

Deck 1

Getting Started With Python

Dr. Hazel “twitch.tv/hazeldotzone” Campbell

Copyright 2026, Dr. Hazel Victoria Campbell, All Rights Reserved

Installing Python

The image shows a screenshot of the Python.org website. The browser address bar displays 'www.python.org' with a red annotation '1. python.org'. The website's navigation menu includes 'Python', 'PSF', 'Docs', and 'PyF'. The main content area features the Python logo and a 'Don' button. Below the logo, there are four menu items: 'About', 'Downloads', 'Documentation', and 'Community'. The 'Downloads' menu is open, showing options for 'All releases', 'Source code', and 'Windows'. A red annotation '2. downloads' is placed over the 'Downloads' menu item. The 'All releases' option is selected, leading to a page with a 'Python install manager' button. A red annotation '4. download it' is placed over this button. In the bottom left corner, there is a code snippet: '# Python 3: Lis', '>>> fruits = ['', '>>> print(len(fruits))', '>>> print(len(fruits))'. A red annotation '3. pick your OS' is placed over the code snippet.

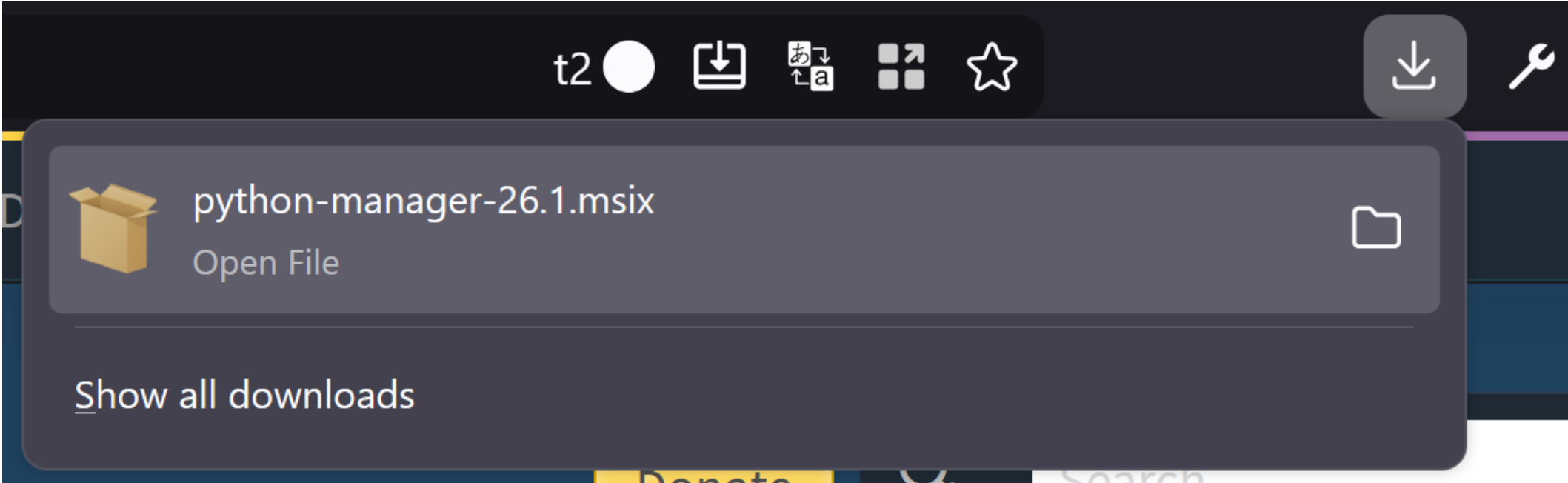
1. python.org

2. downloads

4. download it

3. pick your OS

Installing Python - Windows



5. click the download

Installing Python - Windows



Installing Python - Windows

```
C:\Program Files\WindowsAp x + v
Welcome to the Python installation manager configuration helper.
*****
The global shortcuts directory is not configured.
Configuring this enables commands like python3.14.exe to run from your
terminal, but is not needed for the python or py commands (for example, py
-V:3.14).
We can add the directory (C:\Users\glidi.ROXANNE\AppData\Local\Python\bin) to
PATH now, but you will need to restart your terminal to use it. The entry will
be removed if you run py uninstall --purge, or else you can remove it manually
when uninstalling Python.
Add commands directory to your PATH now? [y/N] |
```

6. Press Y,
7. Enter

Installing Python - Windows

```
C:\Program Files\WindowsAp x + v
Welcome to the Python installation manager configuration helper.
*****
The global shortcuts directory is not configured.

Configuring this enables commands like python3.14.exe to run from your
terminal, but is not needed for the python or py commands (for example, py
-V:3.14).

We can add the directory (C:\Users\glidi.ROXANNE\AppData\Local\Python\bin) to
PATH now, but you will need to restart your terminal to use it. The entry will
be removed if you run py uninstall --purge, or else you can remove it manually
when uninstalling Python.
Add commands directory to your PATH now? [y/N] y
PATH has been updated, and will take effect after opening a new terminal.
*****
You do not have the latest Python runtime.

Install the current latest version of CPython? If not, you can use 'py install
default' later to install.

Install CPython now? [Y/n] |
```

8. Press Y,
9. Enter

View online help?
Up to you.

Launching - Windows

 idle (Python 3.14)

Search for idle in
Start Menu



All

Apps

Documents

Settings

It should should
show up

Best match



IDLE (Python 3.14)

App

The image shows a screenshot of the IDLE Shell 3.14.4 window. The title bar reads "IDLE Shell 3.14.4". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area contains the following text:
Python 3.14.4 (tags/v3.14.4:23116f9, Apr 7 2026, 14:10:54) [MSC v.1944 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>> 1+1
The status bar at the bottom right indicates "Ln: 3 Col: 0".

Using Idle

- Type
1+1
- Hit enter

```
Python 3.14.4 (tags/v3.14.4:23116f9, Apr 7 2026, 14:10:54) [MSC v.1944 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>> 1+1
2
>>>
```

Your Input

Python's
evaluation

Python "REPL"

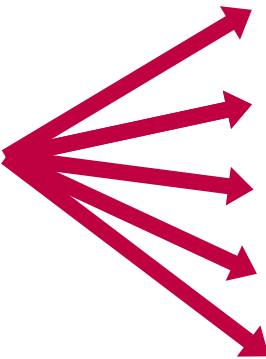
What python will do when it shows >>>

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

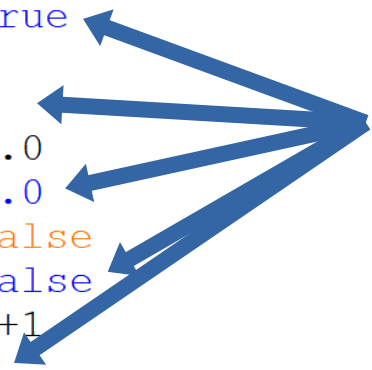
"REPL"

AMD64)] on win32
Enter "help" below or click "Help" above for more information

Your
Input



```
>>> True
True
>>> 1
1
>>> 1.0
1.0
>>> False
False
>>> 1+1
2
>>>
```



Python "REPL"

Python's
evaluation

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”

The traceback

```
>>> 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 error

What python will do when it shows >>>

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

The traceback will tell you where the problem was

The error will tell you what the problem was

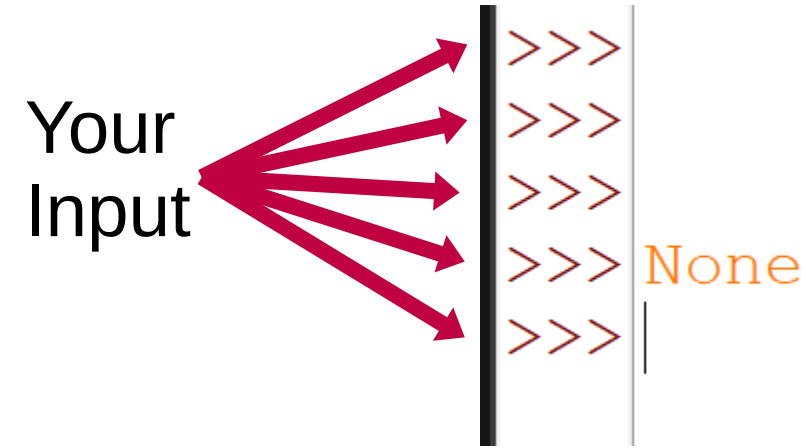
“REPL”

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”



When Python REPL doesn't **P**rint:

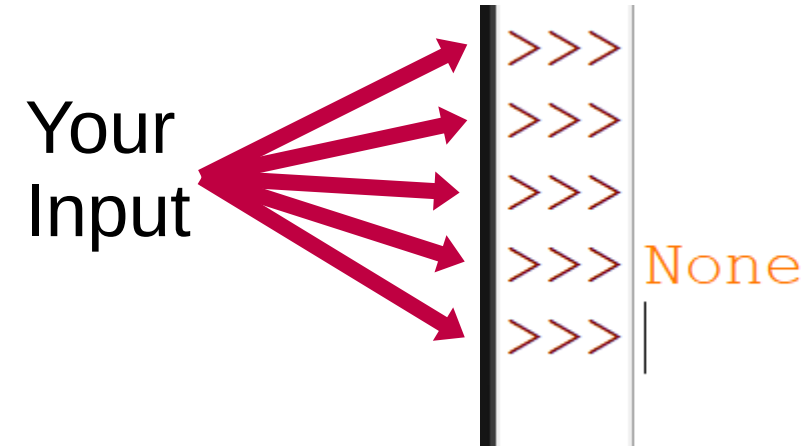
- Error
- If the input *evaluates* to None

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”



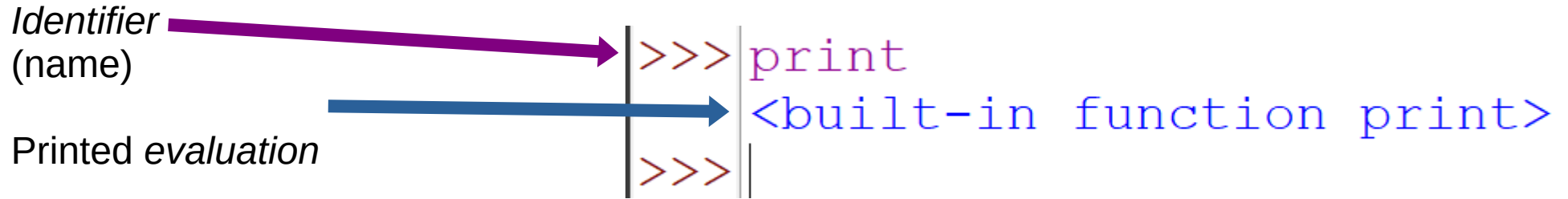
When Python REPL doesn't **Print**:

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

Functions

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

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

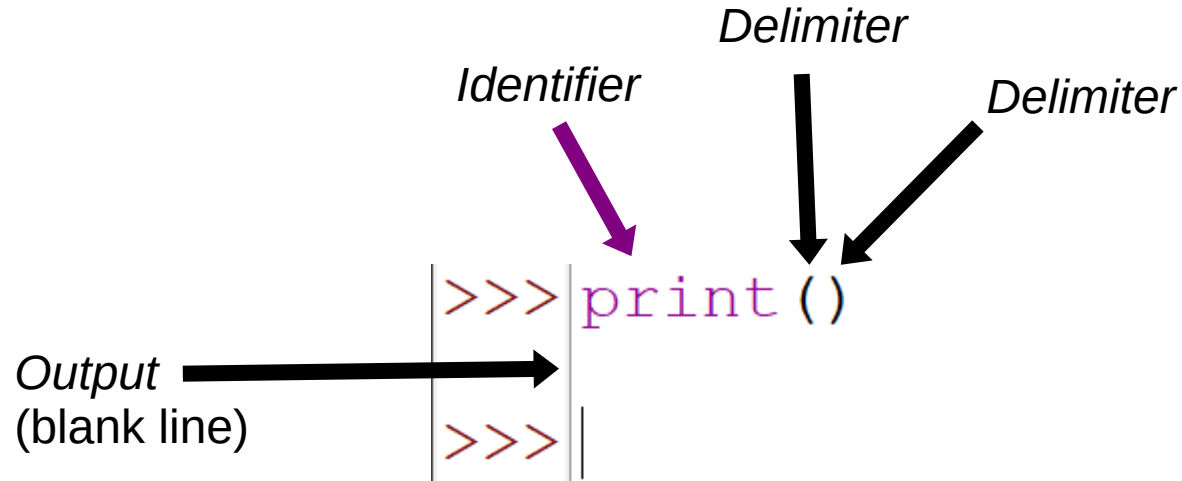


Erm... I can't really show you... but I can tell you some things about the evaluation

- * 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.



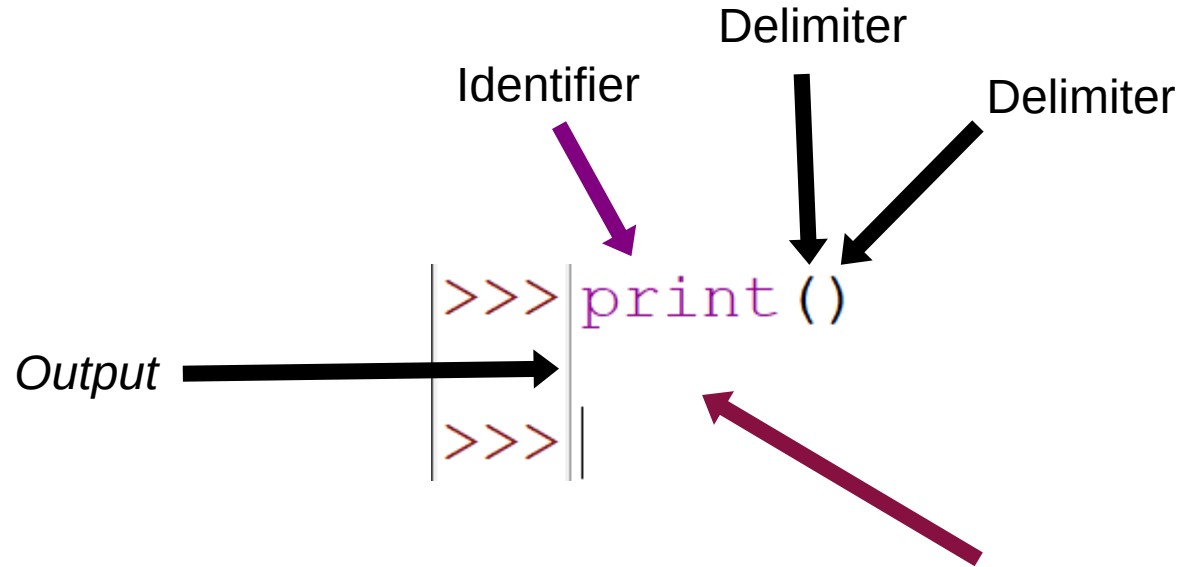
The identifier of a function followed by parenthesis -> Call the function

- *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.

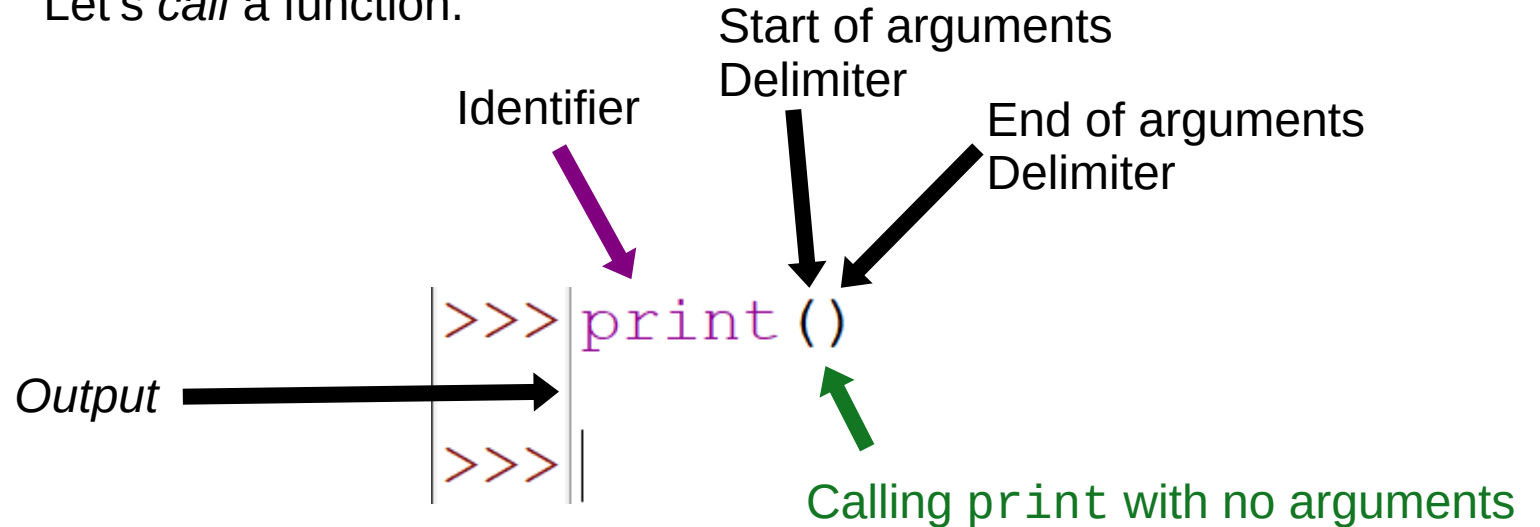


But wait... where's the printed evaluation??? Shouldn't it be right **here**?

- 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.



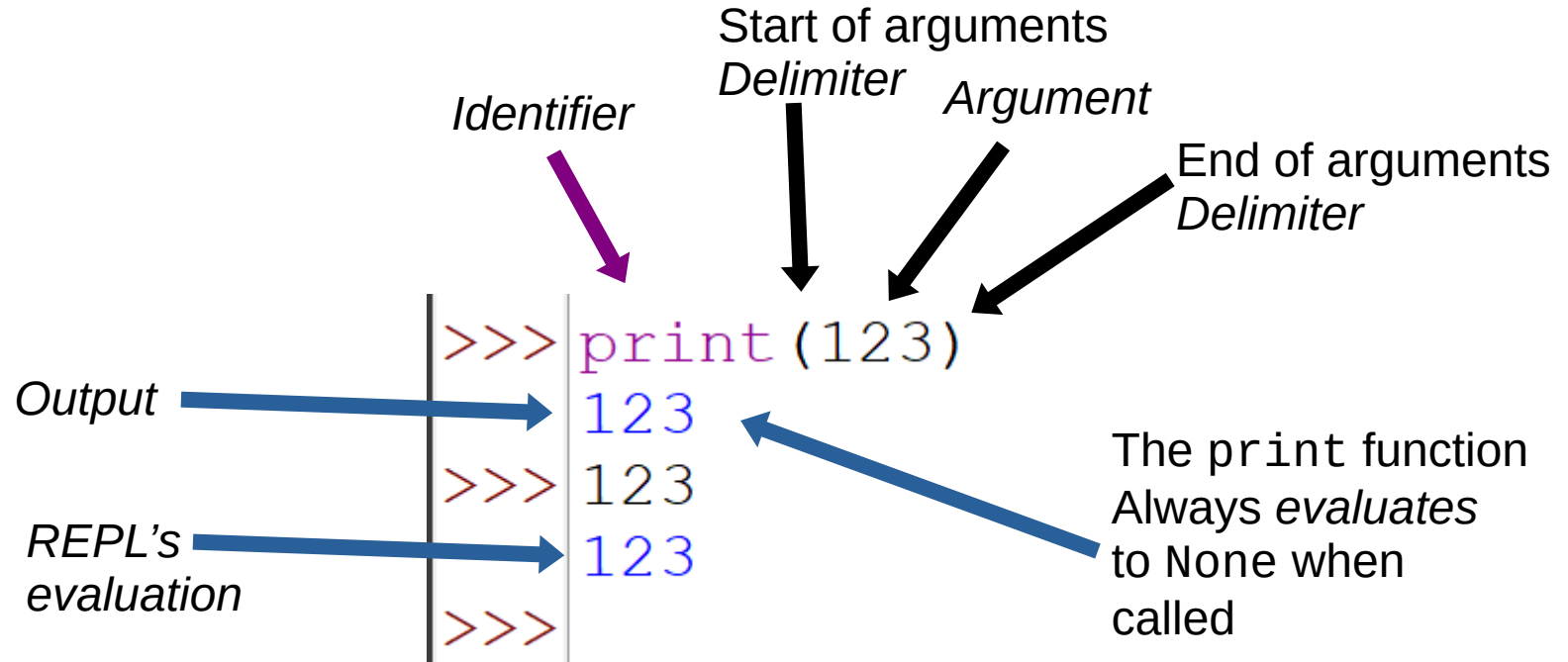
When an *identifier* is followed by the *argument delimiters* () that means we are *calling* it

- 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.



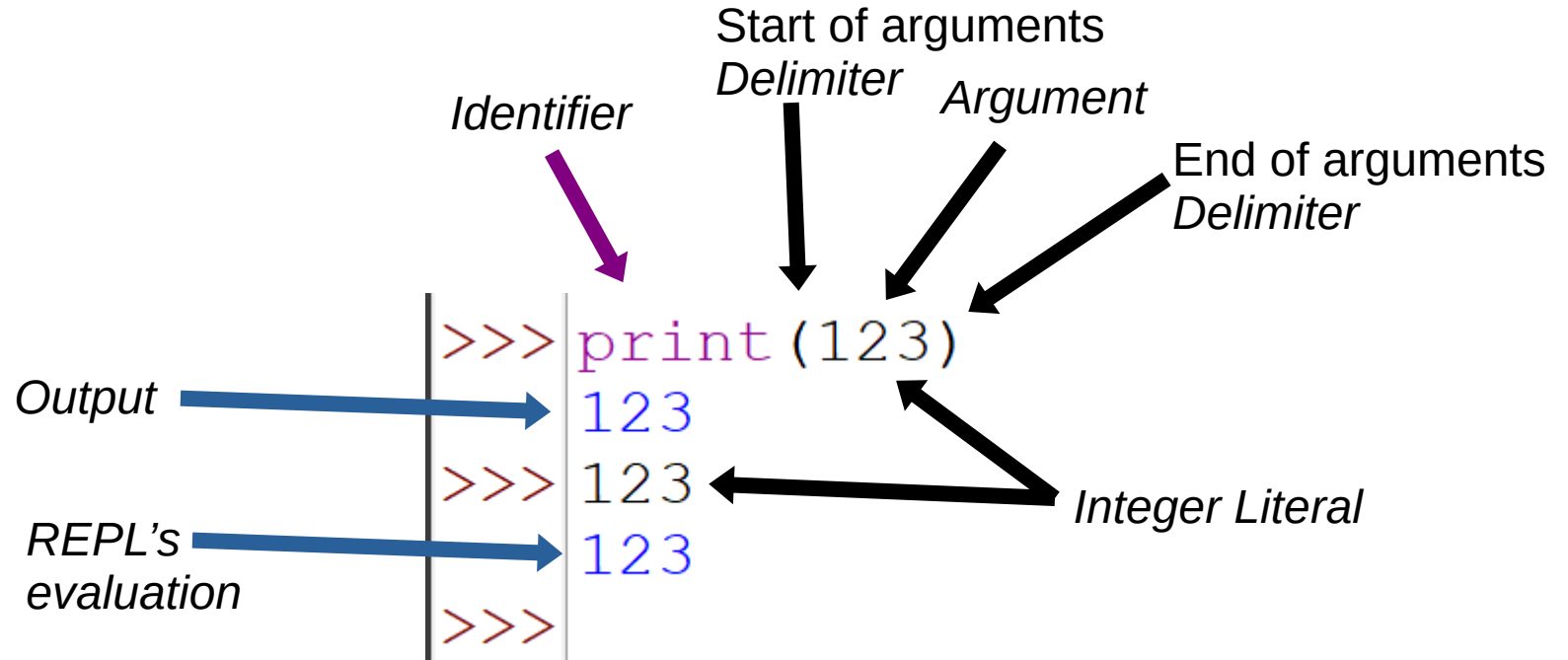
How can we tell the difference between REPL's evaluation and a Function's output

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

We can be sure
this is not what it
evaluated to

Literals

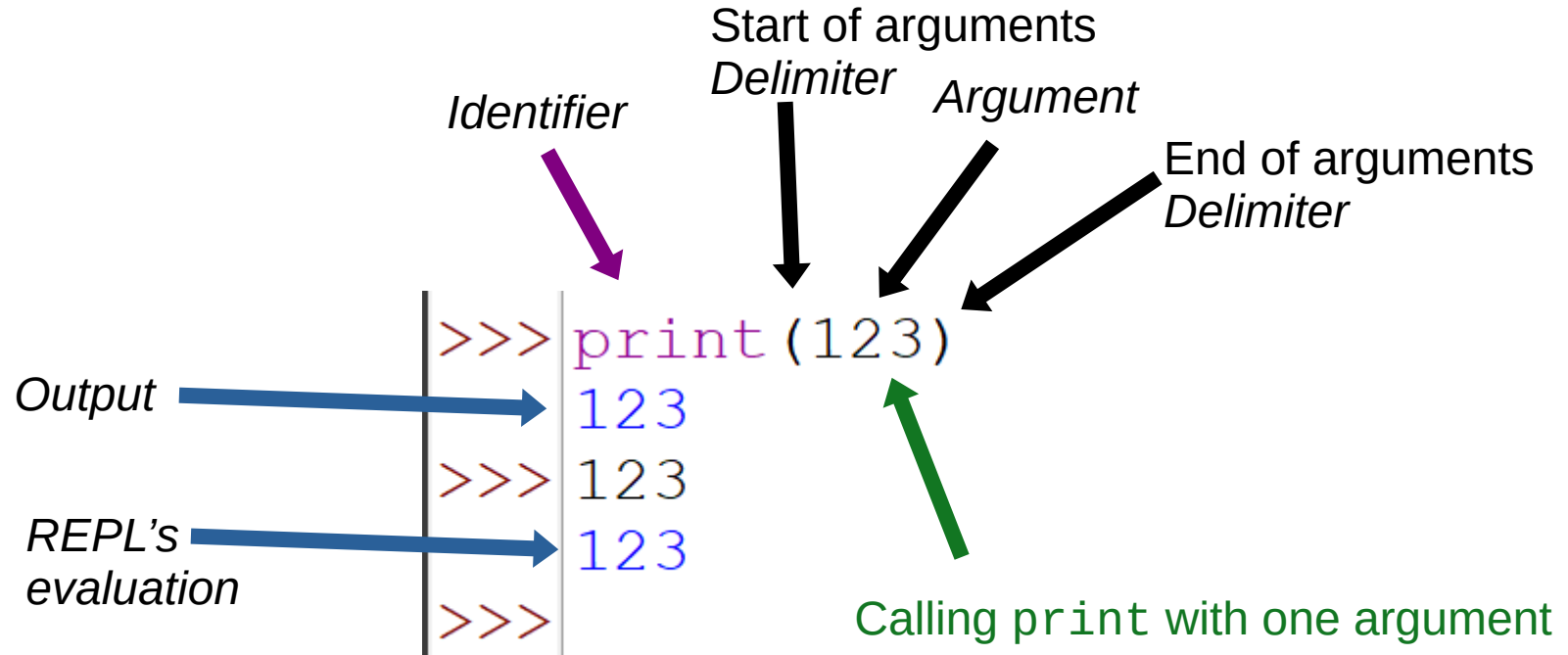
What is 123 anyway?



- 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

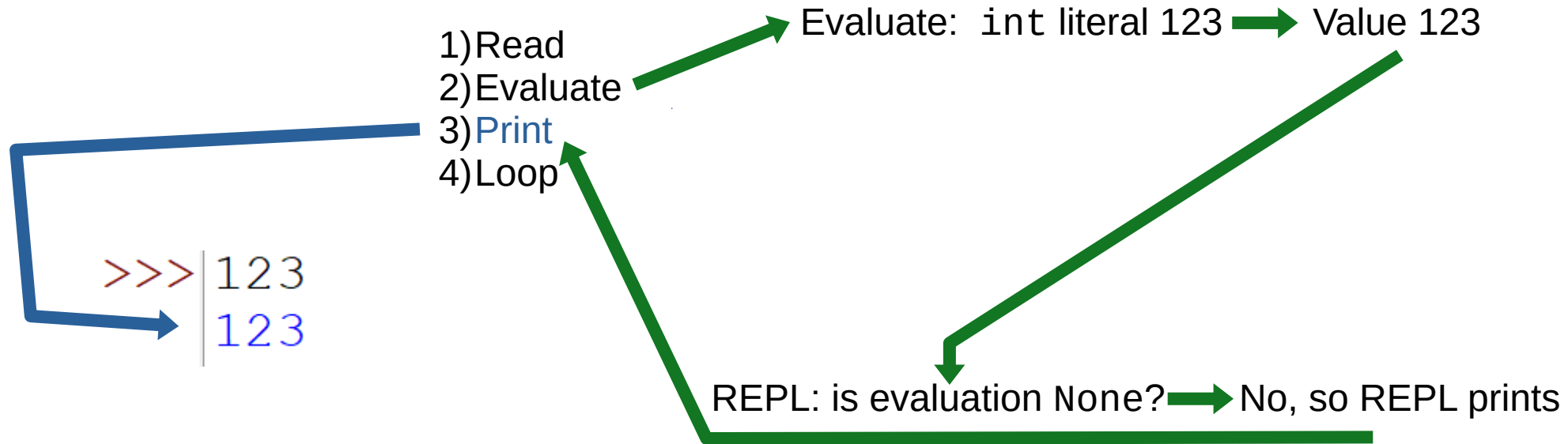
What is 123 anyway?



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!

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

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

Evaluate: int literal 123 → Value 123

Evaluate: outer print(...) → Value None

REPL: is evaluation None? → Don't REPL print

Function Calls

Evaluate the inside first like in math!

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

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

Evaluate: int literal 123 → Value 123

Evaluate: outer print(...) → Value None

REPL: is evaluation None? → Don't REPL print

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'>
```

Evaluate: int literal 123 → Value 123

Evaluate: outer type(...) → Value int

REPL: is evaluation None? → No, REPL prints

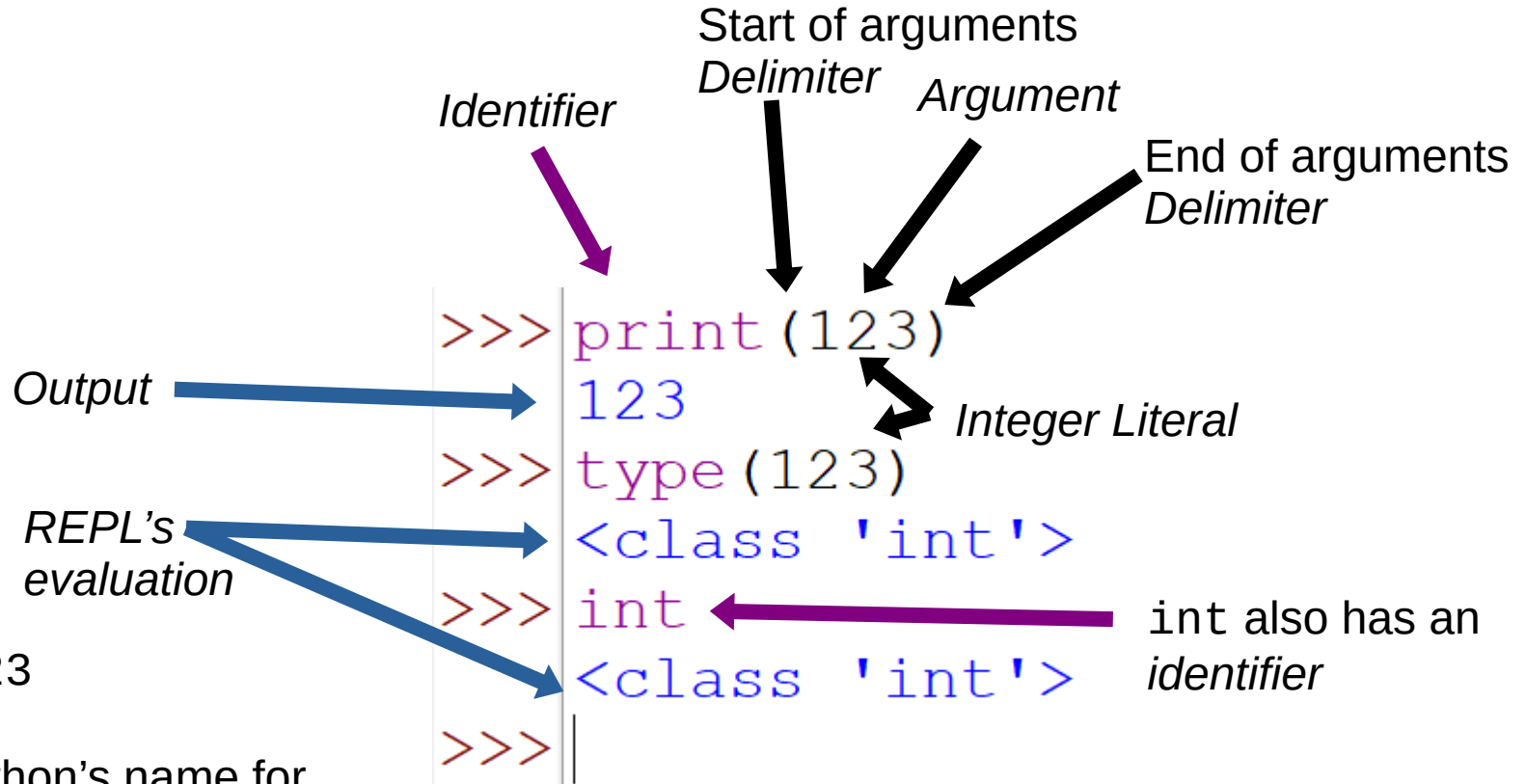
It's critical to remember that the evaluations (123, int)

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

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

Literals

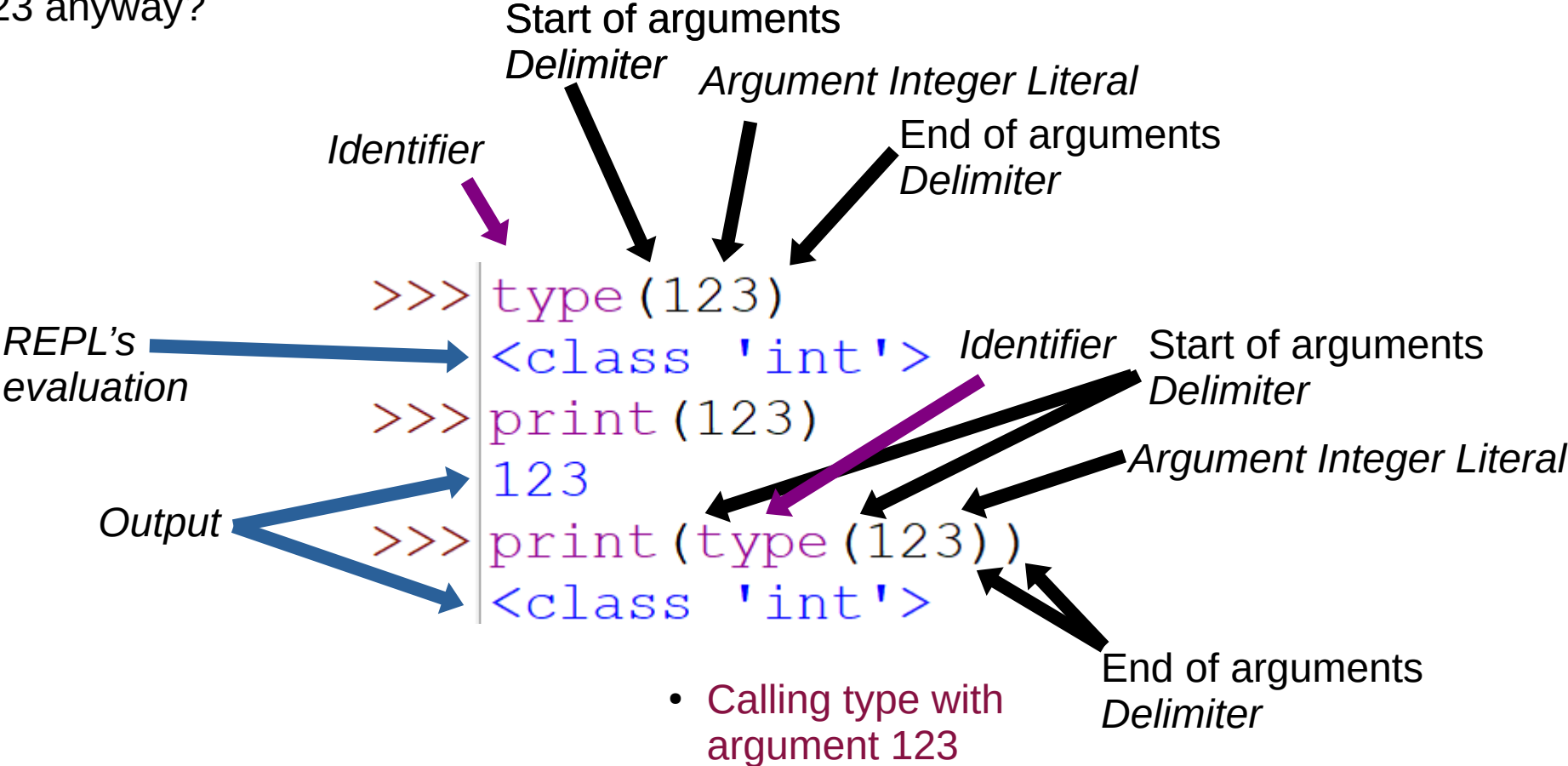
What is 123 anyway?



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

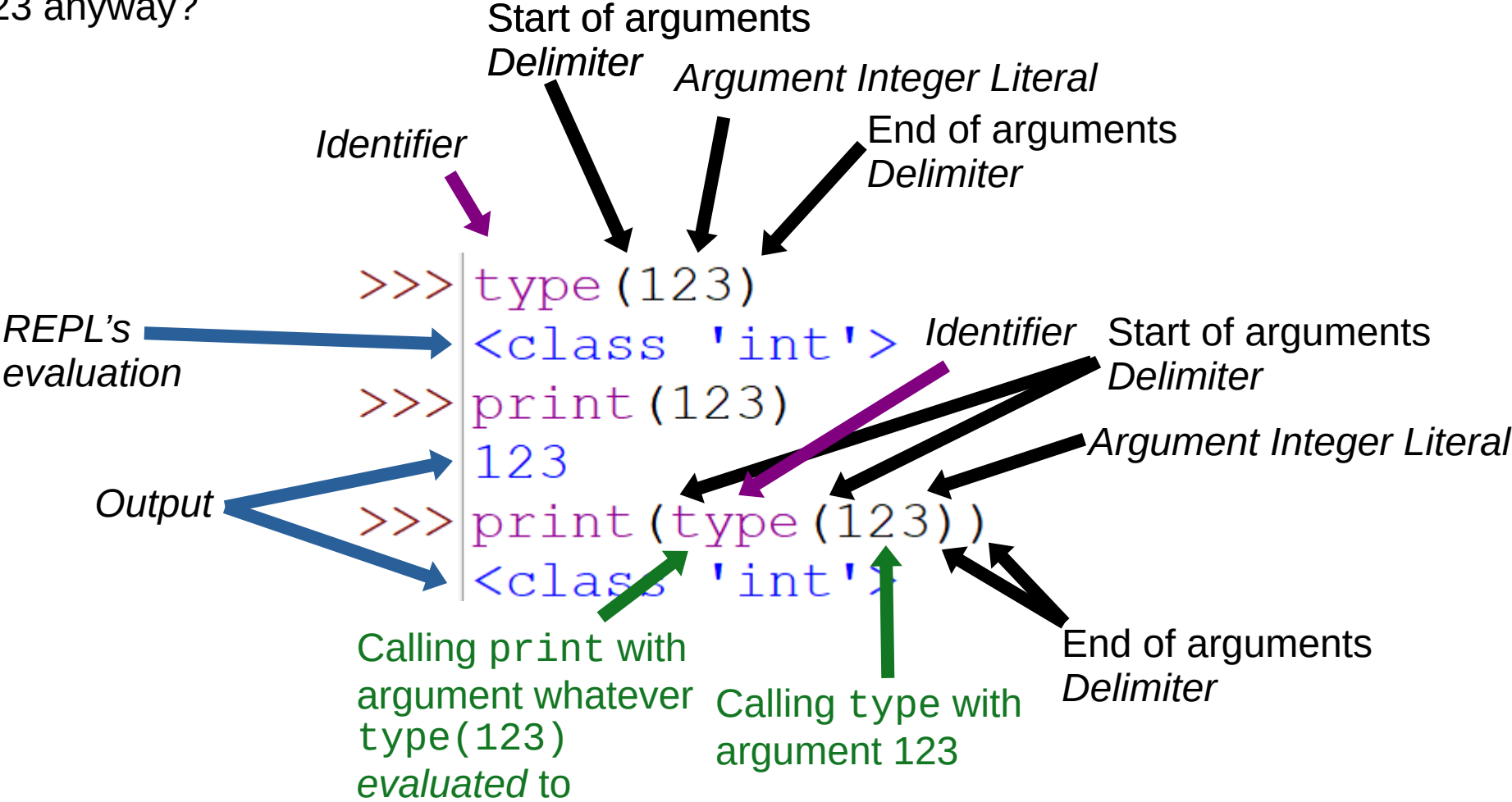
Nesting Function Calls

What is 123 anyway?



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'>  
>>>
```

Evaluate: int literal 123 → Value 123

Evaluate: inner type(...) → Value int

Evaluate: outer print(...) → Value None

REPL: is evaluation None? → Yes, don't print

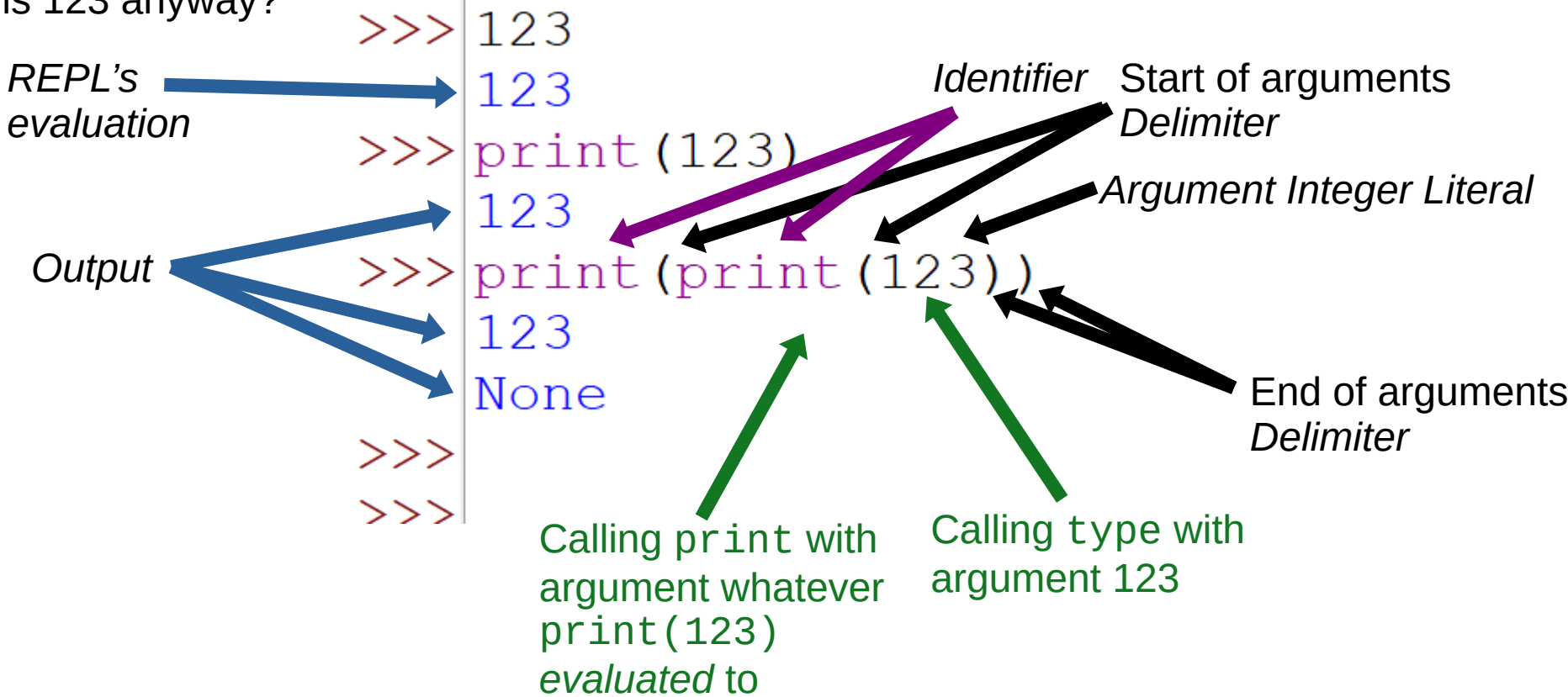
It's critical to remember that the evaluations (123, int, None)

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

What is 123 anyway?

>>>

123

REPL's
evaluation



123

>>>

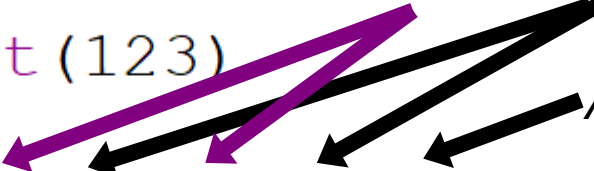
print (123)

123

Identifier

Start of arguments

Delimiter



Argument Integer Literal

>>>

print (print (123))

Output of inner print



123

Output of outer print



None

End of arguments

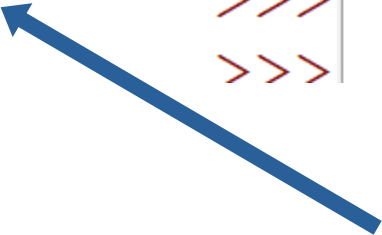
Delimiter

>>>

>>>

Calling print with
argument whatever
print(123)
evaluated to

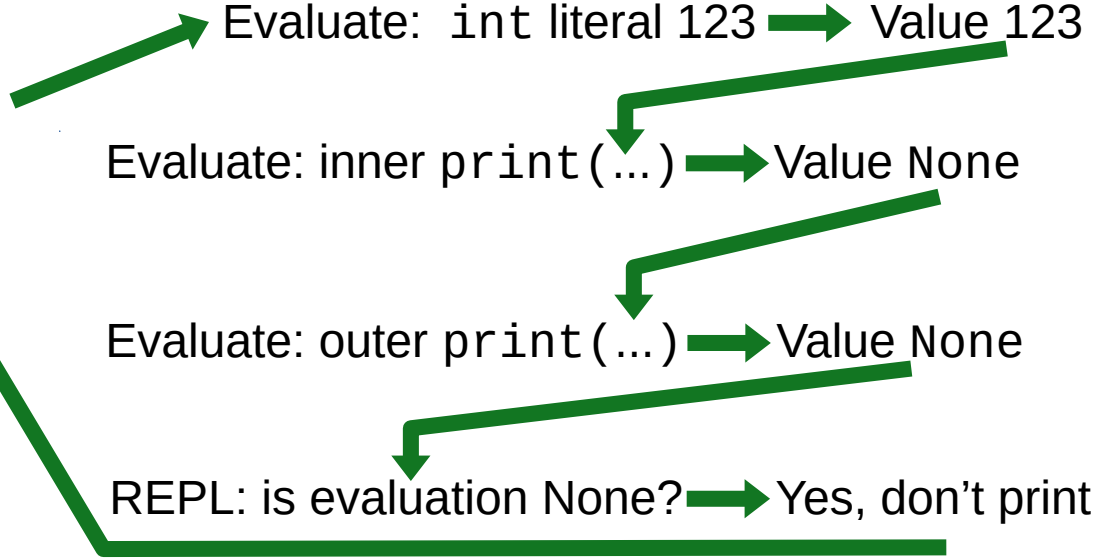
Calling type with
argument 123



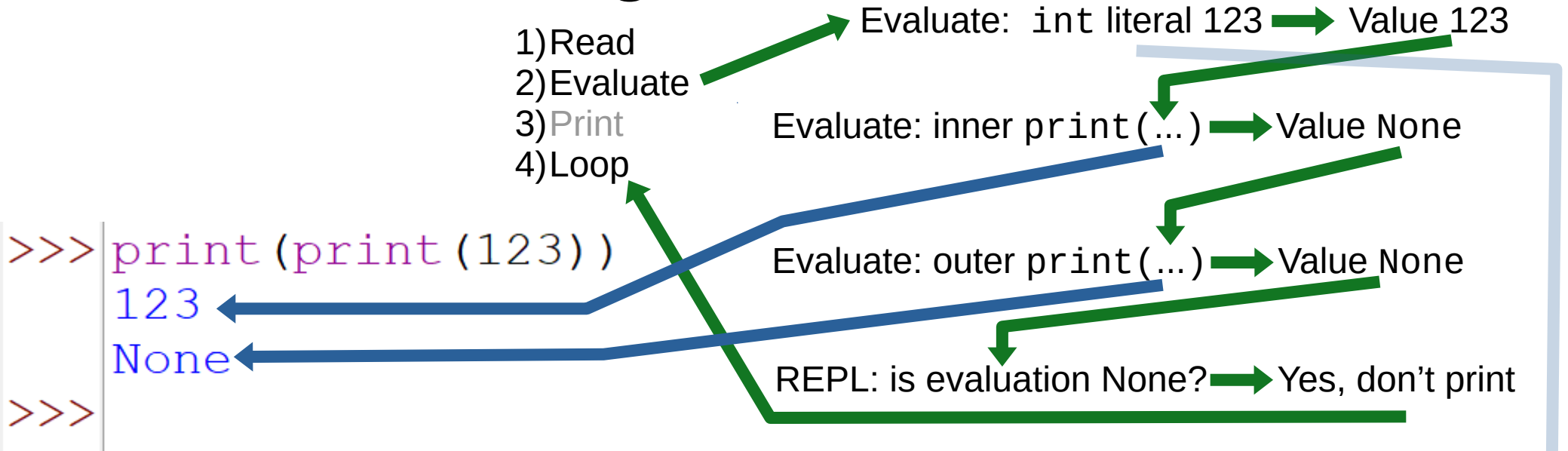
Nesting Function Calls

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

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



Nesting Function Calls

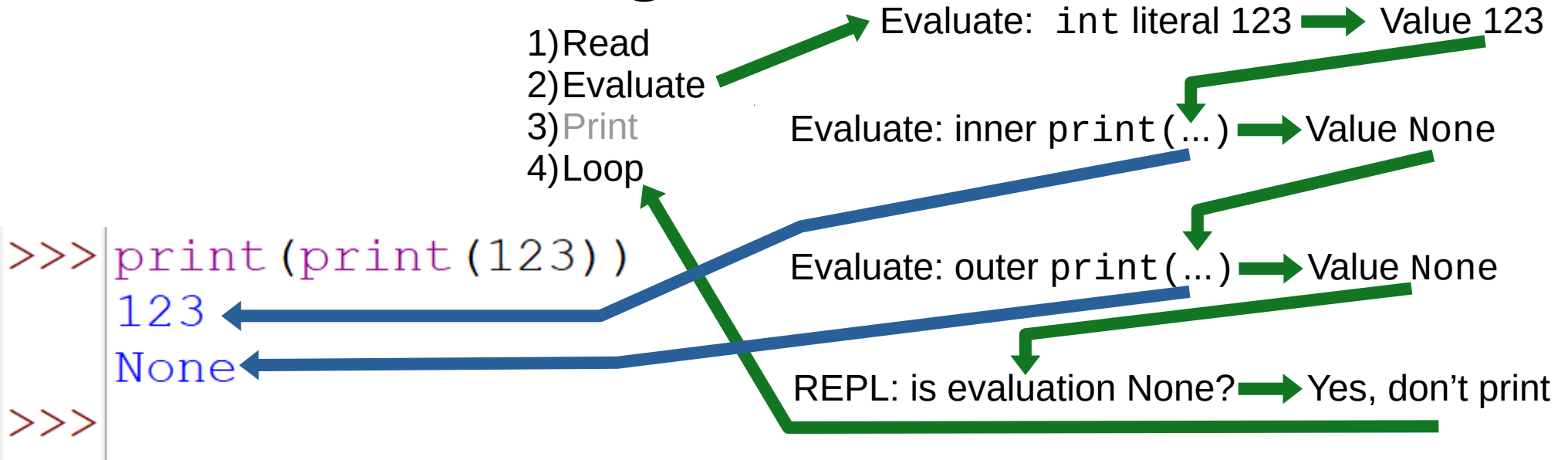


It's critical to remember that the evaluations (123, None, None)

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

