
A Walk with
CPython
Sadhana Srinivasan

Who am I?

- I am a data scientist
- I really like CPython and got into it because of an old project of mine
- You can find me on LinkedIn (/sadhana-srinivasan/)

What is CPython?

- .py files need to be translated to run on a system
- There are alternatives such as PyPy, IronPython, Stackless

What can you expect today?

A quick intro

The talk aims to give you an understanding with which you can dive into the CPython codebase yourself.

You will not be an expert but you will not be entirely lost in the codebase.

There are references at the end of the slides

What are the complier's steps?

What happens after python reads the program?

How does tokenisation, AST building etc. happen?

What does the interpreter get?

What does the interpreter do?

What does the interpreter do with the input?

What are some of the optimisations it has?


Compiler!? *Interpreter*,
right?

The Program

```
def square(num):
    return num*num

print(square(2))
```

The Steps

Tokeniser & Parser

Break the input string into recognisable parts.

Parse it into an AST

AST to CFG*

The AST is a representation of the program. This need to be flattened into OPCODES.

CFG to OPCODES

Flatten the CFG into OPCODES

The Tokeniser

The tokeniser converts the code into meaningful chunks.

It also find out if you've used any character that isn't accepted.

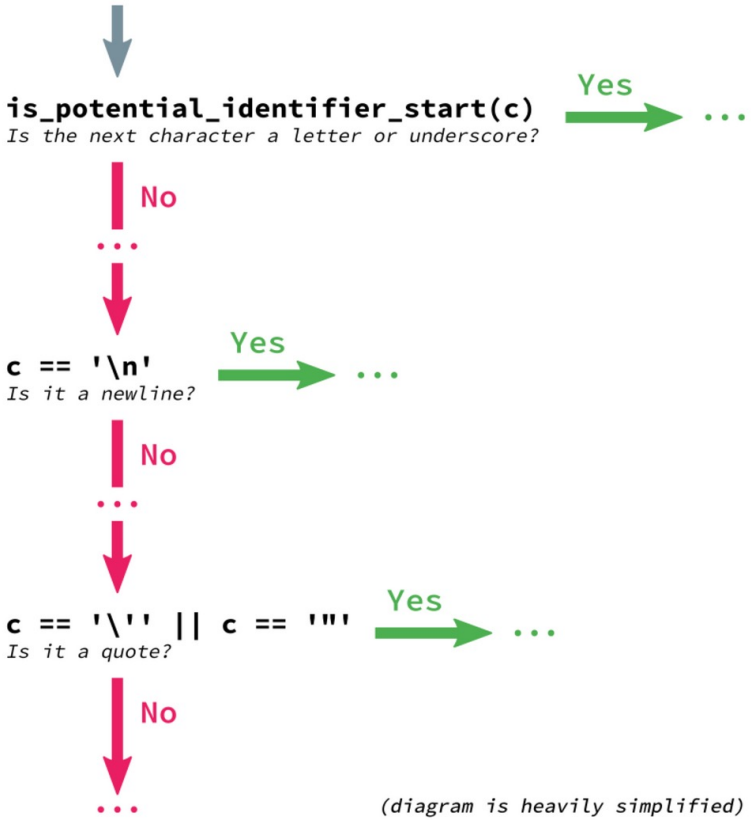


Image Source: https://benjam.info/blog/posts/2019-09-18-python-deep-dive-tokenizer/

The Tokeniser

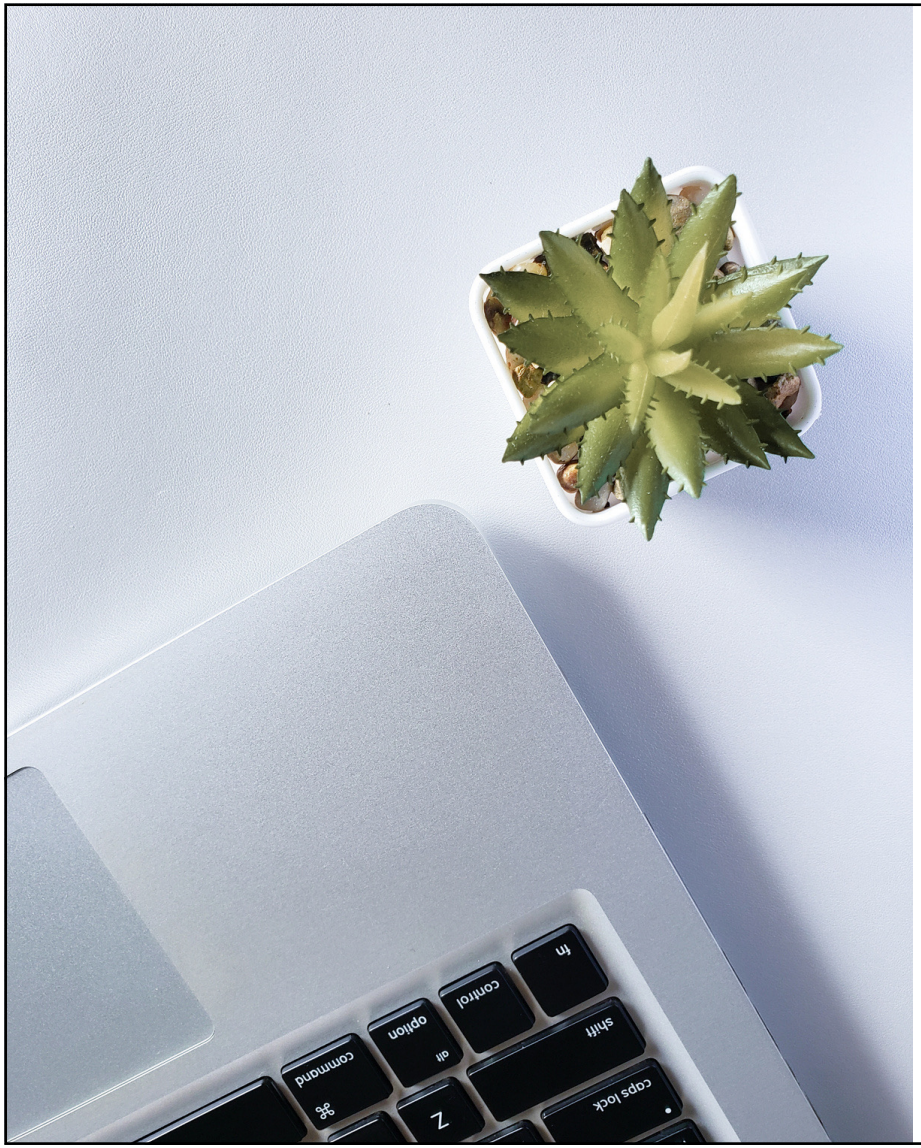
```
0,0-0,0:            ENCODING       'utf-8'
1,0-1,3:            NAME           'def'
1,4-1,10:           NAME           'square'
1,10-1,11:          OP             '('
1,11-1,14:          NAME           'num'
1,14-1,15:          OP             ')'
1,15-1,16:          OP             ':'
1,16-1,17:          NEWLINE        '\n'
2,0-2,4:            INDENT         '    '
2,4-2,10:           NAME           'return'
2,11-2,14:          NAME           'num'
2,14-2,15:          OP             '*'
2,15-2,18:          NAME           'num'
2,19-2,20:          NEWLINE        '\n'
3,0-3,1:            NL             '\n'
4,0-4,0:            DEDENT         ''
4,0-4,5:            NAME           'print'
4,5-4,6:            OP             '('
4,6-4,12:           NAME           'square'
4,12-4,13:          OP             '('
4,13-4,14:          NUMBER         '2'
4,14-4,15:          OP             ')'
4,15-4,16:          OP             ')'
4,16-4,17:          NEWLINE        ''
5,0-5,0:            ENDMARKER      ''
```

```
python -m tokenise <file.py>
```

The Parser

The Parser is where a lot of the syntax checking happens.

The Parser creates the Abstract Syntax Tree. Which is then converted into a CFG and flattened into an OPCODE list.



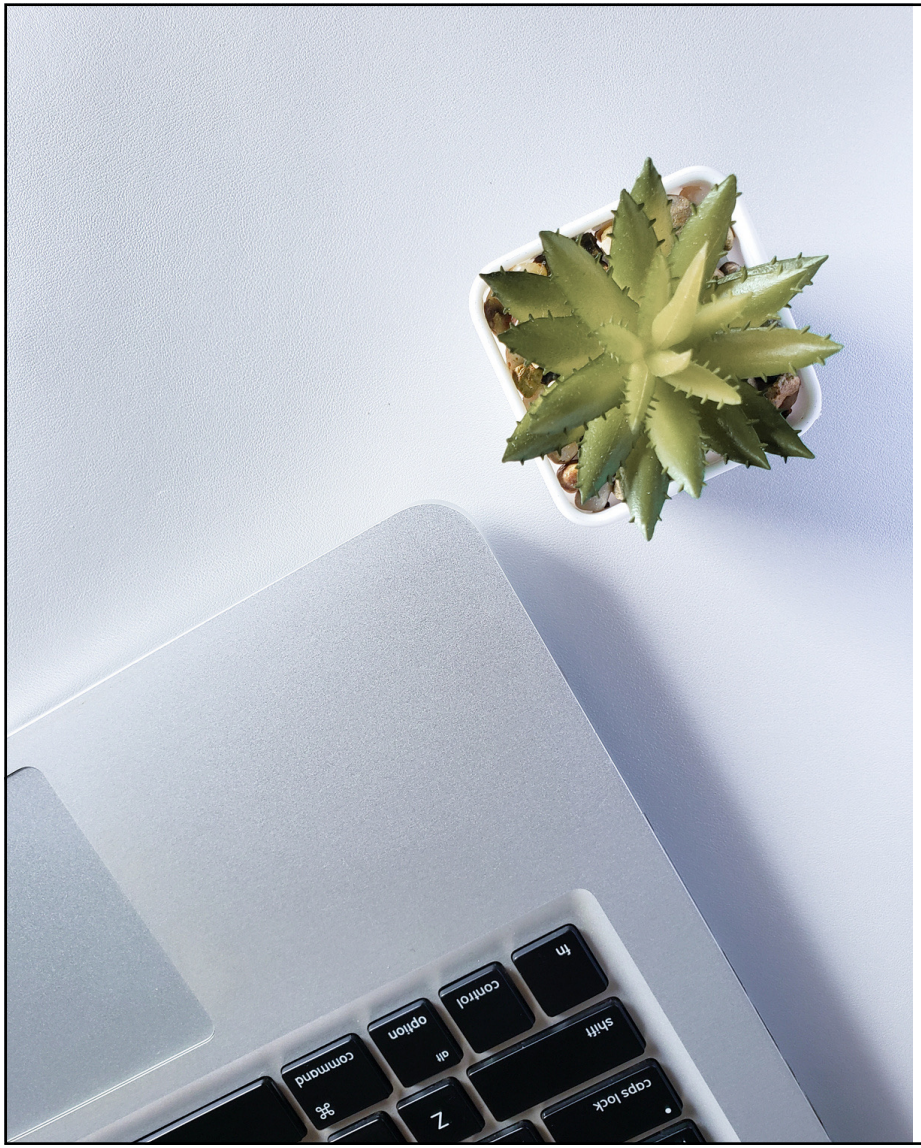
The AST

The Abstract Syntax Tree is a representation of the source code.

Each node in the AST represents a statement/expression or other specialised type like a list comprehension.

Sometime the Parser accepts statements that are syntactically wrong.

This is done to give better errors.



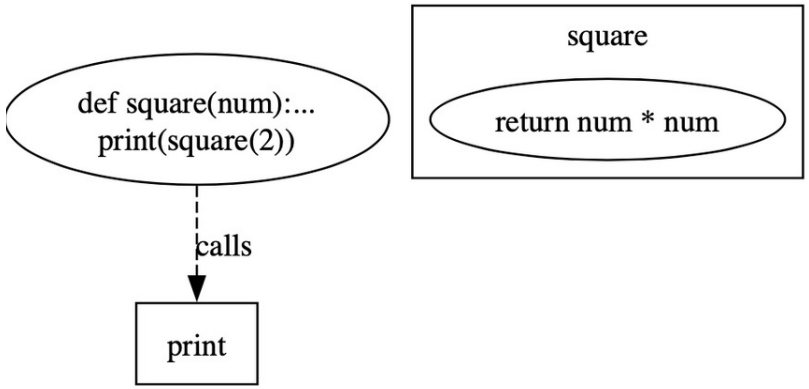
The CFG

The Control Flow Graph is a representation of the flow of a program.

The AST is converted to OPCODEs and a CFG. Which is then flattened for the interpreter

Each node is a list of bytecode that is always executed sequentially.

The CFG



The complexity of our program reflects rather heavily on our CFG....

The Complier Output

```
  1           0 LOAD_CONST               0 (<code object square at 0x1013c4a80, file "sqaure.py", line 1>)
              2 LOAD_CONST               1 ('square')
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (square)

  4           8 LOAD_NAME                1 (print)
             10 LOAD_NAME                0 (square)
             12 LOAD_CONST               2 (2)
             14 CALL_FUNCTION            1
             16 CALL_FUNCTION            1
             18 POP_TOP
             20 LOAD_CONST               3 (None)
             22 RETURN_VALUE

Disassembly of <code object square at 0x1013c4a80, file "sqaure.py", line 1>:
  2           0 LOAD_FAST                0 (num)
              2 LOAD_FAST                0 (num)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE
```

The Interpreter

Steps in the interpreter

Get OPCODE

Generate the list of code blocks that need to be run.

Compute GOTO

Switch Case or GOTO? We feel the need for speed right now.

Execute

The final step, the bit we really care about.

```
1385
1386 main_loop:
1387     for (;;) {
1388         assert(stack_pointer >= f->f_valuestack); /* else underflow */
1389         assert(STACK_LEVEL() <= co->co_stacksize);  /* else overflow */
1390         assert(!_PyErr_Occurred(tstate));
```

```
1487          switch (opcode) {
1488
1489          /* BEWARE!
1490             It is essential that any operation that fails must goto error
1491             and that all operation that succeed call [FAST_]DISPATCH() ! */
1492
1493          case TARGET(NOP): {
1494             FAST_DISPATCH();
1495          }
1496
1497          case TARGET(LOAD_FAST): {
```

```
1497        case TARGET(LOAD_FAST): {
1498            PyObject *value = GETLOCAL(oparg);
1499            if (value == NULL) {
1500                format_exc_check_arg(tstate, PyExc_UnboundLocalError,
1501                                     UNBOUNDLOCAL_ERROR_MSG,
1502                                     PyTuple_GetItem(co->co_varnames, oparg));
1503                goto error;
1504            }
1505            Py_INCREF(value);
1506            PUSH(value);
1507            FAST_DISPATCH();
1508        }
```

OPCODES?

```
  1           0 LOAD_CONST               0 (<code object square at 0x1013c4a80, file "sqaure.py", line 1>)
              2 LOAD_CONST               1 ('square')
              4 MAKE_FUNCTION            0
              6 STORE_NAME               0 (square)

  4           8 LOAD_NAME                1 (print)
             10 LOAD_NAME                0 (square)
             12 LOAD_CONST               2 (2)
             14 CALL_FUNCTION            1
             16 CALL_FUNCTION            1
             18 POP_TOP
             20 LOAD_CONST               3 (None)
             22 RETURN_VALUE

Disassembly of <code object square at 0x1013c4a80, file "sqaure.py", line 1>:
  2           0 LOAD_FAST                0 (num)
              2 LOAD_FAST                0 (num)
              4 BINARY_MULTIPLY
              6 RETURN_VALUE
```

-

Computed GOTOs

01

Once a command has been completed, head to FAST_DISPATCH to find out where to go next.

02

Jump to the correct block of code for the current OPCODE

```
1497         case TARGET(LOAD_FAST): {
1498             PyObject *value = GETLOCAL(oparg);
1499             if (value == NULL) {
1500                 format_exc_check_arg(tstate, PyExc_UnboundLocalError,
1501                                      UNBOUNDLOCAL_ERROR_MSG,
1502                                      PyTuple_GetItem(co->co_varnames, oparg));
1503                 goto error;
1504             }
1505             Py_INCREF(value);
1506             PUSH(value);
1507             FAST_DISPATCH();
1508         }
```

Specialising Adaptive Interpreter (PEP 659)

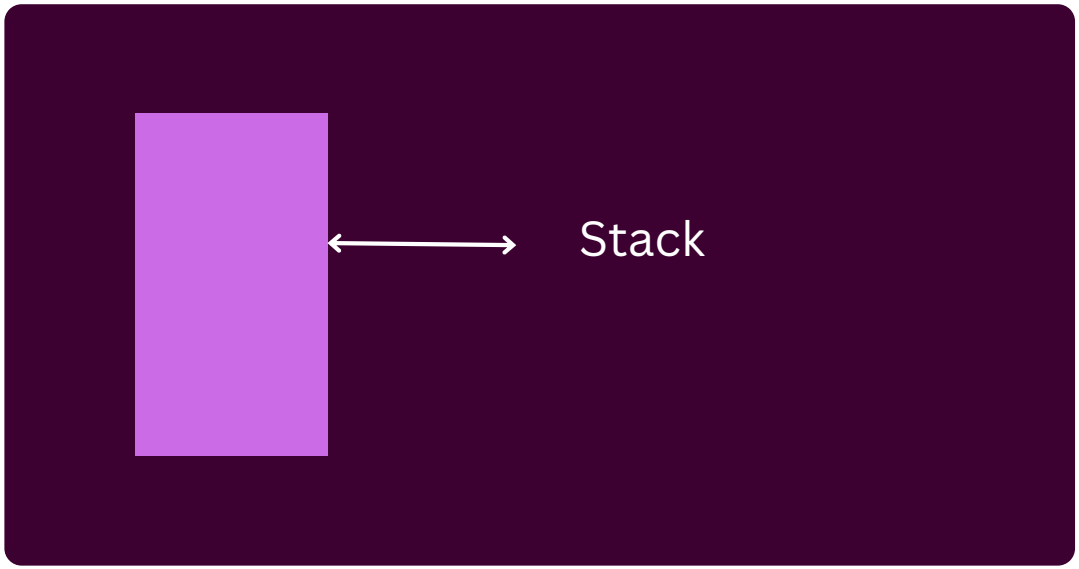
- The interpreter adapts to the program that is being run after some warm up
- The general bycode instruction has a warm up counter
- The specialised instruction has a miss counter

Eg. BINARY_OP and BINARY_OP_ADD_INT

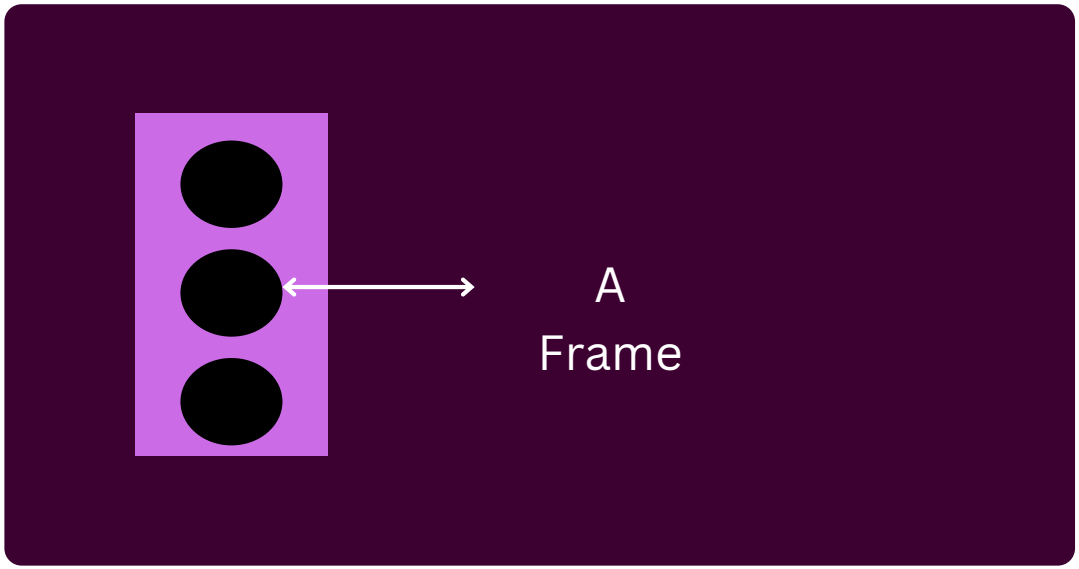
Let’s execute The Program

Frames?

Frames?

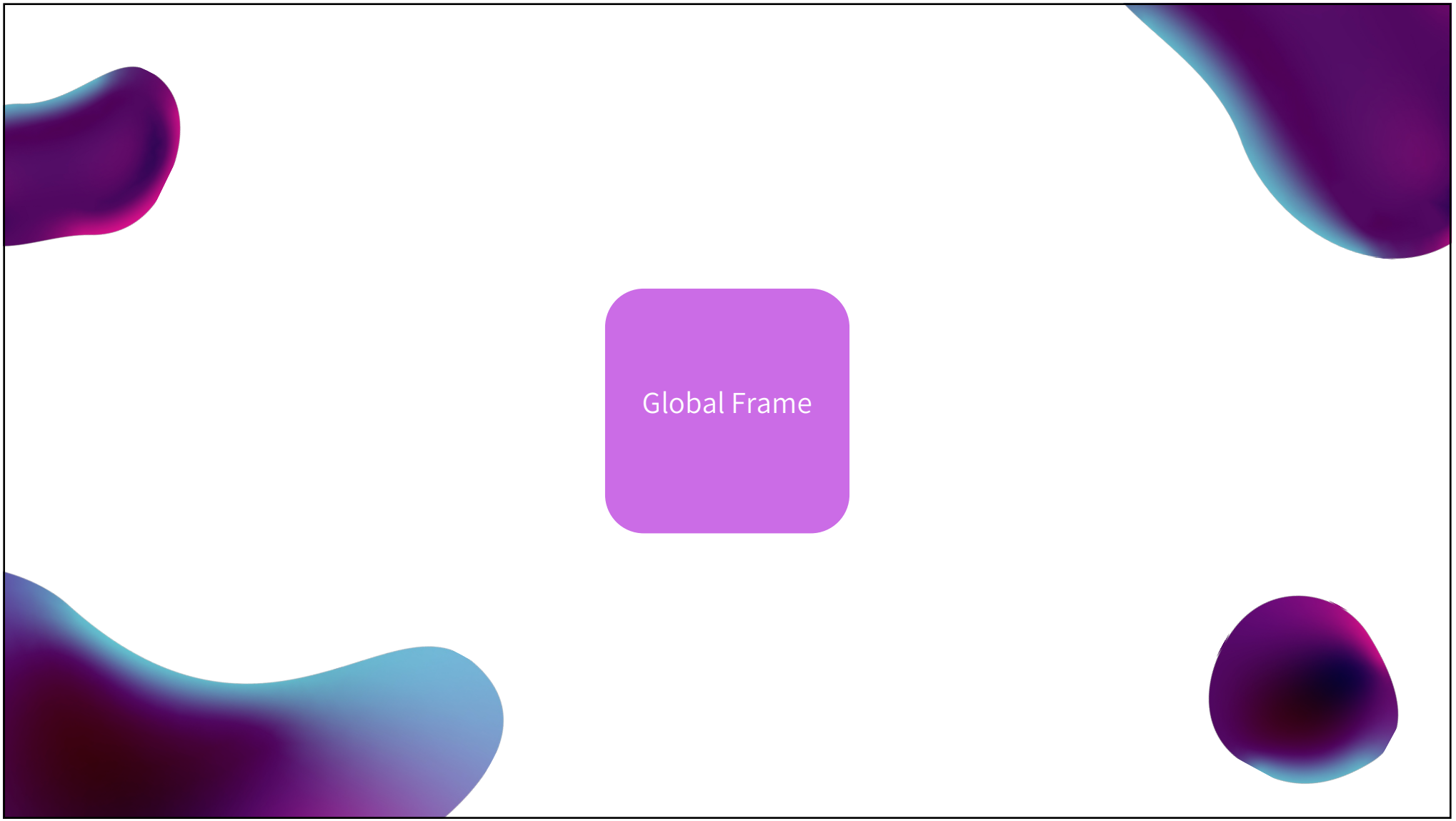


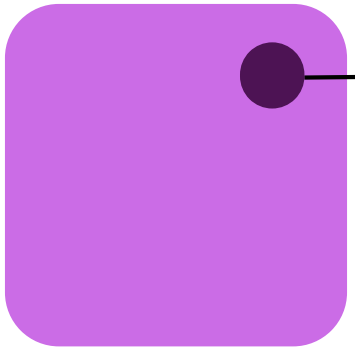
Frames?



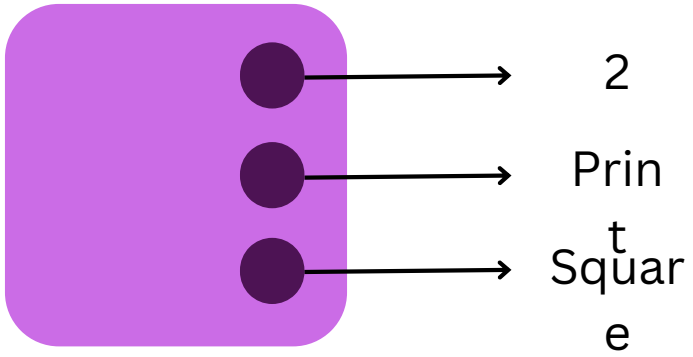
The Program

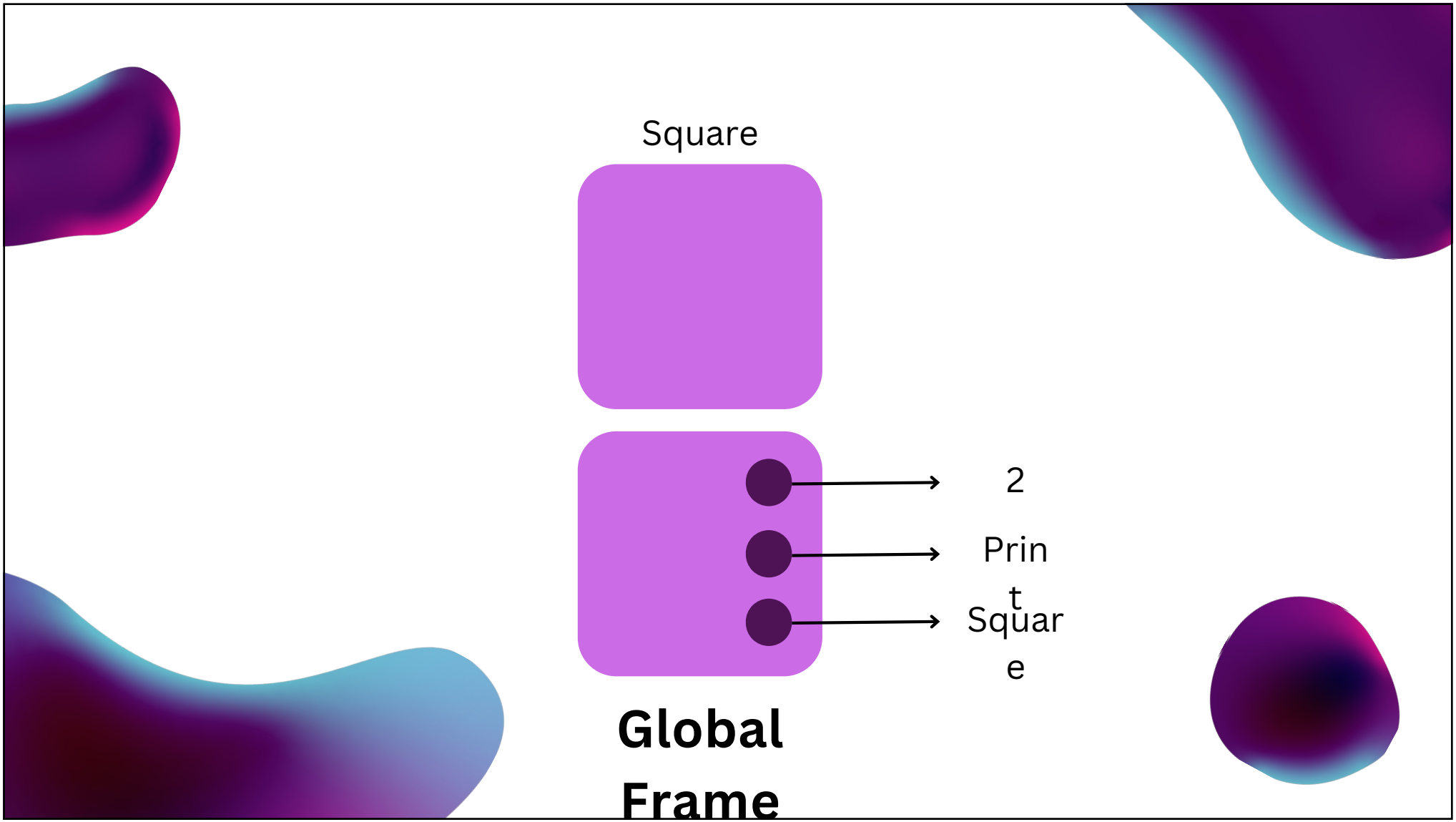
```
def square(num):
    return num*num

print(square(2))
```

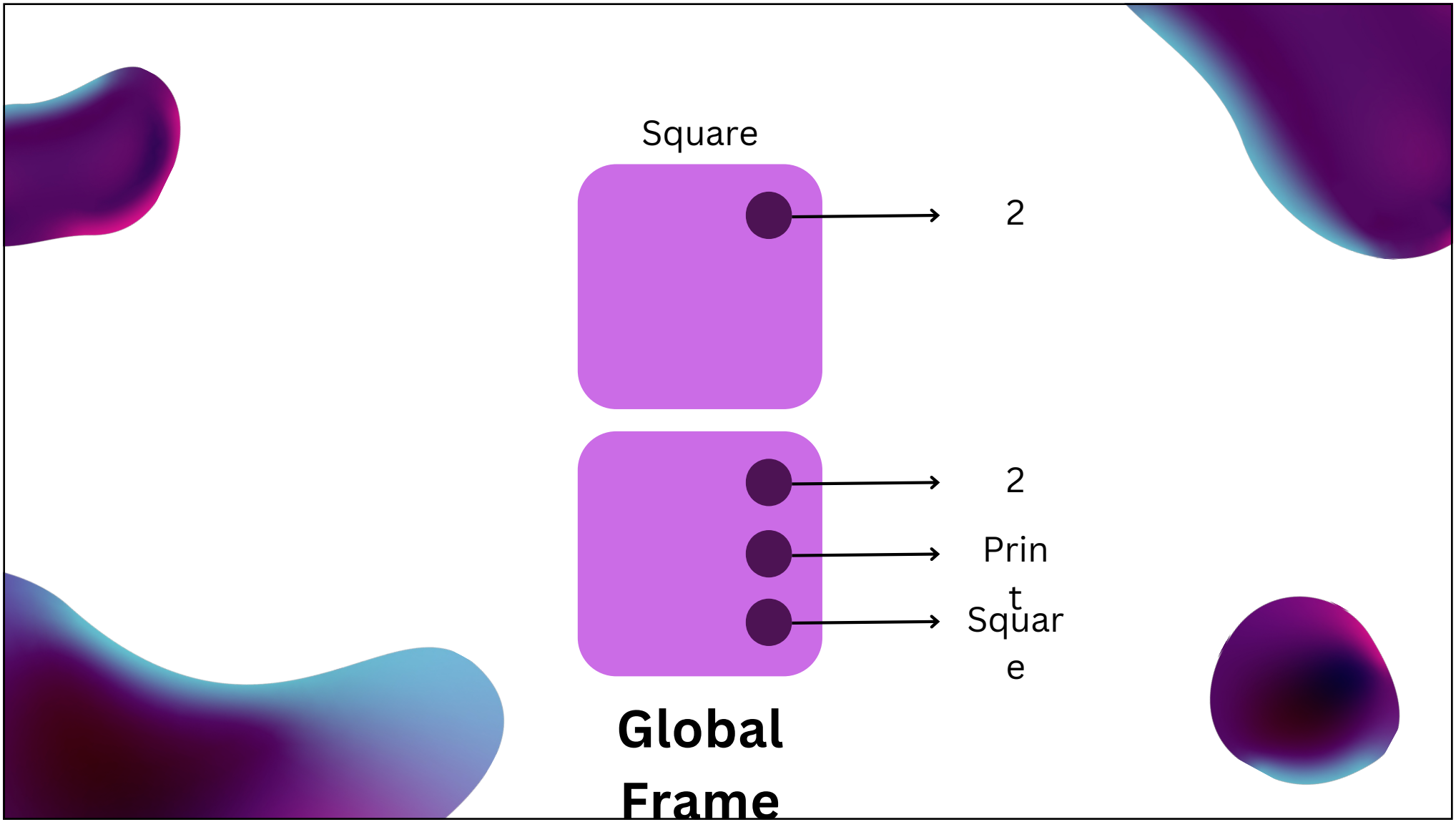

Global Frame

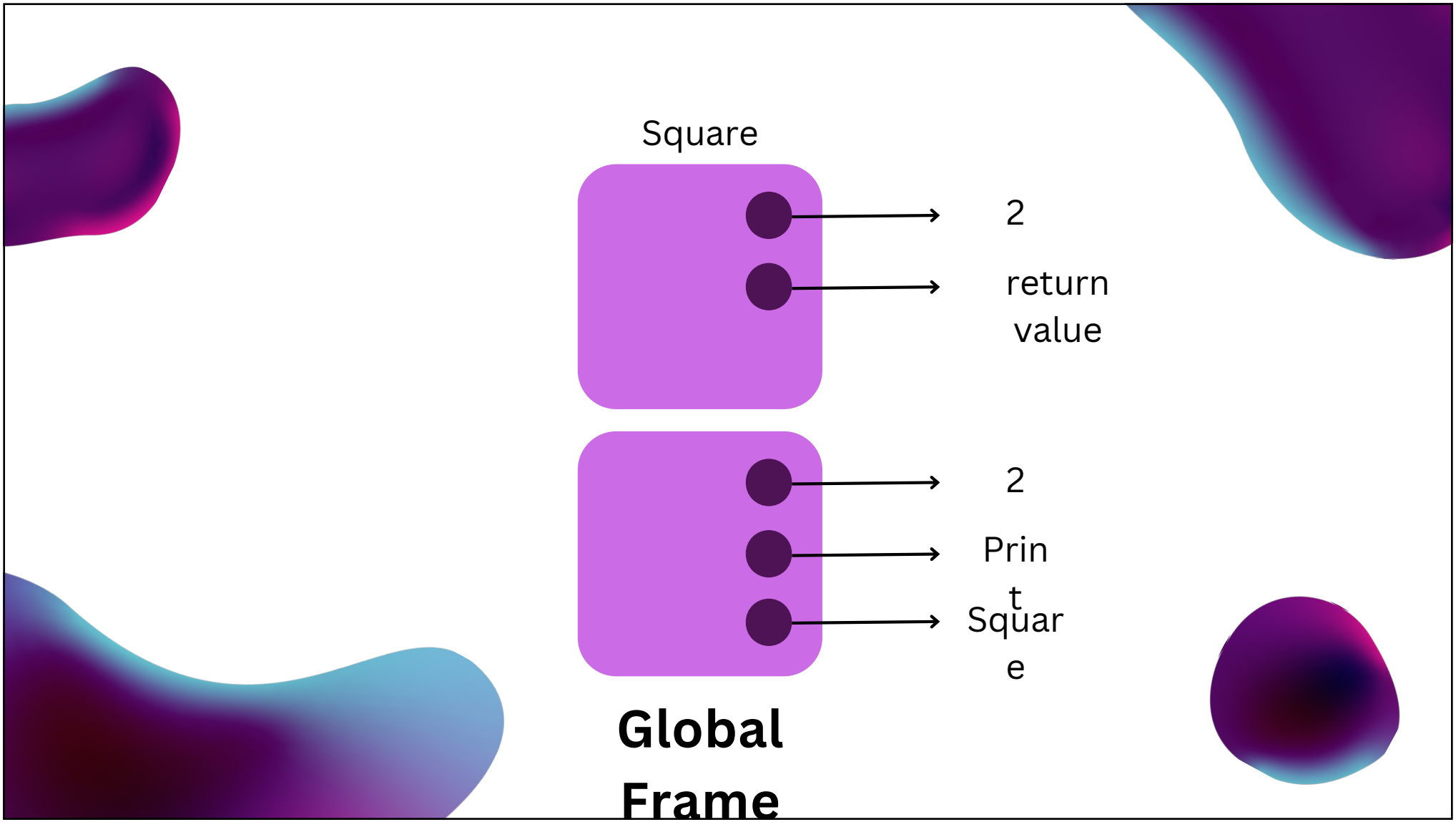


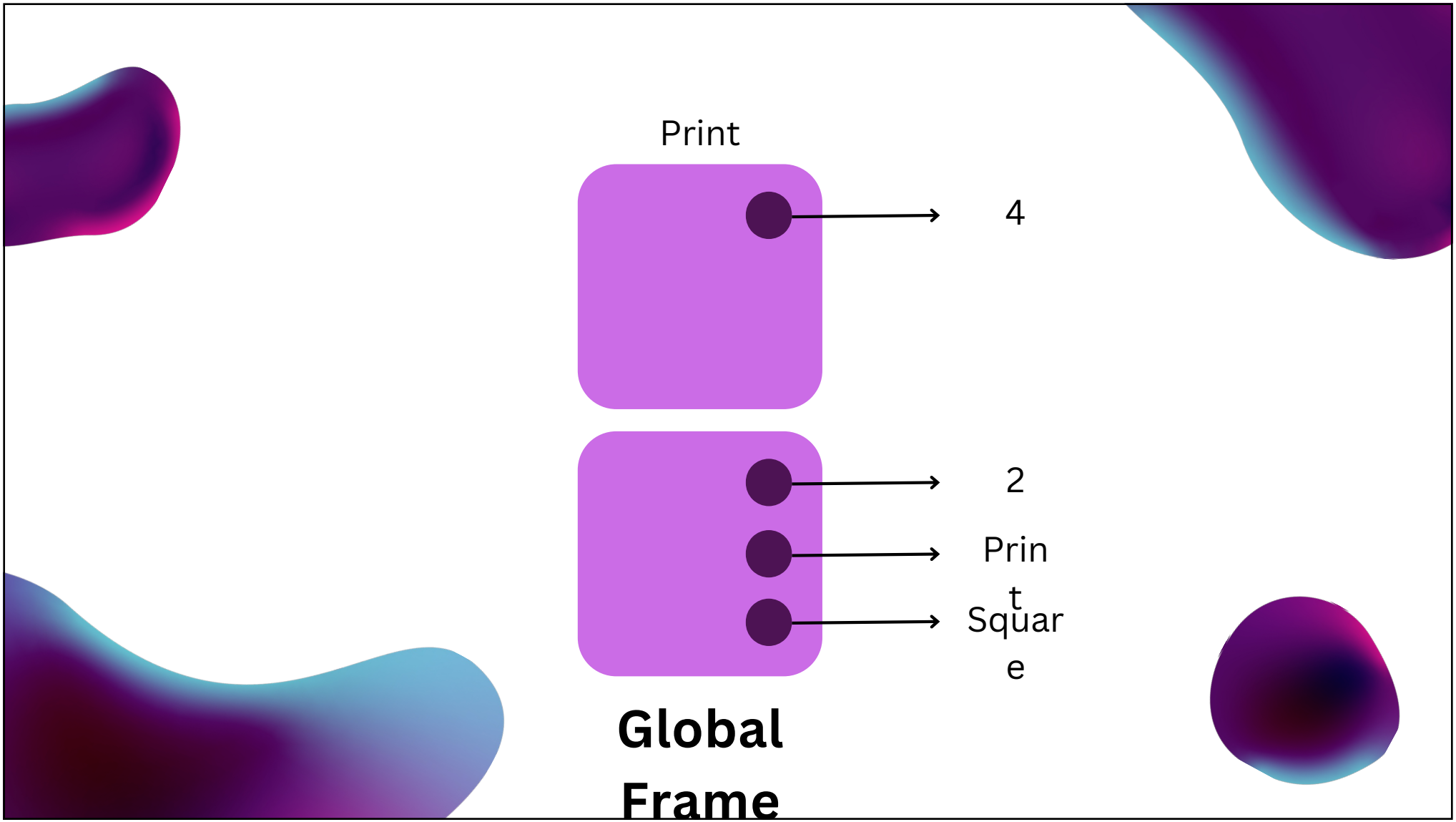
Global Frame

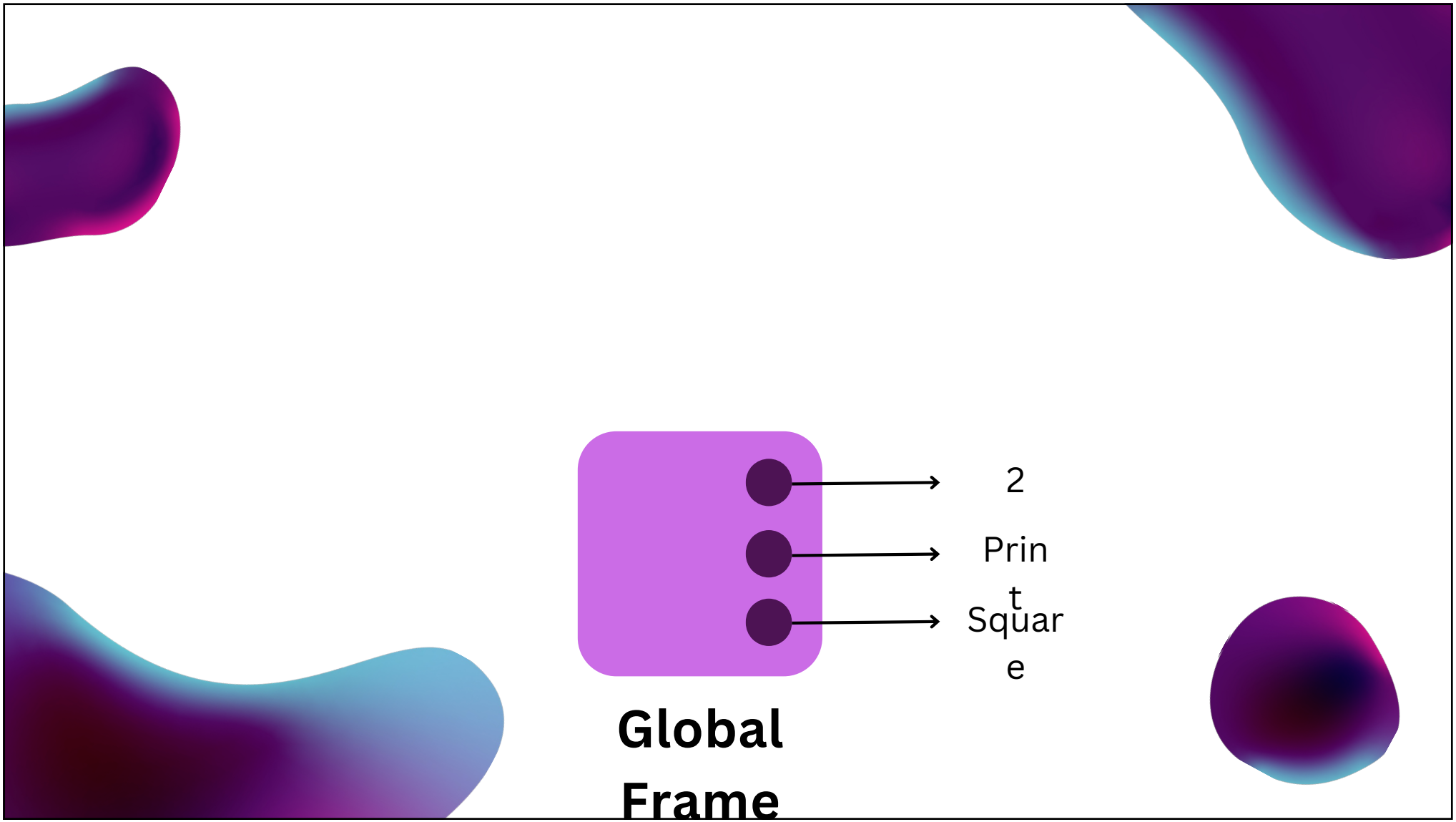

2
Print
Square
Global Frame


Square
2
Print
Square
Global Frame


Square
2
2
Print
Square
Global Frame


Square
2
return value
2
Print
Square
Global Frame


Print
4
2
Print
Square
Global Frame


2
Print
Square
Global Frame

>>> 4


The End

```
743 #ifndef Py_DEFAULT_RECURSION_LIMIT
744 #define Py_DEFAULT_RECURSION_LIMIT 1000
745 #endif
```