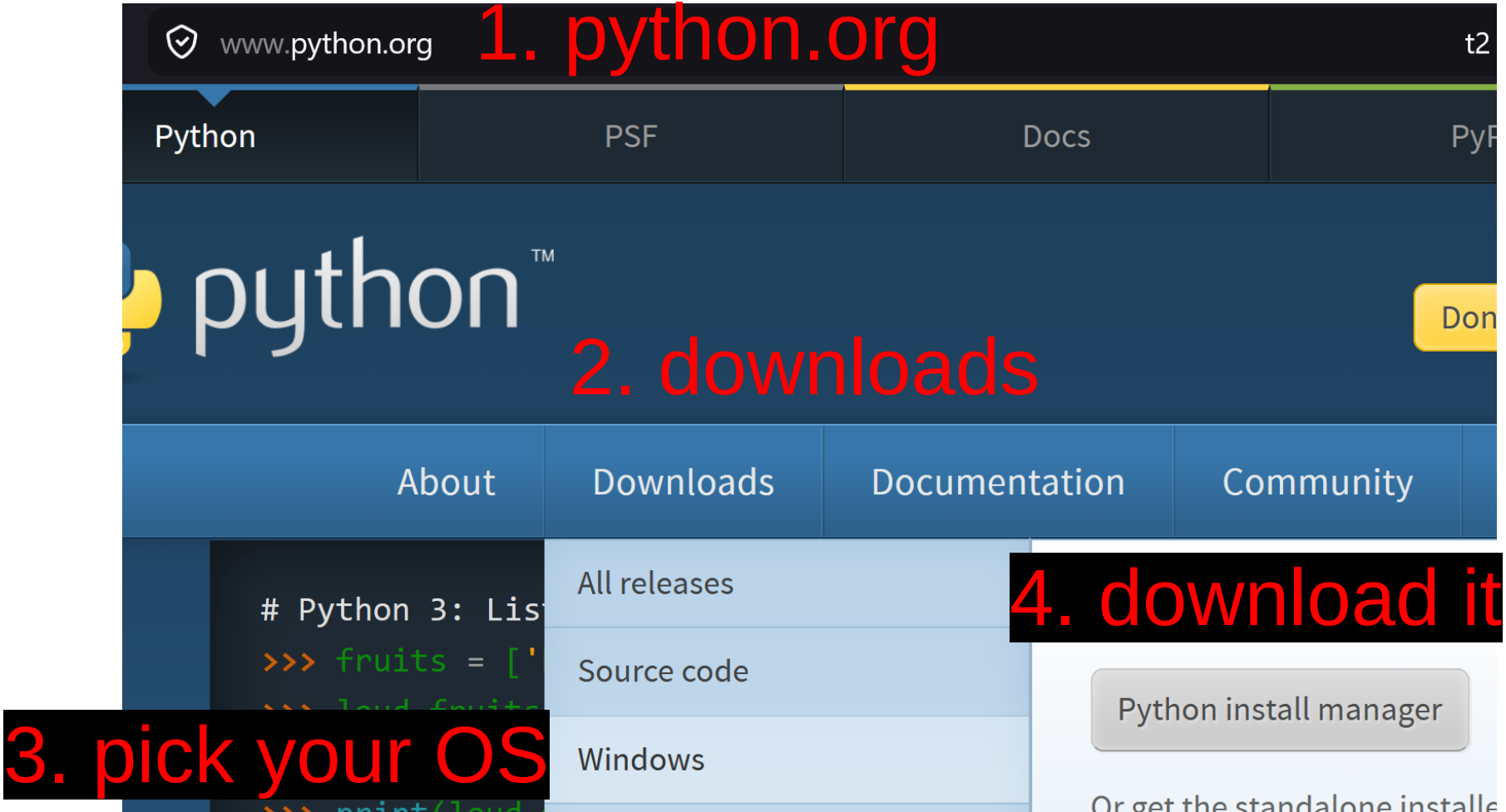
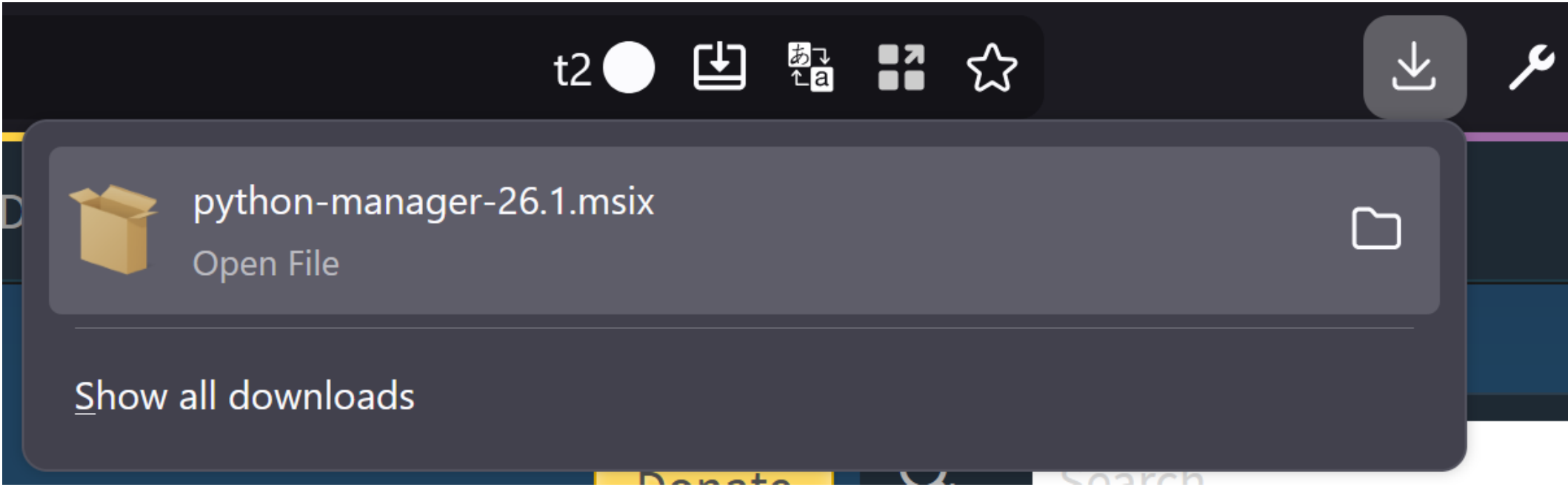
Deck 1
Getting Started With Python

Dr. Hazel “twitch.tv/hazeldotzone” Campbell

Installing Python

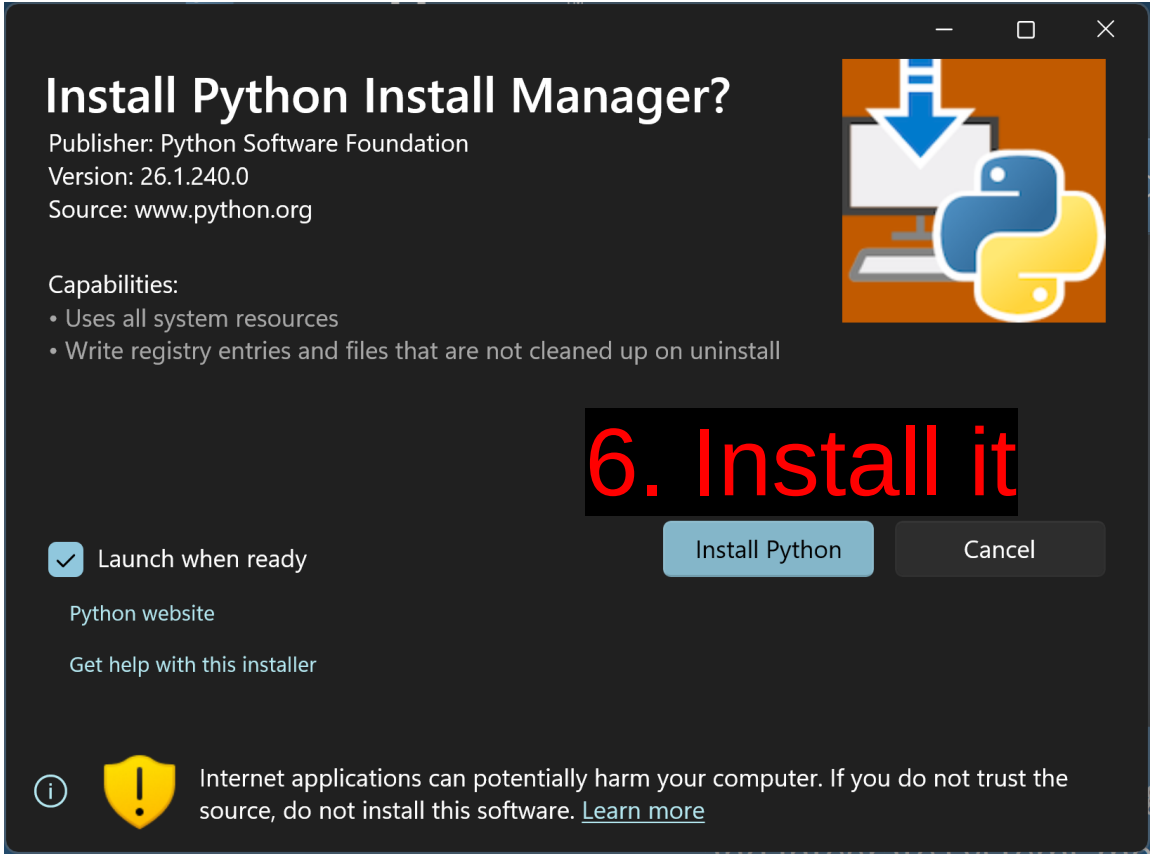


Installing Python - Windows

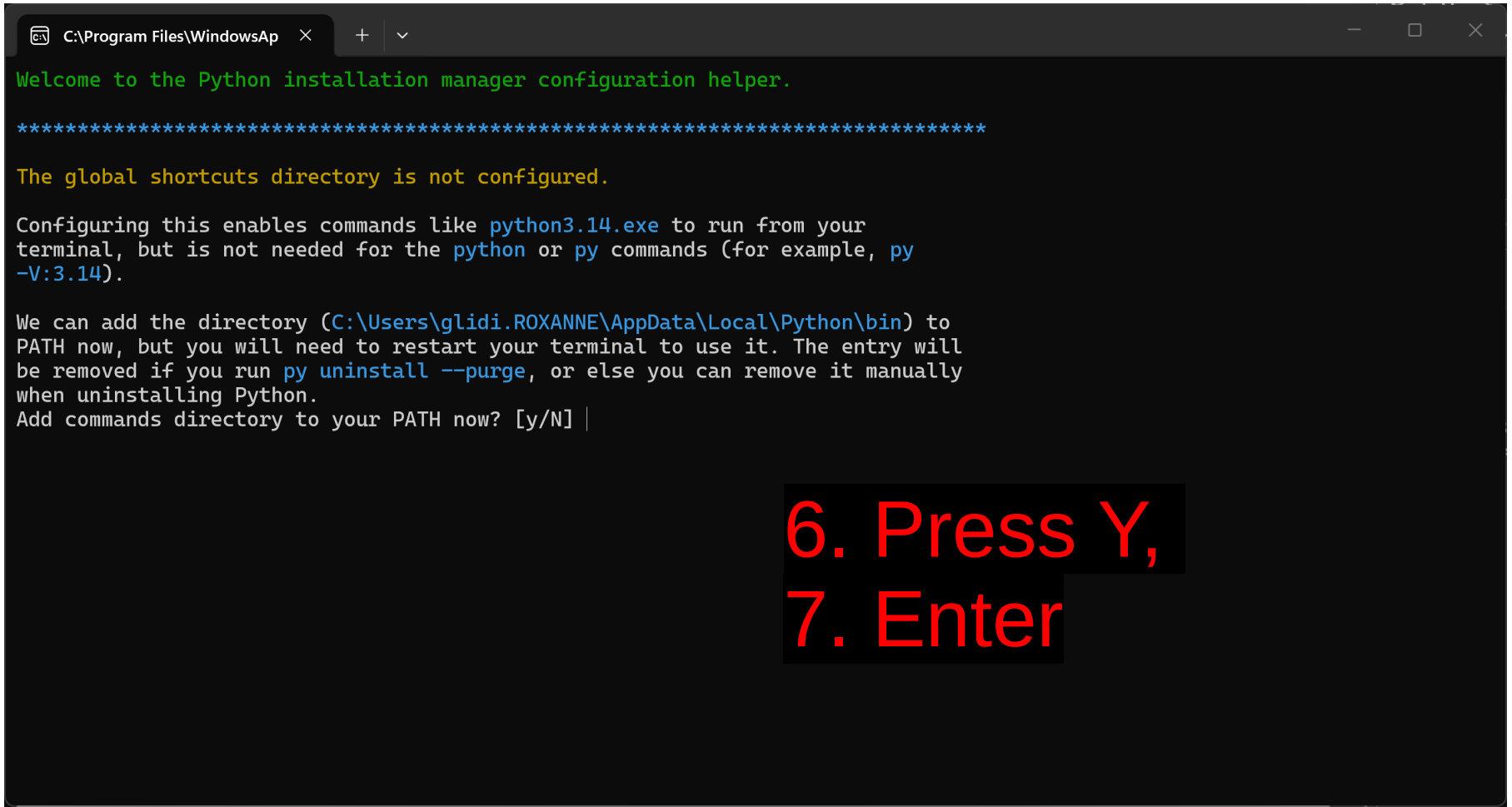


5. click the download

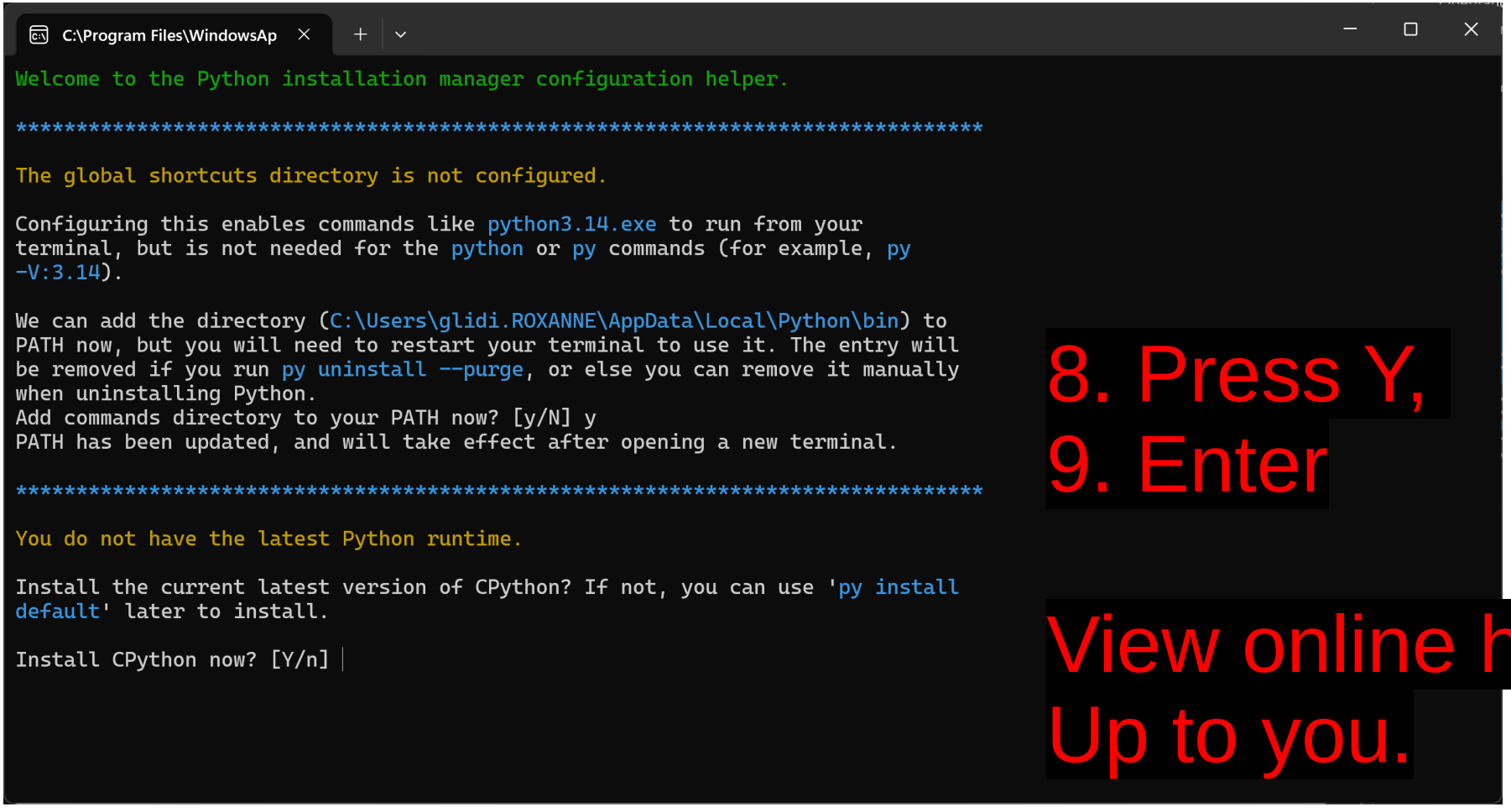
Installing Python - Windows



Installing Python - Windows



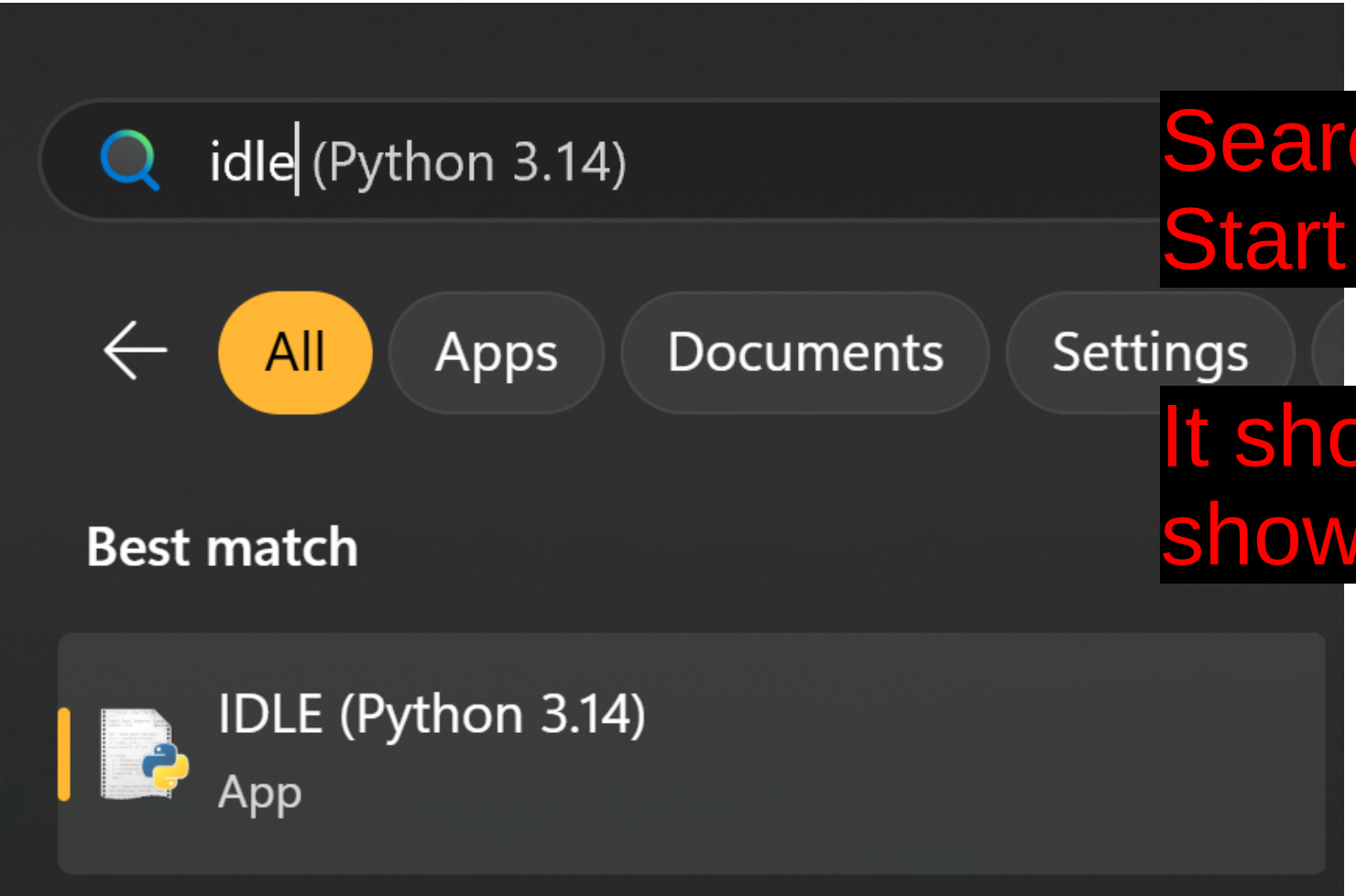
Installing Python - Windows



8. Press Y,
9. Enter

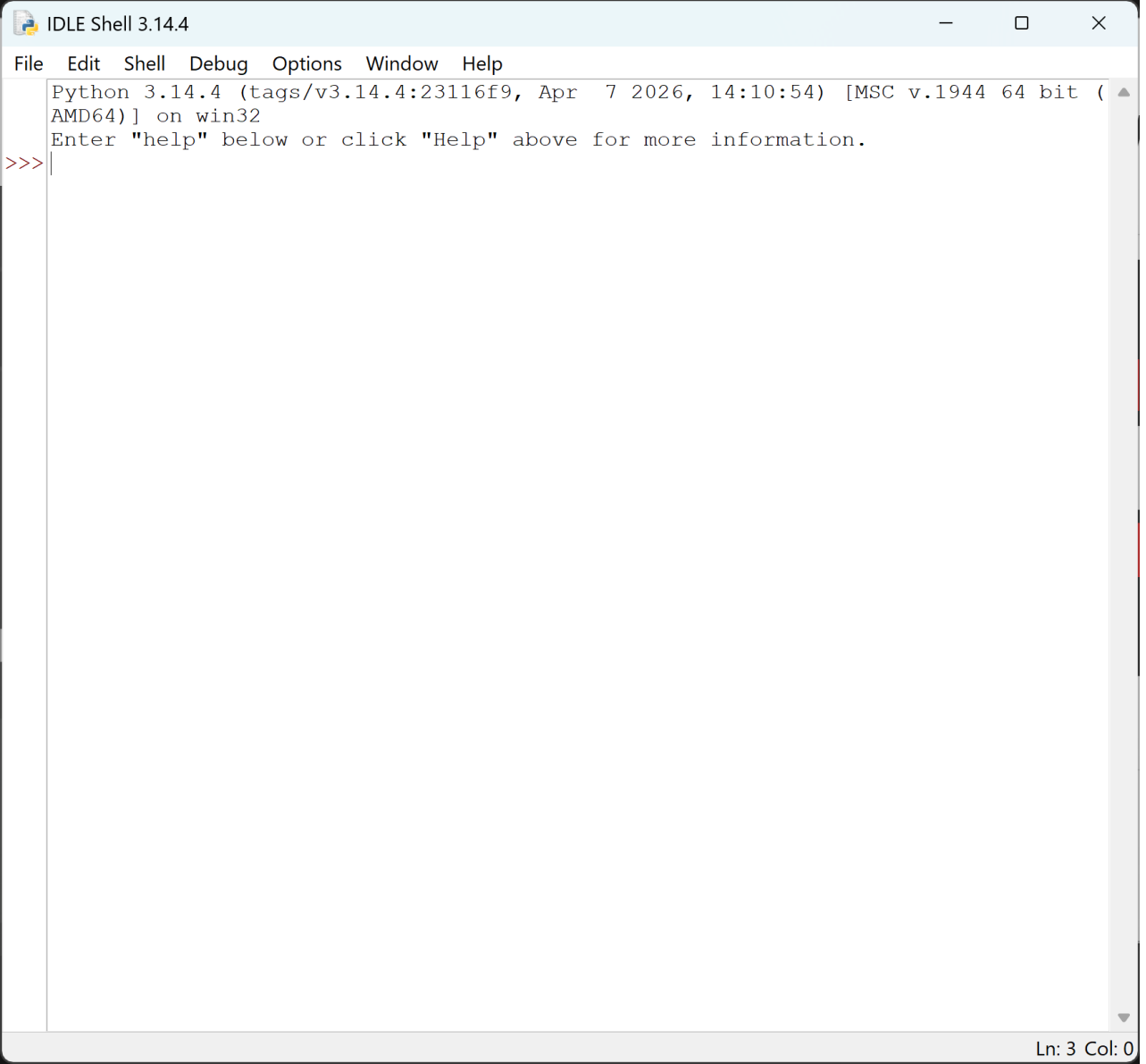
View online help?
Up to you.

Launching - Windows



Search for idle in Start Menu

It should should show up

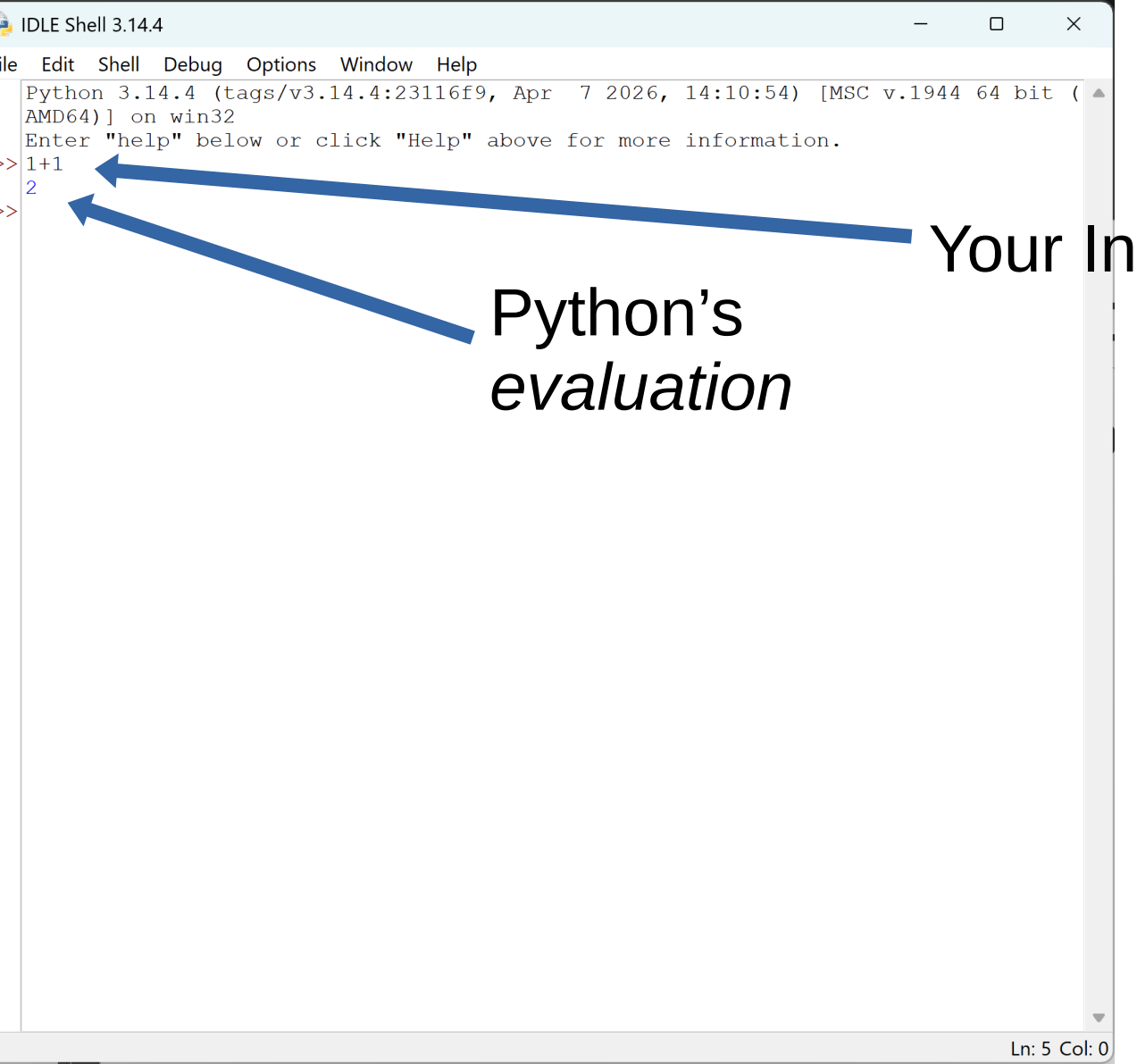


Using Idle

- Type

  1+1

- Hit enter



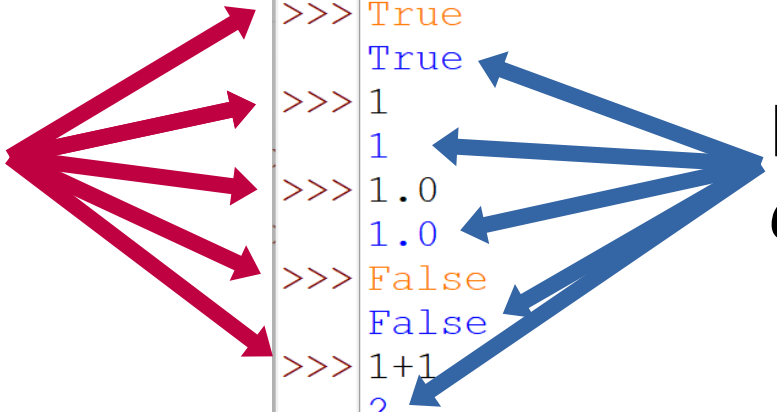
Python "REPL"

What python will do when it shows >>>

1. Read your input
2. Evaluate your input
3. Print the evaluation
4. Loop (Repeat)

"REPL"

Python “REPL”




```
AMD64)] on win32
Enter "help" below or click "Help" above for more information
>>> True
True
>>> 1
1
>>> 1.0
1.0
>>> False
False
>>> 1+1
2
>>>
```

What python will do when it shows >>>

1. Read your input
2. Evaluate your input
3. Print the evaluation
4. Loop (Repeat)

“REPL”

Python “REPL”

```
>>> false
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    false
NameError: name 'false' is not defined. Did you mean: 'False'?
>>>
```

What python will do when it shows >>>

1. Read your input
2. Evaluate your input
3. ~~Print the evaluation~~
4. Loop (Repeat)

“REPL”

In the case of an error during evaluation:

Python prints the error.

Python does not print the evaluation – it didn’t finish evaluating so it doesn’t have anything to print!

Python “REPL”

```
>>> false
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    false
NameError: name 'false' is not defined. Did you mean: 'False'?
>>>
```

The traceback

The error

The traceback will tell you where the problem was

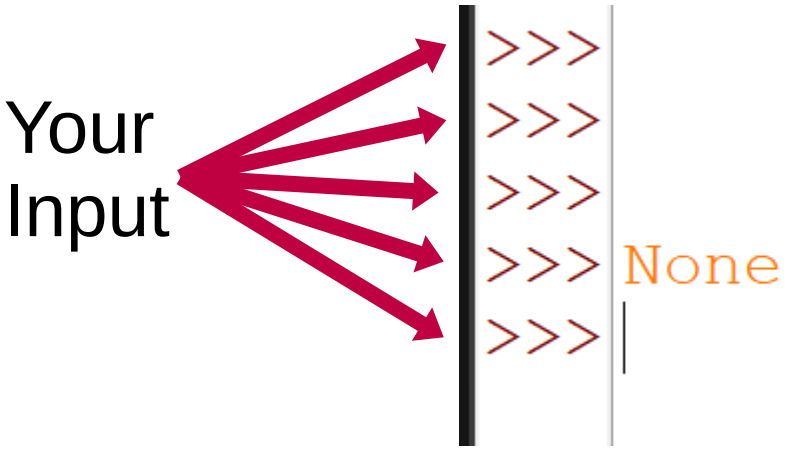
The error will tell you what the problem was

What python will do when it shows >>>

1. Read your input
2. Evaluate your input
3. ~~Print the evaluation~~
4. Loop (Repeat)

“REPL”

Python “REPL”



When Python RE**P**L doesn’t **P**rint:

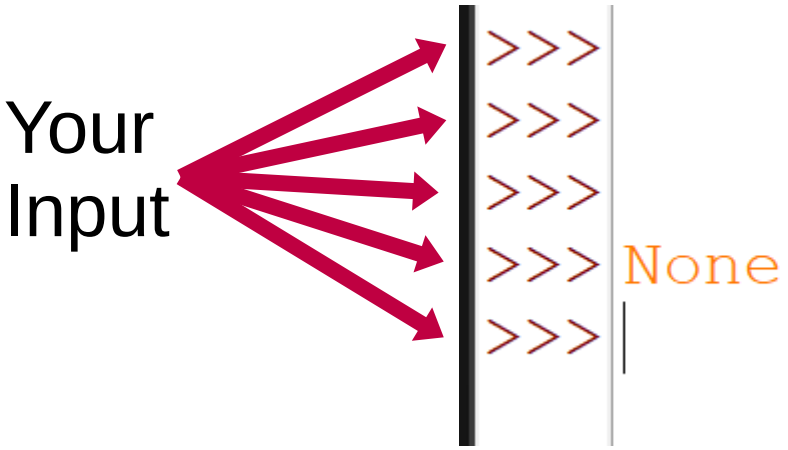
- Error
- If the input *evaluates* to `None`

What python will do when it shows >>>

1. Read your input
2. Evaluate your input
3. ~~Print the evaluation~~
4. Loop (Repeat)

“REPL”

Python “REPL”



When Python RE**P**L doesn’t **P**rint:

- When there’s an error Error
- When the input *evaluates* to None

What python will do when it shows >>>

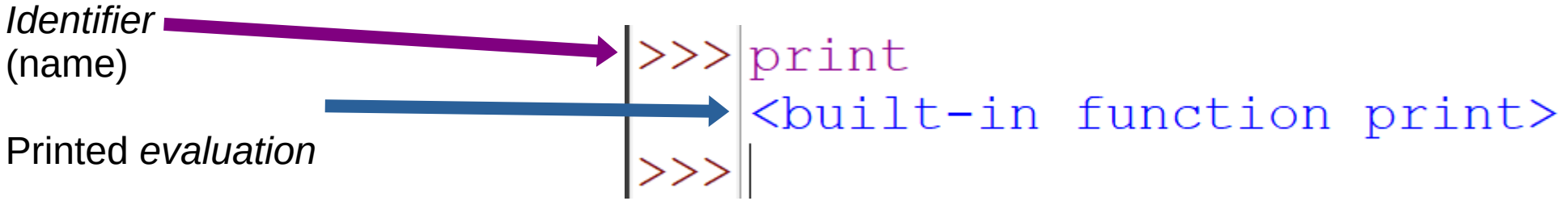
1. Read your input
2. Evaluate your input
~~3. Print the evaluation~~
4. Loop (Repeat)

“REPL”

Functions

To talk about a function in python we use its name:

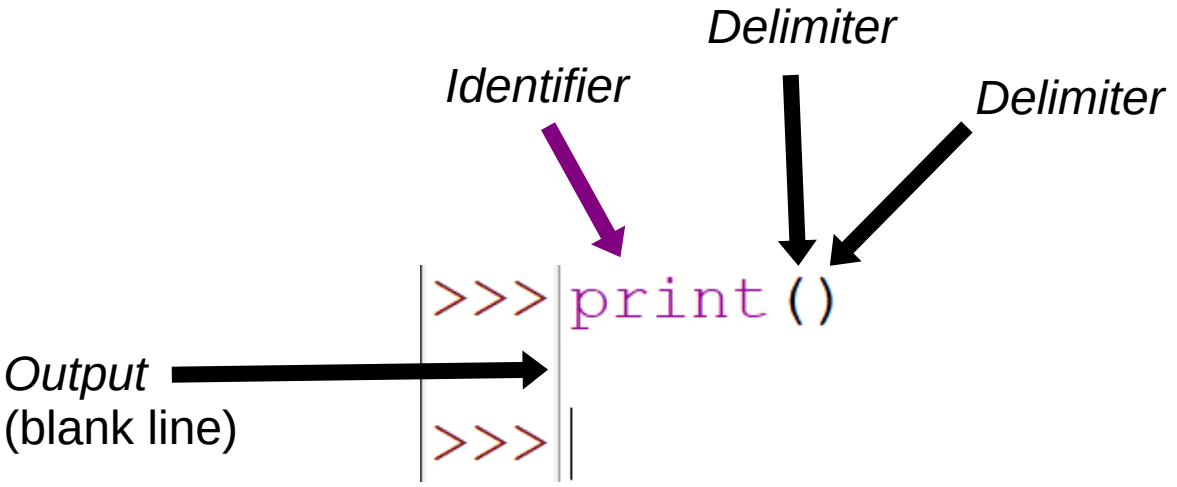
The name of something in the program code is called an *identifier*



Erm... I can't really show you... but I can tell you some things about the evaluation
* it's built in (included in python)
* it's a function
* its name is print

Functions

What if we want to do more than talk about it?
Let's *call* a function.



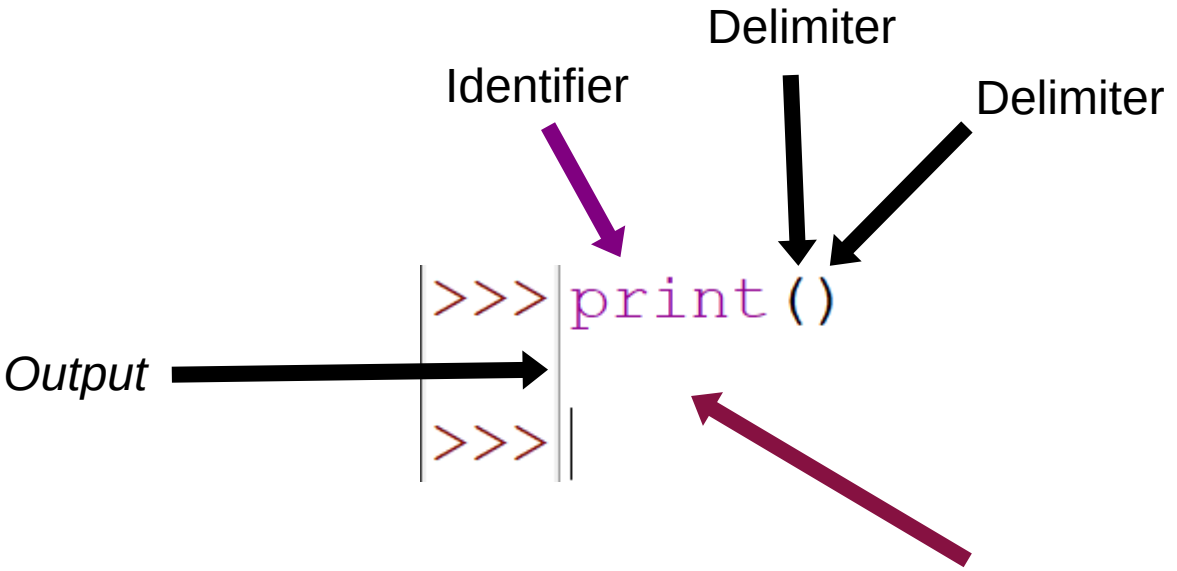
The identifier of a function followed by parenthesis -> Call the function

- *Parentheses* are a kind of *delimiters*

The `print` function *writes* some *output* (here it's a blank line)

Functions

What if we want to do more than talk about it?
Let's *call* a function.

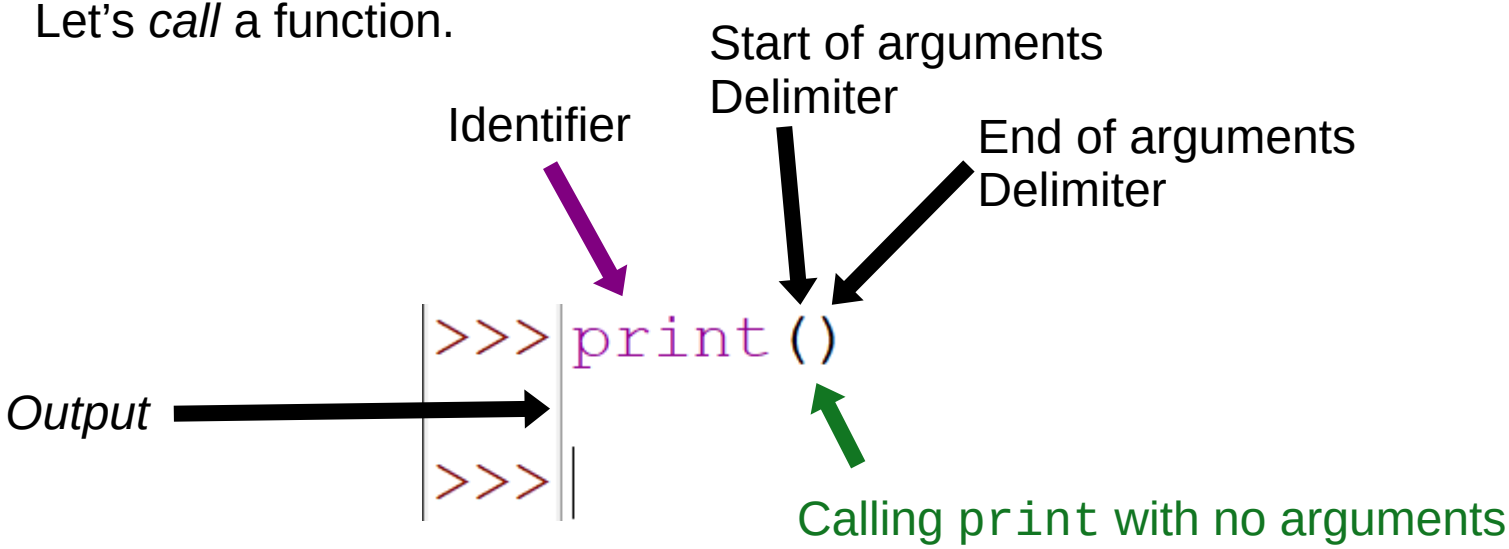


But wait... where's the printed evaluation??? Shouldn't it be right here?

- The input evaluated to `None`
- When it's None we don't get printed evaluation from REPL!

Functions

What if we want to do more than talk about it?
Let's *call* a function.

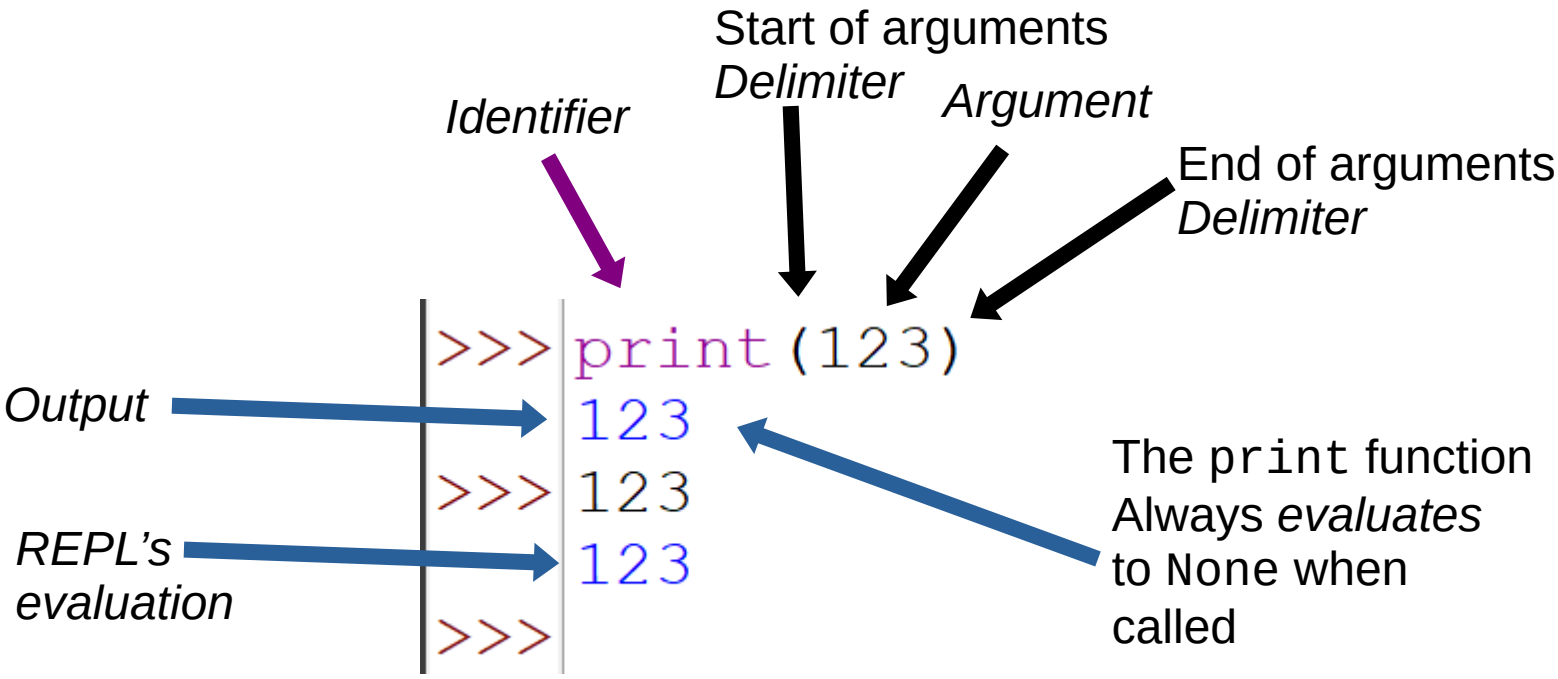


When an *identifier* is followed by the *argument delimiters* ( ) that means we are *calling* it

- There is nothing between the *start of arguments delimiter* (and the *end of arguments delimiter*)
- ... so we have no *arguments*

Functions

What if we want to do more than talk about it?
Let's *call* a function.



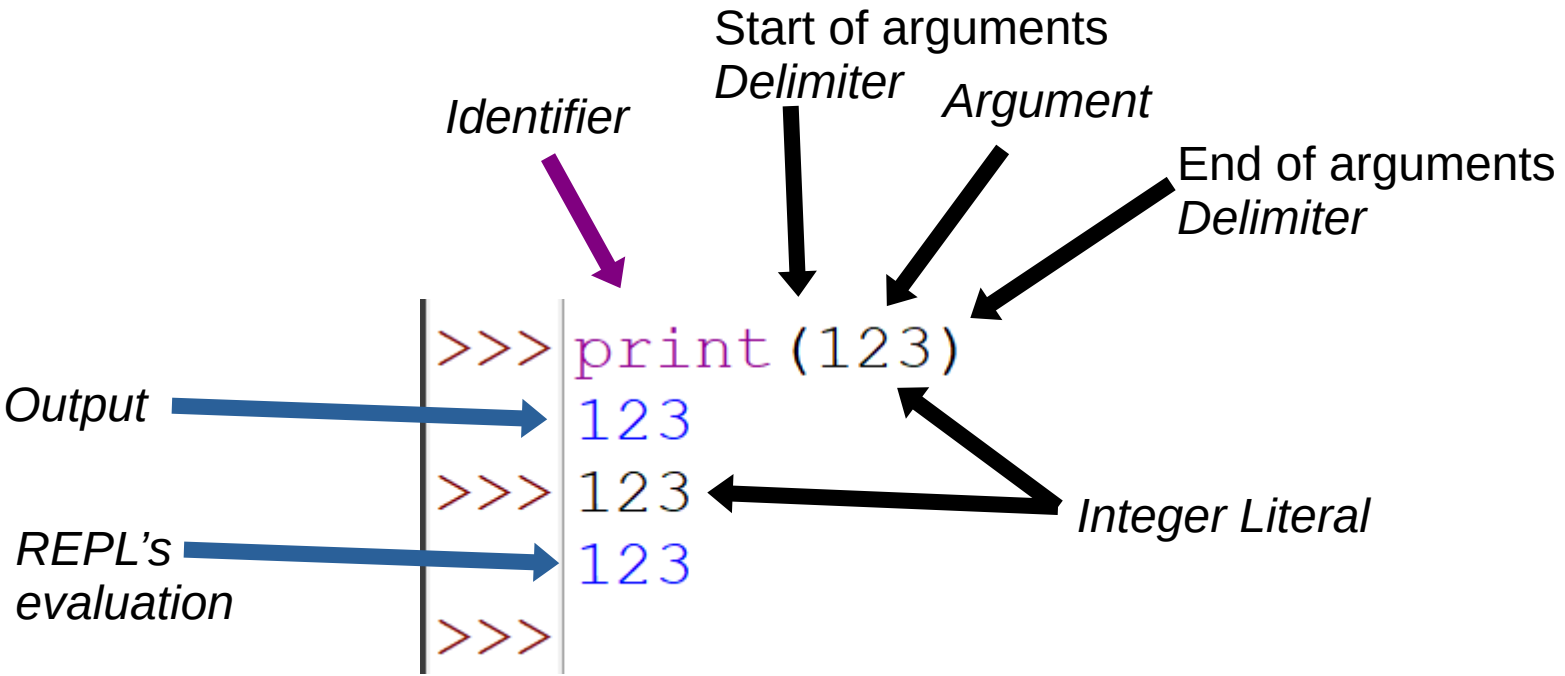
How can we tell the difference between REPL's evaluation and a Function's output

We can be sure this is not what it *evaluated* to

- When in doubt, print it out! (The REPL is only a convenience)

Literals

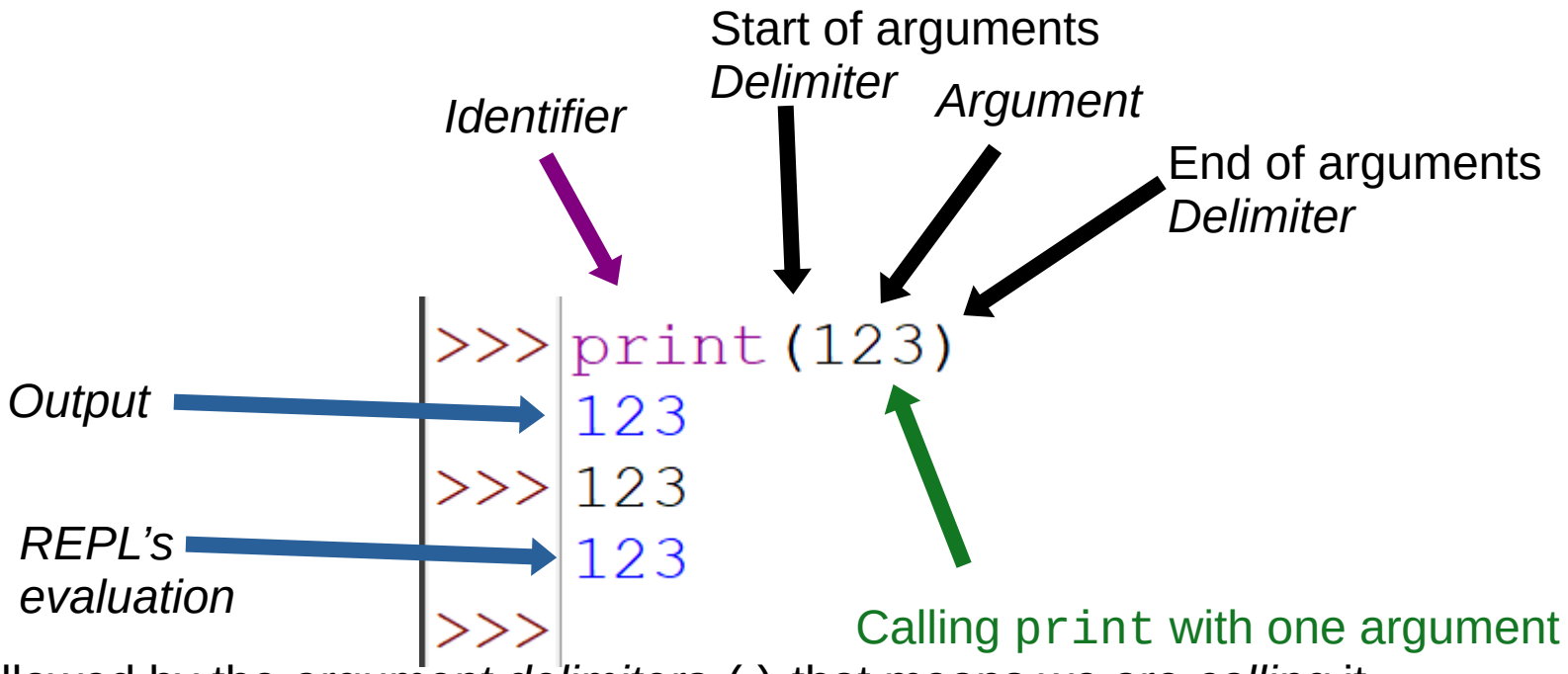
What is 123 anyway?



- Integer: A *type* of number that has no decimals or fractions
- Literal: The *value* is listed out exactly in the program code without using an *identifier*

Literals

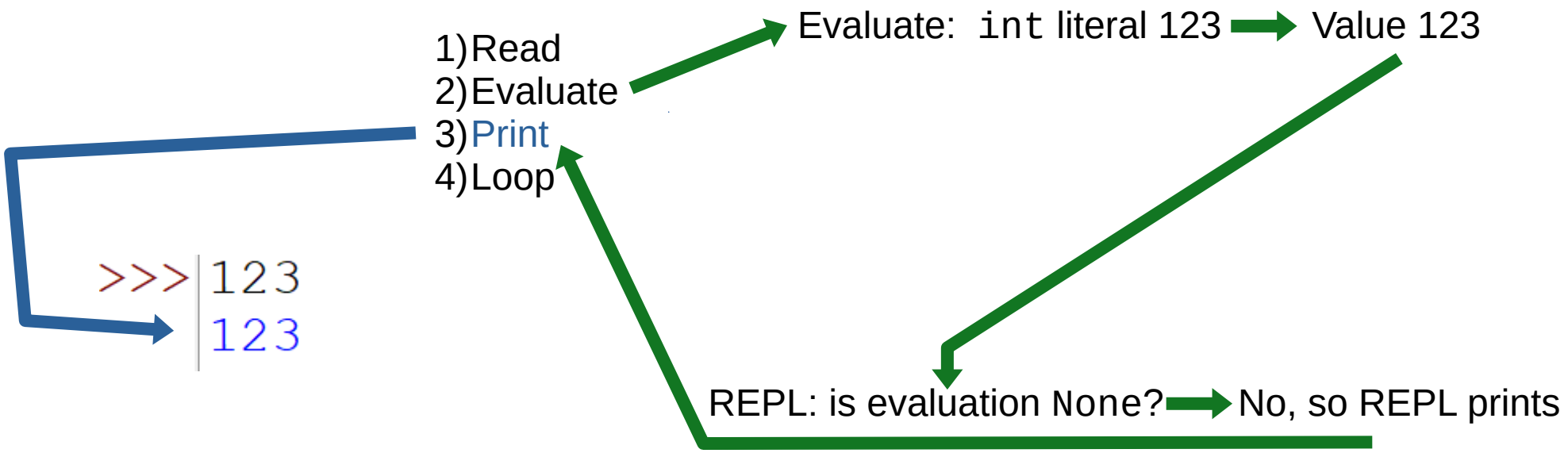
What is 123 anyway?



When an *identifier* is followed by the *argument delimiters* ( ) that means we are *calling* it

- 123 is between the *start of arguments delimiter* (and the *end of arguments delimiter*)
- ... so we are calling `print` with the argument 123

Function Calls



It's critical to remember that the evaluations (`123`, `int`)

Are separate from the written outputs (nothing, `<class 'int'>`)

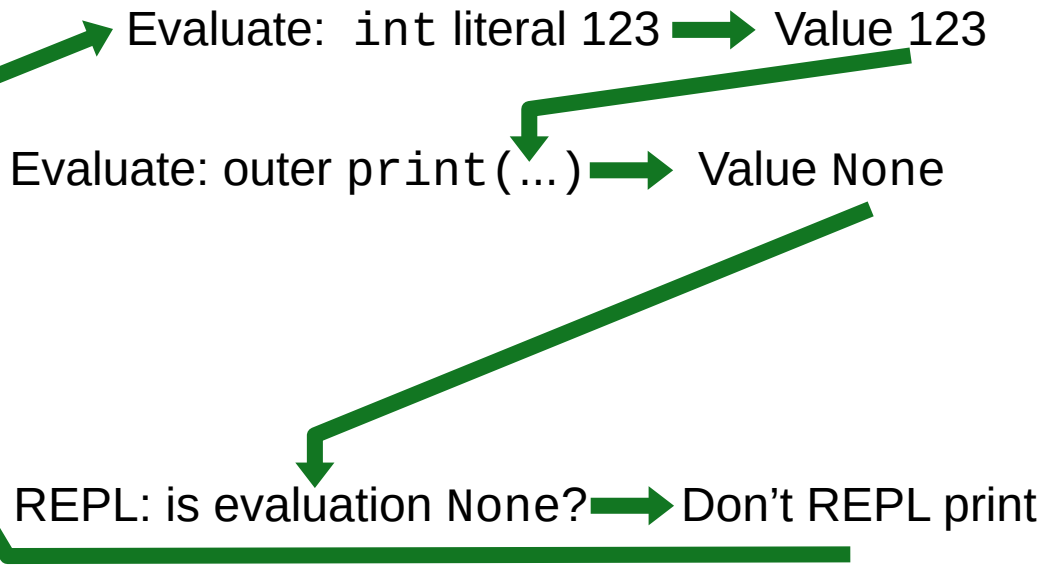
When evaluating the int literal `123`, we don't get a written output

Function Calls

Evaluate the inside first like in math!

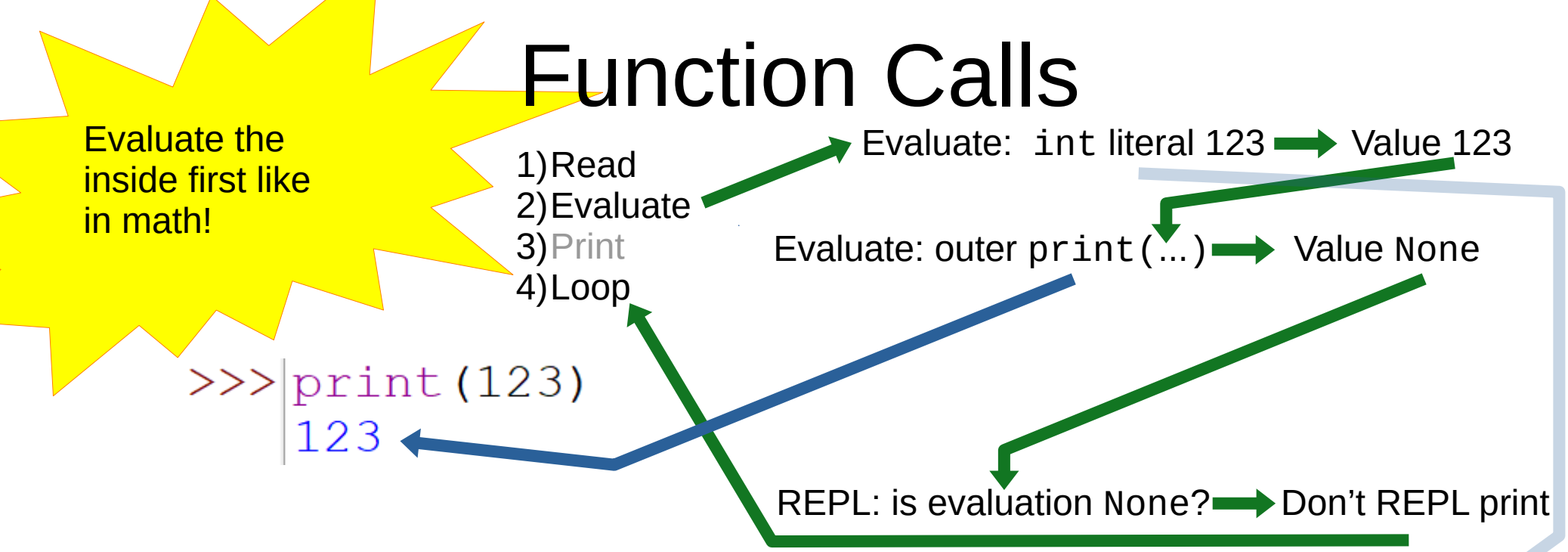
```
>>> print(123)
123
```

1) Read
2) Evaluate
3) Print
4) Loop



Function Calls

Evaluate the inside first like in math!



It's critical to remember that the evaluations (`123`, `None`)

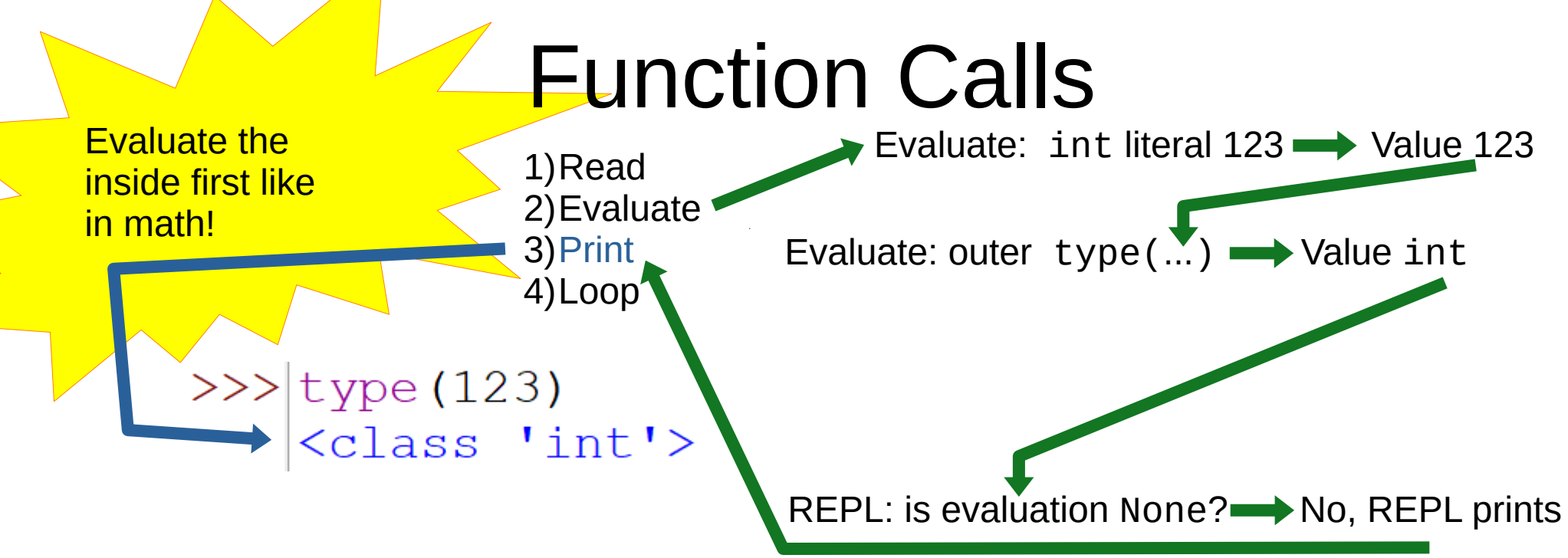
Are separate from the written outputs (nothing, `123`)

When evaluating the int literal `123`, we don't get a written output

Function Calls

Evaluate the inside first like in math!

1) Read
2) Evaluate
3) Print
4) Loop




```
>>> type(123)
<class 'int'>
```

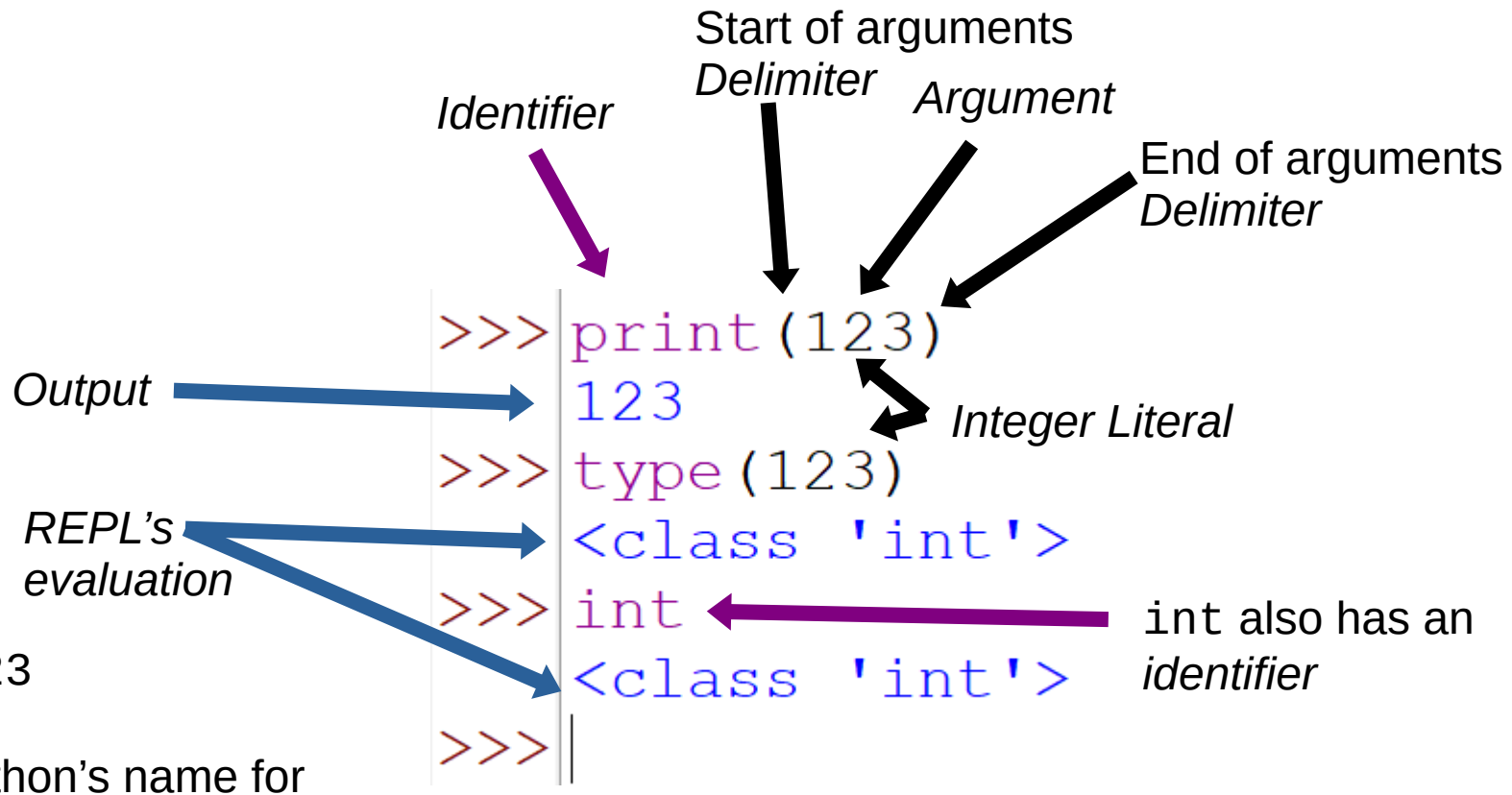
It's critical to remember that the evaluations (`123`, `int`)

Are separate from the written outputs (nothing, `<class 'int'>`)

When evaluating the int literal `123`, we don't get a written output

Literals

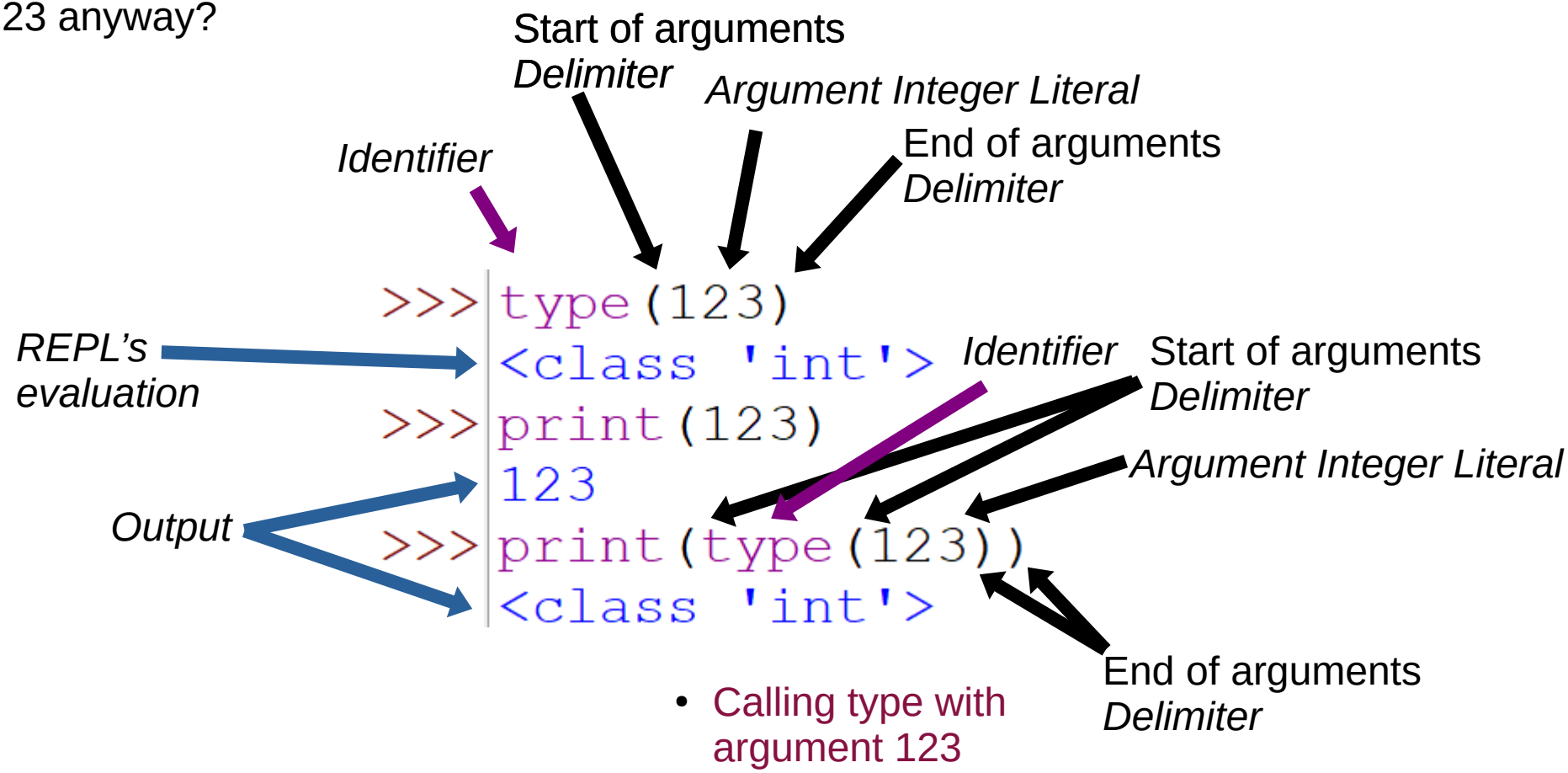
What is 123 anyway?



- The integer literal `123`
  - Value is 123
  - Type is `int` - Python's name for integers

Nesting Function Calls

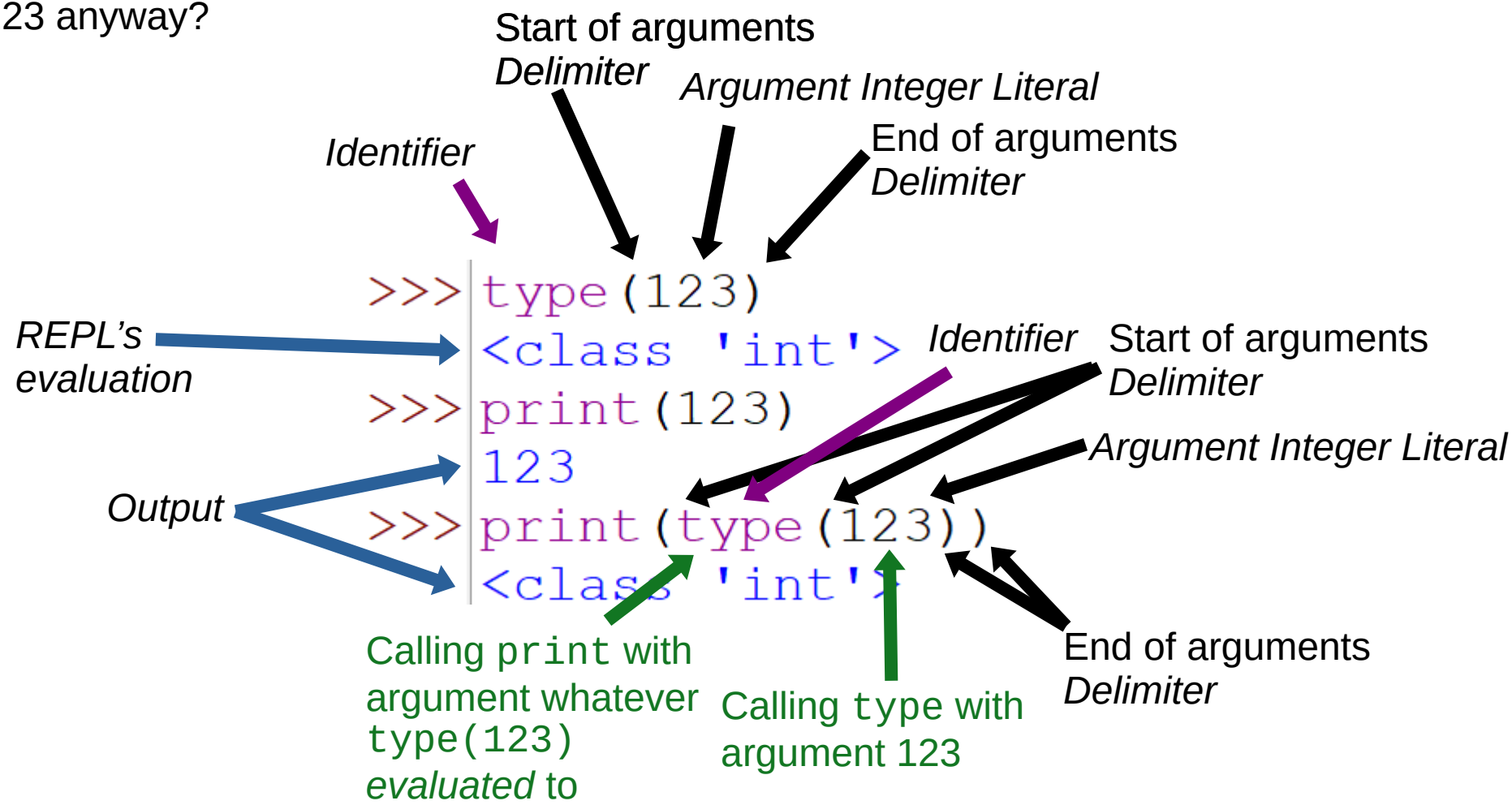
What is 123 anyway?



- Calling type with argument 123

Nesting Function Calls

What is 123 anyway?

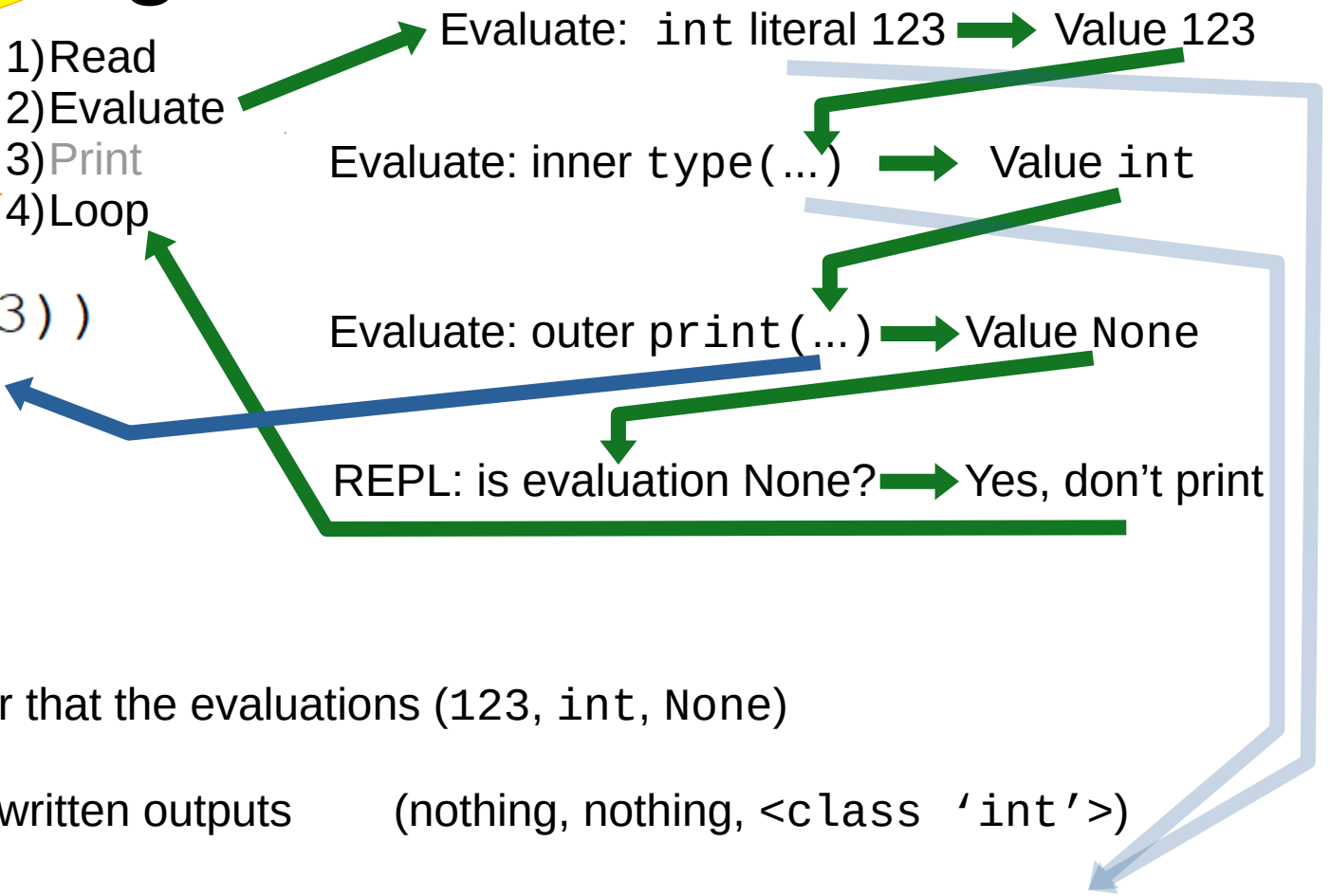


Nesting Function Calls

Evaluate the inside first like in math!

1) Read
2) Evaluate
3) Print
4) Loop

```
>>> print(type(123))
<class 'int'>
>>>
```



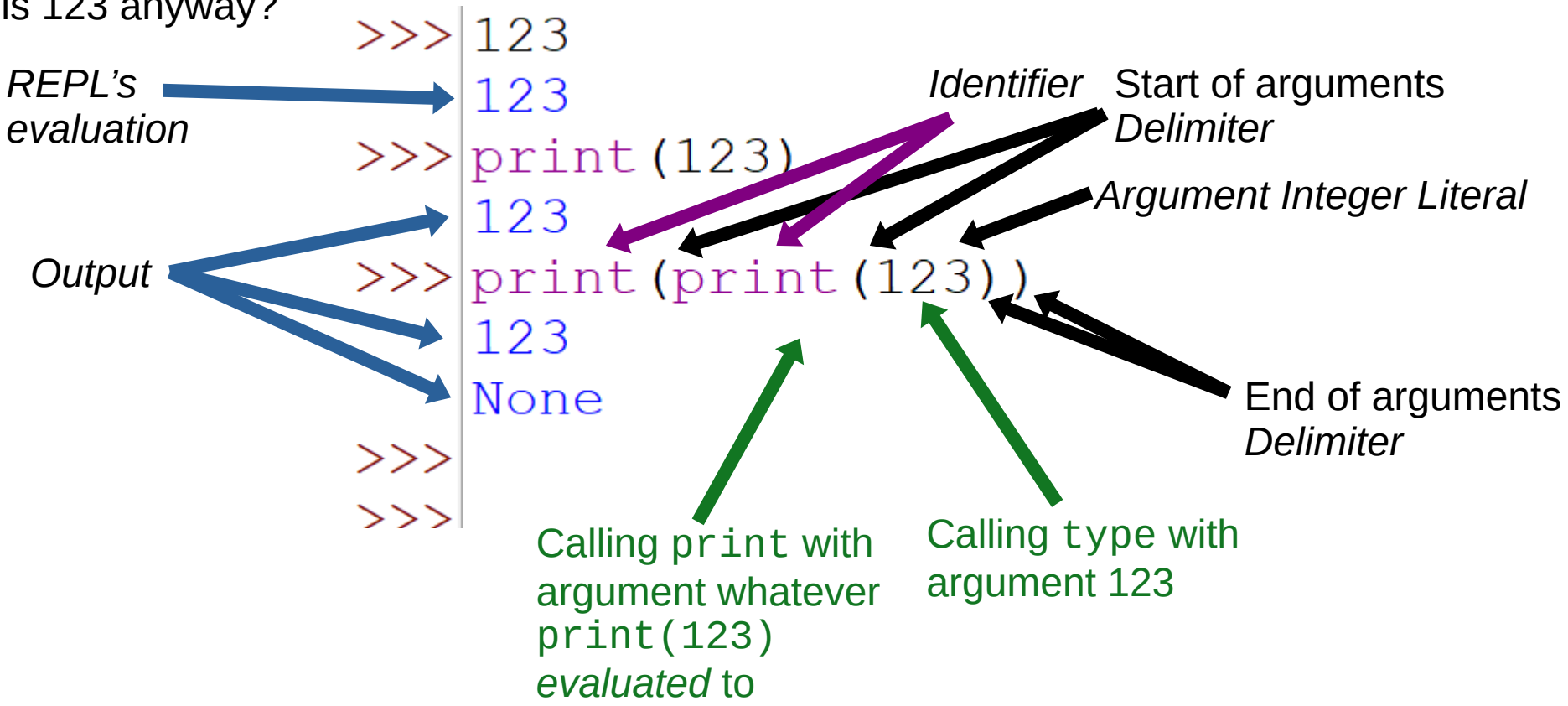
It's critical to remember that the evaluations (`123`, `int`, `None`)

Are separate from the written outputs (nothing, nothing, `<class 'int'>`)

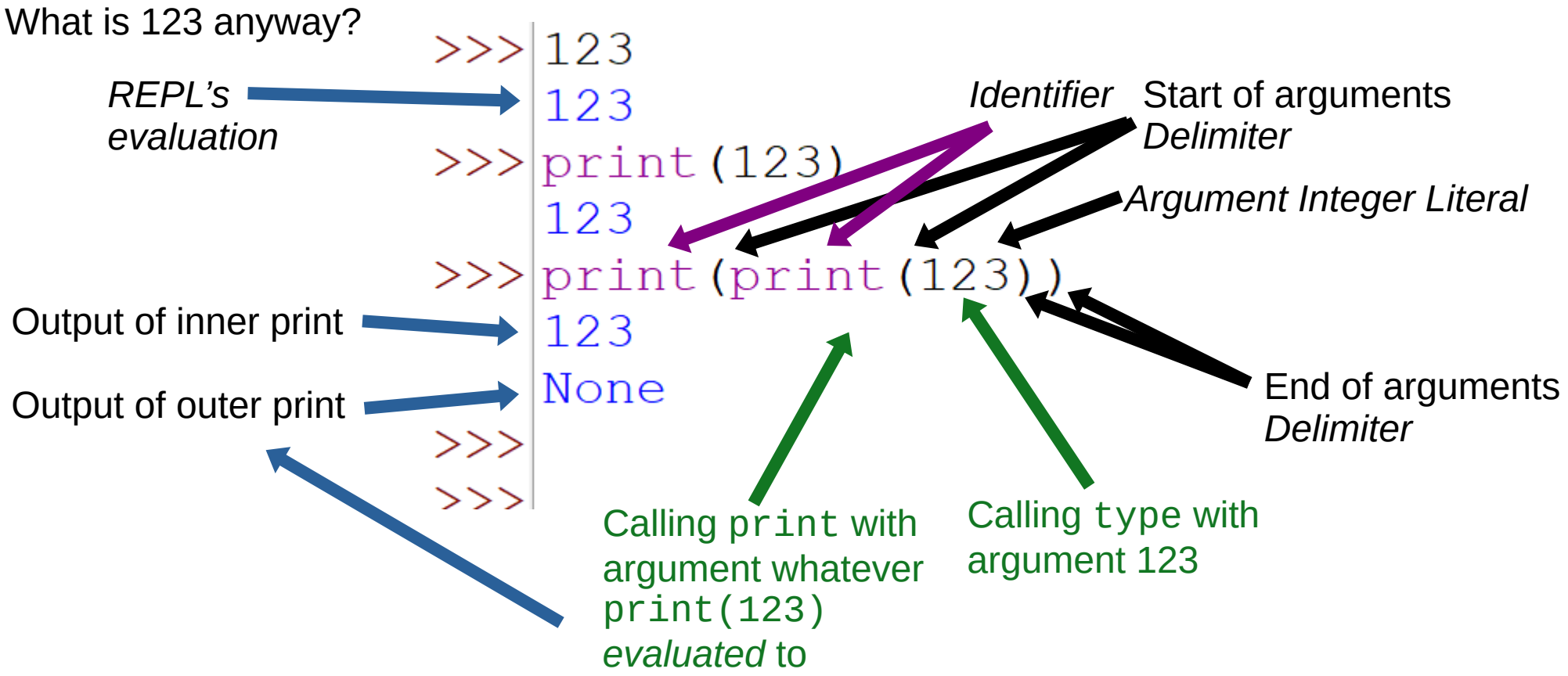
When evaluating the int literal `123` or the call to `type(…)`, we don't get a written output

Nesting Function Calls

What is 123 anyway?

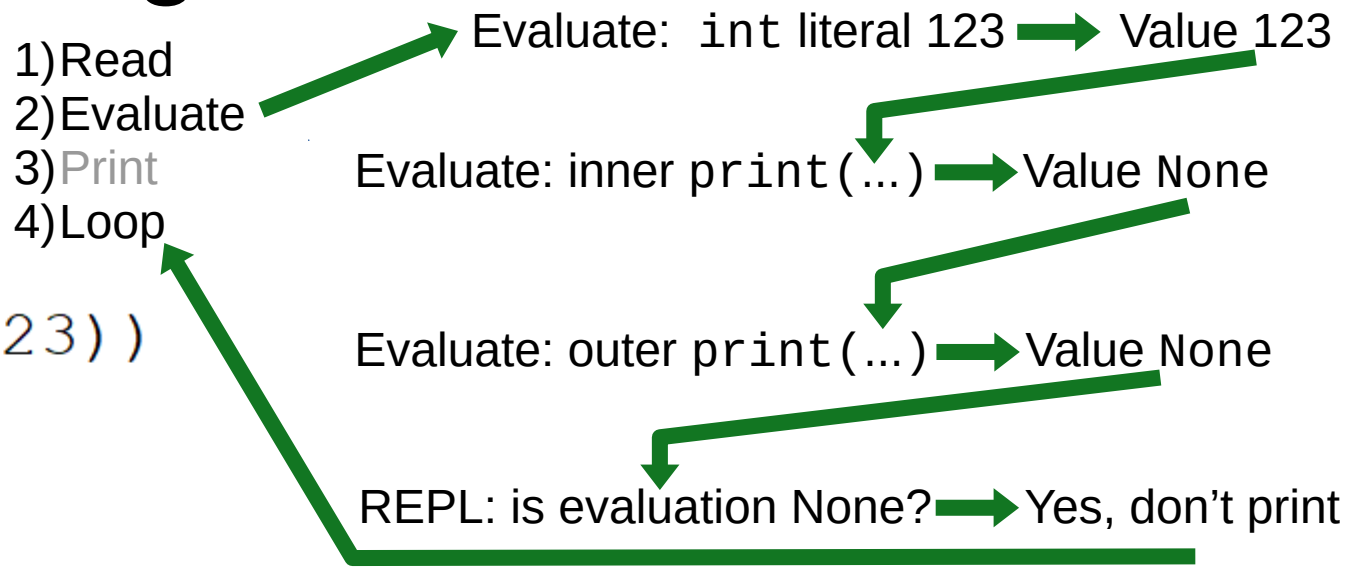


Nesting Function Calls

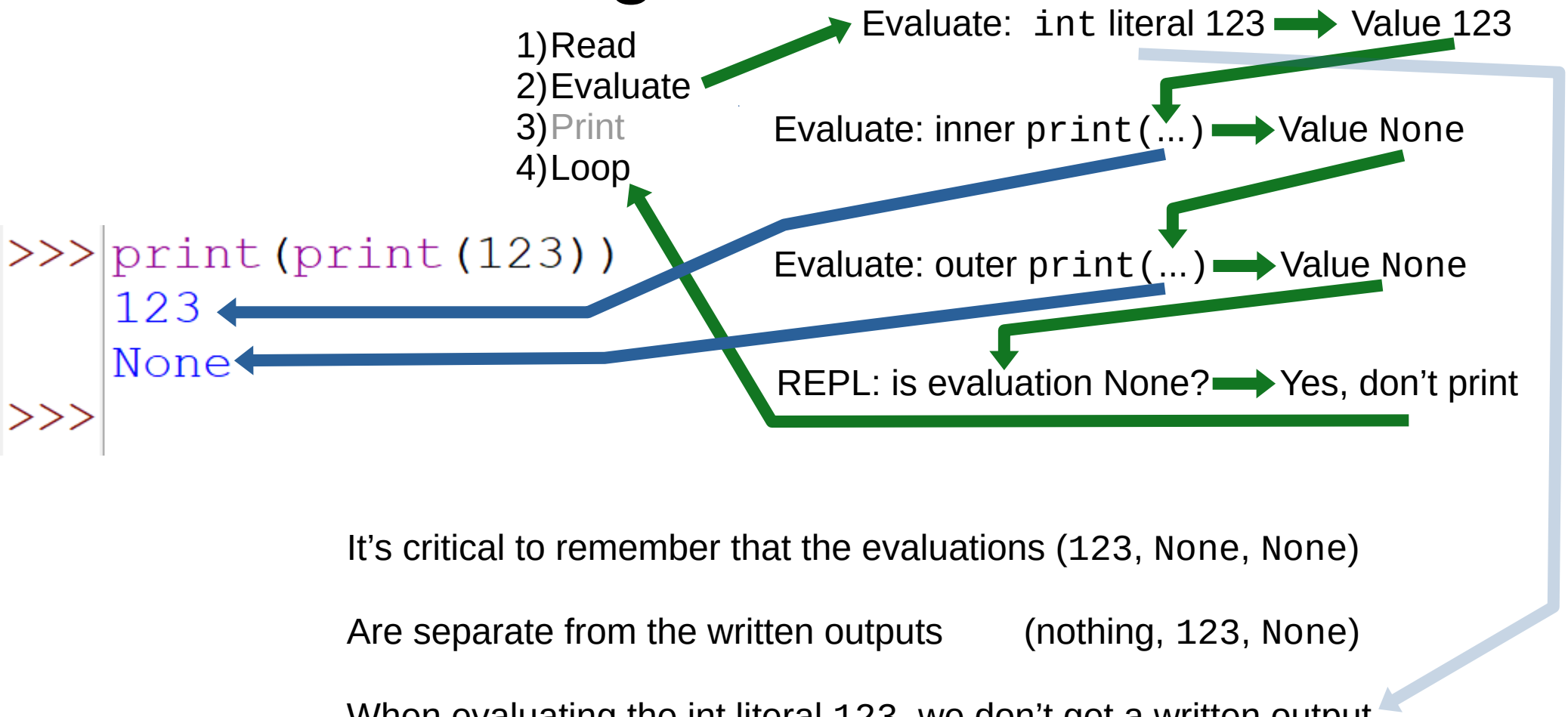


Nesting Function Calls

```
>>> print(print(123))
123
None
>>>
```



Nesting Function Calls

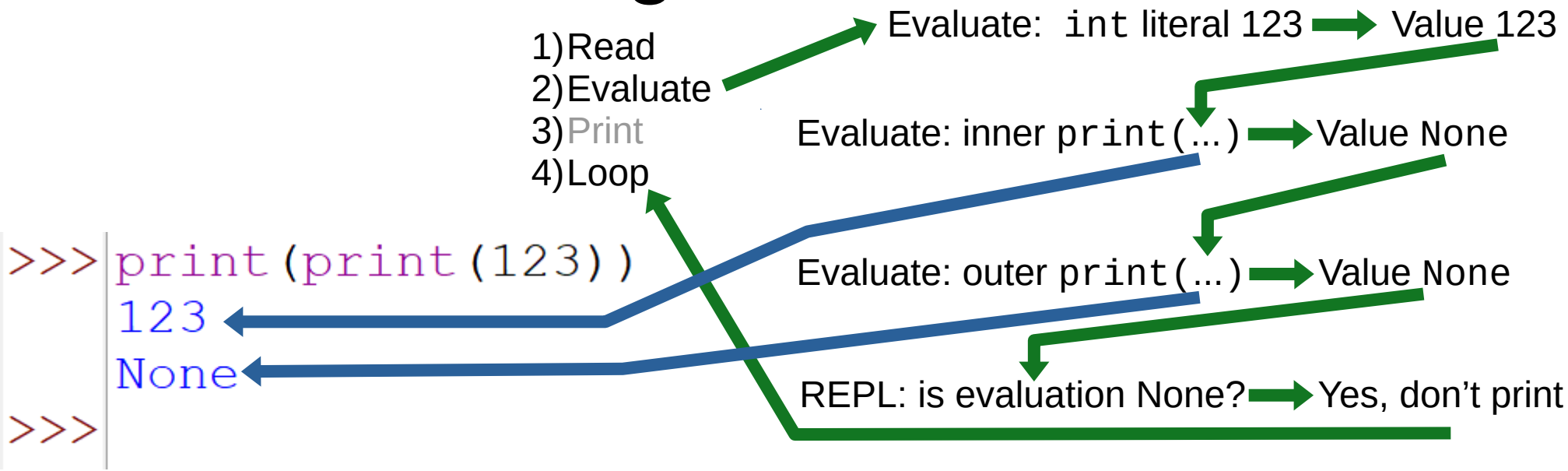


It's critical to remember that the evaluations (`123`, `None`, `None`)

Are separate from the written outputs (nothing, `123`, `None`)

When evaluating the int literal `123`, we don't get a written output

Nesting Function Calls



We are usually only concerned with the *evaluations* (123, None, None)

And we usually aren't thinking about the *written outputs* (nothing, 123, None)

Only a few functions like `print` actually *write output*