Class 5:
User-defined functions

Python for Data Analysts: Method & Tools

Felipe Dominguez - Adjunct Professor - Jan 23, 2023

Today's Class

- What are user-defined functions?
- User defined function structure
- Arguments of a user a function
- The return statement
- How to document your functions
- Text-Adventure Assignments

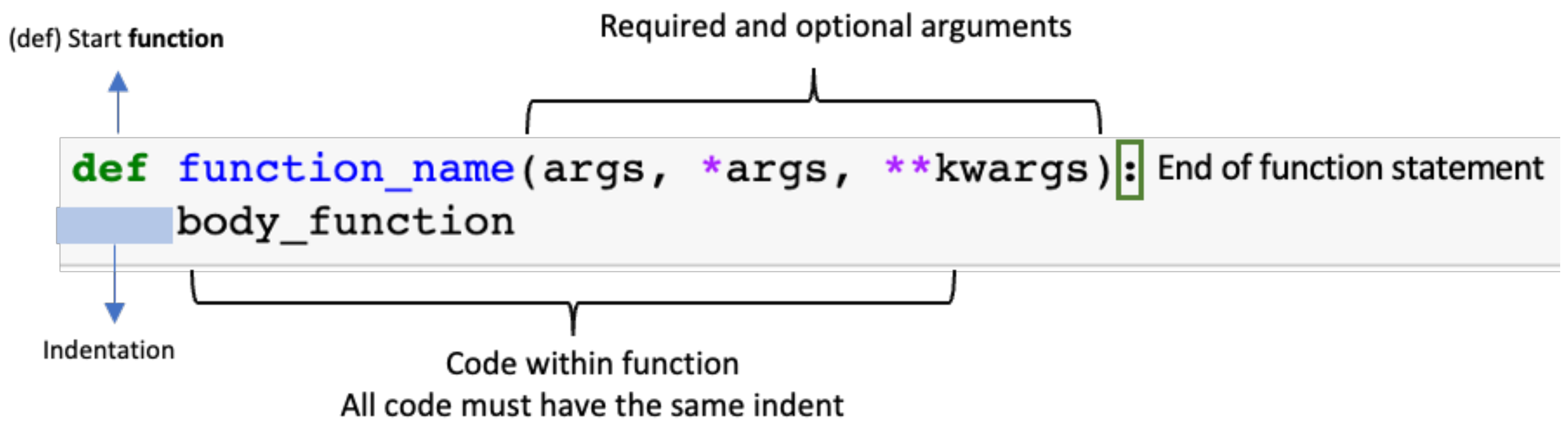
User-defined functions

- Block of code that can be **reused** multiple times
- Created by you to perform a **specific task**
- It **might or not** return an output (.remove() and .pop())

User-defined functions — Structure

User-defined functions — Structure

- Indentation and order **matters!**
- **Unique** function name.
- You can include **mandatory** and/or **optional** arguments.
- Functions **only run** when they **are called**.



```
def function_name(args, *args, **kwargs):
    body_function
```

(def) Start function
Required and optional arguments
End of function statement
Indentation
Code within function
All code must have the same indent

User-defined functions — Structure

Very basic function (No arguments)

```
def hello_world():
    print("Hello, World!")

hello_world()
```

Calling a function

```
>>> "Hello, World!"
```

User-defined functions — Structure

Very basic function (No arguments)

```
def hello_world():
    print(“Hello, World!” + world)
```

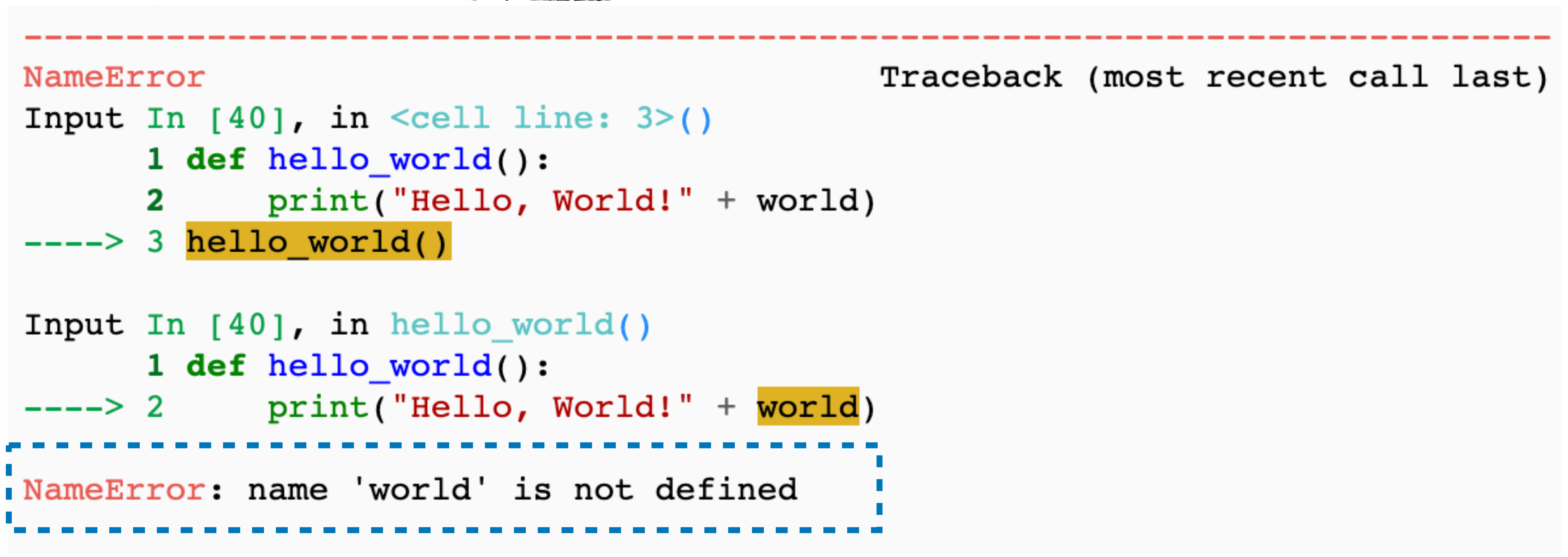
The function **has not been called.**
Python will **not raise an error.**

User-defined functions — Structure

Very basic function (No arguments)

```
def hello_world():
    print("Hello, World!" + world)


hello_world()
```

Calling a function



```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Input In [40], in <cell line: 3>()
      1 def hello_world():
      2     print("Hello, World!" + world)
----> 3 hello_world()

Input In [40], in hello_world()
      1 def hello_world():
----> 2     print("Hello, World!" + world)

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

User-defined functions — Mandatory Arguments

User-defined functions — Mandatory Arguments

- Mandatory arguments are the first to be defined in a function.
- If you don't parse the argument, **python will raise an error**
- Argument lives within the function's body

Square of a number

```
def square_root(x):
    print( x**(1/2) )


square_root(4)
```

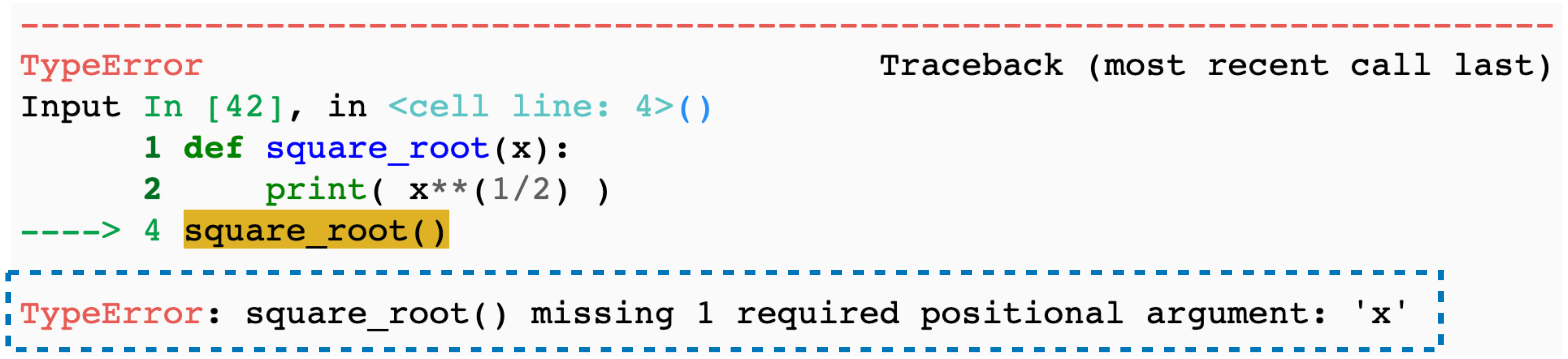
```
>>> 2.0
```

User-defined functions — Mandatory Arguments

- Mandatory arguments are the first to be defined in a function.
- If you don't parse the argument, **python will raise an error.**
- Argument lives within the function's body.

Square of a number

```
def square_root(x):
    print( x**(1/2) )


square_root()
```



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [42], in <cell line: 4>()
      1 def square_root(x):
      2     print( x**(1/2) )
----> 4 square_root()

TypeError: square_root() missing 1 required positional argument: 'x'
```

User-defined functions — Optional Arguments

User-defined functions — Optional Arguments

- Optional arguments are the **second to be defined** in a function.
- You **must indicate a default** value for the optional argument.
- If you don't parse the optional argument, **Python will use the default value.**

n root of a number

```
def n_root(x, y = 2):
    print( x**(1/y) )


n_root(16)
```

```
>>> 4.0
```

User-defined functions — Optional Arguments

- Optional arguments are the **second to be defined** in a function.
- You **must indicate a default** value for the optional argument.
- If you don't parse the optional argument, **Python will use the default value.**

n root of a number

```
def n_root(x, y = 2):
    print( x**(1/y) )


n_root(16, 4)
```

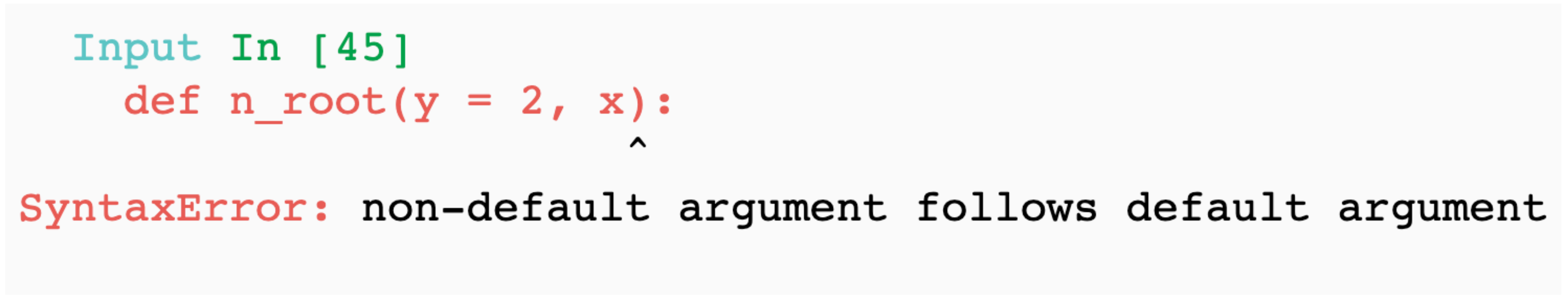
```
>>> 2.0
```

User-defined functions — Structure

- If you define a mandatory argument after an optional argument, **Python will raise an error.**

n root of a number

```
def n_root(y = 2, x):
    print( x**(1/y) )
```



```
  Input In [45]
    def n_root(y = 2, x):
                      ^
SyntaxError: non-default argument follows default argument
```

User-defined functions — return

User-defined functions — return

- We've been cheating. What happened if we don't use the print function?
- Python will **return nothing**, because we didn't explicitly tell the function to return a value
- The **return statement** allows you to pass elements from inside a function to outside of it.

n root of a number

```
def n_root(x, y = 2):
    x**(1/y)


n_root(16)
```

```
>>>
```

User-defined functions — return

- Return a value from the function (local scope) to the global scope.

n root of a number

```
def n_root(x, y = 2):
    return x**(1/y)


n_root(16)
```

```
>>> 4.0
```

User-defined functions — return

- You can create variables and pass those variables to the global scope.
- Now you can leverage functions in your global coding environment.

n root of a number

```
def n_root(x, y = 2):
    value = x**(1/y)
    return value

number = n_root(16)
print(number)
```

```
>>> 4.0
```

User-defined functions — return tuples!

- You can return more than one element from a function.

n root of a number

```
def n_root(x, y = 2):
    value_1 = x**(1/y)
    value_2 = value_1 * 4
    return value_1, value_2

number = n_root(16)
print(number)
```

It's a tuple! $\longrightarrow$

```
>>> (4.0, 16.0)
```

User-defined functions — return lists!

- You can return any data type from a function

n root of a number

```
def n_root(x, y = 2):
    my_list = []
    for i in range(1, y):
        value_1 = x**(1/i)
        my_list.append(value_1)
    return my_list
number = n_root(16, 5)
print(number)
```

It’s a list! →

```
>>> [16.0, 4.0, 2.5198420997897464, 2.0]
```

User-defined functions — DocStrings

User-defined functions — DocString

- DocStrings helps to document functions (for you and others!).

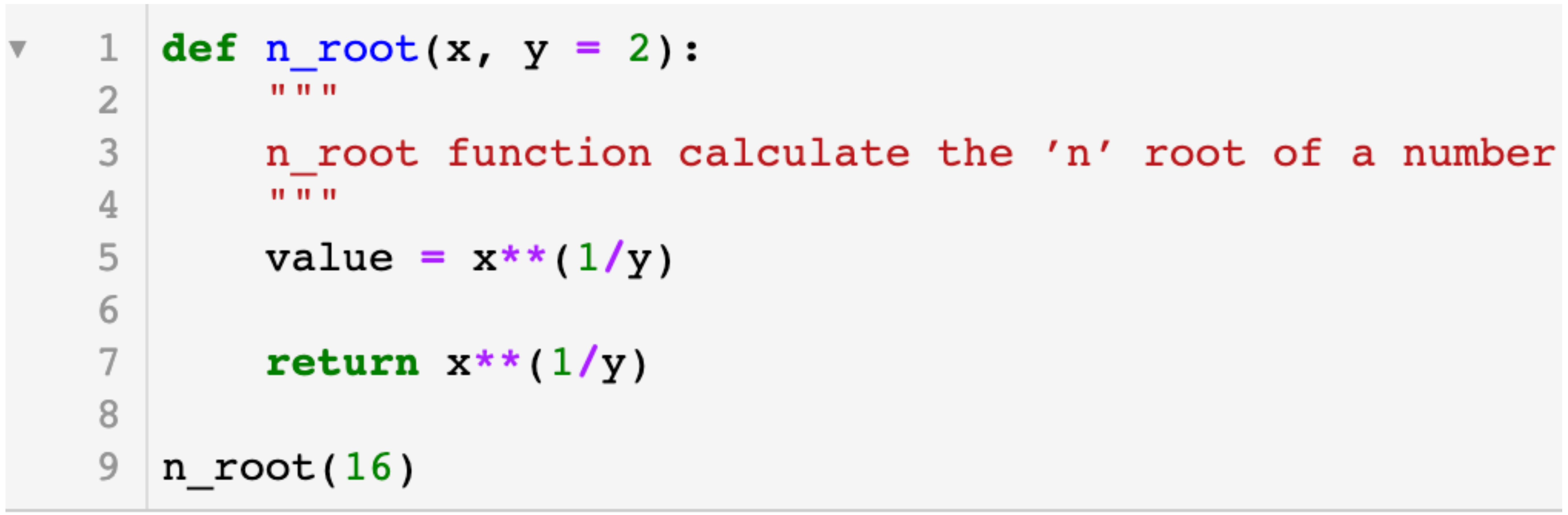
- DocStrings are a best practice and, as a rule of thumb, represent the quality of a function.

n root of a number

```
def n_root(x, y = 2):
    “”” DocString “””
    value = x**(1/y)
    return x**(1/y)
n_root(16)
```

User-defined functions — DocString

- DocStrings helps to document functions (for you and others!).
- DocStrings are a best practice and, as a rule of thumb, represent the quality of a function.
- First, describe the function.



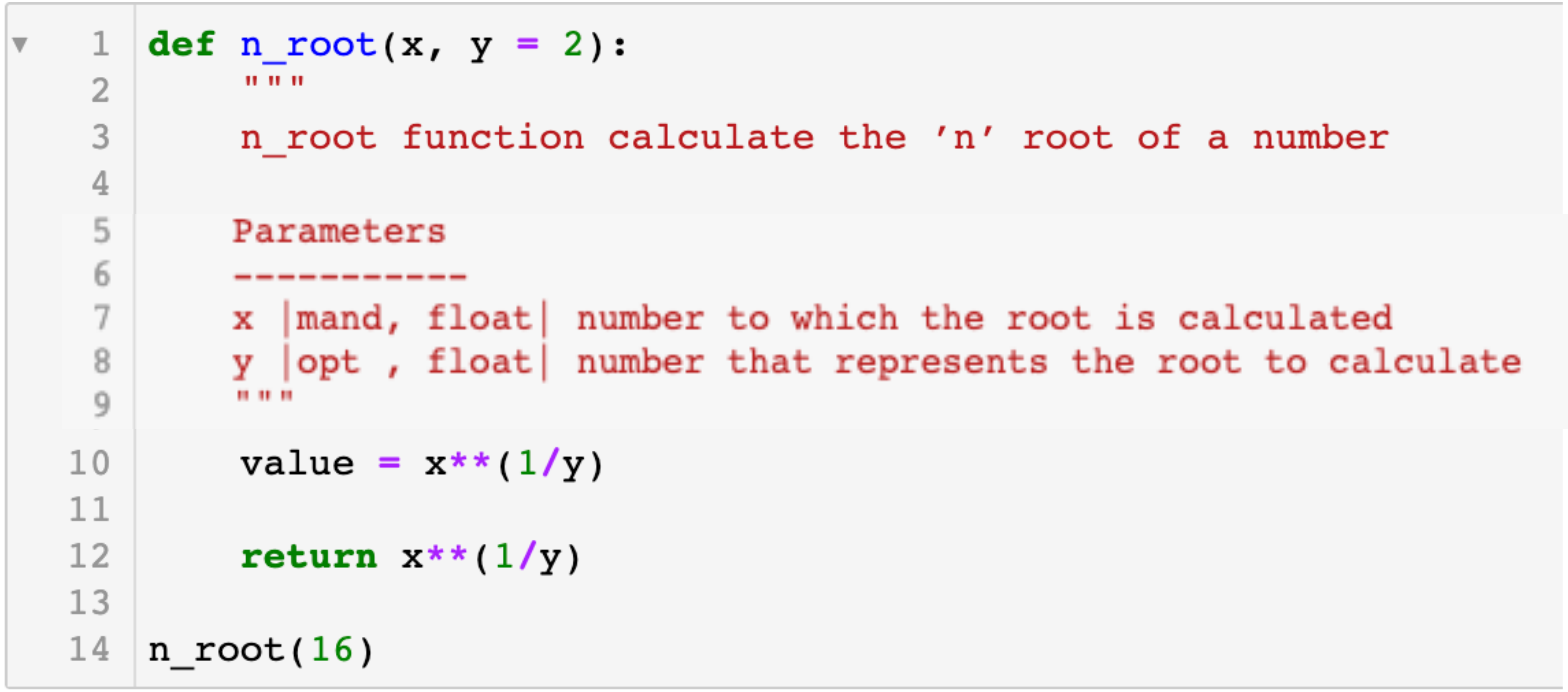
```
def n_root(x, y = 2):
    """
    n_root function calculate the 'n' root of a number
    """
    value = x**(1/y)

    return x**(1/y)

n_root(16)
```

User-defined functions — DocString

- DocStrings helps to document functions (for you and others!).

- DocStrings are a best practice and, as a rule of thumb, represent the quality of a function.

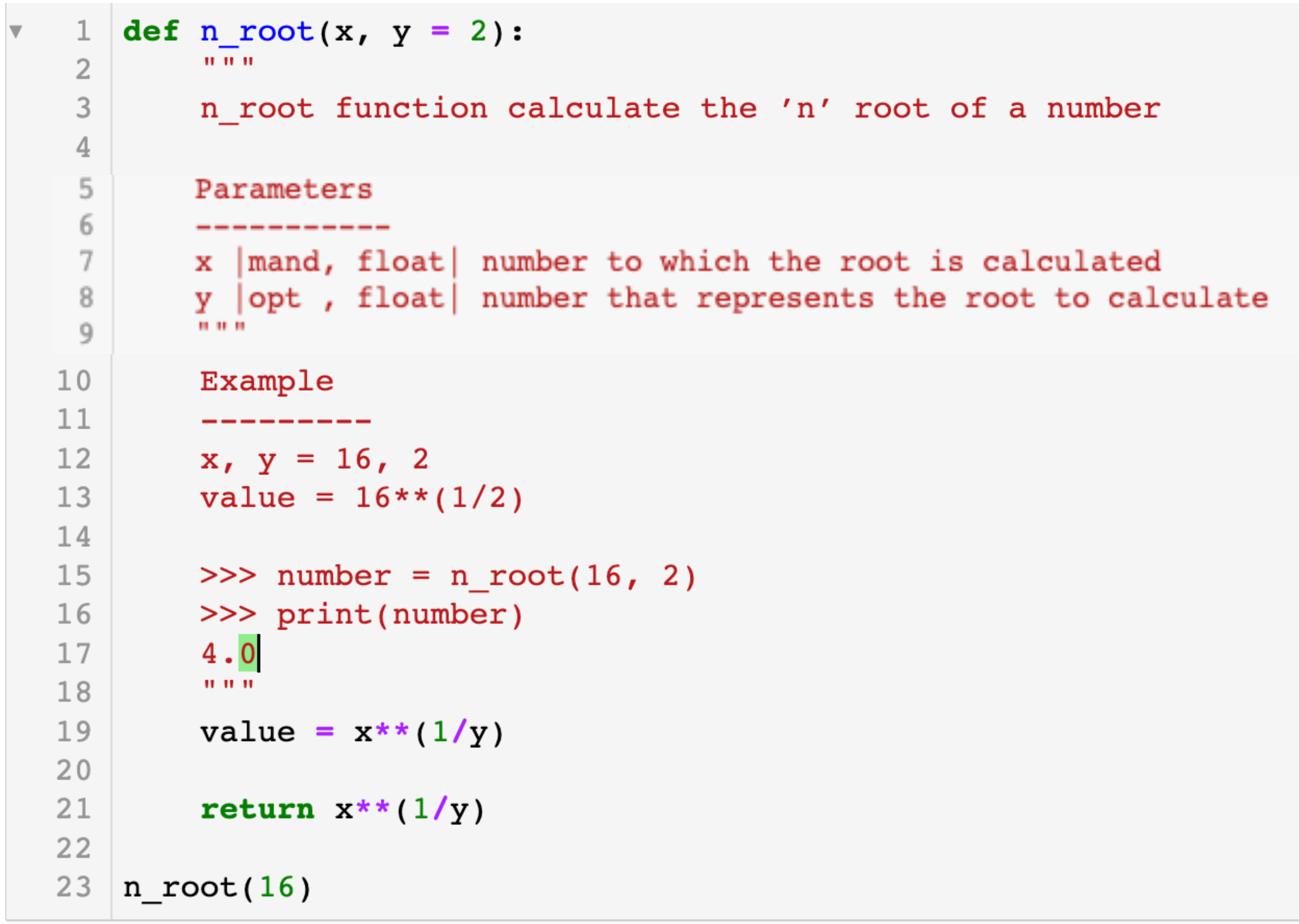
- Second, describe the parameters.



```python
def n_root(x, y = 2):
    """
    n_root function calculate the 'n' root of a number

   Parameters
   -----------
   x |mand, float| number to which the root is calculated
   y |opt , float| number that represents the root to calculate
   """
    value = x**(1/y)

    return x**(1/y)

n_root(16)
```

User-defined functions — DocString

- DocStrings helps to document functions (for you and others!).
- DocStrings are a best practice and, as a rule of thumb, represent the quality of a function.
- Third, provide examples.



```
def n_root(x, y = 2):
    """
    n_root function calculate the 'n' root of a number

    Parameters
    -----------
    x |mand, float| number to which the root is calculated
    y |opt , float| number that represents the root to calculate
    """

    Example
    ----------
    x, y = 16, 2
    value = 16**(1/2)

    >>> number = n_root(16, 2)
    >>> print(number)
    4.0
    """
    value = x**(1/y)

    return x**(1/y)

n_root(16)
```

User-defined functions — try / except

User-defined functions — try / except

- Used to **handle exceptions** (errors)

- If the code after the **try** keywords raise an error, Python will stop and continue with the code after the **except** keyword.

n root of a number

```
try:
    # some code that might throw an exception
except:
    # code to execute if an exception occurs
```

User-defined functions — try / except

- Used to **handle exceptions** (errors)
- If the code after the **try** keywords raise an error, Python will stop and continue with the code after the **except** keyword.

n root of a number

```
try:
    100 / 0
except:
    print("I can't divide by zero!”)
```

Let’s create some functions