## What is Python?

- Python is a **high-level, interpreted, general-purpose programming language** known for its **simplicity and readability**.
- It was created by Guido van Rossum and first released in 1991 and is now one of the most popular programming languages.
- Python is designed to be easy to read and write, with a short learning curve, making it ideal for both beginners and experts.
- It has a large and active community, which means many libraries and modules are available for use in Python programs.
- It is used for various purposes, including web development, data analysis, artificial intelligence, scripting, automation, and Robotics.

---

**NOTE: *Never memorize code!***

## Key Features of Python

✅ **Easy to Learn** – Simple syntax similar to English.
✅ **Dynamically Typed** – No need to declare variable types.
✅ **Interpreted Language** – Runs line-by-line (no compilation required).
✅ **Cross-Platform** – Works on Windows, macOS, and Linux.
✅ **Huge Libraries** – NumPy, Pandas, TensorFlow, Flask, Django, etc.
✅ **Object-Oriented & Functional** – Supports multiple programming paradigms.
✅ **Open-Source** – Free to use and modify.

---

## Where is Python Used?

- **Web Development** (Flask, Django)
- **Data Science & Machine Learning** (NumPy, Pandas, Scikit-Learn)
- **Automation & Scripting**
- **Game Development** (Pygame)
- **Cybersecurity & Ethical Hacking**
- **Embedded Systems & IoT**
- **Blockchain & Cryptocurrency**

---

## Hello World in Python

```
print("Hello, World!")
```

Python is **versatile, powerful, and beginner-friendly**—making it one of the most popular programming languages today!

```
In [2]: print("Hello, World!")
```

```
Hello, World!
```

# Syntax and Semantics in Python

## 1. Understanding Syntax in Python

Syntax in Python refers to the set of rules that determine the structure of the code. If you don't follow these rules, the interpreter will raise a `SyntaxError`, preventing the program from running.

---

## 1.1. Syntax Rules in Python

Here are some essential syntax rules in Python:

### a) Using Proper Indentation

Unlike many other languages that use `{}` (curly braces) to define blocks of code, Python relies on indentation. If you fail to indent properly, Python will raise an `IndentationError`.

✅ **Correct Syntax (Proper Indentation):**

```
if True:
    print("Hello, Python!")  # Indented correctly
```

❌ **Incorrect Syntax (No Indentation)**

```
if True:
print("Hello, Python!")  # IndentationError
```

---

### b) Using Parentheses in Function Calls

Functions in Python must be called with parentheses `()`.

✅ **Correct Syntax:**

```
print("Hello, World!")  # Function call with parentheses
```

❌ **Incorrect Syntax (No Parentheses):**

```
print "Hello, World!"  # SyntaxError in Python 3
```

---

### c) Proper Use of Colons ( `:` )

Colons are required after certain statements (like `if`, `for`, `while`, `def`, and `class`).

✅ **Correct Syntax:**

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

❌ **Incorrect Syntax (Missing Colon):**

```
def greet()  # SyntaxError: Expected ':'
    print("Hello!")
```

---

### d) Variable Naming Rules

- A variable name must start with a letter ( `a-z` , `A-Z` ) or an underscore `_` .
- It cannot start with a number ( `0-9` ).
- Only alphanumeric characters ( `A-Z` , `a-z` , `0-9` ) and underscores are allowed.

✅ **Valid Variable Names:**

```
my_variable = 10
_var123 = "Hello"
```

❌ **Invalid Variable Names:**

```
123var = 5  # SyntaxError: Variable cannot start with a number
my-var = 10  # SyntaxError: Hyphens are not allowed
```

---

### e) Statements and Line Continuation

Python normally treats each line as a separate statement. However, we can use `\` (backslash) to continue a statement onto the next line.

✅ **Using Backslash ( `\` ) for Line Continuation:**

```
result = 10 + 20 + \
         30 + 40
print(result)  # Output: 100
```

✅ **Using Parentheses for Implicit Line Continuation:**

```
result = (10 + 20 +
          30 + 40)
print(result)  # Output: 100
```

---

# 2. Understanding Semantics in Python

Semantics refers to the **meaning** of the code. A program may be syntactically correct but still have semantic errors, leading to unintended behavior.

---

## 2.1. Type Errors (Type Mismatch)

A common semantic error in Python is using incorrect data types in operations.

✅ **Correct Usage:**

```
x = 10
y = 5
print(x + y)  # Output: 15
```

❌ **Incorrect Usage (Type Error):**

```
x = "10"  # String
y = 5     # Integer
print(x + y)  # TypeError: can only concatenate str (not "int") to str
```

🔹 **Fix:** Convert `x` to an integer before addition:

```
print(int(x) + y)  # Output: 15
```

---

## 2.2. Variable Scope Issues

A variable declared inside a function is **local** to that function and cannot be accessed outside it.

✅ **Correct Variable Scope:**

```
def my_function():
    a = 10  # Local variable
    print(a)

my_function()  # Output: 10
```

❌ **Incorrect Variable Scope (NameError):**

```
def my_function():
    a = 10  # Local variable

print(a)  # NameError: name 'a' is not defined
```

🔹 **Fix:** Define `a` globally:

```
a = 10
def my_function():
    print(a)  # Now 'a' is accessible

my_function()  # Output: 10
```

---

## 2.3. Logical Errors (Incorrect Meaning)

Even if a program is syntactically correct, a logic error will produce incorrect results.

✅ **Correct Logic:**

```
def square(n):
    return n * n  # Correct logic

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

❌ **Incorrect Logic (Semantic Error):**

```
def square(n):
    return n + n  # Incorrect logic

print(square(4))  # Output: 8 (wrong result)
```

---

## 2.4. Using Uninitialized Variables

If you try to use a variable that hasn't been assigned a value, Python will raise a `NameError`.

❌ **Incorrect (Variable Not Defined):**

```
print(x)  # NameError: name 'x' is not defined
```

✅ **Correct (Define Variable Before Use):**

```
x = 10
print(x)  # Output: 10
```

---

## 2.5. Division by Zero

Python raises a `ZeroDivisionError` when trying to divide by zero.

❌ **Incorrect (Division by Zero):**

```
x = 10 / 0  # ZeroDivisionError: division by zero
```

✅ **Correct (Handle Division by Zero):**

```
x = 10
y = 0

if y != 0:
    print(x / y)
else:
    print("Cannot divide by zero")
```

---

# 3. Key Differences Between Syntax and Semantics

| Aspect | Syntax | Semantics |
|---|---|---|
| **Definition** | Structure and rules of writing code | Meaning of the code |
| **Error Type** | `SyntaxError` | `TypeError`, `NameError`, `ZeroDivisionError`, `Logical Errors` |
| **Checking Time** | Checked before execution (compilation time) | Checked at runtime |
| **Example of Error** | `print "Hello"` (missing parentheses) | `"10" + 5` (Type mismatch) |

---

# 4. Summary

1. **Syntax** is about the correct structure of the code.
2. **Semantics** is about the correct meaning of the code.

3. **Syntax errors prevent the program from running**, whereas **semantic errors allow execution but produce incorrect results**.
4. Understanding both is essential for writing bug-free Python programs.

# Types of Errors in Python

Errors in Python can be categorized into three main types:

1. **Syntax Errors**
2. **Runtime Errors (Exceptions)**
3. **Logical Errors**

Let's explore each type in detail with examples.

---

# 1. Syntax Errors

Syntax errors occur when Python encounters incorrect code structure. These errors prevent the program from running.

## Example of Syntax Error

```
print "Hello, World!"  # SyntaxError in Python 3 (missing parentheses)
```

**Error Message:**

```
SyntaxError: Missing parentheses in call to 'print'.
```

## Common Causes of Syntax Errors

- Missing or misplacing colons (`:`)
- Forgetting parentheses or brackets
- Incorrect indentation
- Using invalid variable names

✅ **Fix:**

```
print("Hello, World!")  # Correct syntax
```

---

# 2. Runtime Errors (Exceptions)

Runtime errors occur when the program is running. These errors do not stop the code from being compiled but cause it to crash at runtime.

## Types of Runtime Errors:

| Error Type | Description | Example |
|---|---|---|
| **NameError** | Using a variable that hasn't been defined | `print(x)` (if `x` isn't defined) |
| **TypeError** | Performing an operation on incompatible data types | `"10" + 5` |
| **ZeroDivisionError** | Division by zero | `10 / 0` |
| **IndexError** | Accessing an invalid index in a list or string | `my_list[5]` when `my_list` has only 3 items |
| **KeyError** | Accessing a non-existent key in a dictionary | `my_dict["missing_key"]` |
| **AttributeError** | Calling a method that doesn't exist for an object | `5.append(3)` |
| **ImportError** | Importing a module that doesn't exist | `import non_existing_module` |
| **ValueError** | Passing an argument of the wrong type to a function | `int("hello")` |
| **FileNotFoundError** | Trying to open a file that doesn't exist | `open("missing.txt")` |

### Example of a Runtime Error (TypeError)

```
x = "10"
y = 5
print(x + y)  # TypeError: can only concatenate str (not "int") to str
```

✅ **Fix:** Convert `x` to an integer:

```
print(int(x) + y)  # Output: 15
```

### Handling Runtime Errors Using `try-except`

```
try:
    print(10 / 0)  # Will cause ZeroDivisionError
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

**Output:**

```
Cannot divide by zero!
```

---

## 3. Logical Errors

Logical errors (bugs) occur when a program runs without crashing but produces incorrect or unintended results. These errors are the hardest to detect.

### Example of a Logical Error

```python
def square(n):
    return n + n  # Wrong logic

print(square(4))  # Expected 16, but gives 8
```

✅ **Fix:**

```python
def square(n):
    return n * n  # Correct logic

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

### Common Causes of Logical Errors

- Wrong formula or calculations
- Incorrect conditions in `if` statements
- Misplaced loops
- Forgetting to update variables inside loops

---

# Summary Table

| Type of Error | When It Occurs? | Example |
|---|---|---|
| **Syntax Error** | Before execution | `print "Hello"` (missing parentheses) |
| **Runtime Error** | During execution | `"10" + 5` (TypeError) |
| **Logical Error** | After execution (wrong output) | `return n + n` instead of `n * n` |