Name: Kaustubh Shrikant Kabra

Class: SE Comp-1

ERP Roll No. : 34

Teams Serial No. : 20

PPL CASE STUDY

TOPIC: PYTHON INTERPRETER

❖ Introduction

The word "interpreter" can be used in a variety of different ways when discussing Python. Sometimes interpreter refers to the Python REPL, the interactive prompt you get by typing `python` at the command line. Sometimes people use "the Python interpreter" more or less interchangeably with "Python" to talk about executing Python code from start to finish. In this chapter, "interpreter" has a more narrow meaning: it's the last step in the process of executing a Python program.

Before the interpreter takes over, Python performs three other steps: lexing, parsing, and compiling. Together, these steps transform the programmer's source code from lines of text into structured *code objects* containing instructions that the interpreter can understand. The interpreter's job is to take these code objects and follow the instructions.

❖ Python Interpreter –

Byterun is a Python interpreter written in Python. This may strike you as odd, but it's no more odd than writing a C compiler in C. (Indeed, the widely used C compiler gcc is written in C.) You could write a Python interpreter in almost any language.

Writing a Python interpreter in Python has both advantages and disadvantages. The biggest disadvantage is speed: executing code via Byterun is much slower than executing it in CPython, where the interpreter is written in C and carefully optimized. However, Byterun was designed originally as a learning exercise, so speed is not important to us. The biggest advantage to using Python is that we can more easily implement *just* the interpreter, and not the rest of the Python run-time, particularly the object system.

❖ Building an Interpreter

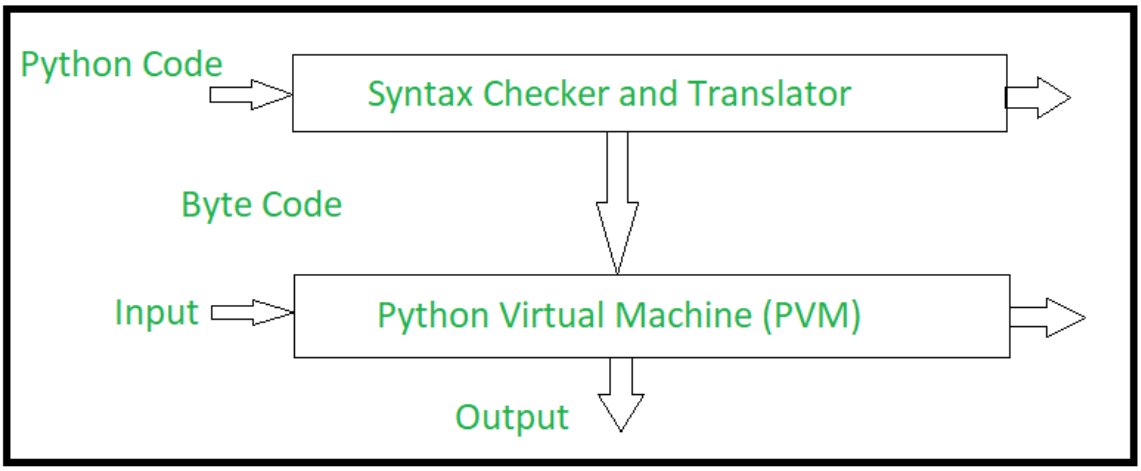
The Python interpreter is a *virtual machine*, meaning that it is software that emulates a physical computer. This particular virtual machine is a stack machine: it manipulates several stacks to perform its operations (as contrasted with a register machine, which writes to and reads from particular memory locations).

The Python interpreter is a *bytecode interpreter*: its input is instruction sets called *bytecode*. When you write Python, the lexer, parser, and compiler generate code objects for the interpreter to operate on. Each code object contains a set of instructions to be executed—that's the bytecode—plus other information that the interpreter will need. Bytecode is an *intermediate representation* of Python code: it expresses the source code that you wrote in a way the interpreter can understand. It's analogous to the way that assembly language serves as an intermediate representation between C code and a piece of hardware.

We need some higher-level context on the structure of the interpreter. How does the Python interpreter work?

❖ How Does Python Interpreter Works?

Python is an object oriented programming language like Java. Python is called an interpreted language. Python uses code modules that are interchangeable instead of a single long list of instructions that was standard for functional programming languages. The standard implementation of python is called “cpython”. It is the default and widely used implementation of the Python.
Python doesn’t convert its code into machine code, something that hardware can understand. It actually converts it into something called byte code. So within python, compilation happens, but it’s just not into a machine language. It is into byte code and this byte code can’t be understood by CPU. So we need actually an interpreter called the python virtual machine. The python virtual machine executes the byte codes.



The Python interpreter performs following tasks to execute a Python program :

- **Step 1 :** The interpreter reads a python code or instruction. Then it verifies that the instruction is well formatted, i.e. it checks the syntax of each line.If it encounters any error, it immediately halts the translation and shows an error message.
- **Step 2 :** If there is no error, i.e. if the python instruction or code is well formatted then the interpreter translates it into its equivalent form in intermediate language called "Byte code".Thus, after successful execution of Python script or code, it is completely translated into Byte code.
- **Step 3 :** Byte code is sent to the Python Virtual Machine(PVM).Here again the byte code is executed on PVM.If an error occurs during this execution then the execution is halted with an error message.

Well, internally, four things happen in a REPL:

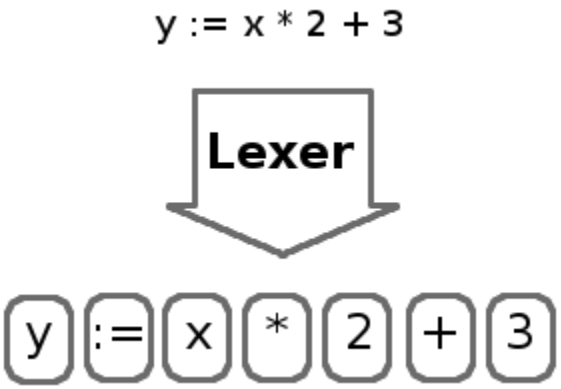
i. Lexing- The lexer breaks the line of code into tokens.

ii. Parsing- The parser uses these tokens to generate a structure, here, an Abstract Syntax Tree, to depict the relationship between these tokens.

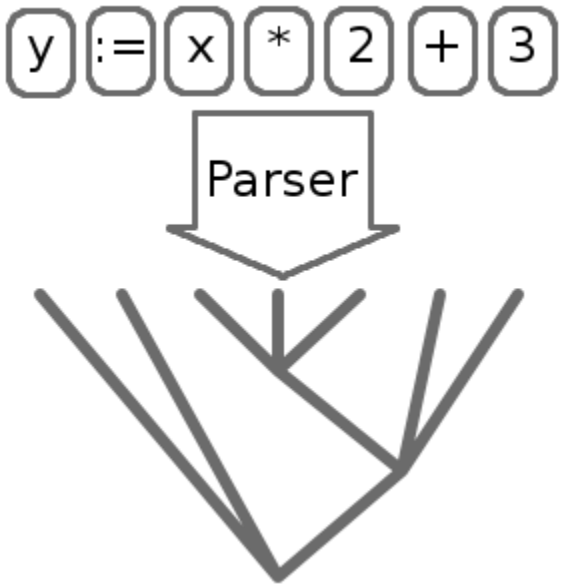
iii. Compiling- The compiler turns this AST into code object(s).

iv. Interpreting- The interpreter executes each code object

The process of splitting characters into tokens is called lexing and is performed by a lexer. Tokens are short, easily digestible strings that contain the most basic parts of the program such as numbers, identifiers, keywords, and operators. The lexer will drop whitespace and comments, since they are ignored by the interpreter.



The process of organizing tokens into an abstract syntax tree (AST) is called parsing. The parser extracts the structure of the program into a form we can evaluate.



Parsing

The parsing process is fairly standard. The basic idea is to first convert the input characters into a more abstract representation like name: x, integer: 7, string: "hello", less-than-or-equal, etc. The abstracted characters are called *tokens*. (There are utilities, such as lex, for automatically generating tokenizers, but Python does not use one.) The tokenization process is fully described in the language reference.

For example, the statement

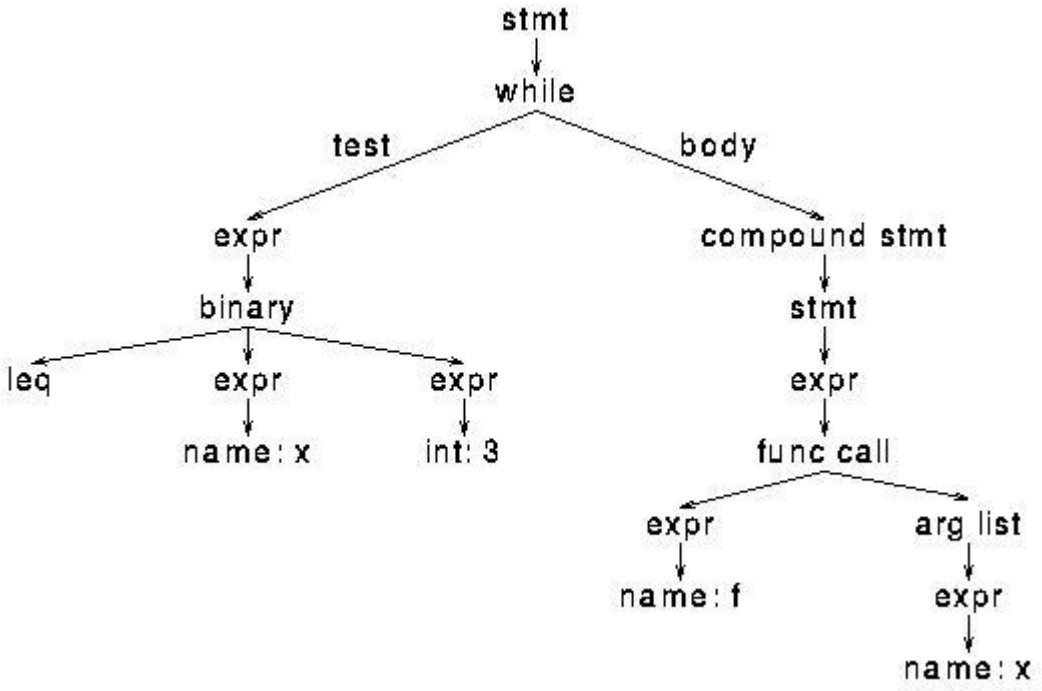
```
while(x <= 3):
    f(x)
```

might be tokenized as

```
keyword: while
left-paren
name: x
leq
int: 3
right-paren
colon
indent
name: f
left-paren
name: x
right-paren
```

Then the tokens are assembled into expressions, statements, function definitions, class definitions, etc. Since function definitions contain statements which contain expressions

which may contain nested expressions, and so on, the resulting data structure of tokens is a tree, called a *parse tree* or *abstract syntax tree*. The tokens above might be parsed into this tree:



Compilation

The parse tree is then compiled into bytecode by a recursive walk. For example, a `while` syntax node contains an expression node for the test and a compound statement node for the body. A `while` node is compiled into:

```
loop:
(code for test)
jump_if_false done
(code for body)
jump loop
done:
```

where the test and body nodes are compiled recursively. Virtually all of the compilation rules can be described as rewrites like this. Compilation has the opposite structure of parsing: it flattens the tree, converting it back to bytes.

Bytecode is virtual machine instructions are packed into an array of bytes. The instructions are based on stack manipulation. Some instructions have no arguments and take up one byte, e.g. `BINARY_ADD` (pop two values from the stack and push their sum), while other instructions have an additional two-byte integer argument, e.g. `LOAD_NAME i` (push the value of the *i*-th variable name). The full list of bytecodes is here. Bytecode is paired with a symbol table and constant pool so that names and literals can be referenced by number in the instructions.

The bytecode for the above parse tree is something like:

```
loop:

load-name 1       (x)
load-const 1      (3)
compare-op le

jump_if_false done

load-name 2       (f)
load-name 1       (x)
call-function 1

jump loop
done:
```

Python does not perform much optimization on the bytecode, except for speeding-up local variable access. The reasons for this are tied to how the interpreter works and will be discussed later.

This conversion pattern, where the input is abstracted, processed, then specialized, is a common one in many varieties of programs and is also presented in Abstraction and Specification in Program Development.

Execution

At face value, the execution algorithm is straightforward: fetch the next instruction, perform the required stack manipulation or, in the case of a jump, reposition the instruction pointer. However, much of the real functionality is hidden in the value objects. For example, there is only one `BINARY_ADD` instruction, yet Python must do very different things when adding integers, floats, strings, or user-defined objects.

❖ Function Objects & Code Objects in Python

When we talk of function objects, we mean to say that in Python, **functions** are first-**class objects** (functions *indeed* are objects).
You can pass them around and talk about them without making a call to them.

```
>>> def bar(a):
```

```
        x=3

        return x+a

>>> bar
```

Output

```
<function bar at 0x107ef7aa2>
```

Now bar.__code__ returns a code object:

```
>>> bar.__code__

<code object bar at 0x107eeccb2, file "<stdin>", line 1>
```

So, we conclude that a code object is an attribute of a function object. The dir() function will tell us more about the function:

```
>>> dir(bar.__code__)
```

Output

```
['__class__', '__cmp__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'co_argcount', 'co_cellvars', 'co_code', 'co_consts', 'co_filename',
'co_firstlineno', 'co_flags', 'co_freevars', 'co_lnotab', 'co_name', 'co_names', 'co_nlocals',
'co_stacksize', 'co_varnames']
```

This gives us the attributes of the code object. Values of some more attributes:

```
>>> bar.__code__.co_varnames
```

Output

```
('a', 'x')
```

```
>>> bar.__code__.co_consts
```

Output

```
(None, 3)
```

```
>>> bar.__code__.co_argcount
```

Output

```
1
```

❖ Bytecode in Python

The following command gives us the bytecode:

```
>>> bar.__code__.co_code
```

Output

```
'd\x01\x00}\x01\x00|\x01\x00|\x00\x00\x17S'
```

This is a series of bytes, each of which the interpreter loops through and then makes an execution.

❖ Disassembling the Bytecode

We will use the dis() method from the dis module to understand what's going on- this isn't part of what the interpreter does.

```
>>>import dis


>>> dis.dis(bar.__code__)
```

Output

```
2              0 LOAD_CONST 1 (3)
3 STORE_FAST 1 (x)
3              6 LOAD_FAST 1 (x)
9 LOAD_FAST 0 (a)
12BINARY_ADD
13RETURN_VALUE
```

In this, the first set of numbers is the line numbers in the actual code. The second one depicts offsets of the bytecode.

Then comes the set of names for the bytes- for human readability.

The next column depicts the arguments and the last column lists the constants and names in the fourth column.

```
>>> bar.__code__.co_consts[1]
```

Output

```
3
```

```
>>> bar.__code__.co_varnames[1]
```

❖ Conclusion

Hence, we can say the compiler for Python generates bytecode for the interpreter. The Python interpreter uses this with the virtual machine.

The same bytecode doesn't always end up doing the same things. This is another thing that makes Python dynamic.

Also, the default prompt for the interpreter is >>>.