What is Python?

Python was created by Guido van Rossum and first released in 1991. It is an interpreted language, meaning code is executed line by line, and it supports multiple programming paradigms including procedural, object-oriented, and functional programming. Python's design philosophy emphasizes code readability and simplicity, allowing programmers to express ideas in fewer lines compared to other languages.

Getting Started

- **Installation**: Download Python from the official website (python.org) and install it on your system.
- **Writing Programs**: Python files use the .py extension. You can write and run Python code in editors like VS Code or IDLE, or directly interact in the command-line Python interpreter.

Your First Python Program

Here's a simple program to print a message:

python

print("Hello, World!")

Core Features of Python

- **Simple and Easy to Learn**: Python's syntax is clear and concise, resembling natural language, which helps new programmers quickly grasp programming concepts.
- **Free and Open Source**: Python can be freely downloaded, used, and modified for any project or purpose.
- **Cross-Platform Compatibility**: Programs written in Python run on Windows, macOS, Linux, and other operating systems without modification.
- **Large Standard Library**: Python comes with a wealth of built-in modules for tasks ranging from file I/O and mathematics to network programming and web development.
- **Dynamically Typed**: Variable types are set at runtime, eliminating the need for explicit type declarations.

- **Interpreted Language**: Python code is executed line by line, which simplifies debugging and experimentation.
- **Object-Oriented**: Supports classes and objects, as well as principles such as inheritance, encapsulation, and polymorphism, aiding the creation of reusable and maintainable code.
- **Multiple Paradigms**: Supports procedural, object-oriented, and functional programming styles for flexible software development.
- **GUI Support**: Provides libraries such as Tkinter and PyQt for building desktop applications with graphical user interfaces.
- **High-Level Language**: Abstracts low-level details, so developers can focus on problem-solving rather than hardware.
- **Extensive Community**: Python enjoys strong support from a global community, leading to constant improvements and a vast pool of resources and third-party libraries.

Key Advantages of Python

- **Easy to Learn and Use**: Python features clear and concise syntax, making code both intuitive and readable. This simplicity lowers the barrier for new programmers.
- **Vast Libraries and Frameworks**: Python's rich standard library and an ecosystem of specialized packages (NumPy, Pandas, TensorFlow, Django, etc.) help solve a huge variety of tasks efficiently.
- **Cross-Platform Compatibility**: Python programs can run unmodified on various operating systems (Windows, Linux, macOS), enhancing portability and reducing deployment hassles.
- **Rapid Development**: With Python's concise syntax and powerful modules, developers can prototype and build applications faster, resulting in increased productivity.
- **Strong Community Support**: Python's large, active developer community provides ample tutorials, guides, and troubleshooting forums, making it easier to find help and learn new skills.
- **Multiple Paradigm Support**: Python supports different programming styles—procedural, object-oriented, and functional—allowing flexibility in solving problems.

- **Interpreted and Dynamically Typed**: Python code runs line by line without compiling, making error detection easier, while dynamic typing adds flexibility.
- **Free and Open Source**: Python can be downloaded and freely used, shared, or modified—even for commercial purposes.
- **Widespread Industry Adoption**: Python is trusted by major companies (Google, Facebook, Netflix), and its powerful security features make it suitable for diverse, mission-critical applications.
- **Integration Capabilities**: Python can easily integrate with code and systems written in other languages (like C, C++, and Java), expanding its range of use cases.

Variables:

Variables in Python are symbolic names that refer to values or objects stored in memory, allowing for the storage, manipulation, and retrieval of data during program execution.

Creating and Using Variables

- A variable is created the moment a value is assigned to it, using the assignment operator =, for example:

python

```
x = 10        # integer
name = "Sam"    # string
pi = 3.14       # float
```

- Variables don't require explicit type declaration; Python automatically infers the type based on the assigned value.
- Variables can be reassigned to hold values of different types at runtime:

python

```
x = 5
x = "Hello"     # Now x is a string
```

Variable Naming Rules

- Variable names can include letters, digits, and underscores, but must not start with a digit.

- They are case-sensitive (Name and name are different).
- Should not use Python keywords (like if, for, class).

Assigning Multiple Variables

- Assign the same value to multiple variables:

python

```
a = b = c = 0
```

- Assign different values to multiple variables in one line:

python

```
x, y, z = 1, 2.5, "Python"
```

Example in Practice

Variables are often used to store, update, or reference data:

python

```
radius = 7
circumference = 2 * 3.1416 * radius
print(circumference)
```

This example uses variables to store the circle's radius and compute its circumference.

Variables are foundational in Python, making code readable, reusable, and flexible for all kinds of programming tasks.

Python Data Types: Detailed Explanation

Understanding Python data types is fundamental for writing efficient and error-free programs. Python provides a rich set of built-in data types to classify and store different kinds of data values.

---

1. Built-in Data Types in Python

Python data types can be broadly categorized as:

- **Text Type:** str (strings)

- **Numeric Types:** int, float, complex
- **Sequence Types:** list, tuple, range
- **Mapping Type:** dict
- **Set Types:** set, frozenset
- **Boolean Type:** bool
- **Binary Types:** bytes, bytearray, memoryview
- **None Type:** NoneType

2. Details and Examples

Text Type (String)

- Represents text data, enclosed in single ('...') or double ("...") quotes.

python

```
name = "Alice"
print(type(name))  # <class 'str'>
```

Numeric Types

- **int:** Integer numbers, e.g., 25, -10
- **float:** Floating point numbers (decimals), e.g., 3.14, -0.001
- **complex:** Complex numbers with real and imaginary parts, e.g., 3 + 4j

python

```
x = 7        # int
pi = 3.1416    # float
z = 2 + 3j     # complex
print(type(x), type(pi), type(z))
```

Sequence Types

- **list:** Ordered, mutable collection of items, e.g., [1][2][3]
- **tuple:** Ordered, immutable collection, e.g., (1, 2, 3)
- **range:** Represents a sequence of numbers, commonly used in loops

python

```
fruits = ["apple", "banana", "cherry"]  # list
coords = (10, 20)                       # tuple
nums = range(5)                         # range object
```

Mapping Type

- **dict:** Key-value pairs collection, e.g., {'name': 'John', 'age': 30}

python

```
person = {"name": "John", "age": 30}
print(type(person))
```

Set Types

- **set:** Unordered collection of unique elements, mutable
- **frozenset:** Immutable version of set

python

```
colors = {"red", "green", "blue"}      # set
immutable_colors = frozenset(["red", "green"])
```

Boolean Type

- Represents truth values, True or False

python

```
is_ready = True
print(type(is_ready))
```

Binary Types

- **bytes:** Immutable sequence of bytes
- **bytearray:** Mutable counterpart of bytes
- **memoryview:** Memory view object referencing buffer

python

```
b = b"Hello"
```

ba = bytearray(5)

mv = memoryview(b"abcde")

None Type

- Special singleton representing the absence of a value

python

result = None

print(type(result)) *# <class 'NoneType'>*

3. Getting and Setting Data Types

- Use type() to get the data type of an object.

python

x = 10

print(type(x)) *# <class 'int'>*

- Python determines data type dynamically based on the assigned value.
- You can explicitly convert types using constructors like int(), str(), float(), etc.

python

num_str = "100"

num_int = int(num_str)

print(num_int, type(num_int))

Python Control Structures: Detailed Explanation with Examples

Control structures guide the flow of execution in Python programs. They determine *which* instructions run, *when*, and *how many times*. Python has three main types:

---

1. Sequential Control

This is the **default mode** where statements execute one after another, in the order they appear.

python

```
print("Step 1")
print("Step 2")
print("Step 3")
```

Output:

text

```
Step 1
Step 2
Step 3
```

This is straightforward: each line runs after the previous one completes.

---

2. Selection Control (Decision Making)

Selection controls execute code based on conditions using if, elif, and else statements.

Basic if statement:

python

```
age = 20
if age >= 18:
    print("You are an adult.")
```

Prints the message only if the condition is True.

if-else for two choices:

python

```
age = 16
if age >= 18:
    print("You can vote.")
```

else:

  print("You cannot vote yet.")

if-elif-else for multiple choices:

python

score = 85

if score >= 90:

  print("Grade: A")

elif score >= 80:

  print("Grade: B")

else:

  print("Grade: C")

---

3. Iterative Control (Loops)

Loops let you run a block of code multiple times until a condition changes.

while loop

Repeats as long as a condition is True.

python

count = 0

while count < 5:

  print(count)

  count += 1

Output:

text

0

1

2

3

4

for loop

Iterates over a sequence or range.

python

```
for i in range(1, 6):
  print(i)
```

Output:

text

1

2

3

4

5

---

4. Loop Control Statements

Control how loops execute:

- **break**: Exit the loop immediately.

python

```
for i in range(10):
  if i == 5:
    break
  print(i)
```

Prints numbers 0 through 4.

- **continue**: Skip the current iteration and move to next.

python

```
for i in range(5):
    if i == 2:
        continue
    print(i)
```

Skips printing 2.

- **pass**: Does nothing; placeholder.

python

```
for i in range(3):
    if i == 1:
        pass  # TODO: add code later
    print(i)
```

Summary

- **Sequential:** Runs code line by line.
- **Selection:** Branches code using conditions (if, elif, else).
- **Iteration:** Repeats code using loops (for, while).
- **Control Statements:** Modify loop flow (break, continue, pass).

Python Functions: Detailed Explanation with Examples

Functions in Python are reusable blocks of code designed to perform a specific task whenever they are called. They help make code modular, easier to read, maintain, and reuse.

---

1. Defining a Function

You define a function using the def keyword, followed by the function name, parentheses (which may include parameters), a colon, and an indented block called the function body.

python

```
def function_name(parameters):
    # function body
```

statements

- **def**: keyword to define a function.
- **function_name**: a valid identifier for calling the function later.
- **parameters**: optional inputs to the function, comma-separated.
- **Function body**: indented code that runs when the function is called.

Example:

python

```
def greet():
    print("Hello, World!")
```

2. Calling a Function

To execute the function, write its name followed by parentheses:

python

```
greet()  # Outputs: Hello, World!
```

---

3. Function Parameters and Arguments

Functions can take inputs — *parameters* — allowing us to pass data when calling them.

Example with parameters:

python

```
def add_numbers(a, b):
    return a + b


result = add_numbers(3, 5)
print(result)  # Outputs: 8
```

- a and b are parameters.
- Values 3 and 5 are arguments sent at the call.

---

4. Types of Function Arguments

- **Positional arguments:** Arguments passed in the exact order of parameters.
- **Keyword arguments:** Arguments specified with parameter names, order doesn't matter.

python

```
print_info(age=30, name="John")
```

- **Default arguments:** Parameters with default values if arguments are omitted.

python

```
def greet(name="Guest"):
  print(f"Hello, {name}!")


greet()     # Hello, Guest!
greet("Alice")  # Hello, Alice!
```

- **Variable-length arguments:** Functions can accept any number of positional (*args) or keyword (**kwargs) arguments.

python

```
def sum_all(*args):
  return sum(args)


print(sum_all(1, 2, 3))  # Outputs: 6
```

---

5. Returning Values

Functions use return to send back a result.

If no return is given, functions return None by default.

Example:

python

```
def square(x):
```

```
    return x * x

print(square(4))  # Outputs: 16
```

---

6. Anonymous (Lambda) Functions

Small one-expression functions created with the lambda keyword.

Example:

python

```
add = lambda x, y: x + y
print(add(3, 5))  # Outputs: 8
```

---

7. Recursive Functions

Functions that call themselves to solve problems by breaking them down.

Example: Factorial

python

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

print(factorial(5))  # Outputs: 120
```

Summary

- Define functions with def name(params): and an indented block.
- Parameters make functions flexible.
- Call functions by their name and provide arguments.

- Use return to output results.
- Use lambda for short anonymous functions.
- Recursion lets functions call themselves.

Python Exception Handling: Detailed Explanation

Exception handling in Python allows your program to manage unexpected errors gracefully without crashing abruptly. It lets you detect errors, handle them appropriately, and keep your program running or exit cleanly.

---

1. What is an Exception?

An exception is a runtime error that interrupts normal program flow, such as dividing by zero, invalid user input, or file handling issues. Python raises exceptions when errors occur.

2. Basic Syntax: try and except

You enclose code that might raise an exception inside a try block. If an exception occurs, control passes to the matching except block.

python

```
try:
    # Code that might raise an exception
    result = 10 / 0
except ZeroDivisionError:
    print("Can't divide by zero!")
```

Output:

text

```
Can't divide by zero!
```

The except block catches the specific exception (ZeroDivisionError) and handles it.

3. Handling Multiple Exceptions

You can handle different exceptions separately:

python

```
try:
    value = int(input("Enter a number: "))
    result = 10 / value
except ValueError:
    print("Error: Invalid number")
except ZeroDivisionError:
    print("Error: Cannot divide by zero")
```

4. else and finally

- The else block runs if no exceptions occur.
- The finally block runs always, regardless of exceptions — often for cleanup.

python

```
try:
    print("Trying...")
except:
    print("Exception caught")
else:
    print("No exceptions raised")
finally:
    print("Always runs")
```

5. Raising Exceptions

You can generate exceptions yourself using raise.

python

```
def check_age(age):
    if age < 18:
        raise ValueError("Age must be at least 18")
```

```
try:
    check_age(15)
except ValueError as e:
    print("Caught error:", e)
```

6. Exception Propagation

If an exception is not caught inside a function, it propagates up to the caller.

python

```
def f():
    x = int('abc')  # raises ValueError


try:
    f()
except ValueError as e:
    print("Error caught:", e)
```