Python Programming Tutorial

Based on Liao Xuefeng's Tutorial

Teaching Slides

Introduction to Python

October 12, 2025

Table of Contents

What is Python?

A High-Level Programming Language

Python is a high-level, interpreted programming language. It is designed to be easy to read and simple to implement.

- **Concise Code**: For the same task, Python often requires significantly fewer lines of code compared to languages like C or Java.
  - C: ∼1000 lines
  - Java: ∼100 lines
  - Python: ∼20 lines
- **Trade-off**: The conciseness comes at the cost of execution speed. C programs are the fastest, while Python is relatively slower.

What Can You Do with Python?

Python is Versatile

You can use Python for a wide range of applications:

- Daily tasks (e.g., automatically backing up your files).
- Building websites (Many famous sites like YouTube use Python).
- Creating the backend for online games.
- And much more!

What Python is NOT Ideal For

Some tasks are better suited for other languages:

- **Operating Systems**: This requires C.
- **Mobile Apps**: Use Swift/Objective-C for iPhone and Java/Kotlin for Android.
- **3D Games**: C or C++ are preferred for performance.

Origin and Philosophy

- **Creator**: Guido van Rossum created Python during the Christmas holiday in 1989.
- **Philosophy**: Guido's design goals for Python were "elegance," "clarity," and "simplicity." Python code should be easy to read and understand.
- **"Batteries Included"**: Python comes with a vast standard library that covers networking, files, GUIs, databases, and more, so you don't have to write everything from scratch.
- **Ecosystem**: Beyond the standard library, Python has a massive ecosystem of third-party libraries for almost any task.

Advantages and Disadvantages

Advantages

- Easy to learn and read.
- High development efficiency.
- Excellent for web applications, scripting, data analysis, and more.

Disadvantages

- **Slower Execution Speed**: As an interpreted language, Python is slower than compiled languages like C. However, for I/O-bound applications (like web apps), the network is the bottleneck, not the language speed.
- **Code is Not Encrypted**: Since it's interpreted, you distribute the source code. This is less of a concern in the modern era of web services and open-source software.

Python 2 vs. Python 3

Installation on macOS and Linux

Important Distinction

Python has two major versions, 2.x and 3.x. These two versions are **incompatible**.

- This course is based on the latest **Python 3.x** version.
- Python 2.x is no longer maintained. All new projects should use Python 3.
- Go to the official Python website (`python.org`) and download the Windows installer.
- **Crucial Step**: During installation, make sure to check the box that says **"Add Python 3.x to PATH"**. This allows you to run Python from the command line easily.
- **macOS**: System may come with Python 2. You can install Python 3 from the official website or using Homebrew (`brew install python3`).
- **Linux**: Most Linux distributions come with Python 3 pre-installed.

Running Python

Starting the Interactive Environment

- Open your command line (PowerShell on Windows, Terminal on macOS/Linux).
- Type `python` (on Windows) or `python3` (on macOS/Linux) and press Enter.
- You will see a prompt that looks like >>>. This is the Python interactive environment, where you can type code and see immediate results.
- To exit, type `exit()` and press Enter.

```
PS C:\Users\user> python
Python 3.12.3 ...
Type "help", "copyright", "credits" or "license" for more info
>>> print("Hello from Python!")
Hello from Python!
>>> exit()
```

Troubleshooting the 'PATH'

If you get an error like "'python' is not recognized ...", it means the Python

Types of Python Interpreters

The "Python" you run is an interpreter that executes your `.py` files. While the language is standardized, there are multiple interpreters.

CPython The official, most widely used interpreter, written in C. When you download from `python.org`, this is what you get. All code in this tutorial runs on CPython.

IPython An enhanced, more user-friendly interactive interpreter built on top of CPython.

PyPy A fast, alternative interpreter that uses a Just-In-Time (JIT) compiler to speed up code execution.

Jython A Python interpreter that runs on the Java Virtual Machine (JVM), allowing Python code to integrate with Java.

IronPython A Python interpreter that runs on Microsoft's .NET framework.

Writing Your First Code

Interactive Mode (‘¿¿¿‘)

In the interactive mode, Python executes your code line by line as soon as you press Enter. It’s great for testing small snippets.

```
>>> 100 + 200
300
>>> print('hello, world')
hello, world
```

The downside is that the code isn’t saved.

Script Mode (Running ‘.py‘ files)

For larger programs, we write code in a text editor and save it as a file with a `.py` extension.

1. **Write Code**: Open a text editor (like VS Code, but **not** Word or Notepad) and type:

   ```
   print('hello, world')
   ```

2. **Save File**: Save the file as `hello.py` in a specific directory (e.g., `C:\work`).

3. **Run from Command Line**: Open the command line, navigate to that directory, and run the script.

   ```
   C:\Users\user> cd C:\work
   C:\work> python hello.py
   hello, world
   ```

Input and Output (I/O)

Programs need to interact with the user.

Output with ‘print()‘

The `print()` function displays information to the screen.

- You can print multiple items, separated by commas. A space will be inserted between them.

```
print('The value is', 100 + 200)
# Output: The value is 300
```

Input with ‘input()‘

The `input()` function reads a line of text from the user and returns it as a string.

- You can provide a prompt to display to the user.

```
name = input('Please enter your name: ')
print('Hello,', name)
```

Basic Syntax

- **Indentation**: Python uses indentation (4 spaces is the convention) to define code blocks. There are no curly braces {}. This forces clean, readable code.
- **Comments**: Comments start with a # and are ignored by the interpreter.
- **Case-Sensitivity**: Python is case-sensitive. `myVar` is different from `myvar`.

```
# This is a comment
a = 100
if a >= 0:
    print(a) # This block is indented
else:
    print(-a) # This block is also indented
```

Data Types

- **Integers (‘int‘)**: Can be of any size. Hexadecimal numbers are prefixed with `0x`. Underscores can be used for readability (`10_000_000`).
- **Floating-Point Numbers (‘float‘)**: Decimal numbers. Scientific notation uses `e` (e.g., `1.23e9`).
- **Strings (‘str‘)**: Text enclosed in single `'` or double `"` quotes.
  - Escape characters: `\n` (newline), `\t` (tab), `\\` (backslash).
  - Raw strings (`r'...'`) ignore escape characters.
  - Multiline strings use triple quotes `'''...'''`.
- **Booleans (‘bool‘)**: `True` or `False`. Used with operators `and`, `or`, `not`.
- **NoneType (‘None‘)**: A special value representing nothingness or null.

Variables

Understanding Assignment

Dynamic Typing

In Python, you don't declare a variable's type. A variable is just a name pointing to an object. You can reassign a variable to an object of a different type.

```
a = 123        # a is an integer
print(a)
a = 'ABC'      # Now a is a string
print(a)
```

Assignment = makes a variable name point to a memory location.

```
a = 'ABC'
b = a
a = 'XYZ'
print(b) # What does this print?
```

It prints 'ABC'! b = a made b point to the same object as a. When a was reassigned, b's pointer did not change.

Character Encoding

- Computers only understand numbers. To process text, we need to convert characters to numbers. This is **encoding**.
- **ASCII**: Early standard, used 1 byte, only covered English characters and symbols.
- **Unicode**: A universal standard that includes characters from all languages.
- **UTF-8**: A variable-length encoding for Unicode. It's backward-compatible with ASCII and is the dominant encoding on the web. English characters take 1 byte, while Chinese characters usually take 3 bytes.

Golden Rule

In modern programming, data in memory is typically Unicode, and data stored on disk or sent over the network is encoded as UTF-8.

Strings in Python 3

‘str‘ and ‘bytes‘

- Python 3’s `str` type represents Unicode strings.
- The `bytes` type represents sequences of raw bytes.
- To convert between them, you must specify an encoding (almost always `utf-8`).
  - `str.encode('utf-8')` → `bytes`
  - `bytes.decode('utf-8')` → `str`

```
# Get the Unicode code point of a character
>>> ord('A')
65
# Get the character from a code point
>>> chr(66)
'B'
# str to bytes
>>> 'ABC'.encode('ascii')
b'ABC'
# bytes to str
>>> b'ABC'.decode('ascii')
```

String Formatting

There are several ways to format strings in Python.

1. f-Strings (Modern and Preferred)

Prefix the string with `f` and put variables inside {}.

```
name = 'Michael'
score = 95
print(f'Hello, {name}, your score is {score}.')
# You can also include expressions and formatting:
pi = 3.14159
print(f'Pi is approximately {pi:.2f}.')
```

2. % Operator (Older Style)

Uses %s for strings, %d for integers, %f for floats.

```
print('Hi, %s, you have $%d.' % ('Michael', 1000000))
```

Lists (‘list‘)

A list is an **ordered** and **mutable** collection of items.

- **Create**: `my_list = ['Michael', 'Bob', 'Tracy']`
- **Access**: By index (0-based). `my_list[0]` is 'Michael'. Negative indices count from the end: `my_list[-1]` is 'Tracy'.
- **Modify**:
  - `my_list.append('Adam')` adds to the end.
  - `my_list.insert(1, 'Jack')` inserts at a specific index.
  - `my_list.pop()` removes from the end. `my_list.pop(1)` removes at an index.
  - `my_list[1] = 'Sarah'` replaces an element.
- Elements can be of different types, including other lists.

Tuples (‘tuple‘)

A tuple is an **ordered** and **immutable** collection. Once created, it cannot be changed.

- **Create**: `my_tuple = ('Michael', 'Bob', 'Tracy')`
- **Immutable**: You cannot `append`, `insert`, or change elements. This makes code safer. Use tuples when you have a collection of items that shouldn't change.
- **Single-Element Tuple Trap**: To create a tuple with only one element, you **must** include a trailing comma.

```
t_wrong = (1)    # This is just the integer 1
t_correct = (1,)  # This is a tuple
```

- **"Mutable" Tuples**: A tuple is immutable in that its elements (references) cannot be changed. However, if an element is a mutable object (like a list), the contents of that object can still be changed.

```
t = ('a', 'b', ['A', 'B'])
t[2][0] = 'X' # This is allowed!
print(t) # ('a', 'b', ['X', 'B'])
```

Conditional Statements

Use `if`, `elif` (else if), and `else` to control the flow of execution based on conditions.

```
age = int(input('Enter your age: ')) # input() returns str, mu
if age >= 18:
    print('adult')
elif age >= 6:
    print('teenager')
else:
    print('kid')
```

Truthiness

Any non-zero number, non-empty string, or non-empty collection is considered `True` in an `if` statement. Zero, `None`, and empty collections are `False`.

```
x = 'hello'
if x: # This is True because x is not an empty string
    print('x is not empty')
```

Pattern Matching (Python 3.10+)

A more powerful way to handle complex conditional logic, similar to `switch` statements in other languages.

```
# A more readable alternative to long if/elif/else chains
score = 'B'
match score:
    case 'A':
        print('score is A.')
    case 'B':
        print('score is B.')
    case 'C':
        print('score is C.')
    case _: # The underscore is a wildcard, matches anything
        print('Invalid score.')
```

Pattern matching can also be used for more complex structures, like matching parts of a list.

Loops

Loops are used to execute a block of code repeatedly.

'for' loop

Iterates over a sequence (like a list, tuple, or string).

```
names = ['Michael', 'Bob', 'Tracy']
for name in names:
    print(name)

# Using range() to generate a sequence of numbers
sum = 0
for x in range(1, 101): # numbers from 1 to 100
    sum = sum + x
print(sum)
```

'while' loop

Repeats as long as a condition is true.

```
sum = 0
```

Dictionaries (‘dict‘)

A dictionary stores key-value pairs. It is **unordered** (in older Python versions) and **mutable**. Lookups are extremely fast.

- **Create**: `scores = {'Michael': 95, 'Bob': 75, 'Tracy': 85}`
- **Access**: `scores['Michael']` returns 95. Using `get()` is safer as it won't raise an error if the key doesn't exist: `scores.get('Thomas', -1)`.
- **Modify**: `scores['Adam'] = 67` adds a new entry or updates an existing one.
- **Remove**: `scores.pop('Bob')` removes the key 'Bob' and returns its value.
- **Key Requirement**: Dictionary keys must be **immutable** types (e.g., strings, numbers, tuples). Lists cannot be keys.

Sets (‘set‘)

A set is an **unordered** collection of **unique**, **immutable** items. It's like a dictionary with only keys.

- **Create**: `s = {1, 2, 3}`. Duplicates are automatically removed: `{1, 1, 2, 3}` becomes `{1, 2, 3}`.
- **Add/Remove**: `s.add(4)`, `s.remove(4)`.
- **Operations**: Sets support mathematical operations like union (`|`) and intersection (`&`).

```
s1 = {1, 2, 3}
s2 = {2, 3, 4}
print(s1 & s2) # Intersection: {2, 3}
print(s1 | s2) # Union: {1, 2, 3, 4}
```

What are Functions?

Calling Functions

Functions are fundamental building blocks in programming that encapsulate a piece of reusable code.

Abstraction

Functions allow us to abstract away details. Instead of writing the same calculation logic repeatedly, we define it once in a function and call it by name. This makes code more organized, readable, and easier to maintain.

Python has many built-in functions you can use immediately.

```
# abs() for absolute value
abs(-100) # Returns 100

# max() for the largest item
max(1, 2, 99, -5) # Returns 99

# Type conversion functions
int('123')     # Returns 123 (the number)
```

Defining Your Own Functions

Use the `def` keyword to define a function.

```
def my_abs(x):
    if not isinstance(x, (int, float)):
        raise TypeError('bad operand type')
    if x >= 0:
        return x
    else:
        return -x
```

- `def`: Starts the function definition.
- `my_abs(x)`: The function name and its parameters in parentheses.
- `:`: The colon marks the beginning of the indented function body.
- `return`: The `return` statement exits the function and optionally returns a value. If there's no `return` statement, the function returns `None`.
- It's good practice to check parameter types using `isinstance()` to make your function more robust.

Returning Multiple Values

A function can return multiple values, which are automatically packed into a tuple.

```
import math

def move(x, y, step, angle=0):
    nx = x + step * math.cos(angle)
    ny = y - step * math.sin(angle)
    return nx, ny # Returns a tuple (nx, ny)

# You can receive the tuple
r = move(100, 100, 60, math.pi / 6)
print(r) # (151.96..., 70.0)

# Or unpack it into multiple variables directly
x, y = move(100, 100, 60, math.pi / 6)
print(x, y) # 151.96... 70.0
```

Types of Arguments

Python's functions have very flexible argument handling.

Positional Arguments The standard arguments that are matched by position. `def power(x, n):`

Default Arguments Arguments with a default value, making them optional. Must come after positional arguments.
`def power(x, n=2):`

Variable Arguments (`*args`) Collects any number of extra positional arguments into a tuple. `def calc(*numbers):`

Keyword Arguments (`**kwargs`) Collects any number of extra keyword arguments into a dictionary.
`def person(name, age, **kw):`

Argument Order

The required order is: positional, default, `*args`, named-keyword, `**kwargs`.

Argument Details and Pitfalls

Named-Keyword Arguments

Warning

Never use a mutable object (like a list or dict) as a default argument value. The default is evaluated only once, when the function is defined, leading to unexpected behavior.

```
# WRONG
def add_end(L=[]):
    L.append('END')
    return L

>>> add_end()
['END']
>>> add_end()
['END', 'END'] # Problem!
```

```
# CORRECT
def add_end(L=None):
    if L is None:
        L = []
    L.append('END')
    return L

>>> add_end()
['END']
>>> add_end()
['END'] # Correct!
```

Recursive Functions

A recursive function is a function that calls itself.

Example: Factorial

The factorial of n ($n!$) can be defined as $n \times (n-1)!$.

```
def fact(n):
    if n == 1:
        return 1
    return n * fact(n - 1)

>>> fact(5)
120
```

The execution of `fact(5)` looks like this:

- `fact(5)` $\rightarrow$ `5 * fact(4)`
- `5 * (4 * fact(3))`
- `5 * (4 * (3 * fact(2)))`
- `5 * (4 * (3 * (2 * fact(1))))`
- `5 * (4 * (3 * (2 * 1)))` $\rightarrow$ `120`

Questions?

This concludes the introductory section covering the first 6 chapters.