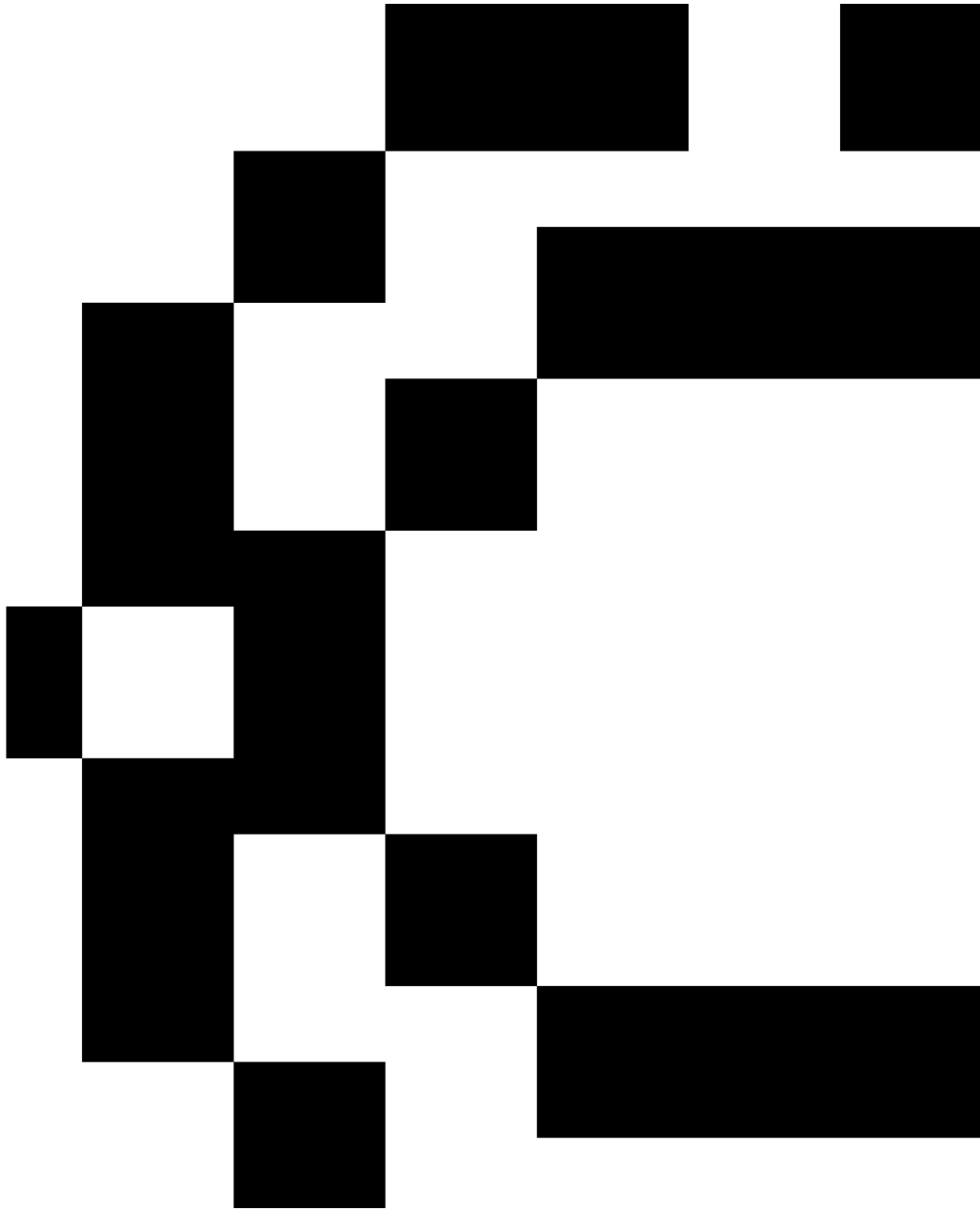
NOVA IMS INFORMATION MANAGEMENT SCHOOL

Programming for Data Science

Lecture 3

Flávio L. Pinheiro

fpinheiro@novaims.unl.pt | www.flaviolpp.com
www.linkedin.com/in/flaviolpp | X @flavio_lpp



Quizzes

https://www.socrative.com

Login as a student

Room Name: **PDS2025**

Student ID: **Student Number**

or

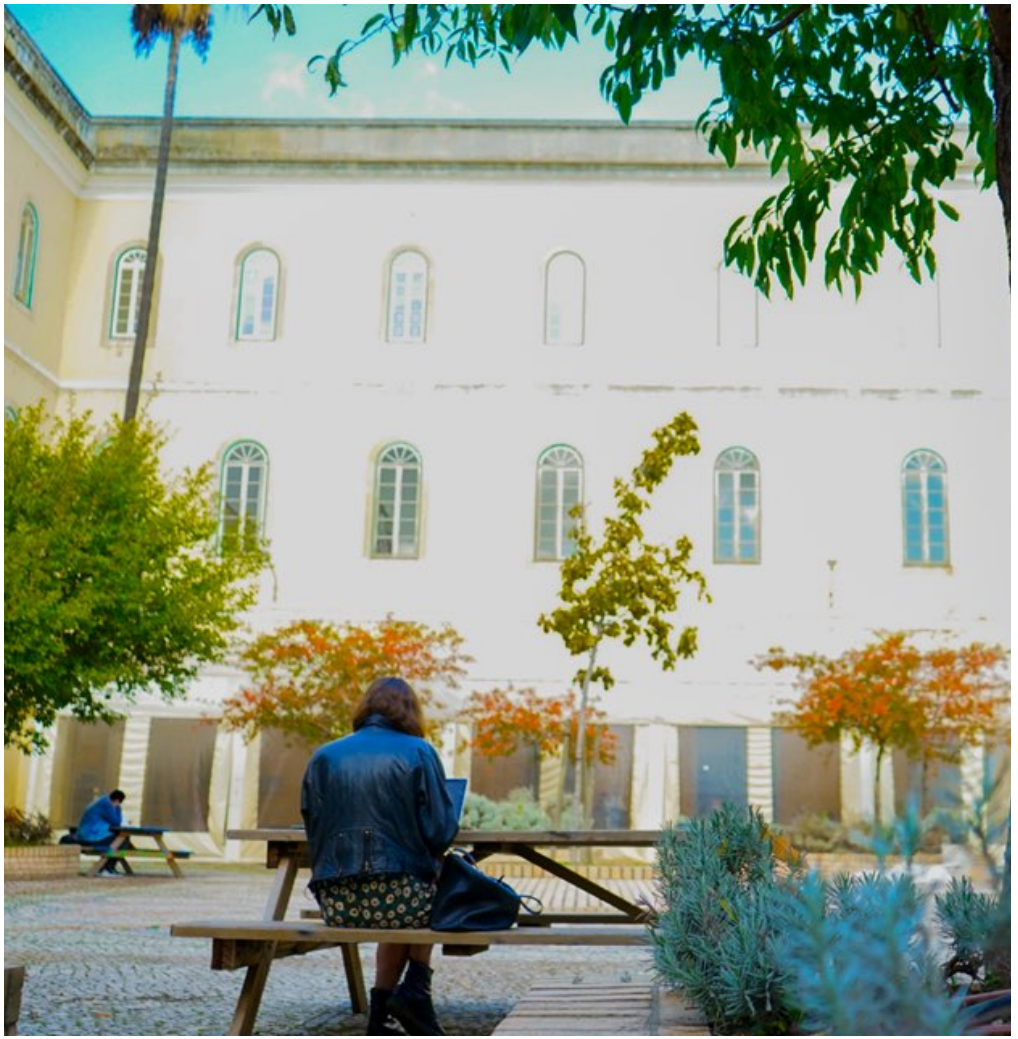
https://api.socrative.com/rc/5NpKRX

Student ID: **Student Number**

Please Join Now!

Lecture 3

- Functions
- Import Modules and Python STL
- NameSpace



What is this code doing?

```
>>> a = 5
>>> b = 1
>>> while a > 0:
...     b *= a
...     a-= 1
>>> print(b)
```

computes the factorial of 5

Let’s decode the code

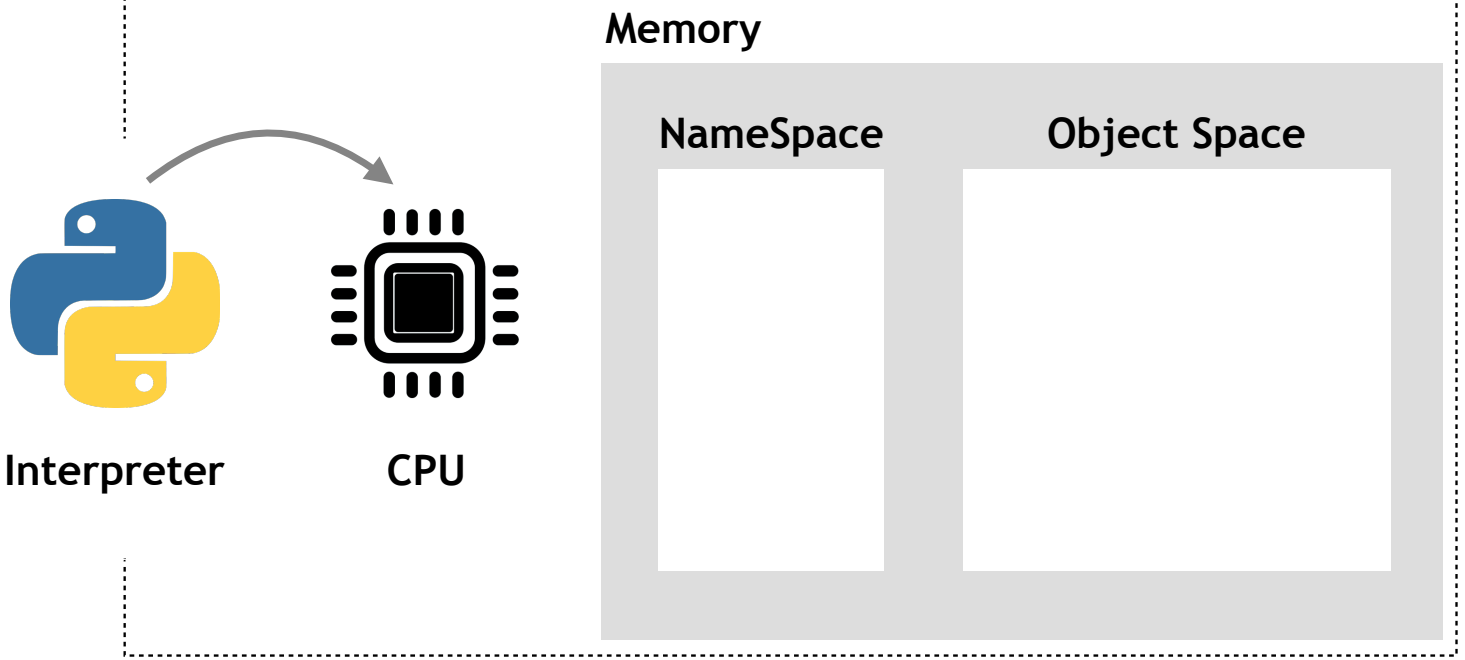
Input Script

```
>>> a = 5
>>> b = 1
>>> while a > 0:
...     b *= a
...     a-= 1
>>> print(b)
```

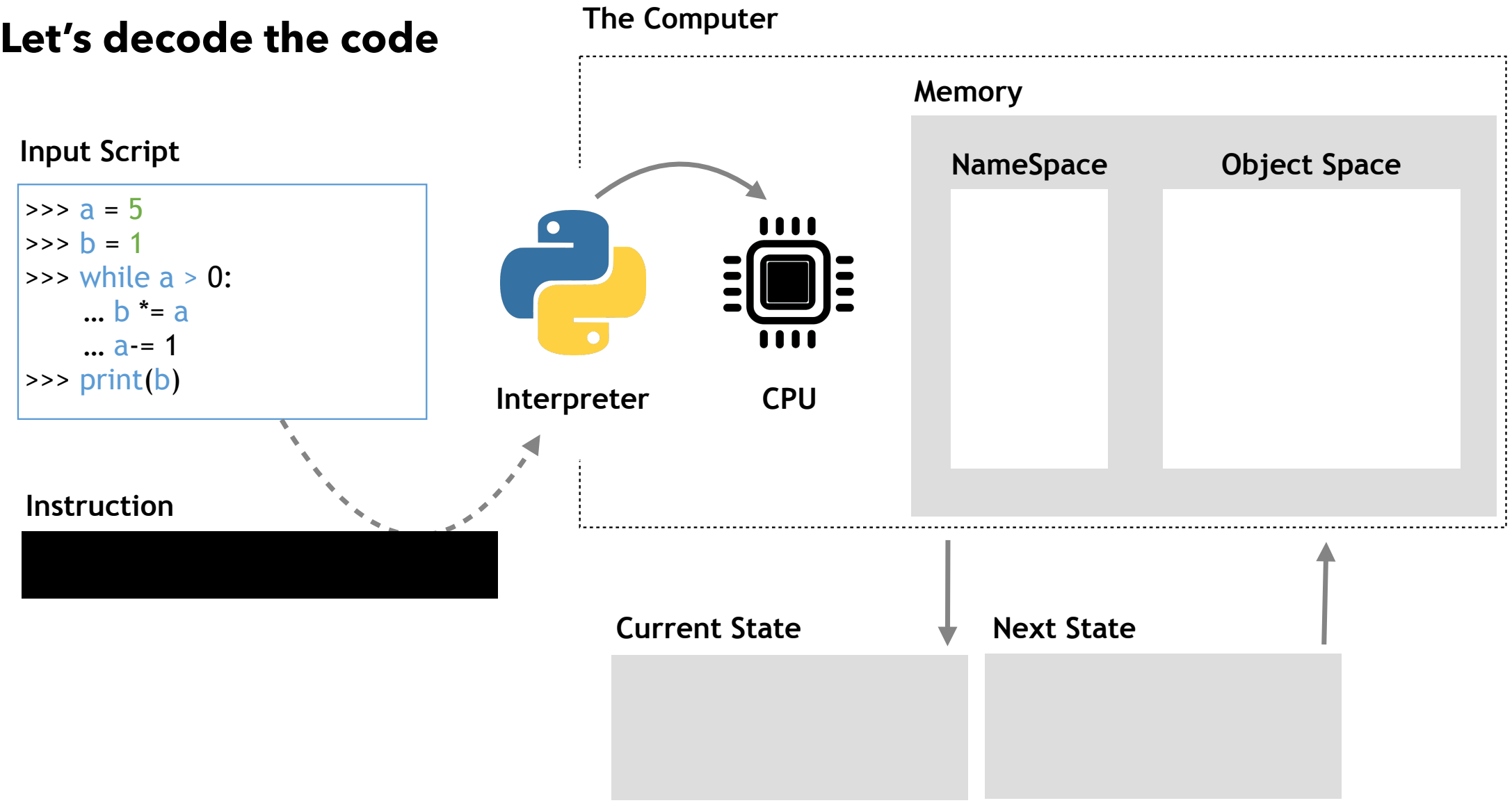
Let's decode the code

Input Script

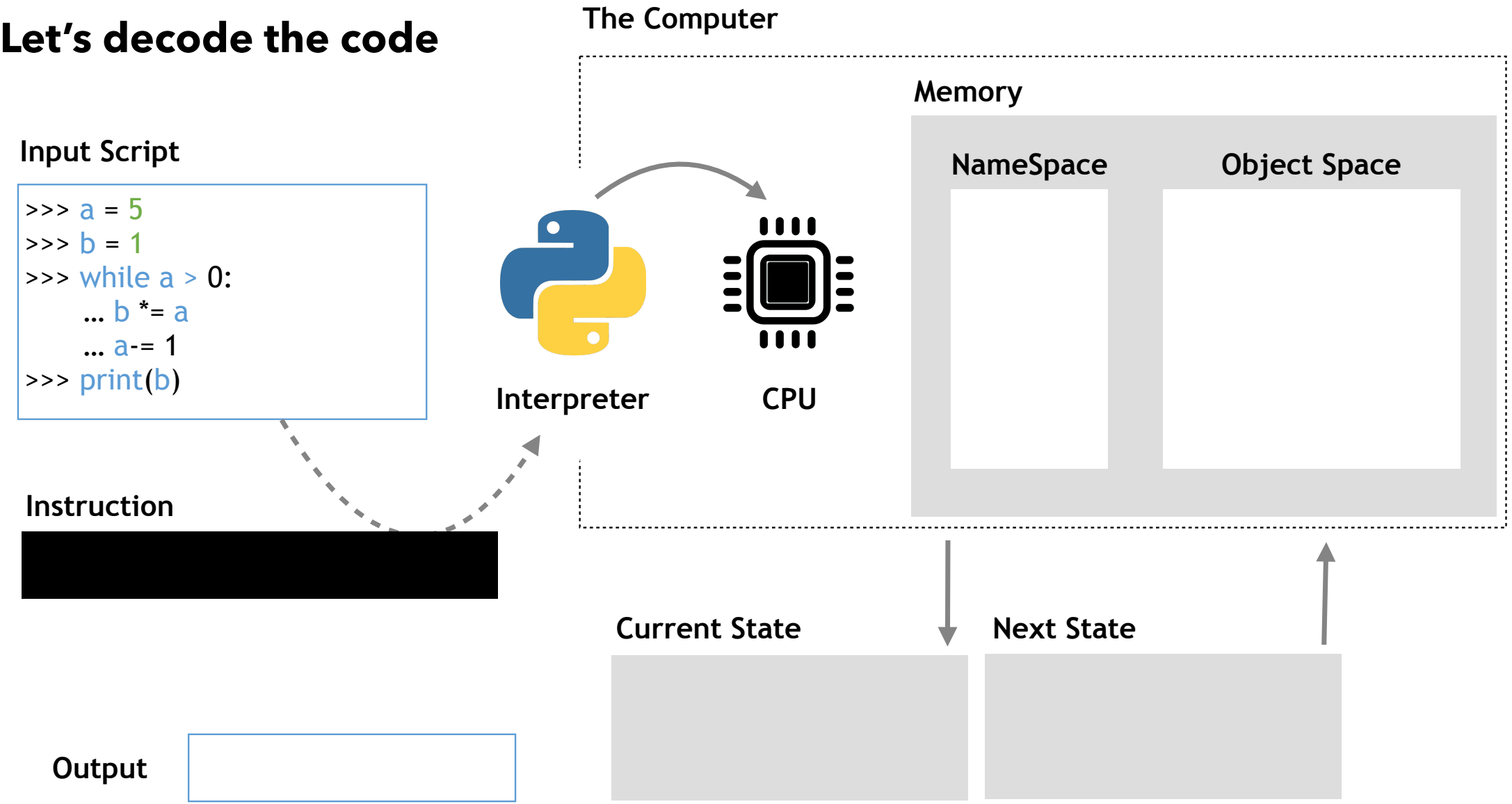
```
>>> a = 5
>>> b = 1
>>> while a > 0:
...     b *= a
...     a-= 1
>>> print(b)
```



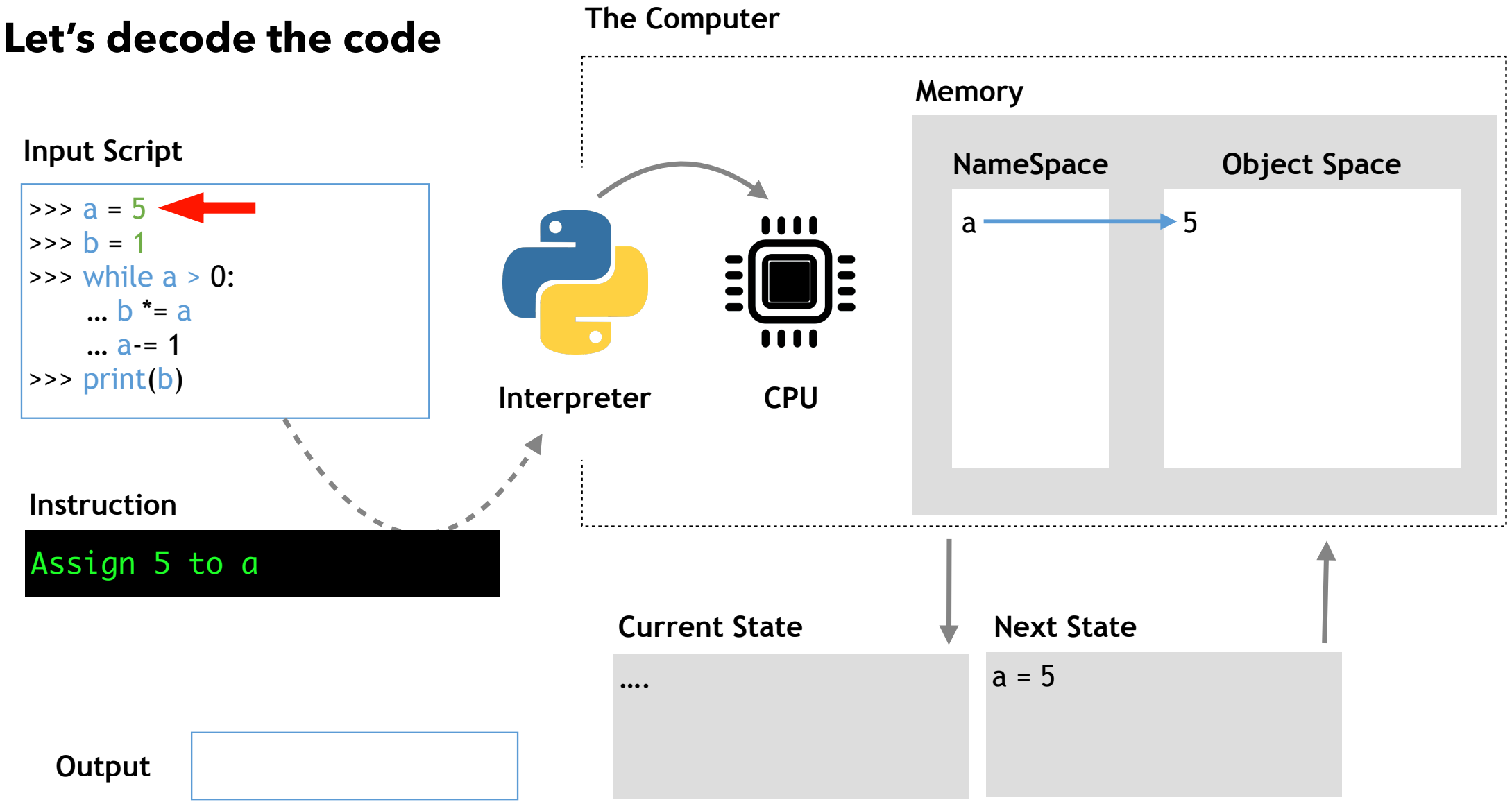
Let's decode the code



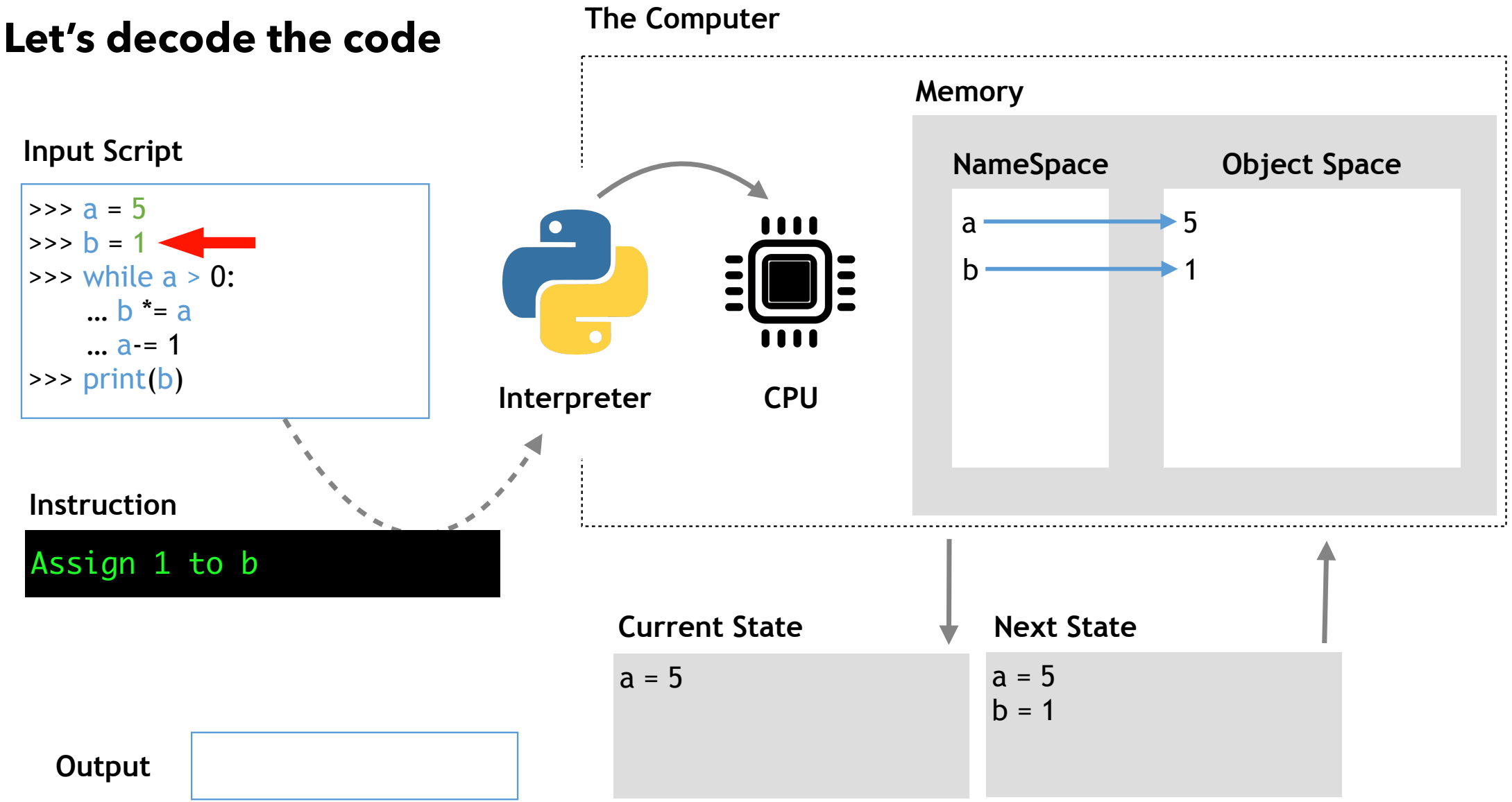
Let's decode the code



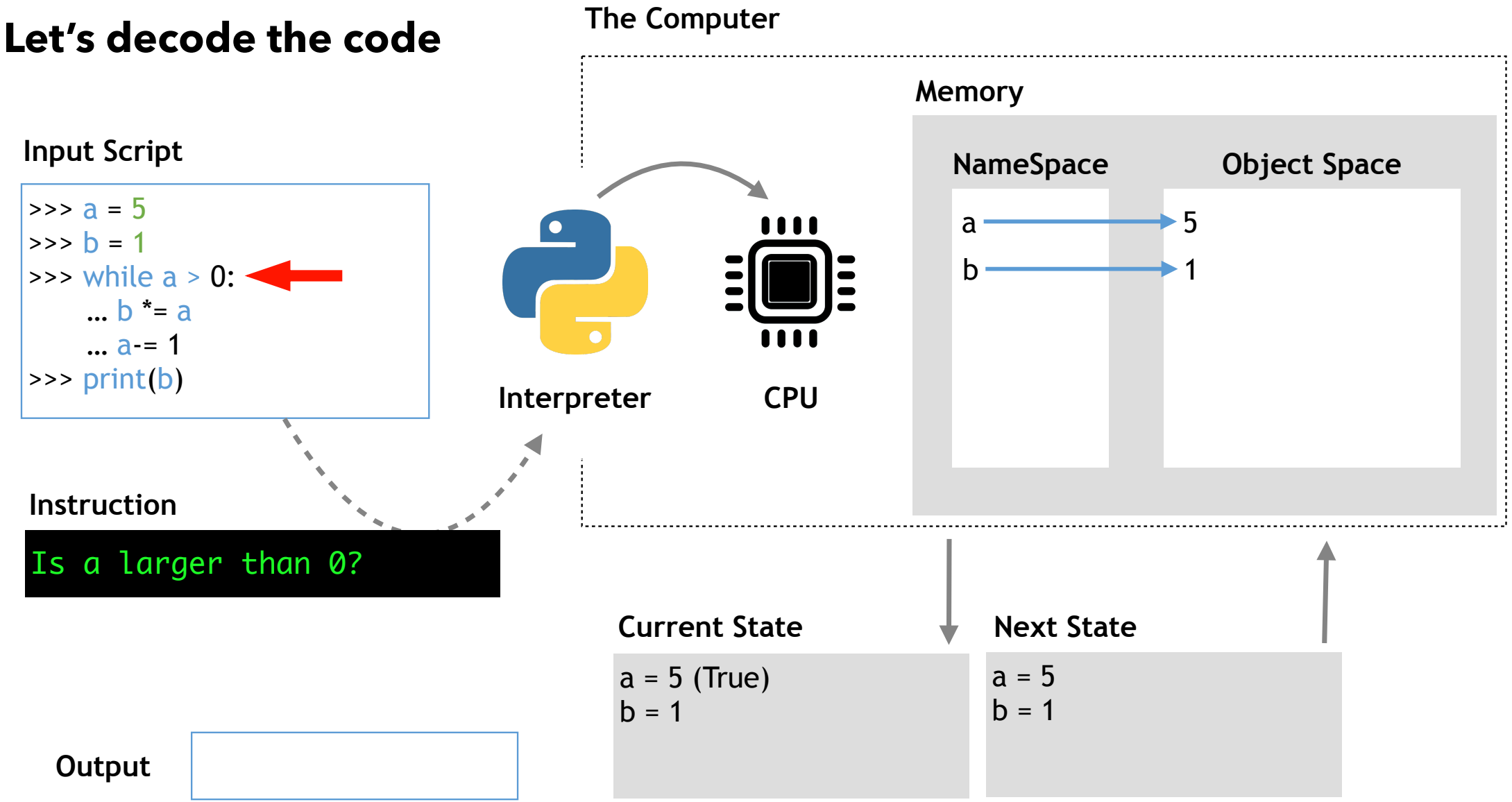
Let's decode the code



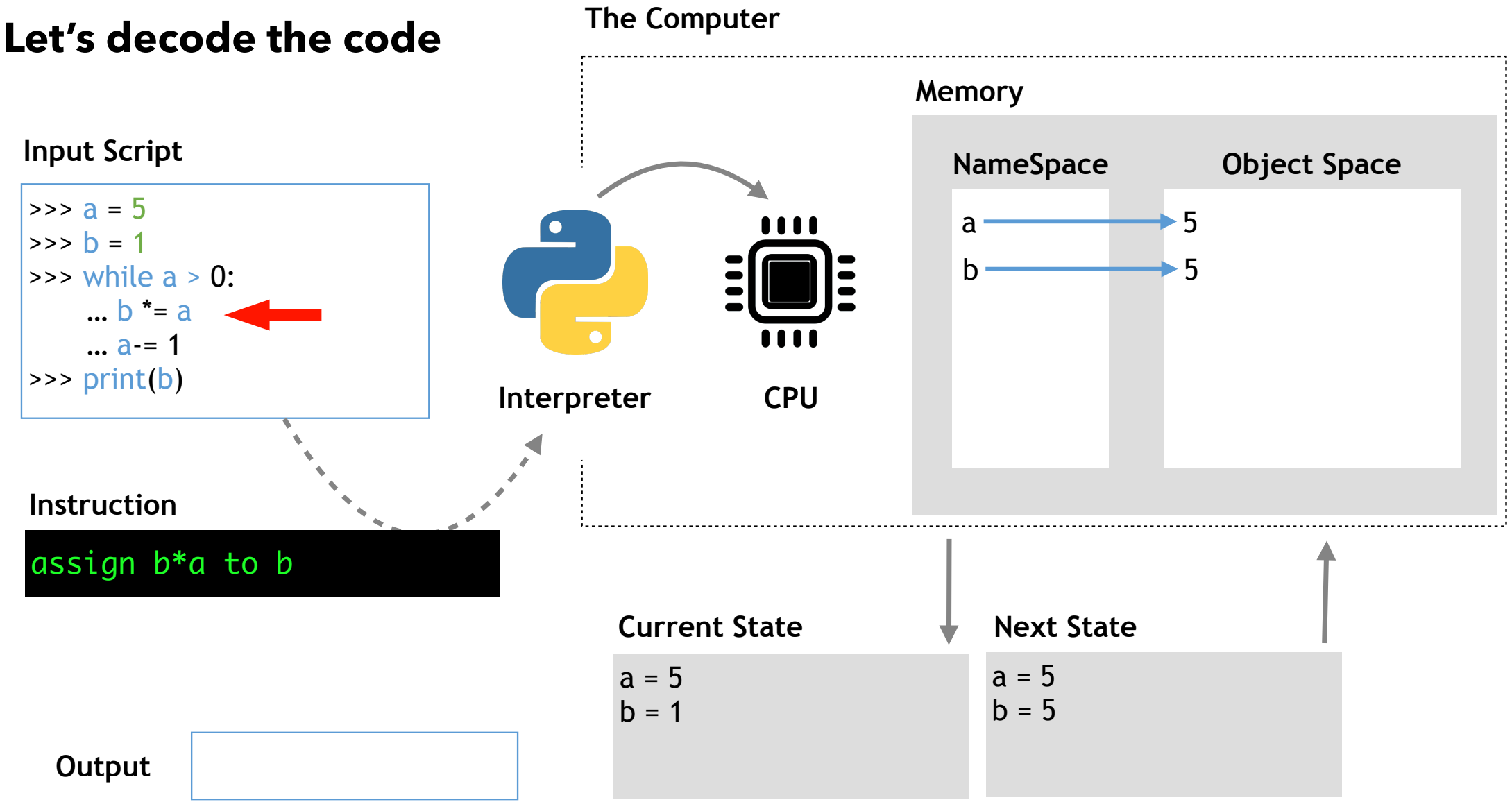
Let's decode the code



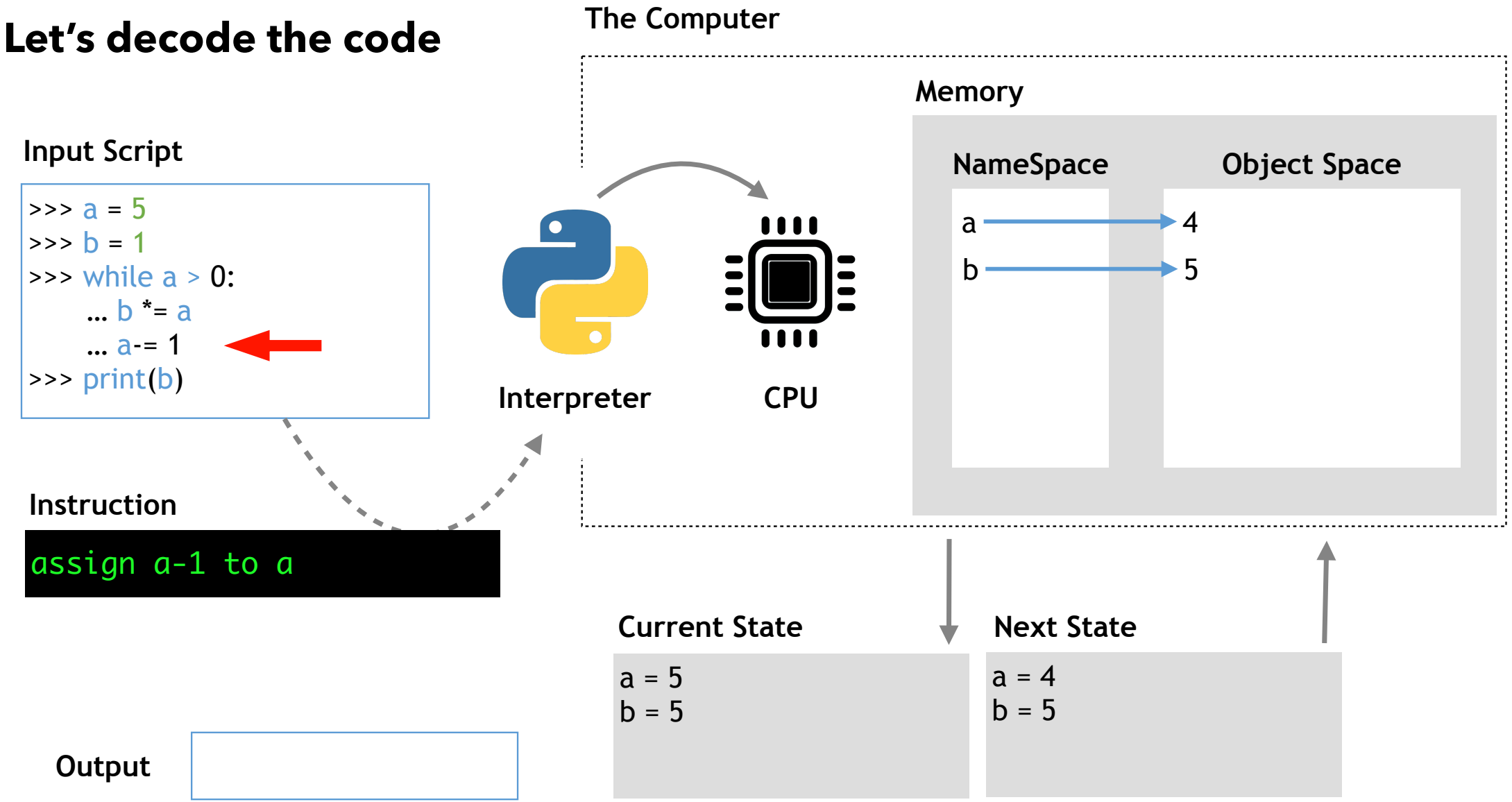
Let's decode the code



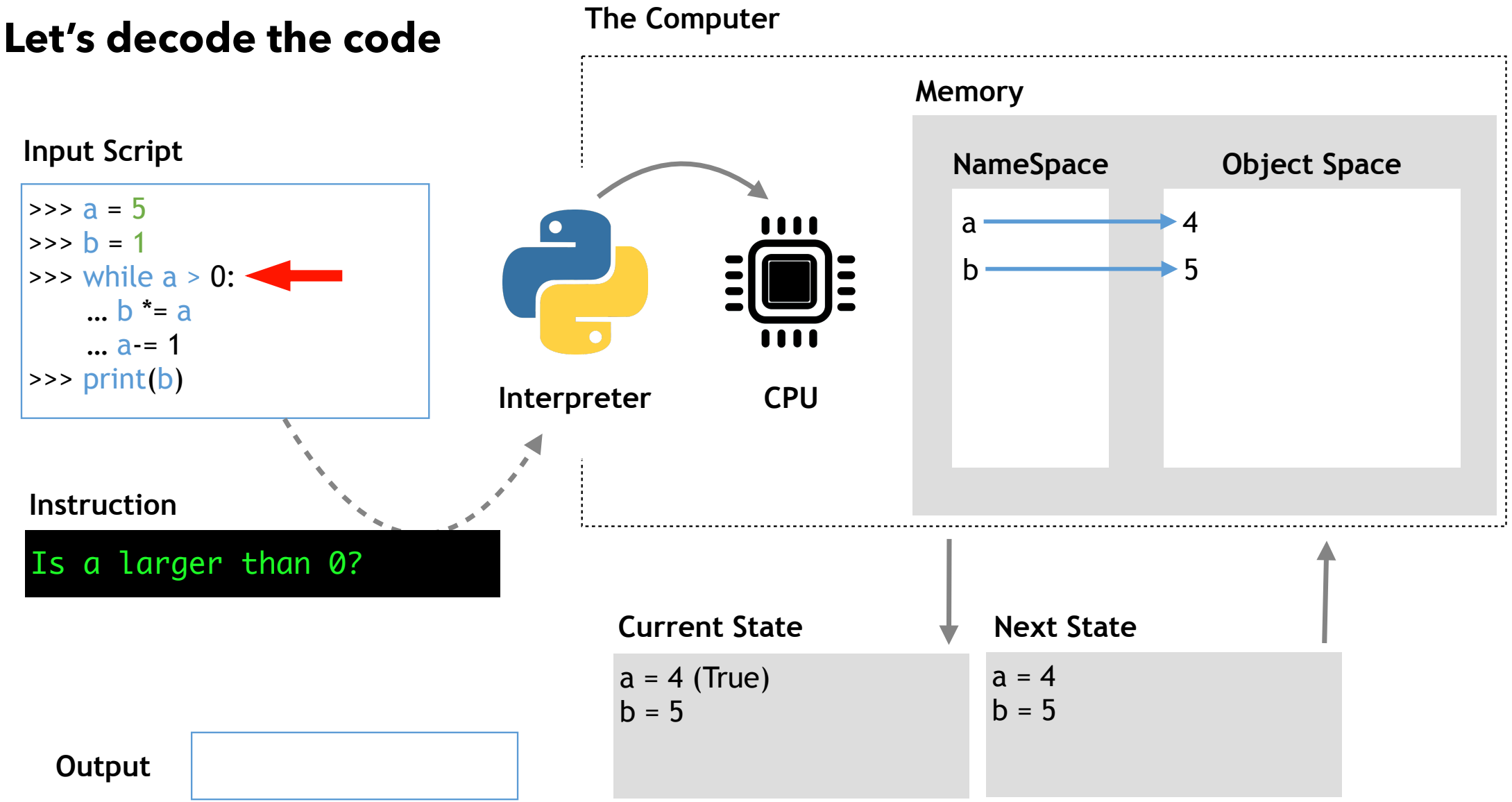
Let's decode the code



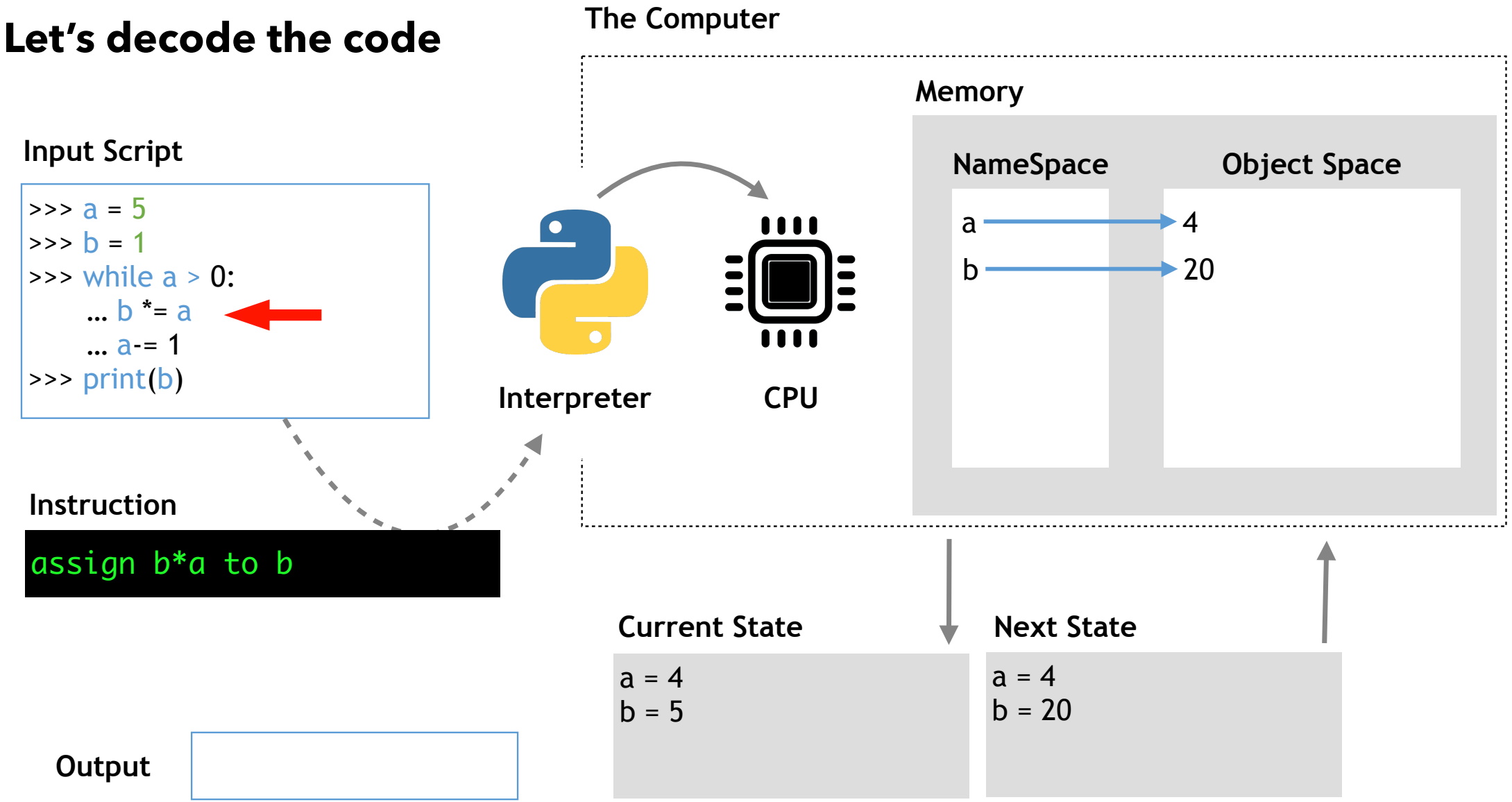
Let's decode the code



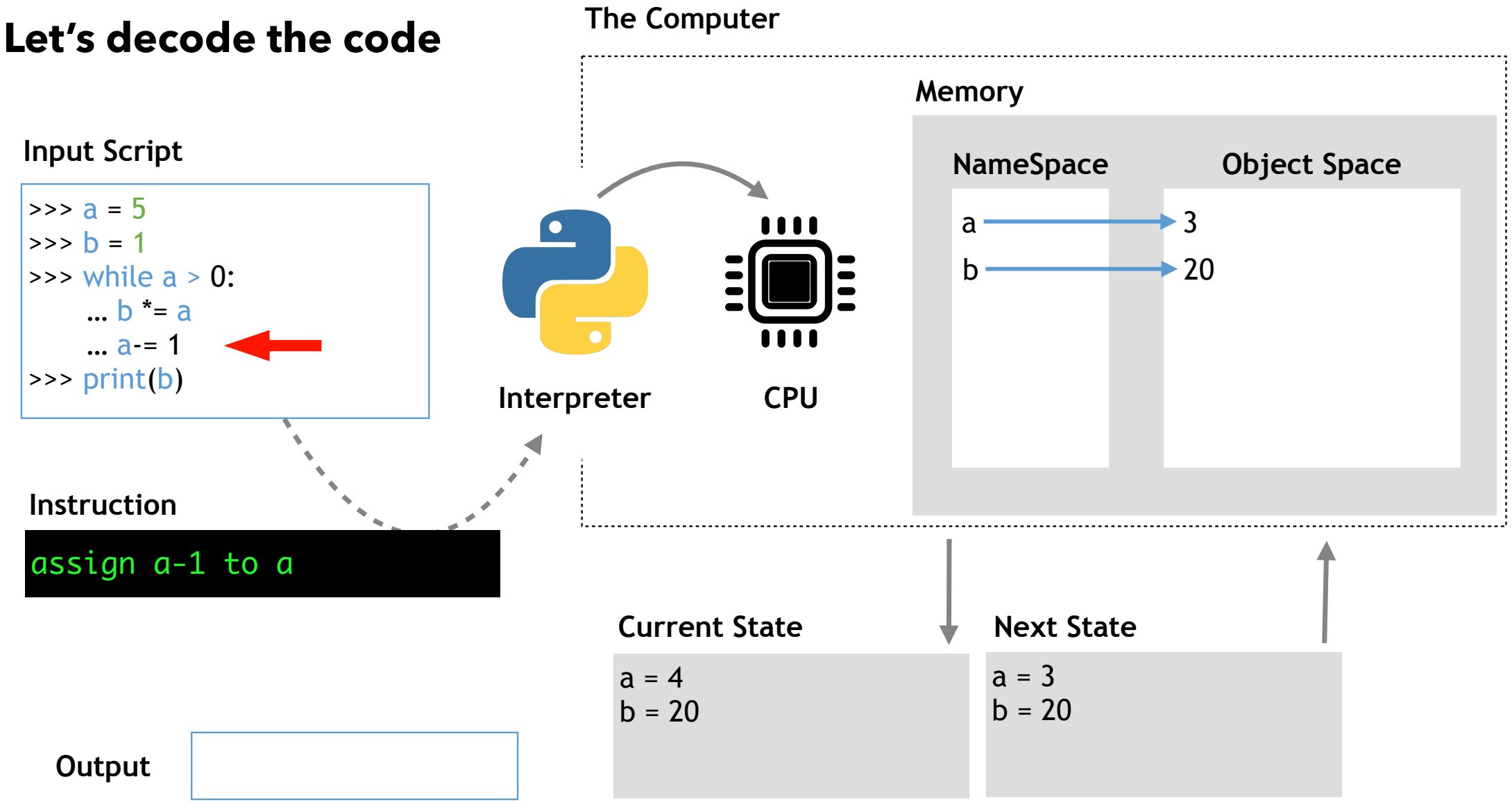
Let's decode the code



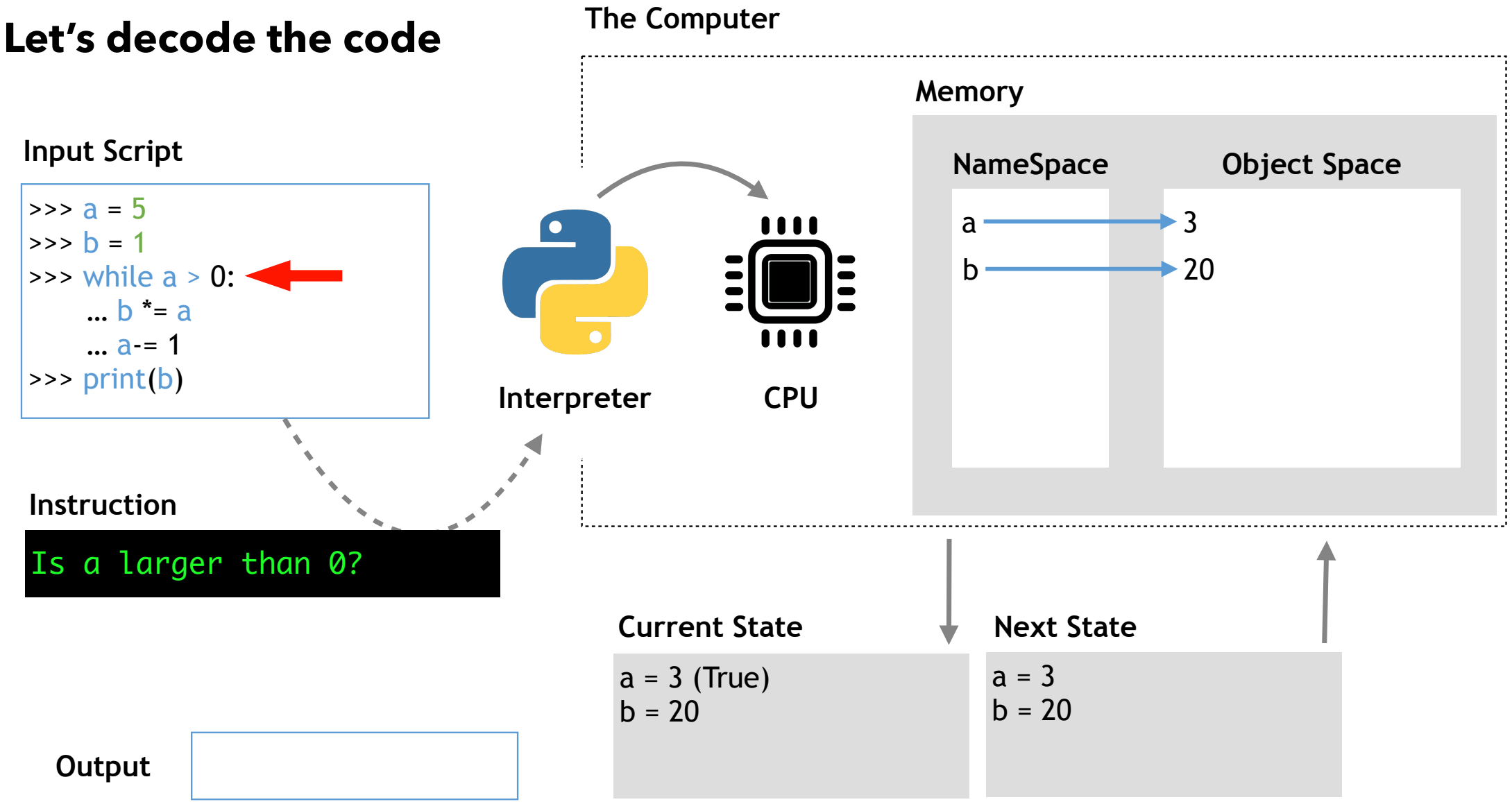
Let's decode the code



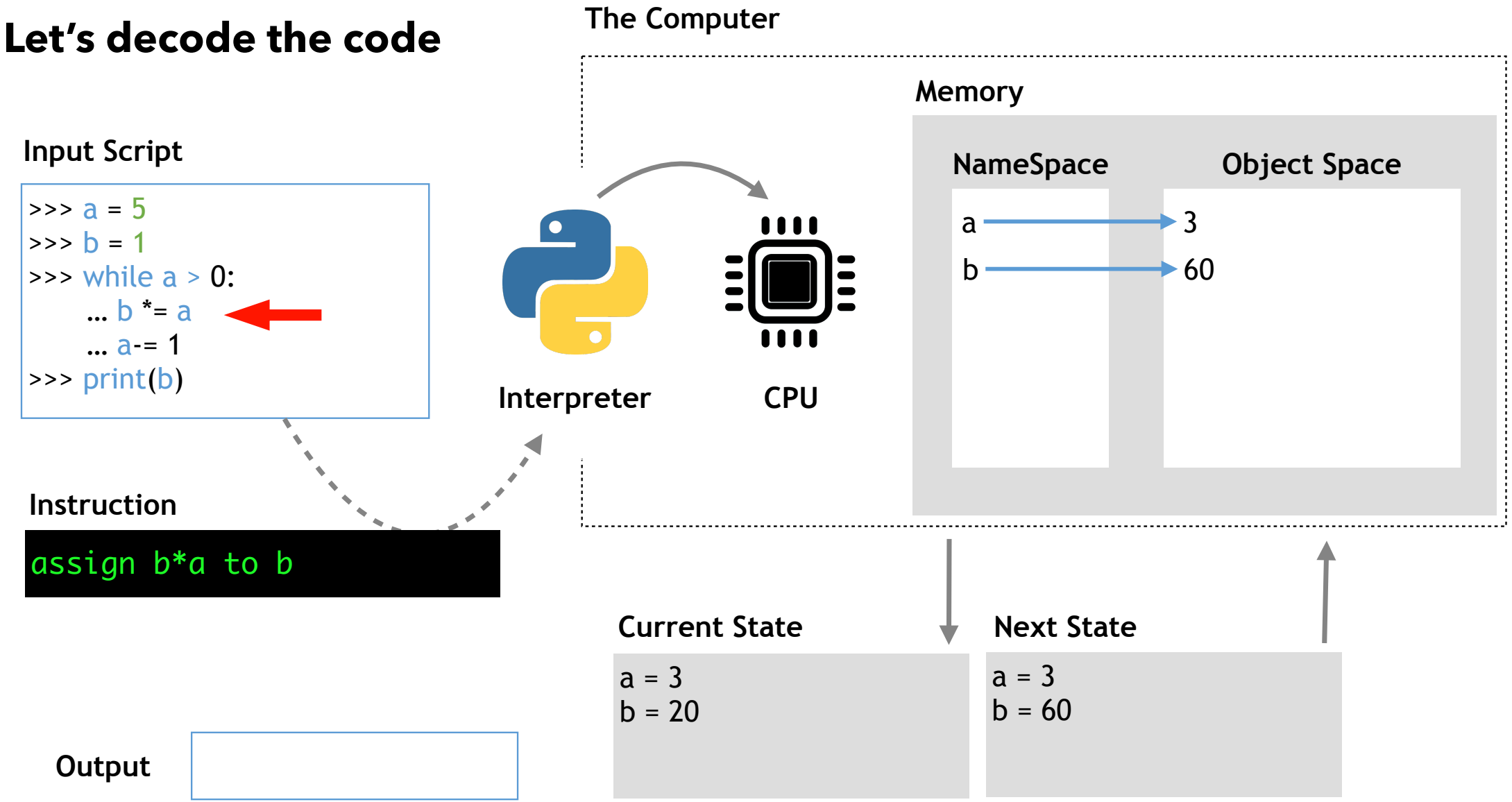
Let's decode the code



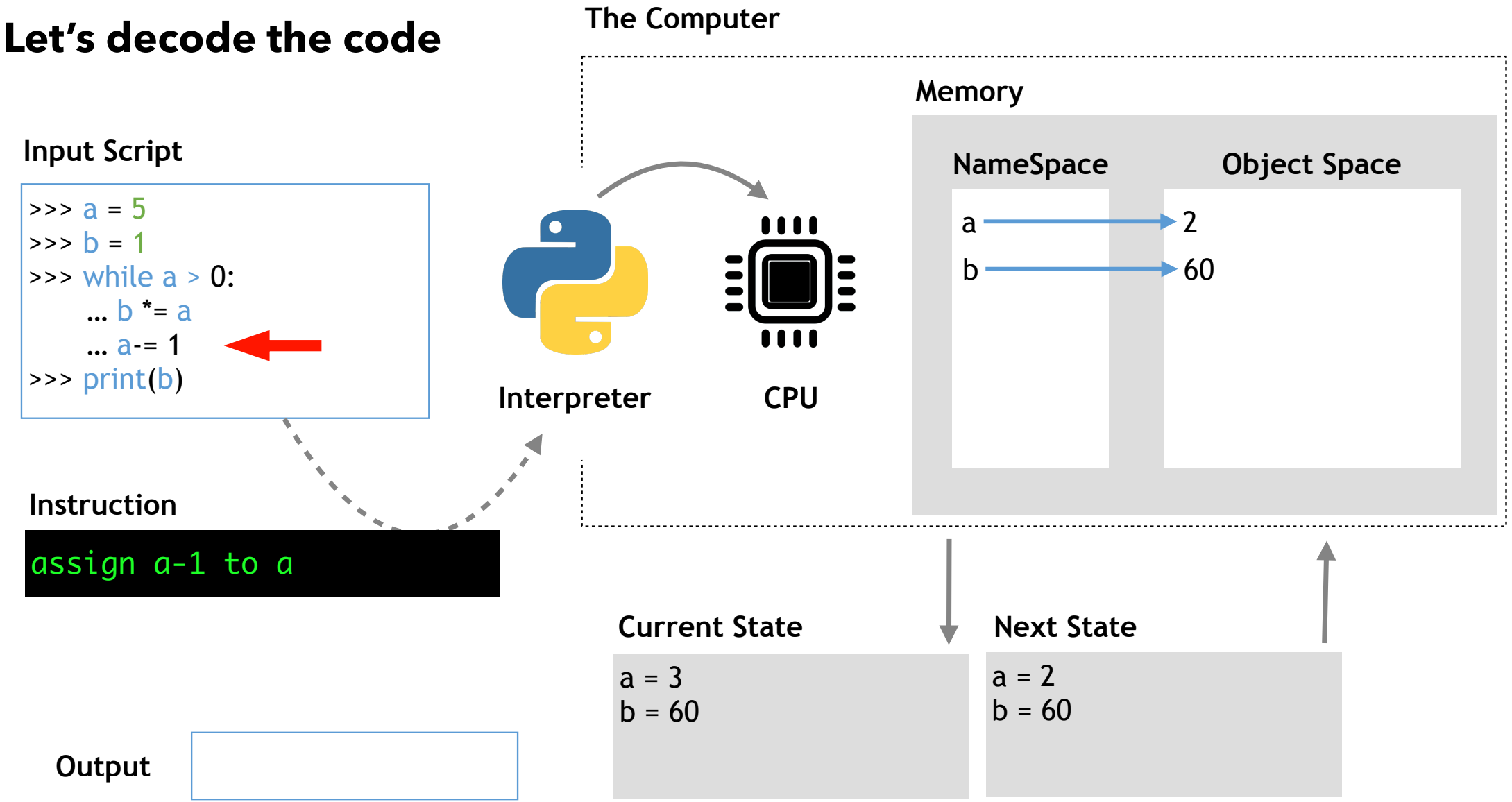
Let's decode the code



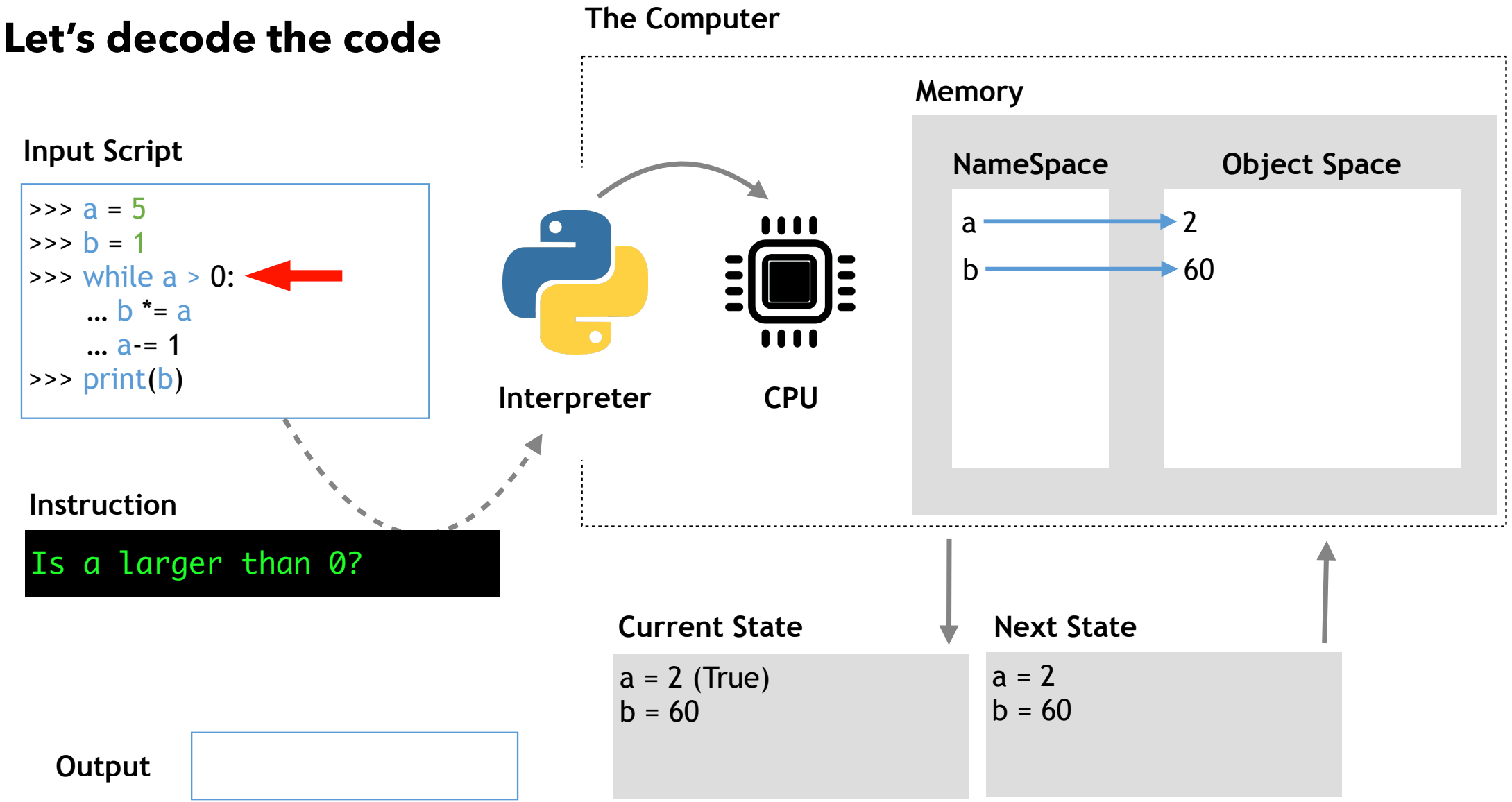
Let's decode the code



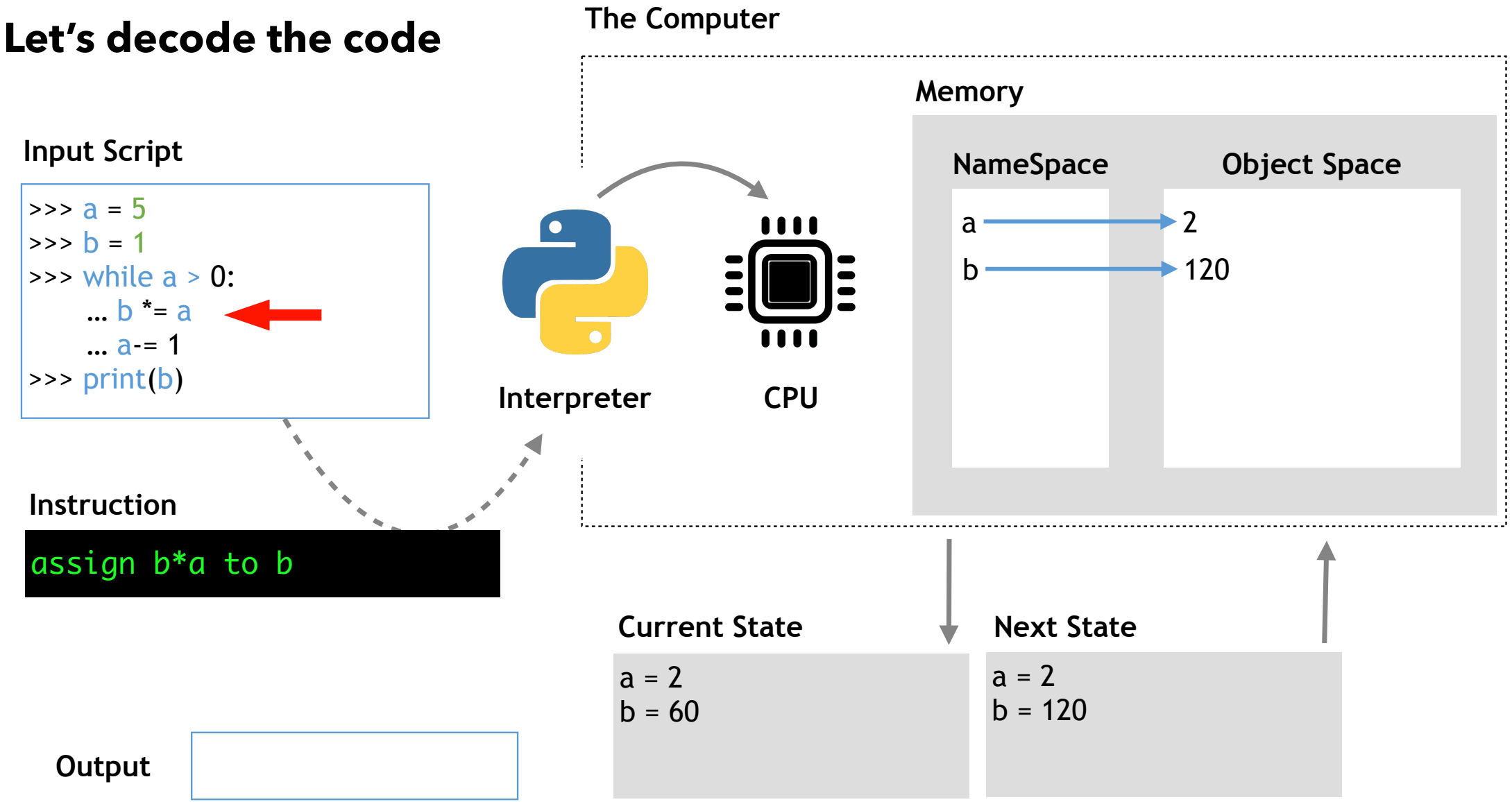
Let's decode the code



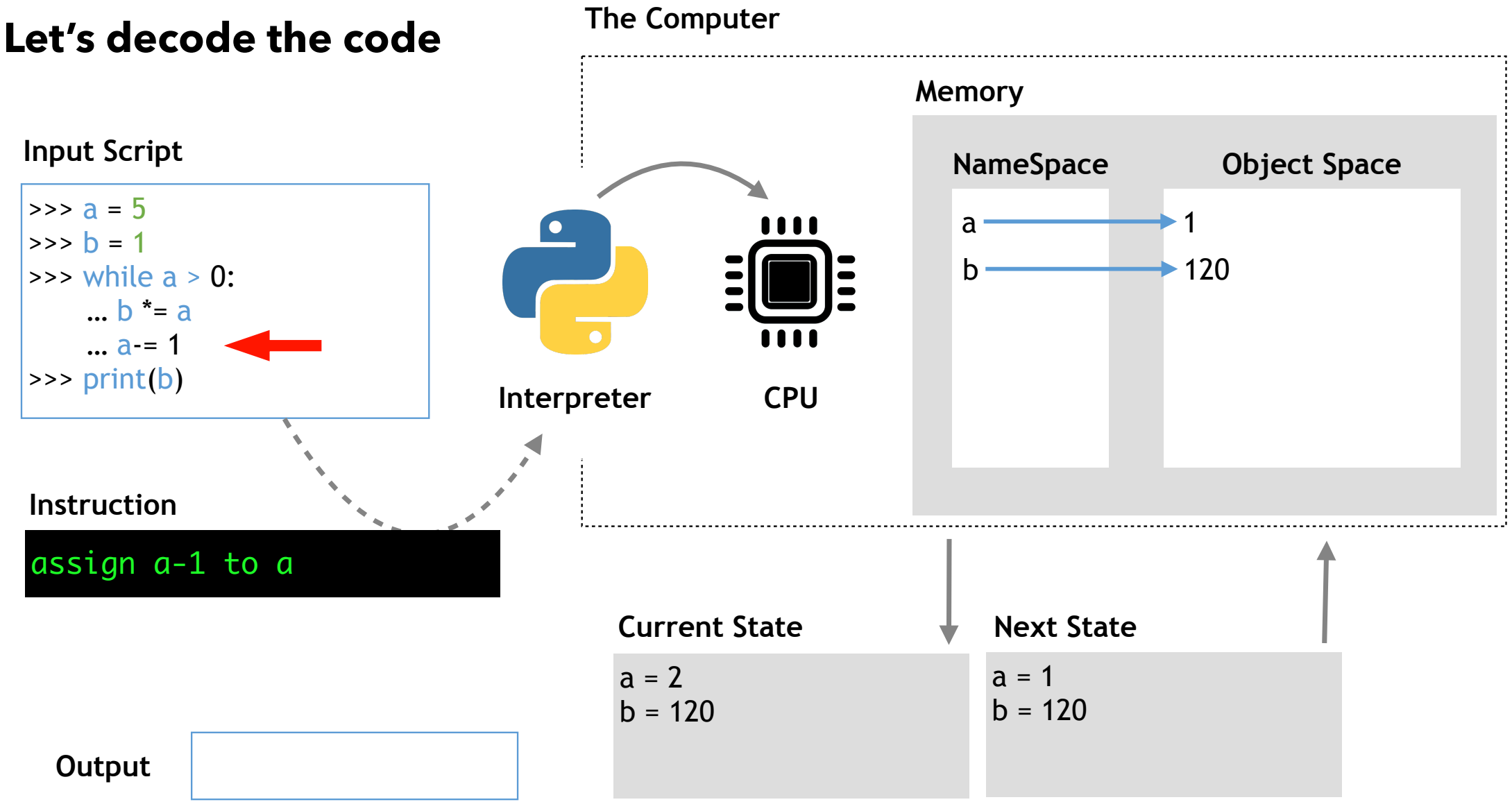
Let's decode the code



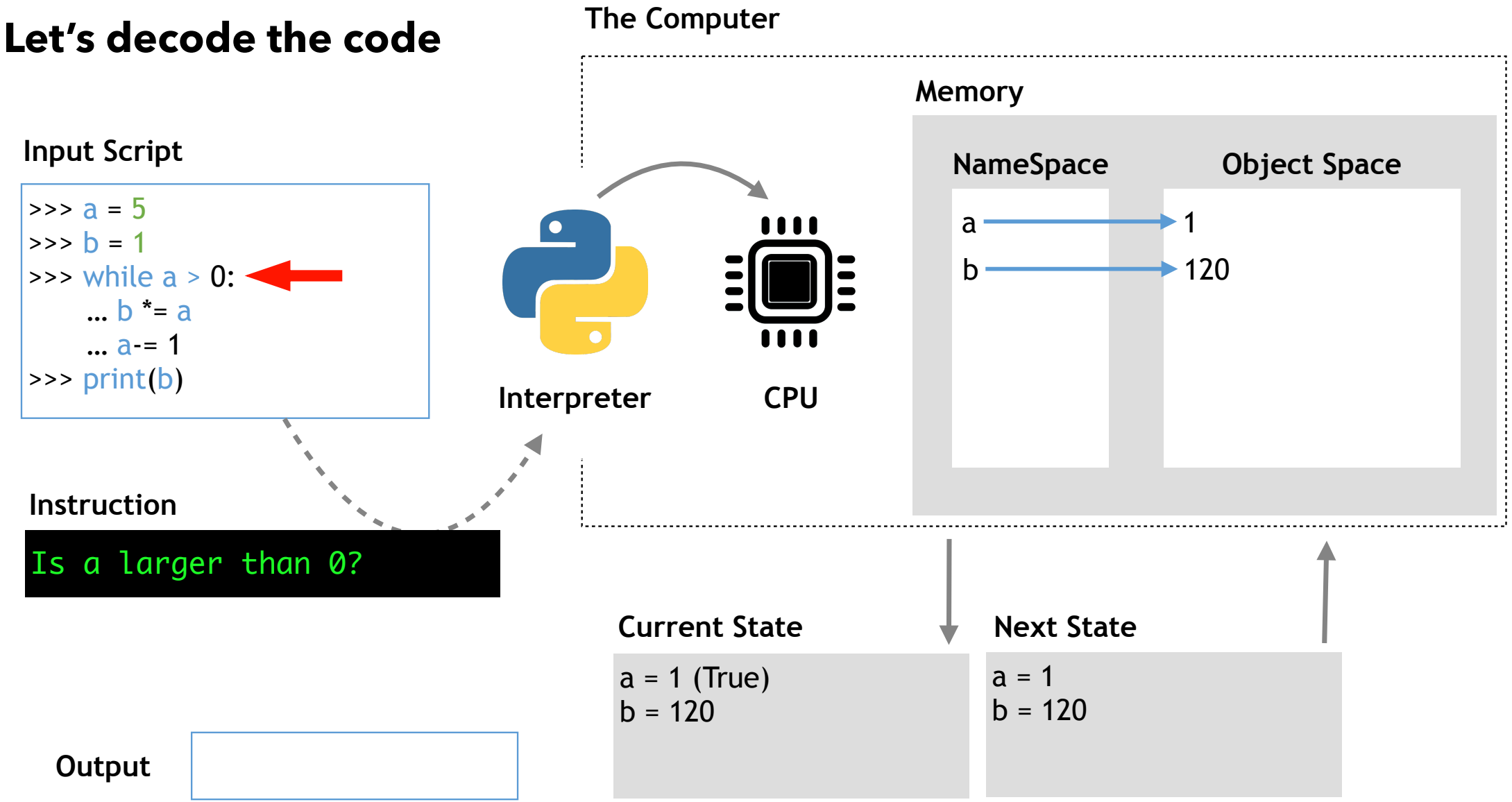
Let's decode the code



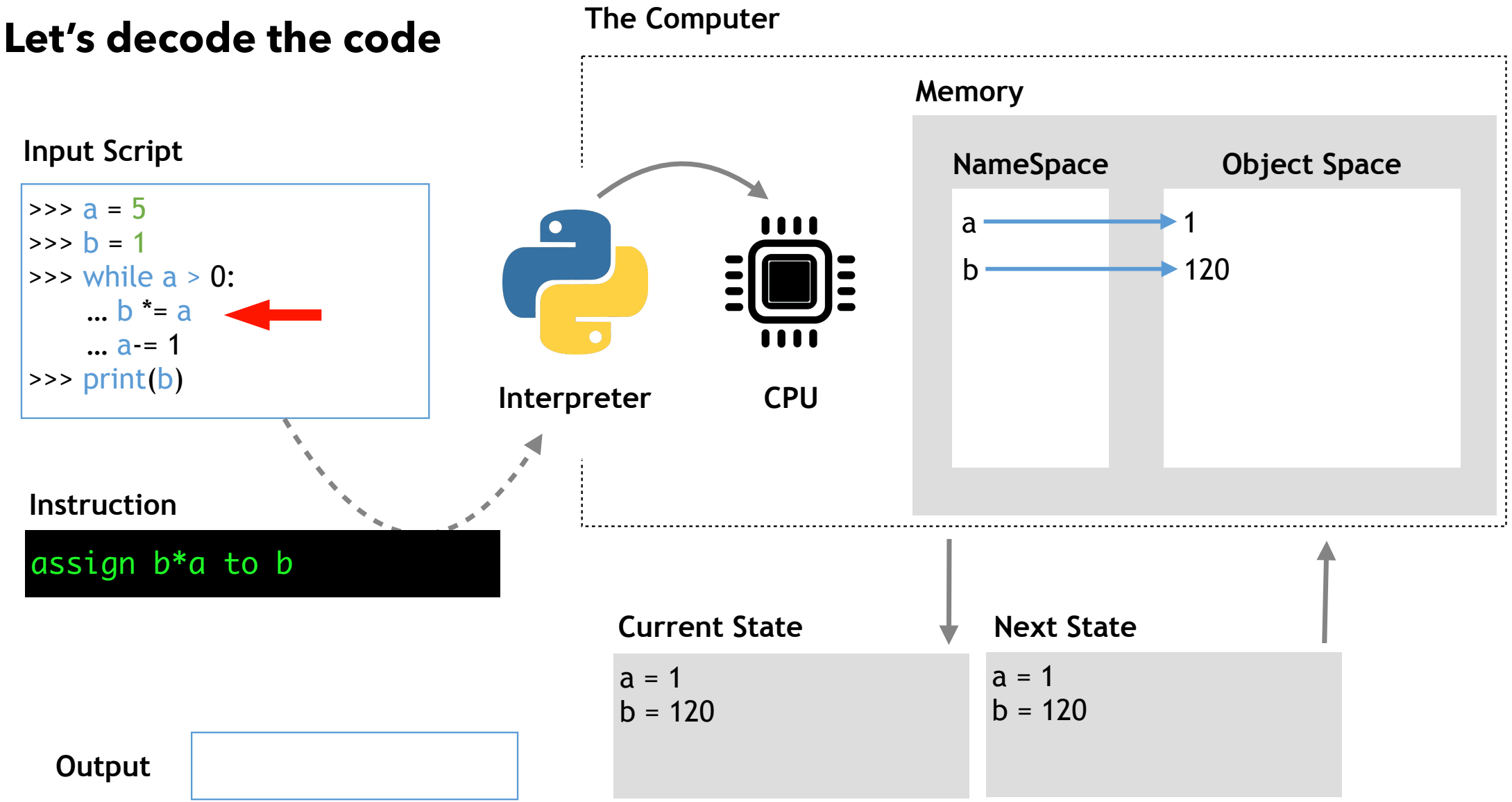
Let's decode the code



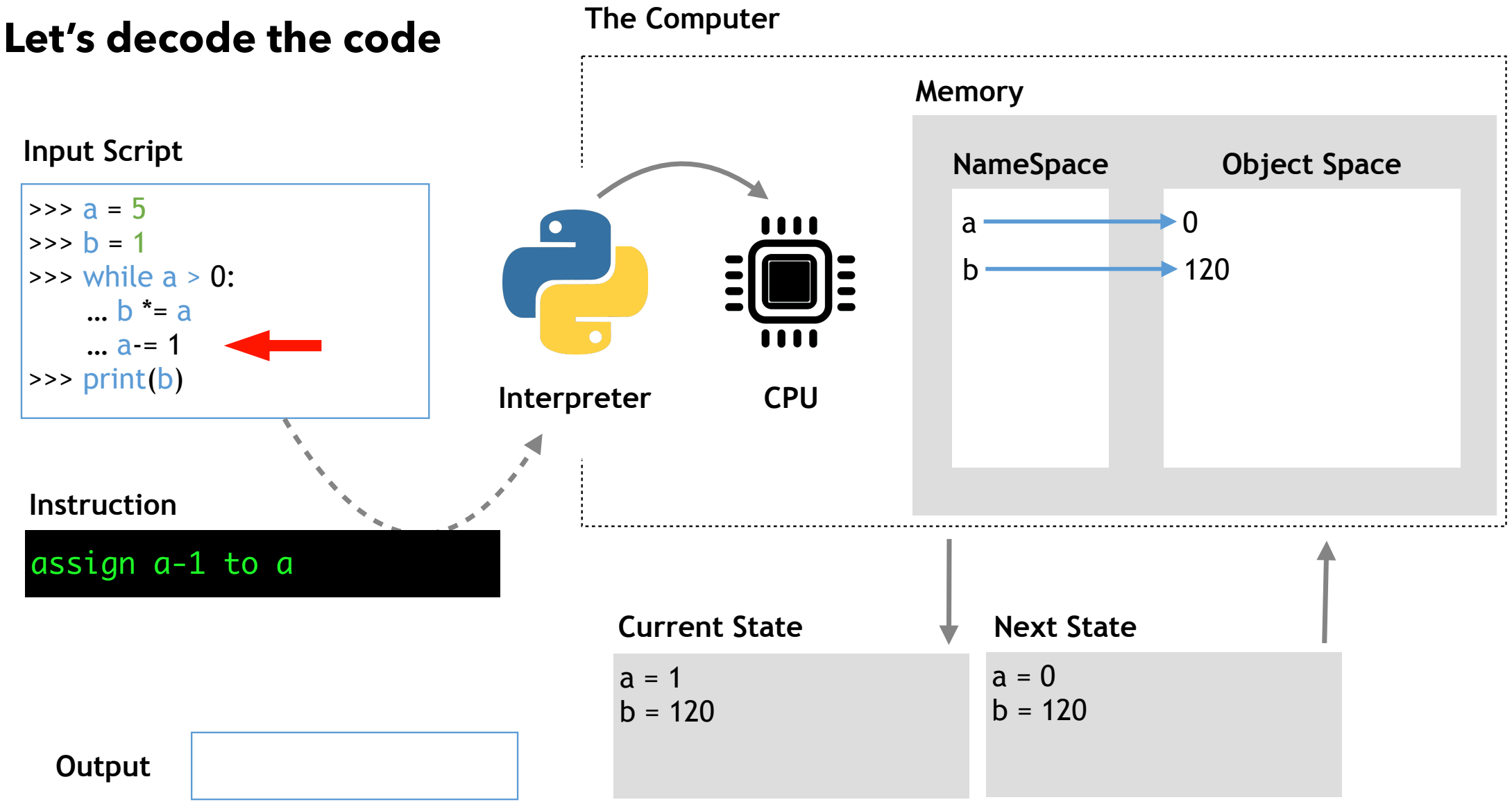
Let's decode the code



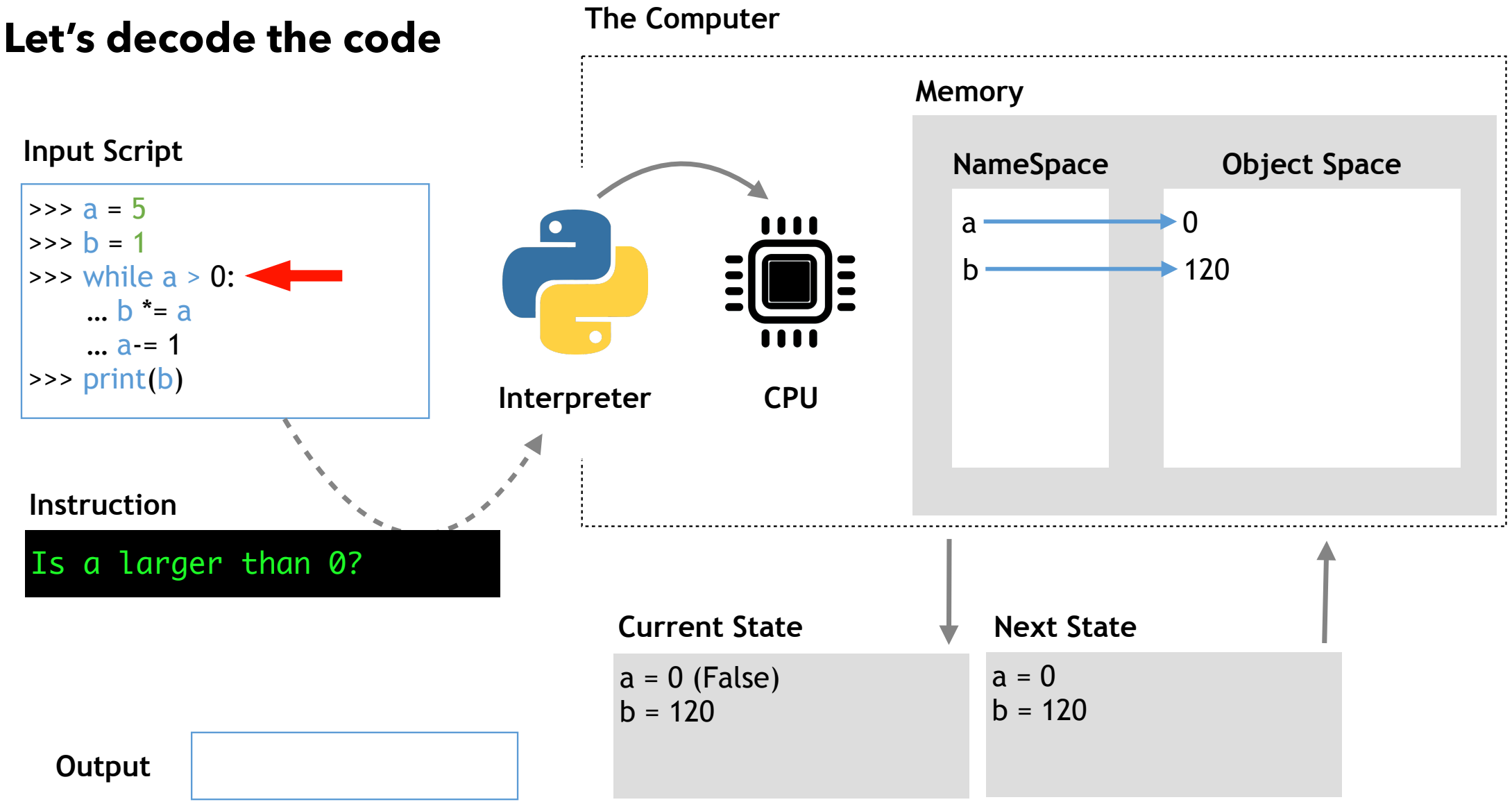
Let's decode the code



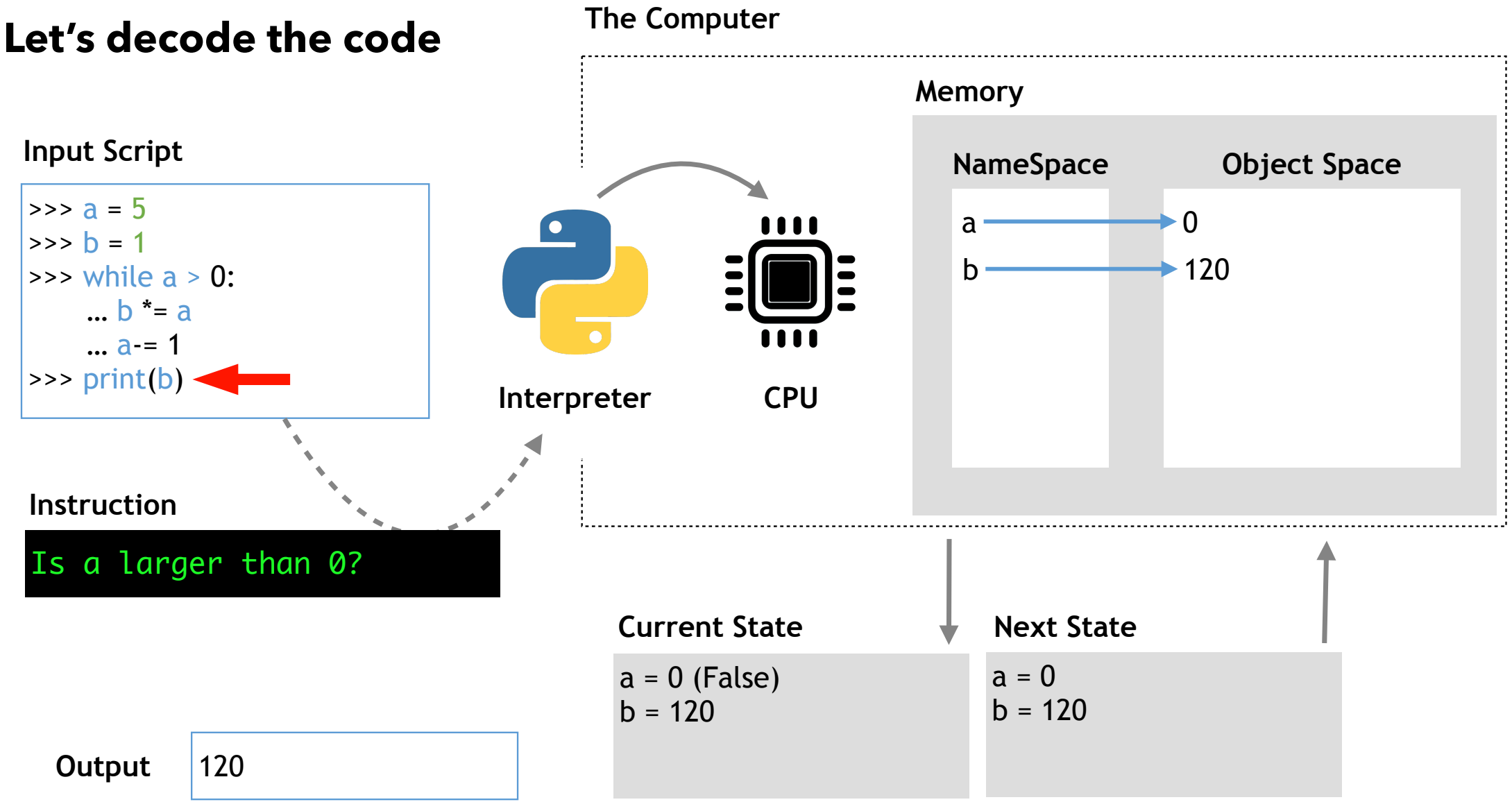
Let's decode the code



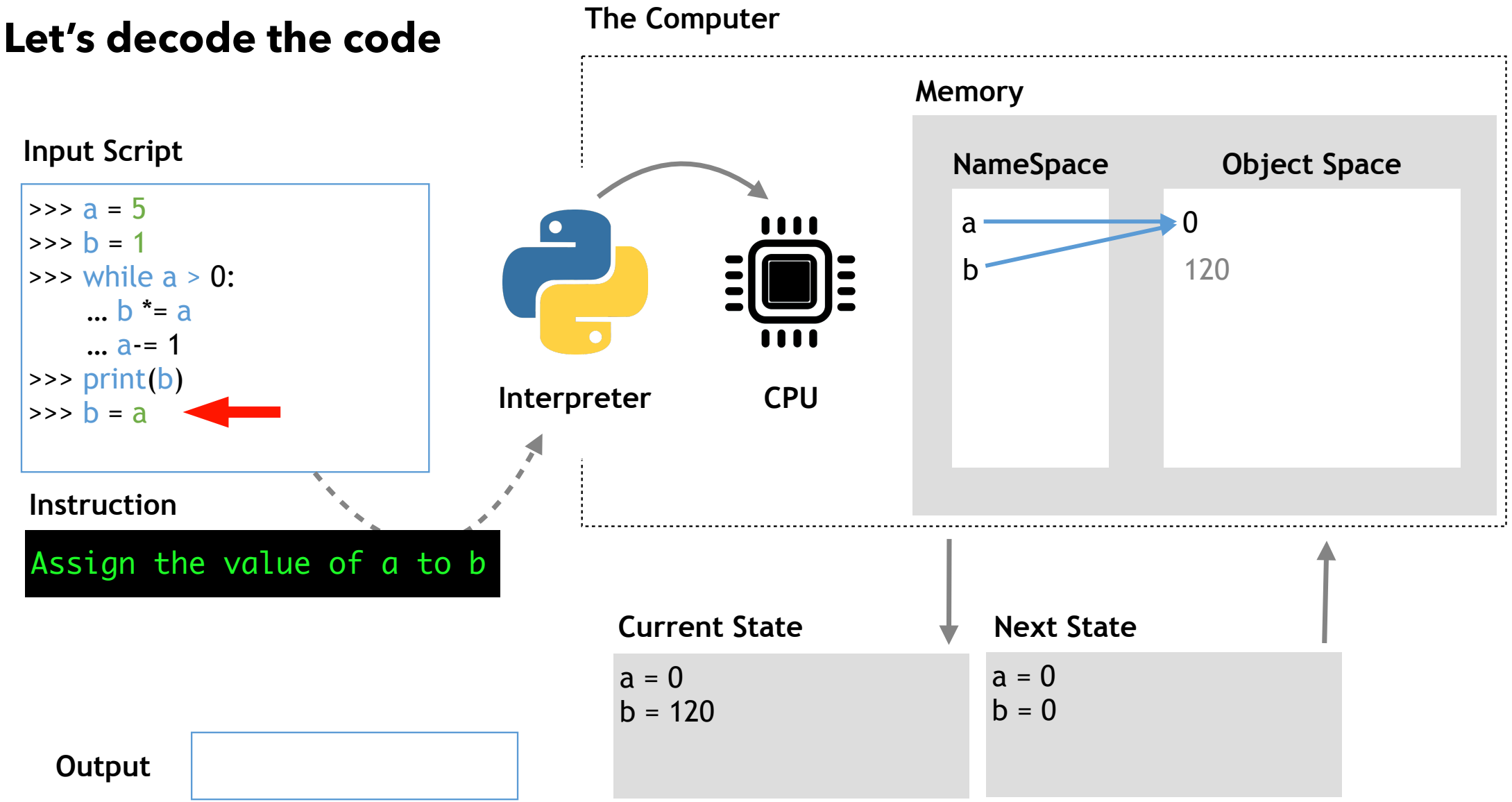
Let's decode the code



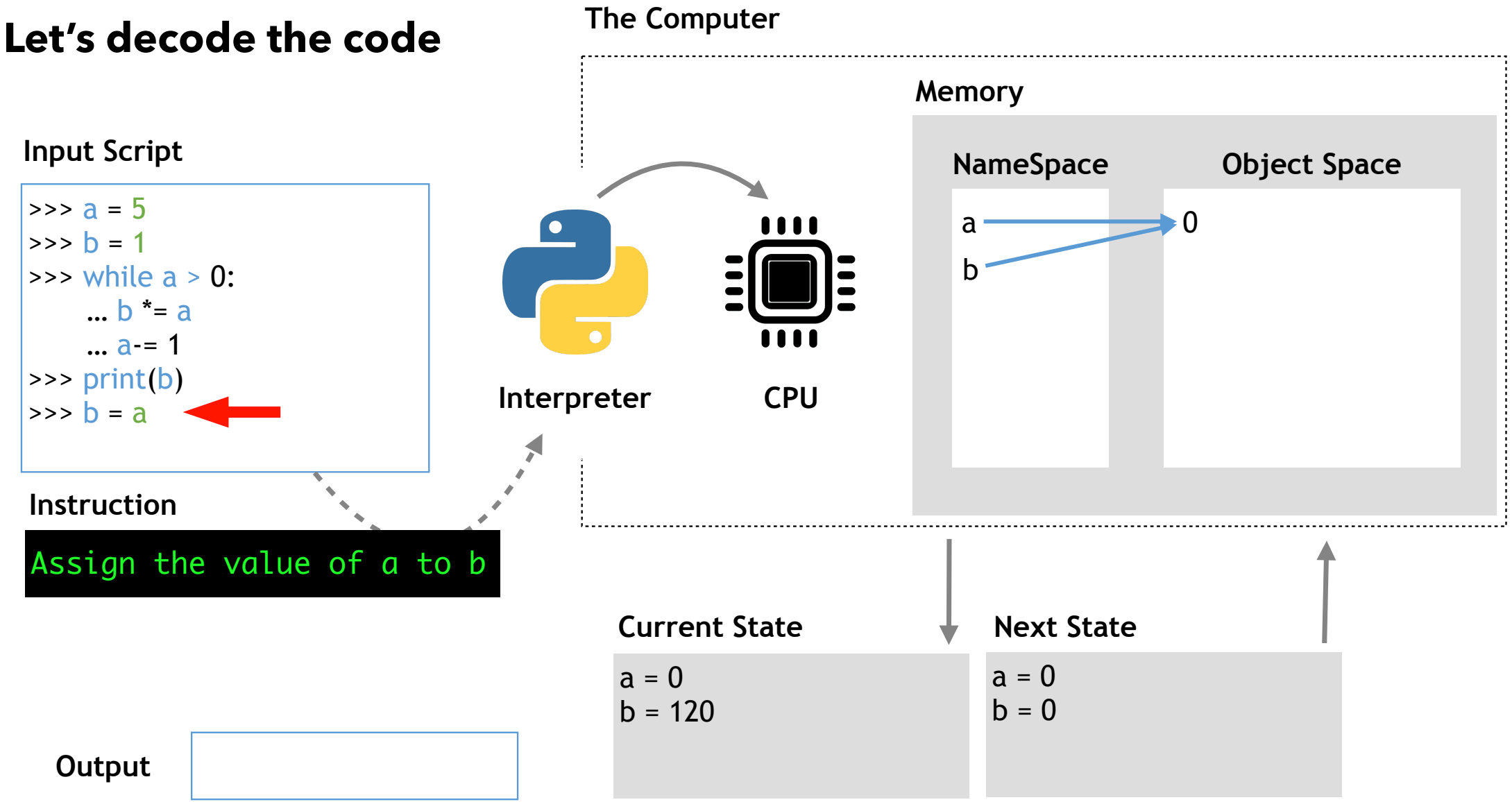
Let's decode the code



Let's decode the code

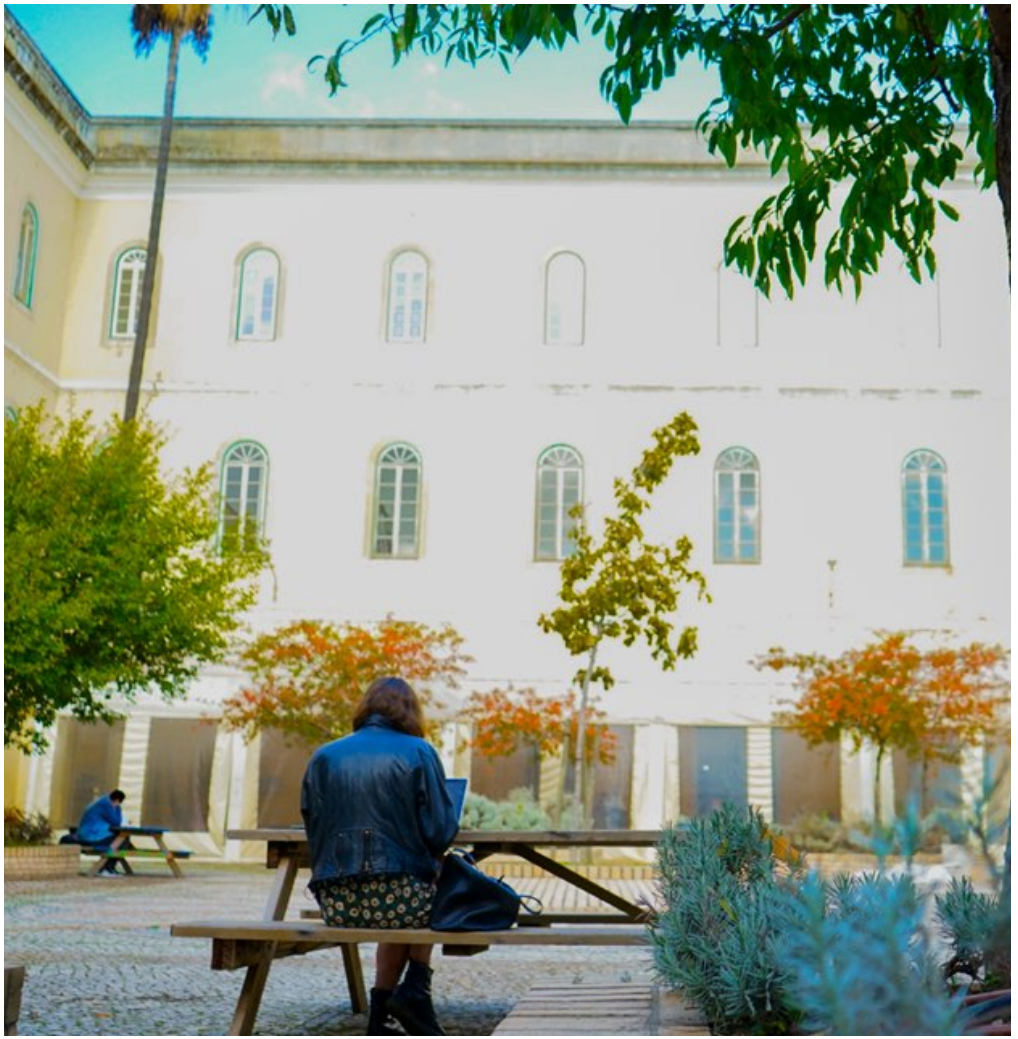


Let's decode the code



Lecture 3

- **Functions**
- Import Modules and Python STL
- NameSpace



Functions

Python already includes many built-in functions, e.g. print(). However sometimes it might be useful to extend the functionalities of our code by writing our own functions.
What are Functions? Functions are blocks of code that do some operations when called.
Functions allow us to split a large task, with repetitive elements, into smaller blocks. Additionally, writing functions eases the process of debugging your code.

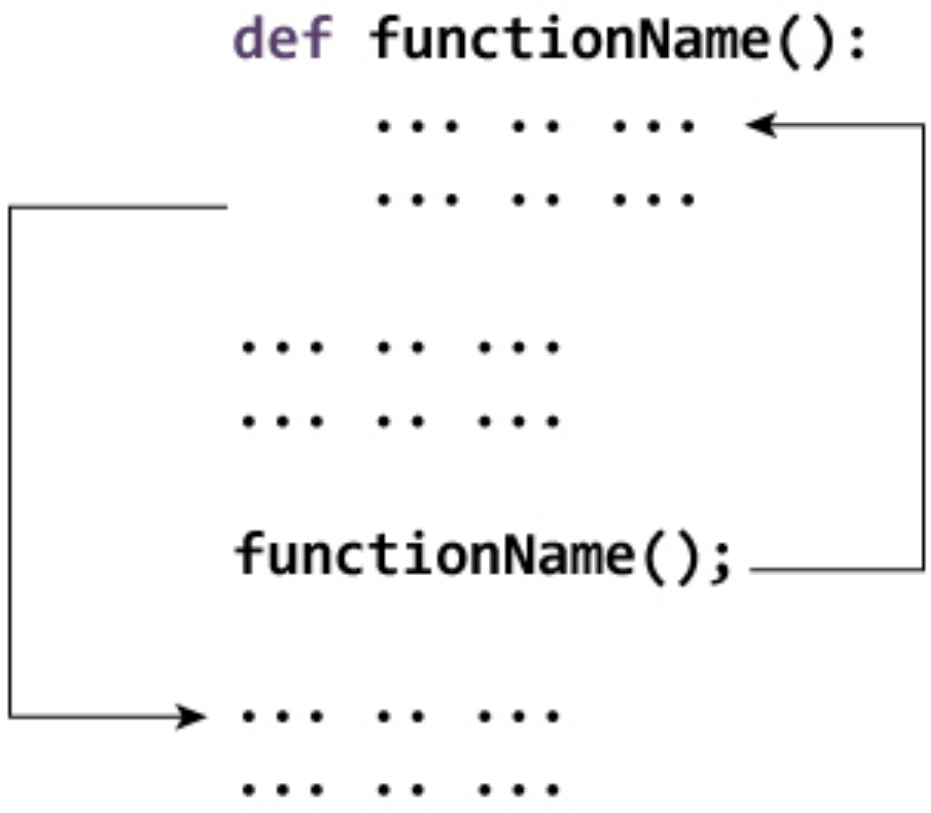
define it

```
def function_name (arguments):
    #perform tasks
    return something
```

call it

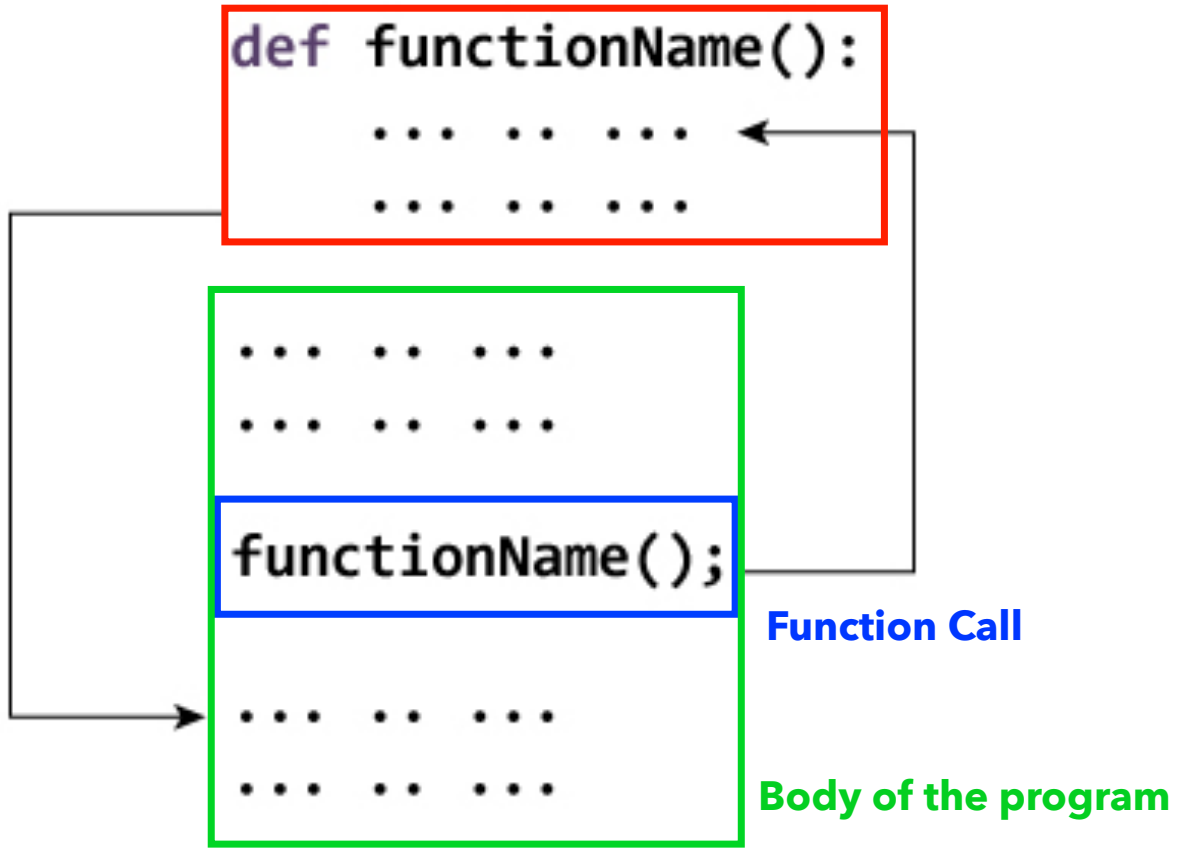
```
function_name(arguments)
```

Functions How functions work?



Functions

How functions work?



Functions Examples

```
def hello():
    print("hello world!")
```

```
hello
```

```
<function __main__.hello()>
```

```
hello()
```

```
hello world!
```

Functions Examples

```
def hello():
    print("hello world!")
```

Function Declaration

```
hello
```

```
<function __main__.hello()>
```

```
hello()
```

Calling the Function

```
hello world!
```

Functions Examples

```
def multiply(a,b):
    output = a*b
    return output

product = multiply(2,2)
print(product)
```

```
4
```

```
def divide(a,b):
    output = a/b

division = divide(2,4)
print(division)
```

```
None
```

Functions Examples

```
def multiply(a,b):
    output = a*b
    return output

product = multiply(2,2)
print(product)
```

Function Declaration

Call function and store output in variable

```
4
```

```
def divide(a,b):
    output = a/b

division = divide(2,4)
print(division)
```

Function Declaration

Call function and store output in variable

```
None
```

Functions Examples

```
def function (lXst, target):
    for i, value in enumerate(lXst):
        if value == target:
            return i
    return -1
```

Functions Examples

Following Function tells you what is the index of a specific value in a list

```
def function (list, target):
    for i, value in enumerate(list):
        if value == target:
            return i
    return -1
```

returns the position in list of target, or -1 if it doesn't find the value

Functions Passing Multiple Arguments

Python allows to pass two wildcard arguments that can represent Lists (*args) or Dictionaries/Keys (**kargs) with multiple elements.

```
def function_name (arguments, *args, **kargs):
    #perform tasks
    return something
```

the trick here are the “*” and the “**”. We can name these arguments whatever we want they will do the same function, but by convention it is good to name them args and kargs.

Functions Passing Multiple Arguments | Examples

```
def greet(*args):
    for name in args:
        print("Hello",name)
greet("Monica","Luke","Steve","John")
```

```
Hello Monica
Hello Luke
Hello Steve
Hello John
```

Functions Passing Multiple Arguments | Examples

```
def multiply(*args):
    output = 1
    for n in args:
        output *= n
    return output

print(multiply(2,4,5,6))
print(multiply(12,34,54,346))
```

```
240
7623072
```

Functions Passing Multiple Arguments | Examples

```
def something(**kwargs):
    for key, value in kwargs.items():
        print("The value of",key,"is", value)

something(January=1,February=2,March=3)
```

```
The value of January is 1
The value of February is 2
The value of March is 3
```

Functions Passing Multiple Arguments | Examples

```
def foo (*args,**kwargs):
    for value in args:
        print("The value",value)
    for key, value in kwargs.items():
        print("The value of",key,"is",value)

foo(1,2,3,4,a=1,b=2)
```

```
The value 1
The value 2
The value 3
The value 4
The value of a is 1
The value of b is 2
```

Hey Doc

Document your Functions

You can also add a description of the function

```
def foo (arguments, *args, **kargs):
    """ foo likes bar bar"""
    #perform tasks
    return something

foo.__doc__
```

```
' foo likes bar bar'
```

Hey Doc

Document your Functions

```
def greet(*args):
    """This function says Hello"""
    for name in args:
        print("Hello",name)
```

```
greet.__doc__
```

```
'This function says Hello'
```

```
greet("Monica","Luke","Steve","John")
```

```
Hello Monica
Hello Luke
Hello Steve
Hello John
```

What's your name? I don't know

Lambda functions

Sometimes we want to write a simple inline function for a specific task.
This is where lambda functions enter, a lambda function has the following form

```
lambda arguments: expression
```

which is the same as doing

```
def function_name (arguments):
    #perform tasks
    return expression;
```

but doesn't have a name

What's your name? I don't know

Lambda functions

```
lambda arguments: expression
```

These are also known as anonymous functions
as they don't require to be named.

What's your name? I don't know

Lambda functions

```
add = lambda x, y : x + y
```

```
add(2,3)
```

```
5
```

```
list(map(lambda x : x*2, [1, 2, 3, 4]))
```

```
[2, 4, 6, 8]
```

Map and Filter

```
add = lambda x, y : x + y
```

```
add(2,3)
```

```
5
```

1st argument 2nd argument

```
list(map(lambda x : x*2, [1, 2, 3, 4]))
```

```
[2, 4, 6, 8]
```

Map applies a function, first argument, to each object in a collection, second argument
in many cases we want to apply a simple function, and we can define it inline with lambda

Map and Filter

Besides the map(), filter() is another handy function that requires passing a function.

```
from random import sample
```

```
X = sample(range(-25,25),25)
```

```
print(X)
```

```
[-22, -7, 12, 4, -20, 1, 23, 14, -1, 21, -17, -16, 24, 20, -12, 2, -14, 17, -19, -9, -3, -6, -24, -2, -13]
```

```
f = filter(lambda x: x>0 , X)
```

```
print(f)
```

```
<filter object at 0x10edd90a0>
```

```
list(f)
```

```
[12, 4, 1, 23, 14, 21, 24, 20, 2, 17]
```

In these cases is often useful to just use lambda functions, and not declare explicitly a functionwith a name.

Map and Filter

Besides the map(), filter() is another handy function that requires passing a function.

```
from random import sample
```

Generates random samples from a container

```
X = sample(range(-25,25),25)
```

```
print(X)
```

```
[-22, -7, 12, 4, -20, 1, 23, 14, -1, 21, -17, -16, 24, 20, -12, 2, -14, 17, -19, -9, -3, -6, -24, -2, -13]
```

```
f = filter(lambda x: x>0 , X)
```

```
print(f)
```

returns an object

```
<filter object at 0x10edd90a0>
```

```
list(f)
```

we always need to typecast the object as a container

```
[12, 4, 1, 23, 14, 21, 24, 20, 2, 17]
```

In these cases is often useful to just use lambda functions, and not declare explicitly a functionwith a name.

Map and Filter

Besides the map(), filter() is another handy function that requires passing a function.

```
from random import sample
```

Generates random samples from a container

```
X = sample(range(-25,25),25)
```

```
print(X)
```

```
[-22, -7, 12, 4, -20, 1, 23, 14, -1, 21, -17, -16, 24, 20, -12, 2, -14, 17, -19, -9, -3, -6, -24, -2, -13]
```

```
f = filter(lambda x: x>0 , X)
```

```
print(f)
```

returns an object

```
<filter object at 0x10edd90a0>
```

```
list(f)
```

we always need to typecast

```
[12, 4, 1, 23, 14, 21, 24, 20, 2, 17]
```

```
for v in f:
    print(v)
```

```
12
4
1
23
14
21
24
20
2
17
```

In these cases is often useful to just use lambda functions, and not declare explicitly a functionwith a name.

What's your name? I don't know

Lambda functions

filter out odd numbers from a list

```
example = [1,2,3,4,5,6,7,8,9,10]
```

```
list(filter(lambda x: x%2 == 0, example))
```

```
[2, 4, 6, 8, 10]
```

What's your name? I don't know

Lambda functions

filter out odd numbers from a list

```
example = [1,2,3,4,5,6,7,8,9,10]
```

```
list(filter(lambda x: x%2 == 0, example))
```

```
[2, 4, 6, 8, 10]
```

1st argument

2nd argument

filter selects elements from a list for which the condition is true.

Recursive Functions

A function walks into a bar ...

A function is said to be recursive if it calls itself.
To implement a recursive function we have to define a termination condition,
otherwise it keeps calling itself forever

Example of how to compute the factorial of a number using a recursive function

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

```
factorial(5)
```

```
120
```

A function walks into a bar ...

A function is said to be recursive if it calls itself.
To implement a recursive function we have to define a **termination condition**,
otherwise it keeps calling itself forever

Example of how to compute the factorial of a number using a recursive function

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

What is going on?

```
factorial(5)
```

```
120
```

A function walks into a bar …

What is going on?

and plays MTG with his buddies

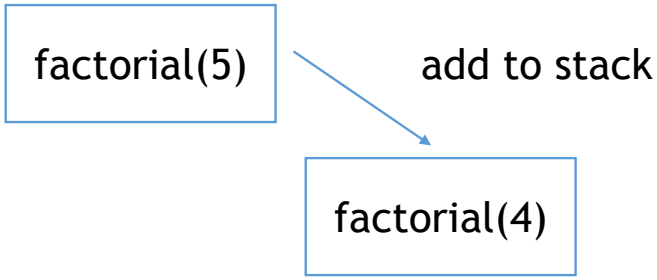
****geek joke****

A function walks into a bar ...

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

```
factorial(5)
```

```
120
```

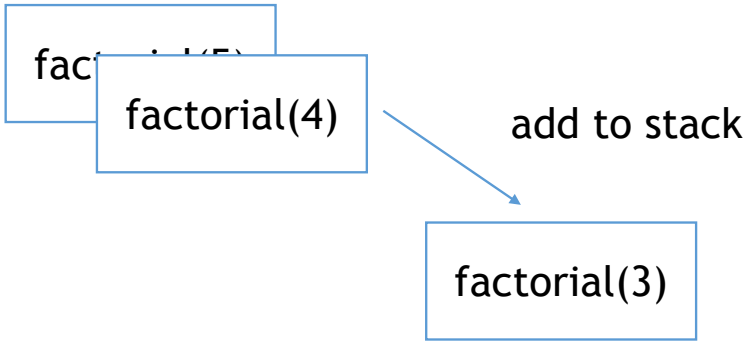


A function walks into a bar ...

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

```
factorial(5)
```

```
120
```

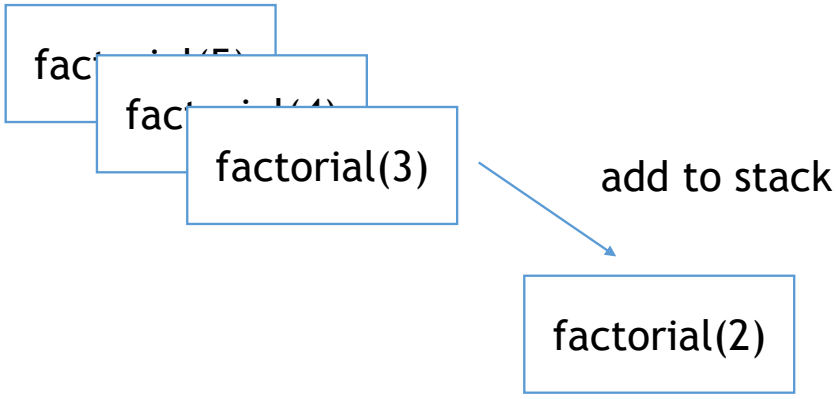


A function walks into a bar ...

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

```
factorial(5)
```

```
120
```

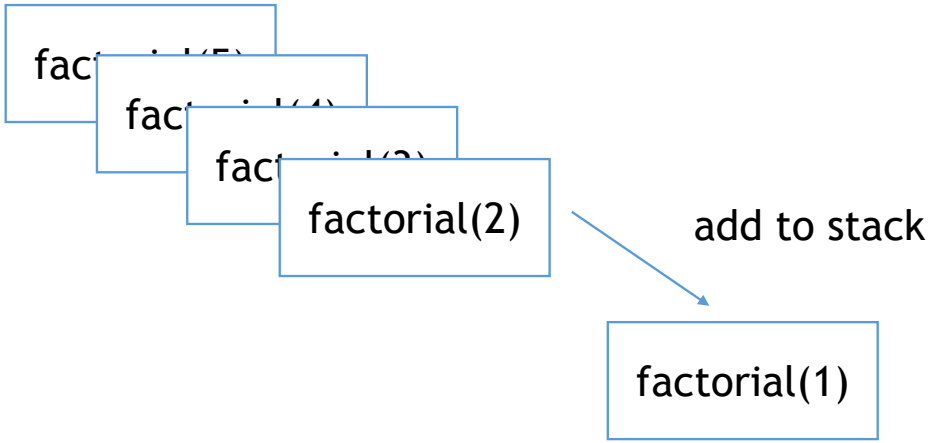


A function walks into a bar ...

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

```
factorial(5)
```

```
120
```

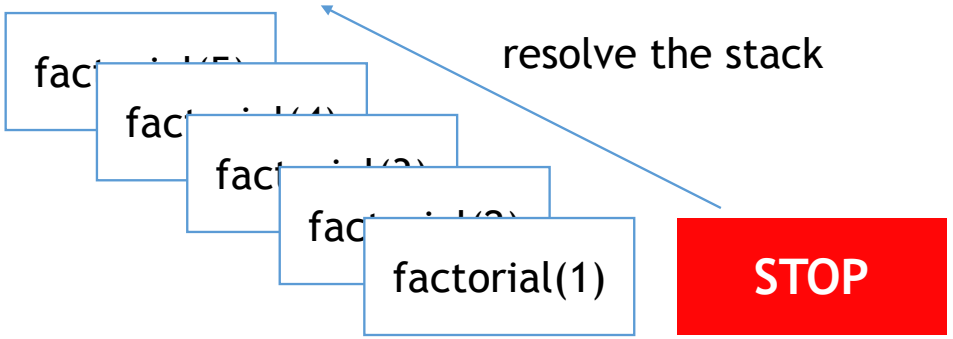


A function walks into a bar ... let me explain

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

```
factorial(5)
```

```
120
```



A function walks into a bar ...

```
def factorial(n):
    print("factorial has been called with n = " + str(n))
    if n == 1:
        return 1
    else:
        res = n * factorial(n-1)
        print("intermediate result for",n,"* factorial(",n-1,"):",res)
        return res
```

```
print(factorial(5))
```

Adding to the Stack

```
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1
```

Resolving the Stack

```
intermediate result for 2 * factorial( 1 ): 2
intermediate result for 3 * factorial( 2 ): 6
intermediate result for 4 * factorial( 3 ): 24
intermediate result for 5 * factorial( 4 ): 120
120
```

# of methods, functions, and attributes

What is the difference?

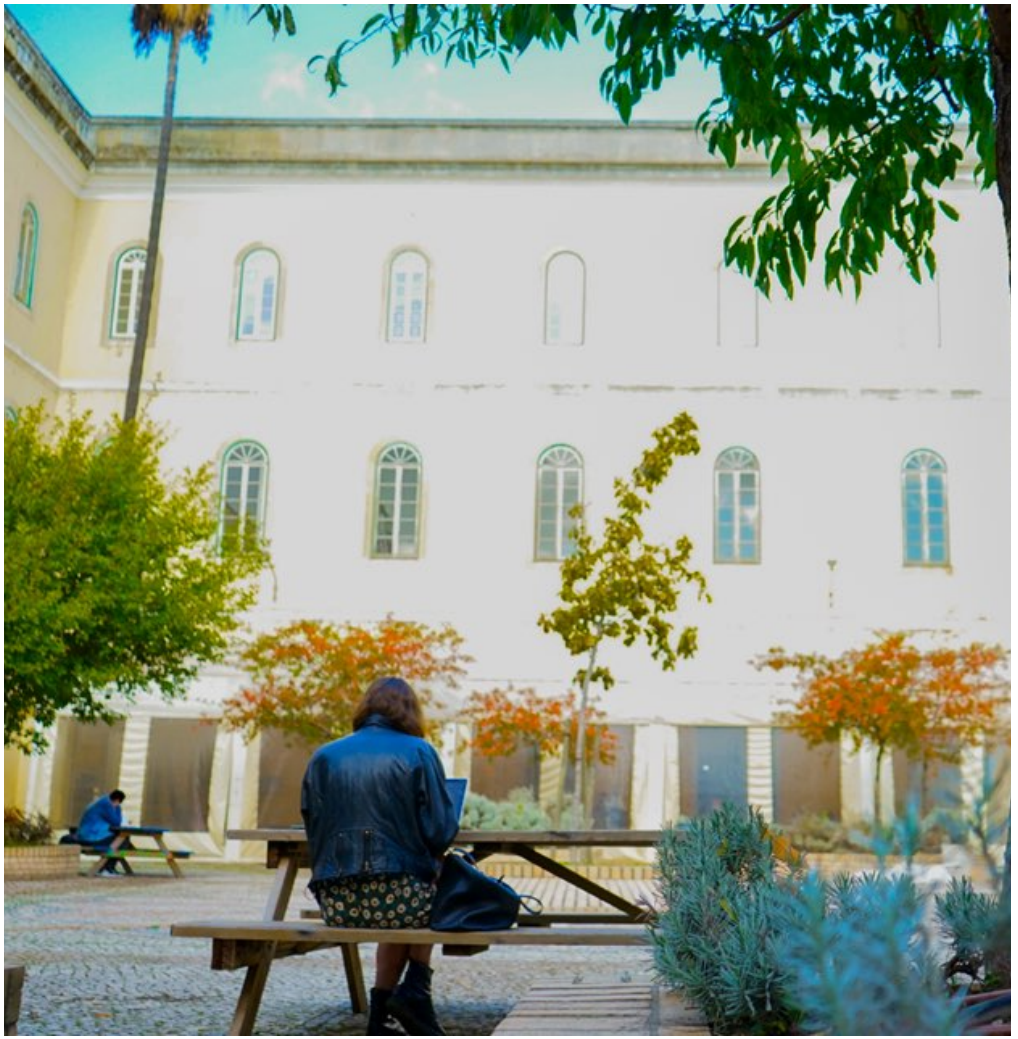
functions are functions as we saw previously. They perform some operations.
Functions are not dependent of an associated instance of an object.
Examples, map(), filter(), random.random(), or pd.DataFrame().

methods are functions that are associated with an instance of an object.
For instance, if you have X to be an instance of a list. Then X will have several methods available like .append() or .pop(). The scope of these functions is defined by the scope of the instance of the object.

attributes like methods, attributes are special variables that are associated with an instance of an object.
For instance, the attribute .shape is a tuple that describes the dimensions of a pandas dataframe.

Lecture 3

- ~~**Functions**~~
- **Import Modules and Python STL**
- NameSpace



Use of external modules

Modules are packages that extend the python library with functions useful for a particular application.
To load a module all we have to do is

```
import module
import module as key
```

We can also load directly a file containing some functions

```
from file import method
from file import method1, method2, method3, ...
```

This way is more useful when we want to break our code in multiple files.

Use of external modules

The System module adds functions that allow us to interact with the OS.

```
import sys
print(sys.path)
```

Loads all methods included in the **sys** package to call a method we need to refer to **sys**

```
import sys as s
print(s.path)
```

Loads all methods included in the **sys** package to call a method we can to refer to **sys** as **s**

Use of external modules

Before loading the **sys**

```
>>> print(sys.path)
---------------------------------------------------------------------
NameError                                 Traceback
(most recent call last)
<ipython-input-1-b35d1d099a2a> in <module>()
----> 1 sys.path

NameError: name 'sys' is not defined
```

After loading the **sys**

```
>>> import sys
>>> print(sys.path)
['', '/anaconda3/lib/python36.zip', '/anaconda3/lib/
python3.6', '/anaconda3/lib/python3.6/lib-dynload',
'/anaconda3/lib/python3.6/site-packages', '/
anaconda3/lib/python3.6/site-packages/aeosa', '/
anaconda3/lib/python3.6/site-packages/IPython/
extensions',  '/Users/flaviopinheiro/.ipython']
```

sys.path returns all the directories that are over watch by the Python interpreter.
For instance, when loading a package Python will search for the indicated package in each one of the listed directories.

Use of external modules

Import a particular method from a module

```
>>> from sys import path
>>> path
['', '/anaconda3/bin', '/anaconda3/lib/python36.zip', '/anaconda3/lib/
python3.6', '/anaconda3/lib/python3.6/lib-dynload', '/anaconda3/lib/
python3.6/site-packages', ‘/anaconda3/lib/python3.6/site-packages/
aeosa']
```

give it a name that makes your code less verbose

```
>>> from sys import path as p
>>> p
['', '/anaconda3/bin', '/anaconda3/lib/python36.zip', '/anaconda3/lib/
python3.6', '/anaconda3/lib/python3.6/lib-dynload', '/anaconda3/lib/
python3.6/site-packages', ‘/anaconda3/lib/python3.6/site-packages/
aeosa']
```

Use of external modules

csv - Write and read tabular data to and from delimited files;
datetime - Basic date and time types;
io - Core tools for working with streams;
json - Encode and decode the JSON format;
math - Mathematical functions (sin() etc.);
os - Miscellaneous operating system interfaces.
random - Generate pseudo-random numbers with various common distributions;
sqlite3 - A DB-API 2.0 implementation using SQLite 3.x.;
xml - Package containing XML processing modules;
zipfile - Read and write ZIP-format archive files;
zlib - Low-level interface to compression and decompression routines compatible with gzip.

Comprehensive list of all built-in modules - https://docs.python.org/2/py-modindex.html

How do I know what each module does?

https://docs.python.org/3/library/random.html

`random` — Generate pseudo-random numbers ¶

Source code: Lib/random.py

This module implements pseudo-random number generators for various distributions.

For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

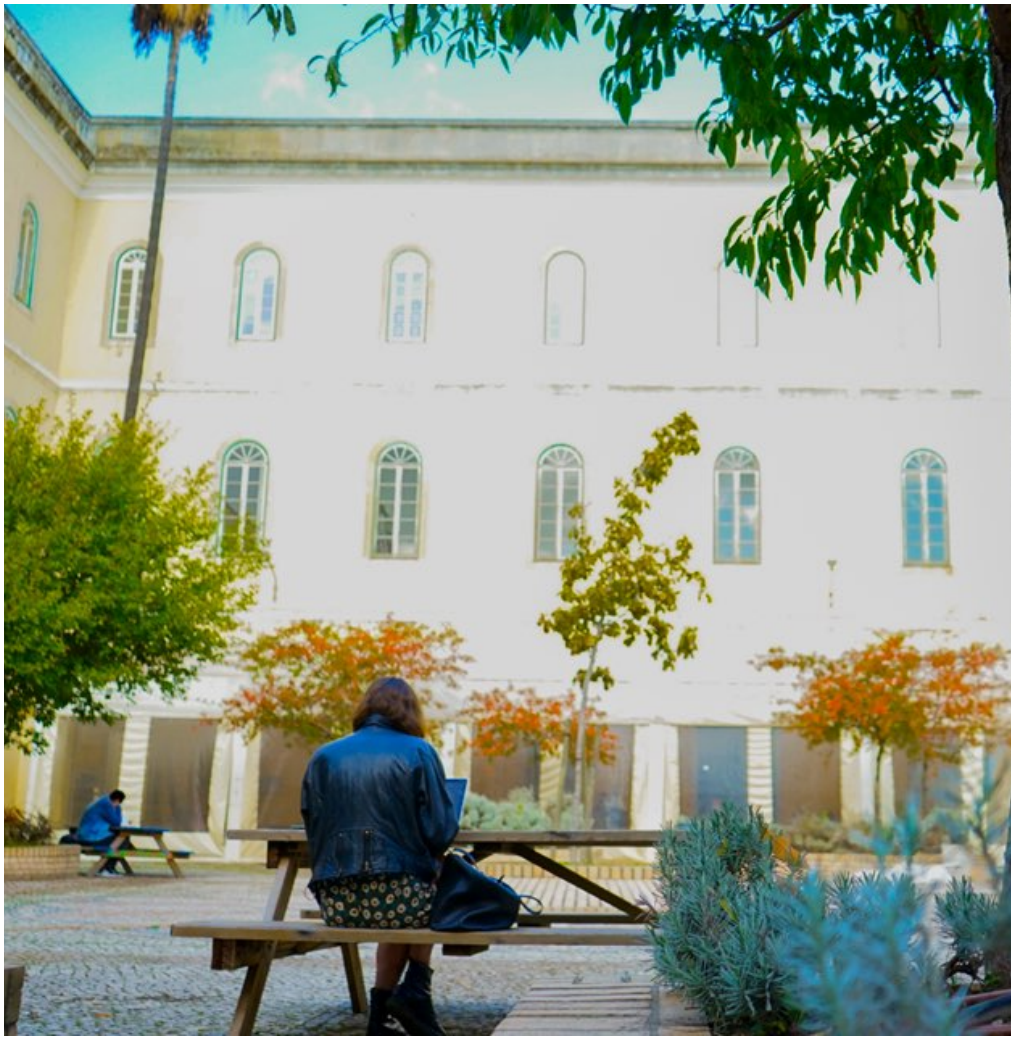
Almost all module functions depend on the basic function `random()`, which generates a random float uniformly in the semi-open range [0.0, 1.0). Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions supplied by this module are actually bound methods of a hidden instance of the `random.Random` class. You can instantiate your own instances of `Random` to get generators that don't share state.

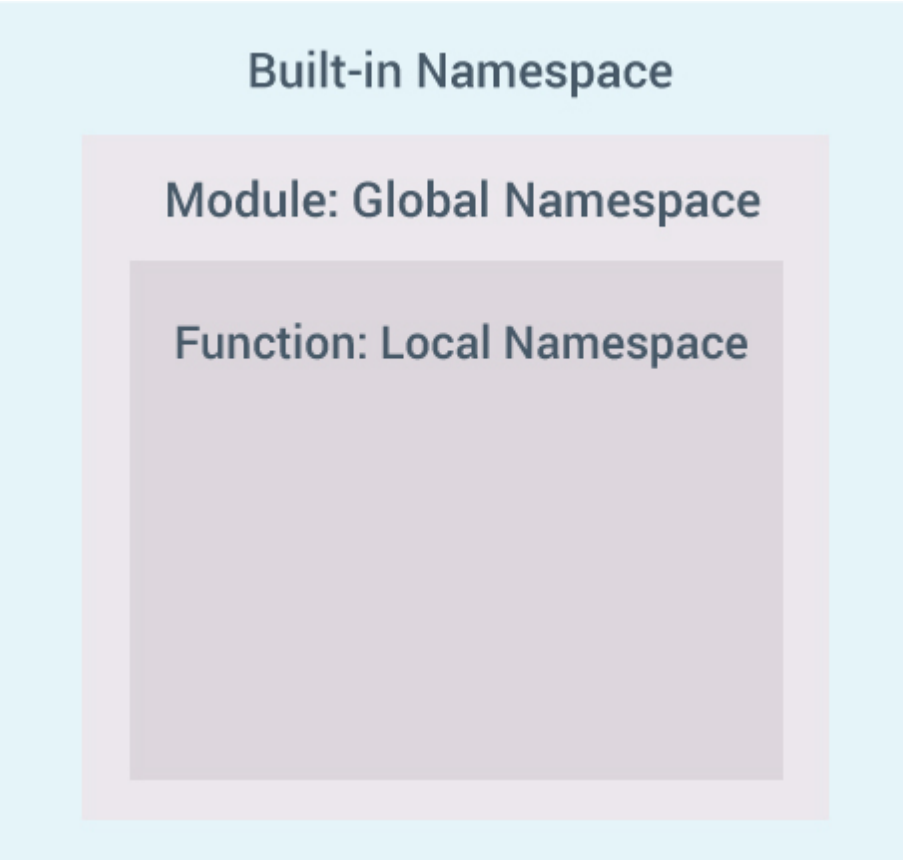

Data Capturing and Storage
Scrapy
BeautifulSoup
jupyter
Data Exploration and Cleaning
PANDAS
NumPy
Seaborn
matplotlib
Data Modeling and Predicting
SciPy
SM StatsModels Statistics in Python
scikits learn
TensorFlow
NetworkX
Final Product and Reporting
Seaborn
matplotlib
Flask
django

Lecture 3

- ~~Functions~~
- ~~Import Modules and Python STL~~
- NameSpace



NameSpace



A namespace represents a collection of symbols/names that reference some object.

Built-in Names are the names of the predefined objects recognised by your python interpreter (e.g., map, list, tuple, or dict).
You can list the objects in the built-in namespace by running **dir(_ _builtins_ _)**.

Global Names include any user-defined names (e.g., assignment of a variable, function name, etc) declared in the body of the program. You can check the global names by running **globals()**.

Local Names are the names that are defined inside a function, and cannot be access in the Global namespace. They have a limited lifespan. You can check the local names by running **locals()**.

yes names have a life too, deal with it!

Scope

Variables can have Global or Local in Scope

Global

Can be accessed throughout the entire program body by all functions

Scope

Variables can have Global or Local in Scope

Global

Can be accessed throughout the entire program body by all functions

Local

Variables can only accessed only inside the function were they have been declared.

Scope

Variables can have Global or Local in Scope

Global

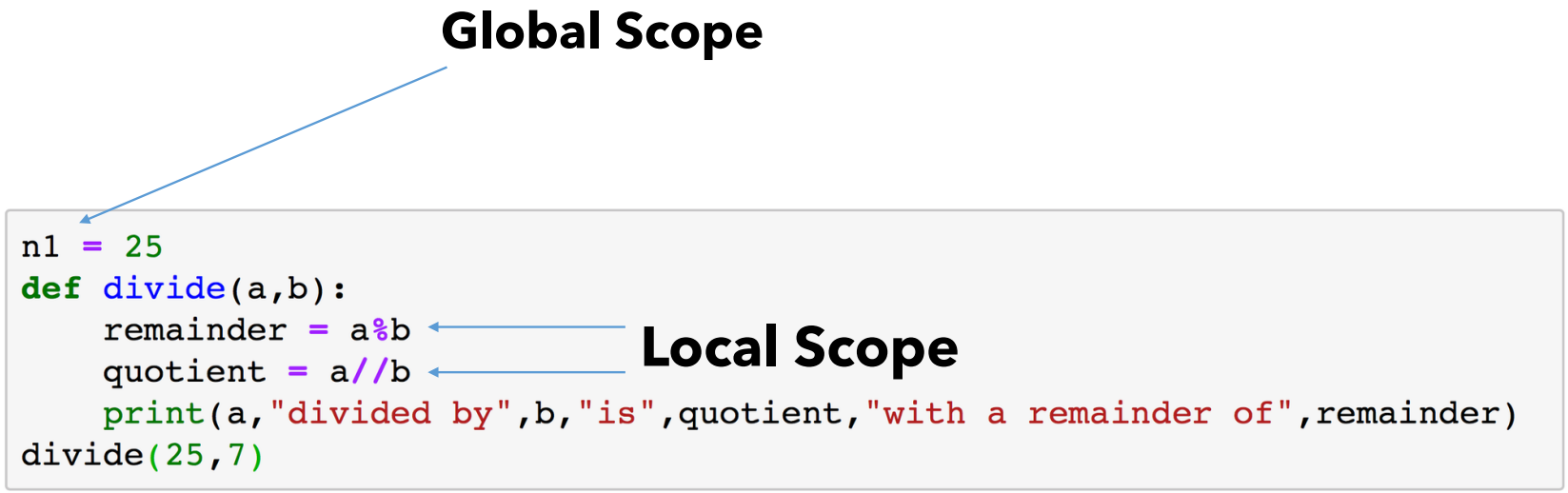
Can be accessed throughout the entire program body by all functions

Local

Variables can only accessed only inside the function were they have been declared.

knowing the right scope for your variables is key for debugging!

Scope




```
25 divided by 7 is 3 with a remainder of 4
```

Scope

Global Scope

```
animal = 'fruitbat'
def print_global():
    print('inside print_global:', animal)
print_global()
```

```
inside print_global: fruitbat
```

Local Scope

```
animal = 'fruitbat'
def change_and_print_global():
    print('inside change_and_print_global:', animal)
    animal = 'wombat'
    print('after the change:', animal)
change_and_print_global()
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-49-23d8bddaaa74> in <module>()
      4     animal = 'wombat'
      5     print('after the change:', animal)
----> 6 change_and_print_global()

<ipython-input-49-23d8bddaaa74> in change_and_print_global()
      1 animal = 'fruitbat'
      2 def change_and_print_global():
----> 3     print('inside change_and_print_global:', animal)
      4     animal = 'wombat'
      5     print('after the change:', animal)

UnboundLocalError: local variable 'animal' referenced before assignment
```

we are trying to print animal before it was defined locally

Scope

Global Scope

local Scope

ensures we are using the global variable inside function

```
animal = 'fruitbat'
def change_and_print_global():
    animal = 'wombat'
    print('after the change:', animal)
change_and_print_global()
```

```
after the change: wombat
```

```
animal = 'fruitbat'
def change_and_print_global():
    global animal
    animal = 'wombat'
    print('inside change_and_print_global:', animal)
print(animal)
change_and_print_global()
print(animal)
```

```
fruitbat
inside change_and_print_global: wombat
wombat
```

What we will not Cover Here

Concepts in the readings that you should take a look at

- Objects (Classes, etc)
- iterators
- Decorators
- Generators
- Handling of errors and exceptions (try and except)


Next Week!
Hello
Pandas!