Introduction à Python

V. Monbet
(document fortement inspiré du livre de R. Cordeau
et des supports de cours de J.-P. Becirspahic)

UFR de Mathématiques
Université de Rennes 1

3 septembre 2018

Outline

Informations diverses

Organisation

- Cet enseignement est organisé sur 12 semaines et 4 blocs.
- Chaque bloc comporte un cours (mardi 8h-10h), un TD et un TP.
- Evaluation : des défits (CC) + un examen terminal (T).
  Calcul de la note : Max((T+CC)/2,T)

Références

- Livres : Cordeau, Swinnen
- Banque d'exercices :
  `https://python.developpez.com/cours/apprendre-python-3/?page=exercices-corriges`

Pour travailler à la fac et/ou à la maison

- Supports de cours et sujets de TD et TP sur Moodle
- Interpréteurs Python
  Spyder (sous ananconda) ou Thonny (facile à installer et léger)

Pourquoi choisir Python ?

- Il existe un très grand nombre de langages de programmation, chacun avec ses avantages et ses inconvénients. Il faut bien en choisir un.

- Éléments d'histoire
  - 1991 : Guido van Rossum publie Python au CWI (Pays-Bas)
  - 1996 : sortie de Numerical Python
  - Fin 2008 : sorties simultanées de Python 2.6 et de Python 3

- Principales caractéristiques de Python : Langage Open Source (importante communauté de développeurs), portable (sur linux, Mac OS, BeOS, NeXTStep, MS-DOS), dynamique, extensible, gratuit, qui permet (sans l'imposer) une approche modulaire et orientée objet de la programmation.

Principales caractéristiques de Python

- **Travail interactif**
  - Nombreux interpréteurs interactifs disponibles
  - Importantes documentations en ligne
  - Développement rapide et incrémentiel
  - Tests et débogage faciles
  - Analyse interactive de données
- **Langage interprété rapide**
  - Interprétation du *bytecode*[1] compilé
  - De nombreux modules sont disponibles à partir de bibliothèques optimisées écrites en C, C++ ou FORTRAN
- **Simplicité du langage**
  - Syntaxe claire et cohérente
  - Indentation significative
  - Gestion automatique de la mémoire (garbage collecteur)
  - Typage dynamique fort : pas de déclaration

1. Le bytecode est un code intermédiaire entre les instructions machines et le code source

Construction des programmes

L'activité essentielle d'un programmeur consiste à résoudre des problèmes.

Considérons par exemple une suite de nombres fournis dans le désordre :
47, 19, 23, 15, 21, 36, 5, 12 . . .
Comment devons-nous nous y prendre pour obtenir d'un ordinateur qu'il les remette dans l'ordre ?

Construction des programmes

L'activité essentielle d'un programmeur consiste à résoudre des problèmes.

Considérons par exemple une suite de nombres fournis dans le désordre :
47, 19, 23, 15, 21, 36, 5, 12 . . .
Comment devons-nous nous y prendre pour obtenir d'un ordinateur qu'il les remette dans l'ordre ?

Pour y arriver, le programmeur devra
- décortiquer tout ce qu'implique pour nous une telle opération de tri (au fait, existe-t-il une méthode unique pour cela, ou bien y en a-t-il plusieurs ?),
- en traduire toutes les étapes en une suite d'instructions simples
telles que par exemple : « comparer les deux premiers nombres, les échanger s'ils ne sont pas dans l'ordre souhaité, recommencer avec le deuxième et le troisième, etc., etc., ...».

Construction des programmes

Le génie logiciel étudie les méthodes de construction des programmes. Plusieurs modèles sont envisageables, entre autres :

- la méthodologie procédurale.
  On emploie l'analyse descendante (division des problèmes) et remontante (réutilisation d'un maximum de sous algorithmes). On s'efforce ainsi de décomposer un problème complexe en sous-programmes plus simples. Ce modèle structure d'abord les actions.

- la méthodologie objet.
  On conçoit des fabriques (classes) qui servent à produire des composants (objets) qui contiennent des données (attributs) et des actions (méthodes). Les classes dérivent (héritage et polymorphisme) de classes de base dans une construction hiérarchique.

Python offre les deux techniques.

Algorithme et programme

Définitions

Définition

Algorithme : ensemble des étapes permettant d'atteindre un but en répétant un nombre fini de fois un nombre fini d'instructions.
Un algorithme se termine en un temps fini.

Définition

Programme : un programme est la traduction d'un algorithme en un langage compilable ou interprétable par un ordinateur.
Il est souvent écrit en plusieurs parties dont une qui pilote les autres : le programme principal.

Compilation

Programme

Les commentaires Un programme source est destiné à l'être humain. Pour en faciliter la lecture, il doit être judicieusement commenté.

La signification de parties non triviales (et uniquement celles-là) doit être expliquée par un commentaire. Un commentaire commence par le caractère # et s'étend jusqu'à la fin de la ligne (voir blocs de gauche). On peut aussi utiliser les triples guillements (voir bloc de droite).

```
#
#  Voici un commentaire
#
```

```
3+4 # et en voici un autre
```

```
"""
 Voici un commentaire,
 utile pour les entetes
 de programmes
"""
```

Commentaires, un autre exemple

```
# Define sharks variable as a list of strings
sharks = ['hammerhead', 'great white', 'dogfish', 'frilled',
    'bullhead', 'requiem']

# For loop that iterates over sharks list
# and prints each string item
for shark in sharks:
   print(shark)
```

Commentaires, exercice

Commenter le code ci-dessous

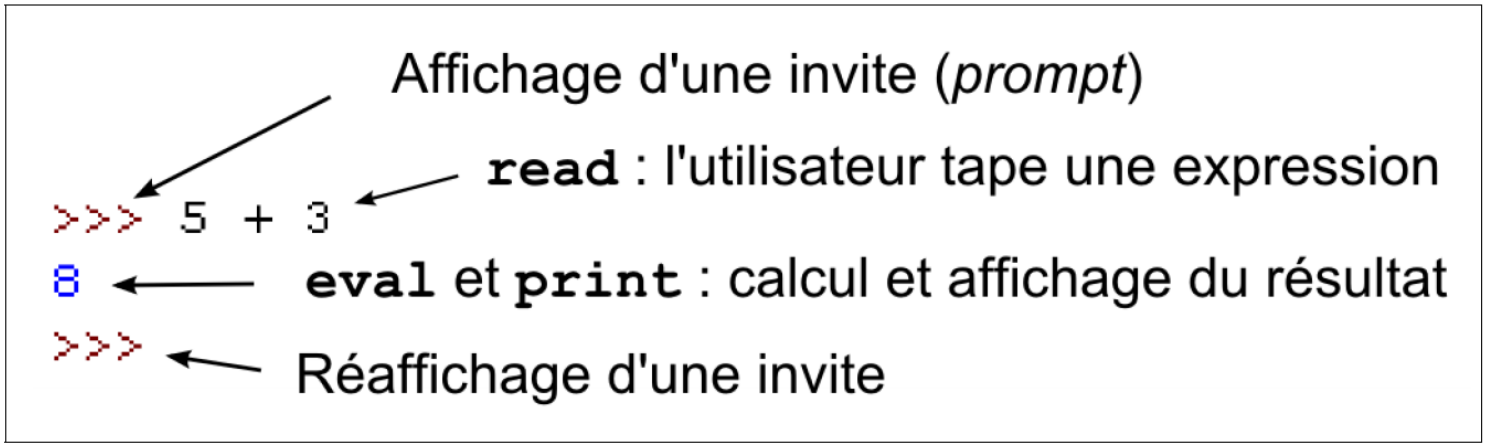
```
list =['while','loop','demo','with','break','and','continue']
max = len(list)
word = "break"
i = 0
while i < max:
    print("Checking element:",i)
    if word == list[i]:
        print("Find it at position:",i)
        break
    i = i + 1
```

Outline

Outline

Deux modes d'exécution

- Soit on enregistre une liste de commandes dans un fichier (*script*) grâce à un éditeur et on l'exécute via un bouton du menu.
- Soit on tape les *commandes en ligne* dans un interpréteur (idle, spyder...) qui exécute la boucle d'évaluation :



Identifiants et mots clé

Comme tout langage, Python utilise des identifiants pour nommer ses objets.

Définition

Un **identifiant** Python valide est une suite non vide de caractères, de longueur quelconque, formée d'un caractère de début et de zéro ou plusieurs caractères de continuation

- un caractère de début peut être n'importe quelle lettre UNICODE ou un souligné.
- un caractère de continuation est un caractère de début, un chiffre ou un point.

Il est important d'utiliser une politique cohérente de nommage des identifiants. Voici les styles préconisés :

- `UPPERCASE` ou `UPPER_CASE` pour les constantes ;
- `TitleCase` pour les classes ;
- `camelCase` pour les fonctions, les méthodes et les interfaces graphiques ;
- `lowercase` ou `lower_case` pour tous les autres identifiants.

Exemples

```
NB_ITEMS = 12 # UPPER_CASE
class MaClasse: pass # TitleCase
def maFonction(): pass # camelCase
mon_id = 5 # lower_case
```

Identifiants et mots clé

Comme tout langage, Python utilise des identifiants pour nommer ses objets.

Définition

Une **expression** est une portion de code que l'interpréteur Python peut évaluer pour obtenir une valeur. Les expressions peuvent être simples ou complexes. Elles sont formées d'une combinaison de littéraux, d'identifiants et d'opérateurs.

Exemple

```
id1 = 15.3
id2 = maFonction(id1)
if id2 > 0:
    id3 = math.sqrt(id2)
else:
    id4 = id1 - 5.5*id2
```

Outline

Le type int

Les entiers se codent de différentes manières.

Exemples

```
>>> 2009 # decimal
2009
>>> 0b11111011001 # binaire
2009
>>> 0o3731        # octal
2009
>>> 0x7d9         # hexadecimal
2009
```

Opérations arithmétiques

```
20 + 3 # 23
20 - 3 # 17
20 * 3 # 60
20**3
20 / 3 # division flotante
20 // 3 # division entiere
20 % 3 # modulo
abs(3 - 20) # valeur absolue
```

Opérations arithmétiques - exercices

1. Donnez la valeur des expressions suivantes :
   ```
   3 + 5 // 3
   4 * 1 // 4
   2 // 3 * 3 - 2
   ```
2. Donnez la valeur des expressions suivantes :
   ```
   18 % 9
   81 + 18 % 9
   (81 + 18) % 9
   ```

Le type booléen

- Deux valeurs possibles : `False`, `True`.
- Opérateurs de comparaison : ==, =!, >,>=, < et <= :

```
2 > 8 # False
2<=8<15 #True
```

- Opérateurs logiques : **not**, **or** et **and**

```
(3 == 3) or (9 > 24) # True (des le premier membre)
(9 > 24) and (3 == 3) # False (des le premier membre)
x = 2
not(x==3) # True
```

Le type booléen, exercice 1

1. Compléter la table suivante

x	y	x and y	x*y	x or y
True	True			
False	True			
True	True			
False	False			

2. Utiliser les opérations arithmétiques ou les fonctions prédéfinies **min** ou **max** sur les entiers 0 (False) et 1 (True) écrire des fonctions `et(x,y)` et `ou(x,y)` qui renvoient la valeur 0 et 1 de façon adéquate.

```
def et(x,y):
    # fonction  et logique
    return(.....)
```

```
def ou(x,y):
    # fonction  ou logique
    return(.....)
```

Le type booléen, exercice 2

1. Que se passe-t-il si on évalue l'expression suivante ?

```
ou(3 == 3, 5 // 1 == 2)
```

2. Quelle est la différence avec l'évaluation de l'expression suivante ?

```
(3==3) or (5//1==2)
```

3. Que se passe-t-il si on évalue l'expression suivante ?

```
et(3 == 4, 5 // 1 == 2)
```

4. Quelle est la différence avec l'évaluation de l'expression suivante ?

```
(3 == 4) and (5 // 1 == 2)
```

Le type flottant

- Un **`float`** est noté avec un point décimal ou en notation exponentielle :

```
2.718
.02
3e8
6.023e23
```

- Les flottants supportent les mêmes opérations que les entiers.
- Les float ont une précision finie indiquée dans `sys.float_info.epsilon`.
- L'import du module math autorise toutes les opérations mathématiques usuelles :

```
import math
print(math.sin(math.pi/4)) # 0.7071067811865475
print(math.degrees(math.pi)) # 180.0
print(math.factorial(9)) # 362880
print(math.log(1024, 2)) # 10.0
```

Le type flottant - Exercice

Écrire un programme PYTHON qui prend en entrée un flottant `x` et renvoie le même chiffre arrondi à 2 chiffres après la virgule.

$$0.70710678118654\,75 \to 0.71$$
$$123.894 \to 123.89$$

Utiliser la fonction **`round`**.

Le type complexe

- Les complexes sont écrits en notation cartésienne formée de deux flottants.
- La partie imaginaire est suffixée par `j` :

```
print(1j) # 1j
et(3 == 4, 5 // 1 == 2) print((2+3j) + (4-7j)) # (6-4j)
print((9+5j).real) # 9.0
print((9+5j).imag) # 5.0
print((abs(3+4j))) # # 5.0 : module
```

- Un module mathématique spécifique (`cmath`) leur est réservé :

```
import cmath
print(cmath.phase(-1 + 0j)) # 3.14159265359
print(cmath.polar(3 + 4j)) # (5.0, 0.927295218001612)
print(cmath.rect(1., cmath.pi/4)) # (0.707106781187+0.707106781187j)
```

Le type complexe, exercice

On considère un point dans le plan défini par ses coordonnées cartésiennes `x` et `y`. Écrire les instructions PYTHON permettant de le caractériser par un nombre complexe de la forme $\rho e^{i\theta}$.

```
import ....
x = 1
y = 2
  ....
```

Outline

Les variables

- On a besoin des **variables** pour stocker les données.

Définition

Une variable est un **identifiant** associé à une valeur. Informatiquement, c'est une **référence d'objet** situé à une adresse mémoire.

- Tant que l'objet n'est pas modifiable (comme les entiers, les flottants, etc.), il n'y a pas de différence notable entre variable et référence d'objet. On verra que la situation change dans le cas des objets modifiables...

Les affectations

Définition

On **affecte** une variable par une valeur en utilisant le signe = (qui n'a rien à voir avec l'égalité en math !). Dans une affectation, le membre de gauche reçoit le membre de droite ce qui nécessite d'évaluer la valeur correspondant au membre de droite avant de l'affecter au membre de gauche.

```
import math
a = 2 # a recoit 2
b = 7.2*math.log(math.e/45.12) - 2*math.pi
c = b**a
```

La valeur d'une variable, comme son nom l'indique, peut évoluer au cours du temps (la valeur antérieure est perdue) :

```
a = a + 1 # 3 (incrementation)
a = a - 1 # 2 (decrementation)
```

Affecter ou comparer ?

Affecter n'est pas comparer

- L'**affectation** a un effet (elle modifie l'état interne du programme en cours d'exécution) mais n'a pas de valeur (on ne peut pas l'utiliser dans une expression) :

```
>>> a = 2
>>> x = (a = 3) + 2
SyntaxError: invalid syntax
```

- La **comparaison** a une valeur utilisable dans une expression mais n'a pas d'effet (l'automate interne représentant l'évolution du programme n'est pas modifié) :

```
>>> x = (a == 3) + 2
>>> x
2
```

Les variantes de l'affectation

Outre l'affectation simple, on peut aussi utiliser les formes suivantes :

```
# affectation simple
v=4
# affectation augmentee
v += 2 #idem a : v = v + 2  si v est deja reference
# affectation de droite a gauche
c = d = 8 # cibles multiples
# affectations paralleles d'une sequence
e, f = 2.7, 5.1 # tuple
g, h, i = ['G', 'H', 'I'] # liste
x, y = coordonneesSouris() # retour multiple d'une fonction
```

Outline

Les chaines de caractères, présentation

Définition

Le type de données non modifiable **`str`** représente une séquence de caractères Unicode. *Non modifiable* signifie qu'une donnée, une fois crée en mémoire, ne pourra plus être changée.

```
syntaxe1 = "Premiere forme avec un retour a la ligne \n"
syntaxe2 = r"Deuxieme forme sans retour a la ligne \n"
syntaxe3 = """
Troisieme forme multi-ligne
"""
guillemets = "L'eau vive"
apostrophes = 'Forme "avec des apostrophes"'
```

Les chaines de caractères, opérations

- Longueur

```
s = "abcde"
len(s) # 5
```

- Concaténation

```
s1 = "abc"
s2 = "defg"
s3 = s1 + s2 # 'abcdefg'
```

- Répétition

```
s4 = "Fi! "
s5=s4*3 #'Fi!Fi!Fi!'
print(s5)
```

Les chaines de caractères, sous chaines

On peut agir sur une chaîne de caractères par des *fonctions* ou des *méthodes*.

- Pour appliquer une *fonction*, on utilise l'opérateur ( ) appliqué à la fonction

```
ch1 = "abc"
long = len(ch1) # 3
```

- On applique une *méthode* à un objet en utilisant la notation pointée entre la donnée/variable à laquelle on applique la méthode, et le nom de la méthode suivi de l'opérateur ( ) appliqué à la méthode :

```
ch2 = "abracadabra"
ch3 = ch2.upper() # "ABRACADABRA"
```

Quelques xemples

```
# s sera notre chaine de test pour toutes les methodes
s = "cHAise basSe"
print(s.lower()) # chaise basse
print(s.upper()) # CHAISE BASSE
print(s.capitalize()) # Chaise basse

print(s.find('se b')) # 4
print(s.replace('HA', 'ha')) # chaise basSe
```

Les chaines de caractères, indiçage simple

Pour indicer une chaîne, on utilise l'opérateur [ ] dans lequel l'indice, un entier signé qui commence à 0 indique la position d'un caractère :

```
s = "Rayon X" # len(s) ==> 7 print(s[0]) # R
print(s[2]) # y
print(s[-1]) # X
print(s[-3]) # n
```

On peut aussi extraire des sous chaines

```
s = "Rayon X" # len(s) ==> 7
s[1:4] # 'ayo' (de l'indice 1 compris  a 4 non compris)
s[-2:] # ' X' (de l'indice -2 compris a la fin)
s[:3]  # 'Ray' (du debut a 3 non compris)
s[3:]  # 'on X' (de l'indice 3 compris a la fin)
s[::2] # 'RynX' (du debut a la fin, de 2 en 2)
```

Les chaines de caractères, exercices

1. Écrire un programme PYTHON pour obtenir une chaîne à partir d'une chaîne donnée où toutes les occurrences de son premier caractère ont été changées en `'&'`, sauf le premier caractère lui-même.

   "restart" $\rightarrow$ "resta&t"

2. Écrire un programme PYTHON pour obtenir une seule chaîne à partir de deux chaînes données, séparées par un espace et permuter les deux premiers caractères de chaque chaîne.
   "abc" et "xyz" $\rightarrow$ "xyc abz"

Outline

Outline

Les instructions conditionnelles

Les instructions conditionnelles se définissent à l'aide du mot clé **if** :

```
if expression booleenne:
    bloc..............
    d instructions 1..
else:
    bloc..............
    d instructions 2..
```

Si l'expression booléenne est vraie, le premier bloc d'instructions est réalisé, dans le cas contraire c'est le second.
Opérateurs courants (à valeurs booléennes) :
$x < y$ (x est strictement plus petit que y) ;
$x > y$ (x est strictement plus grand que y) ;
$x <= y$ (x est inférieur ou égal à y) ;
$x >= y$ (x est supérieur ou égal à y) ;
$x == y$ (x est égal à y) ;
$x! = y$ (x est différent de y).

Les instructions conditionnelles

Exemples

```
if x < 0:
    print("x est negatif")
elif x % 2:
    print("x est positif et impair")
else:
print("x n'est pas negatif et est pair")
```

Les instructions conditionnelles - exercice

Quelles sont les valeurs de `a`, `b` et `c` à la fin de la liste d'instructions.

```
a=2
b=a*a+3
c=b-a
if (c == a) :
    a=1
else :
    a=a+3
if (b + c < a) :
    b=2
else :
    b=4
if (b != c * c) :
    c = 12
else :
    c = -6
```

Les instructions conditionnelles

Les instructions conditionnelles se définissent à l'aide du mot clé **if** :

```
if expression booleenne:
    bloc..............
    d instructions 1..
else:
    bloc..............
    d instructions 2..
```

Les chaînes de caractères peuvent être comparées suivant l'ordre lexicographique :

```
In [1]: 'alpha' < 'omega'
Out[1]: True
In [2]: 'gamma' <= 'beta'
Out[2]: False
```

Les instructions conditionnelles

Les instructions conditionnelles se définissent à l'aide du mot clé **if** :

```
if expression booleenne:
    bloc..............
    d instructions 1..
else:
    bloc..............
    d instructions 2..
```

Expressions booléennes.
L'évaluation d'une expression logique n'est qu'un calcul dont le résultat ne peut prendre que deux valeurs : le vrai (True) et le faux (False).
Les deux fonctions suivantes sont équivalentes :

```
def est_pair(n):
    return n % 2 == 0
```

```
def est_pair(n):
    if n % 2 == 0:
        return True
    else:
        return False
```

Les instructions conditionnelles

Il est possible d'imbriquer plusieurs tests à l'aide du mot-clé **elif** :

```
if expression booleenne 1:
    bloc..............
    d instructions 1..
elif expression booleenne 2:
    bloc..............
    d instructions 2..
else:
    bloc..............
    d instructions 3..
```

- Si l'expression 1 est vraie, le bloc d'instructions 1 est réalisé ;
- Si l'expression 1 est fausse et l'expression 2 vraie, le bloc d'instructions 2 est réalisé ;
- Si les deux expressions sont fausses, le bloc d'instructions 3 est réalisé.

Plusieurs **elif** à la suite peuvent être utilisés pour multiplier les cas possibles.

Les instructions conditionnelles, exercices

Que fait la suite d'instructions suivantes ?

```
x=3
st = "Ca va ?"
ch = ""
x = x // 2
if (x == 1) :
    ch = "How are you ?"
else :
    ch = "Comment allez -vous ?"
ch = ch + "\n"
print (ch)
```

Les instructions conditionnelles, exercices

1. Écrivez une expression conditionnelle, qui à partir d'une température d'eau stockée dans une variable `t` affiche dans le terminal si l'eau à cette température est à l'état liquide, solide ou gazeux.
2. Un service de photocopies facture 0,10 euros les 30 premières photocopies et 0,05 euros les suivantes. Écrire les instructions qui donnent le montant payé après que le client a entré son nombre de photocopies. On pourra utiliser

```
nb = input("Entrez le nombre de photocopies ")
nb = int(nb)  # la sortie de input est une chaine de caracteres,
              # la fonction int permet de la convertir
              # en nombre entier.

# ecrire la suite ici pour calculer le prix en fonction de nb
```

Outline

Les boucles énumérées

La fonction range

La fonction **range** peut prendre entre 1 et 3 arguments entiers :
- **range**(b) énumère les entiers 0,1,2, $\cdots$,b-1 ;
- **range**(a,b) énumère les entiers a,a+1,$\cdots$, b-1 ;
- **range**(a,b,c) énumère les entiers a, a+c, a+2c, $\cdots$, a+nc où n est le plus grand entier vérifiant a +nc $<$ b si c est positif (ie n est la partie entière de (b-a)/c)
et a+nc $>$ b si c est négatif (ie que n est la partie entière de (b-a)/c+1).

```
In [1]: list(range(10))
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
In [2]: list(range(5, 15))
Out[2]: [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
In [3]: list(range(1, 20, 3))
Out[3]: [1, 4, 7, 10, 13, 16, 19]
```

- L'énumération **range**(b) comporte b éléments ;
- l'énumération **range**(a,b) comporte b-a éléments.

La fonction range - exercice

Écrire un code PYTHON permettant d'afficher tous les chiffres impairs inférieurs à une valeur `n` initialement fixée.

Les boucles énumérées

Boucles indexées

On définit une boucle indexée par la construction suivante :

```
for ... in range(...):
    bloc ..........................
    ..............................
    d instructions ................
```

Immédiatement après le mot clé **for** figure le nom d'une variable, qui va prendre les différentes valeurs de l'énumération produite par range. Pour chacune de ces valeurs, le bloc d'instructions qui suit sera exécuté.

```
In [4]: for x in range(2, 10, 3):
...:         print(x, x**2)
2 4
5 25
8 64
```

Les boucles énumérées

Boucles indexées

On définit une boucle indexée par la construction suivante :

```
for ... in range(...):
    bloc ..........................
    ................................
    d instructions ................
```

Il est possible d'imbriquer des boucles à l'intérieur d'autres boucles :

```
In [5]: for x in range(1, 6):
...:             for y in range(1, 6):
...:                 print(x * y, end=' ')
...:             print('/')
1 2 3 4 5 /
2 4 6 8 10 /
3 6 9 12 15 /
4 8 12 16 20 /
5 10 15 20 25 /
```

Boucles itérées - exercices

1. Écrire un programme qui prend en entrée une chaine de caractère puis affiche une lettre sur `k` de cette chaine.
   Exemple : "Le dimanche c'est repos", `k=3`

```
L
d
a
h
c
s
r
o
```

2. Write a program which will find all such numbers which are divisible by 7 but are not a multiple of 5, between 2000 and 3200 (both included). The numbers obtained should be printed in a comma-separated sequence on a single line.
   Hints : Consider use **`range`**(*#begin, #end)* method

Les boucles énumérées

Invariant de boucle

Invariant de boucle

$\rightarrow$ toute assertion vérifiant les conditions suivantes :

Initialisation cette assertion est vraie avant la première itération ;

Conservation si cette assertion est vraie avant une itération, elle le reste avant l'itération suivante ;

Terminaison une fois la boucle terminée, l'invariant fournit une propriété utile pour établir/prouver/analyser l'algorithme.

Exemple : $u_0 = 0$ et $\forall n \in \mathbf{N}$, $u_{n+1} = 2u_n + 1$

```
def u(n):
    x = 0
    for k in range(n):
        x = 2 * x + 1
    return x
```

à l'entrée de la boucle indexée par k, x référence la valeur de u_k.

Les boucles énumérées

Invariant de boucle

Invariant de boucle
$\rightarrow$ toute assertion vérifiant les conditions suivantes :
Initialisation cette assertion est vraie avant la première itération ;
Conservation si cette assertion est vraie avant une itération, elle le reste avant l'itération suivante ;
Terminaison une fois la boucle terminée, l'invariant fournit une propriété utile pour établir/prouver/analyser l'algorithme.

Exemple : $u_0 = 0$ et $\forall n \in \mathbf{N}$, $u_{n+1} = 2u_n + 1$

```
def u(n):
    x = 0
    for k in range(n):
        x = 2 * x + 1
    return x
```

à l'entrée de la boucle indexée par k, x référence la valeur de u_k.
Initialisation : à l'entrée de la boucle indexée par 0, $x = 0 = u_0$.

Les boucles énumérées

Invariant de boucle

$\rightarrow$ toute assertion vérifiant les conditions suivantes :
Initialisation cette assertion est vraie avant la première itération ;
Conservation si cette assertion est vraie avant une itération, elle le reste avant l'itération suivante ;
Terminaison une fois la boucle terminée, l'invariant fournit une propriété utile pour établir/prouver/analyser l'algorithme.

Exemple : $u_0 = 0$ et $\forall n \in \mathbf{N}$, $u_{n+1} = 2u_n + 1$

```
def u(n):
    x = 0
    for k in range(n):
        x = 2 * x + 1
    return x
```

à l'entrée de la boucle indexée par k, x référence la valeur de u_k.
Conservation : si à l'entrée de la boucle indexée par k, $x = u_k$, alors à l'entrée de la boucle indexée par $k+1$, $x = 2u_k + 1 = u_{k+1}$.

Les boucles énumérées

Invariant de boucle

$\rightarrow$ toute assertion vérifiant les conditions suivantes :
Initialisation cette assertion est vraie avant la première itération ;
Conservation si cette assertion est vraie avant une itération, elle le reste avant l'itération suivante ;
Terminaison une fois la boucle terminée, l'invariant fournit une propriété utile pour établir/prouver/analyser l'algorithme.

Exemple : $u_0 = 0$ et $\forall n \in \mathbf{N}$, $u_{n+1} = 2u_n + 1$

```
def u(n):
    x = 0
    for k in range(n):
        x = 2 * x + 1
    return x
```

à l'entrée de la boucle indexée par k, x référence la valeur de u_k.
Terminaison : Le résultat retourné est la valeur de x à l'entrée de la boucle indexée par n, soit u_n.

Les boucles énumérées

Invariant de boucle

$\rightarrow$ toute assertion vérifiant les conditions suivantes :
Initialisation cette assertion est vraie avant la première itération ;
Conservation si cette assertion est vraie avant une itération, elle le reste avant l'itération suivante ;
Terminaison une fois la boucle terminée, l'invariant fournit une propriété utile pour établir/prouver/analyser l'algorithme.

Exemple : $u_0 = 0$ et $\forall n \in \mathbf{N}$, $u_{n+1} = 2u_n + 1$

```
def u(n):
    x = 0
    for k in range(n):
        x = 2 * x + 1
    return x
```

à l'entrée de la boucle indexée par k, x référence la valeur de u_k.
Ici, l'invariant de boucle énoncé prouve la validité de l'algorithme.

Les boucles énumérées

Invariant de boucle

On souhaite calculer $u_n = n!$. Pour cela, on cherche à obtenir l'invariant :
à l'entrée de la boucle indexée par k, x référence la valeur $k!$.
Cet invariant conditionne l'initialisation et la conservation :

```
def fact(n):
    x = 1 # initialisation
    for k in range(n):
        x = x * (k + 1) # conservation
    return x
```

Ici, l'invariant de boucle permet de rédiger l'algorithme.

Les boucles énumérées

Invariant de boucle

On souhaite calculer u_n définie par $u_0 = 0$, $u_1 = 1$ et $u_{k+2} = uk + 1 + uk$.
On cherche à obtenir l'invariant :
à l'entrée de la boucle indexée par k, x référence la valeur de u_k .

Problème : comment calculer $u_k + 1$ à l'aide de la seule valeur de u_k ? Il manque la valeur de u_{k-1}.

Les boucles énumérées

Invariant de boucle

On souhaite calculer u_n définie par $u_0 = 0$, $u_1 = 1$ et $u_{k+2} = uk + 1 + uk$.
On cherche à obtenir l'invariant :
à l'entrée de la boucle indexée par k, x référence la valeur de u_k et y référence la valeur de u_{k-1} .

On en déduit la fonction :

```
def fib(n):
    x, y = 0, 1 # initialisation
    for k in range(n):
        x, y = x + y, x # conservation
    return x
```

Les boucles énumérées

Exercice

On considère $p(x) = \sum_{i=0}^{n-1} a_i x^i$ représenté par $p = [a_0, a_1, a_2, ...]$.

Déterminer un invariant pour établir le rôle de la fonction :

```
def mystere(p, x):
    n = len(p)
    s = 0
    for k in range(n):
        s = x * s + p[n-1-k]
    return s
```

À l'entrée de la boucle indexée par k, ...

Les boucles énumérées

Exercice

On considère $p(x) = \sum_{i=0}^{n-1} a_i x^i$ représenté par $p = [a_0, a_1, a_2, ...]$.

Déterminer un invariant pour établir le rôle de la fonction :

```
def mystere(p, x):
    n = len(p)
    s = 0
    for k in range(n):
        s = x * s + p[n-1-k]
    return s
```

À l'entrée de la boucle indexée par k, s référence la valeur de $\sum_{i=n-k}^{n-1} a_i x^{i+k-n}$

On en déduit que cet algorithme retourne la valeur de $\sum_{i=0}^{n-1} a_i x^i = p(x)$.
Ici, l'invariant de boucle permet d'analyser l'algorithme.

Les boucles énumérées

Exercice

Que font les deux fonctions définies ci-dessous ?

```
def entiers(i, j):
    for k in range(i, j):
        print(k)
    print(j)
entiers(7, 22)
```

```
def entiers(i, j):
    if j % 7 == 0:
    j -= 1
    for k in range(i, j):
        if k % 7 != 0:
            print(k)
    print(j)
entiers(7, 22)
```

Les boucles énumérées

Exercice

Écrire une fonction PYTHON qui correspondra à la fonction $f : \mathbb{N} \to \mathbb{N}$

$$\forall n \in \mathbb{N},\quad f(n) = \begin{cases} n/2 & \text{si} \quad n \quad \text{est pair} \\ 3n+1 & \text{si} \quad n \quad \text{est impair} \end{cases}$$

Comment calculer $fofof(17)$?
Quel résultat doit-on obtenir ?

Outline

Parcours d'une chaîne de caractères

Syntaxe générale de l'instruction **for** :

```
for ... in ...:
    bloc ..........................
    ..............................
    dinstructions ................
```

Ce qui suit le mot-clé **in** doit être une structure de données énumérable (intervalles, chaîne de caractères, listes, fichiers . . .).
Par exemple :

```
def epeler(mot):
    for c in mot:
        print(c)
In [1]: epeler('Pourquoi')

P
o
u
r
q
u
```

Parcours d'une chaîne de caractères

Syntaxe générale de l'instruction**for** :

```
for ... in ...:
    bloc ..........................
    ................................
    dinstructions ................
```

Ce qui suit le mot-clé **in** doit être une structure de données énumérable (intervalles, chaîne de caractères, listes, fichiers . . .).
Dans un langage de programmation n'autorisant que des itérations suivant une progression arithmétique, il faudrait écrire :

```
def epeler(mot):
    for i in range(len(mot)):
        print(mot[i])
```

Parcours d'une chaîne de caractères

Syntaxe générale de l'instruction**for** :

```
for ... in ...:
    bloc ..........................
    ..............................
    dinstructions ................
```

Ce qui suit le mot-clé **in** doit être une structure de données énumérable (intervalles, chaîne de caractères, listes, fichiers . . .).
Il est possible d'énumérer à la fois l'indice et la valeur d'un élément :

```
for (i, c) in enumerate('Pourquoi'):
    print(i, c)
(0, 'P')
(1, 'o')
(2, 'u')
(3, 'r')
(4, 'q')
(5, 'u')
(6, 'o')
(7, 'i')
```

Parcours d'une chaîne de caractères

Syntaxe générale de l'instruction**for** :

```
for ... in ...:
    bloc ..........................
    ..............................
    dinstructions ................
```

Ce qui suit le mot-clé **in** doit être une structure de données énumérable (intervalles, chaîne de caractères, listes, fichiers . . .).
Il est possible d'énumérer 2 énumérables à la fois :

```
for (i, c) in zip(range(1, 10), 'Pourquoi'):
    print(i, c)

(1, 'P')
(2, 'o')
(3, 'u')
(4, 'r')
(5, 'q')
(6, 'u')
(7, 'o')
```

Outline

Boucles conditionnelles

Une boucle conditionnelle exécute une suite d'instructions tant qu'une certaine condition est réalisée.

```
while condition:
    bloc ..........................
    ..............................
    d instructions ................
```

La condition doit être une expression à valeurs booléennes.

```
In [1]: while 1 + 1 == 3:
...:       print('abc')
...:   print('def')
def
```

L'instruction print('abc') n'est jamais exécutée puisque la condition est fausse.

Boucles conditionnelles

Une boucle conditionnelle exécute une suite d'instructions tant qu'une certaine condition est réalisée.

```
while condition:
    bloc ..........................
    ..............................
    d instructions ................
```

La condition doit être une expression à valeurs booléennes.

```
In [1]: while 1 + 1 == 2:
...:       print('abc')
...:   print('def')
abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc
abc abc abc abc abc abc abc abc abc abc abc abc abc abc abc ...
```

L'instruction est éternellement vérifiée.... ce qui conduit au blocage de l'interprète.

Boucles conditionnelles

Une boucle conditionnelle exécute une suite d'instructions tant qu'une certaine condition est réalisée.

```
while condition:
    bloc ..........................
    ................................
    d instructions ................
```

En général, la condition dépend d'une variable au moins dont le contenu sera susceptible d'être modifié dans le corps de la boucle.
Par exemple :

```
In [3]: x = 10
In [4]: while x > 0:
...:           print(x)
...:           x -= 1
10 9 8 7 6 5 4 3 2 1
```

Terminaison d'une boucle conditionnelle

Exercice

Donner le rôle des fonctions suivantes :

```
def mystere(n):
    p = 0
    while (p+1)*(p+1) <= n:
        p += 1
    return p
```

Terminaison d'une boucle conditionnelle

Prouver la terminaison d'une boucle conditionnelle, c'est prouver qu'elle retourne un résultat en **un temps fini**. La recherche d'un invariant de boucle adéquat fournit le plus souvent la solution.
Exemple :

```
def mystere(a, b):
    q, r = 0, a
    while r >= b:
        q, r = q + 1, r - b
    return q, r
```

En général, la condition dépend d'une variable au moins dont le contenu sera susceptible d'être modifié dans le corps de la boucle.
Par exemple :

```
In [3]: x = 10
In [4]: while x > 0:
...:            print(x)
...:            x -= 1
10 9 8 7 6 5 4 3 2 1
```

Terminaison d'une boucle conditionnelle

Prouver la terminaison d'une boucle conditionnelle, c'est prouver qu'elle retourne un résultat en **un temps fini**. La recherche d'un invariant de boucle adéquat fournit le plus souvent la solution.
Exemple :

```
def mystere(a, b):
    q, r = 0, a
    while r >= b:
        q, r = q + 1, r - b
    return q, r
```

Exemple :

```
def mystere(a, b):
    q, r = 0, a
    while r >= b:
    q, r = q + 1, r - b
    return q, r
```

La boucle conditionnelle réalise l'itération de deux suites $(q_n)_{n\in\mathbb{N}}$ et $(r_n)_{n\in\mathbb{N}}$ définies par $q_0 = 0$, $r_0 = a$, et $q_{n+1} = q_n + 1$, $r_{n+1} = r_n - b$.

Terminaison d'une boucle conditionnelle

Prouver la terminaison d'une boucle conditionnelle, c'est prouver qu'elle retourne un résultat en **un temps fini**. La recherche d'un invariant de boucle adéquat fournit le plus souvent la solution.
Exemple :

```
def mystere(a, b):
    q, r = 0, a
    while r >= b:
        q, r = q + 1, r - b
    return q, r
```

Exemple :

```
def mystere(a, b):
    q, r = 0, a
    while r >= b:
    q, r = q + 1, r - b
    return q, r
```

Conclusion : cette fonction retourne la valeur d'un couple (q, r) vérifiant

Terminaison d'une boucle conditionnelle

Exercice

Donner le rôle de la fonction suivante :

```
def mystere(n):
    i, s = 0, 0
    while s < n:
        s += 2 * i + 1
        i += 1
    return i
```

Terminaison d'une boucle conditionnelle

Exercice

Donner le rôle de la fonction suivante :

```
def mystere(n):
    i, s = 0, 0
    while s < n:
        s += 2 * i + 1
        i += 1
    return i
```

À l'entrée de la boucle de rang k, $i = k$ et $s = \sum_{i=0}^{k-1}(2i+1) = k^2$.
Il faut observer que $(p+1)^2 = p^2 + 2p + 1$.
Il existe un unique entier k tel que $(k-1)^2 < n \leq k^2$ donc l'algorithme se termine et retourne cette valeur de $k = [\sqrt{n}]$.

Forcer la sortie d'une boucle

Pour sortir prématurément d'une boucle : **return** ou **break**.
Exemple : recherche d'un caractère dans une chaîne de caractères.

```
def cherche(c, chn):
    for x in chn:
        if x == c:
        return True
    return False
```

Dès que le caractère *c* est trouvé dans la chaîne *chn*, le parcours cesse.

Forcer la sortie d'une boucle

Pour sortir prématurément d'une boucle : **return** ou **break**.
break permet d'interrompre le déroulement des instructions du bloc interne à la boucle.

```
s = 0
while True:
    s += 1
    if randint(1,7) == 6 and randint(1,7) == 6:
        break
```

La boucle ne se termine que si on réalise un double 6 (la terminaison n'est que *probable*).

Forcer la sortie d'une boucle

Pour obtenir le nombre moyen de jets nécessaire à l'obtention d'un double 6, on réalise cette expérience un nombre suffisant de fois :

```
from numpy.random import randint
def test(n):
    e = 0
    for k in range(n):
        s = 0
        while True:
            s += 1
            if randint(1, 7) == 6 and randint(1, 7) == 6:
                break
        e += s
    return e / n
```

```
test(100000)
Out[61]: 36.19483

test(100000)
Out[62]: 35.78437
```

```
test(100000)
Out[63]: 36.08243

test(100000)
Out[64]: 36.05108
```