Partie I

Introduction à la programmation

Informatique 1
Introduction à la programmation

Mission 3 : RESTRUCTURATION

Fonctions, portée des variables, spécifications, modules

Kim Mens Siegfried Nijssen Charles Pecheur

Définition de **fonction**

Une fonction qui retourne la factorielle de n : *n! = n.(n-1).(n-2).....1*

```
def fact(n):
    """
    pre:  n ≥ 0
    post: retourne la factorielle de `n`
    """
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
```

(1) En-tête

(2) Spécifications

(3) Corps

(4) return

```
>>> fact(4)
24
```

appel

4*3*2*1

Paramètres et arguments

```
def fact(n):
    """
    pre:  n ≥ 0
    post: retourne la factorielle de `n`
    """
    val = 1
    for i in range(2, n+1):
        val *= i
    return val

nbre = 4
permuts = fact(nbre)
```

Paramètre = une **variable**

(1) En-tête

Appel de fonction

Argument = une **expression**

docstring

docstring = **DOC**umentation **STRING**s

On les utilise pour **spécifier** le comportement d'une fonction, d'une méthode, d'une classe, d'un module

(2) Spécifications

```
def fact(n):
    """
    pre:  n ≥ 0
    post: retourne la factorielle de `n`
    """
    val = 1
    for i in r
        val *=
    return val

print(fact.__doc__)
```

Vous pouvez accéder à cette documentation à n'importe quel moment dans votre code grâce à la méthode `__doc__`

```
pre:  n ≥ 0
post: retourne la factorielle de `n`
```

Fonction **avec** résultat

fonction fructueuse
Ch. 8 – Fruitful functions

```
def fact(n):
    """
    pre:  n ≥ 0
    post: retourne la factorielle de `n`
    """
    val = 1
    for i in range(2, n+1):
        val *= i
    return val

nbre = 4
permuts = fact(nbre)
```

retour avec **résultat**
une **expression**

appel
une **expression**

Fonction **sans** résultat

```
def facteur(n):
    """
    pre:  n entier > 0
    post: imprime le plus grand
          facteur propre de `n`
    """
    for i in range(n-1, 0, -1):
        if n%i == 0:
            print(i)
            return

nbre = 39
facteur(nbre)
```

retour sans résultat

appel

une **instruction**

Afficher versus Retourner

```
def affiche_somme(a, b):
    """
    pre:  -
    post: Affiche la somme
          de a et b
    """
    print(a+b)

affiche_somme(25, 34)
```

59

Fonction **sans** résultat

```
def somme(a, b):
    """
    pre:  -
    post: Retourne la somme
          de a et b
    """
    return a+b

s = somme(25, 34)
print(s)
s2 = s * s
```

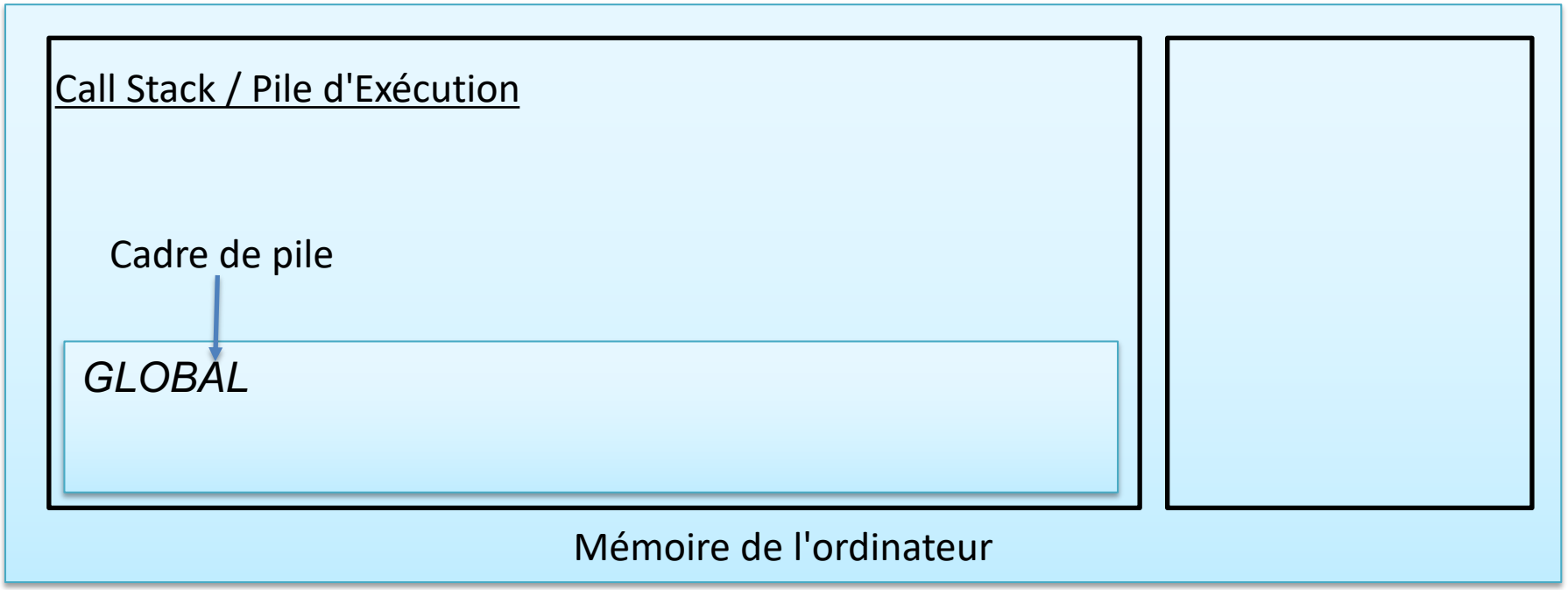
59

Fonction **avec** résultat

SINF

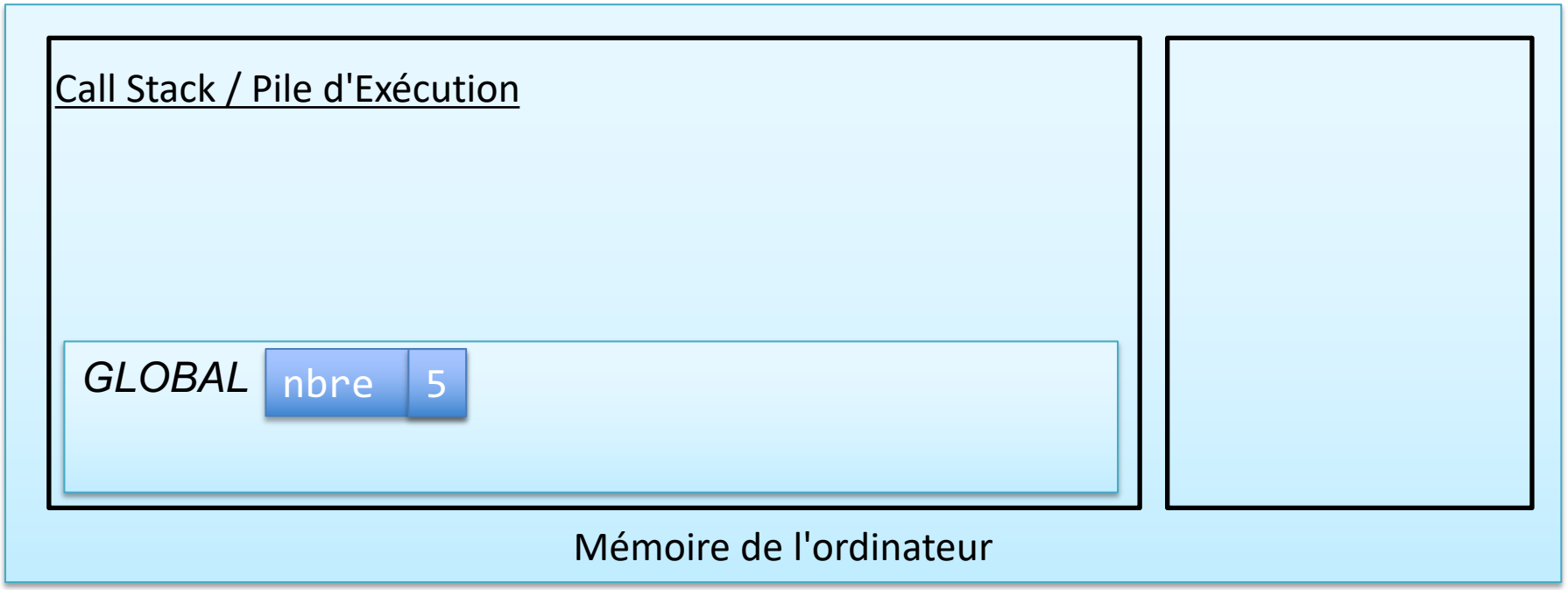
Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```



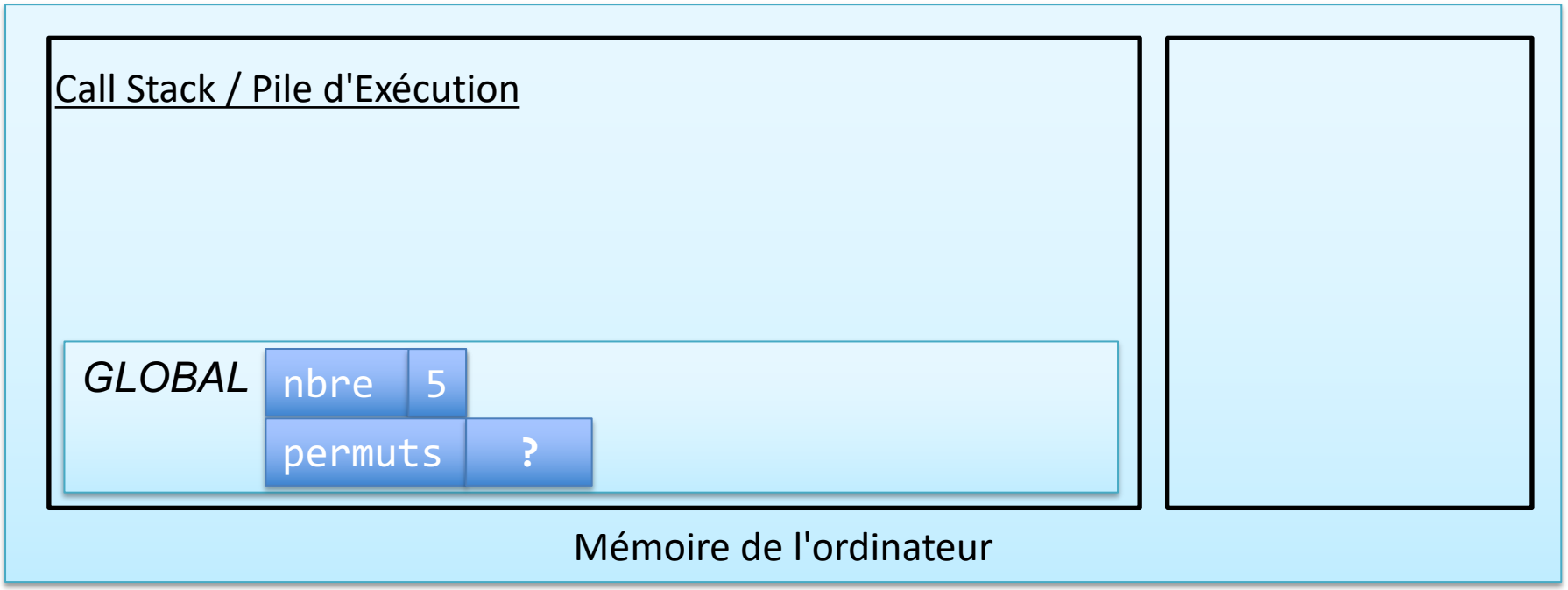
Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```



Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```



Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```

Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```

Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```

Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```

Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```

Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```

Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```

Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```

Exécuter une fonction

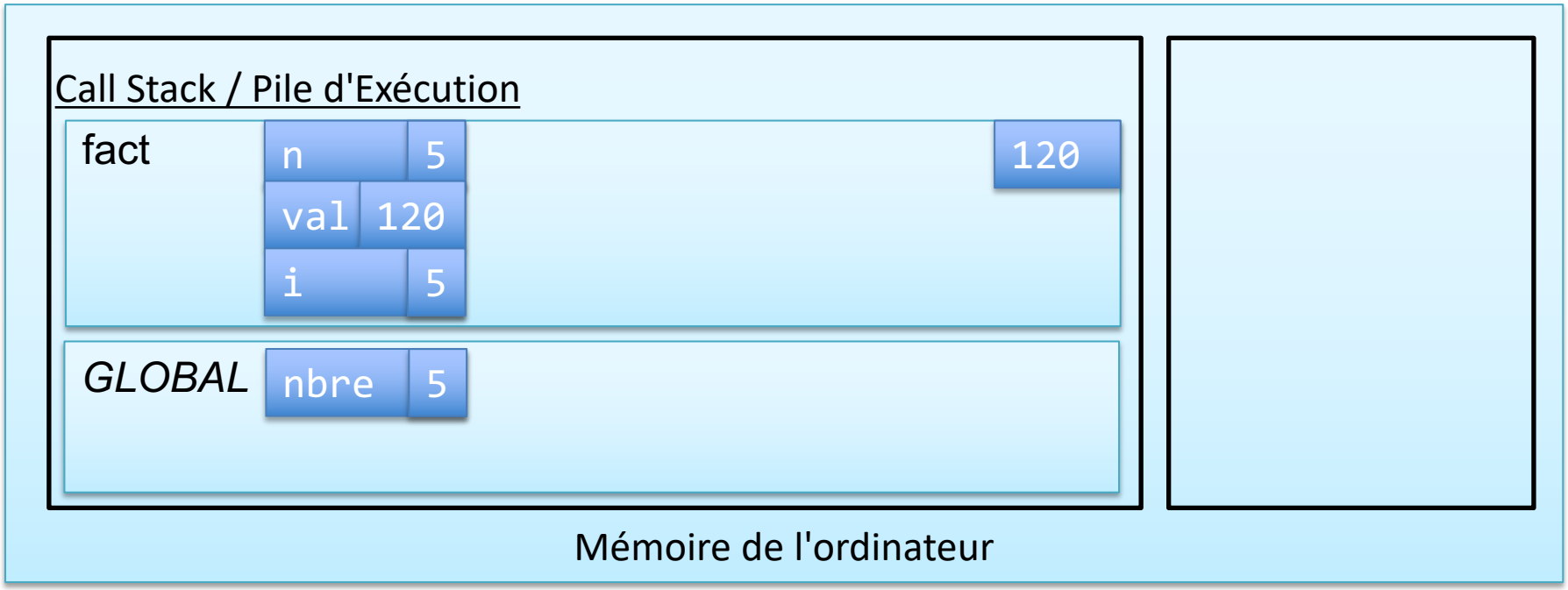
```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```

Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```

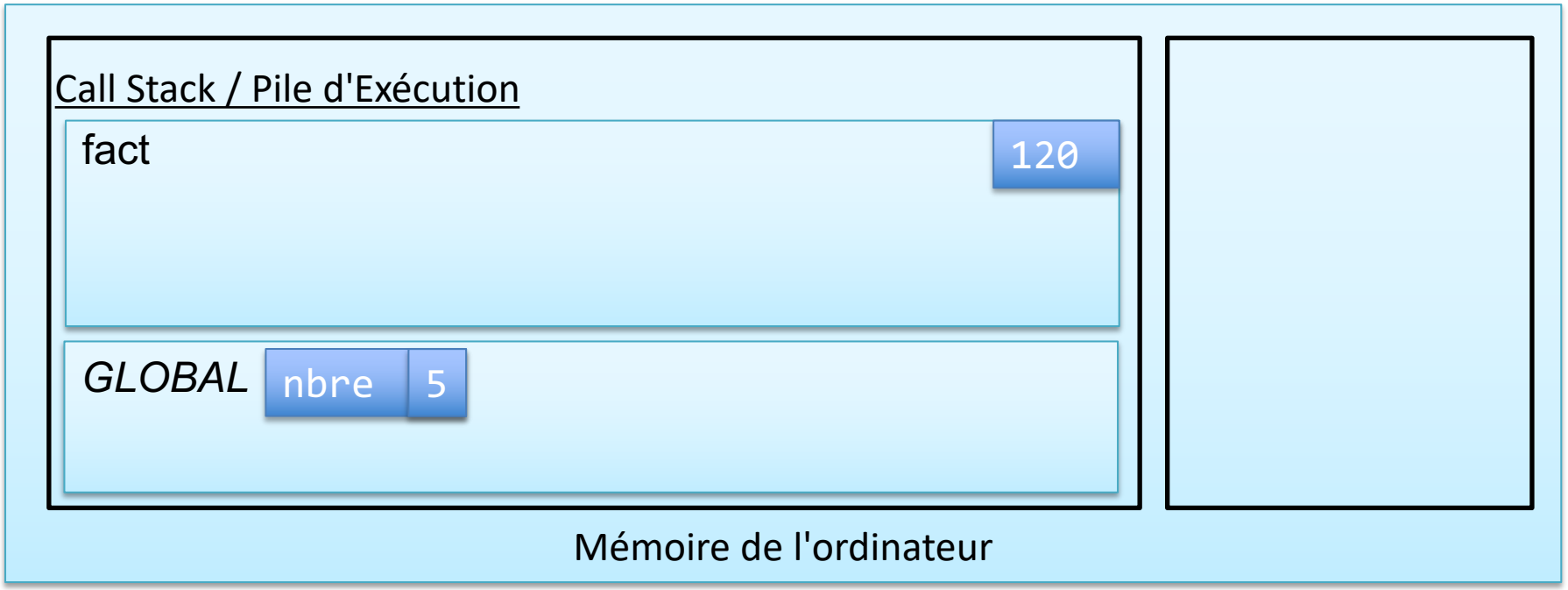
Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val

nbre = 5

permuts = fact(nbre)
```



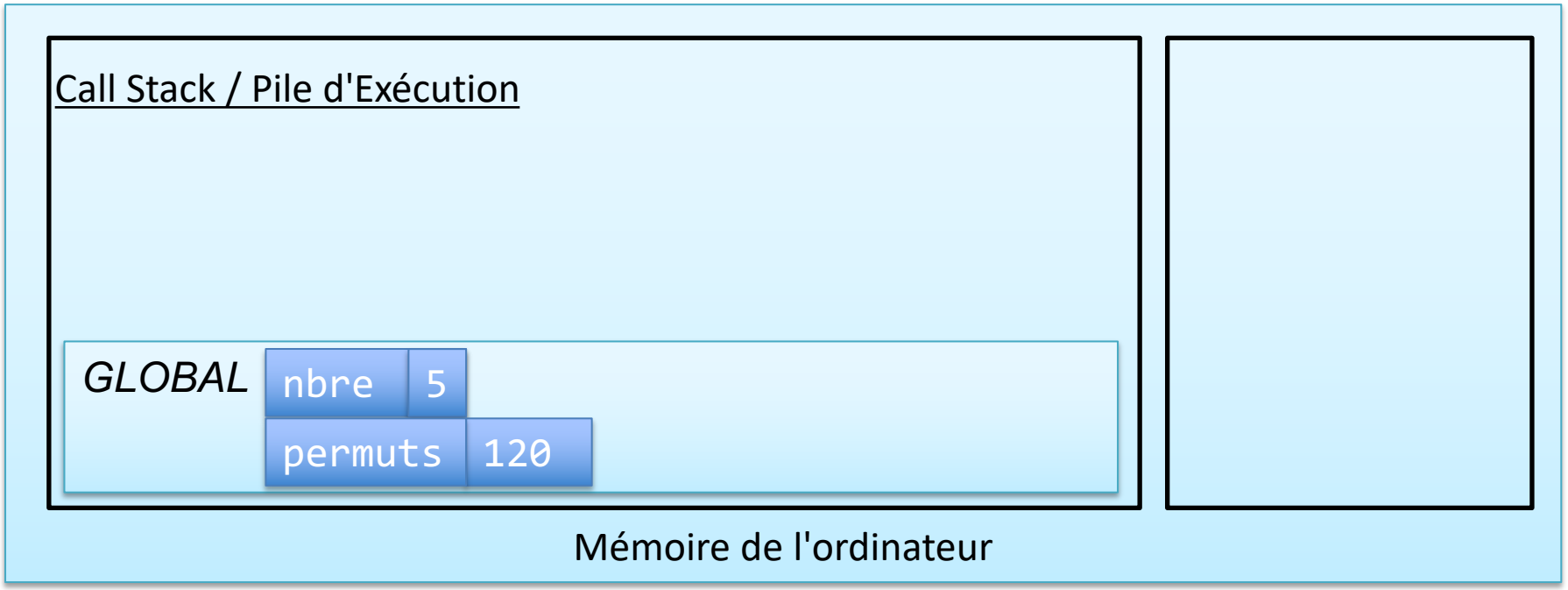
Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```



Exécuter une fonction

```
def fact(n):
    val = 1
    for i in range(2, n+1):
        val *= i
    return val
nbre = 5
permuts = fact(nbre)
```



Variables locales

```
c = 100
val = bits(c)
print(c)
print(val)
```

```
def bits(n):
    """ ... """
    c = 0
    while n > 0:
        n //= 2
        c += 1
    return c
```

calculer combien de bits sont nécessaires pour stocker un entier donné

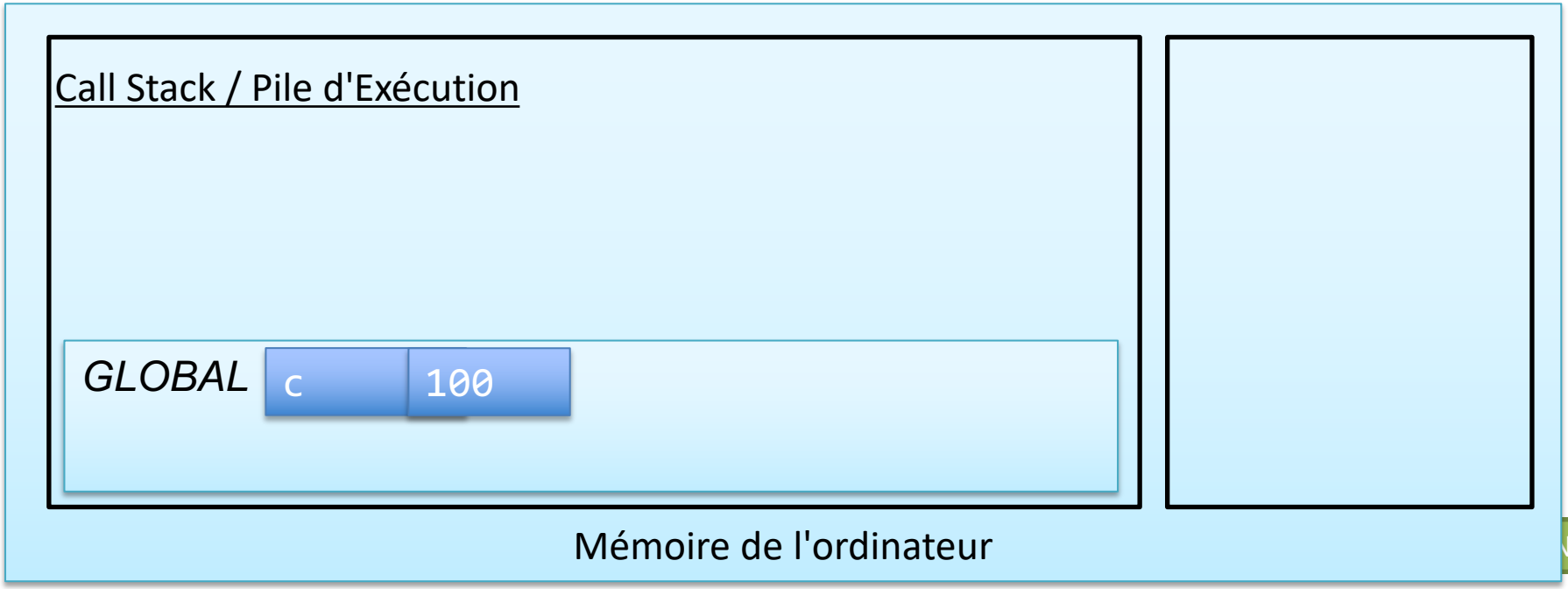
2^6	2^5	2^4	2^3	2^2	2^1	2^0
1	1	0	0	1	0	0

64 + 32 + 4 = 100

Variables locales

```
c = 100
val = bits(c)
print(c)
print(val)
```

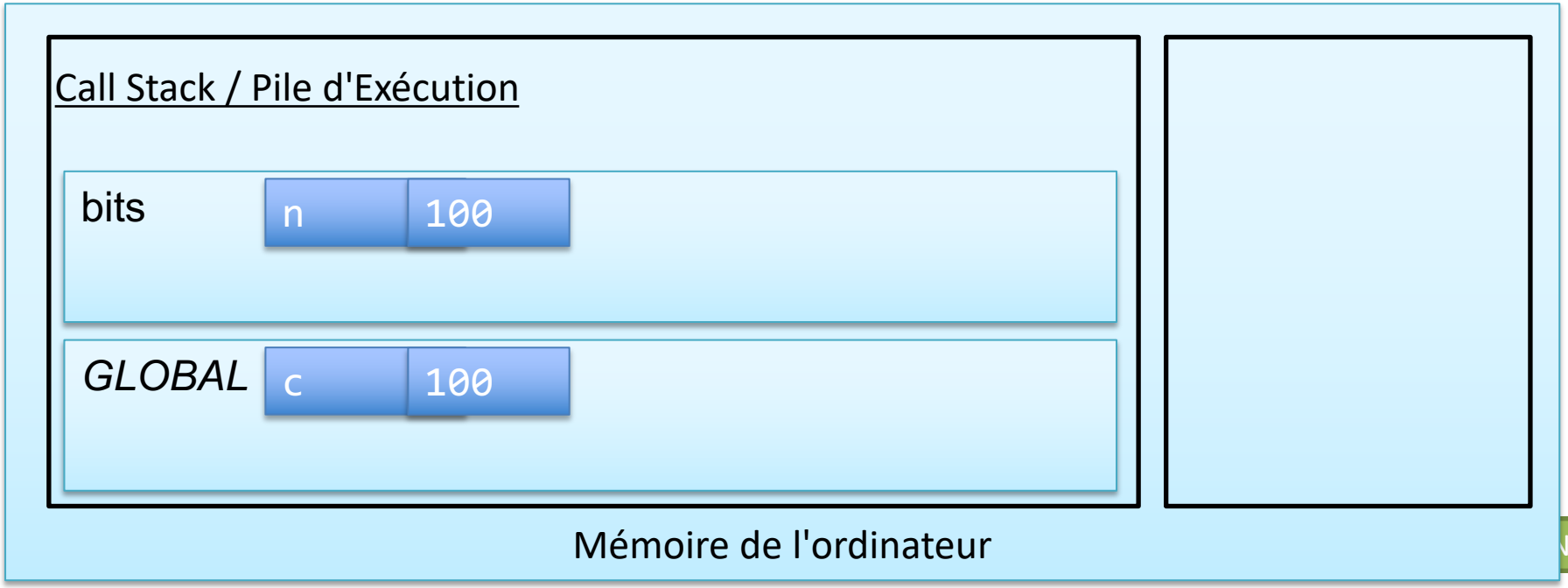
```
def bits(n):
    c = 0
    while n > 0:
        n //= 2
        c += 1
    return c
```



Variables locales

```
c = 100
val = bits(c)
print(c)
print(val)
```

```
def bits(n):
    c = 0
    while n > 0:
        n //= 2
        c += 1
    return c
```

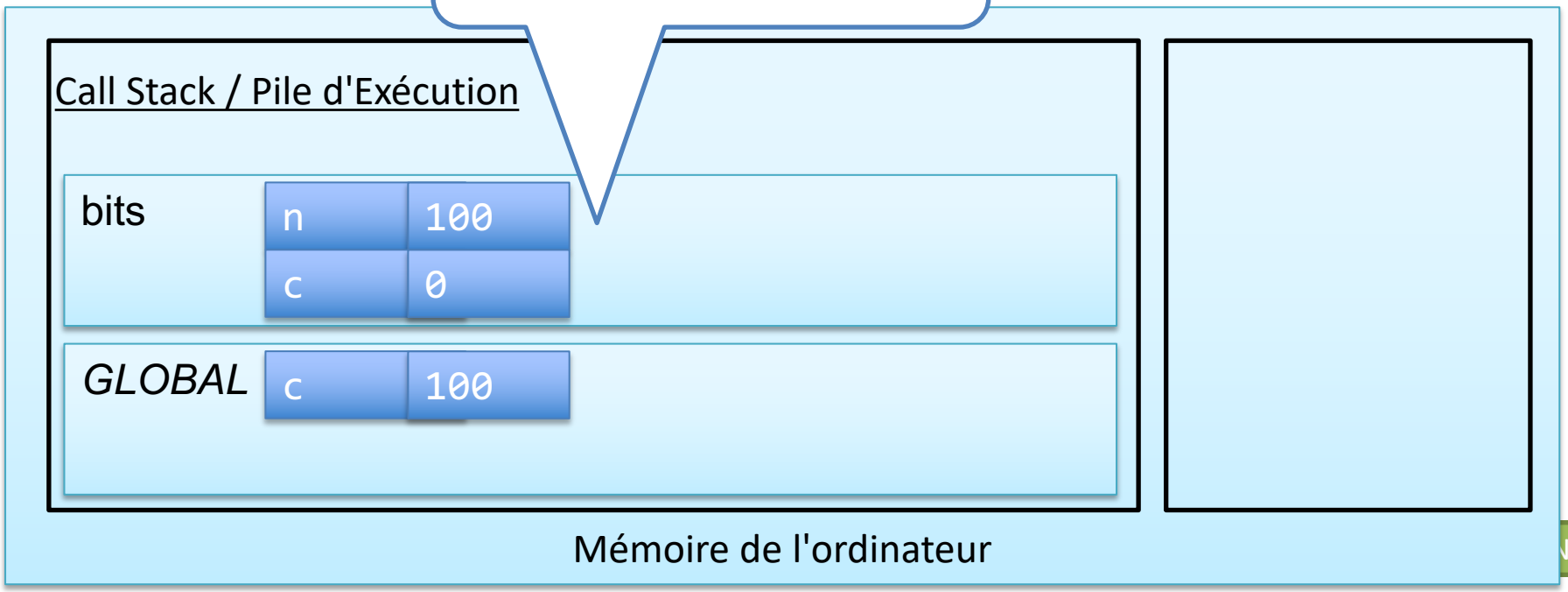


Variables locales

```
c = 100
val = bits(c)
print(c)
print(val)
```

```
def bits(n):
    c = 0
    while n > 0:
        n //= 2
        c += 1
    return c
```

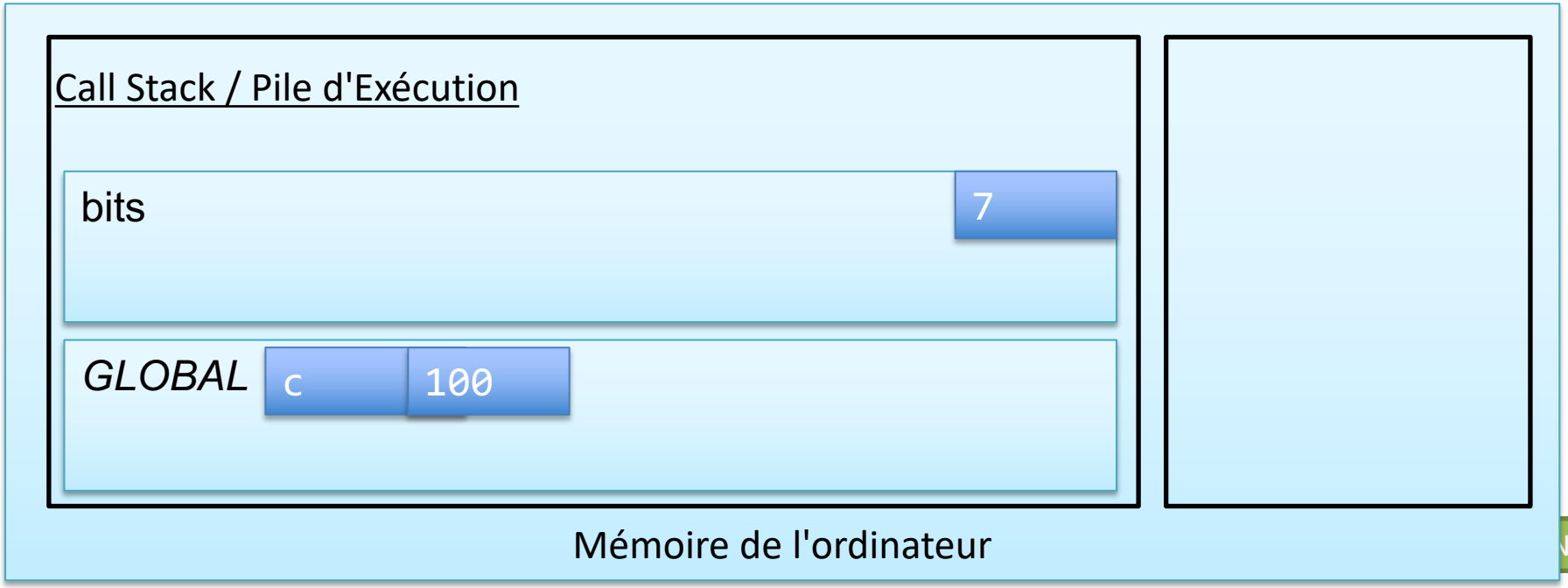
n et **c** sont **locales** dans la fonction `bits`



Variables locales

```
c = 100
val = bits(c)
print(c)
print(val)
```

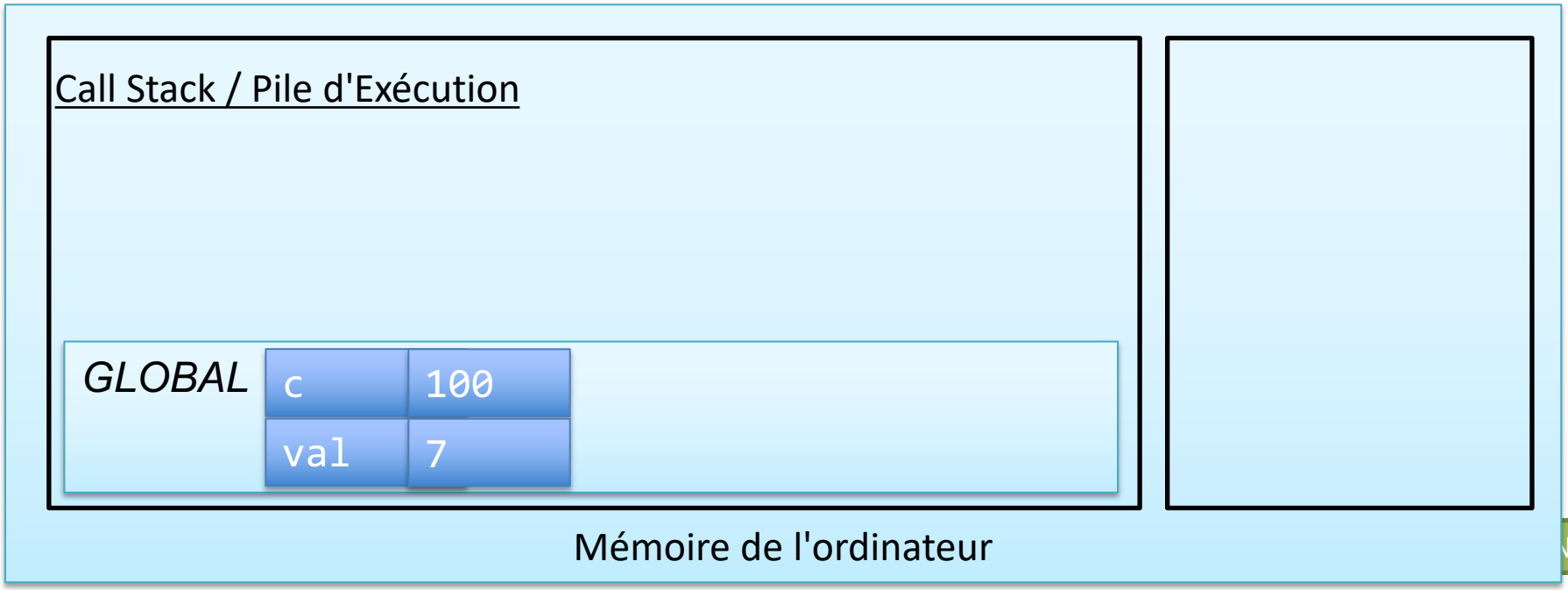
```
def bits(n):
    c = 0
    while n > 0:
        n //= 2
        c += 1
    return c
```



Variables locales

```
c = 100
val = bits(c)
print(c)
print(val)
```

```
def bits(n):
    c = 0
    while n > 0:
        n //= 2
        c += 1
    return c
```



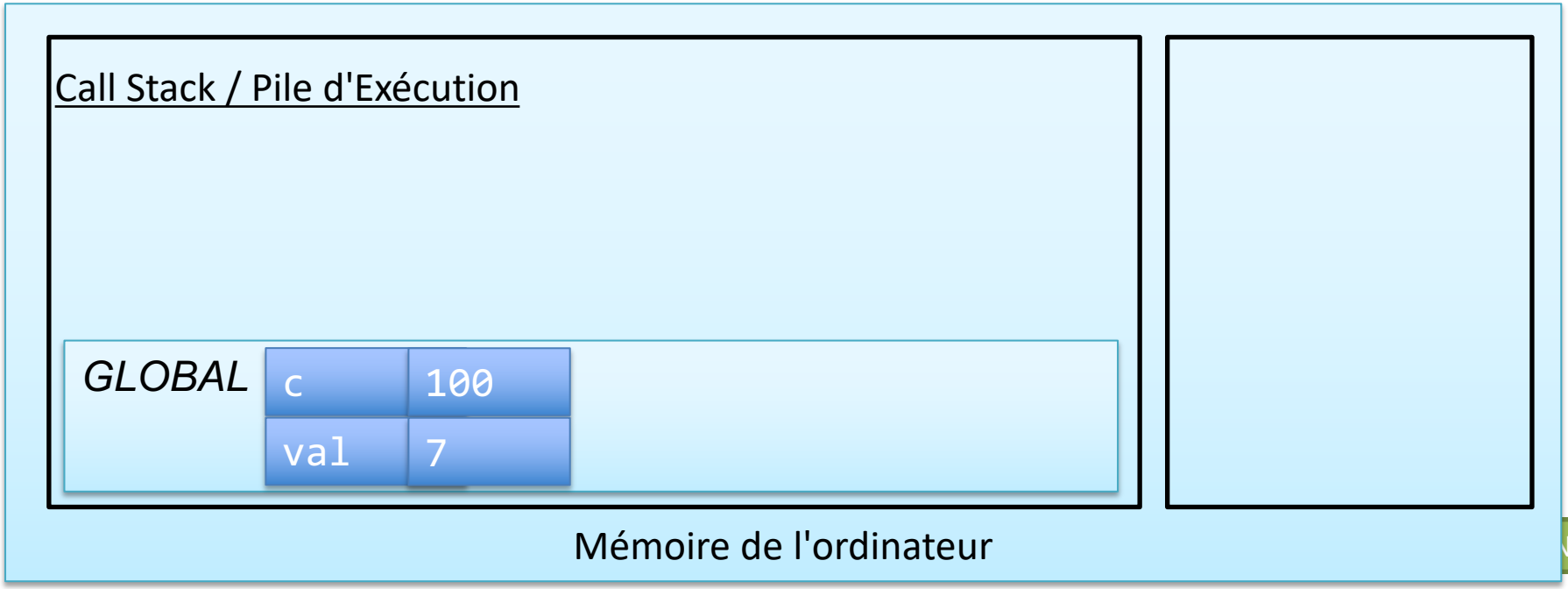
Variables locales

```
c = 100
val = bits(c)
print(c)
print(val
```

100

7

```
def bits(n):
    c = 0
    while n > 0:
        n //= 2
        c += 1
    return c
```



Variables globales

```
taux_tva = 21/100    # 21%
total = 0

def facture(htva):
    global total
    tvac = htva * (1 + t
    total += tvac
    return tvac

print(total)
print(facture(200))
print(facture(500))
print(total)
```

`taux_tva` et `total` sont **globales**

`htva` et `tvac` sont **locales**

pour **affecter** localement une variable globale
(PAS recommandé!!!)

0

242.0

605.0

847.0

Spécification d'une fonction

Pré-conditions

Les conditions sous lesquelles la fonction est applicable

- sur les **données** (valeurs des **paramètres**)
  `pre: n > 0`
- sur l'**état initial**
  `pre: la tortue est orientée au nord`

Post-conditions

Ce que fera *ou* retournera la fonction exécutée

- sur le **résultat** (valeur **retournée**)
  `post: retourne la factorielle de n`
- sur l'**état final**
  `post: la tortue a dessiné un sapin`

Fonction correcte

La **spécification** est un **contrat** entre la fonction et ses utilisateurs.

SI les **pré-conditions** sont satisfaites **avant**
ALORS les **post-conditions** sont satisfaites **après**

La fonction est **correcte** si elle **satisfait le contrat**.

Spécification

Cette fonction est-elle correcte par rapport à sa spécification ?

OUI
(mais peu utile)

```
def minimum(a, b):
    """
    pre:  a > b
    post: retourne le minimum entre a et b
    """
    return b
```

Spécification

Cette fonction est-elle correcte par rapport à sa spécification ?

OUI
(de manière absurde !)

```
def magritte(x):
    """
    pre:  x < 0 et x > 10
    post: retourne un chapeau melon
    """
    return "BOUH!"
```

Quand les poules auront des dents,
Alors les vaches pondront des oeufs

Spécification

Cette fonction est-elle correcte par rapport à sa spécification ?

NON

```
def log2(n):
    """
    pre:  --
    post: retourne k tel que n == 2**k
    """
    k = 0
    while 2**k != n:
        k += 1
    return k
```

```
>>> log2(64)
6
>>> log2(4)
2
>>> log2(10)
```

ne se termine pas

Fonction correcte et terminaison

La **spécification** est un **contrat**
La fonction est **correcte** si elle **satisfait le contrat :**

SI les **pré-conditions** sont satisfaites **avant**
ALORS l'exécution de la fonction se **termine**
ET les **post-conditions** sont satisfaites **après**

Une fonction qui ne se **termine pas**
n'est **jamais correcte**

Modules

```
import math
print(math.pi)
3.141592653589793
print(math.sqrt(2.0))
1.4142135623730951
```

math.py

```
pi = 3.14159…

def sqrt(x):
    …

def cos(x):
    …
```

```
import time
t0 = time.clock()
for i in range(10**8): x = i*i
t1 = time.clock()
print(round(t1-t0, 3))
12.603
```

time.py

```
def clock():
    …
```

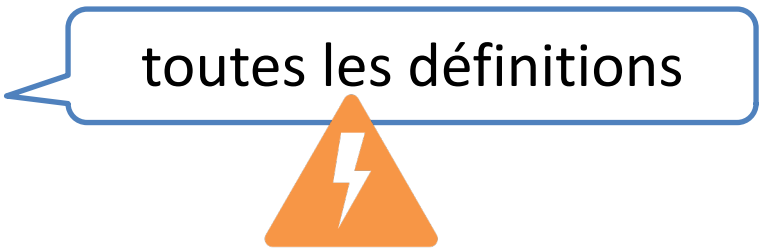
Importer

```
import turtle
tortue = turtle.Turtle()
```

```
import turtle as ttl
tortue = ttl.Turtle()
```

```
from turtle import Turtle
tortue = Turtle()
```

```
from turtle import *
tortue = Turtle()
```



import *

```
e = 0.001
def gamma(s):
    return 10*s

from math import *
print(e)
2.718281828459045
print(gamma(0.5))
1.7724538509055159
```

re-définit e et gamma !

Objets et méthodes

```
from turtle import Turtle

alice = Turtle()
bob = Turtle()

alice.color("blue")
bob.color("red")
alice.forward(50)
bob.circle(50, 90)
```

Deux **objets instances** de `Turtle`

Appels de **méthodes** de la classe `Turtle`