Cours 3: Fonctions

Nguyễn Kim Thắng

kimthang.nguyen@univ-evry.fr

Mails: [L3INF]

bureau 209, IBGBI
IBISC, Univ. Evry, University Paris-Saclay

Fonctions en Python

```
def fonction_nom(param1, param2):
    ...
    return quelques_valeurs
```

Fonctions en Python

```
def fonction_nom(param1, param2):
    ...
    return quelques_valeurs
```

- Toutes fonctions retournent des valeurs
  - Retourner None si rien n’est précisé

Fonctions en Python

```
def fonction_nom(param1, param2):
    ...
    return quelques_valeurs
```

- Toutes fonctions retournent des valeurs
  - Retourner None si rien n'est précisé
- Une fonction peut retourner plusieurs valeurs
  - Utiliser les tuples

```
return val1, val2, val3
```

Execution et Portée des fonctions

- L'exécution d'une fonction: introduire une table des symboles locales
  - une dictionnaire associant noms aux valeurs

Execution et Portée des fonctions

- L'exécution d'une fonction: introduire une table des symboles locales
  - une dictionnaire associant noms aux valeurs

- **Affectation des variables**: ajouter un élément à la table des symboles ou re-écrire les éléments existants

```
x = 10
```

Execution et Portée des fonctions

- L'exécution d'une fonction: introduire une table des symboles locales
  - une dictionnaire associant noms aux valeurs

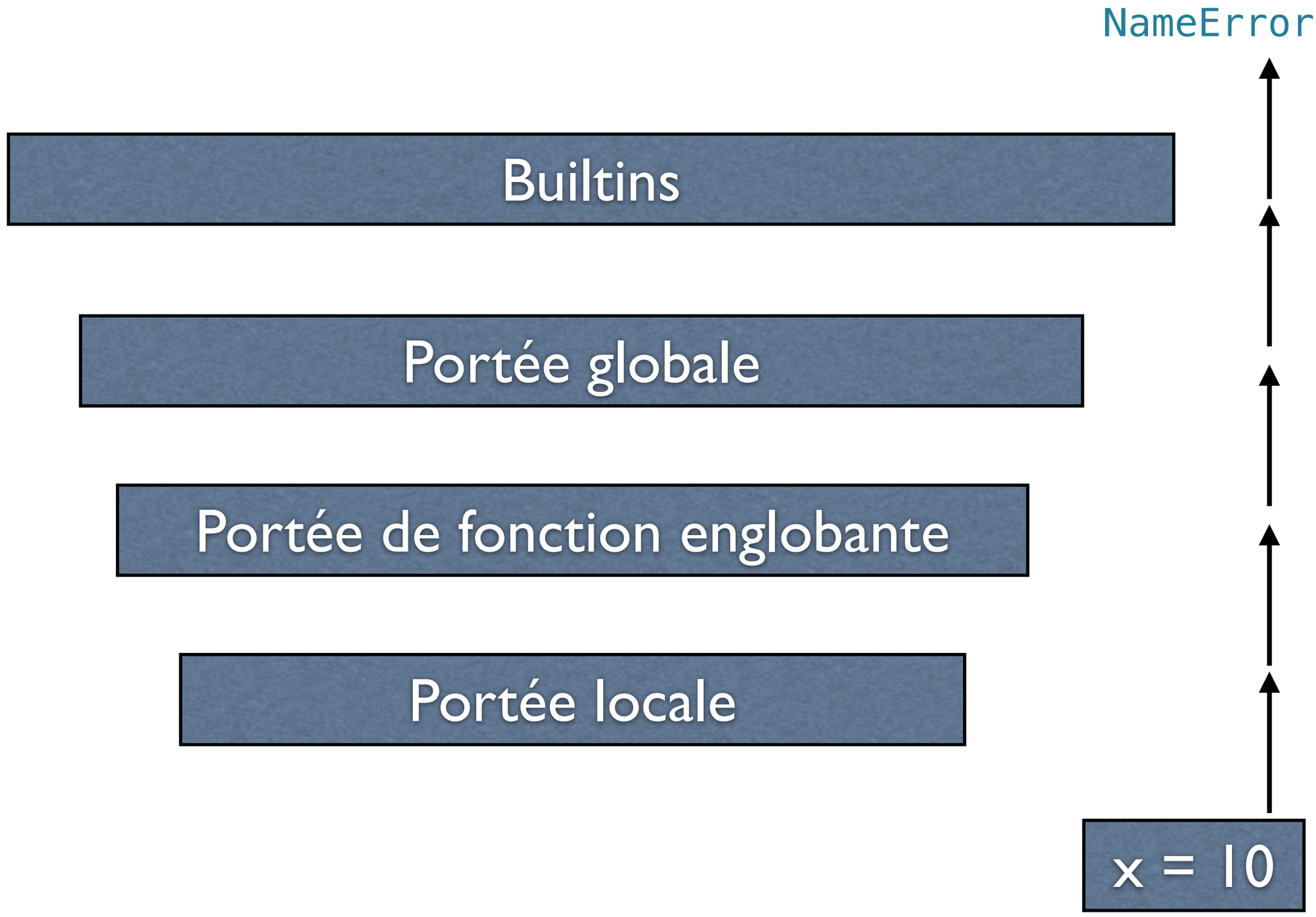
- **Affectation des variables**: ajouter un élément à la table des symboles ou re-écrire les éléments existants

```
x = 10
```

- **Références des variables**: vérifier la tour des portées (scopes)

```
print(x)
```

Résolution des variables



Portées des fonctions: exemple

```
x = 10

def foo(y)
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)

foo(3)
```

Portées des fonctions: exemple

```
x = 10

def foo(y)
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)

foo(3)



# {'y': 3, 'z': 5}
# 10
# 10 3 5
```

Portées des fonctions: exemple

```
x = 10

def foo(y)
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)

foo(3)


# {'y': 3, 'z': 5}
# 10
# 10 3 5
```

```
x = 10

def foo(y)
    x = 20
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)

foo(3)
```

Portées des fonctions: exemple

```
x = 10

def foo(y)
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)

foo(3)


# {'y': 3, 'z': 5}
# 10
# 10 3 5
```

```
x = 10

def foo(y)
    x = 20
    z = 5
    print(locals())
    print(globals()['x'])
    print(x, y, z)

foo(3)


# {'y': 3, 'z': 5}
# 10
# 20 3 5
```

Portées et Espace des noms

- Un espace des noms (namespace) est une dictionnaire affectant un nom (str) à une valeur

- Une portée définie l'ordre des espaces des noms pour la recherche

Portées et Espace des noms

- Un espace des noms (namespace) est une dictionnaire affectant un nom (str) à une valeur

- Une portée définie l'ordre des espaces des noms pour la recherche

- Les portées peuvent se chevaucher mais l'intersection entre deux espaces des noms est vide.

- Seuls les fonctions (et les classes) créent de nouvelles portées et espaces des noms.

Nouvelle portée

- if, for, while, … ne créent pas de nouvelle portée

```
if succes:
    message = "Gagnant"
else
    message = "Perdant"

print(message)
```

Nouvelle portée

- if, for, while, … ne créent pas de nouvelle portée

```
if succes:
    message = "Gagnant"
else
    message = "Perdant"

print(message)
```

`message` est lié à
la portée la plus interne,
englobant l'instruction if

Passe par valeur ou par référence

```
x = 10
x += 1
x
# => 11
```

Passe par valeur ou par référence

```
x = 10
x += 1
x
# => 11
```

Un nouveau objet est
créé et lié au nom “x”

Passe par valeur ou par référence

```
x = 10
x += 1
x
# => 11
```

Un nouveau objet est créé et lié au nom “x”

```
def foo(x):
    x *= 2

x = 10
foo(x)
x
# => 20
```

Passe par valeur ou par référence

```
x = 10
x += 1
x
# => 11
```

Un nouveau objet est créé et lié au nom “x”

```
def foo(x):
    x *= 2

x = 10
foo(x)
x
# => 20
```

Un nouveau objet est créé et lié, mais “x” seulement sur “foo”, donc update local.

Passe par valeur ou par référence

```
x = 10
x += 1
x
# => 11
```

Un nouveau objet est créé et lié au nom “x”

```
def foo(x):
    x *= 2

x = 10
foo(x)
x
# => 20
```

Un nouveau objet est créé et lié, mais “x” seulement sur “foo”, donc update local.

```
x = [5]
x.append(10)
x
# => [5, 10]
```

Passe par valeur ou par référence

```
x = 10
x += 1
x
# => 11
```

Un nouveau objet est créé et lié au nom “x”

```
x = [5]
x.append(10)
x
# => [5, 10]
```

Aucun nouveau objet est créé. La valeur de “x” est modifié

```
def foo(x):
    x *= 2

x = 10
foo(x)
x
# => 20
```

Un nouveau objet est créé et lié, mais “x” seulement sur “foo”, donc update local.

Passe par valeur ou par référence

```
x = 10
x += 1
x
# => 11
```

Un nouveau objet est créé et lié au nom “x”

```
x = [5]
x.append(10)
x
# => [5, 10]
```

Aucun nouveau objet est créé. La valeur de “x” est modifié

```
def foo(x):
    x *= 2

x = 10
foo(x)
x
# => 20
```

Un nouveau objet est créé et lié, mais “x” seulement sur “foo”, donc update local.

```
def foo(x):
    x.append(20)

x = 5
foo(x)
x
# => [5, 20]
```

Passe par valeur ou par référence

```
x = 10
x += 1
x
# => 11
```

Un nouveau objet est créé et lié au nom “x”

```
x = [5]
x.append(10)
x
# => [5, 10]
```

Aucun nouveau objet est créé. La valeur de “x” est modifié

```
def foo(x):
    x *= 2

x = 10
foo(x)
x
# => 20
```

Un nouveau objet est créé et lié, mais “x” seulement sur “foo”, donc update local.

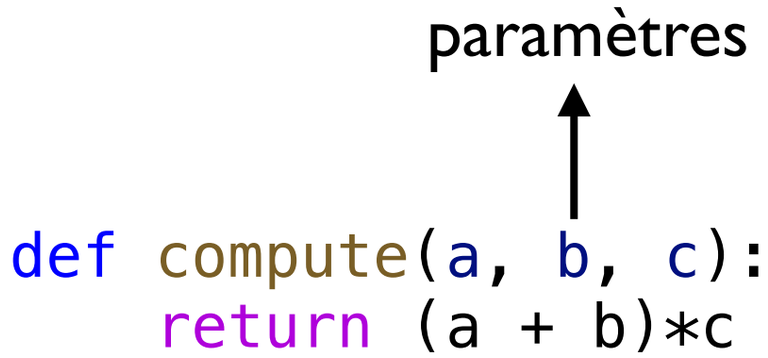
```
def foo(x):
    x.append(20)

x = 5
foo(x)
x
# => [5, 20]
```

“x” n’est pas lié au foo, les motifs sont propagés.

Paramètres

Jusqu'au maintenant …



- Nous avons vu les *paramètres de position*

Paramètres par défaut

- Spécifier un valeur par défaut pour un ou plusieurs paramètres
  - On peut appeler avec moins d'argument si on veut

- Fournir les paramètres pour la fonction et spécifier certaines valeurs par défaut.
  - Présentation d'une interface simple
  - Adaptation aux différents niveaux

Paramètres par défaut

- Spécifier un valeur par défaut pour un ou plusieurs paramètres
  - On peut appeler avec moins d'argument si on veut

- Fournir les paramètres pour la fonction et spécifier certaines valeurs par défaut.
  - Présentation d'une interface simple
  - Adaptation aux différents niveaux

```
def demander(prompt, retries=4, complaint="Entrer Y ou N"):
    ...
```

Paramètres par défaut

```
def demander(prompt, retries=4, complaint="Entrer Y ou N"):
    ...
```

- “prompt” est obligatoire, “retries” est optionnel avec une valeur par défaut 4, “complaint” est pareil.

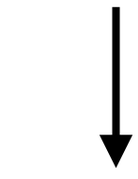
Paramètres par défaut

```
def demander(prompt, retries=4, complaint="Entrer Y ou N"):
    ...
```

- “prompt” est obligatoire, “retries” est optionnel avec une valeur par défaut 4, “complaint” est pareil.

- Un appel valide

```
demander(“Ok pour reécrire le fichier?”, 2)
```



prompt

redéfinir “retries”

Paramètres par défaut

```
def demander(prompt, retries=4, complaint="Entrer Y ou N"):
    if i in range(retries):
        ans = input(prompt)
        if ans in "yY":
            return True
        if ans in "nN":
            return False
        print(complaint)
```

- Dedans la fonction, traiter les paramètres comme toujours.

Arguments de mot-clé

- Les paramètres peuvent-être appelés en paramètre de position ou *paramètre de mot-clé*
  - Spécifier le nom des variables dans l'appel

Arguments de mot-clé

- Les paramètres peuvent-être appelés en paramètre de position ou *paramètre de mot-clé*
  - Spécifier le nom des variables dans l'appel

- Pourquoi?
  - rendre l'intention sur les paramètres plus claire, les noms expriment plus que les positions
  - réduire le risque des appels incorrects

Arguments de mot-clé

```
def demander(prompt, retries=4, complaint="Entrer Y ou N"):
    ...
```

- Les appels valides

```
demander(“Ok pour reécrire le fichier?”, 2)
demander(“Ok pour reécrire le fichier?”, retries = 2)
demander(prompt = “Ok pour reécrire le fichier?”, retries = 2)
demander(retries = 2, prompt = “Ok pour reécrire le fichier?”)
```

Arguments variadiques

Arguments variadiques de position

- Un paramètre de forme `*args` capture les arguments de position excessifs
  - ces arguments sont regroupés dans le tuple `args`
- Pourquoi?
  - appeler une fonction avec un nombre arbitraires des arguments de position
  - capturer les arguments et transmettre à un autre gestionnaire
  - utiliser dans les sous-classes, proxies, etc

Arguments variadiques de position

```
# echelle * (a1 + a2 + ... + an)
def somme_echelle(*args, echelle = 1)
    return echelle * sum(args)

somme_echelle(1, 2, 3)                  # => 6
somme_echelle(1, 5)                     # => 6
somme_echelle(1, 2, echelle=10)         # => 30
```

Arguments variadiques de position

```python
# echelle * (a1 + a2 + ... + an)
def somme_echelle(*args, echelle = 1)
    return echelle * sum(args)

somme_echelle(1, 2, 3)                  # => 6
somme_echelle(1, 5)                     # => 6
somme_echelle(1, 2, echelle=10)         # => 30
```

```python
# une fonction vérifiant si un nombre est premier
def is_prime(p):
    ...

premiers = [p in range(1,100) if is_prime(p)]

print(somme_echelle(*premiers))
```

Arguments variadiques de mot-clé

- Un paramètre de forme `**kwargs` capture les arguments de mot-clé excessifs
  - ces arguments sont regroupés dans le dictionnaire `kwargs`

- Pourquoi?
  - autoriser un nombre arbitraire des arguments de mot-clé, typiquement utiliser dans les configurations.
  - capturer les arguments et transmettre à un autre gestionnaire
  - utiliser dans les sous-classes, proxies, etc

Arguments variadiques de mot-clé

```
def etudiant(filiere, **info):
    print("> {}".format(filiere))
    print('-' * (len(filiere) + 2))

    for k, v in info.items():
        print(“{}: {}".format(k, v))

info = {
    "Nom": "ABC"
    "Date de naissance": 1/1/2000
    "Numéro d'étudiant": 123456
}
```

Arguments variadiques de mot-clé

```python
def etudiant(filiere, **info):
    print("> {}".format(filiere))
    print('-' * (len(filiere) + 2))

    for k, v in info.items():
        print("{}: {}".format(k, v))

info = {
    "Nom": "ABC"
    "Date de naissance": 1/1/2000
    "Numéro d'étudiant": 123456
}


etudiant("L3ASR", **info)

# > L3ASR
# -------
# Nom: ABC
# Date de naissance: 1/1/2000
# Numéro d'étudiant: 123456
```

Exemple: Format

```
str.format(*args, **kwargs)
```

Ordres des paramètres

Règles des paramètres

- Les arguments de mot-clé doivent suivre ceux de position.
- Tous les arguments doivent identifier un paramètre
- Aucun paramètre reçoit plus qu'une valeur.

Appels corrects

```
def foo(voltage, etat='Inactive', action='Fort', type="Bien"):
    . . .

foo(100)

foo(voltage=100)

foo(voltage=100, action="Faible")

foo(action="Faible", voltage=100)

foo("cent", "Active", "Moyenne")

foo("cent", etat="Active")
```

Appels incorrects

```
def foo(voltage, etat='Inactive', action='Fort', type="Bien")

# manque d'argument
foo()

# argument de mot-clé après argument de position
foo(voltage=100, "Active")

# dupliquer la valeur
foo(100, voltage=100)
```

Règles des paramètres

- / : Arguments avant '/' sont de position seulement

- * : Arguments avant '*' sont de position ou de mot-clé. Arguments après '*' sont de mot-clé seulement.

- *a: Capturer les arguments de position.

- **kw: Capturer les arguments de mot-clé.

Règles des paramètres

```
def fn(a, b, /, c, d=1, *, e, f=2, **kwargs)
```

Règles des paramètres

```
def fn(a, b, /, c, d=1, *, e, f=2, **kwargs)
```

position
seulement

Règles des paramètres

```
def fn(a, b, /, c, d=1, *, e, f=2, **kwargs)
```

position seulement

mot-clé seulement

Règles des paramètres

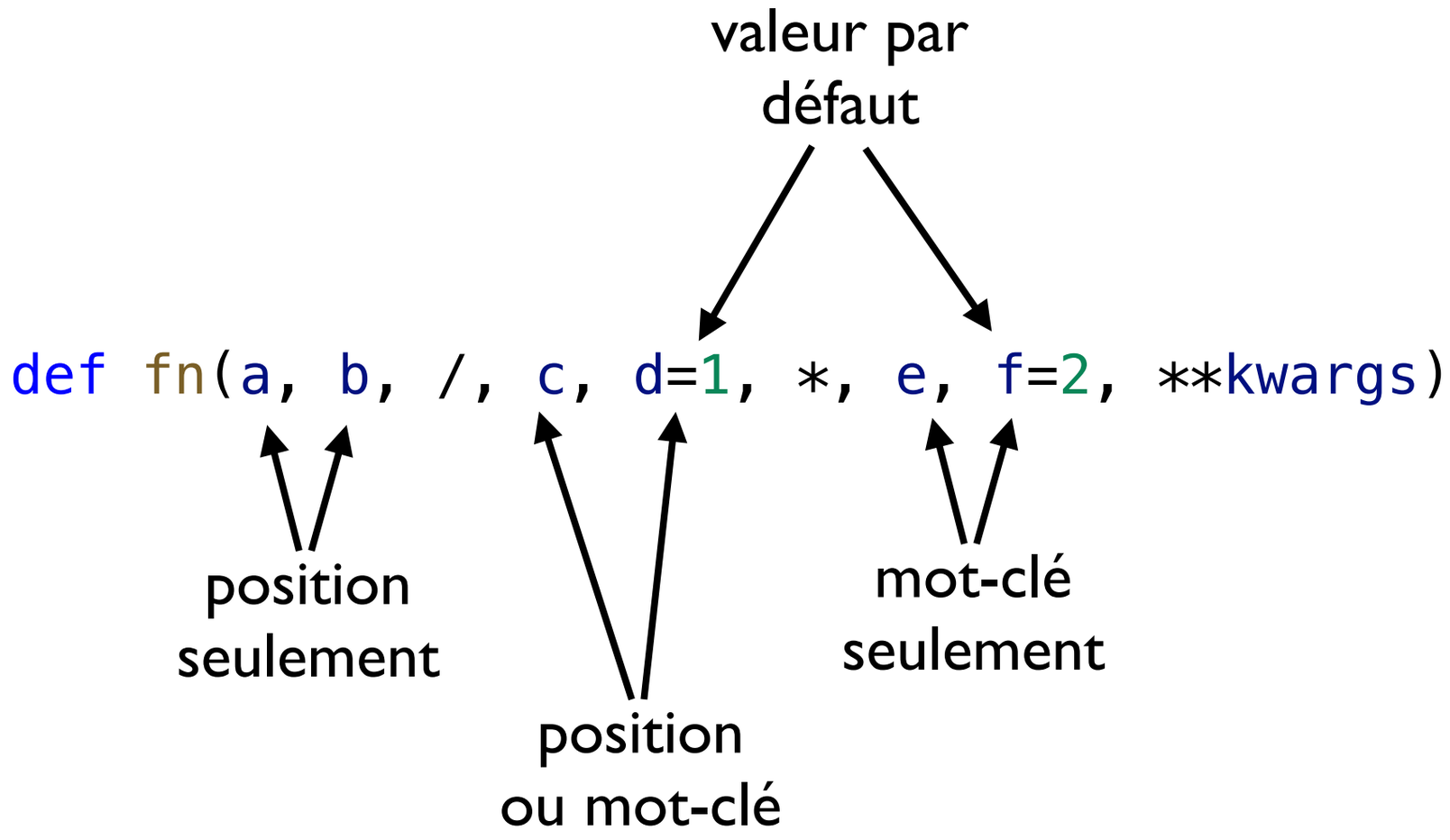
```
def fn(a, b, /, c, d=1, *, e, f=2, **kwargs)
```

position seulement

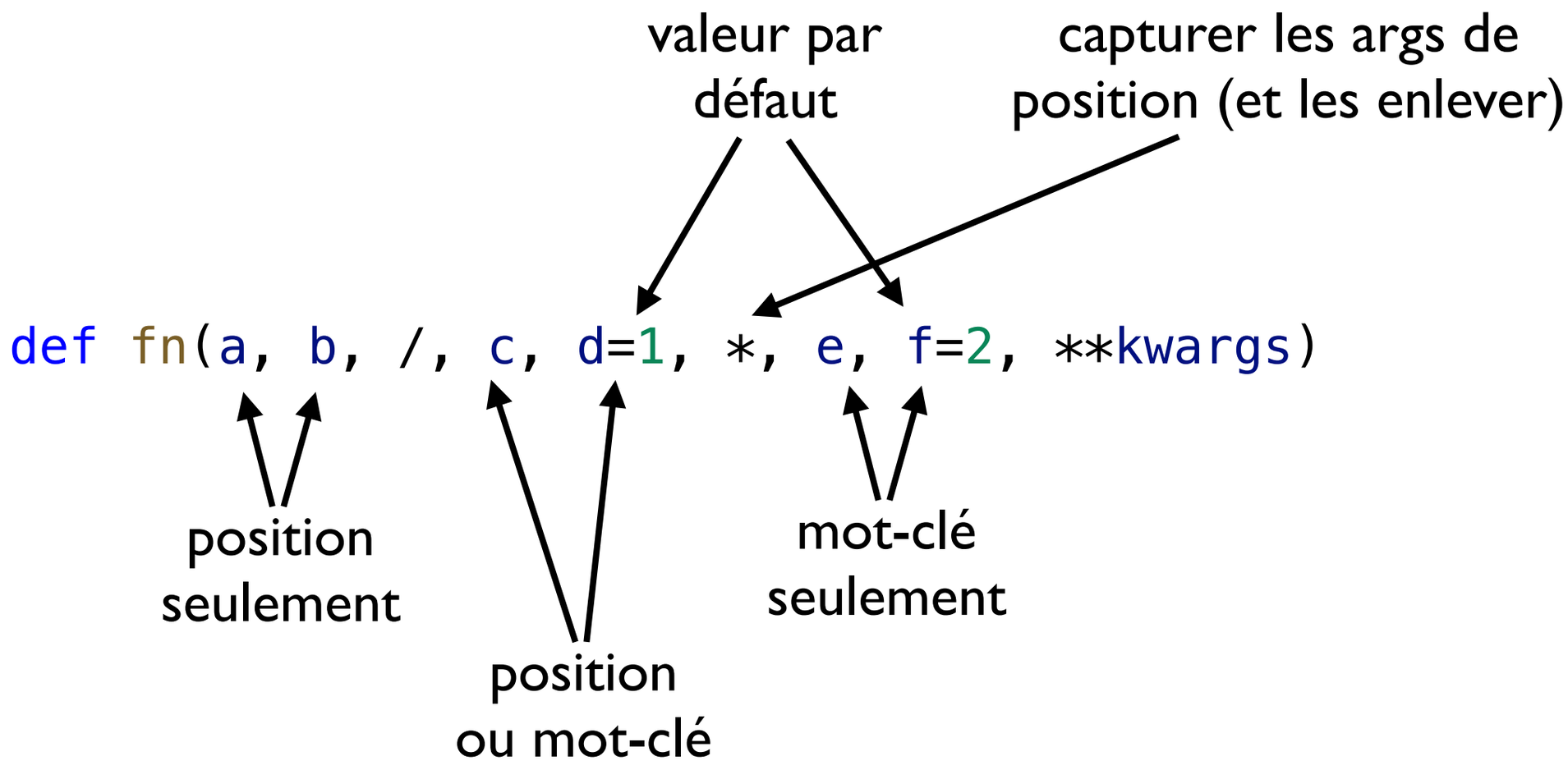
position ou mot-clé

mot-clé seulement

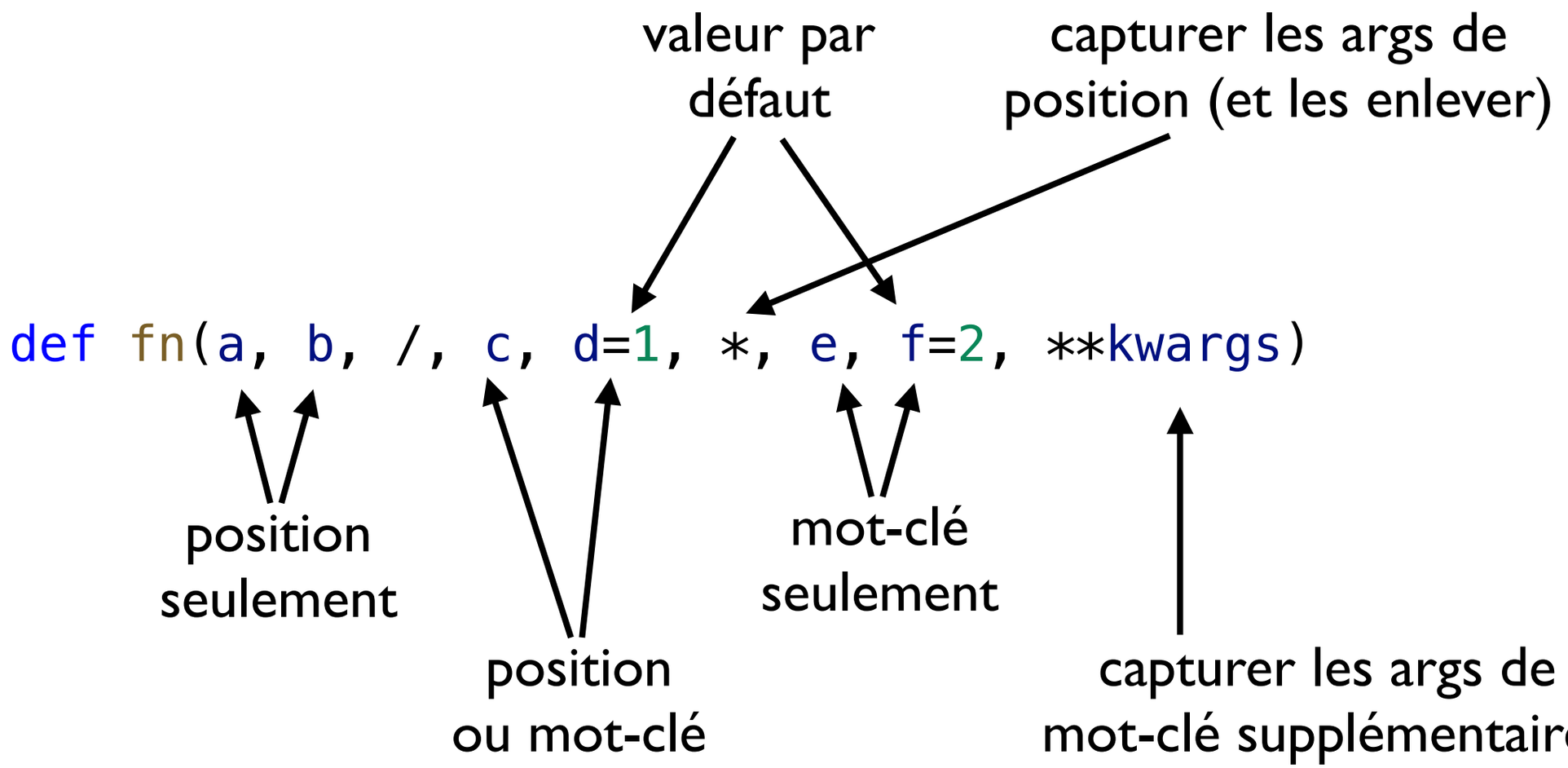
Règles des paramètres



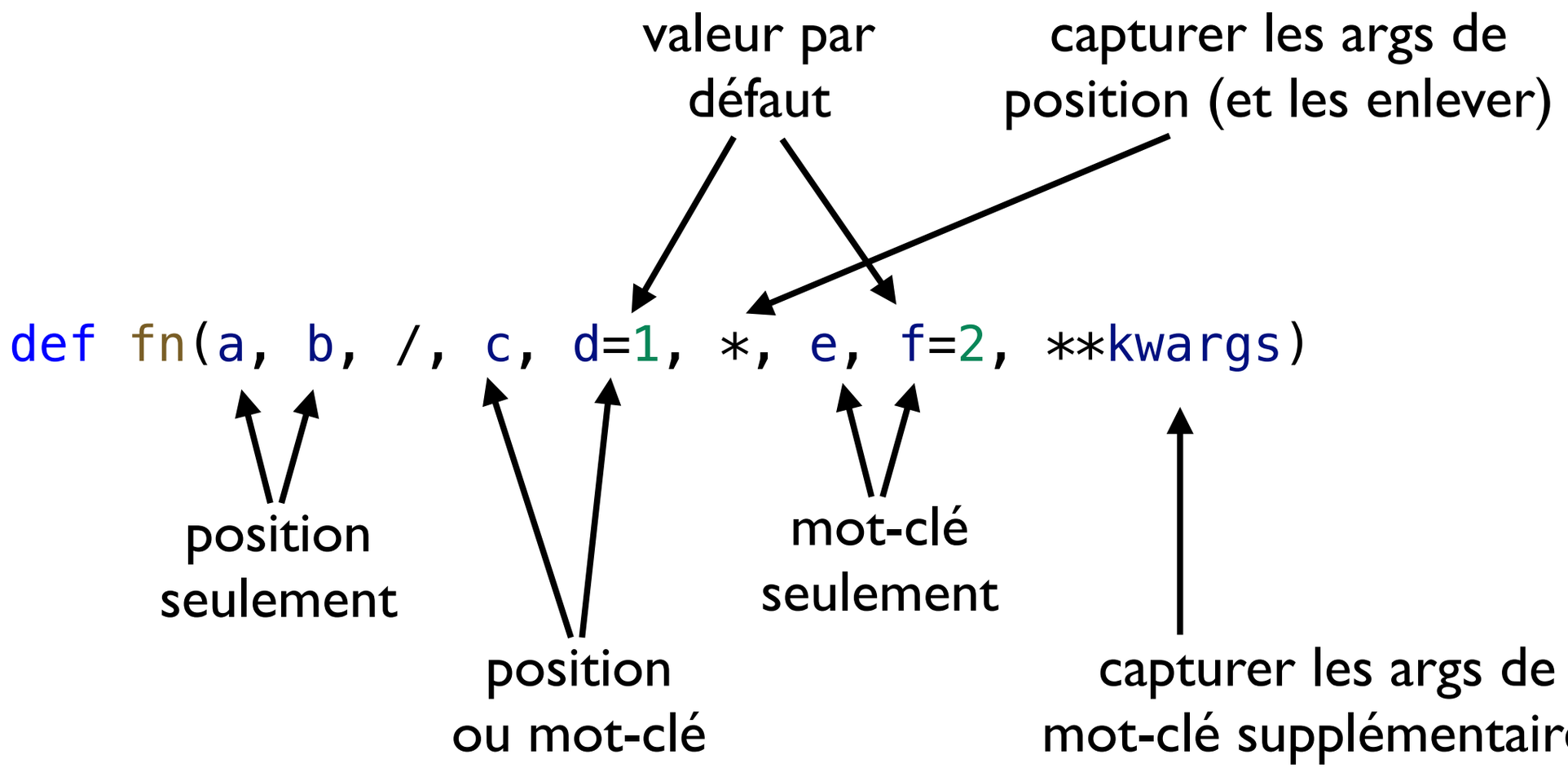
Règles des paramètres



Règles des paramètres



Règles des paramètres



Note: ca ne sera pas valide de l'appeler avec c mot-clé et d position

Style de Python

Commentaires des fonctions

- La première chaîne de caractères *dans* le corps de la fonction est un *doctring*.
  - Première ligne: résumé de la fonction
  - Lignes suivantes: description supplémentaire de la fonction
- Décrire les paramètres (valeurs, types, etc) et la sortie (valeur, type, etc)

Exemple: Docstrings

```
def ma_fonction():
    """ Résumé d'une ligne
    Description: cette fonction prend ... et retourne ...
    L'idée de l'algorithme est ...
    """
    pass
```

Exemple: Docstrings

```
def ma_fonction():
    """ Résumé d'une ligne
    Description: cette fonction prend ... et retourne ...
    L'idée de l'algorithme est ...
    """
    pass
```

```
print(ma_fonction.__doc__)

# Résumé d'une ligne
# Description: cette fonction prend ... et retourne ...
# L'idée de l'algorithme est ...
```

Conseils

- Espaces

```
a = f(1, 2) + g(3, 4)          # bien
a = f( 1, 2 ) + g ( 3, 4)      # pas très bien
```

  - Première ligne: résumé de la fonction
  - Espaces autour des opérateurs, mais pas les délimiter

Conseils

- Espaces

```
a = f(1, 2) + g(3, 4)          # bien
a = f( 1, 2 ) + g ( 3, 4)      # pas très bien
```

  - Première ligne: résumé de la fonction
  - Espaces autour des opérateurs, mais pas les délimiter

- Commentaires
  - Commenter les fonctions non-triviales
  - Commentaires au début du fichier
  - Si c’est possible, commentaires dans la même ligne de l’instruction.

Conseils

- Espaces

```
a = f(1, 2) + g(3, 4)          # bien
a = f( 1, 2 ) + g ( 3, 4)      # pas très bien
```

  - Première ligne: résumé de la fonction
  - Espaces autour des opérateurs, mais pas les délimiter

- Commentaires
  - Commenter les fonctions non-triviales
  - Commentaires au début du fichier
  - Si c'est possible, commentaires dans la même ligne de l'instruction.

- Noms:
  - Utiliser les caractères: `minuscule` pour les variables; `MA_JUSCULE` pour les constantes; `ClasseCase` pour les classes.

Evaluation

1/4 DM1 + 1/4 DM2 + 1/2 Partiel