Semestre 1 / Cours 1 - Fondamentaux

ITC MPSI & PCSI – Année 2024-2025

Sommaire

Fonctions

Déclaration et exécution d'une fonction

Une fonction produit un **résultat** à partir de **paramètres**.

Déclaration et exécution d'une fonction

Une fonction produit un **résultat** à partir de **paramètres**.

Déclaration :

```
def nom_de_la_fonction(paramètre_1,...,paramètre_n) :
    instructions
    return(valeur_à_renvoyer)
```

Déclaration et exécution d'une fonction

Une fonction produit un **résultat** à partir de **paramètres**.

Déclaration :

```
def nom_de_la_fonction(paramètre_1,...,paramètre_n) :
    instructions
    return(valeur_à_renvoyer)
```

Exécution :

```
nom_de_la_fonction(valeur_1,...,valeur_n)
```

Premier exemple

Une fonction f ayant deux paramètres x et y qui renvoie $(x + y)^2$.

Premier exemple

Une fonction f ayant deux paramètres x et y qui renvoie $(x+y)^2$.

Déclaration :

```
def f(x,y) :
    return (x+y)**2
```

Premier exemple

Une fonction f ayant deux paramètres x et y qui renvoie $(x+y)^2$.

Déclaration :

```
def f(x,y) :
    return (x+y)**2
```

Exécution :

`f(1,2)` est évalué en la valeur 9.

Deuxième exemple

Une fonction `maximum` ayant deux paramètres *x* et *y* qui renvoie la plus grande de ces deux valeurs.

Deuxième exemple

Une fonction `maximum` ayant deux paramètres *x* et *y* qui renvoie la plus grande de ces deux valeurs.

Déclaration :

```
def maximum(x,y) :
    if x <= y :
        return y
    else :
        return x
```

Deuxième exemple

Une fonction `maximum` ayant deux paramètres *x* et *y* qui renvoie la plus grande de ces deux valeurs.

Déclaration :

```
def maximum(x,y) :
    if x <= y :
        return y
    else :
        return x
```

```
>>> maximum(2,5)
5
>>> maximum(7,3)
7
```

Signature d'une fonction

Précise le *type* de chaque paramètre et celui du résultat.

Signature d'une fonction

Précise le *type* de chaque paramètre et celui du résultat.

Types usuels :

- `int`, `float`

Signature d'une fonction

Précise le *type* de chaque paramètre et celui du résultat.

Types usuels :

- `int`, `float`
- `bool`

Signature d'une fonction

Précise le *type* de chaque paramètre et celui du résultat.

Types usuels :

- `int`, `float`
- `bool`
- `str`, `list`, `tuple`, `dict`

Signature d'une fonction

Précise le *type* de chaque paramètre et celui du résultat.

Types usuels :

- `int`, `float`
- `bool`
- `str`, `list`, `tuple`, `dict`
- `None`

Exemple

```
def maximum(x : int, y : int) -> int :
    if x <= y :
        return y
    else :
        return x
```

Exemple

```
def maximum(x : int, y : int) -> int :
    if x <= y :
        return y
    else :
        return x
```

Le type des paramètres **n'est pas contrôlé** par l'ordinateur à l'appel de la fonction.

Commentaires

Pour préciser l'effet de la fonction par un texte encadré par « """ ».

Commentaires

Pour préciser l'effet de la fonction par un texte encadré par « """ ».

Aide en ligne :

Ce texte est affiché à l'appel de `help(nom_de_la_fonction)`.

Exemple

```
def maximum(x : int, y : int) -> int :
    """
    Renvoie la plus grande valeur
    entre celle de x et y.
    """
    if x <= y :
        return y
    else :
        return x
```

Exemple

```
def maximum(x : int, y : int) -> int :
    """
    Renvoie la plus grande valeur
    entre celle de x et y.
    """
    if x <= y :
        return y
    else :
        return x
```

```
>>> help(maximum)
Help on function maximum:

maximum(x: int, y: int) -> int
    Renvoie la plus grande valeur
    entre celle de x et y.
```

D'autres commentaires

Pour préciser l'effet d'une instruction, on utilise « # » en fin de ligne.

D'autres commentaires

Pour préciser l'effet d'une instruction, on utilise « # » en fin de ligne.

Exemple :

```
def maximum(x : int, y : int) -> int :
    """
    Renvoie la plus grande valeur
    entre celle de x et y.
    """
    if x <= y :     # si y est le plus grand
        return y
    else :          # si x est le plus grand
        return x
```

Variable locale

Stocke un résultat intermédiaire utile au calcul.

Variable locale

Stocke un résultat intermédiaire utile au calcul.

Exemple :

```
from math import sqrt, cos

def g(x : float) -> float :
    """
    Renvoie la valeur sqrt(x) * cos(sqrt(x))
    """
    rac = sqrt(x)
    return rac*cos(rac)
```

Fonctionnement des variables

Premier exemple :

```
1 x = 1
2 x = 5
3 y = 2*x
4 x = x+1
```

Fonctionnement des variables

Premier exemple :

```
1 x = 1
2 x = 5
3 y = 2*x
4 x = x+1
```

Effet :

ligne	x	y
#1	1	
#2	5	
#3	5	10
#4	6	10

Fonctionnement des variables

Bien distinguer « = » (affectation) et « == » (test d'égalité).

Fonctionnement des variables

Bien distinguer « = » (affectation) et « == » (test d'égalité).

Exemple :

```
1 x = 1
2 y = x == 0
```

Fonctionnement des variables

Bien distinguer « = » (affectation) et « == » (test d'égalité).

Exemple :

```
1 x = 1
2 y = x == 0
```

Effet :

ligne	x	y
#1	1	
#2	1	`False`

Fonctionnement des variables

Possibilité d'affectations simultanées pour plusieurs variables.

Fonctionnement des variables

Possibilité d'affectations simultanées pour plusieurs variables.

Exemple :

```
1 x,y = 1,2
2 x,y = 2*x,x+y
3 x,y = y,x
```

Fonctionnement des variables

Possibilité d'affectations simultanées pour plusieurs variables.

Exemple :

```
1 x,y = 1,2
2 x,y = 2*x,x+y
3 x,y = y,x
```

Effet :

ligne	x	y
#1	1	2
#2	2	3
#3	3	2

Fonctions imbriquées

Une fonction peut utiliser une autre fonction.

Fonctions imbriquées

Une fonction peut utiliser une autre fonction.

Exemple :

```
def f(x : float)-> \
↪ float :
    """
    Renvoie x**2.
    """
    return x**2
```

```
def g(x : float)-> float :
    """
    Renvoie (x+1)**2 * x.
    """
    return f(x+1)*x
```

```
>>> g(2)
18
```

car ...

Fonctions imbriquées

… des variables **de même nom** dans des fonctions différentes sont considérées comme des **variables différentes**.

Fonctions imbriquées

… des variables **de même nom** dans des fonctions différentes sont considérées comme des **variables différentes**.

Le programme précédent est similaire à :

```
def f(x1 : float)-> float :
    """
    Renvoie x1**2.
    """
    return x1**2
```

Fonctions imbriquées

… des variables **de même nom** dans des fonctions différentes sont considérées comme des **variables différentes**.

Ou encore :

```
def g(x2 : float)-> float :
    """
    Renvoie (x2+1)**2 * x2.
    """
    return f(x2+1)*x2
```

Boucle for

Boucle for

Répète des instructions en faisant prendre à une variable une **liste de valeurs** successives.

Boucle for

Répète des instructions en faisant prendre à une variable une **liste de valeurs** successives.

Syntaxe :

```
for nom_de_variable in liste_à_décrire :
    instructions
```

Boucle for

Certaines listes de valeurs entières peuvent être décrites avec l'instruction `range`.

Boucle for

Certaines listes de valeurs entières peuvent être décrites avec l'instruction `range`.

Exemples :

- `range(3,10)` décrit 3, 4, 5, 6, 7, 8, 9.
- `range(10)` décrit 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- `range(1,10,3)` décrit 1, 4, 7.
  `range(10,0,-1)` décrit 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.

Boucle for

Certaines listes de valeurs entières peuvent être décrites avec l'instruction `range`.

Exemples :

- `range(3,10)` décrit 3, 4, 5, 6, 7, 8, 9.
- `range(10)` décrit 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
- `range(1,10,3)` décrit 1, 4, 7.
  `range(10,0,-1)` décrit 10, 9, 8, 7, 6, 5, 4, 3, 2, 1.

On peut obtenir une liste **vide** (par exemple `range(0)`).

Exemple

```
>>> for k in range(1,10) :
...     print(2**k)
...
2
4
8
16
32
64
128
256
512
```

affiche les puissances de deux de 2^1 à 2^9.

Autre exemple

```
>>> for k in range(5) :
...     print("Hello world")
...
Hello world
Hello world
Hello world
Hello world
Hello world
```

Autre exemple

Ici la variable k n'intervient pas dans les instructions répétées.

Autre exemple

Ici la variable *k* n'intervient pas dans les instructions répétées.

On peut aussi écrire :

```
for _ in range(5) :
    print("Hello world")
```

Dans une fonction

Écrivons une fonction renvoyant $n \times x$, pour n entier naturel, en utilisant

$$n \times x = \underbrace{x + x + \cdots + x}_{n \text{ fois}}.$$

Dans une fonction

Écrivons une fonction renvoyant $n \times x$, pour n entier naturel, en utilisant

$$n \times x = \underbrace{x + x + \cdots + x}_{n \text{ fois}}.$$

```
def produit(n : int, x : float) -> float :
    """
    Renvoie n*x pour n entier naturel.
    """
    R = 0
    for k in range(n) :
        R = R+x
    return R
```

Dans une fonction

Déroulement de l'appel `produit(4,1.2)` en affichant le contenu des variables après chaque passage dans la boucle.

Dans une fonction

Déroulement de l'appel `produit(4,1.2)` en affichant le contenu des variables après chaque passage dans la boucle.

i	k_i	R_i
0		0
1	0	1.2
2	1	2.4
3	2	3.6
4	3	4.8

Exercice 1

Écrire une fonction `puissance` calculant sur le même modèle x^n.

Détailler dans un tableau l'appel `puissance(4,2)` (ici $n = 2, x = 4$)

Exercice 1 : solutions

```
def puissance(n : int, x : float) -> float :
    """
    Renvoie x**n pour n entier naturel.
    """
    R = 1
    for k in range(n) :
        R = R*x
    return R
```

Exercice 1 : solutions

```python
def puissance(n : int, x : float) -> float :
    """
    Renvoie x**n pour n entier naturel.
    """
    R = 1
    for k in range(n) :
        R = R*x
    return R
```

i	k_i	R_i
0		1
1	0	2
2	1	4
3	2	8
4	3	16

Exercice 2

Écrire une fonction

```
somme(n : int, p : float) -> float
```

calculant la somme

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \cdots + n^p.$$

Dérouler dans un tableau l'appel `somme(4,2)`.

Exercice 2 : solutions

```python
def somme(n : int, p : float) -> float :
    """
    Renvoie 1**p + 2**p + ... + n**p pour n>0
    et 0 sinon.
    """
    S = 0
    for k in range(1,n+1) :
        S = S + k**p
    return S
```

Exercice 2 : solutions

```python
def somme(n : int, p : float) -> float :
    """
    Renvoie 1**p + 2**p + ... + n**p pour n>0
    et 0 sinon.
    """
    S = 0
    for k in range(1,n+1) :
        S = S + k**p
    return S
```

i	k_i	S_i
0		0
1	1	1
2	2	5
3	3	14
4	4	30

Exercice 3

Que renvoie cette fonction ? On commencera par dérouler dans un tableau l'appel `mystere(4, 2)`.

```
def mystere(n : int, x : float) -> float :
    """
    Mystère
    """
    S = 0
    R = 1
    for k in range(n):
        S += R
        R = R*x
    return S
```

Exercice 3

i	k_i	S_i	R_i
0		0	1
1	1	1	2
2	2	3	4
3	3	7	8
4	4	15	16

Exercice 3

```python
def mystere(n : int, x : float) -> float :
    """
    Mystère
    """
    S = 0
    R = 1
    for k in range(n):
        S += R
        R = R*x
    return S
```

La fonction retourne la somme géométrique $\sum_{k=0}^{n-1} x^k$.

Tests conditionnels

if-else : SYNTAXE GÉNÉRALE

```
if condition :
    bloc de commandes 1
else :
    bloc de commandes 2
```

`if-else` : SYNTAXE GÉNÉRALE

```
if condition :
    bloc de commandes 1
else :
    bloc de commandes 2
```

- `condition` est un booléen (`True` ou `False`), souvent issu d'un test,

`if-else` : SYNTAXE GÉNÉRALE

```
if condition :
    bloc de commandes 1
else :
    bloc de commandes 2
```

- `condition` est un booléen (`True` ou `False`), souvent issu d'un test,
- `bloc de commandes 1` : ensemble de lignes de code exécutées si et seulement si `condition` vaut `True`,

`if-else` : SYNTAXE GÉNÉRALE

```
if condition :
    bloc de commandes 1
else :
    bloc de commandes 2
```

- `condition` est un booléen (`True` ou `False`), souvent issu d'un test,
- `bloc de commandes 1` : ensemble de lignes de code exécutées si et seulement si `condition` vaut `True`,
- `bloc de commandes 2` : ensemble de lignes de code exécutées si et seulement si `condition` vaut `False`

Exemple 1 : fonction positif

```
def positif(a):
    if a >= 0:
        return True
    else:
        return False
```

Exemple 1 : fonction `positif`

```
def positif(a):
    if a >= 0:
        return True
    else:
        return False
```

- Si `a` est positif ou nul, `positif(a)` renvoie **`True`**,
- sinon, `positif(a)` renvoie **`False`**.

Exemple 2 : fonction nb_div

```
def nb_div(n):
    N = 0
    for k in range(2,n):
        if n%k == 0: # si k divise n
            N = N+1
    return N
```

Exemple 2 : fonction nb_div

```
def nb_div(n):
    N = 0
    for k in range(2,n):
        if n%k == 0: # si k divise n
            N = N+1
    return N
```

`nb_div(n)` renvoie le nombre de diviseurs de l'entier `n` passé en entrée (autres que 1 et `n`).

Exercice

On donne :

```
def mult_3(n):
    N = 0
    S = 0
    for k in range(1,n):
        if k%3 == 0: # si 3 divise k
            N = N+1
            S = S+k
    return N,S
```

- Que renvoie `mult_3(4)`, `mult_3(10)` ?

Exercice

On donne :

```
def mult_3(n):
    N = 0
    S = 0
    for k in range(1,n):
        if k%3 == 0: # si 3 divise k
            N = N+1
            S = S+k
    return N,S
```

- Que renvoie `mult_3(4)`, `mult_3(10)` ?
- Réponse : `mult_3(4)` renvoie `(1,3)` et `mult_3(10)` renvoie `(3,18)`.

Exercice

On donne :

```
def mult_3(n):
    N = 0
    S = 0
    for k in range(1,n):
        if k%3 == 0: # si 3 divise k
            N = N+1
            S = S+k
    return N,S
```

- Que renvoie `mult_3(n)` (pour `n` quelconque) ?

Exercice

On donne :

```
def mult_3(n):
    N = 0
    S = 0
    for k in range(1,n):
        if k%3 == 0: # si 3 divise k
            N = N+1
            S = S+k
    return N,S
```

- Que renvoie `mult_3(n)` (pour `n` quelconque) ?
- Réponse : `mult_3(n)` renvoie le tuple `N,S` où `N` est le nombre d'entiers k multiples de 3 vérifiant $1 \leq k < n$ et `S` est la somme de ces mêmes entiers.

Précisions syntaxiques

- Nécessité du symbole « : » après la condition et après **`else`**.

Précisions syntaxiques

- Nécessité du symbole « : » après la condition et après **`else`**.
- Indentation obligatoire de `bloc de commande`.

Précisions syntaxiques

- Nécessité du symbole « : » après la condition et après **`else`**.
- Indentation obligatoire de `bloc de commande`.
- Instruction **`else`** optionnelle.

Instruction if-elif-...-else

Un exemple de tests imbriqués : Fonction déterminant le nombre de chiffres d'un entier n, avec $0 \leq n < 10000$.

```
def nb_chiffres(n):
    if n < 10:
        Nb = 1
    else:
        if n < 100:
            Nb = 2
        else :
            if n < 1000:
                Nb = 3
            else:
                Nb = 4
    return Nb
```

Instruction if-elif-...-else

Un exemple de tests imbriqués : Fonction déterminant le nombre de chiffres d'un entier n, avec $0 \leq n < 10000$.

```
def nb_chiffres(n):
    if n < 10:
        Nb = 1
    else:
        if n < 100:
            Nb = 2
        else :
            if n < 1000:
                Nb = 3
            else:
                Nb = 4
    return Nb
```

Les tests imbriqués sont peu lisibles.

Instruction if-elif-...-else

Syntaxe générale

```
if condition 1 :
    bloc de commandes 1
elif condition 2 :
    bloc de commandes 2
elif condition 3 :
    bloc de commandes 3
...
elif condition n :
    bloc de commandes n
else :
    bloc de commandes n+1
```

Instruction `if-elif-...-else`

Syntaxe générale

```
if condition 1 :
    bloc de commandes 1
elif condition 2 :
    bloc de commandes 2
elif condition 3 :
    bloc de commandes 3
...
elif condition n :
    bloc de commandes n
else :
    bloc de commandes n+1
```

- Exécution du `bloc de commande` associé à la première `condition` de valeur `True` (à l'exclusion de toutes les autres),
- exécution du `bloc de commande` associé à `else` sinon (si l'instruction `else` est présente).

Réécriture de la fonction précédente

```
def nb_chiffres_bis(n):
    if n < 10:
        Nb = 1
    elif n < 100:
        Nb = 2
    elif n < 1000:
        Nb = 3
    else:
        Nb = 4
    return Nb
```

Exercice

On donne les deux fonctions suivantes :

- fonction div

```
def div(a):
    d1,d2 = None,None
    if a%2 == 0:
        d1 = 2
    if a%3 == 0:
        d2 = 3
    return d1,d2
```

- fonction div_bis

```
def div_bis(a):
    d1,d2 = None,None
    if a%2 == 0:
        d1 = 2
    elif a%3 == 0:
        d2 = 3
    return d1,d2
```

Exercice

On donne les deux fonctions suivantes :

fonction div

```
def div(a):
    d1,d2 = None,None
    if a%2 == 0:
        d1 = 2
    if a%3 == 0:
        d2 = 3
    return d1,d2
```

fonction div_bis

```
def div_bis(a):
    d1,d2 = None,None
    if a%2 == 0:
        d1 = 2
    elif a%3 == 0:
        d2 = 3
    return d1,d2
```

- Que renvoie `div(6)` ?

Exercice

On donne les deux fonctions suivantes :

fonction div

```
def div(a):
    d1,d2 = None,None
    if a%2 == 0:
        d1 = 2
    if a%3 == 0:
        d2 = 3
    return d1,d2
```

fonction div_bis

```
def div_bis(a):
    d1,d2 = None,None
    if a%2 == 0:
        d1 = 2
    elif a%3 == 0:
        d2 = 3
    return d1,d2
```

- Que renvoie `div(6)` ? Réponse : `(2,3)`
- Que renvoie `div_bis(6)` ?

Exercice

On donne les deux fonctions suivantes :

fonction div

```
def div(a):
    d1,d2 = None,None
    if a%2 == 0:
        d1 = 2
    if a%3 == 0:
        d2 = 3
    return d1,d2
```

fonction div_bis

```
def div_bis(a):
    d1,d2 = None,None
    if a%2 == 0:
        d1 = 2
    elif a%3 == 0:
        d2 = 3
    return d1,d2
```

- Que renvoie `div(6)` ? Réponse : `(2,3)`
- Que renvoie `div_bis(6)` ? Réponse :`(2,None)`

Tests élémentaires

Les conditions qui suivent les **`if`** et/ou **`elif`** sont souvent réalisées à l'aide des tests élémentaires suivants :

math	python
$<$	<
$\leq$	<=
$>$	>
$\geq$	>=
$=$	==
$\neq$	!=

Tests multiples

On peut aussi combiner plusieurs tests élémentaires à l'aide d'opérateurs logiques :

math	python
et	`and`
ou	`or`
non	`not`
ou exclusif	`^`

Exemple *n* positif et divisible par 3 $\Rightarrow$ `n >= 0 and n%3 == 0`

Tests multiples

On peut aussi combiner plusieurs tests élémentaires à l'aide d'opérateurs logiques :

math	python
et	`and`
ou	`or`
non	`not`
ou exclusif	`^`

Exemple *n* positif et divisible par 3 $\Rightarrow$ `n >= 0 and n%3 == 0`

Evaluation paresseuse Test `a and b` : si `a` est `False`, alors `b` n'est pas évalué.

Généralisation

Seule contrainte sur `condition` : a pour valeur `True` ou `False`

Généralisation

Seule contrainte sur `condition` : a pour valeur `True` ou `False`

On peut l'obtenir par :

- un résultat de test,
- une variable booléenne,
- une fonction renvoyant un booléen,
- etc ...

Généralisation

Seule contrainte sur `condition` : a pour valeur `True` ou `False`

On peut l'obtenir par :

- un résultat de test,
- une variable booléenne,
- une fonction renvoyant un booléen,
- etc ...

Exemple On se donne une fonction `premier(n)` qui renvoie `True` si `n` est un nombre entier premier, et `False` sinon.

Généralisation

Expliquer le comportement de la fonction suivante :

```
def somme_prem(n):
    S = 0
    for k in range(n):
        if premier(k):
            S = S+k
    return S
```

Généralisation

Expliquer le comportement de la fonction suivante :

```
def somme_prem(n):
    S = 0
    for k in range(n):
        if premier(k):
            S = S+k
    return S
```

Renvoie la somme des nombres premiers strictement inférieurs à n.

Boucle while

Syntaxe générale

```
while condition :
    bloc de commandes
```

où

- `condition` est un booléen (ayant pour valeur `True` ou `False`), souvent issu d'un test,
- `bloc de commandes` est un ensemble de lignes de code.

Syntaxe générale

```
while condition :
    bloc de commandes
```

où

- `condition` est un booléen (ayant pour valeur `True` ou `False`), souvent issu d'un test,
- `bloc de commandes` est un ensemble de lignes de code.

Lors de l'exécution :

- `condition` vaut `False` $\Rightarrow$ `bloc de commandes` non exécuté, puis on passe à la suite du programme (on quitte la boucle),

Syntaxe générale

```
while condition :
    bloc de commandes
```

où

- `condition` est un booléen (ayant pour valeur `True` ou `False`), souvent issu d'un test,
- `bloc de commandes` est un ensemble de lignes de code.

Lors de l'exécution :

- `condition` vaut `False` $\Rightarrow$ `bloc de commandes` non exécuté, puis on passe à la suite du programme (on quitte la boucle),
- `condition` vaut `True` $\Rightarrow$ `bloc de commandes` exécuté, puis **on évalue à nouveau `condition`**, et ainsi de suite.

Remarques

```
while condition :
    bloc de commandes
```

- `bloc de commandes` généralement exécuté plusieurs fois (différence avec le **`if`** sans partie **`else`**),

Remarques

```
while condition :
    bloc de commandes
```

- `bloc de commandes` généralement exécuté plusieurs fois (différence avec le **`if`** sans partie **`else`**),
- pour éviter la boucle infinie, il faut que `bloc de commandes` influe sur `condition` (afin de rendre la condition fausse au bout d'un moment),

Remarques

```
while condition :
    bloc de commandes
```

- `bloc de commandes` généralement exécuté plusieurs fois (différence avec le **`if`** sans partie **`else`**),
- pour éviter la boucle infinie, il faut que `bloc de commandes` influe sur `condition` (afin de rendre la condition fausse au bout d'un moment),
- la boucle **`while`** possède d'autres possibilités que celles d'une boucle **`for`**.

Premier exemple : calcul de somme

Deux façons différentes de calculer la somme des entiers inférieurs à `n`

```
def somme(n):
    S = 0
    for k in range(n):
        S = S+k
    return S
```

```
def somme(n):
    S = 0
    k = 0
    while k < n:
        S = S+k
        k = k+1
    return S
```

Premier exemple : calcul de somme

Deux façons différentes de calculer la somme des entiers inférieurs à n

```
def somme(n):
    S = 0
    for k in range(n):
        S = S+k
    return S
```

```
def somme(n):
    S = 0
    k = 0
    while k < n:
        S = S+k
        k = k+1
    return S
```

Pour la boucle **while** :

- il est nécessaire d'initialiser la variable k (instruction k = 0),
- il est nécessaire d'incrémenter la variable k (instruction k = k+1). En l'absence de cette dernière instruction, on aurait une boucle infinie.

Premier exemple : calcul de somme

Deux façons différentes de calculer la somme des entiers inférieurs à n

```
def somme(n):
    S = 0
    for k in range(n):
        S = S+k
    return S
```

```
def somme(n):
    S = 0
    k = 0
    while k < n:
        S = S+k
        k = k+1
    return S
```

Boucle while possible mais sans intérêt ici

Deuxième exemple : suite de Syracuse

$$u_0 = p \ (p \in \mathbb{N}) \quad \text{et :} \quad \forall n \in \mathbb{N}, \quad u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

Conjecture : $\forall p \in \mathbb{N}, \exists k \in \mathbb{N}, u_k = 1$

Deuxième exemple : suite de Syracuse

$$u_0 = p \ (p \in \mathbb{N}) \quad \text{et :} \quad \forall n \in \mathbb{N}, \quad u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon.} \end{cases}$$

Conjecture : $\forall p \in \mathbb{N}, \exists k \in \mathbb{N}, u_k = 1$

Fonction `indice_Syracuse` qui renvoie, pour p donné, le premier indice k tel que $u_k = 1$.

Deuxième exemple : suite de Syracuse

```
def indice_Syracuse(p):
    u = p
    k = 0
    while u != 1:
        k = k+1
        if u%2 == 1:
            u = 3*u+1
        else:
            u = u // 2
    return k
```

Deuxième exemple : suite de Syracuse

```
def indice_Syracuse(p):
    u = p
    k = 0
    while u != 1:
        k = k+1
        if u%2 == 1:
            u = 3*u+1
        else:
            u = u // 2
    return k
```

Remarques

- la boucle termine pour tout entier p uniquement si la conjecture de Syracuse est vérifiée,

Deuxième exemple : suite de Syracuse

```
def indice_Syracuse(p):
    u = p
    k = 0
    while u != 1:
        k = k+1
        if u%2 == 1:
            u = 3*u+1
        else:
            u = u // 2
    return k
```

Remarques

- l'emploi de la division entière `//` a pour unique rôle de maintenir le type `int` de la variable `u`.

Deuxième exemple : suite de Syracuse

```
def indice_Syracuse(p):
    u = p
    k = 0
    while u != 1:
        k = k+1
        if u%2 == 1:
            u = 3*u+1
        else:
            u = u // 2
    return k
```

Remarques

Boucle while nécessaire ici

FOR OU WHILE ?

Problématique usuelle :

boucle **`for`** ou boucle **`while`** ?

FOR OU WHILE ?

Problématique usuelle :

boucle **`for`** ou boucle **`while`** ?

Critère :

FOR OU WHILE ?

Problématique usuelle :

boucle **`for`** ou boucle **`while`** ?

Critère :

- nombre d'itérations de la boucle connu avant exécution $\Rightarrow$ boucle **`for`**,

FOR OU WHILE ?

Problématique usuelle :

boucle **`for`** ou boucle **`while`** ?

Critère :

- nombre d'itérations de la boucle connu avant exécution ⇒ boucle **`for`**,
- nombre d'itérations de la boucle inconnu avant exécution ⇒ boucle **`while`** ?

for ou while ?

Problématique usuelle :

Exemples sur la suite de Syracuse

- Si l'on veut calculer u_{20} à partir d'un entier $u_0 = p$ quelconque, alors on sait qu'on devra faire 20 itérations $\Rightarrow$ boucle **`for`**.

for ou while ?

Problématique usuelle :

Exemples sur la suite de Syracuse

- Si l'on veut calculer u_{20} à partir d'un entier $u_0 = p$ quelconque, alors on sait qu'on devra faire 20 itérations $\Rightarrow$ boucle **`for`**.
- Si l'on souhaite déterminer le plus petit indice k tel que u_k soit égal à 1, à partir d'un entier $u_0 = p$ quelconque, alors on ne sait pas combien d'itérations seront nécessaires $\Rightarrow$ boucle **`while`**.