Python!

By- Mugdha Panhale,

Finitely Infinite Systems Pvt. Ltd.

What happens at background if we Run Python code :

Running a Python statement involves several steps happening in the background. Here's a brief overview:

1. **Parsing:**
The Python interpreter first reads your code and converts it into a format it can understand. This involves checking for syntax errors and creating an Abstract Syntax Tree (AST), which represents the structure of your code.

2. **Compiling:**
The AST is then compiled into bytecode, which is a low-level, platform-independent representation of your code. Bytecode is stored in `.pyc` files for faster execution in the future.

3. **Execution:**
The Python Virtual Machine (PVM) reads and executes the bytecode. The PVM is responsible for interpreting the bytecode and performing the necessary operations.

What happens at background if we Run Python code :

4. **Memory Management:**
During execution, Python handles memory management for you. It allocates memory for objects and variables and uses a garbage collector to free up memory that's no longer needed.

5. **Built-in Functions and Modules:**
If your code calls any built-in functions or modules, the interpreter loads them as needed. This includes standard library modules and any third-party modules you may have installed.

6. **Output Generation:**
Finally, the interpreter generates the output of your code, whether it's printing to the console, writing to a file, or any other action you specified.

What happens at background if we Run Python code :

When you write `print('Hello')`, the Python interpreter processes it through several stages to convert it into an Abstract Syntax Tree (AST). Here's a simplified breakdown:

1. **Lexical Analysis:**
The interpreter breaks the code into tokens, which are the smallest units of meaning, like `print`, `'Hello'`, and parentheses `(` `)`.

2. **Parsing:**
The tokens are then analyzed to understand their grammatical structure according to Python's syntax rules. This step involves creating a parse tree, which represents the code's hierarchical structure.

3. **AST Generation:**
The parse tree is transformed into an AST, which is a more abstract and compact representation of the code's structure. In the AST, each node represents a construct occurring in the code.

What happens at background if we Run Python code :

For `print('Hello')`, the AST might look something like this:

```plaintext
Module(body=[
  Expr(value=Call(func=Name(id='print', ctx=Load()), args=[
    Constant(value='Hello')
  ], keywords=[]))
])
```

- `Module`: The top-level structure that contains the entire code.
- `body`: A list of statements in the module.
- `Expr`: An expression statement (in this case, calling the `print` function).
- `Call`: A function call.
- `func`: The function being called (`print`).
- `args`: The arguments passed to the function (``'Hello'``).
- `Constant`: A constant value (``'Hello'``).

The AST helps the interpreter understand the structure and meaning of your code, enabling it to execute the `print` statement and display ``'Hello'`` on the console.

What happens at background if we Run Python code :

To see actual AST:
Create a file my_ast.py. Write code:

```
import ast
import astpretty

code = "print('Hello')"
tree = ast.parse(code)
astpretty.pprint(tree)
```

If astpretty is not available on your machine –> open terminal -> run command :

```
pip install astpretty
```

Run the Code:

Output will look like this:

What happens at background if we Run Python code :

```
Module(
   body=[
      Expr(
         lineno=1,
         col_offset=0,
         end_lineno=1,
         end_col_offset=14,
         value=Call(
            lineno=1,
            col_offset=0,
            end_lineno=1,
            end_col_offset=14,
            func=Name(lineno=1, col_offset=0, end_lineno=1, end_col_offset=5, id='print',
ctx=Load()),
            args=[Constant(lineno=1, col_offset=6, end_lineno=1, end_col_offset=13, value='Hello',
kind=None)],
            keywords=[],
         ),
      ),
   ],
   type_ignores=[],
)
```

What happens at background if we Run Python code :

ByteCode:

When you run the print('Hello') statement, Python compiles it into bytecode, which is a lower-level, platform-independent representation of your code. Bytecode is a set of instructions that the Python Virtual Machine (PVM) can execute.

To see the actual bytecode, you can use the dis module in Python. Here's how you can do it:

```
import dis

code = "print('Hello')"
compiled_code = compile(code, '<string>', 'exec')
dis.dis(compiled_code)
```

What happens at background if we Run Python code :

When you run this code, it will display the bytecode for the print('Hello') statement. The output might look something like this:

```
  1           0 LOAD_NAME                0 (print)
              2 LOAD_CONST               0 ('Hello')
              4 CALL_FUNCTION            1
              6 POP_TOP
              8 LOAD_CONST               1 (None)
             10 RETURN_VALUE
```

What happens at background if we Run Python code :

Here's a brief explanation of the bytecode instructions:

LOAD_NAME 0 (print): Loads the print function.

LOAD_CONST 0 ('Hello'): Loads the constant string 'Hello'.

CALL_FUNCTION 1: Calls the print function with one argument.

POP_TOP: Removes the result of the print function from the stack (since print returns None and we don't need to store that result).

LOAD_CONST 1 (None): Loads the constant None.

RETURN_VALUE: Returns the value (in this case, None).

This bytecode is what the PVM executes to carry out the print('Hello') statement.

Errors and Exceptions:

There are 2 Types of Errors:

Syntax Error:

Example:

If(5 >3)

print(“Test”) → SyntaxError : invalid syntax → bcoz we forgot : semicolon

This is because the interpreter recognizes this Syntax

Logical Error:

Example:

div_result = x/ y # x/y will throw an exception:

x =5 ,y =0 OR y = “one”

→ ZeroDivisionError: division by zero

→ TypeError: Unsupported operand type(s) for / : ‘int’ and ‘str’

→ Python creates exception object for these kind of scenarios

Errors and Exceptions

Python has a robust system of inbuilt exceptions that helps with error handling and debugging. Here are some of the common ones:

1. **IndexError: Raised when an index is out of range.**
2. **KeyError: Raised when a dictionary key is not found.**
3. **ValueError: Raised when a function receives an argument of the right type but inappropriate value.**
4. **TypeError: Raised when an operation or function is applied to an object of inappropriate type.**
5. **ZeroDivisionError: Raised when a division or modulo by zero is attempted.**
6. **FileNotFoundError: Raised when a file or directory is requested but doesn't exist.**
7. **IOError: Raised when an I/O operation fails.**
8. **ImportError: Raised when an import statement fails.**
9. **AttributeError: Raised when an invalid attribute reference is made.**
10. **RuntimeError: Raised when an error does not fall under any other category.**

These are just a few examples. Python has many more exceptions for different error scenarios.

Errors and Exceptions:

To see in- built exception list on console in a string format, we need to print output of following code:

```
list = dir( locals()['__builtins__'] )

for d in list:
  print(d + '\n')
```

output :

All built in exception names on console.

Errors and Exceptions:

Handling Exceptions:

Code:

```
def divider(x, y):
  return x / y

try:
  result = divider(5, '2')
except:
  print('Are you sure you are inputting two numbers?')
else:
  # The else block is going to get executed if there are no exceptions occurred.
  print("Everything runs smoothly..")
  print( f" Div  = {result} ")
finally:
   print(" I always run !...." )
```

Errors and Exceptions:

```
def divider(x, y):
  return x / y

try:
  result = divider(5, 0)
  # result = divider(5, 2)
except ZeroDivisionError:
  print(ZeroDivisionError.__doc__)
except:
  print('Are you sure you are inputting two numbers?')
else:
  print(f"Div  = {result}")
finally:
  print('I always run’)

# output:
Second argument to a division or modulo operation was zero.
I always run
```

Errors and Exceptions:

```
def divider(x, y):
  return x / y

def enter_a_number():
  while True:
    try:
      num = int(input("Enter a number: "))
    except:
      print("That is not a number. Try again! ")
      continue
    else:
      print(f"No: {num} is a Good choice !")
      break

  return num
```

Errors and Exceptions:

```
enter_a_number()

# output:
Enter a number: two
That is not a number. Try again!
Enter a number: //
That is not a number. Try again!
Enter a number: 3
No: 3 is a Good choice !
```

Errors and Exceptions:

Raising Errors and user- defined errors:

1. age = 'thirty'

```
if not type(age) is int:
  raise TypeError("Sorry! only intergers are allowed!")

#TypeError: Sorry! only intergers are allowed!
```

2.
```
def play_again():
  replay = input('Do you want to play again?: ')

  if replay.lower() not in ['yes', 'no']:
    raise ValueError("Enter yes or no only")

play_again()
```

Errors and Exceptions:

Errors and Exceptions stops further code running. So if we wish to avoid that we use try , except blocks.

```
def play_again():
  replay = input('Do you want to play again?: ')

  if replay.lower() not in ['yes', 'no']:
    raise ValueError("Enter yes or no only")

try:
  play_again()
except:
  print('enter only yes or no?')
  play_again()
```

Errors and Exceptions:

User –defined / Custom Exceptions:

Code:

```
class UserDefinedException(Exception):
  def __init__(self, message):
    self.message = message


marks = 35
print(f" You got {marks} marks..")

if(marks < 36):
  raise UserDefinedException("You didn't get enough markrks! ")
```

Run this code. This will raise a user defined exception and it will stop the program from further execution.

Errors and Exceptions:

In order to continue further program execution:
Code:

```
class UserDefinedException(Exception):
  def __init__(self, message):
    self.message = message

try:
  marks = int(input("Enter yoyr marks: "))
  print(f"You got {marks} marks..")

  if(marks < 36):
    raise UserDefinedException("You didn't get enough marks! ")
except UserDefinedException as err:
  print(err.message)
else:
  print("Congratulations! You passed!")
```

→ Run the code in terminal.

Unit testing:

In order to unit test we use assert functions.

Code:

```
# test_example.py
def add(a, b):
    return a + b

def test_add():
    assert add(1, 2) == 31, "the value should be 3"
    assert add(-1, -1) == -2
    assert add(0, 0) == 0

test_add()
```

Run the dose: o/p →

```
assert add(1, 2) == 31, "the value should be 3"
        ^^^^^^^^^^^^^^^^
AssertionError: the value should be 3
```

Unit testing:

Unit testing using unittest library.

1. Create a file my_script.py.
2. Code:

```
def cap_upper(text):
        return text.upper()

def adder(x, y):
        return x + y -1  # we know this is a mistake by developer adding -1
```

To test this :

We will write a unit test case:
So instead of using separate assert() function we will use a package unittest and run the entire test case once to see the result.

Run the code and see the result if there are any errors in our my_script.py file.

Unit testing:

```
import unittest
import my_script

class TestUpperAdd(unittest.TestCase):
  def test_one_word(self):
    text = 'python'
    result = my_script.cap_upper(text)
    self.assertEqual(result,'PYTHON')

  def test_multiple_words(self):
    text = 'monty python'
    result = my_script.cap_upper(text)
    self.assertEqual(result,'MONTY PYTHON')

  def test_adder(self):
    x = 5
    y = 10
    result = my_script.adder(x, y)
    self.assertEqual(result,15)

if __name__ == "__main__":
  unittest.main()
```