# TECNICHE DI RAPPRESENTAZIONE E MODELLIZZAZIONE DEI DATI

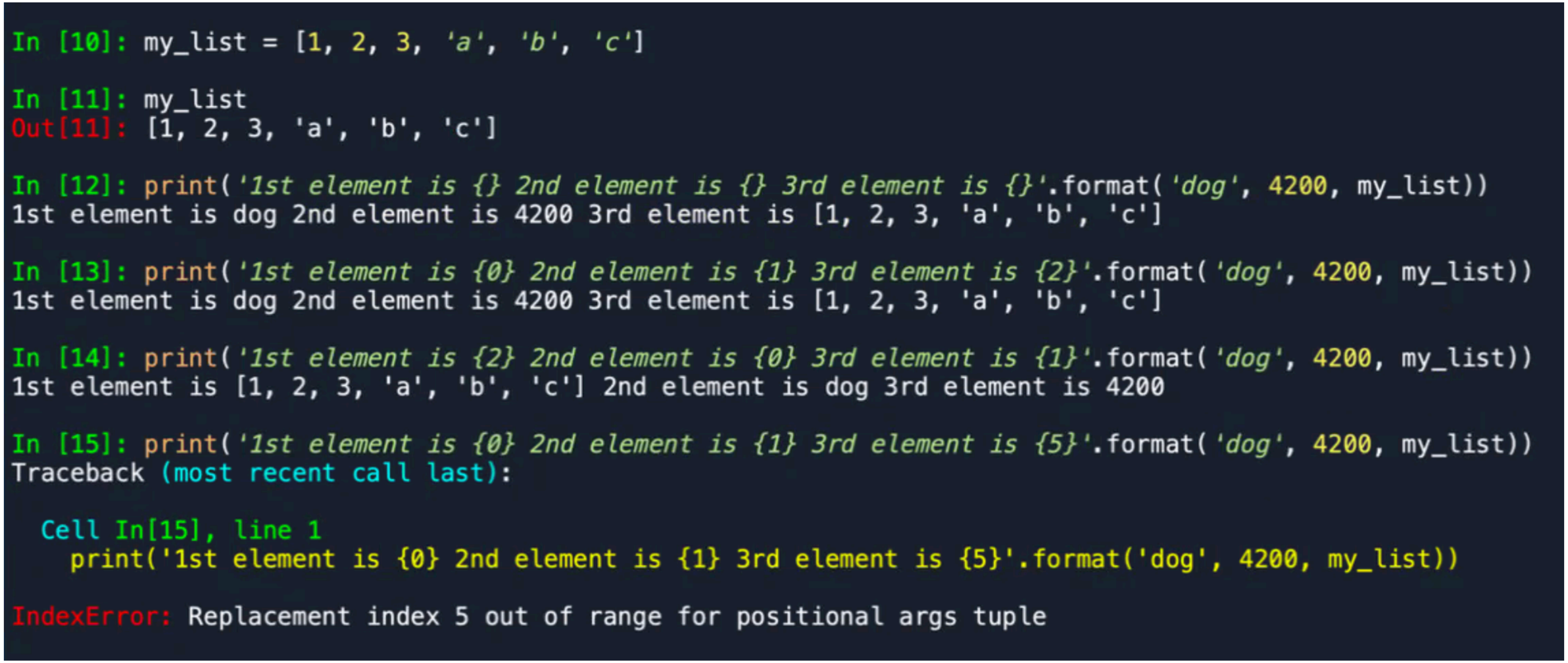
## — Part 1 —

**(2 CFU out of 6 total CFU)**

**Link moodle**: https://moodle2.units.it/course/view.php?id=11703

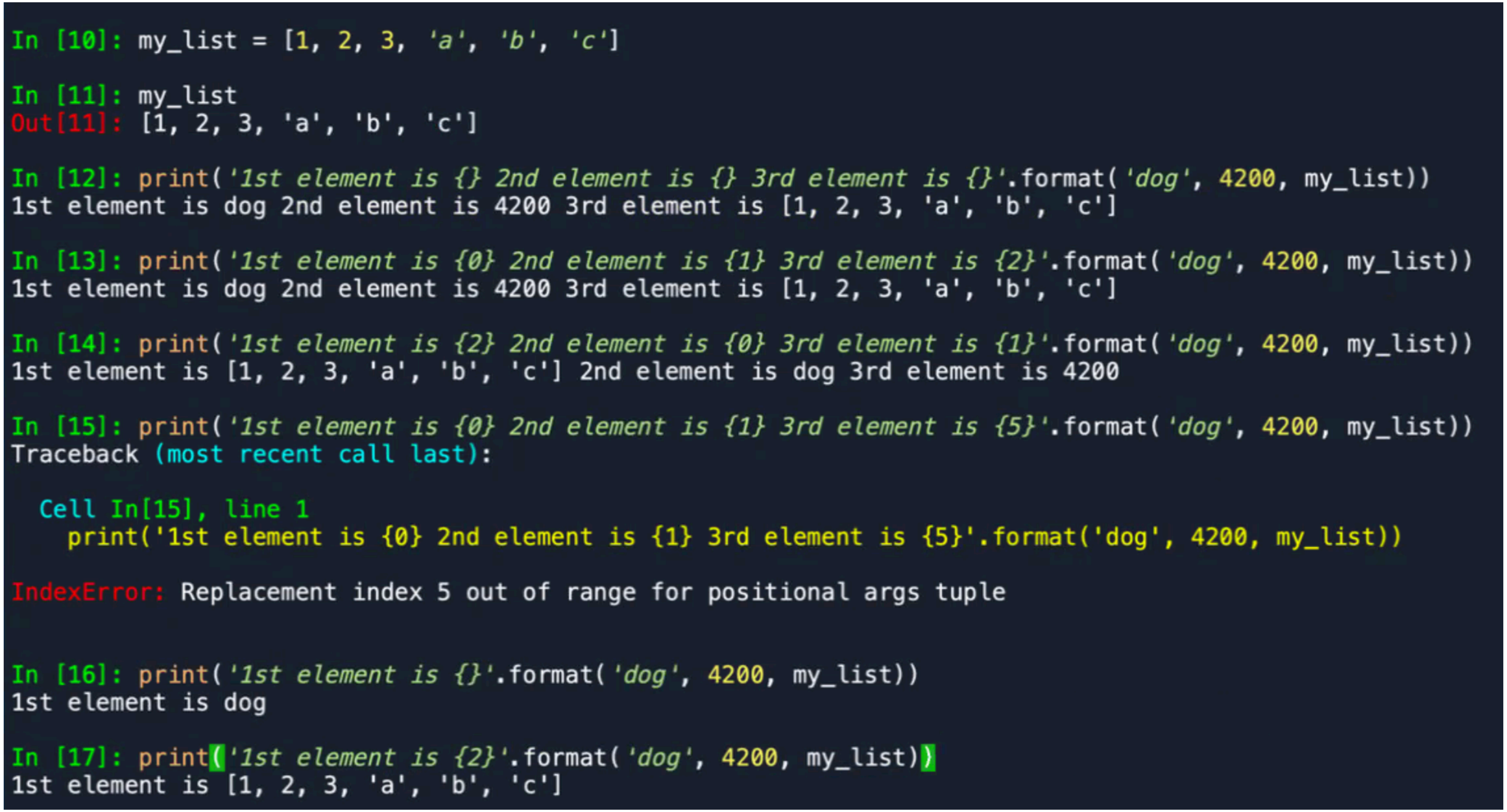
Teams code: 0ftoqj8

# Python: .format



```
In [10]: my_list = [1, 2, 3, 'a', 'b', 'c']

In [11]: my_list
Out[11]: [1, 2, 3, 'a', 'b', 'c']

In [12]: print('1st element is {} 2nd element is {} 3rd element is {}'.format('dog', 4200, my_list))
1st element is dog 2nd element is 4200 3rd element is [1, 2, 3, 'a', 'b', 'c']

In [13]: print('1st element is {0} 2nd element is {1} 3rd element is {2}'.format('dog', 4200, my_list))
1st element is dog 2nd element is 4200 3rd element is [1, 2, 3, 'a', 'b', 'c']

In [14]: print('1st element is {2} 2nd element is {0} 3rd element is {1}'.format('dog', 4200, my_list))
1st element is [1, 2, 3, 'a', 'b', 'c'] 2nd element is dog 3rd element is 4200

In [15]: print('1st element is {0} 2nd element is {1} 3rd element is {5}'.format('dog', 4200, my_list))
Traceback (most recent call last):

  Cell In[15], line 1
    print('1st element is {0} 2nd element is {1} 3rd element is {5}'.format('dog', 4200, my_list))

IndexError: Replacement index 5 out of range for positional args tuple
```

## Python: .format



```
In [10]: my_list = [1, 2, 3, 'a', 'b', 'c']

In [11]: my_list
Out[11]: [1, 2, 3, 'a', 'b', 'c']

In [12]: print('1st element is {} 2nd element is {} 3rd element is {}'.format('dog', 4200, my_list))
1st element is dog 2nd element is 4200 3rd element is [1, 2, 3, 'a', 'b', 'c']

In [13]: print('1st element is {0} 2nd element is {1} 3rd element is {2}'.format('dog', 4200, my_list))
1st element is dog 2nd element is 4200 3rd element is [1, 2, 3, 'a', 'b', 'c']

In [14]: print('1st element is {2} 2nd element is {0} 3rd element is {1}'.format('dog', 4200, my_list))
1st element is [1, 2, 3, 'a', 'b', 'c'] 2nd element is dog 3rd element is 4200

In [15]: print('1st element is {0} 2nd element is {1} 3rd element is {5}'.format('dog', 4200, my_list))
Traceback (most recent call last):

  Cell In[15], line 1
    print('1st element is {0} 2nd element is {1} 3rd element is {5}'.format('dog', 4200, my_list))

IndexError: Replacement index 5 out of range for positional args tuple


In [16]: print('1st element is {}'.format('dog', 4200, my_list))
1st element is dog

In [17]: print('1st element is {2}'.format('dog', 4200, my_list))
1st element is [1, 2, 3, 'a', 'b', 'c']
```

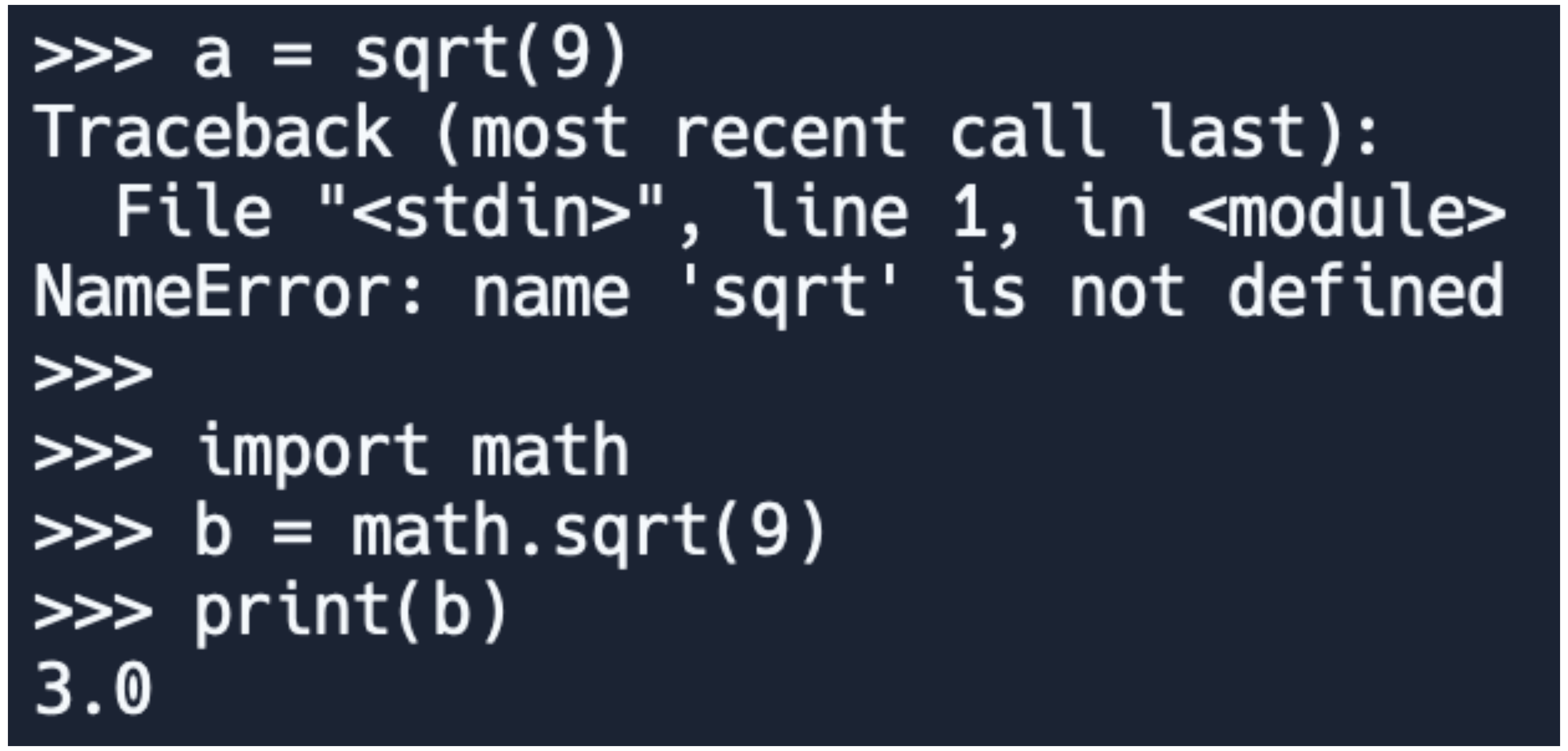
## Python: modules

Python features several **modules**.

Each module contains instructions, constants and functions already available for the users.

Modules (like libraries) can be imported through the **import** statement.

Instructions of the module are available within the module **namespace** (i.e. the name of the module)



```
>>> a = sqrt(9)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sqrt' is not defined
>>>
>>> import math
>>> b = math.sqrt(9)
>>> print(b)
3.0
```

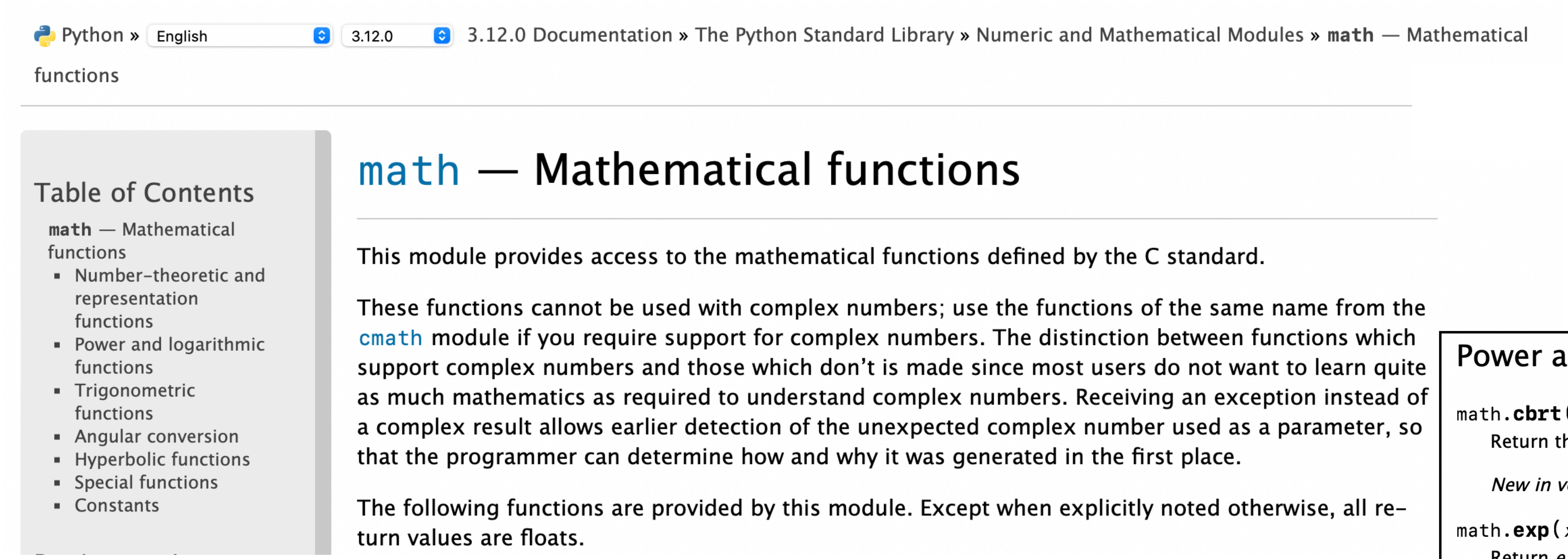
# Python: modules

Python features several **modules**.

Each module contains instructions, constants and functions already available for the users.

Modules (like libraries) can be imported through the **import** statement.

Instructions of the module are available within the module **namespace** (i.e. the name of the module)



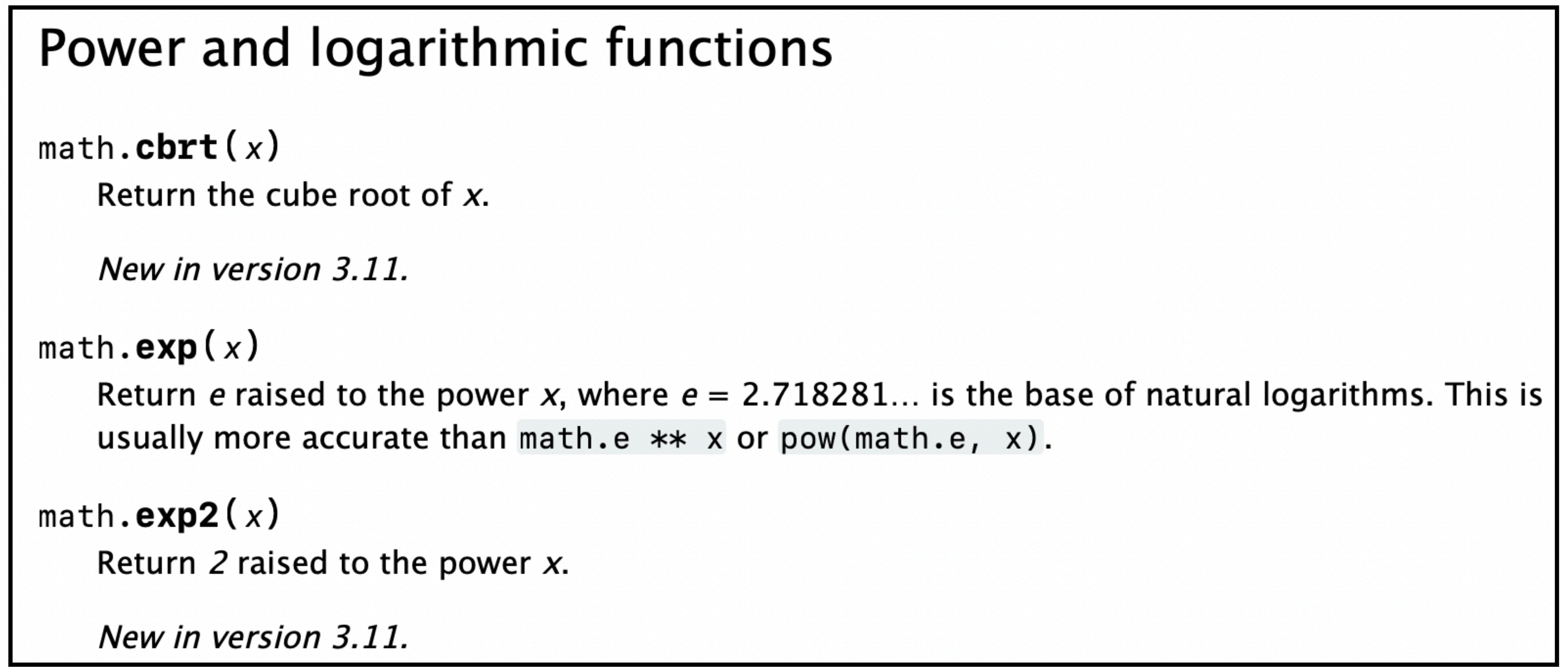
Python » English 3.12.0 3.12.0 Documentation » The Python Standard Library » Numeric and Mathematical Modules » **math** — Mathematical functions

## math — Mathematical functions

This module provides access to the mathematical functions defined by the C standard.

These functions cannot be used with complex numbers; use the functions of the same name from the `cmath` module if you require support for complex numbers. The distinction between functions which support complex numbers and those which don't is made since most users do not want to learn quite as much mathematics as required to understand complex numbers. Receiving an exception instead of a complex result allows earlier detection of the unexpected complex number used as a parameter, so that the programmer can determine how and why it was generated in the first place.

The following functions are provided by this module. Except when explicitly noted otherwise, all return values are floats.



### Power and logarithmic functions

`math.cbrt(x)`
Return the cube root of *x*.

*New in version 3.11.*

`math.exp(x)`
Return *e* raised to the power *x*, where *e* = 2.718281… is the base of natural logarithms. This is usually more accurate than `math.e ** x` or `pow(math.e, x)`.

`math.exp2(x)`
Return *2* raised to the power *x*.

*New in version 3.11.*

**https://docs.python.org/3/library/math.html**

# Python: modules

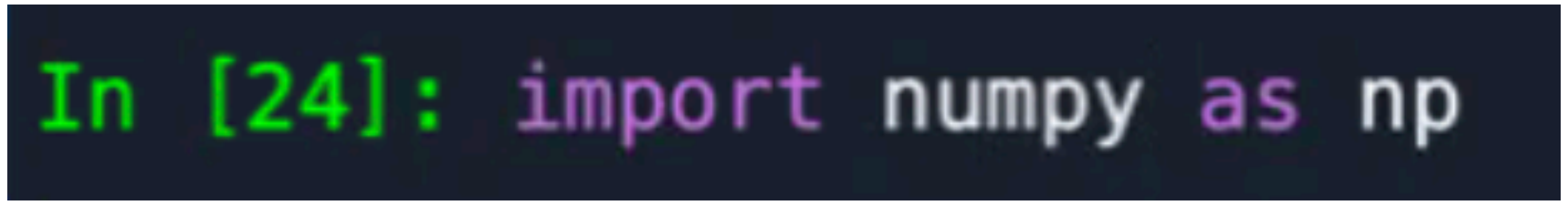
Python features several **modules**.

Each module contains instructions, constants and functions already available for the users.

Modules (like libraries) can be imported through the **import** statement.

Instructions of the module are available within the module **namespace** (i.e. the name of the module)

It is possible to map the namespace of the module we are importing into an **alias**:



```
In [24]: import numpy as np
```

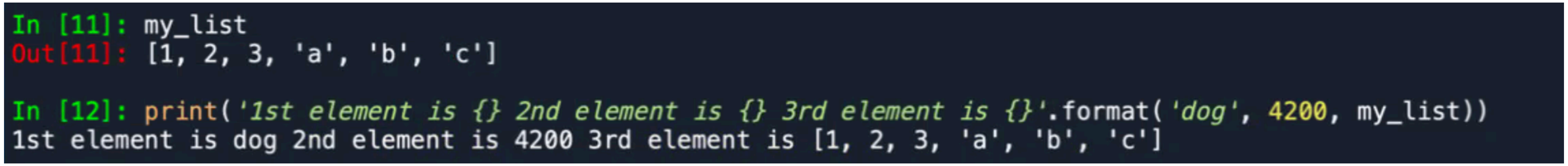
https://numpy.org/doc/stable/

NumPy

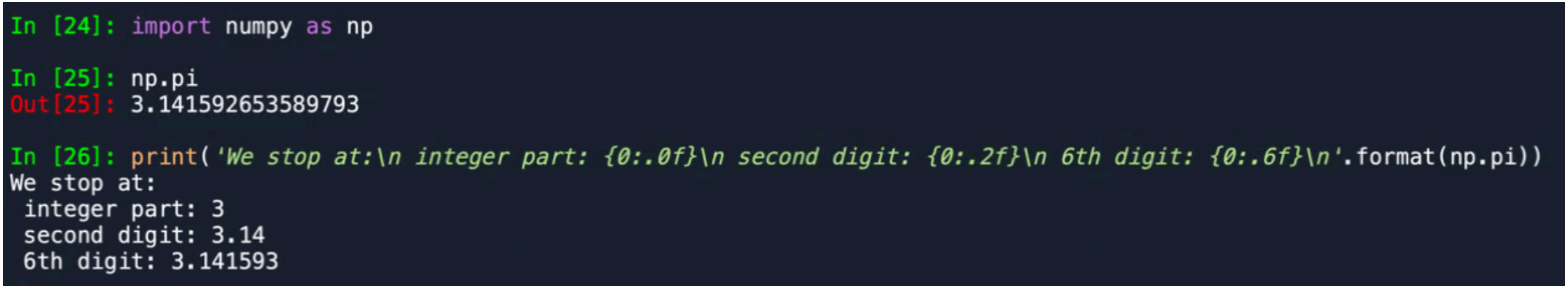
The fundamental package for scientific computing with Python

LATEST RELEASE: NUMPY 1.26. VIEW ALL RELEASES

# Python: .format (additional features)



```
In [11]: my_list
Out[11]: [1, 2, 3, 'a', 'b', 'c']

In [12]: print('1st element is {} 2nd element is {} 3rd element is {}'.format('dog', 4200, my_list))
1st element is dog 2nd element is 4200 3rd element is [1, 2, 3, 'a', 'b', 'c']
```



```
In [24]: import numpy as np

In [25]: np.pi
Out[25]: 3.141592653589793

In [26]: print('We stop at:\n integer part: {0:.0f}\n second digit: {0:.2f}\n 6th digit: {0:.6f}\n'.format(np.pi))
We stop at:
 integer part: 3
 second digit: 3.14
 6th digit: 3.141593
```

## Python: modules

Exercises

Write a Python script that displays the following:

- — the factorial of 39
- — the number *e*
- — the logarithm (in base e, 2, 3 and 10) of 1500
- — a random number in the range [0, 1) ( hint: use the function *random* )
- — a random float in the range [3.5, 13.5]
- — an integer in the range [5, 50]
- — an even integer in the range [6, 60]
- — compute the mode of the list [1, 1, 2, 3, 3, 3, 3, 4]
- — compute the mean of the list [1, 2, 3, 4, 5, 6, 7, 8]

**Useful webpages:**

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

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

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

## Python functions

A **function** is a block of code which corresponds to a set of instructions and only runs when it is called.

Main **aim** of functions: split the script in logical blocks.

**Syntax** of functions:

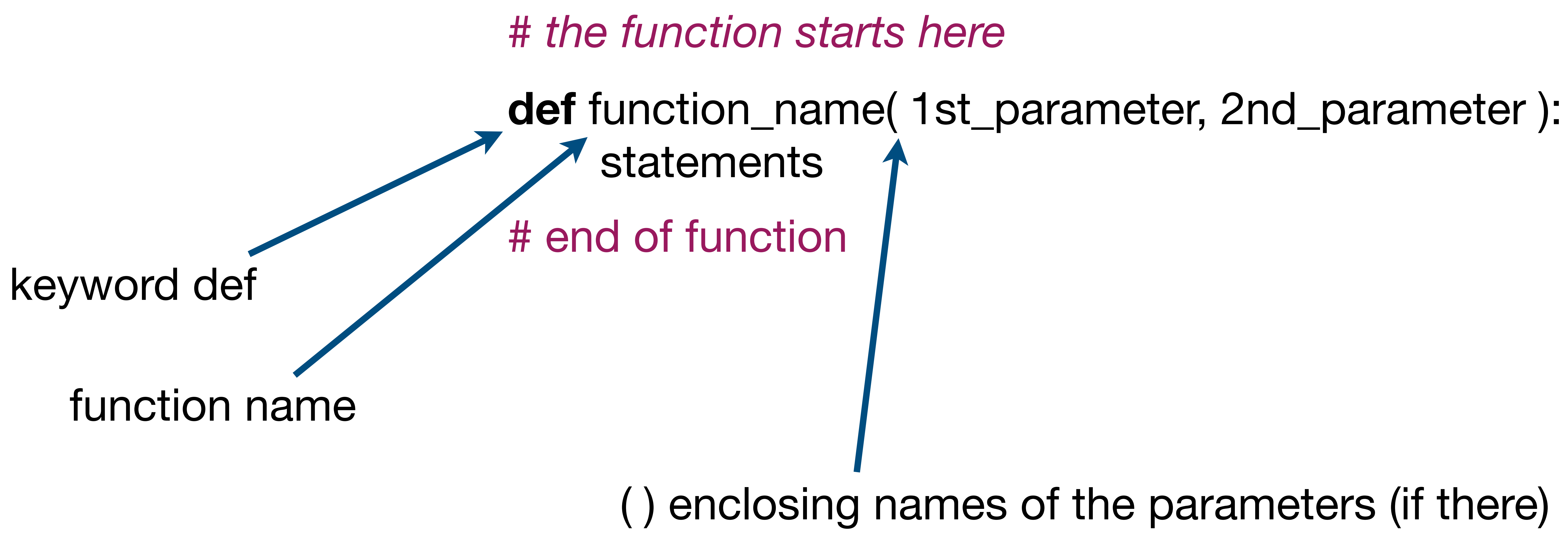
```
# the function starts here
def function_name( 1st_parameter, 2nd_parameter ):
    statements
# end of function
```

# Python functions

A **function** is a block of code which corresponds to a set of instructions and only runs when it is called.

Main **aim** of functions: split the script in logical blocks.

**Syntax** of functions:



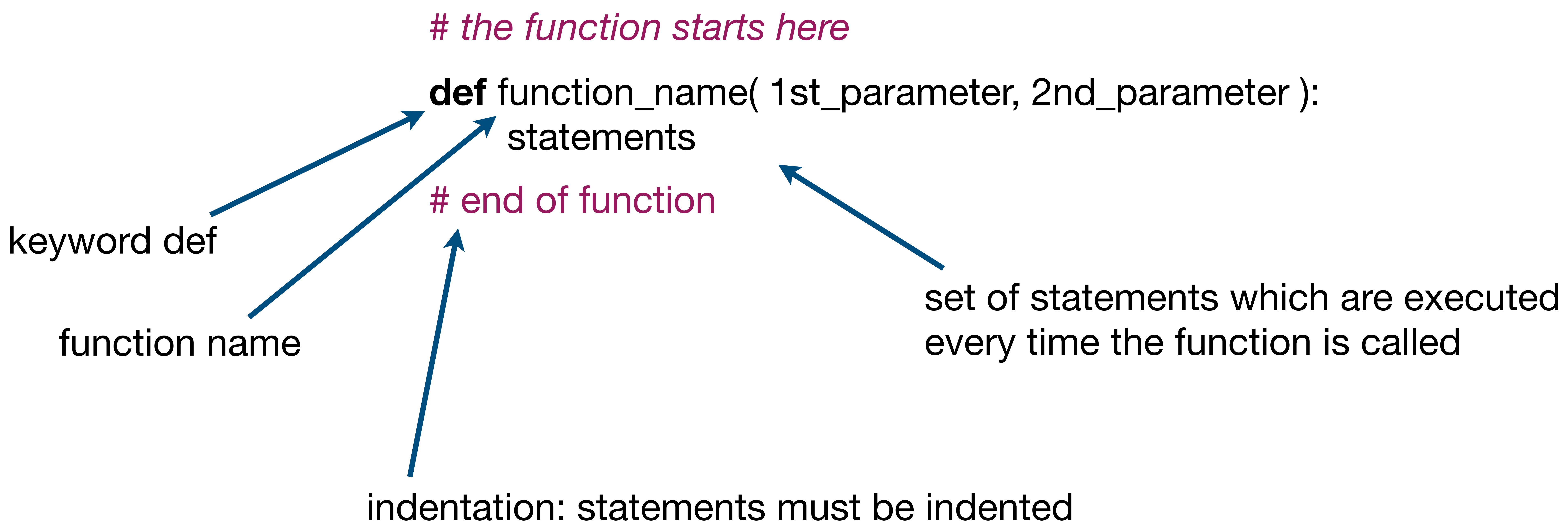
```
# the function starts here
def function_name( 1st_parameter, 2nd_parameter ):
    statements
# end of function
```

# Python functions

A **function** is a block of code which corresponds to a set of instructions and only runs when it is called.

Main **aim** of functions: split the script in logical blocks.

**Syntax** of functions:



```
# the function starts here
def function_name( 1st_parameter, 2nd_parameter ):
    statements
# end of function
```

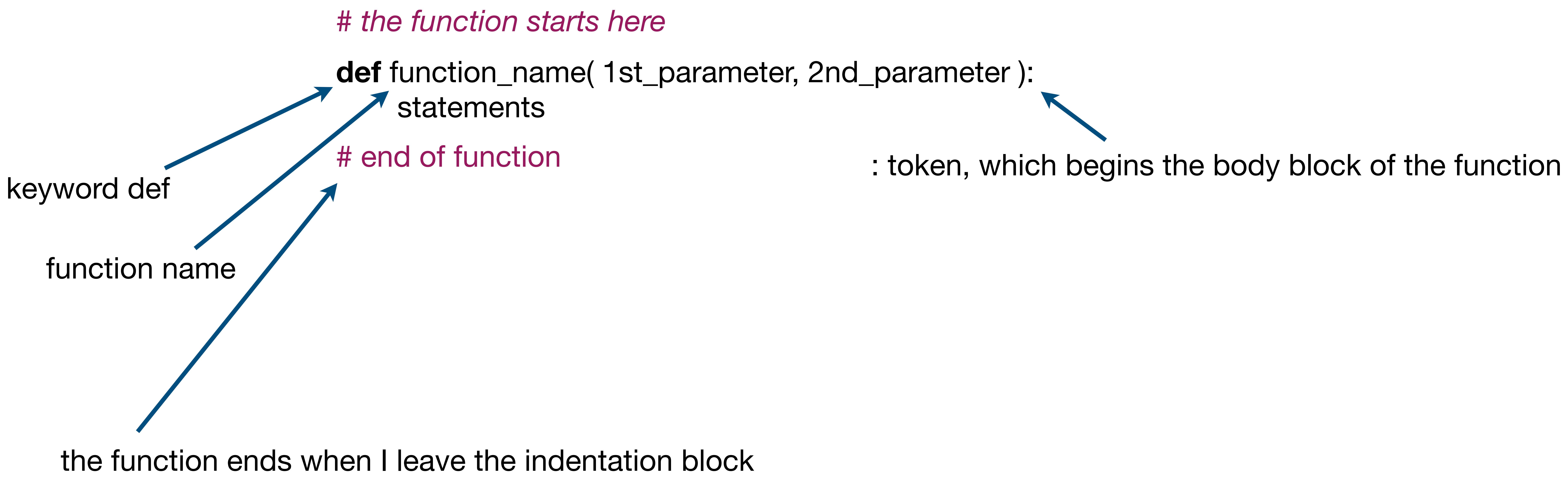
keyword def
function name
indentation: statements must be indented
set of statements which are executed every time the function is called

# Python functions

A **function** is a block of code which corresponds to a set of instructions and only runs when it is called.

Main **aim** of functions: split the script in logical blocks.

**Syntax** of functions:



```
# the function starts here
def function_name( 1st_parameter, 2nd_parameter ):
    statements
# end of function
```

keyword def

function name

: token, which begins the body block of the function

the function ends when I leave the indentation block

## Python functions

A script can have as many functions as the user wants.

Functions can be called only after they’ve been defined (i.e., below).

Functions can call other functions.

Names of the function arguments are independent of those outside the function.

Functions help the script readability.

# Python functions

Functions can be broadly split into two subgroups:

**void functions** and **functions returning values**

Examples of void functions:

```
>>> def greetings():
...     print("Hello!")
...     print("Have a nice day!")
...
>>> greetings()
Hello!
Have a nice day!
>>>
```

## Python functions

Functions can be broadly split into two subgroups:

**void functions** and **functions returning values**

Examples of void functions:

```
>>> def greetings():
...     print("Hello!")
...     print("Have a nice day!")
...
>>> greetings()
Hello!
Have a nice day!
>>>
```

```
>>> def greetings(name):
...     print("Hello, {}".format(name))
...     print("Have a nice day!")
...
>>> greetings("John")
Hello, John
Have a nice day!
>>> greetings("Anna")
Hello, Anna
Have a nice day!
```

**Void functions** perform an action but do not return any computed value to the caller (they actually return None).

# Python functions

Functions can be broadly split into two subgroups:

**void functions** and **functions returning values**

Example of a function returning a value:

```
>>> def sum_of_numbers(a, b):
...     result = a + b
...     return result
...
>>> first_number = 3
>>> second_number = 5
>>> final_result = sum_of_numbers(first_number, second_number)
>>> print("The result is: {}".format(final_result))
The result is: 8
```

# Python functions

Functions can be broadly split into two subgroups:

**void functions** and **functions returning values**

Example of a function returning a value:

```
>>> def sum_of_numbers(a, b):
...     result = a + b
...     return result
...
>>> first_number = 3
>>> second_number = 5
>>> final_result = sum_of_numbers(first_number, second_number)
>>> print("The result is: {}".format(final_result))
The result is: 8
```

**function header**
**temporary variable**
**return temporary variable**

Variables created within functions only exist within them

Function arguments are **local variables**, too

# Python functions

Some examples.

```python
import numpy as np

#Example 1: function that computes the volume of a sphere
def sphere_vol(radius):

    vol = (4./3.)*np.pi*(radius**3)

    return vol

```

```python
# ===========================================================================

#Main program: function calls

#We set radius value to 2 and call the first function
r = 2
v = sphere_vol(r)
print('The volume of a sphere of radius {} is {}'.format(r, v))

```

```
(base) milena:Desktop milenavalentini$ python Function_examples.py
The volume of a sphere of radius 2 is 33.510321638291124
```

# Python functions

```python
#Example 2: same function as before, but the value of radius defaults to 1 if not provided
def sphere_vol_default(radius = 1):

    print('This is example number 2: the chosen value for the radius is {}'.format(radius))

    vol = (4./3.)*np.pi*(radius**3)

    return vol

#We set radius value to 2 and call the second function: same result as before
r = 2
v = sphere_vol_default(r)
print('The volume of a sphere of radius {} is {}'.format(r, v))

#We give no radius value and call the second function: it defaults to 1
#WARNING: first function would have returned error. Try this.
v = sphere_vol_default()
print('The volume of a sphere of radius {} is {}'.format('?', v))
```

```
(base) milena:Desktop milenavalentini$ python Function_examples.py
The volume of a sphere of radius 2 is 33.510321638291124
This is example number 2: the chosen value for the radius is 2
The volume of a sphere of radius 2 is 33.510321638291124
This is example number 2: the chosen value for the radius is 1
The volume of a sphere of radius ? is 4.1887902047863905
```

# Python functions

```python
#Example 3: function with no parameters. This function returns the volume of a sphere of radius 5 (hardcoded) and takes no input.
def sphere_vol_no_input():

    radius = 5

    print('This is example number 3: the chosen value for the radius is {}'.format(radius))

    vol = (4./3.)*np.pi*(radius**3)

    return vol
```

```python
#We call the third function with no arguments
v = sphere_vol_no_input()
print('The volume of a sphere of radius {} is {}'.format('?', v))

#WARNING: variable names used in main program and in functions are separate!
#We define a variable named 'radius' exactly as the one used in the functions
#Then we call function number 3 without arguments. Even if we define a variable 'radius' before calling the function, it is ignored and the value inside the function is used
radius = 20
v = sphere_vol_no_input()
print('The volume of a sphere of radius {} is {}'.format('?', v))
```

```
(base) milena:Desktop milenavalentini$ python Function_examples.py
This is example number 3: the chosen value for the radius is 5
The volume of a sphere of radius ? is 523.5987755982989
This is example number 3: the chosen value for the radius is 5
The volume of a sphere of radius ? is 523.5987755982989
```

# Python functions

```python
#Example 4: void function
def sphere_vol_void(radius = 1):

    print('This is example number 4: the chosen value for the radius is {}'.format(radius))

    vol = (4./3.)*np.pi*(radius**3)

# ============================================================================

#Main program: function calls

#We set radius value to 2 and call the fourth function: returns None
r = 2
v = sphere_vol_void(r)
print('The volume of a sphere of radius {} is {}'.format(r, v))
```

```
(base) milena:Desktop milenavalentini$ python Function_examples.py
This is example number 4: the chosen value for the radius is 2
The volume of a sphere of radius 2 is None
(base) milena:Desktop milenavalentini$
```

# Python functions

The “““ Test here to document ””” string within a function is called **docstring**.
Placed at the very top of the function body, it acts as a documentation on the function.
This string gets printed out when you call help( ) on the function.

```
>>> def sum(a, b):
...     """Sum function: takes two values as arguments and returns its sum."""
...     result = a + b
...     return result
...
>>> help(sum)
```

```
Help on function sum in module __main__:

sum(a, b)
    Sum function: takes two values as arguments and returns its sum.
```

## Python functions

Example.

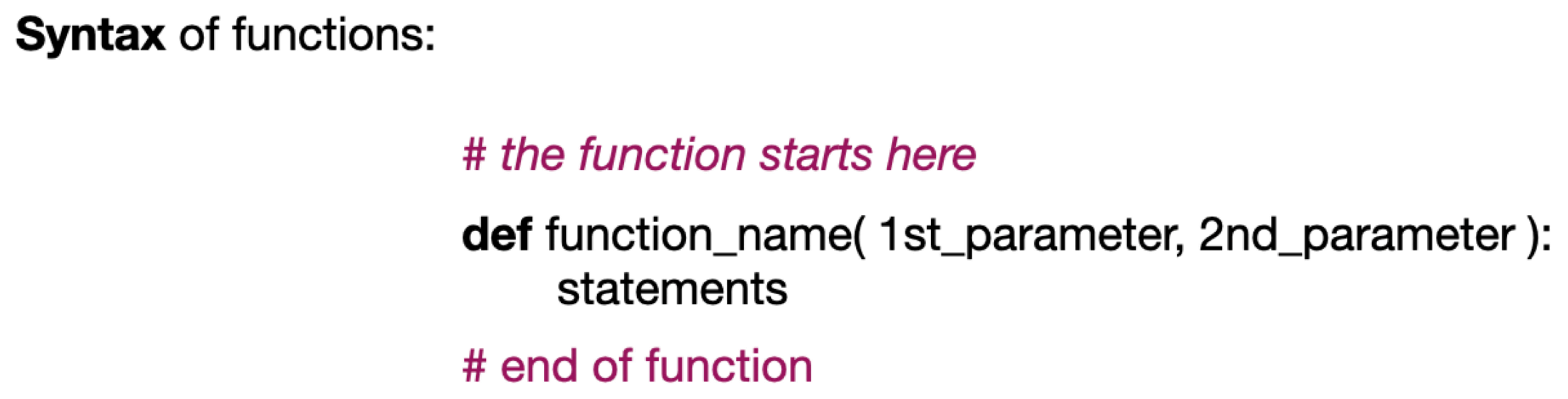
```python
import numpy as np

#Example 1: function that computes the volume of a sphere
def sphere_vol(radius):
    """
    This function computes the volume of a sphere given its radius. It also provides an example of function documentation for the creation of a manual.

    Parameters
    ----------
    radius : float
        The radius of the sphere for which the volume has to be computed.

    Returns
    -------
    vol: float
        The volume of the sphere.
    """

    vol = (4./3.)*np.pi*(radius**3)

    return vol

```

## Python functions

Exercises.

Write a function that computes the volume of a cylinder given radius and height. Make it so that radius and height default to 1 if they are not given.

Write another function that prints a sentence (‘Hello World’) and takes no input. Make it a void function.



**Syntax** of functions:

```
# the function starts here
def function_name( 1st_parameter, 2nd_parameter ):
      statements
# end of function
```

## Python functions

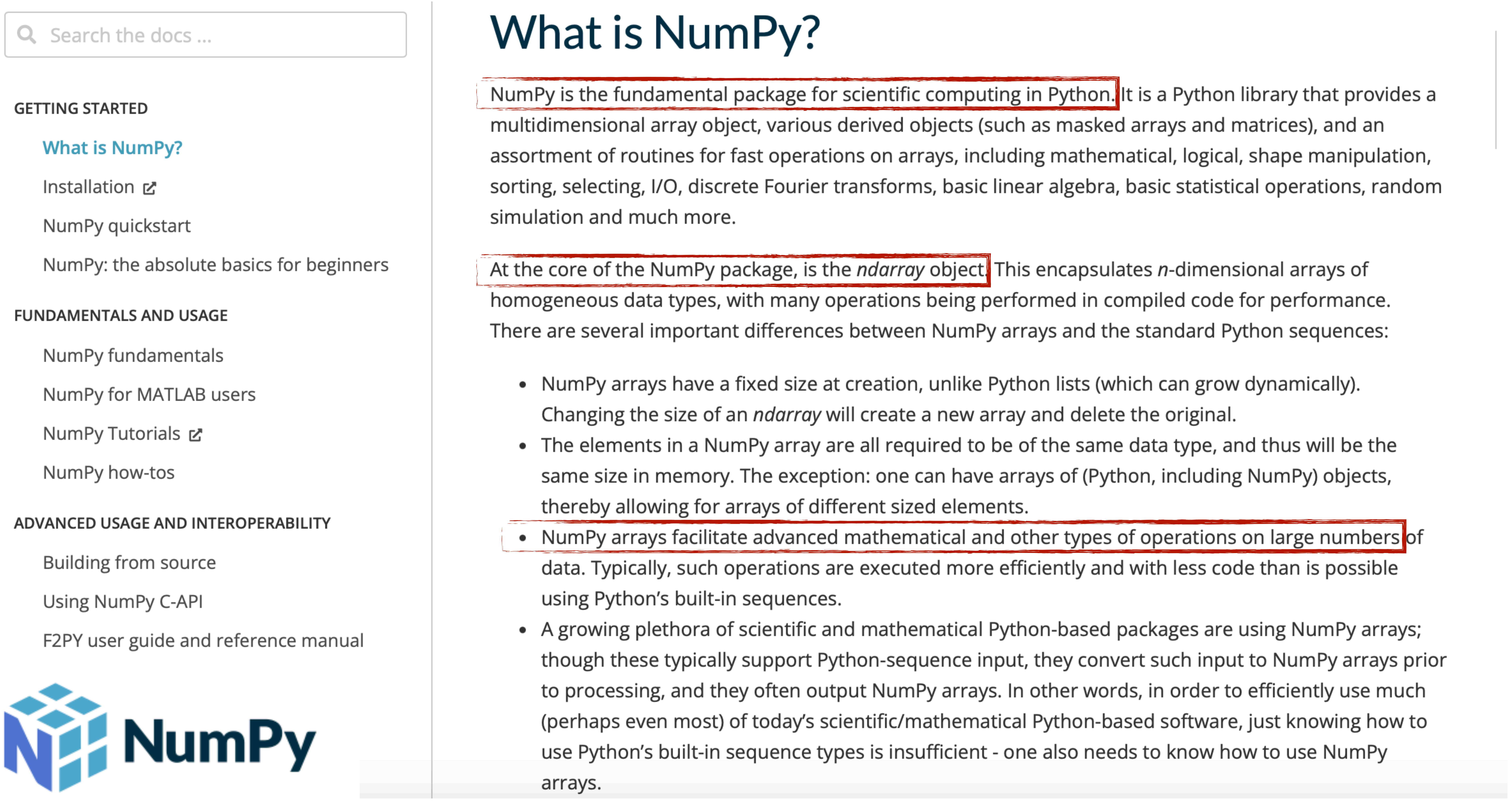
Exercises.

Write a function that computes the volume of a cylinder given radius and height. Make it so that radius and height default to 1 if they are not given.

Write another function that prints a sentence (‘Hello World’) and takes no input. Make it a void function.

```python
import numpy as np

#We create the first function
def cyl_vol(radius = 1, height = 1):

    vol = np.pi*(radius**2)*height

    return vol

#We create the second function
def print_sentence():

    print('Hello World')
```

# Python: numpy



Search the docs ...

GETTING STARTED

- What is NumPy?
- Installation
- NumPy quickstart
- NumPy: the absolute basics for beginners

FUNDAMENTALS AND USAGE

- NumPy fundamentals
- NumPy for MATLAB users
- NumPy Tutorials
- NumPy how-tos

ADVANCED USAGE AND INTEROPERABILITY

- Building from source
- Using NumPy C-API
- F2PY user guide and reference manual

NumPy

## What is NumPy?

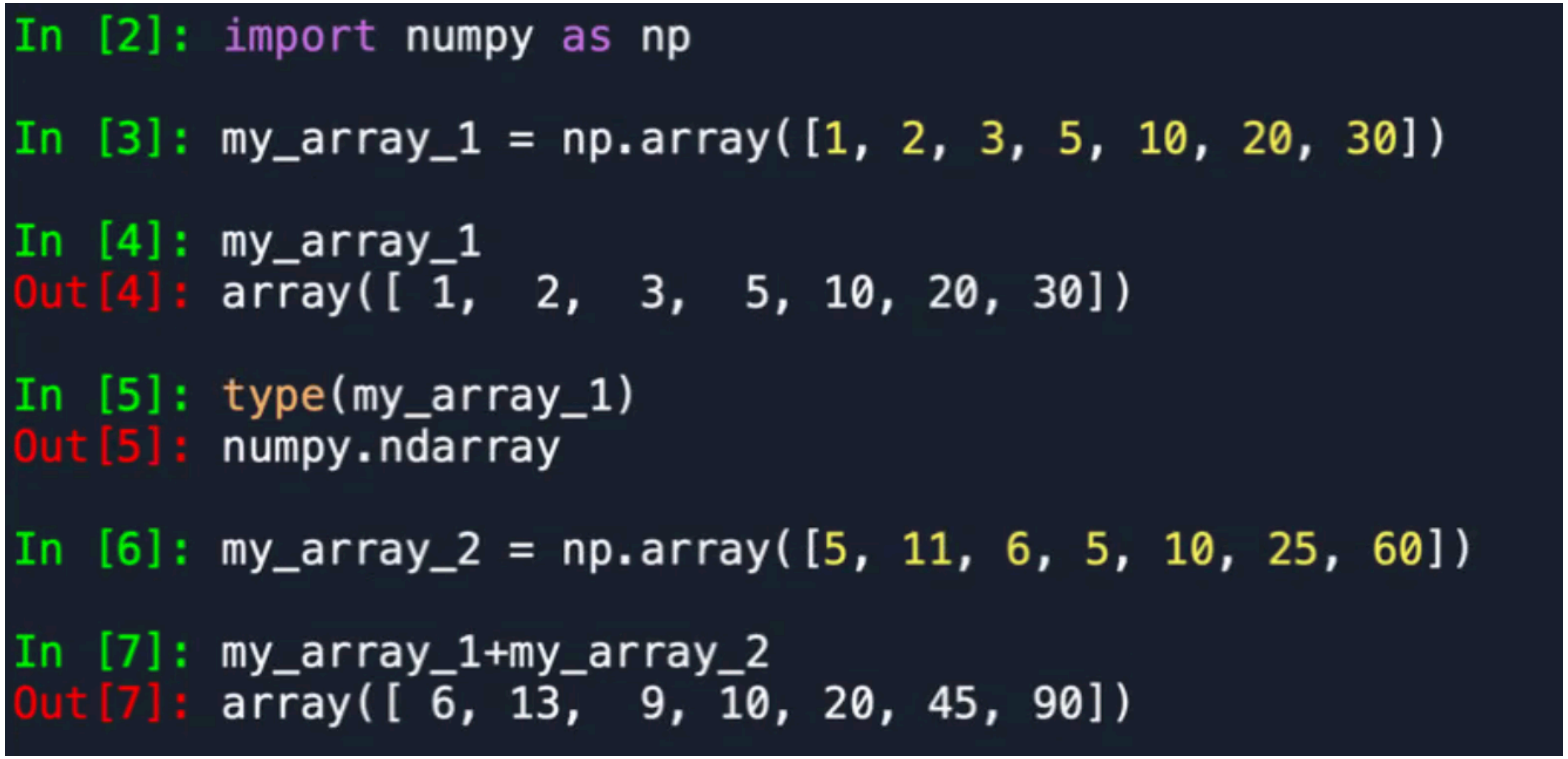
NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the *ndarray* object. This encapsulates *n*-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an *ndarray* will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

# Python: numpy

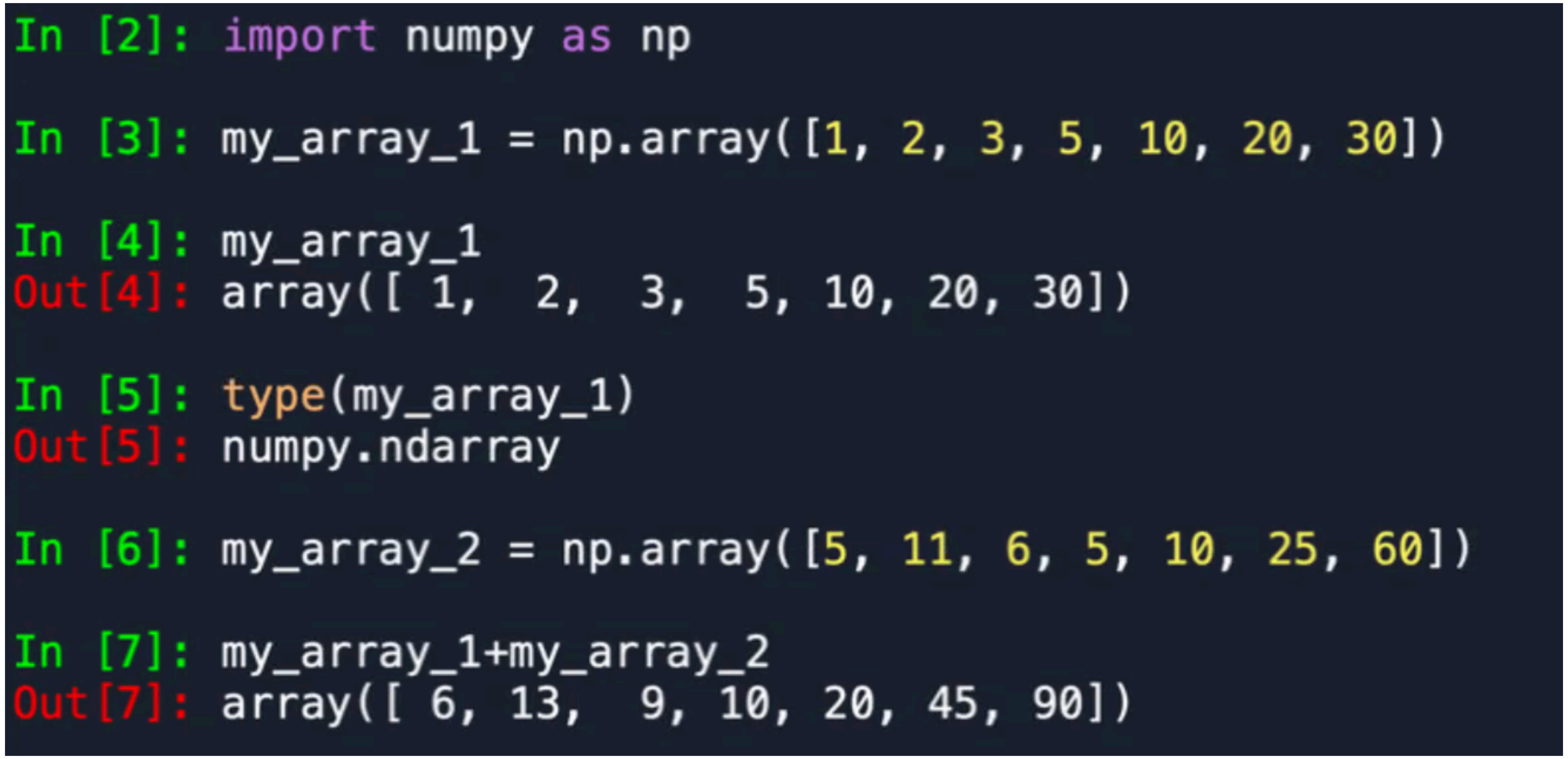
The N-dim array (ndarray) object and operations with arrays



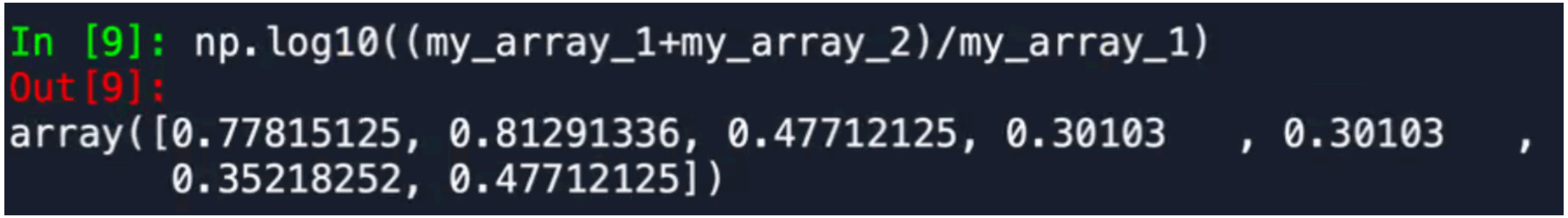
```
In [2]: import numpy as np

In [3]: my_array_1 = np.array([1, 2, 3, 5, 10, 20, 30])

In [4]: my_array_1
Out[4]: array([ 1,  2,  3,  5, 10, 20, 30])

In [5]: type(my_array_1)
Out[5]: numpy.ndarray

In [6]: my_array_2 = np.array([5, 11, 6, 5, 10, 25, 60])

In [7]: my_array_1+my_array_2
Out[7]: array([ 6, 13,  9, 10, 20, 45, 90])
```

# Python: numpy

The N-dim array (ndarray) object and operations with arrays



```
In [2]: import numpy as np

In [3]: my_array_1 = np.array([1, 2, 3, 5, 10, 20, 30])

In [4]: my_array_1
Out[4]: array([ 1,  2,  3,  5, 10, 20, 30])

In [5]: type(my_array_1)
Out[5]: numpy.ndarray

In [6]: my_array_2 = np.array([5, 11, 6, 5, 10, 25, 60])

In [7]: my_array_1+my_array_2
Out[7]: array([ 6, 13,  9, 10, 20, 45, 90])
```

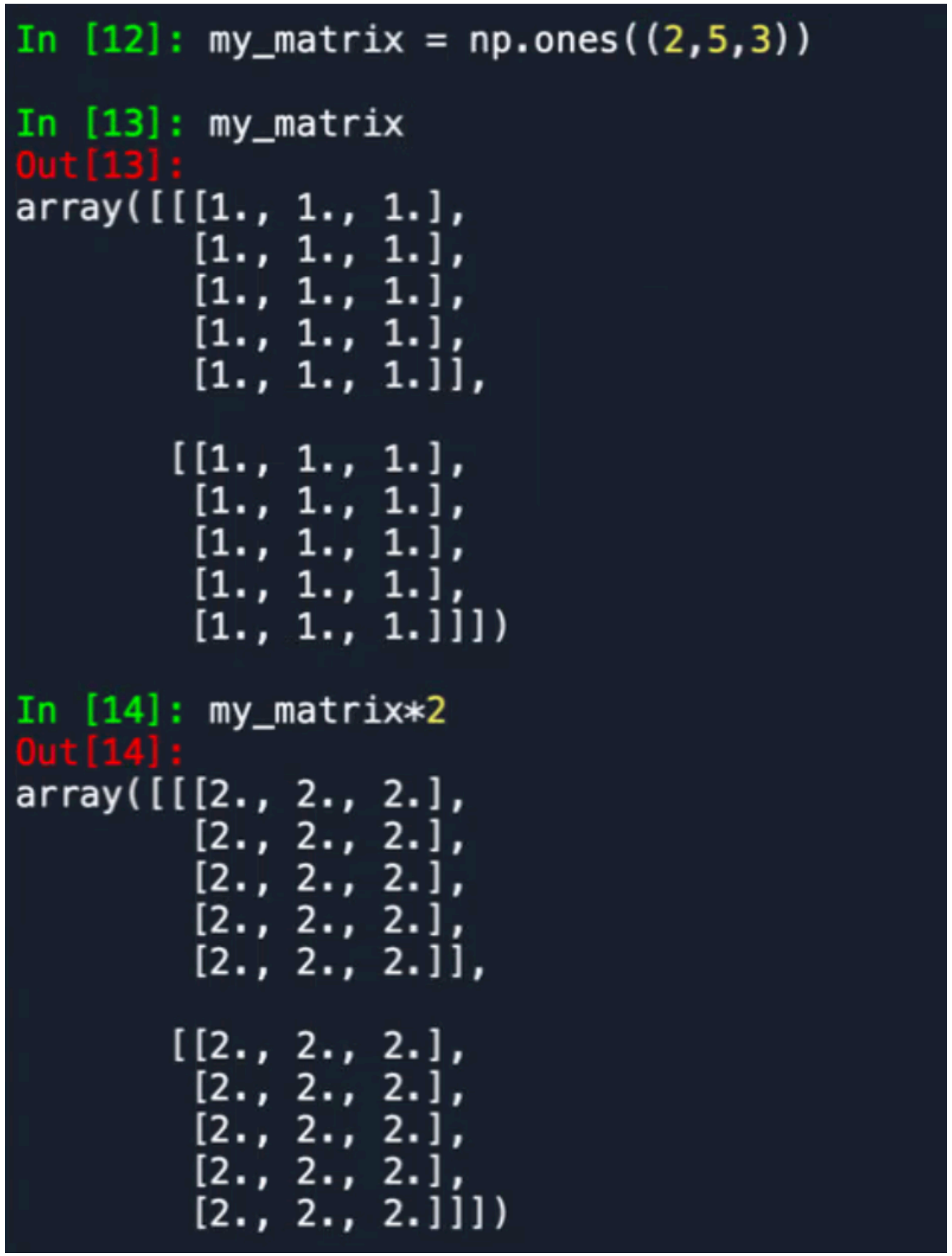


```
In [9]: np.log10((my_array_1+my_array_2)/my_array_1)
Out[9]:
array([0.77815125, 0.81291336, 0.47712125, 0.30103   , 0.30103   ,
       0.35218252, 0.47712125])
```

# Python: numpy

How to create an N-dim matrix

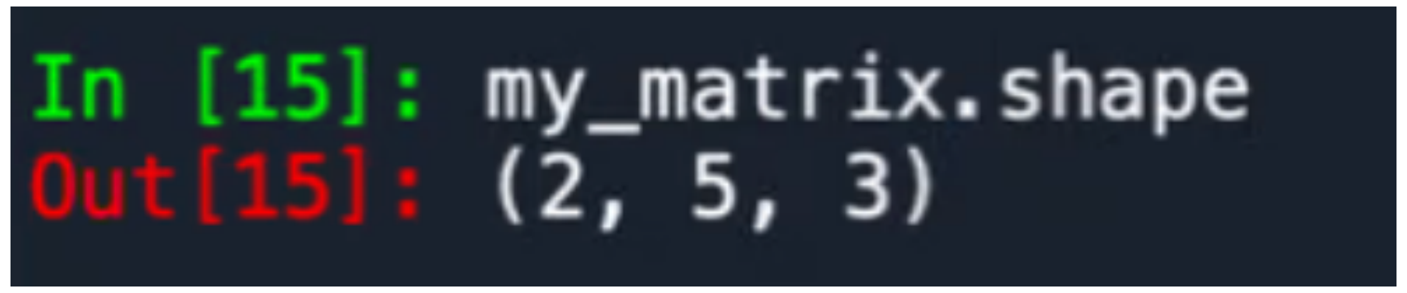
numpy.ones creates an array and fills it with 1.



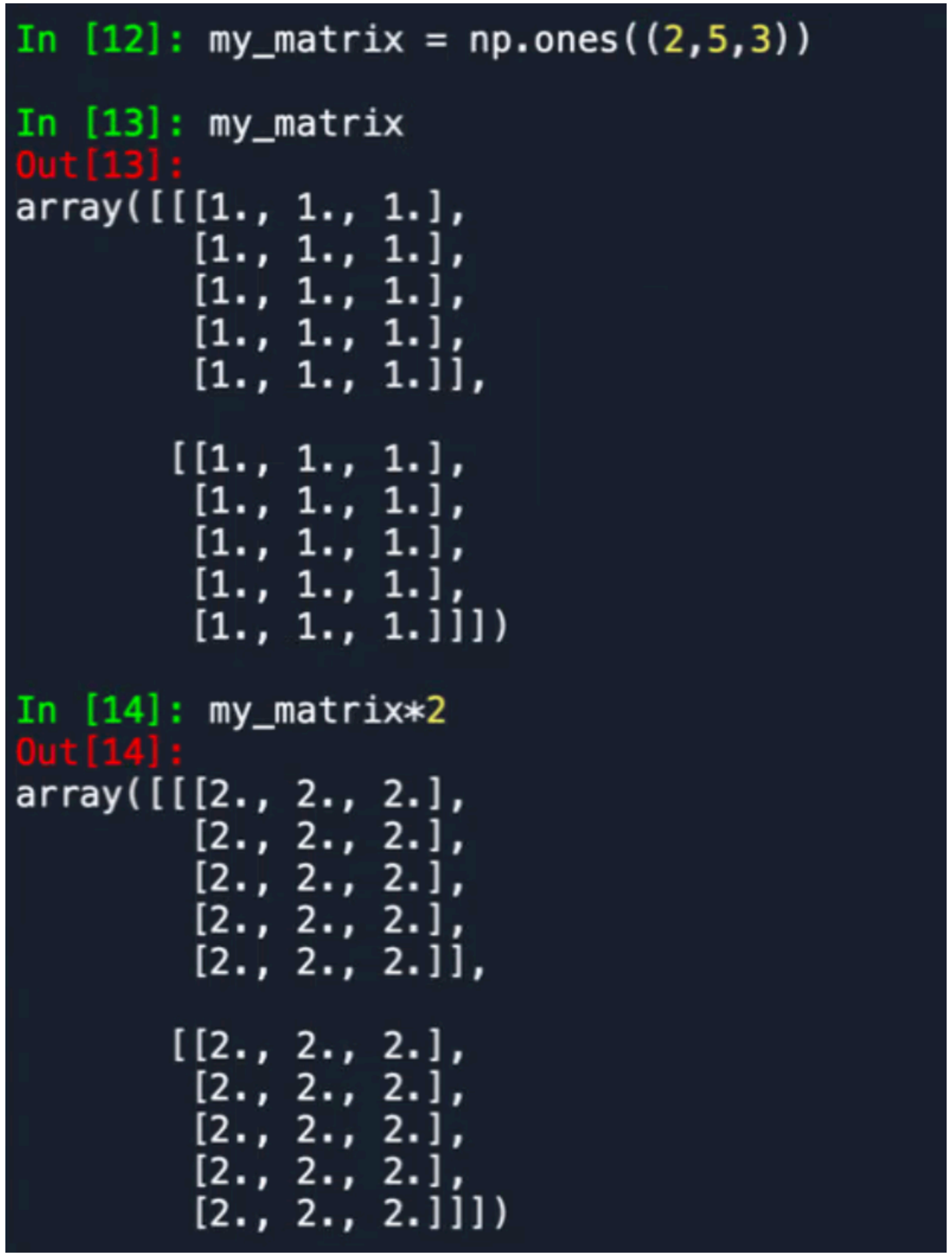
```
In [12]: my_matrix = np.ones((2,5,3))

In [13]: my_matrix
Out[13]:
array([[[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]])

In [14]: my_matrix*2
Out[14]:
array([[[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]],

       [[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]]])
```

# Python: numpy

What are the dimensions of the matrix?

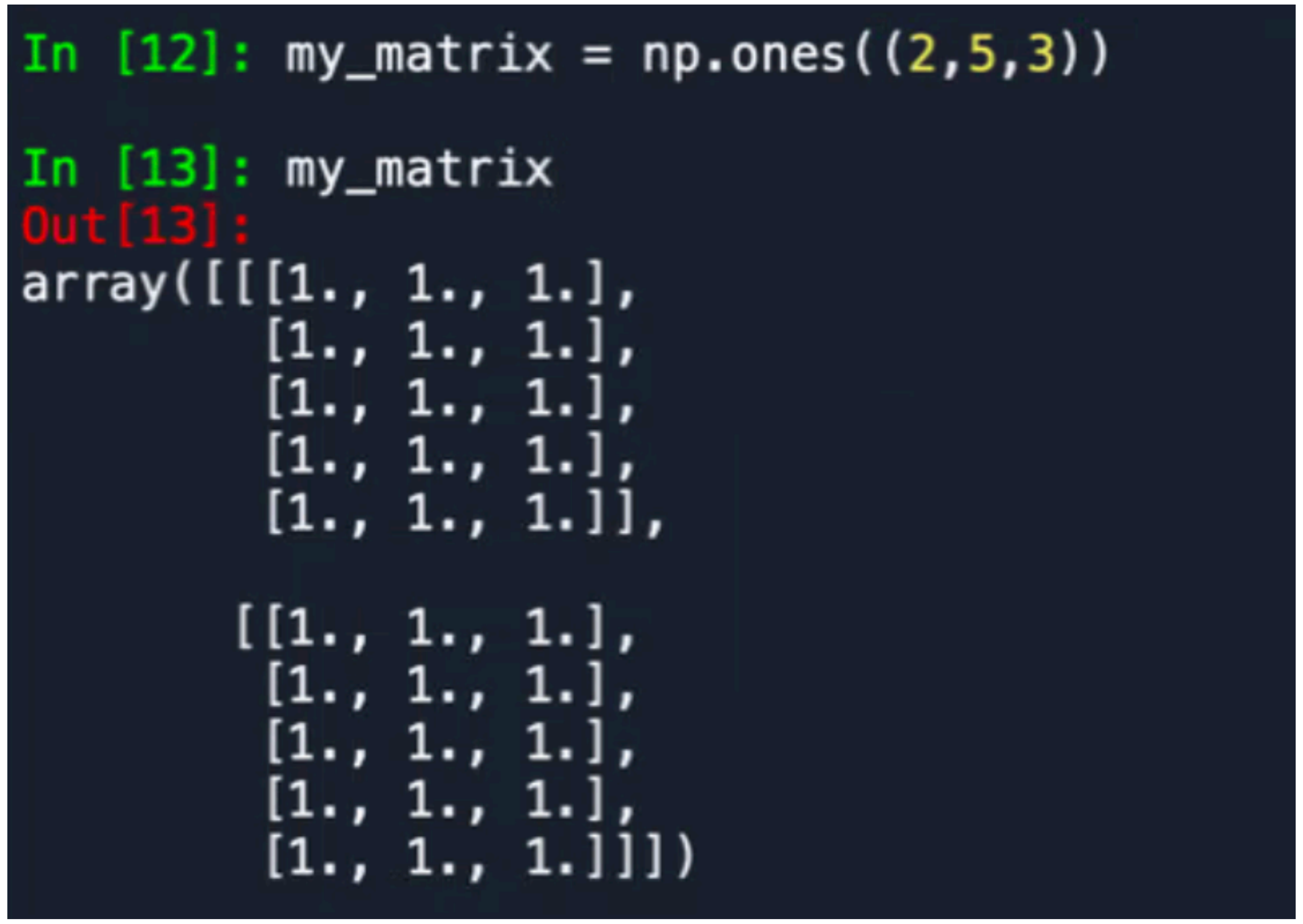


```
In [15]: my_matrix.shape
Out[15]: (2, 5, 3)
```

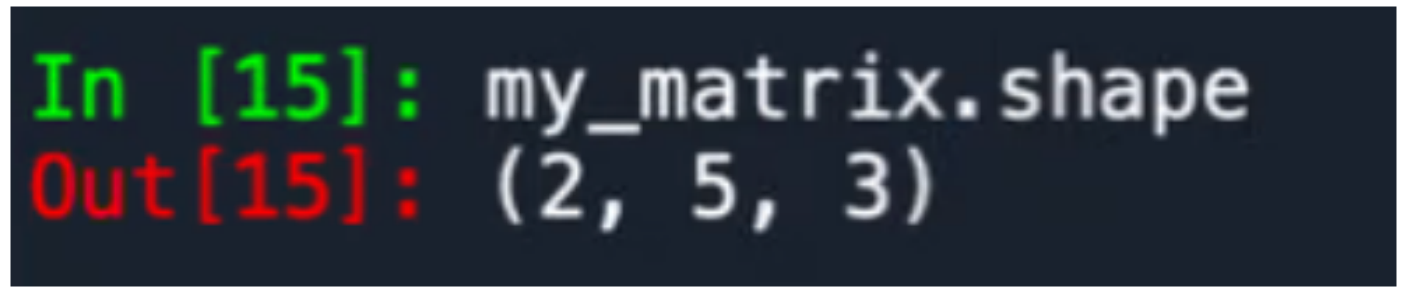


```
In [12]: my_matrix = np.ones((2,5,3))

In [13]: my_matrix
Out[13]:
array([[[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]])

In [14]: my_matrix*2
Out[14]:
array([[[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]],

       [[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]]])
```

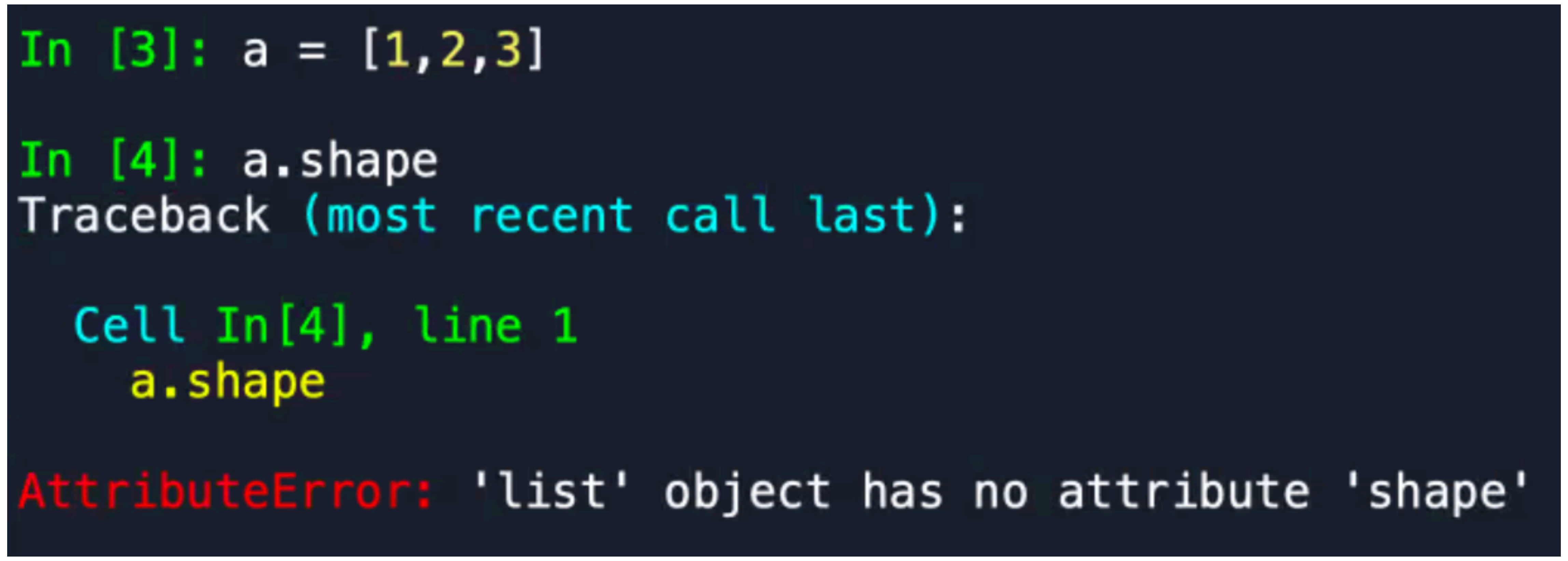
# Python: numpy



```
In [12]: my_matrix = np.ones((2,5,3))

In [13]: my_matrix
Out[13]:
array([[[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]]])
```



```
In [15]: my_matrix.shape
Out[15]: (2, 5, 3)
```

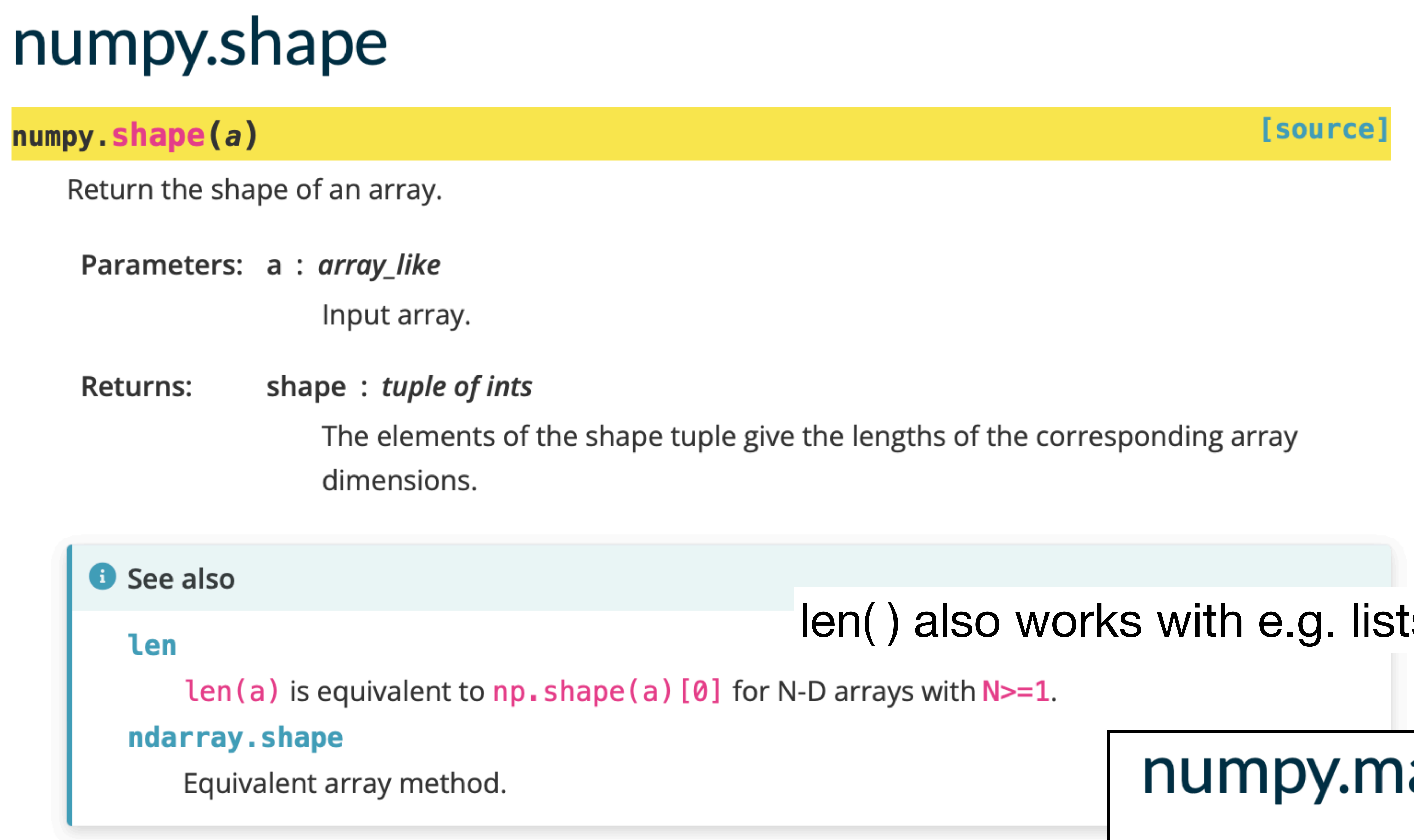


```
In [3]: a = [1,2,3]

In [4]: a.shape
Traceback (most recent call last):

  Cell In[4], line 1
    a.shape

AttributeError: 'list' object has no attribute 'shape'
```

The shape attribute only works with arrays

An attribute is a ~feature of the data structure
that you can access with the .
(if the method is present for that data structure)

## Python: numpy



## numpy.shape

`numpy.shape(a)` [source]

Return the shape of an array.

**Parameters:** **a** : *array_like*

Input array.

**Returns:** **shape** : *tuple of ints*

The elements of the shape tuple give the lengths of the corresponding array dimensions.

**See also**

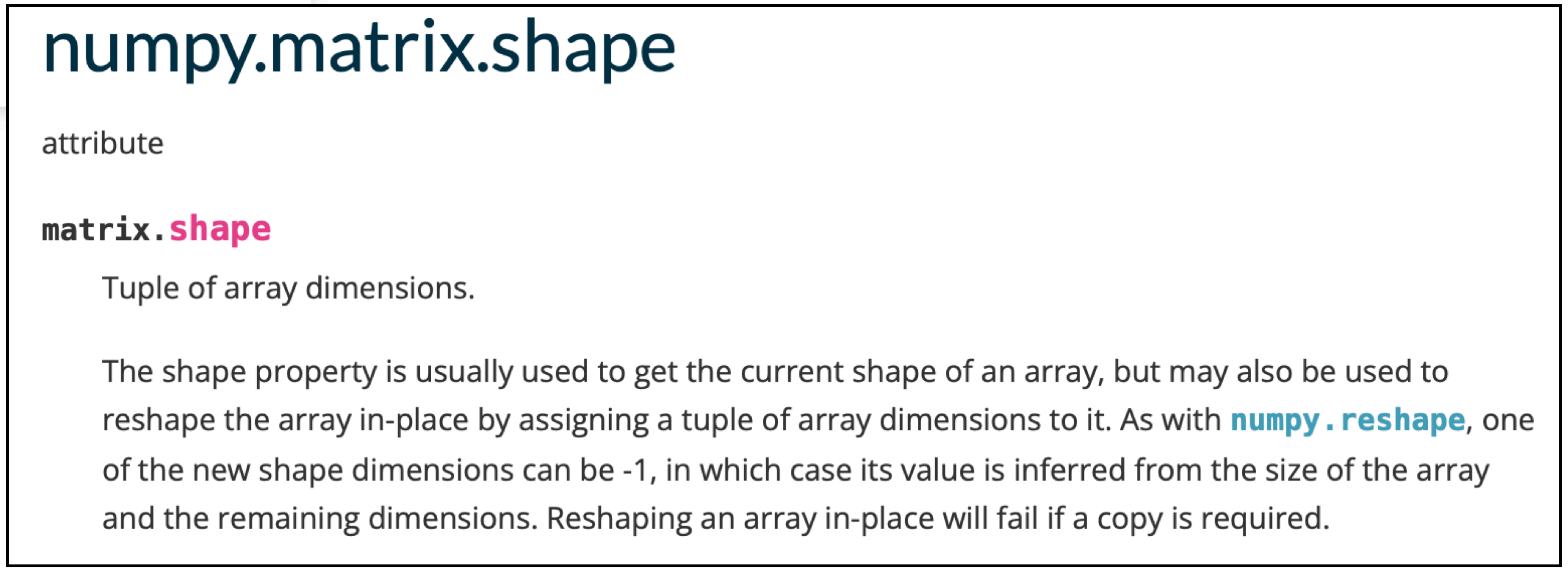
`len`

`len(a)` is equivalent to `np.shape(a)[0]` for N-D arrays with `N>=1`.

`ndarray.shape`

Equivalent array method.

len( ) also works with e.g. lists



## numpy.matrix.shape

attribute

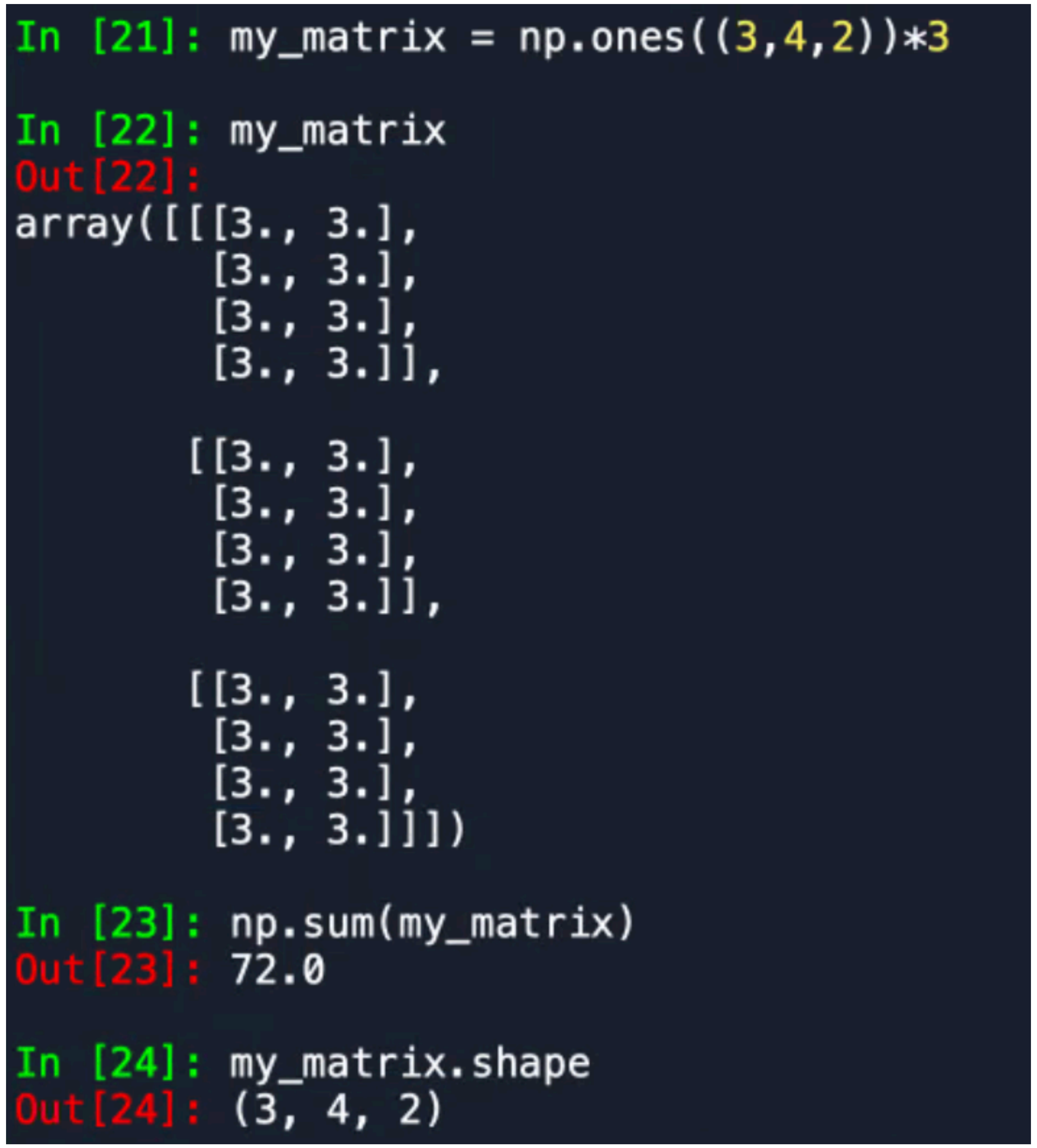
`matrix.shape`

Tuple of array dimensions.

The shape property is usually used to get the current shape of an array, but may also be used to reshape the array in-place by assigning a tuple of array dimensions to it. As with `numpy.reshape`, one of the new shape dimensions can be -1, in which case its value is inferred from the size of the array and the remaining dimensions. Reshaping an array in-place will fail if a copy is required.

## Python: numpy

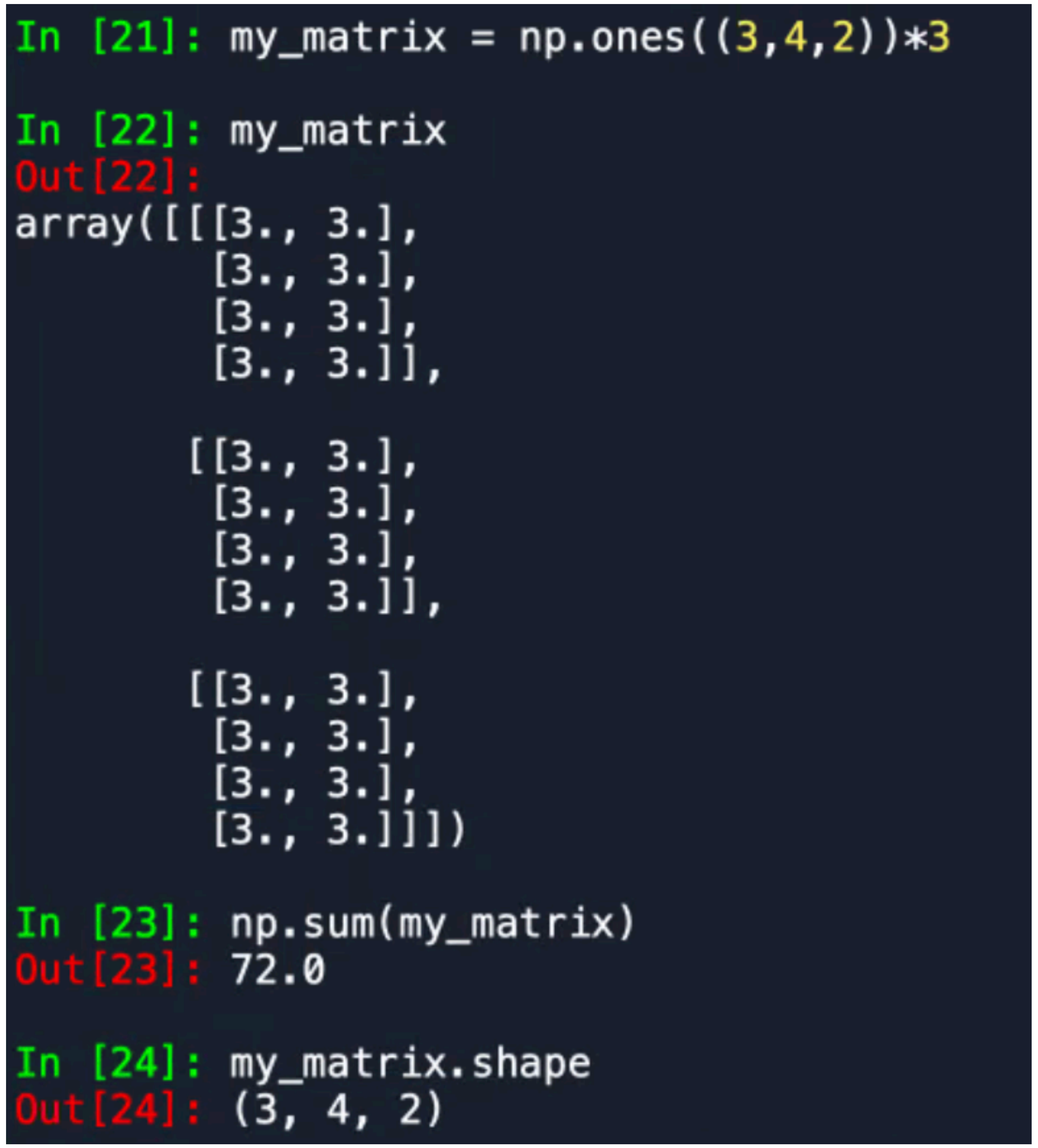
Operations with matrices



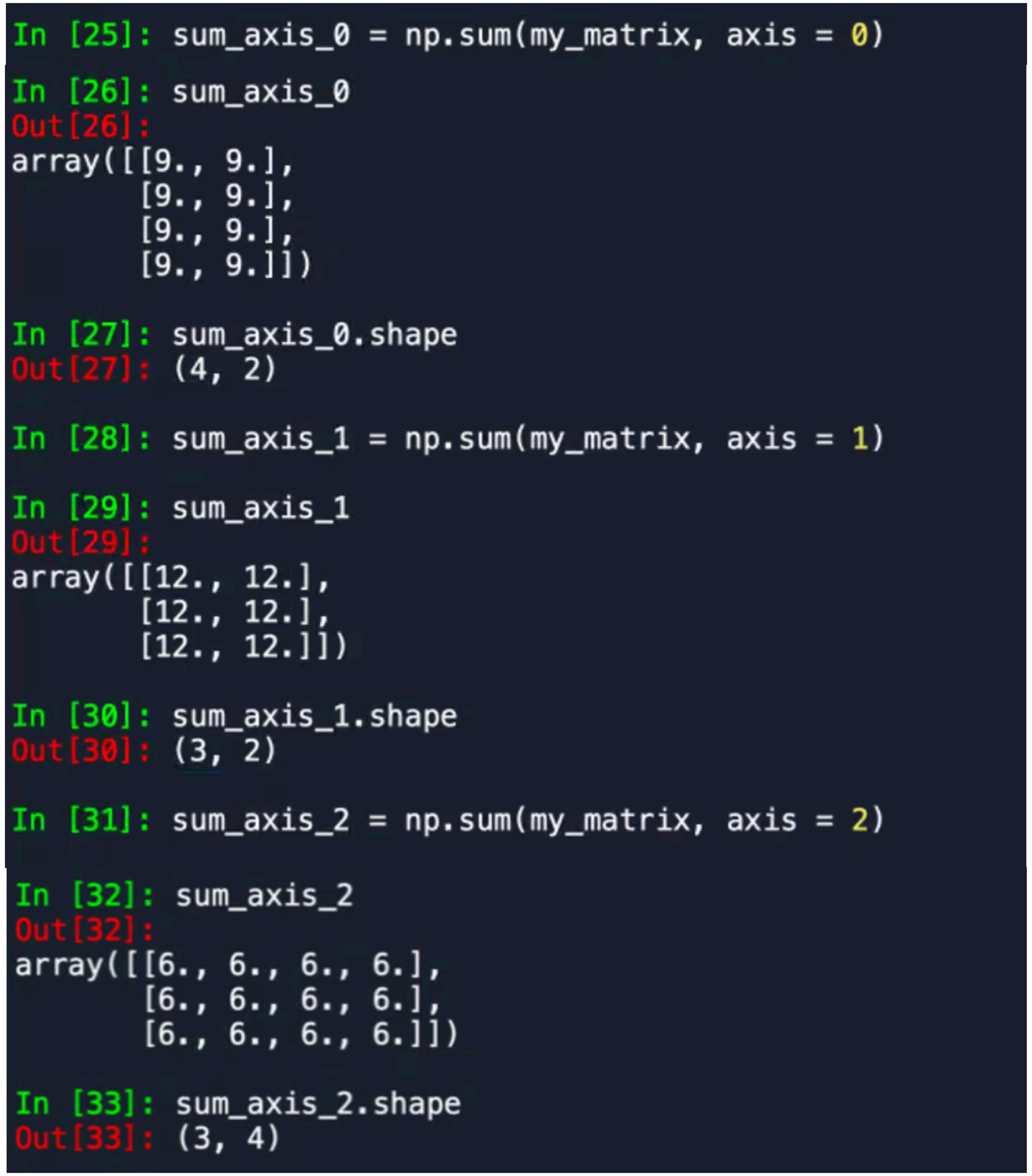
```
In [21]: my_matrix = np.ones((3,4,2))*3

In [22]: my_matrix
Out[22]:
array([[[3., 3.],
        [3., 3.],
        [3., 3.],
        [3., 3.]],

       [[3., 3.],
        [3., 3.],
        [3., 3.],
        [3., 3.]],

       [[3., 3.],
        [3., 3.],
        [3., 3.],
        [3., 3.]]])

In [23]: np.sum(my_matrix)
Out[23]: 72.0

In [24]: my_matrix.shape
Out[24]: (3, 4, 2)
```

# Python: numpy

Operations with matrices



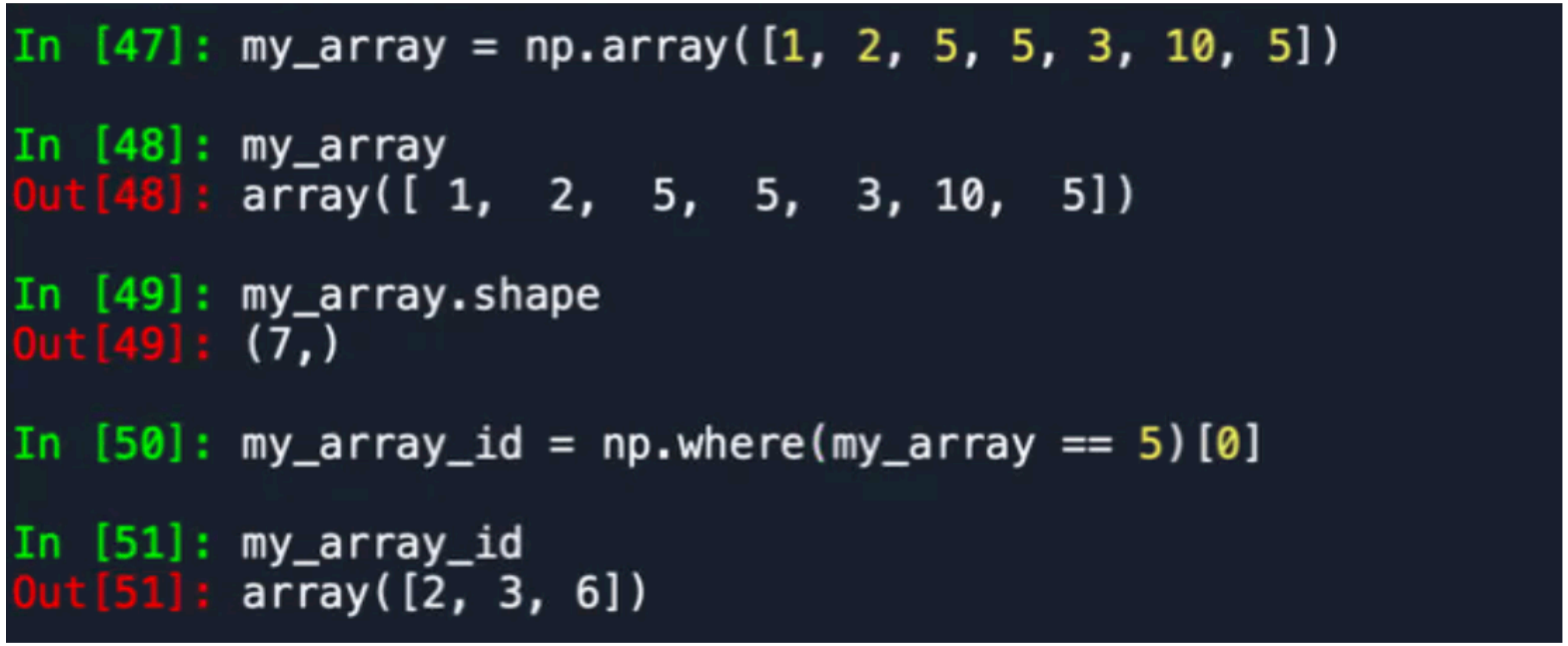
```
In [21]: my_matrix = np.ones((3,4,2))*3

In [22]: my_matrix
Out[22]:
array([[[3., 3.],
        [3., 3.],
        [3., 3.],
        [3., 3.]],

       [[3., 3.],
        [3., 3.],
        [3., 3.],
        [3., 3.]],

       [[3., 3.],
        [3., 3.],
        [3., 3.],
        [3., 3.]]])

In [23]: np.sum(my_matrix)
Out[23]: 72.0

In [24]: my_matrix.shape
Out[24]: (3, 4, 2)
```



```
In [25]: sum_axis_0 = np.sum(my_matrix, axis = 0)

In [26]: sum_axis_0
Out[26]:
array([[9., 9.],
       [9., 9.],
       [9., 9.],
       [9., 9.]])

In [27]: sum_axis_0.shape
Out[27]: (4, 2)

In [28]: sum_axis_1 = np.sum(my_matrix, axis = 1)

In [29]: sum_axis_1
Out[29]:
array([[12., 12.],
       [12., 12.],
       [12., 12.]])

In [30]: sum_axis_1.shape
Out[30]: (3, 2)

In [31]: sum_axis_2 = np.sum(my_matrix, axis = 2)

In [32]: sum_axis_2
Out[32]:
array([[6., 6., 6., 6.],
       [6., 6., 6., 6.],
       [6., 6., 6., 6.]])

In [33]: sum_axis_2.shape
Out[33]: (3, 4)
```

# Python: numpy

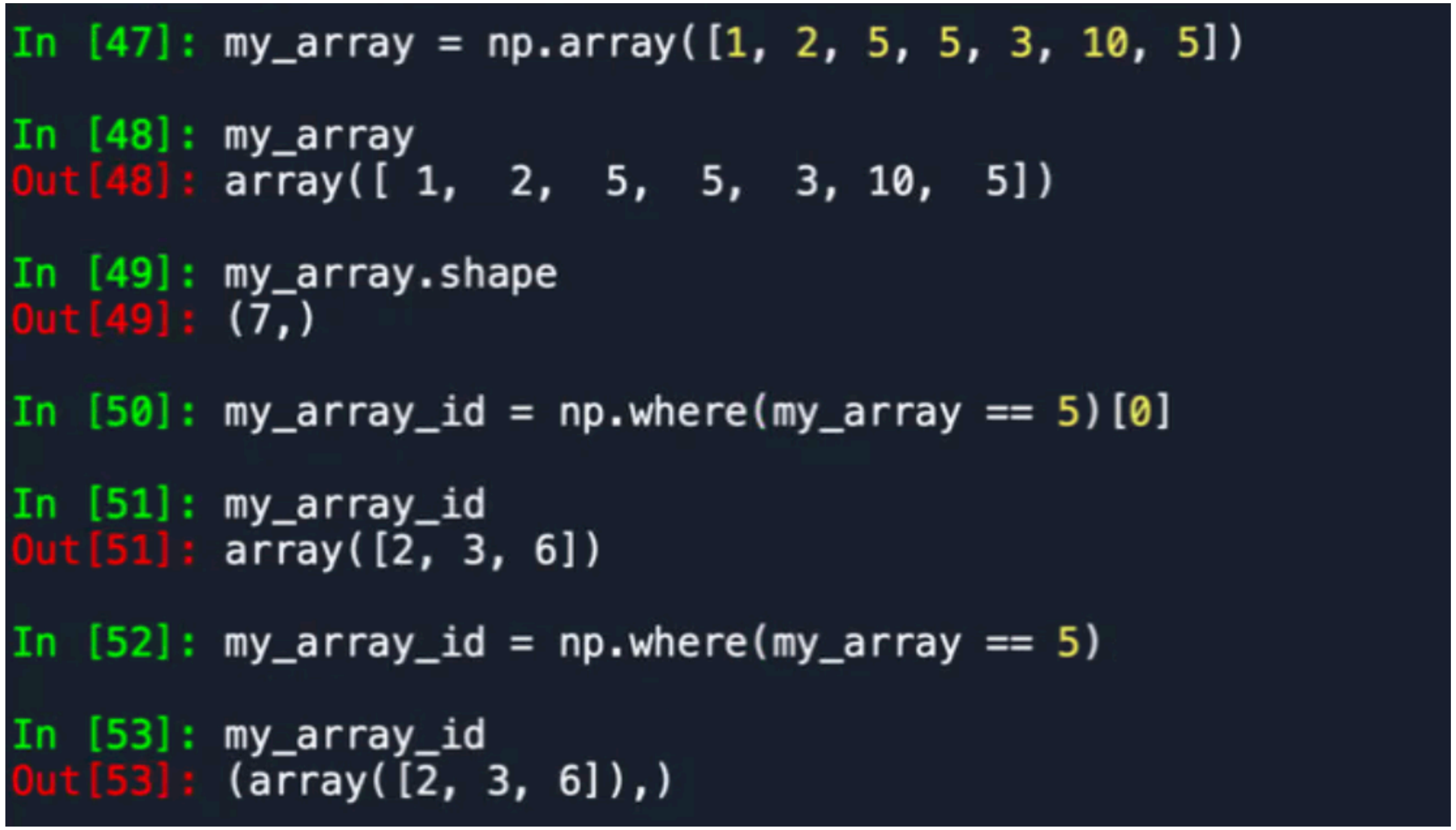
**numpy.where**



```
In [47]: my_array = np.array([1, 2, 5, 5, 3, 10, 5])

In [48]: my_array
Out[48]: array([ 1,  2,  5,  5,  3, 10,  5])

In [49]: my_array.shape
Out[49]: (7,)

In [50]: my_array_id = np.where(my_array == 5)[0]

In [51]: my_array_id
Out[51]: array([2, 3, 6])
```

## Python: numpy

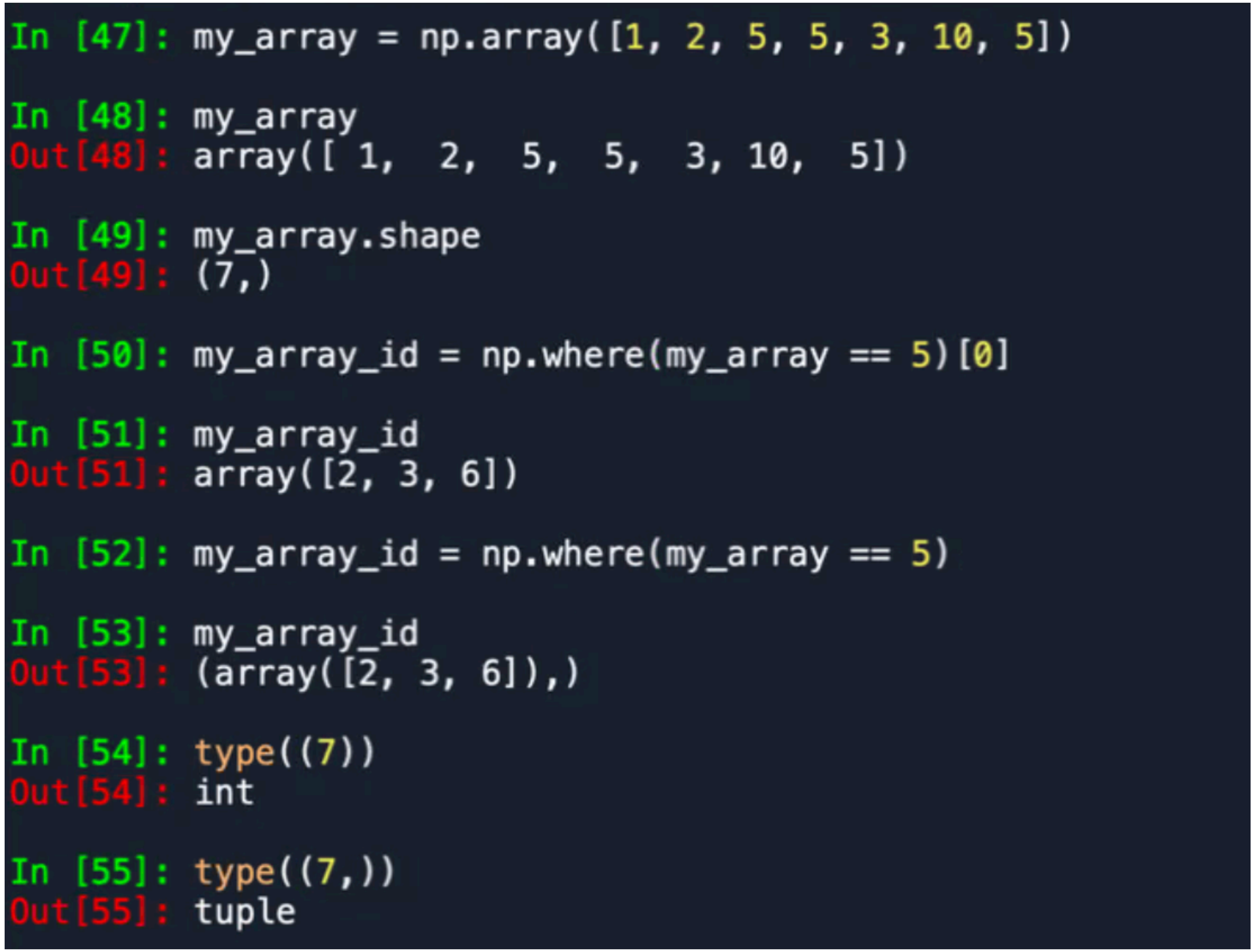
**numpy.where**



```
In [47]: my_array = np.array([1, 2, 5, 5, 3, 10, 5])

In [48]: my_array
Out[48]: array([ 1,  2,  5,  5,  3, 10,  5])

In [49]: my_array.shape
Out[49]: (7,)

In [50]: my_array_id = np.where(my_array == 5)[0]

In [51]: my_array_id
Out[51]: array([2, 3, 6])

In [52]: my_array_id = np.where(my_array == 5)

In [53]: my_array_id
Out[53]: (array([2, 3, 6]),)
```

## Python: numpy

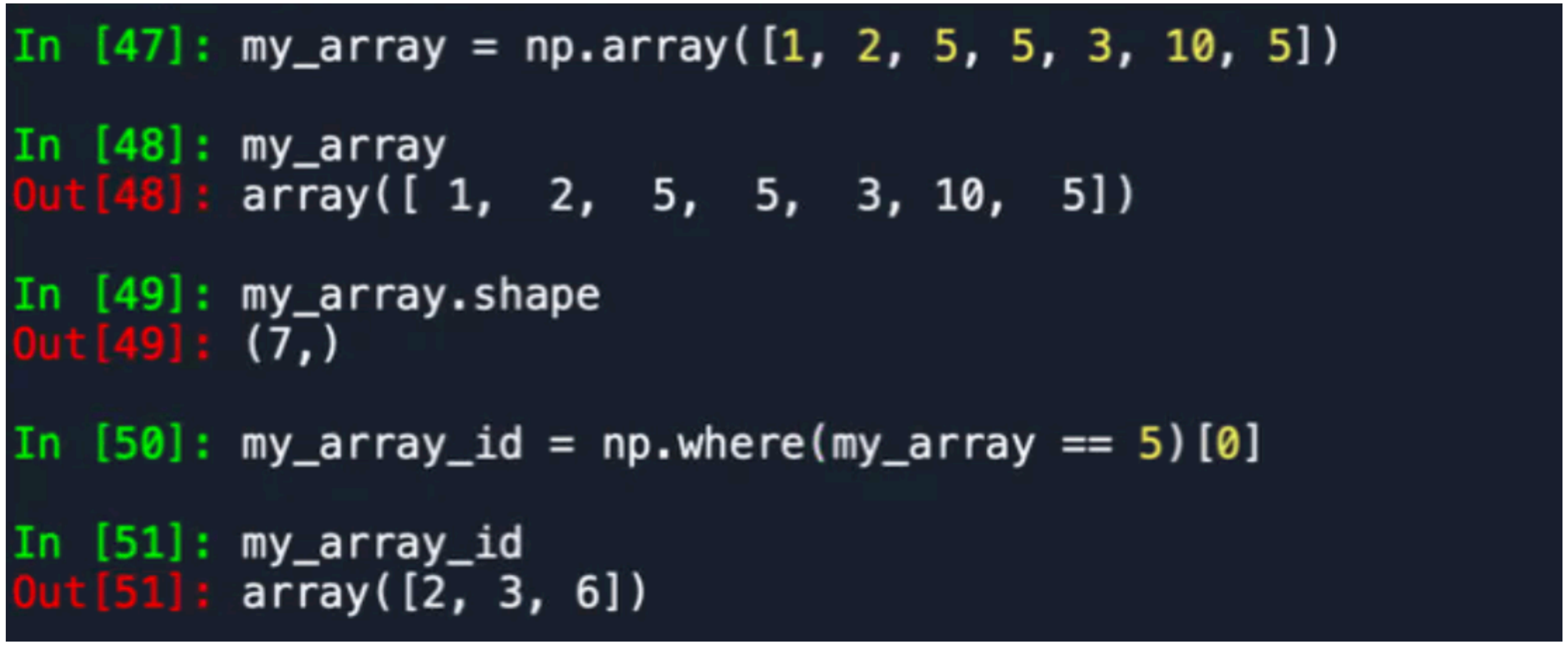
**numpy.where**



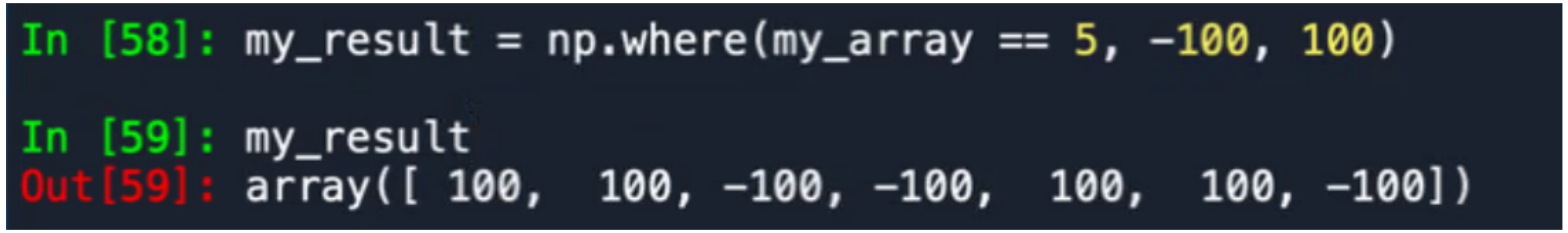
```
In [47]: my_array = np.array([1, 2, 5, 5, 3, 10, 5])

In [48]: my_array
Out[48]: array([ 1,  2,  5,  5,  3, 10,  5])

In [49]: my_array.shape
Out[49]: (7,)

In [50]: my_array_id = np.where(my_array == 5)[0]

In [51]: my_array_id
Out[51]: array([2, 3, 6])

In [52]: my_array_id = np.where(my_array == 5)

In [53]: my_array_id
Out[53]: (array([2, 3, 6]),)

In [54]: type((7))
Out[54]: int

In [55]: type((7,))
Out[55]: tuple
```

# Python: numpy arrays

**numpy.where**



```
In [47]: my_array = np.array([1, 2, 5, 5, 3, 10, 5])

In [48]: my_array
Out[48]: array([ 1,  2,  5,  5,  3, 10,  5])

In [49]: my_array.shape
Out[49]: (7,)

In [50]: my_array_id = np.where(my_array == 5)[0]

In [51]: my_array_id
Out[51]: array([2, 3, 6])
```



```
In [58]: my_result = np.where(my_array == 5, -100, 100)

In [59]: my_result
Out[59]: array([ 100,  100, -100, -100,  100,  100, -100])
```