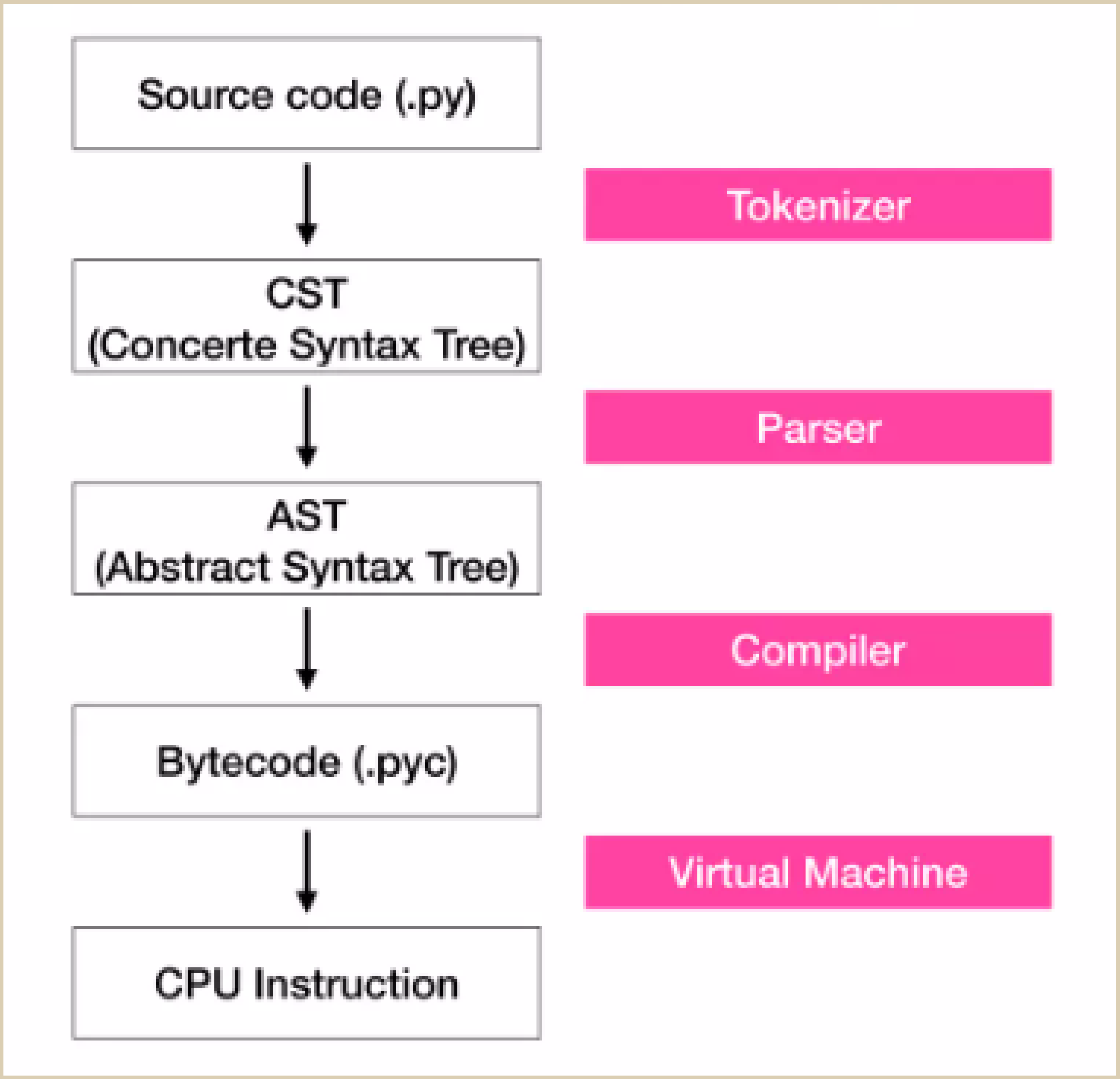
How CPython Transforms Code into Execution



Next Slide

Introduction

As shown in previous slide, CPython transforms code into execution through several steps: The Tokenizer breaks down source code into tokens, the Parser organizes it into a Concrete Syntax Tree (CST) and then an Abstract Syntax Tree (AST), and the Compiler converts this into bytecode (.pyc). Finally, the Virtual Machine interprets the bytecode to run on the CPU, ensuring smooth and efficient execution.

This tutorial is so much inspired from Ricky's slides.

Next Slide →

Tokenizer

Consider the following code snippet, which performs a simple sum of 1 and 2 inside `simple_test.py` file. Let's see how it can be tokenized.

```
$ cat  simple_test.py
one_plus_two = 1 + 2

$ python3  -m  tokenize  simple_test.py
0,0-0,0:          ENCODING          'utf-8'
1,0-1,12:         NAME              'one_plus_two'
1,13-1,14:        OP                '='
1,15-1,16:        NUMBER            '1'
1,17-1,18:        OP                '+'
1,19-1,20:        NUMBER            '2'
1,20-1,21:        NEWLINE           '\n'
2,0-2,0:          ENDMARKER         ''
```

Next Slide

Concrete Syntax Tree (CST)

Using the tokenizer from Python's libcst library, we obtain the CST (Concrete Syntax Tree) of our code snippet.

```
>>> import libcst as cst
>>> code = "1 + 2"
>>> source_tree = cst.parse_expression(code)
>>> print(source_tree)
```

We can see the result in the next slide.

Next Slide

```
BinaryOperation(
    left=Integer(
        value='1',
        lpar=[],
        rpar=[],
    ),
    operator=Add(
        whitespace_before=SimpleWhitespace(
            value=' ',
        ),
        whitespace_after=SimpleWhitespace(
            value=' ',
        ),
    ),
    right=Integer(
        value='2',
        lpar=[],
        rpar=[],
    ),
    lpar=[],
    rpar=[],
)
```

Next Slide →

Abstract Syntax Tree (AST) and Parser

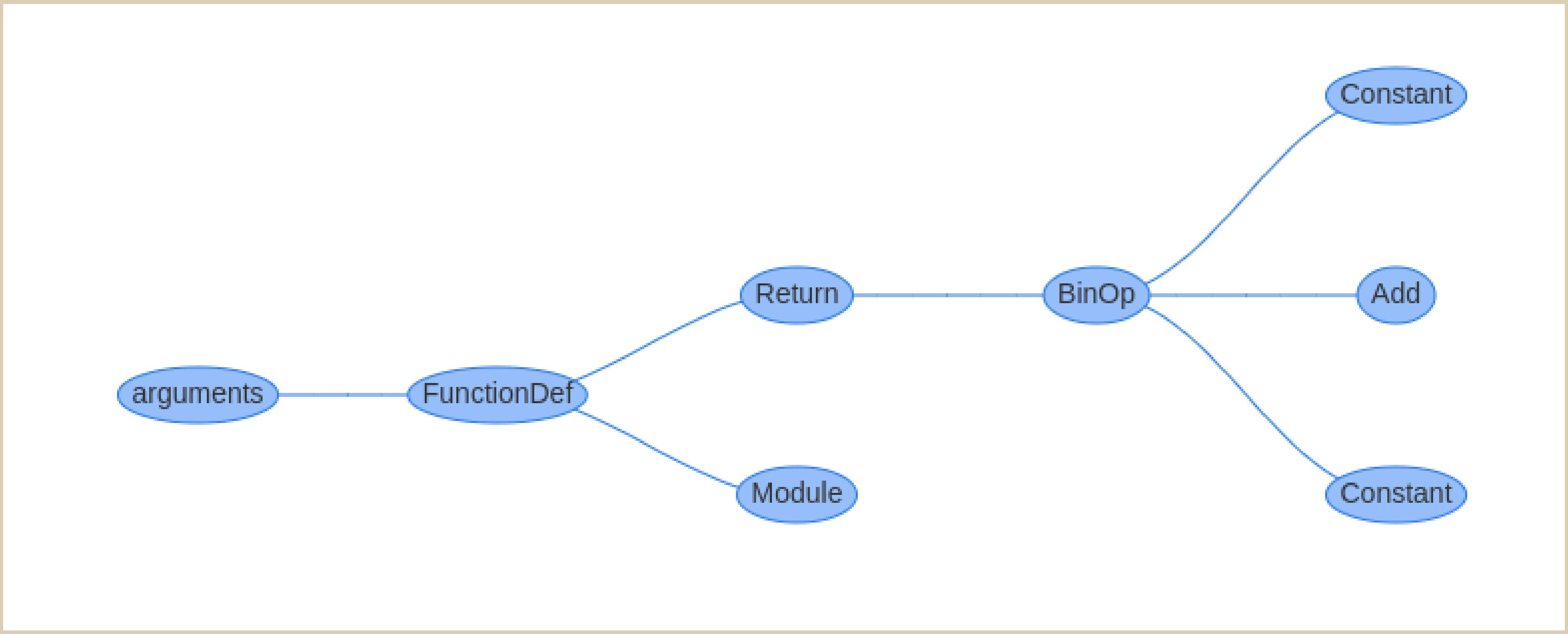
We can also convert our code snippet into an AST using the ast parser.

```
>>> import ast
>>> code = "one_plus_two = 1 + 2"
>>> tree = ast.parse(code)
>>> print(ast.dump(tree, indent=2))
Module(
  body=[
    Assign(
      targets=[
        Name(id='one_plus_two', ctx=Store())],
      value=BinOp(
        left=Constant(value=1),
        op=Add(),
        right=Constant(value=2)))],
  type_ignores=[])
```

Next Slide

Now Let's see an AST using instaviz Python library.

```
>>> import instaviz
>>> def one_plus_two():
...     return 1 + 2
...
>>> instaviz.show(one_plus_two)
```

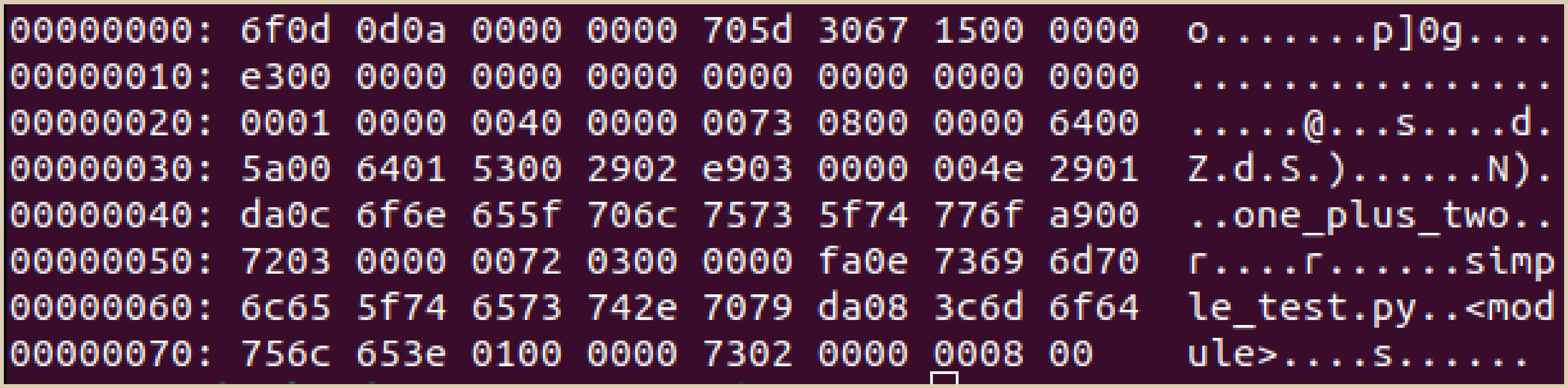


Next Slide

Bytecode

Now, we can compile our `simple_test.py` file into bytecode like this:

```
$ python3 -m py_compile simple_test.py
$ xxd __pycache__/simple_test.cpython-310.pyc
```



```
00000000: 6f0d 0d0a 0000 0000 705d 3067 1500 0000  o.......p]0g....
00000010: e300 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0001 0000 0040 0000 0073 0800 0000 6400  .....@...s....d.
00000030: 5a00 6401 5300 2902 e903 0000 004e 2901  Z.d.S.)......N).
00000040: da0c 6f6e 655f 706c 7573 5f74 776f a900  ..one_plus_two..
00000050: 7203 0000 0072 0300 0000 fa0e 7369 6d70  r....r......simp
00000060: 6c65 5f74 6573 742e 7079 da08 3c6d 6f64  le_test.py..<mod
00000070: 756c 653e 0100 0000 7302 0000 0008 00    ule>....s......
```

Next Slide

CPU Instructions

Using Python's dis module, we can view the CPU instructions for our code.

```
>>> import dis
>>> dis.dis("one_plus_two = 1 + 2")
  1           0 LOAD_CONST               0 (3)
              2 STORE_NAME               0 (one_plus_two)
              4 LOAD_CONST               1 (None)
              6 RETURN_VALUE
```