Introduction to

# Python

## Programming Language

**Introductory to Intermediate • Syntax, OOP, Functions, Modules, Libraries & Ecosystem**

`Python 3.12+`

**Compared with Java, Go & Ruby | ~60 min session**

# Agenda

| | | |
|---|---|---|
| 01 | What is Python? — History, Philosophy & Zen of Python | 5 min |
| 02 | Variables, Types & Dynamic Typing | 9 min |
| 03 | Functions — First-class, Decorators & Generators | 10 min |
| 04 | Object-Oriented Python — Classes, Inheritance & Dunder Methods | 10 min |
| 05 | Control Flow — Comprehensions, Conditionals & Loops | 8 min |
| 06 | Modules, Packages & the Standard Library | 7 min |
| 07 | Popular Libraries — NumPy, Pandas, Requests, FastAPI | 6 min |
| 08 | Modern Python — 3.10 to 3.12 features | 7 min |
| 09 | Q & A / Wrap-Up | 2 min |

01

# What is Python?

Created by Guido van Rossum (Netherlands, 1991) — named after Monty Python!

Interpreted, dynamically typed, multi-paradigm — procedural, OOP, functional

Emphasizes readability — 'code is read more than written' (PEP 8)

Largest ecosystem of libraries — AI/ML, data science, web, automation, scripting

**1991**
Created

**#1**
TIOBE 2024

**500K+**
Libraries on PyPI

## The Zen of Python (import this):

The #1 language for beginners AND #1 for AI/ML professionals (2024)

**Beautiful > Ugly**

**Explicit > Implicit**

**Simple > Complex**

**Readability counts**

**Errors loudly!**

# Your First Python Program

hello.py

```
# hello.py — minimal, no boilerplate!
print("Hello, Pythonistas!")

# Variables — no type declaration needed
name = "Guido"
year = 1991
version = 3.12
is_awesome = True

# f-strings (Python 3.6+) — best way to format
print(f"Welcome to Python {version}!")
print(f"{name} created Python in {year}")

# Multi-line string
message = """
Python is:
  - Simple
  - Powerful
  - Beautiful
"""
print(message)
```

### `print()`

Built-in function — no import needed. Handles any type automatically.

### `# comment`

Hash for comments. Python has no block comment syntax — use docstrings.

### `f"..."`

f-strings: fastest and most readable string formatting. Any expression inside {}.

### `"""..."""`

Triple-quoted strings for multi-line content or docstrings (documentation).

### `No semicolons`

Newlines end statements. Indentation (4 spaces) defines code blocks — mandatory!

# Your First Python Program

terminal / REPL

```
$ python hello.py
Hello, Pythonistas!
Welcome to Python 3.12!
Guido created Python in 1991

Python is:
  - Simple
  - Powerful
  - Beautiful

# Interactive REPL
$ python3
>>> 2 + 2
4
>>> "hello".upper()
'HELLO'
```

### `print()`

Built-in function — no import needed. Handles any type automatically.

### `# comment`

Hash for comments. Python has no block comment syntax — use docstrings.

### `f"..."`

f-strings: fastest and most readable string formatting. Any expression inside {}.

### `"""..."""`

Triple-quoted strings for multi-line content or docstrings (documentation).

### `No semicolons`

Newlines end statements. Indentation (4 spaces) defines code blocks — mandatory!

# 02

# Variables, Types & Dynamic Typing

types.py

```
# No type declarations — Python infers!
x = 42              # int
y = 3.14            # float
name = "Python"     # str
flag = True         # bool
nothing = None      # NoneType (like null)

# Type hints (Python 3.5+) — optional but great!
age: int = 25
score: float = 98.6
greeting: str = "Hello"

# Numbers
big = 1_000_000     # readable with underscores
binary = 0b1010     # 10
hex_val = 0xFF      # 255
complex_n = 3 + 4j  # complex number!
```

### Dynamic typing

Variable type is determined at runtime. The same variable can hold different types.

### None

Python's null. Only None and False are falsy — 0, '', [] are also falsy (unlike Ruby!).

### Type hints

def greet(name: str) -> str: — hints help IDEs and tools but NOT enforced at runtime.

### Immutable types

int, float, str, tuple are immutable. list, dict, set are mutable. Important for correctness!

### Everything is object

Even 42 is an object: (42).bit_length() → 6. Same as Ruby, unlike Java primitives.

# 02

# Variables, Types & Dynamic Typing

types.py

```
# Strings — immutable sequences
s = "Hello, Python!"
s.upper()           # "HELLO, PYTHON!"
s.split(", ")       # ["Hello", "Python!"]
s[0:5]              # "Hello" (slicing)
s[::-1]             # reversed!
len(s)              # 14

# Collections (mutable)
my_list  = [1, "two", 3.0, True]   # ordered, mutable
my_tuple = (1, 2, 3)               # ordered, IMMUTABLE
my_set   = {1, 2, 3, 2}            # unique: {1,2,3}
my_dict  = {"key": "value", "n":1} # key-value

# Type conversion
int("42")       # 42
str(3.14)       # "3.14"
list((1,2,3))   # [1, 2, 3]
bool(0)         # False
```

### Dynamic typing

Variable type is determined at runtime. The same variable can hold different types.

### None

Python's null. Only None and False are falsy — 0, '', [] are also falsy (unlike Ruby!).

### Type hints

def greet(name: str) -> str: — hints help IDEs and tools but NOT enforced at runtime.

### Immutable types

int, float, str, tuple are immutable. list, dict, set are mutable. Important for correctness!

### Everything is object

Even 42 is an object: (42).bit_length() → 6. Same as Ruby, unlike Java primitives.

# 03

# Functions — First-class, Decorators & Generators

functions.py

```
# Basic function
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"

print(greet("Alice"))           # Hello, Alice!
print(greet("Bob", "Hi"))       # Hi, Bob!

# *args and **kwargs
def log(*args, **kwargs):
    for a in args: print(a)
    for k,v in kwargs.items():
        print(f"{k}={v}")

log("msg1","msg2", level="INFO", user="Alice")
```

### Default args

def f(x, y=10): — defaults must come after required args. Evaluated once at definition!

### *args/**kwargs

*args = positional tuple, **kwargs = keyword dict. Enables flexible APIs.

### Decorators

@decorator syntax sugar for func = decorator(func). Used for logging, auth, caching, timing.

### Generators

yield pauses execution and returns a value. Resumes on next(). Memory-efficient for large sequences.

### First-class

Functions are objects — assign to variables, pass as args, return from functions, store in lists.

# 03

# Functions — First-class, Decorators & Generators

functions.py

```
# Lambda — anonymous function
square = lambda x: x**2
nums = [1,2,3,4,5]
squares = list(map(square, nums)) # [1,4,9,16,25]
evens = list(filter(lambda x: x%2==0, nums))

# Decorators — wrap functions!
def timer(func):
    import time
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        print(f"Took {time.time()-start:.3f}s")
        return result
    return wrapper

@timer
```

### Default args

def f(x, y=10): — defaults must come after required args. Evaluated once at definition!

### *args/**kwargs

*args = positional tuple, **kwargs = keyword dict. Enables flexible APIs.

### Decorators

@decorator syntax sugar for func = decorator(func). Used for logging, auth, caching, timing.

### Generators

yield pauses execution and returns a value. Resumes on next(). Memory-efficient for large sequences.

### First-class

Functions are objects — assign to variables, pass as args, return from functions, store in lists.

# 03

# Functions — First-class, Decorators & Generators

functions.py

```
def slow_function():
    import time; time.sleep(0.1)

slow_function()  # Took 0.100s

# Generators — lazy sequences (memory efficient!)
def fibonacci():
    a, b = 0, 1
    while True:
        yield a          # pause and return value
        a, b = b, a+b

fib = fibonacci()
print([next(fib) for _ in range(8)])  # [0,1,1,2,3,5,8,13]

# Generator expression (like list comp but lazy)
big_squares = (x**2 for x in range(1_000_000))
```

### Default args

def f(x, y=10): — defaults must come after required args. Evaluated once at definition!

### *args/**kwargs

*args = positional tuple, **kwargs = keyword dict. Enables flexible APIs.

### Decorators

@decorator syntax sugar for func = decorator(func). Used for logging, auth, caching, timing.

### Generators

yield pauses execution and returns a value. Resumes on next(). Memory-efficient for large sequences.

### First-class

Functions are objects — assign to variables, pass as args, return from functions, store in lists.

# 04
# Object-Oriented Python

oop.py

```
class Animal:
    species_count = 0          # class variable

    def __init__(self, name, age):   # constructor
        self.name = name             # instance var
        self.age  = age
        Animal.species_count += 1

    def speak(self):                 # instance method
        return f"{self.name} makes a sound."

    @classmethod
    def count(cls):                  # class method
        return cls.species_count

    @staticmethod
    def is_valid_age(age):           # no self/cls
        return 0 < age < 100
```

### Dunder methods

__init__, __str__, __eq__, __len__... — special methods that define Python behavior. 'Magic methods'.

### @classmethod

Receives the class (cls) not instance. Use for alternative constructors or class-level operations.

### @staticmethod

No self or cls. Just a function in the class namespace. Use for utility functions.

### Dataclasses

@dataclass auto-generates __init__, __repr__, __eq__. Like Java records or Go structs.

# 04
# Object-Oriented Python

oop.py

```
    def __str__(self):                # like toString()
        return f"Animal({self.name}, {self.age})"

    def __repr__(self):               # dev repr
        return f"Animal(name={self.name!r})"

    def __eq__(self, other):          # == operator
        return self.name == other.name

# Inheritance
class Dog(Animal):
    def __init__(self, name, age, breed):
        super().__init__(name, age)  # call parent
        self.breed = breed

    def speak(self):                  # override
        return f"{self.name} says: Woof!"

# Multiple inheritance (unlike Java!)
```

| Dunder methods | @classmethod |
| --- | --- |
| __init__, __str__, __eq__, __len__... — special methods that define Python behavior. 'Magic methods'. | Receives the class (cls) not instance. Use for alternative constructors or class-level operations. |

| @staticmethod | Dataclasses |
| --- | --- |
| No self or cls. Just a function in the class namespace. Use for utility functions. | @dataclass auto-generates __init__, __repr__, __eq__. Like Java records or Go structs. |

# 04
# Object-Oriented Python

oop.py

```
# Multiple inheritance (unlike Java!)
class ServiceDog(Dog, Trainable):
    pass

# Dataclass (Python 3.7+) — auto __init__, __repr__, __eq__
from dataclasses import dataclass

@dataclass
class Point:
    x: float
    y: float
    z: float = 0.0

p = Point(1.0, 2.0)
print(p)  # Point(x=1.0, y=2.0, z=0.0)
```

| Dunder methods | @classmethod |
| --- | --- |
| __init__, __str__, __eq__, __len__... — special methods that define Python behavior. 'Magic methods'. | Receives the class (cls) not instance. Use for alternative constructors or class-level operations. |

| @staticmethod | Dataclasses |
| --- | --- |
| No self or cls. Just a function in the class namespace. Use for utility functions. | @dataclass auto-generates __init__, __repr__, __eq__. Like Java records or Go structs. |

# 05
# Control Flow & Comprehensions

control.py

```
# if / elif / else
score = 85
if score >= 90:
    grade = "A"
elif score >= 80:
    grade = "B"
else:
    grade = "C"

# Ternary (conditional expression)
grade = "Pass" if score >= 60 else "Fail"
```

**Indentation**
4 spaces define blocks — mandatory, not optional! Consistent indentation is Python's identity.

**for...in**
Python's for always iterates over a sequence. No C-style for(i=0;i<n;i++).

**enumerate()**
Get index AND value together — no manual counter needed.

**match (3.10+)**
Structural pattern matching — destructures and matches complex data structures.

**Comprehensions**
One-line list/dict/set construction — faster than loops, more readable.

**walrus :=**
x := expr assigns AND returns value in one step: while chunk := f.read(8192): process(chunk)

# 05
# Control Flow & Comprehensions

control.py

```
# match statement (Python 3.10+) — like switch
match command:
    case "quit":
        quit()
    case "help" | "?":
        show_help()
    case ("go", direction):
        move(direction)
    case _:
        print("Unknown")

# for loops + range
for i in range(10): print(i)
for i, val in enumerate(["a","b","c"]):
    print(f"{i}: {val}")
for k, v in my_dict.items():
    print(f"{k} = {v}")
```

**Indentation**
4 spaces define blocks — mandatory, not optional! Consistent indentation is Python's identity.

**for...in**
Python's for always iterates over a sequence. No C-style for(i=0;i<n;i++).

**enumerate()**
Get index AND value together — no manual counter needed.

**match (3.10+)**
Structural pattern matching — destructures and matches complex data structures.

**Comprehensions**
One-line list/dict/set construction — faster than loops, more readable.

**walrus :=**
x := expr assigns AND returns value in one step: while chunk := f.read(8192): process(chunk)

# 05
# Control Flow & Comprehensions

control.py

```
# while
while condition:
    do_work()
    if done: break
    if skip: continue

# List comprehensions — Python's superpower!
squares   = [x**2 for x in range(10)]
evens     = [x for x in range(20) if x%2==0]
matrix    = [[i*j for j in range(3)] for i in range(3)]
words     = [w.upper() for w in text.split()]

# Dict & set comprehensions
sq_dict   = {x: x**2 for x in range(5)}
unique_sq = {x**2 for x in [-2,-1,0,1,2]}

# Generator expression (lazy!)
total = sum(x**2 for x in range(1000000))
```

### Indentation
4 spaces define blocks — mandatory, not optional! Consistent indentation is Python's identity.

### for...in
Python's for always iterates over a sequence. No C-style for(i=0;i<n;i++).

### enumerate()
Get index AND value together — no manual counter needed.

### match (3.10+)
Structural pattern matching — destructures and matches complex data structures.

### Comprehensions
One-line list/dict/set construction — faster than loops, more readable.

### walrus :=
x := expr assigns AND returns value in one step: while chunk := f.read(8192): process(chunk)

# 06

# Modules, Packages & Popular Libraries

modules.py

```
# Importing modules
import os
import sys
from pathlib import Path
from datetime import datetime, timedelta
from collections import Counter, defaultdict
from typing import List, Dict, Optional

# Your own module (mymodule.py)
from mypackage.utils import helper_fn
from mypackage import models

# Package management
# pip install requests pandas numpy fastapi
```

### NumPy

Fast numerical arrays and math operations. Foundation of scientific Python.

### Pandas

DataFrames for data analysis — like Excel in Python. Essential for data science.

### Matplotlib

Plotting and visualization — charts, graphs, heatmaps.

### FastAPI

Modern async web API framework with automatic OpenAPI docs.

### SQLAlchemy

ORM for databases — like Hibernate for Java or ActiveRecord for Ruby.

# 06

# Modules, Packages & Popular Libraries

modules.py

```
# requests — HTTP made simple
import requests
resp = requests.get("https://api.example.com/data")
data = resp.json()

# pathlib — modern file paths
p = Path("data") / "input.csv"
text = p.read_text()
p.write_text("new content")

# collections
words = ["apple","banana","apple","cherry"]
counts = Counter(words)  # Counter({'apple': 2,...})
```

### NumPy

Fast numerical arrays and math operations. Foundation of scientific Python.

### Pandas

DataFrames for data analysis — like Excel in Python. Essential for data science.

### Matplotlib

Plotting and visualization — charts, graphs, heatmaps.

### FastAPI

Modern async web API framework with automatic OpenAPI docs.

### SQLAlchemy

ORM for databases — like Hibernate for Java or ActiveRecord for Ruby.

# 06

# Modules, Packages & Popular Libraries

modules.py

```
# dataclasses + typing
from dataclasses import dataclass, field
from typing import ClassVar

@dataclass
class Config:
    host: str = "localhost"
    port: int = 8080
    tags: list = field(default_factory=list)

# Virtual environments
# python -m venv .venv
# source .venv/bin/activate   (Mac/Linux)
# .venv\Scripts\activate      (Windows)
# pip install -r requirements.txt
```

### NumPy

Fast numerical arrays and math operations. Foundation of scientific Python.

### Pandas

DataFrames for data analysis — like Excel in Python. Essential for data science.

### Matplotlib

Plotting and visualization — charts, graphs, heatmaps.

### FastAPI

Modern async web API framework with automatic OpenAPI docs.

### SQLAlchemy

ORM for databases — like Hibernate for Java or ActiveRecord for Ruby.

# Python Quick Reference

## Strings

```
s = "Hello World"
s.lower(); s.strip()
s.replace("o","0")
", ".join(["a","b"])
s.startswith("He")
f"value={x:.2f}"
```

## Lists

```
a = [3,1,4,1,5]
a.append(9); a.pop()
a.sort(); a.reverse()
a[1:3]; a[::2]
[x for x in a if x>2]
sum(a); min(a); max(a)
```

## Dicts

```
d = {"a":1,"b":2}
d.get("c", 0)    # safe
d.keys(); d.values()
d.items()        # pairs
{**d, "c":3}     # merge
d.pop("a", None)
```

## File I/O

```
# Read
with open("f.txt") as f:
    text = f.read()
    lines = f.readlines()
# Write
with open("out.txt","w") as f:
    f.write("hello\n")
```

## Exceptions

```
try:
    risky()
except ValueError as e:
    print(e)
except (TypeError,KeyError):
    handle()
finally:
    cleanup()
raise ValueError("bad!")
```

## Async

```
import asyncio

async def fetch(url):
    async with session.get(url) as r:
        return await r.json()

asyncio.run(main())
# await, async for, async with
```

# Python vs Java vs Go vs Ruby — Summary

| Feature | Python | Java | Go | Ruby |
|---|---|---|---|---|
| **Typing** | Dynamic, duck typing | Static, explicit | Static, inferred | Dynamic, duck typing |
| **OOP** | Multi-paradigm | Class-based (strict) | Structs+interfaces | Pure OOP (everything) |
| **Null** | None (falsy) | null (NPE risk) | nil (pointer only) | nil (object!) |
| **Error handling** | try/except | try/catch/throw | (val, err) return | begin/rescue |
| **Concurrency** | asyncio / GIL limit | Threads / VT (21+) | goroutines (built-in) | Threads / Ractor |
| **Compilation** | Interpreted (CPython) | → JVM bytecode | → native binary | Interpreted |
| **Speed** | Slow (C ext help) | Fast (JIT) | Near-native | Slow |
| **Package mgmt** | pip / Poetry | Maven / Gradle | go modules | gem / Bundler |
| **Primary use** | AI/ML, data, web | Enterprise, Android | Cloud, CLIs, APIs | Web (Rails), scripting |
| **Learning curve** | Very low | High | Medium | Low |

# What's Next?

- 🤖 NumPy, Pandas & Matplotlib — data science stack
- 🌐 FastAPI / Django / Flask — web development
- 🧪 pytest — modern testing framework
- 🔁 asyncio — async/await concurrency deep-dive
- 🧠 scikit-learn & TensorFlow / PyTorch — ML/AI
- 📦 Packaging your own library & publishing to PyPI
- ⚡ Cython & C extensions — speeding up Python

## Resources

- → docs.python.org/3 — Official docs
- → realpython.com — Best tutorials
- → exercism.io/tracks/python
- → kaggle.com — Data science practice