Algorithmique et Programmation

Fonctions

Elise Bonzon

elise.bonzon@mi.parisdescartes.fr

LIPADE - Université Paris Descartes
http://www.math-info.univ-paris5.fr/∽bonzon/

Fonctions

1. Pourquoi des fonctions ?
2. Procédures
3. Fonctions
4. Exemple : le binôme de Newton
5. Variables locales, variables globales
6. Chaîne de documentation
7. Pour conclure

Pourquoi des fonctions ?

Réutilisabilité des algorithmes

Réutilisabilité des algorithmes

Problème : comment réutiliser un algorithme existant sans avoir à le réécrire ?

Réutilisabilité des algorithmes

Problème : comment réutiliser un algorithme existant sans avoir à le réécrire ?

```
# Factorielle de 5
n = 5
fact = 1
i = 2
while i <= n :
    fact = fact * i
    i = i + 1
print(fact)
```

```
# Factorielle de 8
n = 8
fact = 1
i = 2
while i <= n :
    fact = fact * i
    i = i + 1
print(fact)
```

Réutilisabilité des algorithmes

Problème : comment réutiliser un algorithme existant sans avoir à le réécrire ?

```
# Factorielle de 5
n = 5
fact = 1
i = 2
while i <= n :
    fact = fact * i
    i = i + 1
print(fact)
```

```
# Factorielle de 8
n = 8
fact = 1
i = 2
while i <= n :
    fact = fact * i
    i = i + 1
print(fact)
```

Elément de réponse

Encapsuler le code dans des fonctions ou procédures.

```
>>> factorielle(5)
120
```

```
>>> factorielle(8)
40320
```

Structuration des programmes

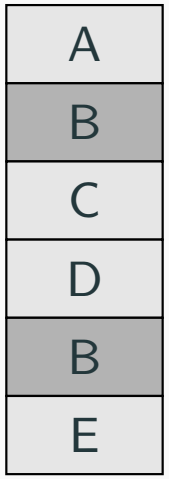
Structuration

Problème : Comment structurer un programme pour le rendre plus compréhensible ?

Structuration des programmes

Structuration

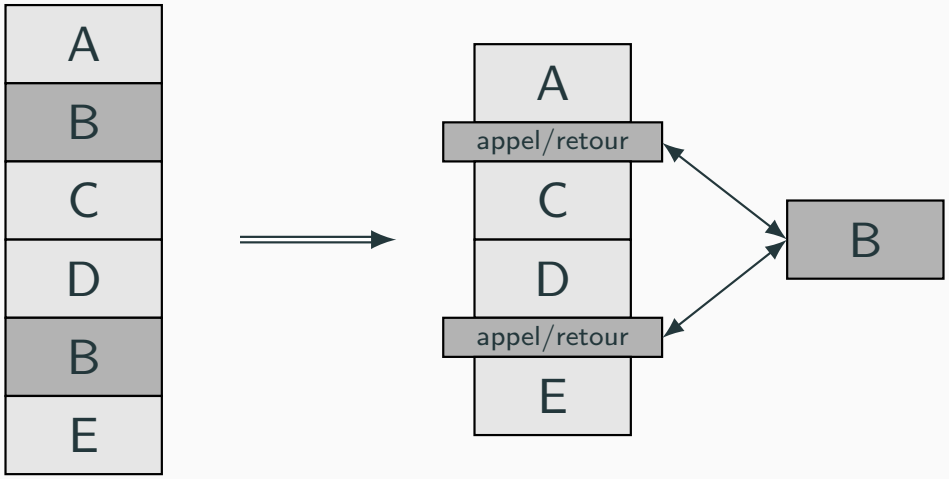
Problème : Comment structurer un programme pour le rendre plus compréhensible ?



Structuration des programmes

Structuration

Problème : Comment structurer un programme pour le rendre plus compréhensible ?



Structuration des programmes

Structuration des programmes

Les fonctions et les procédures permettent de **décomposer** un programme complexe en une série de sous-programmes plus simples, lesquels peuvent à leur tour être décomposés en fragments plus petits, et ainsi de suite...

Fonctions

Fonction

Un fonction est un bloc d'instructions **nommé** et **paramétré**, réalisant une tâche donnée.
Elle admet zéro, un ou plusieurs **paramètres** et **retourne toujours un résultat**.

Fonctions

Fonction

Un fonction est un bloc d'instructions **nommé** et **paramétré**, réalisant une tâche donnée.
Elle admet zéro, un ou plusieurs **paramètres** et **retourne toujours un résultat**.

- Une fonction est donc une suite ordonnée d'instructions qui *retourne une valeur*
- Une fonction joue le rôle d'une *expression*. Elle enrichit le jeu des expressions possibles.
- Par exemple, `len()` est une fonction prédéfinie

Procédures

Procédure

Un procédure est un bloc d'instructions **nommé** et **paramétré**, réalisant une tâche donnée.
Elle admet zéro, un ou plusieurs **paramètres** et **ne retourne pas de résultat**.

Procédures

Procédure

Un procédure est un bloc d'instructions **nommé** et **paramétré**, réalisant une tâche donnée.
Elle admet zéro, un ou plusieurs **paramètres** et **ne retourne pas de résultat**.

- Une procédure est donc une suite ordonnée d'instructions qui *ne retourne pas de valeur*
- Une procédure joue le rôle d'une *instruction*. Elle enrichit le jeu des instructions possibles.
- Par exemple, `print()` est une procédure prédéfinie

Procédures

Procédure

Un procédure est un bloc d'instructions **nommé** et **paramétré**, réalisant une tâche donnée.
Elle admet zéro, un ou plusieurs **paramètres** et **ne retourne pas de résultat**.

- Une procédure est donc une suite ordonnée d'instructions qui *ne retourne pas de valeur*
- Une procédure joue le rôle d'une *instruction*. Elle enrichit le jeu des instructions possibles.
- Par exemple, `print()` est une procédure prédéfinie

Une **fonction** *vaut* quelque chose (son *retour*), une **procédure** *fait* quelque chose.

Syntaxe d'une fonction ou d'une procédure

```
def nom_fonction(liste_de_parametres) :
    """ chaine de documentation """
    bloc_instructions
```

- `nom_fonction` : doit respecter les règles suivantes :

Syntaxe d'une fonction ou d'une procédure

```
def nom_fonction(liste_de_parametres) :
    """ chaine de documentation """
    bloc_instructions
```

- `nom_fonction` : doit respecter les règles suivantes :
  - Aucun caractère spécial (hormis "_"), aucun caractère accentué

Syntaxe d'une fonction ou d'une procédure

```
def nom_fonction(liste_de_parametres) :
    """ chaine de documentation """
    bloc_instructions
```

- `nom_fonction` : doit respecter les règles suivantes :
  - Aucun caractère spécial (hormis "_"), aucun caractère accentué
  - Commence par une minuscule (les majuscules seront utilisées pour les classes)

Syntaxe d'une fonction ou d'une procédure

```
def nom_fonction(liste_de_parametres) :
    """ chaine de documentation """
    bloc_instructions
```

- `nom_fonction` : doit respecter les règles suivantes :
  - Aucun caractère spécial (hormis "_"), aucun caractère accentué
  - Commence par une minuscule (les majuscules seront utilisées pour les classes)
  - Choisir un nom **suffisamment explicite** !

Syntaxe d'une fonction ou d'une procédure

```
def nom_fonction(liste_de_parametres) :
    """ chaine de documentation """
    bloc_instructions
```

- `nom_fonction` : doit respecter les règles suivantes :
  - Aucun caractère spécial (hormis "_"), aucun caractère accentué
  - Commence par une minuscule (les majuscules seront utilisées pour les classes)
  - Choisir un nom **suffisamment explicite** !
- `liste_de_parametres` : paramètres de la fonction, séparés par une virgule. Peut-être vide.

Syntaxe d'une fonction ou d'une procédure

```
def nom_fonction(liste_de_parametres) :
    """ chaine de documentation """
    bloc_instructions
```

- `nom_fonction` : doit respecter les règles suivantes :
  - Aucun caractère spécial (hormis "_"), aucun caractère accentué
  - Commence par une minuscule (les majuscules seront utilisées pour les classes)
  - Choisir un nom **suffisamment explicite** !
- `liste_de_parametres` : paramètres de la fonction, séparés par une virgule. Peut-être vide.
- `chaine de documentation` : facultative mais **fortement** conseillée. Doit contenir :

Syntaxe d'une fonction ou d'une procédure

```
def nom_fonction(liste_de_parametres) :
    """ chaine de documentation """
    bloc_instructions
```

- `nom_fonction` : doit respecter les règles suivantes :
  - Aucun caractère spécial (hormis "_"), aucun caractère accentué
  - Commence par une minuscule (les majuscules seront utilisées pour les classes)
  - Choisir un nom **suffisamment explicite** !
- `liste_de_parametres` : paramètres de la fonction, séparés par une virgule. Peut-être vide.
- `chaine de documentation` : facultative mais **fortement** conseillée. Doit contenir :
  - la **signature** de la fonction

Syntaxe d'une fonction ou d'une procédure

```
def nom_fonction(liste_de_parametres) :
    """ chaine de documentation """
    bloc_instructions
```

- `nom_fonction` : doit respecter les règles suivantes :
  - Aucun caractère spécial (hormis "_"), aucun caractère accentué
  - Commence par une minuscule (les majuscules seront utilisées pour les classes)
  - Choisir un nom **suffisamment explicite** !
- `liste_de_parametres` : paramètres de la fonction, séparés par une virgule. Peut-être vide.
- `chaine de documentation` : facultative mais **fortement** conseillée. Doit contenir :
  - la **signature** de la fonction
  - une liste d'expressions booléennes qui précisent les **conditions d'application** de la fonction si besoin

Syntaxe d'une fonction ou d'une procédure

```
def nom_fonction(liste_de_parametres) :
    """ chaine de documentation """
    bloc_instructions
```

- `nom_fonction` : doit respecter les règles suivantes :
  - Aucun caractère spécial (hormis "_"), aucun caractère accentué
  - Commence par une minuscule (les majuscules seront utilisées pour les classes)
  - Choisir un nom **suffisamment explicite** !
- `liste_de_parametres` : paramètres de la fonction, séparés par une virgule. Peut-être vide.
- `chaine de documentation` : facultative mais **fortement** conseillée. Doit contenir :
  - la **signature** de la fonction
  - une liste d'expressions booléennes qui précisent les **conditions d'application** de la fonction si besoin
  - une phrase qui explique **ce que fait la fonction**

Syntaxe d'une fonction ou d'une procédure

```
def nom_fonction(liste_de_parametres) :
    """ chaine de documentation """
    bloc_instructions
```

- `nom_fonction` : doit respecter les règles suivantes :
  - Aucun caractère spécial (hormis "_"), aucun caractère accentué
  - Commence par une minuscule (les majuscules seront utilisées pour les classes)
  - Choisir un nom **suffisamment explicite** !
- `liste_de_parametres` : paramètres de la fonction, séparés par une virgule. Peut-être vide.
- `chaine de documentation` : facultative mais **fortement** conseillée. Doit contenir :
  - la **signature** de la fonction
  - une liste d'expressions booléennes qui précisent les **conditions d'application** de la fonction si besoin
  - une phrase qui explique **ce que fait la fonction**
- Attention : l'indentation délimite la fonction ; ne pas oublier les :

Procédures

Procédure simple sans paramètre (1)

```
>>> def table7() :
...     """
...     None --> None
...     Procédure affichant les 10 premiers termes
...     de la table de multiplication de 7
...     """
...
...     for i in range(1, 11):
...         print(i * 7, end = ' ')
...     print(" ")
...
```

Procédure simple sans paramètre (1)

```
>>> def table7() :
...     """
...     None --> None
...     Procédure affichant les 10 premiers termes
...     de la table de multiplication de 7
...     """
...
...     for i in range(1, 11):
...         print(i * 7, end = ' ')
...     print(" ")
...
>>> table7()
7 14 21 28 35 42 49 56 63 70
```

Procédure simple sans paramètre (1)

```
>>> def table7() :
...     """
...     None --> None
...     Procédure affichant les 10 premiers termes
...     de la table de multiplication de 7
...     """
...
...     for i in range(1, 11):
...         print(i * 7, end = ' ')
...     print(" ")
...
>>> table7()
7 14 21 28 35 42 49 56 63 70
```

Il est maintenant possible de réutiliser cette procédure autant de fois que souhaité.

Procédure simple sans paramètre (2)

```
>>> def table7triple() :
...     """
...     None --> None
...     Procédure affichant la table de 7 à trois reprises
...     """
...
...     print("La table de 7 en triple exemplaire :")
...     table7()
...     table7()
...     table7()
...
```

Procédure simple sans paramètre (2)

```
>>> def table7triple() :
...     """
...     None --> None
...     Procédure affichant la table de 7 à trois reprises
...     """
...
...     print("La table de 7 en triple exemplaire :")
...     table7()
...     table7()
...     table7()
...
>>> table7triple()
La table de 7 en triple exemplaire :
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
```

Procédure simple sans paramètre (2)

```
>>> def table7triple() :
...     """
...     None --> None
...     Procédure affichant la table de 7 à trois reprises
...     """
...
...     print("La table de 7 en triple exemplaire :")
...     table7()
...     table7()
...     table7()
...
>>> table7triple()
La table de 7 en triple exemplaire :
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
7 14 21 28 35 42 49 56 63 70
```

Mais si l'on veut maintenant afficher la table de 9 ?

Procédure avec un paramètre (1)

- Nous voulons à présent écrire une procédure permettant d'afficher la table de multiplication du nombre de notre choix
- Il faut donc pouvoir indiquer à la procédure quelle table nous souhaitons afficher

$\Rightarrow$ Argument d'appel de la procédure

- Il faut donc prévoir, dans la définition de la procédure, une **variable** particulière permettant de recevoir l'**argument** transmis

$\Rightarrow$ Paramètre de la procédure

Procédure avec un paramètre (2)

```
>>> def table(base) :
...     """
...     Number --> None
...     Procédure affichant la table de multiplication
...     du nombre donné en paramètre
...     """
...
...     for i in range(1, 11):
...         print(i * base, end = ' ')
...     print(" ")
...
```

Procédure avec un paramètre (2)

```
>>> def table(base) :
...     """
...     Number --> None
...     Procédure affichant la table de multiplication
...     du nombre donné en paramètre
...     """
...
...     for i in range(1, 11):
...         print(i * base, end = ' ')
...     print(" ")
...
>>> table(9)
9 18 27 36 45 54 63 72 81 90
```

Procédure avec un paramètre (2)

```
>>> def table(base) :
...     """
...     Number --> None
...     Procédure affichant la table de multiplication
...     du nombre donné en paramètre
...     """
...
...     for i in range(1, 11):
...         print(i * base, end = ' ')
...     print(" ")
...
>>> table(9)
9 18 27 36 45 54 63 72 81 90
>>> table(13)
13 26 39 52 65 78 91 104 117 130
```

Procédure avec un paramètre (3)

- L'argument utilisé dans l'appel d'une fonction peut-être une variable

Procédure avec un paramètre (3)

- L'argument utilisé dans l'appel d'une fonction peut-être une variable

```
>>> n = 1
>>> while n < 10 :
...     table(n)
...     n = n + 1
...
```

Procédure avec un paramètre (3)

- L'argument utilisé dans l'appel d'une fonction peut-être une variable

```
>>> n = 1
>>> while n < 10 :
...     table(n)
...     n = n + 1
...
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
```

Procédure avec un paramètre : à noter !

A noter !

Le nom d'une variable passée en argument d'une procédure n'a rien à voir avec le nom du paramètre correspondant dans la procédure !

- Dans l'exemple précédent, le contenu de la variable `n` est donné en argument de la procédure, et est affecté au paramètre `base`
- Ces deux variables ne désignent pas la même chose !

Procédure avec plusieurs paramètres (1)

- Nous voulons à présent écrire une procédure permettant d'afficher d'autres termes que les dix premiers d'une table de multiplication
- Il faut donc pouvoir indiquer à la procédure quelle table, mais également quels sont les termes que nous souhaitons afficher

$\Rightarrow$ Ajouts de paramètres supplémentaires

Procédure avec plusieurs paramètres

```
>>> def table_multi(base, debut, fin) :
...     """
...     Number x Int x Int --> None
...     Procédure affichant la table de multiplication
...     de base à partir de l'entier debut jusqu'à l'entier fin
...     """
...
...     print("Fragment de la table de multiplication de", base)
...
...     for i in range(debut, fin + 1) :
...         print(i, 'x', base, '=', i * base)
...
```

Procédure avec plusieurs paramètres (2)

```
>>> table_multi(8, 13, 17)
Fragment de la table de multiplication de 8
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
```

Procédure avec plusieurs paramètres (2)

```
>>> table_multi(8, 13, 17)
Fragment de la table de multiplication de 8
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
>>> table_multi(2.4, 13, 17)
Fragment de la table de multiplication de 2.4
13 x 2.4 = 31.2
14 x 2.4 = 33.6
15 x 2.4 = 36.0
16 x 2.4 = 38.4
17 x 2.4 = 40.8
```

Procédure avec plusieurs paramètres (2)

```
>>> table_multi(8, 13, 17)
Fragment de la table de multiplication de 8
13 x 8 = 104
14 x 8 = 112
15 x 8 = 120
16 x 8 = 128
17 x 8 = 136
>>> table_multi(2.4, 13, 17)
Fragment de la table de multiplication de 2.4
13 x 2.4 = 31.2
14 x 2.4 = 33.6
15 x 2.4 = 36.0
16 x 2.4 = 38.4
17 x 2.4 = 40.8
>>> table_multi(2, 13.2, 17)
Fragment de la table de multiplication de 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 10, in table_multi
TypeError: 'float' object cannot be interpreted as an integer
```

Instruction `assert`

Vérification des préconditions

Les **préconditions** que doivent vérifier les paramètres d'entrée d'une fonction ou procédure seront définies grâce à l'instruction `assert`.
A l'exécution du code, cette instruction *lèvera une exception* si la condition testée est fausse.
Les préconditions seront placées juste après l'en-tête de la fonction.

Instruction `assert`

Vérification des préconditions

Les **préconditions** que doivent vérifier les paramètres d'entrée d'une fonction ou procédure seront définies grâce à l'instruction `assert`.
A l'exécution du code, cette instruction *lèvera une exception* si la condition testée est fausse.
Les préconditions seront placées juste après l'en-tête de la fonction.

```
assert expr[, message]
```

Instruction `assert`

Vérification des préconditions

Les **préconditions** que doivent vérifier les paramètres d'entrée d'une fonction ou procédure seront définies grâce à l'instruction `assert`.
A l'exécution du code, cette instruction *lèvera une exception* si la condition testée est fausse.
Les préconditions seront placées juste après l'en-tête de la fonction.

```
assert expr[, message]
```

- `expr` est une **expression booléenne**.
  - Si `expr == True`, on passe à l'instruction suivante
  - Sinon, l'exécution est interrompue, et une exception `AssertionError` est levée
  - Dans ce cas, le `message`, **optionnel**, est affiché

Instruction assert

```
>>> def table_multi(base, debut, fin) :
...     """
...     Number x Int x Int --> None
...     Procédure affichant la table de multiplication
...     de base à partir de l'entier debut jusqu'à l'entier fin
...     """
...     assert type(base) is int or type(base) is float, "base
        doit être un nombre"
...     assert type(debut) is int, "debut doit être un entier"
...     assert type(fin) is int, "fin doit être un entier"
...
...     print("Fragment de la table de multiplication de", base)
...     for i in range(debut, fin + 1) :
...         print(i, 'x', base, '=', i * base)
...
```

Instruction assert

```
>>> def table_multi(base, debut, fin) :
...     """
...     Number x Int x Int --> None
...     Procédure affichant la table de multiplication
...     de base à partir de l'entier debut jusqu'à l'entier fin
...     """
...     assert type(base) is int or type(base) is float, "base
        doit être un nombre"
...     assert type(debut) is int, "debut doit être un entier"
...     assert type(fin) is int, "fin doit être un entier"
...
...     print("Fragment de la table de multiplication de", base)
...     for i in range(debut, fin + 1) :
...         print(i, 'x', base, '=', i * base)
...
>>> table_multi(2, 2.2, 6)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 7, in table_multi
AssertionError: debut doit être un entier
```

Procédure avec plusieurs paramètres (3)

Attention à respecter l'ordre des paramètres !

```
>>> table_multi(2, 4, 8)
Fragment de la table de multiplication de 2
4 x 2 = 8
5 x 2 = 10
6 x 2 = 12
7 x 2 = 14
8 x 2 = 16
```

Procédure avec plusieurs paramètres (3)

Attention à respecter l'ordre des paramètres !

```
>>> table_multi(2, 4, 8)
Fragment de la table de multiplication de 2
4 x 2 = 8
5 x 2 = 10
6 x 2 = 12
7 x 2 = 14
8 x 2 = 16
>>> table_multi(4, 2, 8)
Fragment de la table de multiplication de 4
2 x 4 = 8
3 x 4 = 12
4 x 4 = 16
5 x 4 = 20
6 x 4 = 24
7 x 4 = 28
8 x 4 = 32
```

Fonctions

Procédures vs fonctions

- Les procédures que nous avons vu jusque là *effectuent* une action, mais ne *retournent* rien
- Les fonctions retournent une valeur grâce à l'instruction `return`.

Procédures vs fonctions

- Les procédures que nous avons vu jusque là *effectuent* une action, mais ne *retournent* rien
- Les fonctions retournent une valeur grâce à l'instruction `return`.

```
>>> def cube(nb):
...     """
...     Number --> Number
...     Fonction retournant le cube du nombre donné en entrée
...     """
...
...     return nb * nb * nb
...
```

Procédures vs fonctions

- Les procédures que nous avons vu jusque là *effectuent* une action, mais ne *retournent* rien
- Les fonctions retournent une valeur grâce à l'instruction `return`.

```
>>> def cube(nb):
...     """
...     Number --> Number
...     Fonction retournant le cube du nombre donné en entrée
...     """
...
...     return nb * nb * nb
...
>>> n = 3
>>> cube(n)
27
```

Procédures vs fonctions

- Les procédures que nous avons vu jusque là *effectuent* une action, mais ne *retournent* rien
- Les fonctions retournent une valeur grâce à l'instruction `return`.

```
>>> def cube(nb):
...     """
...     Number --> Number
...     Fonction retournant le cube du nombre donné en entrée
...     """
...
...     return nb * nb * nb
...
>>> n = 3
>>> cube(n)
27
>>> p = cube(5)
>>> p
125
```

Fonction à plusieurs paramètres

Retour aux tables de multiplications

Fonction à plusieurs paramètres

Retour aux tables de multiplications

```
>>> def tables(base, debut, fin) :
...     """Number x Int x Int --> List
...     Fonction retournant dans une liste
...     la table de multiplication de base à
...     partir de l'entier debut jusqu'à l'entier fin"""
```

Fonction à plusieurs paramètres

Retour aux tables de multiplications

```
>>> def tables(base, debut, fin) :
...     """Number x Int x Int --> List
...     Fonction retournant dans une liste
...     la table de multiplication de base à
...     partir de l'entier debut jusqu'à l'entier fin"""
...     assert type(base) is int or type(base) is float, "base
        doit être un nombre"
...     assert type(debut) is int, "debut doit être un entier"
...     assert type(fin) is int, "fin doit être un entier"
```

Fonction à plusieurs paramètres

Retour aux tables de multiplications

```
>>> def tables(base, debut, fin) :
...     """Number x Int x Int --> List
...     Fonction retournant dans une liste
...     la table de multiplication de base à
...     partir de l'entier debut jusqu'à l'entier fin"""
...     assert type(base) is int or type(base) is float, "base
        doit être un nombre"
...     assert type(debut) is int, "debut doit être un entier"
...     assert type(fin) is int, "fin doit être un entier"
...     liste_resultat = []
...     for i in range(debut, fin + 1) :
...         prod = base * i
...         liste_resultat.append(prod)
...     return liste_resultat
...
```

Fonction à plusieurs paramètres

Retour aux tables de multiplications

```
>>> def tables(base, debut, fin) :
...     """Number x Int x Int --> List
...     Fonction retournant dans une liste
...     la table de multiplication de base à
...     partir de l'entier debut jusqu'à l'entier fin"""
...     assert type(base) is int or type(base) is float, "base
        doit être un nombre"
...     assert type(debut) is int, "debut doit être un entier"
...     assert type(fin) is int, "fin doit être un entier"
...     liste_resultat = []
...     for i in range(debut, fin + 1) :
...         prod = base * i
...         liste_resultat.append(prod)
...     return liste_resultat
...
>>> l = tables(12, 4, 8)
>>> l
[48, 60, 72, 84, 96]
```

Tests des fonctions

Jeu de tests

Les fonctions doivent être testées grâce à l'instruction `assert`.

Tests des fonctions

Jeu de tests

Les fonctions doivent être testées grâce à l'instruction `assert`.

```
>>> assert tables(2, 1, 5) == [2, 4, 6, 8, 10]
>>> assert tables(0, 0, 0) == [0]
>>> assert tables(2, 8, 4) == []
```

Tests des fonctions

Jeu de tests

Les fonctions doivent être testées grâce à l'instruction `assert`.

```
>>> assert tables(2, 1, 5) == [2, 4, 6, 8, 10]
>>> assert tables(0, 0, 0) == [0]
>>> assert tables(2, 8, 4) == []
```

- Les jeux de tests servent au programmeur pour valider la définition d'une fonction
- Ils doivent couvrir tous les cas possibles
  - les cas de base
  - les cas extrêmes
- Ils n'affichent rien, sauf si un test ne passe pas

Exemple : le binôme de Newton

Le binôme de Newton

Pour tout nombre complexes a et b, et tout nombre entier $n \neq 0$:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

Avec : $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, et $k! = 1 * 2 * 3 * \ldots * k$

Le binôme de Newton

Pour tout nombre complexes a et b, et tout nombre entier $n \neq 0$:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

Avec : $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, et $k! = 1 * 2 * 3 * \ldots * k$

- $(a+b)^2 = a^2 + 2ab + b^2$
- $(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$
- $(a+b)^6 = a^6 + 6a^5b + 15a^4b^2 + 20a^3b^3 + 15a^2b^4 + 6ab^5 + b^6$

Le binôme de Newton

Pour tout nombre complexes a et b, et tout nombre entier $n \neq 0$:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

Avec : $\binom{n}{k} = \frac{n!}{k!(n-k)!}$, et $k! = 1 * 2 * 3 * \ldots * k$

- $(a+b)^2 = a^2 + 2ab + b^2$
- $(a+b)^3 = a^3 + 3a^2b + 3ab^2 + b^3$
- $(a+b)^6 = a^6 + 6a^5b + 15a^4b^2 + 20a^3b^3 + 15a^2b^4 + 6ab^5 + b^6$

Déjà vu en MC1 !

Le binôme de Newton

On veut écrire un programme qui affiche le binôme de Newton pour un entier saisi par l'utilisateur. Pour cela, il faut :

- Calculer des factorielles –> *définir une fonction*

Le binôme de Newton

On veut écrire un programme qui affiche le binôme de Newton pour un entier saisi par l'utilisateur. Pour cela, il faut :

- Calculer des factorielles –> *définir une fonction*
- Calculer des coefficients binomiaux –> *définir une fonction*

Le binôme de Newton

On veut écrire un programme qui affiche le binôme de Newton pour un entier saisi par l'utilisateur. Pour cela, il faut :

- Calculer des factorielles –> *définir une fonction*
- Calculer des coefficients binomiaux –> *définir une fonction*
- Afficher le binôme de Newton –> *définir une procédure*

Le binôme de Newton

On veut écrire un programme qui affiche le binôme de Newton pour un entier saisi par l'utilisateur. Pour cela, il faut :

- Calculer des factorielles –> *définir une fonction*
- Calculer des coefficients binomiaux –> *définir une fonction*
- Afficher le binôme de Newton –> *définir une procédure*
- Appeler la procédure pour afficher le binôme

Calcul de factorielles

```
def factorielle(n):
```

Calcul de factorielles

```
def factorielle(n):
    """Int --> Int
    Fonction retournant la factorielle de n"""
```

Calcul de factorielles

```
def factorielle(n):
    """Int --> Int
    Fonction retournant la factorielle de n"""

    assert type(n) is int, "Factorielle d'un entier"
```

Calcul de factorielles

```
def factorielle(n):
    """Int --> Int
    Fonction retournant la factorielle de n"""

    assert type(n) is int, "Factorielle d'un entier"

    fact = 1 #initialisation
```

Calcul de factorielles

```
def factorielle(n):
    """Int --> Int
    Fonction retournant la factorielle de n"""

    assert type(n) is int, "Factorielle d'un entier"

    fact = 1 #initialisation
    for i in range(1, n+1) :
        fact = fact * i
```

Calcul de factorielles

```
def factorielle(n):
    """Int --> Int
    Fonction retournant la factorielle de n """

    assert type(n) is int, "Factorielle d'un entier"

    fact = 1 #initialisation
    for i in range(1, n+1) :
        fact = fact * i
    return(fact)
```

Calcul des coefficients binomiaux

```
def coefficient_binomial(n, k):
```

Calcul des coefficients binomiaux

```
def coefficient_binomial(n, k):
    """Int x Int --> Int, avec k <= n
    Fonction retournant le coefficient binomial de k et n.
    Pour k in [0, n], le coefficient binomial est un entier"""
```

Calcul des coefficients binomiaux

```
def coefficient_binomial(n, k):
    """Int x Int --> Int, avec k <= n
    Fonction retournant le coefficient binomial de k et n.
    Pour k in [0, n], le coefficient binomial est un entier"""

    assert type(n) is int, "n est un entier"
    assert type(k) is int, "k est un entier"
    assert k <= n, "k <= n"
```

Calcul des coefficients binomiaux

```
def coefficient_binomial(n, k):
    """Int x Int --> Int, avec k <= n
    Fonction retournant le coefficient binomial de k et n.
    Pour k in [0, n], le coefficient binomial est un entier"""

    assert type(n) is int, "n est un entier"
    assert type(k) is int, "k est un entier"
    assert k <= n, "k <= n"

    coeff = factorielle(n)//(factorielle(k) * factorielle(n-k))
```

Calcul des coefficients binomiaux

```
def coefficient_binomial(n, k):
    """Int x Int --> Int, avec k <= n
    Fonction retournant le coefficient binomial de k et n.
    Pour k in [0, n], le coefficient binomial est un entier"""

    assert type(n) is int, "n est un entier"
    assert type(k) is int, "k est un entier"
    assert k <= n, "k <= n"

    coeff = factorielle(n)//(factorielle(k) * factorielle(n-k))
    return(coeff)
```

Calcul des coefficients binomiaux

```
def coefficient_binomial(n, k):
    """Int x Int --> Int, avec k <= n
    Fonction retournant le coefficient binomial de k et n.
    Pour k in [0, n], le coefficient binomial est un entier"""

    assert type(n) is int, "n est un entier"
    assert type(k) is int, "k est un entier"
    assert k <= n, "k <= n"

    coeff = factorielle(n)//(factorielle(k) * factorielle(n-k))
    return(coeff)
```

On peut maintenant écrire la procédure permettant d'afficher le binôme de Newton

Binôme de Newton

```
def newton(n):
```

Binôme de Newton

```
def newton(n):
    """ Int --> None, n !=0.
    (a+b)^n = Somme(k=0 à n) (n!//k!(n-k)!a^{n-k}b^k). """
```

Binôme de Newton

```
def newton(n):
    """ Int --> None, n !=0.
    (a+b)^n = Somme(k=0 à n) (n!//k!(n-k)!a^{n-k}b^k). """
    assert type(n) is int, "n est un entier"
    assert not(n == 0), "n différent de 0"
```

Binôme de Newton

```
def newton(n):
    """ Int --> None, n !=0.
    (a+b)^n = Somme(k=0 à n) (n!//k!(n-k)!a^{n-k}b^k). """
    assert type(n) is int, "n est un entier"
    assert not(n == 0), "n différent de 0"
    chaine = "(a + b)^" + str(n) + " = " #Init. de la chaine
```

Binôme de Newton

```
def newton(n):
    """ Int --> None, n !=0.
    (a+b)^n = Somme(k=0 à n) (n!//k!(n-k)!a^{n-k}b^k). """
    assert type(n) is int, "n est un entier"
    assert not(n == 0), "n différent de 0"
    chaine = "(a + b)^" + str(n) + " = " #Init. de la chaine
    for k in range(n+1) :
```

Binôme de Newton

```
def newton(n):
    """ Int --> None, n !=0.
    (a+b)^n = Somme(k=0 à n) (n!//k!(n-k)!a^{n-k}b^k). """
    assert type(n) is int, "n est un entier"
    assert not(n == 0), "n différent de 0"
    chaine = "(a + b)^" + str(n) + " = " #Init. de la chaine
    for k in range(n+1) :
        c, p = coefficient_binomial(n, k), n - k
```

Binôme de Newton

```
def newton(n):
    """ Int --> None, n !=0.
    (a+b)^n = Somme(k=0 à n) (n!//k!(n-k)!a^{n-k}b^k). """
    assert type(n) is int, "n est un entier"
    assert not(n == 0), "n différent de 0"
    chaine = "(a + b)^" + str(n) + " = " #Init. de la chaine
    for k in range(n+1) :
        c, p = coefficient_binomial(n, k), n - k
        if not(c == 1) : #on écrit pas le facteur 1
            chaine = chaine + str(c)
```

Binôme de Newton

```
def newton(n):
    """ Int --> None, n !=0.
    (a+b)^n = Somme(k=0 à n) (n!//k!(n-k)!a^{n-k}b^k). """
    assert type(n) is int, "n est un entier"
    assert not(n == 0), "n différent de 0"
    chaine = "(a + b)^" + str(n) + " = " #Init. de la chaine
    for k in range(n+1) :
        c, p = coefficient_binomial(n, k), n - k
        if not(c == 1) : #on écrit pas le facteur 1
            chaine = chaine + str(c)
        if not(p == 0) : #si p = 0, on écrit pas a
            chaine = chaine + "a"
```

Binôme de Newton

```
def newton(n):
    """ Int --> None, n !=0.
    (a+b)^n = Somme(k=0 à n) (n!//k!(n-k)!a^{n-k}b^k). """
    assert type(n) is int, "n est un entier"
    assert not(n == 0), "n différent de 0"
    chaine = "(a + b)^" + str(n) + " = " #Init. de la chaine
    for k in range(n+1) :
        c, p = coefficient_binomial(n, k), n - k
        if not(c == 1) : #on écrit pas le facteur 1
            chaine = chaine + str(c)
        if not(p == 0) : #si p = 0, on écrit pas a
            chaine = chaine + "a"
            if not(p == 1) :
                chaine = chaine + "^" + str(p)
```

Binôme de Newton

```
def newton(n):
    """ Int --> None, n !=0.
    (a+b)^n = Somme(k=0 à n) (n!//k!(n-k)!a^{n-k}b^k). """
    assert type(n) is int, "n est un entier"
    assert not(n == 0), "n différent de 0"
    chaine = "(a + b)^" + str(n) + " = " #Init. de la chaine
    for k in range(n+1) :
        c, p = coefficient_binomial(n, k), n - k
        if not(c == 1) : #on écrit pas le facteur 1
            chaine = chaine + str(c)
        if not(p == 0) : #si p = 0, on écrit pas a
            chaine = chaine + "a"
            if not(p == 1) :
                chaine = chaine + "^" + str(p)
        if not(k == 0):
            chaine = chaine + "b"
```

Binôme de Newton

```
def newton(n):
    """ Int --> None, n !=0.
    (a+b)^n = Somme(k=0 à n) (n!//k!(n-k)!a^{n-k}b^k). """
    assert type(n) is int, "n est un entier"
    assert not(n == 0), "n différent de 0"
    chaine = "(a + b)^" + str(n) + " = " #Init. de la chaine
    for k in range(n+1) :
        c, p = coefficient_binomial(n, k), n - k
        if not(c == 1) : #on écrit pas le facteur 1
            chaine = chaine + str(c)
        if not(p == 0) : #si p = 0, on écrit pas a
            chaine = chaine + "a"
            if not(p == 1) :
                chaine = chaine + "^" + str(p)
        if not(k == 0):
            chaine = chaine + "b"
            if not(k == 1) :
                chaine = chaine + "^" + str(k)
```

Binôme de Newton

```
def newton(n):
    """ Int --> None, n !=0.
    (a+b)^n = Somme(k=0 à n) (n!//k!(n-k)!a^{n-k}b^k). """
    assert type(n) is int, "n est un entier"
    assert not(n == 0), "n différent de 0"
    chaine = "(a + b)^" + str(n) + " = " #Init. de la chaine
    for k in range(n+1) :
        c, p = coefficient_binomial(n, k), n - k
        if not(c == 1) : #on écrit pas le facteur 1
            chaine = chaine + str(c)
        if not(p == 0) : #si p = 0, on écrit pas a
            chaine = chaine + "a"
            if not(p == 1) :
                chaine = chaine + "^" + str(p)
        if not(k == 0):
            chaine = chaine + "b"
            if not(k == 1) :
                chaine = chaine + "^" + str(k)
        if k < n: #On continue avec + si il y a encore un terme
            chaine = chaine + " + "
```

Binôme de Newton

```
def newton(n):
    """ Int --> None, n !=0.
    (a+b)^n = Somme(k=0 à n) (n!//k!(n-k)!a^{n-k}b^k). """
    assert type(n) is int, "n est un entier"
    assert not(n == 0), "n différent de 0"
    chaine = "(a + b)^" + str(n) + " = " #Init. de la chaine
    for k in range(n+1) :
        c, p = coefficient_binomial(n, k), n - k
        if not(c == 1) : #on écrit pas le facteur 1
            chaine = chaine + str(c)
        if not(p == 0) : #si p = 0, on écrit pas a
            chaine = chaine + "a"
            if not(p == 1) :
                chaine = chaine + "^" + str(p)
        if not(k == 0):
            chaine = chaine + "b"
            if not(k == 1) :
                chaine = chaine + "^" + str(k)
        if k < n: #On continue avec + si il y a encore un terme
            chaine = chaine + " + "
print(chaine)
```

Binôme de Newton : script final

```
def factorielle(n):
    #Fonction factorielle telle que définie en slide 26

def coefficient_binomial(n, k):
    #Fonction coefficient_binomoal telle que définie en slide 27

def newton(n) :
    #Procédure Newton telle que définie en salide 28

#Appel principal :
n = int(input("Calculer le binôme de newton pour n = "))
newton(n)
```

Binôme de Newton : script final

```
def factorielle(n):
    #Fonction factorielle telle que définie en slide 26

def coefficient_binomial(n, k):
    #Fonction coefficient_binomoal telle que définie en slide 27

def newton(n) :
    #Procédure Newton telle que définie en salide 28

#Appel principal :
n = int(input("Calculer le binôme de newton pour n = "))
newton(n)
```

```
Calculer le binôme de newton pour n = 2
(a + b)^2 = b^2 + 2ab + a^2
```

Binôme de Newton : script final

```
def factorielle(n):
    #Fonction factorielle telle que définie en slide 26

def coefficient_binomial(n, k):
    #Fonction coefficient_binomoal telle que définie en slide 27

def newton(n) :
    #Procédure Newton telle que définie en salide 28

#Appel principal :
n = int(input("Calculer le binôme de newton pour n = "))
newton(n)
```

```
Calculer le binôme de newton pour n = 2
(a + b)^2 = b^2 + 2ab + a^2
```

```
Calculer le binôme de newton pour n = 5
(a + b)^5 = a^5 + 5a^4b + 10a^3b^2 + 10a^2b^3 + 5ab^4 + b^5
```

Variables locales, variables globales

Variables locales, variable globales

Variables locales, variables globales

Les variables définies **à l'intérieur** du corps d'une fonction ou d'une procédure ne sont accessibles qu'à la fonction elle-même. Ce sont des variables locales à la fonction.

Le contenu des variables locales est **inaccessible depuis l'extérieur de la fonction**.

Variables locales, variable globales

Variables locales, variables globales

Les variables définies **à l'intérieur** du corps d'une fonction ou d'une procédure ne sont accessibles qu'à la fonction elle-même. Ce sont des variables locales à la fonction.
Le contenu des variables locales est **inaccessible depuis l'extérieur de la fonction**.

Les variables définies **à l'extérieur** d'une fonction ou procédure sont des variables globales. Leur contenu est « visible » de l'intérieur d'une fonction, mais la fonction **ne le modifie pas**.

Variables locales, variable globales : exemple

```
>>> def test(): #définition d'une précédure test
...     p = 20 #p est une variable locale
...     print(p, q) #q non définie : variable globale
...
```

Variables locales, variable globales : exemple

```
>>> def test(): #définition d'une précédure test
...     p = 20 #p est une variable locale
...     print(p, q) #q non définie : variable globale
...
>>> p, q = 1, 2 #initialisation de p et q : variables globales
```

Variables locales, variable globales : exemple

```
>>> def test(): #définition d'une précédure test
...     p = 20 #p est une variable locale
...     print(p, q) #q non définie : variable globale
...
>>> p, q = 1, 2 #initialisation de p et q : variables globales
>>> print(p, q)
1 2
```

Variables locales, variable globales : exemple

```
>>> def test(): #définition d'une précédure test
...     p = 20 #p est une variable locale
...     print(p, q) #q non définie : variable globale
...
>>> p, q = 1, 2 #initialisation de p et q : variables globales
>>> print(p, q)
1 2
>>> test() #appel de la procédure test
20 2
```

Variables locales, variable globales : exemple

```
>>> def test(): #définition d'une précédure test
...     p = 20 #p est une variable locale
...     print(p, q) #q non définie : variable globale
...
>>> p, q = 1, 2 #initialisation de p et q : variables globales
>>> print(p, q)
1 2
>>> test() #appel de la procédure test
20 2
>>> print(p, q) #la valeur de p n'a pas changée
1 2
```

Chaîne de documentation

Chaîne de documentation

- La chaîne de documentation placée au début des fonctions et procédures ne joue aucun rôle fonctionnel dans le script
- Elle est traitée comme *un simple commentaire* par Python
- Mais elle est mémorisée à part dans un système de documentation interne automatique

Chaîne de documentation

- La chaîne de documentation placée au début des fonctions et procédures ne joue aucun rôle fonctionnel dans le script
- Elle est traitée comme *un simple commentaire* par Python
- Mais elle est mémorisée à part dans un système de documentation interne automatique

```
>>> def factorielle(n):
...     """Int --> Int
...     Fonction retournant la factorielle de n
...     """
...     assert type(n) is int, "Factorielle d'un entier"
...     fact = 1
...     for i in range(1, n+1) :
...         fact = fact * i
...     return(fact)
...
>>> print(factorielle.__doc__)
Int --> Int
Fonction retournant la factorielle de n
```

Pour conclure

Résumé du cours

Aujourd'hui, on a vu

- Ce qu'est une fonction, une procédure, et la différence entre les deux
- L'utilisation de la fonction `assert` pour vérifier les préconditions et faire des jeux de tests
- Comment utiliser des fonctions et procédures dans un script
- Les variables globales et locales
- L'importance de la documentation