Functions 2/2

Intro to Python

What we already know

- We define our own functions using the **def** keyword
- Functions are a way to avoid code duplication
- Functions help us make our code modular
- Functions must be defined before being called
- Declaring a function does not execute any code

```
def <name_function>(<parameters>):    # Function header
    <instruction1>
    <instruction2>                    # Function body
    ...
    return <value>                    # return is not required
```

```
1  def factorial(n):
2      a = 1
3      for k in range(1, n+1):
4          a *= k
5      return a
```

Scope of a variable

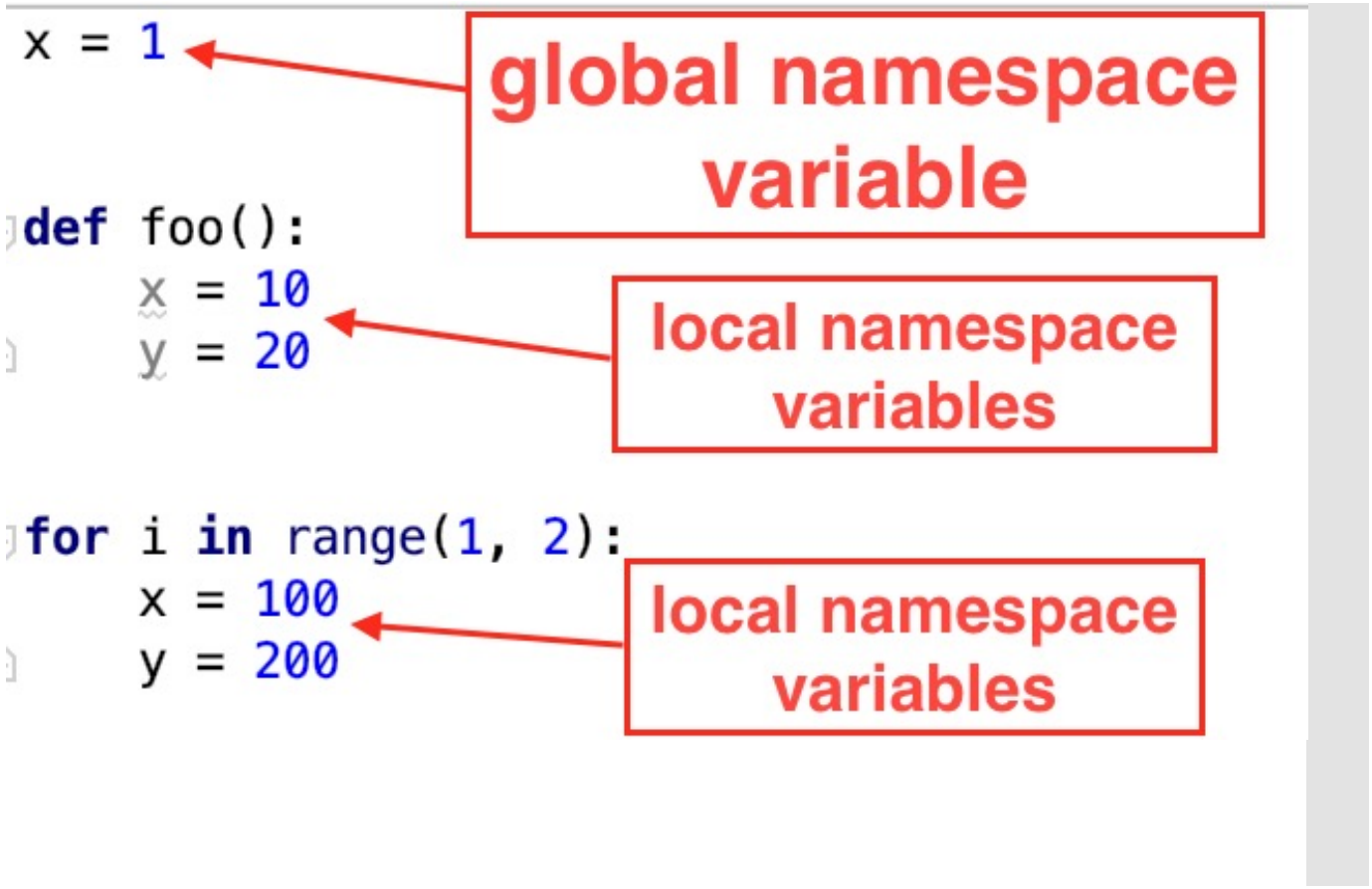
- A function can use a variable defined outside that function
- The variable needs to be defined before calling the function

```
1  a = 2
2
3  def f(x):
4      return a * x
5
6  f(5)
```

Scope of a variable

- Variables defined inside a function are local variables
- They do not exist outside that function
- If a variable in a function has the same name as a variable in the program (global), the function will use the value of the local variable
- In this case, the local and global variable share their name, but use different areas of the computer's memory

Scope of a variable



Scope of a variable

- What will this piece of code display? Can you explain?

```
:
1  def test():
2      b = 5
3      print(a, b)
4
5  a = 2    # from here we are outside the function
6  b = 7
7  test()
8  print(a,b)
```

Scope of a variable

- To reuse a global variable inside a function, we need to use the **global** keyword

- What will be displayed here?

```
1  def test():
2      global b
3      b = 5
4      print(a, b)
5
6  a = 2
7  b = 7
8  test()
9  print(a, b)
```

Parameter mutability

```
1  def add_one(x):
2      x += 1
```

```
1  a=2
2  add_one(a)
3  print(a)    # The variable a has not been modified
```

```
2
```

Parameter mutability

```
def add_zero(liste):
    liste.append(0)
```

```
lst = [1,2]
add_zero(lst)
print(lst)    # The variable has been modified
```

```
[1, 2, 0]
```

Parameter mutability

- Variables of immutable types can only be modified by assignment
- Variables of mutable types can be modified by a function
- Simple types are immutables:
  - integers
  - floats
  - strings
  - booleans
- Aggregated types are usually mutable:
  - Lists
  - Dictionaries (more on that soon)

Anonymous functions

- An anonymous, or lambda function, is an easy way to define a function in python
- It is an alternative to using **def**, but is also very powerful in complex software development
- We do not need to give a name to a lambda

```
def f(x):
    return x**2
```

```
f = lambda x: x**2
```

```
1 g = lambda x, y: x**2 + y**2
2 g(1,2)
```

```
1 (lambda x: 1/x)(4)
```

Functions as parameters of functions

```
1  def is_greater_than(f,a,b):      # f is any function
2      if f(a) > f(b):
3          return True
4      else:
5          return False
```

```
1  is_greater_than(lambda x:1/x,1,2)
```

Function and sequences

- Python functions can be used to compute the terms of a mathematical sequence
- This works for explicit sequences or sequences defined recursively

Explicit sequences

Let the sequence (U_n) be defined on $\mathbb{N}$ by:

$\forall n \; in \mathbb{N}, U_n = 2n - 1$

This sequence is defined explicitly.

For any natural number n, we have $U_n = f(n)$ where f is the affine function $x \to 2x - 1$.

> To code this suite, just code the affine function in Python

```
def U(n):
    return 2*n-1
```

or

```
U = lambda n : 2*n-1
```

The calculation of each term is direct, for example to display the index term 83:

```
print(U(83))    # displays 165
```

Recursive sequences

We consider the sequence u defined on $\mathbb{N}$ by :

$$\begin{cases} u_0 = 2 \\ \forall n \in \mathbb{N}, U_{n+1} = 2U_n - 1 \end{cases}$$

:

```python
def u(n):
    # 1. We associate the first term with a variable.
    u0 = 2
    # 2. If the term requested during the call to the function is equal to the first term, it is returned.
    if n == 0:
        return u0
    else:
        # 3. Otherwise we create a loop that goes from the second term to rank n + 1
        for i in range(1,n+1):
            # In this loop, we apply the function associated with the first term and we overwrite its value
            u0 = 2*u0-1
        # 5. At the end of the loop we return the value thus calculated.
        return u0

print(u(0))
print(u(1))
print(u(2))
```

```
2
3
5
```

Double recurrence

We define the sequence a_n by:

$$\begin{cases} a_0 = a_1 = 1 \\ \forall n \in \mathbb{N}, a_{n+2} = a_{n+1} + 2a_n \end{cases}$$

```python
def a(n):
    # 1. We associate the first two terms with variables
    a0 = a1 = 1
    # 2. If the term requested when calling the function is equal to the first term or to the second term
     # we send it back.
    if n == 0:
        return a0
    elif n == 1:
        return a1
    else:
        # 3. Otherwise we create a loop that goes from the third term to rank n + 1
        for i in range(2, n+1):
            # We apply the recurrence relation to the first two terms.
             # We store the result in a new variable.
            a = a1 + 2*a0

            # u0 takes the value of u1
            a0 = a1

             # u1 takes the value of u
            a1 = a

        return a1

print(a(1))
print(a(2))
print(a(3))
```

```
1
3
5
```

Arithmetic sums

Mathematical notation uses a symbol that represents the **sum of a series of terms**: the summation symbol **Σ**, an expanded form of the capital Greek letter sigma. This is defined as follows :

$$\sum_{i=m}^{n} a_i = a_m + a_{m+1} + a_{m+2} + \ldots + a_n$$

Let's illustrate this through an example; here is an example showing a sum of squares.

$$\sum_{i=3}^{6} i^2 = 3^2 + 4^2 + 5^2 + 6^2 = 86$$

```
1  def sum_arithmetic(inf, sup):
2      somme = 0
3      for i in range(inf, sup+1):
4          somme += i**2
5      return somme
6
7  sum_arithmetic(3,6)
```

```
86
```