {last_name}") greet("John", "Doe") # Hello John Doe
```

Positional vs Keyword Arguments

```
def greet(first_name, last_name): print(f"Hello {first_name} {last_name}") # Positional arguments (order matters) greet("John", "Doe") # Keyword arguments (order doesn't matter) greet(last_name="Doe", first_name="John") # Mix (positional first, then keyword) greet("John", last_name="Doe")
```

Default Parameters

```
def greet(name="Guest"): print(f"Hello {name}") greet("John") # Hello John greet() # Hello Guest
```

Multiple Default Parameters

```
def calculate_total(price, tax=0.1, shipping=5): total = price + (price * tax) + shipping return total print(calculate_total(100)) # 115.0 print(calculate_total(100, 0.2)) # 125.0 print(calculate_total(100, 0.2, 10)) # 130.0 print(calculate_total(100, shipping=0)) # 110.0
```

*args (Variable Arguments)

```
def sum_all(*numbers): total = 0 for num in numbers: total += num return total print(sum_all(1, 2, 3)) # 6 print(sum_all(1, 2, 3, 4, 5)) # 15
```

**kwargs (Keyword Arguments)

```
def save_user(**details): print(details) save_user(name="John", age=25, city="NYC") # Output: {'name': 'John', 'age': 25, 'city': 'NYC'}
```

• --

15. Return Values

Functions That Return Values

```
def square(x): return x * x result = square(5) print(result) # 25
```

Functions Without Return

```
def greet(name): print(f"Hello {name}") # Returns None by default result = greet("John")
print(result) # None
```

Multiple Return Values

```
def get_coordinates(): return 10, 20 # Returns tuple x, y = get_coordinates()
print(f"x={x}, y={y}") # x=10, y=20
```

Early Return

```
def is_even(number): if number % 2 == 0: return True return False print(is_even(4)) # True
print(is_even(5)) # False
```

Practical Examples

```
# Calculate area def calculate_area(length, width): return length * width area =
calculate_area(5, 3) print(f"Area: {area}") # Area: 15 # Convert temperature def
celsius_to_fahrenheit(celsius): return (celsius * 9/5) + 32 temp_f =
celsius_to_fahrenheit(25) print(f"{temp_f}°F") # 77.0°F # Find maximum def find_max(a, b):
if a > b: return a return b print(find_max(10, 20)) # 20
```

- --

16. Lists

Creating Lists

```
# Empty list numbers = [] # List with items numbers = [1, 2, 3, 4, 5] names = ["John",
"Mary", "Bob"] mixed = [1, "Hello", True, 3.14] print(numbers) # [1, 2, 3, 4, 5]
```

Accessing Items

```
names = ["John", "Mary", "Bob"] # Index (0-based) print(names[0]) # John print(names[1]) #
Mary print(names[-1]) # Bob (last item) print(names[-2]) # Mary (second-to-last)
```

Modifying Lists

```
numbers = [1, 2, 3] # Change item numbers[0] = 10 print(numbers) # [10, 2, 3] # Add item
(end) numbers.append(4) print(numbers) # [10, 2, 3, 4] # Insert item (at index)
numbers.insert(1, 20) print(numbers) # [10, 20, 2, 3, 4] # Remove item numbers.remove(2) #
Remove by value print(numbers) # [10, 20, 3, 4] # Remove by index numbers.pop(0)
print(numbers) # [20, 3, 4] # Remove last item numbers.pop() print(numbers) # [20, 3] #
Clear all items numbers.clear() print(numbers) # []
```

List Methods

```
numbers = [3, 1, 4, 1, 5, 9, 2] # Sort (modifies original) numbers.sort() print(numbers) #
[1, 1, 2, 3, 4, 5, 9] # Reverse numbers.reverse() print(numbers) # [9, 5, 4, 3, 2, 1, 1] #
Count occurrences print(numbers.count(1)) # 2 # Find index print(numbers.index(5)) # 1 #
Check if exists print(5 in numbers) # True print(10 in numbers) # False # Length
print(len(numbers)) # 7
```

List Slicing

```
numbers = [1, 2, 3, 4, 5] print(numbers[0:3]) # [1, 2, 3] print(numbers[:3]) # [1, 2, 3]
print(numbers[2:]) # [3, 4, 5] print(numbers[::2]) # [1, 3, 5] (every 2nd)
```

```
print(numbers[::-1]) # [5, 4, 3, 2, 1] (reverse)
```

List Comprehension

```
# Create list of squares numbers = [1, 2, 3, 4, 5] squares = [n * n for n in numbers]
print(squares) # [1, 4, 9, 16, 25] # With condition even_squares = [n * n for n in numbers
if n % 2 == 0] print(even_squares) # [4, 16]
```

2D Lists (Matrices)

```
matrix = [ [1, 2, 3], [4, 5, 6], [7, 8, 9] ] print(matrix[0]) # [1, 2, 3]
print(matrix[0][0]) # 1 print(matrix[1][2]) # 6
```

• --

17. Tuples

Creating Tuples

```
# Tuples are immutable (cannot be changed) coordinates = (1, 2, 3) print(coordinates) #
(1, 2, 3) # Single item tuple (need comma) single = (1,) print(type(single)) # <class
'tuple'>
```

Accessing Tuple Items

```
coordinates = (10, 20, 30) print(coordinates[0]) # 10 print(coordinates[-1]) # 30
```

Unpacking Tuples

```
coordinates = (10, 20) x, y = coordinates print(f"x={x}, y={y}") # x=10, y=20
```

Tuples vs Lists

| Feature | List | Tuple |

|---------|------|-------|

| **Mutable** | Yes | No |

| **Syntax** | [1, 2, 3] | (1, 2, 3) |

| **Performance** | Slower | Faster |

| **Use Case** | Dynamic data | Fixed data |

```
# List (mutable) numbers = [1, 2, 3] numbers[0] = 10 # ✓ Works # Tuple (immutable)
coordinates = (1, 2, 3) # coordinates[0] = 10 # ✗ Error!
```

• --

18. Dictionaries

Creating Dictionaries

```
# Empty dictionary person = {} # Dictionary with data person = { "name": "John", "age": 25,
"city": "NYC" } print(person)
```

Accessing Values

```
person = {"name": "John", "age": 25} # Using square brackets print(person["name"]) # John
# Using get() method (safer) print(person.get("name")) # John print(person.get("email")) #
None (no error) print(person.get("email", "N/A")) # N/A (default value)
```

Modifying Dictionaries

```
person = {"name": "John", "age": 25} # Update existing key person["age"] = 26 # Add new key person["email"] = "john@example.com" print(person) # {'name': 'John', 'age': 26, 'email': 'john@example.com'} # Remove key del person["email"] print(person) # {'name': 'John', 'age': 26}
```

Dictionary Methods

```
person = {"name": "John", "age": 25, "city": "NYC"} # Get all keys print(person.keys()) # dict_keys(['name', 'age', 'city']) # Get all values print(person.values()) # dict_values(['John', 25, 'NYC']) # Get all items (key-value pairs) print(person.items()) # dict_items([('name', 'John'), ('age', 25), ('city', 'NYC')]) # Check if key exists print("name" in person) # True print("email" in person) # False # Clear all items person.clear() print(person) # {}
```

Looping Through Dictionaries

```
person = {"name": "John", "age": 25, "city": "NYC"} # Loop through keys for key in person: print(key) # Loop through values for value in person.values(): print(value) # Loop through key-value pairs for key, value in person.items(): print(f"{key}: {value}") # Output: # name: John # age: 25 # city: NYC
```

• --

19. Exception Handling

Try-Except

```
try: age = int(input("Age: ")) income = 20000 risk = income / age print(f"Risk: {risk}") except ValueError: print("Invalid age!") except ZeroDivisionError: print("Age cannot be 0!")
```

Catching All Exceptions

```
try: age = int(input("Age: ")) risk = 20000 / age except Exception as ex: print(f"Error: {ex}")
```

Try-Except-Else-Finally

```
try: file = open("data.txt") age = int(file.read()) except FileNotFoundError: print("File not found") except ValueError: print("Invalid data") else: print("File read successfully") finally: print("This always executes")
```

Raising Exceptions

```
def calculate_xfactor(age): if age <= 0: raise ValueError("Age cannot be 0 or less") return 10 / age try: calculate_xfactor(-1) except ValueError as error: print(error)
```

• --

20. Classes & Objects

What is a Class?

A class is a blueprint for creating objects.

```
# Define a class class Point: def __init__(self, x, y): self.x = x self.y = y def move(self): print("Moving") def draw(self): print("Drawing") # Create object (instance) point1 = Point(10, 20) print(point1.x) # 10 point1.move() # Moving
```

Constructor (__init__)

```
class Person: def __init__(self, name, age): self.name = name self.age = age def greet(self): print(f"Hi, I'm {self.name}") # Create instances john = Person("John", 25) mary = Person("Mary", 30) john.greet() # Hi, I'm John mary.greet() # Hi, I'm Mary
```

Class vs Instance Attributes

```
class Person: # Class attribute (shared by all instances) species = "Human" def __init__(self, name): # Instance attribute (unique to each instance) self.name = name p1 = Person("John") p2 = Person("Mary") print(p1.species) # Human print(p2.species) # Human print(p1.name) # John print(p2.name) # Mary
```

Class Methods

```
class Person: def __init__(self, name): self.name = name def greet(self): print(f"Hello, I'm {self.name}") @classmethod def from_birth_year(cls, name, birth_year): age = 2025 - birth_year return cls(name) # Regular instantiation p1 = Person("John") # Using class method p2 = Person.from_birth_year("Mary", 1995)
```

Magic Methods

```
class Point: def __init__(self, x, y): self.x = x self.y = y def __str__(self): return f"({self.x}, {self.y})" def __eq__(self, other): return self.x == other.x and self.y == other.y p1 = Point(1, 2) p2 = Point(1, 2) print(p1) # (1, 2) print(p1 == p2) # True
```

• --

21. Inheritance

Basic Inheritance

```
# Parent class class Animal: def __init__(self, name): self.name = name def eat(self): print(f"{self.name} is eating") # Child class class Dog(Animal): def bark(self): print("Woof!") # Usage dog = Dog("Buddy") dog.eat() # Buddy is eating dog.bark() # Woof!
```

Method Overriding

```
class Animal: def make_sound(self): print("Some sound") class Dog(Animal): def make_sound(self): print("Woof!") class Cat(Animal): def make_sound(self): print("Meow!") dog = Dog() cat = Cat() dog.make_sound() # Woof! cat.make_sound() # Meow!
```

Super() Function

```
class Animal: def __init__(self, name): self.name = name class Dog(Animal): def __init__(self, name, breed): super().__init__(name) # Call parent constructor self.breed = breed dog = Dog("Buddy", "Golden Retriever") print(dog.name) # Buddy print(dog.breed) # Golden Retriever
```

• --

22. Modules

What is a Module?

A module is a file containing Python code (functions, classes, variables).

Creating a Module

```
# converters.py def kg_to_lbs(kg): return kg * 2.20462 def lbs_to_kg(lbs): return lbs / 2.20462
```

Importing a Module

```
# app.py import converters weight_kg = 70 weight_lbs = converters.kg_to_lbs(weight_kg)
print(weight_lbs) # 154.3234
```

Import Specific Functions

```
from converters import kg_to_lbs weight_lbs = kg_to_lbs(70) print(weight_lbs)
```

Import with Alias

```
import converters as conv weight_lbs = conv.kg_to_lbs(70) print(weight_lbs)
```

Built-in Modules

```
# Math module import math print(math.sqrt(16)) # 4.0 print(math.pi) # 3.141592653589793 #
Random module import random print(random.randint(1, 10)) # Random number 1-10
print(random.choice([1, 2, 3, 4, 5])) # Random choice # Datetime module from datetime
import datetime now = datetime.now() print(now) # Current date and time
```

- --

Part 2: Django Web Development

23. Django Introduction

What is Django?

Django is a high-level Python web framework for building web applications quickly.

• *Key Features:**

• **Fast development**: Less code, more features

• **Secure**: Built-in protection against common attacks

• **Scalable**: Used by Instagram, Pinterest, NASA

• **Batteries included**: Admin panel, ORM, authentication

Installing Django

```
# Create virtual environment python -m venv venv # Activate (Windows)
venv\Scripts\activate # Activate (Mac/Linux) source venv/bin/activate # Install Django pip
install django
```

Creating a Django Project

```
# Create project django-admin startproject vidly # Project structure vidly/ ■■■ manage.py
# Command-line utility ■■■ vidly/ # Main project package ■ ■■■ __init__.py ■ ■■■
settings.py # Project settings ■ ■■■ urls.py # URL routing ■ ■■■ wsgi.py # Web server
gateway
```

Running Development Server

```
cd vidly python manage.py runserver # Visit: http://127.0.0.1:8000
```

Creating an App

```
python manage.py startapp movies # App structure movies/ ■■■ __init__.py ■■■ admin.py #
Admin configuration ■■■ apps.py # App configuration ■■■ models.py # Database models ■■■
views.py # View functions ■■■ tests.py # Tests ■■■ migrations/ # Database migrations
```

Registering App

```
# settings.py INSTALLED_APPS = [ 'django.contrib.admin', 'django.contrib.auth',
'movies.apps.MoviesConfig', # Add this ]
```

• --

24. Models & Databases

Defining Models

```
# movies/models.py from django.db import models from django.utils import timezone class
Genre(models.Model): name = models.CharField(max_length=255) def __str__(self): return
self.name class Movie(models.Model): title = models.CharField(max_length=255) release_year
= models.IntegerField() number_in_stock = models.IntegerField() daily_rate =
models.FloatField() genre = models.ForeignKey(Genre, on_delete=models.CASCADE)
date_created = models.DateTimeField(default=timezone.now) def __str__(self): return
self.title
```

Field Types

| Field Type | Description | Example |

|---|---|---|

| CharField | Short text | max_length=100 |

| TextField | Long text | Description |

| IntegerField | Integer | Age, Count |

| FloatField | Decimal | Price, Rating |

| BooleanField | True/False | is_active |

| DateField | Date | Birth date |

| DateTimeField | Date + Time | Created at |

| ForeignKey | Relationship | Genre |

| EmailField | Email | user@example.com |

Creating Migrations

```
# Create migrations python manage.py makemigrations # Apply migrations python manage.py migrate # View SQL python manage.py sqlmigrate movies 0001
```

Database Queries

```
# Get all movies movies = Movie.objects.all() # Filter movies movies = Movie.objects.filter(release_year=2020) # Get single movie movie = Movie.objects.get(id=1) # Create movie movie = Movie( title="Inception", release_year=2010, number_in_stock=10, daily_rate=4.99, genre=genre_obj ) movie.save() # Update movie movie.title = "Inception 2" movie.save() # Delete movie movie.delete()
```

• --

25. Admin Panel

Creating Superuser

```
python manage.py createsuperuser # Username: admin # Password: ****
```

Registering Models

```
# movies/admin.py from django.contrib import admin from .models import Genre, Movie # Basic registration admin.site.register(Genre) # Customized registration @admin.register(Movie) class MovieAdmin(admin.ModelAdmin): list_display = ('title', 'release_year', 'number_in_stock', 'daily_rate') list_filter = ('genre', 'release_year') search_fields = ('title',) exclude = ('date_created',)
```

Admin Features

• ■ CRUD operations

• ■ Search and filtering

• ■ Sorting

• ■ Bulk actions

• ■ Inline editing

• --

26. Views & Templates

Creating Views

```
# movies/views.py from django.shortcuts import render, get_object_or_404 from .models import Movie def index(request): movies = Movie.objects.all() return render(request, 'movies/index.html', {'movies': movies}) def detail(request, movie_id): movie = get_object_or_404(Movie, pk=movie_id) return render(request, 'movies/detail.html', {'movie': movie})
```

Creating Templates

• *Base Template** (templates/base.html):

```
<!DOCTYPE html> <html> <head> <title>Vidly</title> <link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css"> </head> <body> <nav class="navbar navbar-dark bg-dark"> <a class="navbar-brand" href="/">Vidly</a> </nav> <main class="container mt-4"> {% block content %} {% endblock %} </main> </body> </html>
```

• *Index Template** (templates/movies/index.html):

```
{% extends 'base.html' %} {% block content %} <h2>Movies</h2> <table class="table table-bordered table-hover"> <thead> <tr> <th>Title</th> <th>Genre</th> <th>Stock</th> <th>Daily Rate</th> </tr> </thead> <tbody> {% for movie in movies %} <tr> <td> <a href="{% url 'movies:detail' movie.id %}"> {{ movie.title }} </a> </td> <td>{{ movie.genre }}</td> <td>{{ movie.number_in_stock }}</td> <td>${{ movie.daily_rate }}</td> </...
```

• --

27. URL Routing

Main URLs

```
# vidly/urls.py from django.contrib import admin from django.urls import path, include urlpatterns = [ path('admin/', admin.site.urls), path('movies/', include('movies.urls')), ]
```

App URLs

```
# movies/urls.py from django.urls import path from . import views app_name = 'movies' urlpatterns = [ path('', views.index, name='index'), path('<int:movie_id>/', views.detail, name='detail'), ]
```

URL Parameters

```
# URL patterns path('<int:id>/', views.detail) # Integer path('<str:slug>/', views.by_slug) # String path('<int:year>/<int:month>/', views.archive) # Multiple
```

• --

28. REST APIs

Installing TastyPie

```
pip install django-tastypie
```

Creating API Resource

```
# api/models.py from tastypie.resources import ModelResource from movies.models import Movie class MovieResource(ModelResource): class Meta: queryset = Movie.objects.all()
```

```
resource_name = 'movies' excludes = ['date_created']
```

Registering API

```
# vidly/urls.py from api.models import MovieResource movie_resource = MovieResource()
urlpatterns = [ path('admin/', admin.site.urls), path('api/',
include(movie_resource.urls)), ]
```

API Endpoints

```
GET /api/movies/ # List all GET /api/movies/1/ # Get one POST /api/movies/ # Create PUT
/api/movies/1/ # Update DELETE /api/movies/1/ # Delete
```

- --

29. Deployment

Heroku Deployment

```
# Install Heroku CLI brew install heroku # Install dependencies pip install gunicorn
whitenoise # Create Procfile echo "web: gunicorn vidly.wsgi" > Procfile # Configure static
files STATIC_ROOT = os.path.join(BASE_DIR, 'static') python manage.py collectstatic # Git
setup git init git add . git commit -m "Initial commit" # Deploy to Heroku heroku login
heroku create git push heroku master heroku run python manage.py migrate
```

- --

Part 3: Machine Learning

30. Machine Learning Basics

What is Machine Learning?

ML enables computers to learn patterns from data without explicit programming.

• *Use Cases:**

• Image recognition

• Speech recognition

• Recommendation systems

• Fraud detection

• Stock prediction

ML Workflow

```
1. Import Data (CSV, Database) 2. Clean Data (Remove nulls, duplicates) 3. Split Data (80% train, 20% test) 4. Create Model (Select algorithm) 5. Train Model 6. Make Predictions 7. Evaluate Accuracy 8. Fine-tune
```

Installing Libraries

```
# Install Anaconda (includes all libraries) # Or install individually: pip install numpy pandas matplotlib scikit-learn jupyter
```

• --

31. Pandas & Data Analysis

Loading Data

```
import pandas as pd # Read CSV df = pd.read_csv('music.csv') # View data df.head() # First 5 rows df.tail() # Last 5 rows df.shape # (rows, columns) df.describe() # Statistics
```

Data Selection

```
# Select column ages = df['age'] # Select multiple columns subset = df[['age', 'gender']] # Filter rows young = df[df['age'] < 25]
```

Data Cleaning

```
# Check nulls df.isnull().sum() # Remove nulls df = df.dropna() # Fill nulls df['age'].fillna(df['age'].mean(), inplace=True) # Remove duplicates df = df.drop_duplicates()
```

• --

32. Model Training

Preparing Data

```
import pandas as pd from sklearn.model_selection import train_test_split from sklearn.tree import DecisionTreeClassifier from sklearn.metrics import accuracy_score # Load data df = pd.read_csv('music.csv') # Input/Output sets X = df.drop(columns=['genre']) # Features y = df['genre'] # Target # Split data X_train, X_test, y_train, y_test = train_test_split( X, y, test_size=0.2 )
```

Training Model

```
# Create model model = DecisionTreeClassifier() # Train model model.fit(X_train, y_train) # Make predictions predictions = model.predict(X_test) # Evaluate accuracy = accuracy_score(y_test, predictions) print(f"Accuracy: {accuracy * 100}%")
```

Model Persistence

```
from sklearn.externals import joblib # Save model joblib.dump(model, 'music-model.joblib') # Load model model = joblib.load('music-model.joblib') # Predict predictions = model.predict([[21, 1]]) print(predictions) # ['HipHop']
```

• --

33. Decision Trees

How Decision Trees Work

```
[age <= 30.5] / \ [gender <= 0.5] [Classical] / \ [Dance] [HipHop]
```

Visualizing Decision Trees

```
from sklearn import tree # Export tree tree.export_graphviz( model, out_file='tree.dot', feature_names=['age', 'gender'], class_names=sorted(y.unique()), label='all', rounded=True, filled=True )
```

Key Concepts

• *Accuracy Factors:**

1. **More data** = Better accuracy

2. **Clean data** = Better patterns

3. **Feature selection** = Relevant features only

4. **Algorithm choice** = Right tool for job

• --

Summary & Best Practices

Python Fundamentals

■ Use meaningful variable names

■ Follow PEP 8 style guide

■ Write functions for reusability

■ Handle exceptions properly

■ Use list comprehensions

■ Document your code

Django Development

■ Use migrations for DB changes

■ Never delete migrations

■ Name URLs and use {% url %} tag

■ Use get_object_or_404()

■ Organize templates with base templates

■ Use environment variables for secrets

Machine Learning

■ Always split train/test data

■ Clean data thoroughly

■ Save trained models

■ Measure accuracy

■ Use more data for better results

■ Try different algorithms

• --

Quick Reference

Python Syntax Cheat Sheet

```
# Variables x = 10 # Functions def greet(name): return f"Hello {name}" # Lists numbers = [1, 2, 3] # Dictionaries person = {"name": "John", "age": 25} # Loops for i in range(5): print(i) # Conditions if x > 10: print("Large") else: print("Small") # Classes class Person: def __init__(self, name): self.name = name
```

Django Commands

```
# Create project django-admin startproject myproject # Create app python manage.py startapp myapp # Migrations python manage.py makemigrations python manage.py migrate # Admin python manage.py createsuperuser # Run server python manage.py runserver
```

ML Quick Start

```
import pandas as pd from sklearn.model_selection import train_test_split from sklearn.tree import DecisionTreeClassifier # Load & prepare df = pd.read_csv('data.csv') X = df.drop(columns=['target']) y = df['target'] # Train X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2) model = DecisionTreeClassifier() model.fit(X_train, y_train) # Predict predictions = model.predict(X_test)
```

• --

• *End of Complete Python Guide** | © 2025 | From Transcript