2 – Lists and Tuples

Bálint Aradi

Course: Scientific Programming / Wissenchaftliches Programmieren (Python)

https://www.bccms.uni-bremen.de/people/b-aradi/wissen-progr/python/2023

Outline

- Tuples
- Lists
- Quick introduction to functions

Tuple

- Sequences of objects of arbitrary data type
- Items within a tuple can have different data type
- Delimited by **(** and **),** elements separated by **,**

```
t1 = (1, 3.0, "Hello")
t1
(1, 3.0, 'Hello')
```

- If non-ambiguous, the delimiters can be omitted

```
t1 = 1, 3.0, "Hello"
```

- **Empty tuple** is specified with **()**:

```
t0 = ()
t0
()
```

- For tuples with one element, a trailing comma is needed (to avoid ambiguousity):

```
t1bad = (1)
t1bad
1

t1good = (1,)
t1good
(1,)
```

- For tuples with more than one elements trailing comma can be added:

```
t1multi = (1, 2,)
t1multi
(1, 2)
```

Accessing tuple elements

- Tuple elements, tuple ranges can be accessed by the **[ ]** operator
- Works exactly as for substring/character selection in strings

```
t1
(1, 3.0, 'Hello')
t1[0]
1
t1[-1]
'Hello'
t1[1:3]
(3.0, 'Hello')
t1[::-1]
('Hello', 3.0, 1)
```

Negative indices count elements backwards:
-1 = last element

- Tuples are immutable, and can not be changed once they have been created

```
t1[0] = 24
… TypeError: …
```

Tuple operations

- Tuples can be appended with the **+** operator

```
t1 = (1, 2, 3)
t2 = (4, 5)
t3 = t1 + t2
t3
(1, 2, 3, 4, 5)
```

- Tuples can be repeated with the **+** operator

```
t4 = t2 * 3
t4
(4, 5, 4, 5, 4, 5)
```

- Number of items in a tuple can be queried by the **len()** function:

```
len(t4)
6
```

Tuple unpacking

- Components of a tuple can be assigned to individual variables within an assignment

```
mytuple = (1, 2)
t1, t2 = mytuple
t1
1
t2
2
```

Assigning entire tuple to one variable

Assigning tuple components to individual variables

- The number of variables on the left hand side must be compatible with the tuple length:

```
mytuple = (1, 2, 3)
t1, t2 = mytuple
ValueError: too many values to unpack (expected 2)
```

Lists

- Lists are very similar to tuples, but they are **mutable**
- Lists are delimited by **[** and **]**, lists elements are separated by **,**
- Element and range selection, **len()** function, operators **+** and ***** work analogously to tuples

```
l1 = [1, 3.0, 'Hello']
l1
[1, 3.0, 'Hello']
l1[0]
1
l1[-1]
'Hello'
l1[1:3]
[3.0, 'Hello']
l1[::-1]
['Hello', 3.0, 1]
```

```
len(t1)
3
l2 = []
len(l2)
0
l3 = [1, 4,]
l4 = l1 + l3
l4
['Hello', 3.0, 1, 1, 4]
l5 = l3 * 2
l5
[1, 4, 1, 4]
```

Modifying lists

- Changing elements

```
l1 = [3, 2, "test", 1.5]
l1
[3, 2, 'test', 1.5]
l1[0] = 42
l1
[42, 2, 'test', 1.5]
```

- Changing ranges

```
l1[0:2] = [1, -1]
l1
[1, -1, 'test', 1.5]
l1[0:4:2] = [0, 0]
l1
[0, -1, 0, 1.5]
```

- If the range is continuous, it can be replaced with a list (iterable) of arbitrary size. The size of the list will change accordingly

```
l1[0:3] = [9,]
l1
[9, 1.5]
len(l1)
2
```

- A given element or range can be deleted by the **del** statement

```
del l1[0]
l1
[1.5]
```

```
l3 = [1, 2, 3, 4]
del l3[0::2]
l3
[2, 4]
```

List methods

- The **append()** method can be used to append one element to the list

```
l5 = []
l5.append(1)
l5
[1]
```

```
l5.append(2)
l5
[1, 2]
```

- The **extend()** method can be used to extend the list by an other list (iterable)

```
l5.extend([3, 4, 5])
l5
[1, 2, 3, 4, 5]
```

or

```
l5 += [4, 5, 6]
l5
[1, 2, 3, 4, 5]
```

- Further methods for list manipulation
  - **insert()**, **index()**, **reverse()**, …
  - See **Python Standard Library documentation**: Sequence types

List methods

- Lists can be sorted by the **sort()** method:

```
ll =  [9, -1, 3, 8, 5]
ll.sort()
ll
[-1, 3, 5, 8, 9]
```

```
ll =  [9, -1, 3, 8, 5]
ll.sort(reverse=True)
ll
[-1, 3, 5, 8, 9]
```

- The **in** operator can be used to query for the presence of an element in the list

- It checks each list element individually, do not use it for large structures (*O(N)*)

```
l5
[1, 2, 3, 4, 5, 6]
3 in l5
True
-1 in l5
False
```

List unpacking

- Components of a list can be assigned to individual variables within an assignment

```
mylist = [1, 2]
l1, l2 = mylist
l1
1
l2
2
```

```
mylist = [1, 2, 3, 4]
l1, l2, *ll = mylist
l1
1
l2
2
ll
[3, 4]
```

Packs remaining entries into a list

- In function calls lists/tuples can be unpacked into arguments:

Unpacks entries into individual arguments

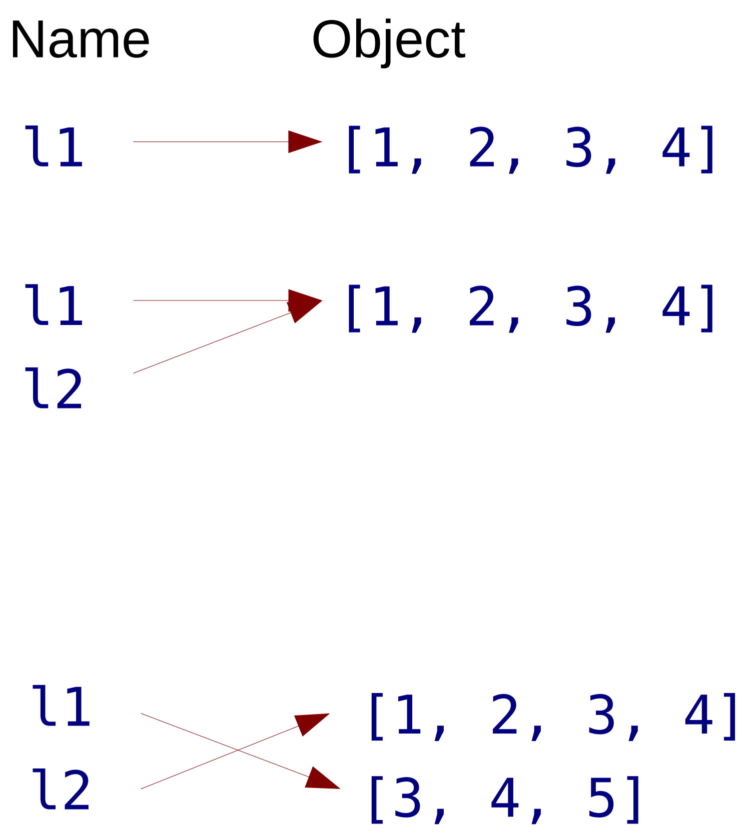
```
coords = (0, 10)
pen.goto(*coords)  # equiv. to: pen.goto(coords[0], coords[1])
```

Assignment of mutable types

- Analogous to immutable types

```
l1 = [1, 2, 3, 4]
l2 = l1
l1
[1, 2, 3, 4]
l2
[1, 2, 3, 4]

l1 = [3, 4, 5]
l1
[3, 4, 5]
l2
[1, 2, 3, 4]
```

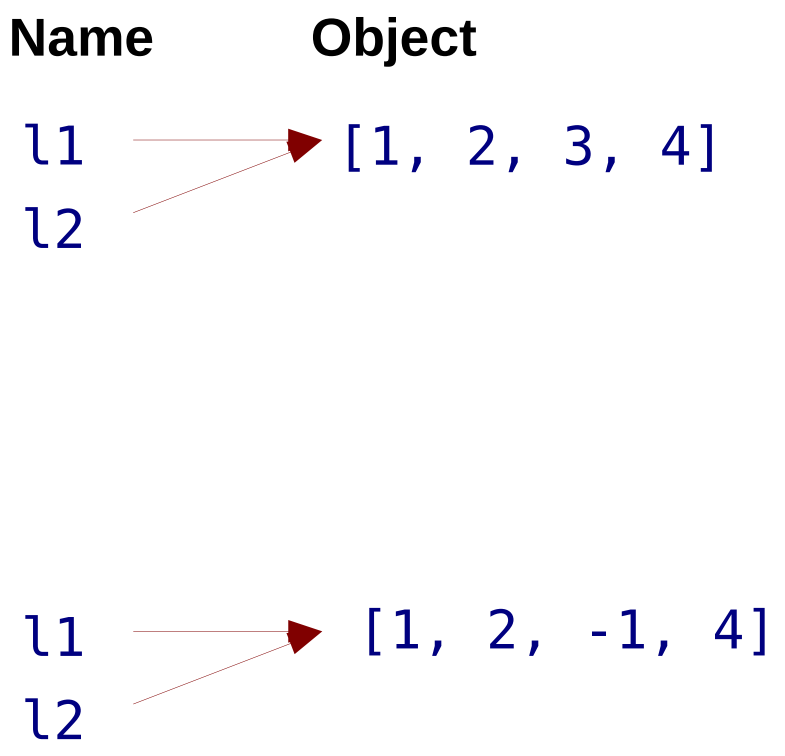


Assignment of mutable types

- If the content of a mutable variable is changed, the change is apparent in all variables, which are associated with that instance

```
l1 = [1, 2, 3, 4]
l2 = l1
l1
[1, 2, 3, 4]
l2
[1, 2, 3, 4]

l1[2] = -1
l1
[1, 2, -1, 4]
l2
[1, 2, -1, 4]
```



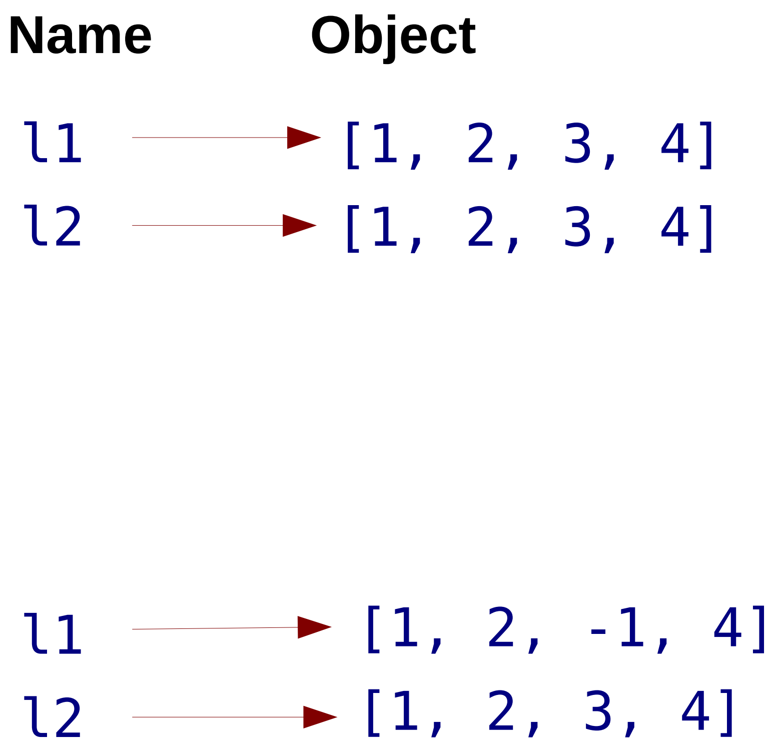
- Efficient, no copy is made
- Watch out for **unwanted side effects** with mutable types

Assignment of mutable types

- If a copy is needed, it must be explicetly created
- Try to avoid making copies, unless really necessary

```
l1 = [1, 2, 3, 4]
l2 = list(l1)
l1
[1, 2, 3, 4]
l2
[1, 2, 3, 4]

l1[2] = -1
l1
[1, 2, -1, 4]
l2
[1, 2, 3, 4]
```

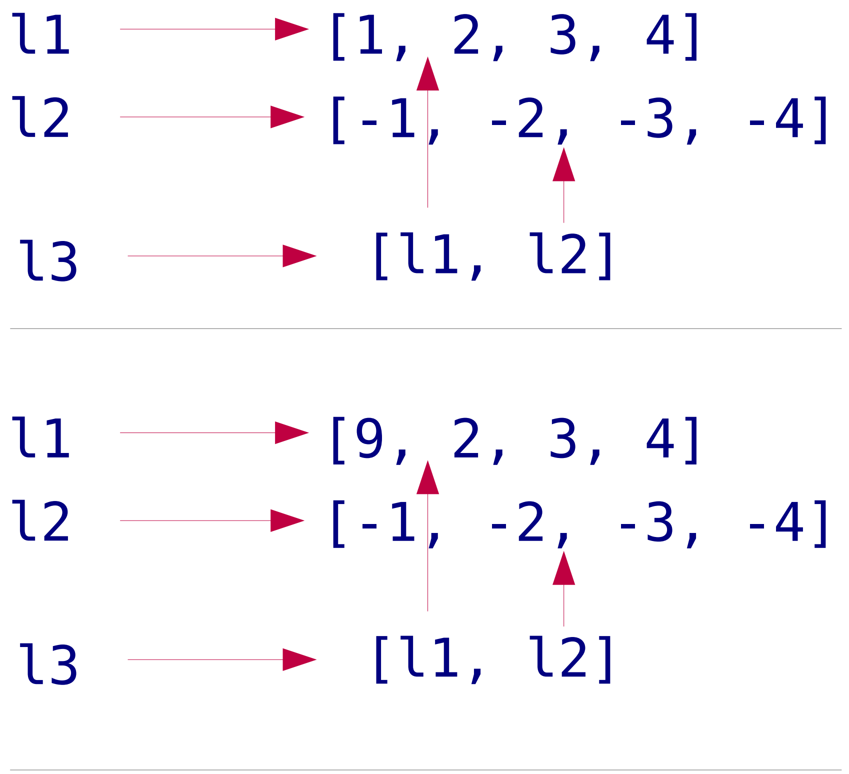


Assignment of mutable types

- If you copy a nested mutable object, only top layer is copied (**shallow copy**)

```
l1 = [1, 2, 3, 4]
l2 = [-1, -2, -3, -4]
l3 = list([l1, l2])
l3
[[1, 2, 3, 4], [-1, -2, -3, -4]]

l3[0][0] = 9
l3
[[9, 2, 3, 4], [-1, -2, -3, -4]]
l1
[9, 2, 3, 4]
```



- Function **deepcopy()** in module copy can be used, if true nested copy is needed

In-place operations

- In-place operations store the result of an arithmetic operation in the first operand:

```
aa += bb
aa -= bb
aa *= bb
aa /= bb
aa //= bb
```

```
aa = aa + bb
aa = aa - bb
aa = aa * bb
aa = aa / bb
aa = aa // bb
```

- For mutable objects it can help to avoid creating unnecessary copies

```
long = [1, 2, … ]
short = [-1, -2]
```

```
long = long + short
```

Creates temporary copy of long, extends it with short and replaces long with result

```
long += short
```

In-place addition (usually without temporary copy)

```
long.extend(short)
```

Extends list directly without temporary copy

Containers & iterators

- All containers can be used as iterators (e.g. in for-loops)
- Lists and tuples return their elements ordered by their index (position)

```
ll = [1, "test", 12.6, -1+3j]
for item in ll:
    print("Next item: ", item)
```

```
Next item:  1
Next item:  test
Next item:  12.6
Next item:  (-1+3j)
```

- Lists and tuples can be created from iterators
- Container will containt iterated elements

```
list('test')
['t', 'e', 's', 't']
```

Every string can be used as an iterator over the charaters in it

```
list((1, 2, 3, 4))
[1, 2, 3, 4]

tuple([1, 2, 3, 4])
(1, 2, 3, 4)
```

Enumerate

- If within an iteration you need both, the iterator value and the current iteration number
- **enumerate()** returns a new iterator over tuples containing the current iteration number and the value from the passed iterator

```
ll = [1, 'test', 12.6, (-1+3j)]
```

```
list(enumerate(ll))
```

```
[(0, 1), (1, 'test'), (2, 12.6), (3, (-1+3j))]
```

```
for ind in range(len(ll)):
    print(f"Item {ind:d}: {ll[ind]}")
```

equivalent

```
for ind, item in enumerate(ll):
    print(f"Item {ind:d}: {item}")
```

```
Item 0: 1
Item 1: test
Item 2: 12.6
Item 3: (-1+3j)
```

List comprehension

- Creates list with (slightly) modified or filtered content of an iterator

filtering is optional

```
[expr for itervar in iterator if condition]
```

Converts every character in a string to lowercase

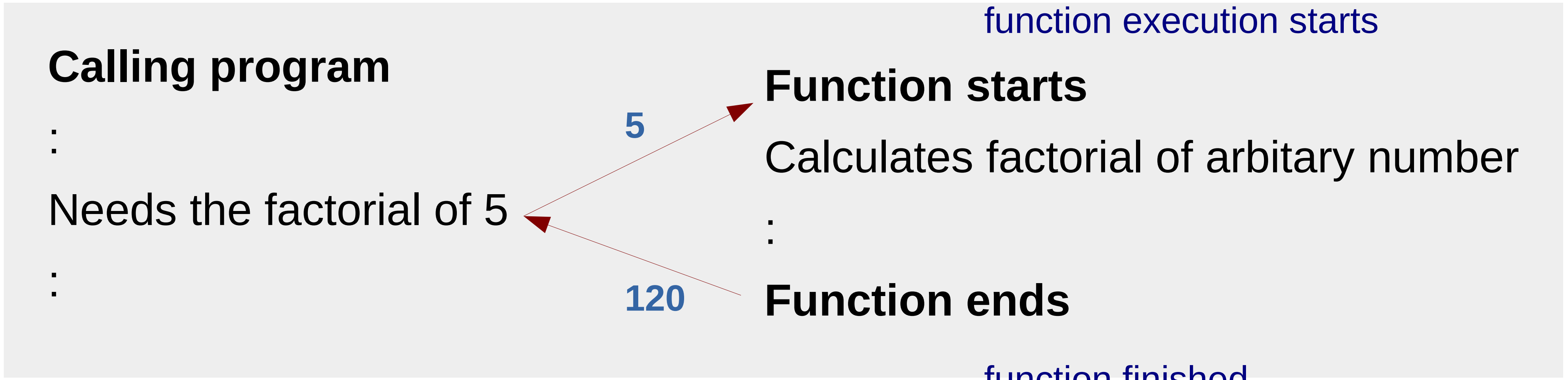
```
words = ["Wort", "Word", "WORT", "word"]
loweredwords = [word.lower() for word in words]
loweredwords
['wort', 'word', 'wort', 'word']
```

```
nums = [1, 3, 2, 9, 8, 3]
oddsquares = [num**2 for num in nums if num % 2 != 0]
oddsquares
[1, 9, 81, 9]
```

Functions in a nutshell

Function (procedure) = reusable code container, which communicates with other parts of the code only through a well defined interfaces

calling program suspended,
necessary information passed,
function execution starts



Calling program

:

Needs the factorial of 5

:

Function starts

Calculates factorial of arbitary number

:

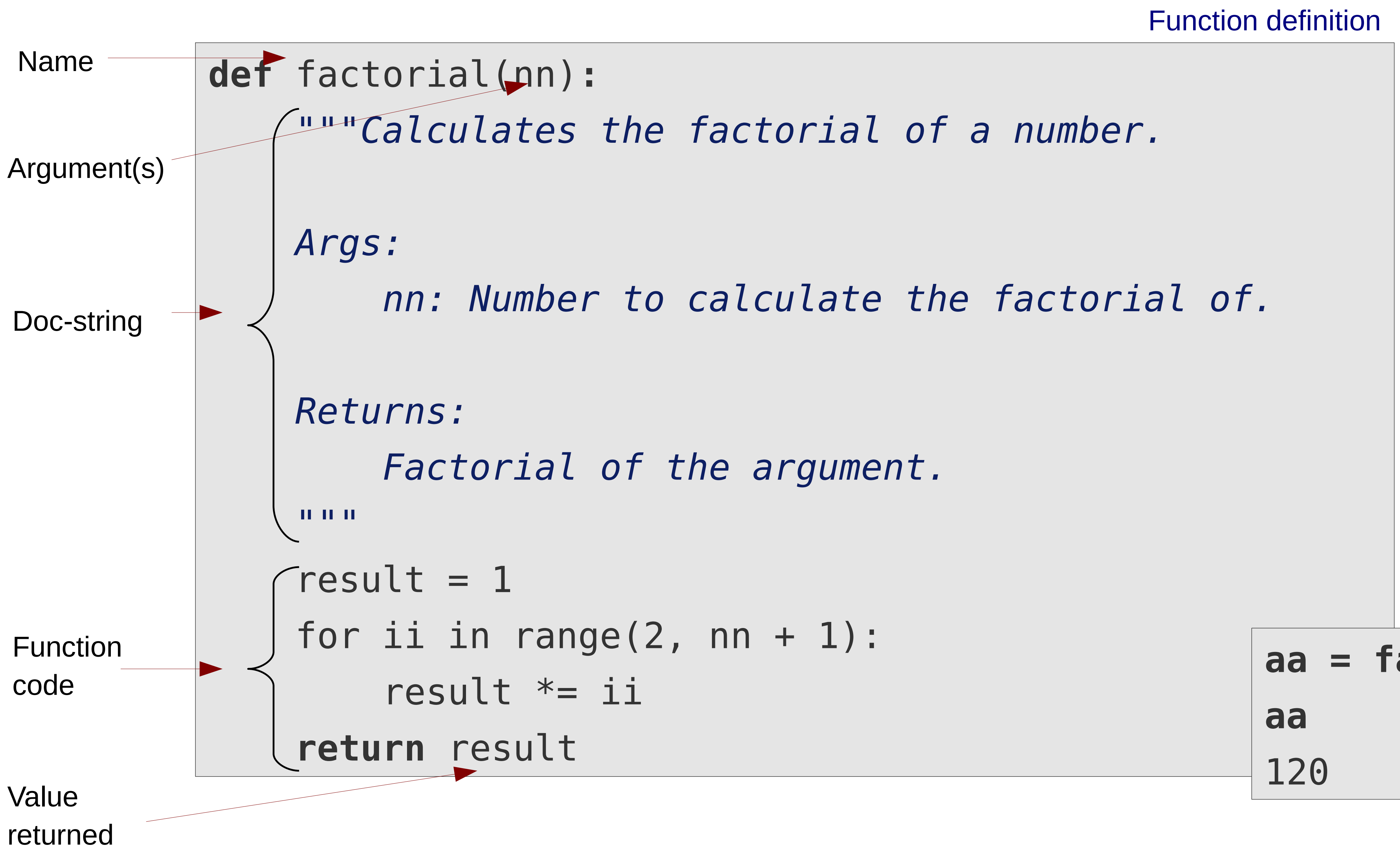
Function ends

function finished

result passed back

calling program continues

Functions in a nutshell



```python
def factorial(nn):
    """Calculates the factorial of a number.

    Args:
        nn: Number to calculate the factorial of.

    Returns:
        Factorial of the argument.
    """
    result = 1
    for ii in range(2, nn + 1):
        result *= ii
    return result
```

Calling code

```
aa = factorial(5)
aa
120
```

Functions in a nutshell

```
def functionname(arg1, arg2, ...):
    """Documentation string"""
    Subprogram statements
    ...
    return result
```

- Multiple arguments are possible

```
def multiply_numbers(aa, bb):
   return aa * bb
```

```
product = multiply_numbers(10, 12)
product
120
```

- Return value is optional

```
def print_greeting(name):
    print(f"Hello {name}!")
```

```
print_greeting("World")
Hello World!
```

Have fun!