HSLU Lucerne University of Applied Sciences and Arts

Python for Data Science: SW06

Functions part II
Recursion

Information Technology
March 27, 2025

Content

- Python Functions II
  - Namespace and scope
  - Local and Global Variables
  - Keyword: `global`
  - Keyword: `nonlocal`
- Properties of Parameters
  - Parameter Value
  - Unpacking Arguments (*args, **kwargs)
- Recursion
- Exercises

Python Functions II

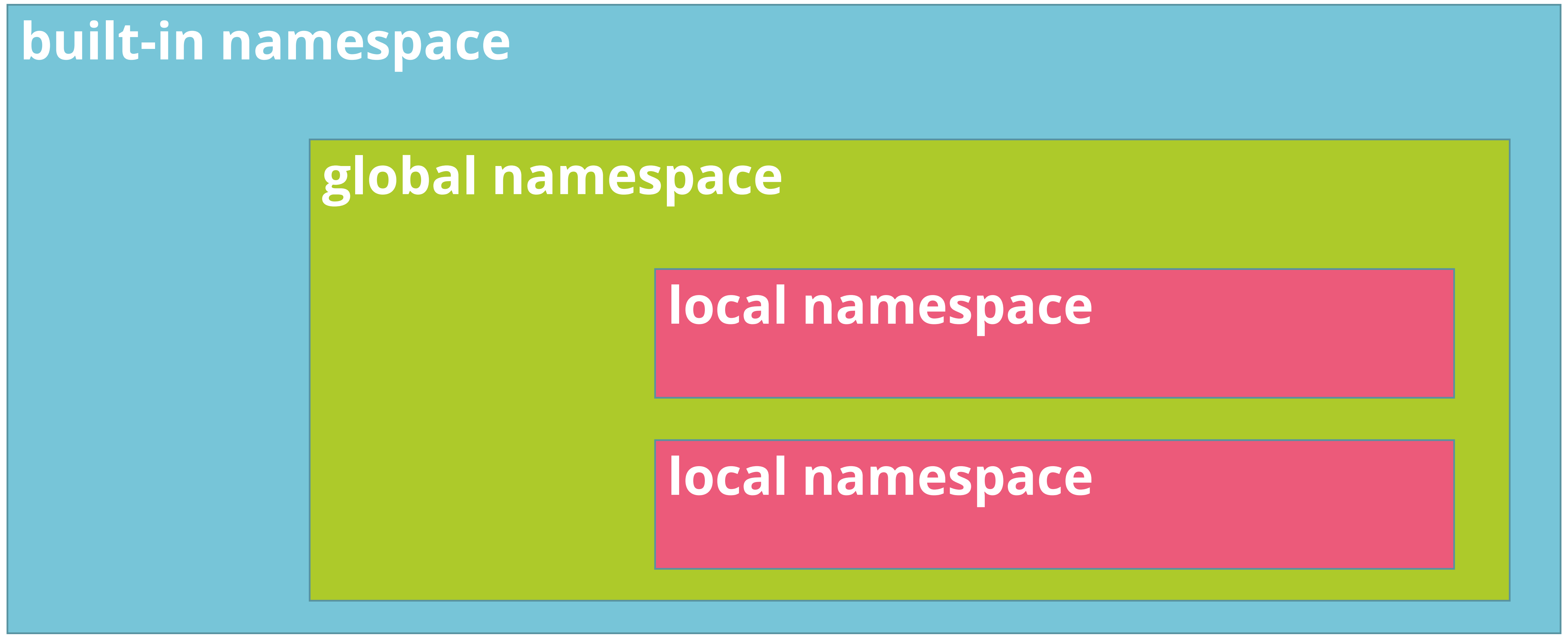
Information Technology

March 27, 2025

Python Functions II: Namespace and Scope

Namespace is the definition of the visibility of a **unique** name for every single object (variables or methods). Python has three different namespaces:

- built-in: encompasses no programmer defined objects.
- global: programmer defined objects available across the whole script.
- local: programmer defined objects in function blocks.



Python Functions II: Namespace and Scope

- Object names must only be **unique** within a given namespace. Hence, the global namespace can have multiple local namespaces having objects with the same naming.
- The **scope** of an object refers to the code section from which an object is accessible.
- Namespaces have a limited **lifetime** that ends when the scope of an object ends:
  - Built-in namespace: ends when the application ends.
  - Global namespace: ends when the module (i.e. script) is unloaded or the application ends.
  - Local namespace: ends when the function (i.e. block) has been finished.

What if a function (a) returns a reference to another function (b)?

- a scope returns a new scope.
- the lifetime of the scope of function (a) ends with the return of the scope (b).
- the scope of function (b) is created when the function (b) is called and ends when reaches the end of its block.

```python
def my_fun():
    def num_sum(a, b):
        return a + b
    return num_sum

print(my_fun()(1,2))
```

Python Functions II: Local and Global Variables

Python allows creating variables without any restricted visibility. This variables in the global scope are accessible in the whole script and are called:

- **global** variable

In contrast, variables being restricted to a specific **function** block are called:

- **local** variable

`global` ⟷ `local`

Python Functions II: Local and Global Variables

free variables (global) are defined on main level or in any control statement including **iterator variables** such as:

- `if-else`
- `match-case`
- `for`-loop (including iterator variables)
- `while`-loop

```
welcome = 'hello world'       #global
numbers = [1, 2, 3, 4, 5]     #global
list_sum = 0                  #global
for i in numbers:             #global
    list_sum += i
    last_element = i          #global
print(welcome)
print('sum: ', list_sum)
print(f'{last_element=}')
print(f'last iterator:{i}')
```

local variable are only visible within a particular function block including **function parameters**:

```
welcome = 'hello world'       #global
numbers = [1, 2, 3, 4, 5]     #global
def sum_list(list_ref):       #local
    list_sum = 0              #local
    for i in list_ref:        #local
        list_sum += i
    return list_sum
print(welcome)
print('sum: ', sum_list(numbers))
```

Python Functions II: Keyword: `global`

In some problem solutions it is meaningful to break the rules of functional functions and to have a **global variable** which gets updated from within a local scope of a function. In this case, we use a **side effect**.

Typical applications are:

- Global access counter (or sum).
- Global buffer index.
- Global (file) name variable.

```python
access_cnt = 0
buffer_name = 'b_name'

def read_buffer():
    global access_cnt
    access_cnt += 1
    with open('b_name') as f:
        new_content = f.readlines()
    return new_content
```

- In this case, the global variable can be accessed by a local re-declaration using the keyword: **global**
- Yet, for the sake of complying with functional style, we try to **avoid** global access whenever possible.

Docu: https://docs.python.org/3/reference/simple_stmts.html#the-global-statement

Python Functions II: Keyword `nonlocal`

A local namespace (x) may comprise another local (nested) namespace (y) that is the case for a nested function. In this case, Python allows to access objects defined in parents namespace by using the keyword: nonlocal

- Accessing an object of parents namespace requires a re-declaration:

```
nonlocal parent_variable
```

- Nonlocal declarations can **not have** a value assignment.

```
nonlocal x = 55          # not allowed
```

```
x = 99                        #global
def bar():
    x = 1                     #local
    print(x)
    def foo():
        nonlocal x            #local
        x = 55                #local
        y = 77                #nested local
        print(x)
        print(y)
    foo()
    print(x)
print(x)
bar()
print(x)
```

Properties of Parameters

Information Technology

March 27, 2025

Properties of Parameters: Parameter Value

Despite the separation of the namespaces for local variables, we always pass **references** to the parameter list.

- The value (and only this) where the reference points to remains unchanged.
- **Risk**: values of consecutive references (the value from a reference to a reference) can still change!

```python
def my_add(p):
    p = p+1
    return p


var = 987654320
print(my_add(var))
print(var)

#output:
987654321
987654320
```

```python
def my_add(p):
    c = p
    c[-1] = 99
    return c

var = [9,8,7,6,5,4,3,2,0]
print(my_add(var))
print(var)

#output:
[9, 8, 7, 6, 5, 4, 3, 2, 99]
[9, 8, 7, 6, 5, 4, 3, 2, 99]
```

Properties of Parameters: Unpacking Arguments (*args and **kwargs)

Assigning multiple items to **one** variable creates a tuple by default:

`var = 1, ‘hello’, 22` is equivalent to `var = (1, ‘hello’, 22)`

Variables can also be assigned in a group, multiple values to multiple variables in one assignment:

`var1, var2, var3 = 1, ‘hello’, 22` is equivalent to `var1 = 1; var2 = ‘hello’; var3 = 22`

Assigning m values to n variables (where m >= n) can be achieved by the packing operator: *

`var1, *var2, var3 = 1, ‘hello’, ‘-‘, ‘world’, 22` is equivalent to

`var1 = 1; var2 = (‘hello’, ‘-‘, ‘world’); var3 = 22`

Python assigns one object to the first and the last variable and **packs** all the remaining objects into a tuple that is assigned to the variable with the **packing** operator (*****): `var2`.

- Missing the packing operator will cause an error: “`too many values to unpack`”

Properties of Parameters: Unpacking Arguments (*args and **kwargs)

The same concept also holds for **parameter** variables. This allows to pass an **undefined number** of objects to a function, such as for the print function (see: https://docs.python.org/3/library/functions.html#print):

- Function definition: `print(*objects, sep=' ', end='\n', file=None, flush=False)`
- `*objects`: `packing operator (*) packs multiple elements into a tuple`

In a similar way, the double asterisk operator ** packs keywords to a dictionary.

- Function definition: `my_fun(*args, **kwargs)`
- `*args`: packing operator (*) packs all non-keyword arguments into a tuple.
- `**kwargs`: double packing operator(**) packs all keyword arguments into a dictionary.

```
def my_fun(*args, **kwargs):
    print(args); print(kwargs)

my_fun('hello', 'world', arg1=33, arg2=55)

#output
('hello', 'world')
{'arg1': 33, 'arg2': 55}
```

Python Functions II: *args and **kwargs

In contrast, lists and tuples can be **unpacked** using the packing operator (*) and (**) for dictionaries, respectively.

For lists and tuples, the unpacking operator passes each sequence element separately to the parameter instead a list as one parameter.

print call	output
`print([1,2,3], [7,8,9], sep=';')`	`[1,2,3];[7,8,9]`
`print(*[1,2,3], *[7,8,9], sep=';')`	`1;2;3;7;8;9`

Python Functions II: *args and **kwargs

The double un-packing operator (**) allows to unpack dictionaries and to assign values of keys to parameter keywords.

CAUTION: This only works, if dictionary key have **exactly same spelling** as parameter keywords.

```
def my_fun(*args, arg1, arg2):
    print(args); print(f'{arg1=}'); print(f'{arg2=}')

my_fun('hello', 'world', **{'arg1':33, 'arg2':55})

#output
('hello', 'world')
arg1=33
arg2=55
```

For further information, read the docu: https://docs.python.org/3/reference/expressions.html#calls

Recursion

Information Technology

March 27, 2025

Recursion

Origin of recursion: https://www.dictionary.com/browse/recursion

1925–30; Late Latin recursiōn- (stem of recursiō) a running back, equivalent to recurs (us):

In programming, this means a function that calls itself.

Advantage: for data scientists, it is especially useful to loop through a data set with undefined dimension and undefined iteration count.

- Simple iteration through tree-like structures
- Concise implementation

Disadvantage:

- Hard to implement and debug functionality.
- Resource intensive.
- Risk of memory overflow.

Recursion

Simple example: Implementation to calculate the factorial of a number.

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

Recursion

Despite increased complexity and risks, recursion allows to solve complex problems in a short way.

For instance, the bubble sort algorithm that, in a regular way, requires a nested for-loop and complex indexing.

- Alternatively, it can be solved with a few lines of code in a **recursive** function:

```python
def my_sort(li):
    for i in range(len(li)-1):
        if li[i] > li[i+1]:
            li[i], li[i+1] = li[i+1], li[i]
            li = my_sort(li)
    return li
```

HSLU Lucerne University of Applied Sciences and Arts

School of Computer Science and Information Technology
Research
Ramón Christen
Research Associate Doctoral Student

Phone direct +41 41 757 68 96
ramon.christen@hslu.ch

HSLU T&A, Competence Center Thermal Energy Storage
Research
Andreas Melillo
Lecturer

Phone direct +41 41 349 35 91
andreas.melillo@hslu.ch