Bonnes pratiques de programmation

- Les *commentaires* s'écrivent en faisant commencer la ligne par le caractères #

Bonnes pratiques de programmation

- Les *commentaires* s'écrivent en faisant commencer la ligne par le caractères #
- Les noms de variables et de fonction doivent être explicites.

Bonnes pratiques de programmation

- Les *commentaires* s'écrivent en faisant commencer la ligne par le caractères `#`
- Les noms de variables et de fonction doivent être explicites.
- L'instruction `assert <condition>` permet de vérifier que `<condition>` est vérifiée avant de continuer l'exécution du programme. On peut ainsi tester des fonctions ou vérifier des *préconditions* sur des arguments.

Utilisation de librairies

- On peut importer la totalité de la librairie <lib> à l'aide de `import <lib>`. Dans ce cas les fonctions de cette librairie doivent être utilisées en les faisant précéder du nom de la librairie

Exemple

Utilisation de librairies

- On peut importer la totalité de la librairie `<lib>` à l'aide de `import <lib>`. Dans ce cas les fonctions de cette librairie doivent être utilisées en les faisant précéder du nom de la librairie
- Cet import peut se faire en donnant un *alias* : `import <lib> as <alias>`

Exemple

Utilisation de librairies

- On peut importer la totalité de la librairie `<lib>` à l'aide de `import <lib>`. Dans ce cas les fonctions de cette librairie doivent être utilisées en les faisant précéder du nom de la librairie
- Cet import peut se faire en donnant un *alias* : `import <lib> as <alias>`
- Pour importer simple la fonction `<fonc>` de la librairie `<lib>`, on utilise `from <lib> import <fonc>`. Le nom de la fonction est alors utilisé directement.

Exemple

Utilisation de librairies

- On peut importer la totalité de la librairie <lib> à l'aide de `import <lib>`. Dans ce cas les fonctions de cette librairie doivent être utilisées en les faisant précéder du nom de la librairie
- Cet import peut se faire en donnant un *alias* : `import <lib> as <alias>`
- Pour importer simple la fonction <fonc> de la librairie <lib>, on utilise `from <lib> import <fonc>`. Le nom de la fonction est alors utilisé directement.

Exemple

```
1 import randint
2 de = randint(1,6)
```

Utilisation de librairies

- On peut importer la totalité de la librairie <lib> à l'aide de `import <lib>`. Dans ce cas les fonctions de cette librairie doivent être utilisées en les faisant précéder du nom de la librairie
- Cet import peut se faire en donnant un *alias* : `import <lib> as <alias>`
- Pour importer simple la fonction <fonc> de la librairie <lib>, on utilise `from <lib> import <fonc>`. Le nom de la fonction est alors utilisé directement.

Exemple

```
1 import randint
2 de = randint(1,6)
```

```
1 from random import randint
2 de = randint(1,6)
```

Types de base

Type	Opérations	Commentaires
`int`	`+`, `-`, `*`, `//`, `%`	Entiers signés ou non signés. Taille dynamique limitée par la mémoire

Types de base

Type	Opérations	Commentaires
`int`	`+`, `-`, `*`, `//`, `%`	Entiers signés ou non signés. Taille dynamique limitée par la mémoire
`float`	`+`, `-`, `*`, `/`, `**`	Représentation des nombres en virgule flottante (norme ieee754 : mantisse sur 53 bits, exposant sur 11 bits). Fonctions élémentaires dans `math.h`

Types de base

Type	Opérations	Commentaires
`int`	`+`, `-`, `*`, `//`, `%`	Entiers signés ou non signés. Taille dynamique limitée par la mémoire
`float`	`+`, `-`, `*`, `/`	Représentation des nombres en virgule flottante (norme ieee754 : mantisse sur 53 bits, exposant sur 11 bits). Fonctions élémentaires dans `math`
`bool`	`or` `and`, `not`, `all`, `any`	Evaluations paresseuses des expressions.

Définir une fonction en Python

Pour définir une fonction en Python :

Définir une fonction en Python

Pour définir une fonction en Python :

- qui ne renvoie pas de valeur :

```
def <nom_fonction>(<arguments>):
    <instruction>
```

- qui renvoie une valeur :

```
def <nom_fonction>(<arguments>):
    <instruction>
    return <resultat>
```

Instructions conditionnelles

- Sans clause `else`

```
1  if <condition>:
2      <instructions>
```

Exécute les `<instructions>` si la `condition` est vérifiée.

Instructions conditionnelles

- Sans clause `else`

```
if <condition>:
    <instructions>
```

Exécute les `<instructions>` si la `condition` est vérifiée.

- Avec clause `else`

```
if <condition>:
    <instructions1>
else:
    <instructions2>
```

Cela permet d'exécuter les `<instructions1>` si la `condition` est vérifiée, sinon on exécute les `<instructions2>`.

Opérateurs de comparaison

- L'égalité se teste avec ==

Opérateurs de comparaison

- L'égalité se teste avec ==
- La différence avec !=

Opérateurs de comparaison

- L'égalité se teste avec `==`
- La différence avec `!=`
- Plus grand ou égal avec `>=`, plus petit ou égal avec `<=`

Opérateurs de comparaison

- L'égalité se teste avec `==`
- La différence avec `!=`
- Plus grand ou égal avec `>=`, plus petit ou égal avec `<=`
- Plus grand strictement avec `>`, plus petit strictement avec `<`

Boucles `while`

Boucles `while`

- La syntaxe d'une boucle `while` en Python est :

```
while <condition>:
    <instruction>
```

Cela permet d'exécuter les `<instructions>` tant que la `<condition>` est vérifiée.

Boucles `while`

- La syntaxe d'une boucle `while` en Python est :

```
while <condition>:
    <instruction>
```

  Cela permet d'exécuter les `<instructions>` tant que la `<condition>` est vérifiée.

- L'instruction `break` permet de sortir de la boucle de façon anticipée.

Boucles `while`

- La syntaxe d'une boucle `while` en Python est :

```
while <condition>:
    <instruction>
```

  Cela permet d'exécuter les `<instructions>` tant que la `<condition>` est vérifiée.

- L'instruction `break` permet de sortir de la boucle de façon anticipée.
- On ne sait pas a priori combien de fois cette boucle sera exécutée (et elle peut même être infinie), on dit que c'est une boucle non bornée.

Boucles `for`

Boucles `for`

- Les instructions :

```
for <variable> in range(<entier>):
    <instructions>
```

créent une variable parcourant les entiers de 0 à `<entier>` (exclu).

Boucles `for`

- Les instructions :

```
for <variable> in range(<entier>):
    <instructions>
```

  créent une variable parcourant les entiers de 0 à `<entier>` (exclu).

- Les `<instructions>` indentées qui suivent seront exécutées pour chaque valeur prise par la variable.

Boucles `for`

- Les instructions :

```
for <variable> in range(<entier>):
    <instructions>
```

  créent une variable parcourant les entiers de 0 à `<entier>` (exclu).
- Les `<instructions>` indentées qui suivent seront exécutées pour chaque valeur prise par la variable.
- L'instruction `break` permet de sortir de la boucle de façon anticipée.

Boucles `for`

- Les instructions :

```
for <variable> in range(<entier>):
    <instructions>
```

  créent une variable parcourant les entiers de 0 à `<entier>` (exclu).
- Les `<instructions>` indentées qui suivent seront exécutées pour chaque valeur prise par la variable.
- L'instruction `break` permet de sortir de la boucle de façon anticipée.
- La boucle `for` permet donc de répéter un nombre prédéfini de fois des instructions, on dit que c'est une boucle bornée.

Exemple 1

Ecrire et tester une fonction `syracuse` qui prend en argument un entier naturel n et renvoie $n/2$ si n est pair et $3n+1$ sinon.

Exemple 1

Ecrire et tester une fonction `syracuse` qui prend en argument un entier naturel n et renvoie $n/2$ si n est pair et $3n+1$ sinon.

```
1  def syracuse(n):
2      if n%2 == 0:
3          return n//2
4      else:
5          return 3*n+1
```

Exemple 2

Ecrire une fonction `serie_harmonique` qui prend en argument un entier n et renvoie la somme $\sum_{k=1}^{n} \frac{1}{k}$

Exemple 2

Ecrire une fonction `serie_harmonique` qui prend en argument un entier n et renvoie la somme $\sum_{k=1}^{n} \frac{1}{k}$

```
1  def serie_harmonique(n):
2      somme = 0
3      for i in range(1,n+1):
4          somme = somme + 1/i
5      return somme
```

Exemple 3

Ecrire une fonction `pgcd` qui prend en argument deux entiers naturels a et b et renvoie leur PGCD.

Exemple 3

Ecrire une fonction `pgcd` qui prend en argument deux entiers naturels a et b et renvoie leur PGCD.

🛟 on rappelle que l'algorithme consiste –tant que b n'est pas nul- à effectuer la division euclidienne de a par b. En remplaçant à chaque étape a par b et b par r.

Exemple 3

Ecrire une fonction pgcd qui prend en argument deux entiers naturels a et b et renvoie leur PGCD.

 on rappelle que l'algorithme consiste –tant que b n'est pas nul- à effectuer la division euclidienne de a par b. En remplaçant à chaque étape a par b et b par r.

- Version 1 :

```
1  def pgcd(a,b):
2      while b!=0:
3          a,b = b, a%b
4      return a
```

Exemple 3

Ecrire une fonction pgcd qui prend en argument deux entiers naturels a et b et renvoie leur PGCD.

🛟 on rappelle que l'algorithme consiste –tant que b n'est pas nul- à effectuer la division euclidienne de a par b. En remplaçant à chaque étape a par b et b par r.

- Version 1 :

```
1  def pgcd(a,b):
2      while b!=0:
3          a,b = b, a%b
4      return a
```

- Version 2 :

```
1  def pgcd(a,b):
2      if b == 0:
3          return a
4      return pgcd(b,a%b)
```

Exemple 3

Ecrire une fonction pgcd qui prend en argument deux entiers naturels a et b et renvoie leur PGCD.

 on rappelle que l'algorithme consiste –tant que b n'est pas nul- à effectuer la division euclidienne de a par b. En remplaçant à chaque étape a par b et b par r.

- Version 1 : iterative

```
1  def pgcd(a,b):
2      while b!=0:
3          a,b = b, a%b
4      return a
```

- Version 2 : récursive

```
1  def pgcd(a,b):
2      if b == 0:
3          return a
4      return pgcd(b,a%b)
```

Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du mêmte type).

Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du mêmte type).
- Une liste se note entre crochets : [et]

Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du mêmte type).
- Une liste se note entre crochets : [et]
- Les éléments sont séparés par des virgules

Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du mêmte type).
- Une liste se note entre crochets : [et]
- Les éléments sont séparés par des virgules
- Les éléments d'une liste sont repérés par leur position dans la liste, on dit leur indice. Attention, la numérotation commence à zéro.

Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du mêmte type).
- Une liste se note entre crochets : [et]
- Les éléments sont séparés par des virgules
- Les éléments d'une liste sont repérés par leur position dans la liste, on dit leur indice. Attention, la numérotation commence à zéro.

Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du mêmte type).
- Une liste se note entre crochets : [et]
- Les éléments sont séparés par des virgules
- Les éléments d'une liste sont repérés par leur position dans la liste, on dit leur indice. Attention, la numérotation commence à zéro.
- On peut accéder à un élément en indiquant le nom de la liste puis l'indice de cet élément entre crochet

Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du mêmte type).
- Une liste se note entre crochets : `[` et `]`
- Les éléments sont séparés par des virgules
- Les éléments d'une liste sont repérés par leur position dans la liste, on dit leur indice. Attention, la numérotation commence à zéro.
- On peut accéder à un élément en indiquant le nom de la liste puis l'indice de cet élément entre crochet
- L'erreur `IndexError` indique qu'on tente d'accéder à un indice qui n'existe pas.

Les listes de Python

- Les listes de Python sont des structures contenant zéro, une ou plusieurs valeurs (pas forcément du mêmte type).
- Une liste se note entre crochets : `[` et `]`
- Les éléments sont séparés par des virgules
- Les éléments d'une liste sont repérés par leur position dans la liste, on dit leur indice. Attention, la numérotation commence à zéro.
- On peut accéder à un élément en indiquant le nom de la liste puis l'indice de cet élément entre crochet
- L'erreur `IndexError` indique qu'on tente d'accéder à un indice qui n'existe pas.
- La longueur d'une liste (ie. son nombre d'éléments) s'obtient à l'aide de la fonction `len`.

Opérations sur les listes

Les opérations suivantes permettent de manipuler les listes (ajout, suppression, insertion d'éléments). On fera bien attention à la syntaxe on met le nom de la liste suivi d'un point suivi de l'opération à effectuer (voir exemples)

- `append` : permet d'ajouter un élément à la fin d'une liste. Par exemple : `ma_liste.append(elt)` va ajouter `elt` à la fin de `ma_liste`.

Opérations sur les listes

Les opérations suivantes permettent de manipuler les listes (ajout, suppression, insertion d'éléments). On fera bien attention à la syntaxe on met le nom de la liste suivi d'un point suivi de l'opération à effectuer (voir exemples)

- `append` : permet d'ajouter un élément à la fin d'une liste. Par exemple : `ma_liste.append(elt)` va ajouter `elt` à la fin de `ma_liste`.
- `pop` permet de récupérer un élement de la liste tout en le supprimant de la liste. Par exemple `elt=ma_liste.pop(2)` va mettre dans `elt` `ma_liste[2]` et dans le même temps supprimer cet élément de la liste.

Opérations sur les listes

Les opérations suivantes permettent de manipuler les listes (ajout, suppression, insertion d'éléments). On fera bien attention à la syntaxe on met le nom de la liste suivi d'un point suivi de l'opération à effectuer (voir exemples)

- `append` : permet d'ajouter un élément à la fin d'une liste. Par exemple : `ma_liste.append(elt)` va ajouter `elt` à la fin de `ma_liste`.
- `pop` permet de récupérer un élement de la liste tout en le supprimant de la liste. Par exemple `elt=ma_liste.pop(2)` va mettre dans `elt` `ma_liste[2]` et dans le même temps supprimer cet élément de la liste.
  ❗ On utilisera le plus souvent `pop` sans argument, dans ce cas c'est le dernier élément de la liste qui est supprimé

Spécificité de Python

Les listes de Python sont mutables, c'est à dire que les modifications faites sur une liste passée en argument à une fonction sont effectivement réalisées sur la liste. Ce n'est pas le cas sur les arguments de type entier ou flottants.

Exemples

- Ce programme affiche 42 car `n` étant de type entier l'opération effectuée sur `n` ne se répercute pas sur l'argument de la fonction.

```
1  def carre(n):
2      n = n * n
3
4  n = 42
5  carre(n)
6  print(n)
```

Exemples

- Ce programme affiche 42 car `n` étant de type entier l'opération effectuée sur `n` ne se répercute pas sur l'argument de la fonction.

```
1  def carre(n):
2    n = n * n
3
4  n = 42
5  carre(n)
6  print(n)
```

- Ce programme modifie la liste passée en argument et donc affichera `[5,7]`

```
1  def ajoute(liste,valeur):
2    liste.append(valeur)
3
4  liste = [5]
5  liste.ajoute(7)
6  print(liste)
```

Création de listes

On peut créer des listes de diverses façons en Python :

Création de listes

On peut créer des listes de diverses façons en Python :

- Par ajout succesif d'élement on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.

Création de listes

On peut créer des listes de diverses façons en Python :

- Par ajout succesif d'élement on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- Par répétition du même élément on utilise alors le caractère `*` pour indiquer le nombre de répétitions.

Création de listes

On peut créer des listes de diverses façons en Python :

- Par ajout succesif d'élement on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- Par répétition du même élément on utilise alors le caractère `*` pour indiquer le nombre de répétitions.
  Par exemple : `hesitation = ["euh"]*4`

Création de listes

On peut créer des listes de diverses façons en Python :

- Par ajout succesif d'élement on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- Par répétition du même élément on utilise alors le caractère `*` pour indiquer le nombre de répétitions.
  Par exemple : `hesitation = ["euh"]*4`

Création de listes

On peut créer des listes de diverses façons en Python :

- Par ajout succesif d'élement on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- Par répétition du même élément on utilise alors le caractère `*` pour indiquer le nombre de répétitions.
  Par exemple : `hesitation = ["euh"]*4`
- Par compréhension, c'est à dire en indiquant la définition des éléments qui composent la liste.

Création de listes

On peut créer des listes de diverses façons en Python :

- Par ajout succesif d'élement on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- Par répétition du même élément on utilise alors le caractère `*` pour indiquer le nombre de répétitions.
  Par exemple : `hesitation = ["euh"]*4`
- Par compréhension, c'est à dire en indiquant la définition des éléments qui composent la liste.
  Par exemple la liste `puissances2 = [1, 2, 4, 8, 16, 32, 64, 128]` est constitué des huits premières puissances de 2

Création de listes

On peut créer des listes de diverses façons en Python :

- Par ajout succesif d'élement on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- Par répétition du même élément on utilise alors le caractère `*` pour indiquer le nombre de répétitions.
  Par exemple : `hesitation = ["euh"]*4`
- Par compréhension, c'est à dire en indiquant la définition des éléments qui composent la liste.
  Par exemple la liste `puissances2 = [1, 2, 4, 8, 16, 32, 64, 128]` est constitué des huits premières puissances de 2
  Elle contient donc $2^0, 2^1, 2^2, \ldots 2^7$, ce qui se traduit en Python par :

Création de listes

On peut créer des listes de diverses façons en Python :

- Par ajout succesif d'élement on part alors d'une liste (éventuellement vide) et on ajoute chaque élément à l'aide d'instruction `append`.
- Par répétition du même élément on utilise alors le caractère `*` pour indiquer le nombre de répétitions.
  Par exemple : `hesitation = ["euh"]*4`
- Par compréhension, c'est à dire en indiquant la définition des éléments qui composent la liste.
  Par exemple la liste `puissances2 = [1, 2, 4, 8, 16, 32, 64, 128]` est constitué des huits premières puissances de 2
  Elle contient donc $2^0, 2^1, 2^2, \dots 2^7$, ce qui se traduit en Python par :
  `puissances2 = [2**k for k in range(8)]`

Tranches (*slices*)

- On peut extraire une tranche d'une liste en donnant entre crochets l'indice du premier élément puis l'indice du dernier (qui sera exclu) séparé par un `:`.

Tranches (*slices*)

- On peut extraire une tranche d'une liste en donnant entre crochets l'indice du premier élément puis l'indice du dernier (qui sera exclu) séparé par un `:`.
  Par exemple si la liste est `l=[2,3,5,7,11,13,17,19]` alors `l[2:4]` est une liste qui contient `[5,7]`.

Tranches (*slices*)

- On peut extraire une tranche d'une liste en donnant entre crochets l'indice du premier élément puis l'indice du dernier (qui sera exclu) séparé par un `:`.
  Par exemple si la liste est `l=[2,3,5,7,11,13,17,19]` alors `l[2:4]` est une liste qui contient `[5,7]`.
- Si l'indice du premier est omis alors la tranche commmence à l'indice 0.

Tranches (*slices*)

- On peut extraire une tranche d'une liste en donnant entre crochets l'indice du premier élément puis l'indice du dernier (qui sera exclu) séparé par un `:`.
  Par exemple si la liste est `l=[2,3,5,7,11,13,17,19]` alors `l[2:4]` est une liste qui contient `[5,7]`.
- Si l'indice du premier est omis alors la tranche commmence à l'indice 0.
  Avec la même liste `l`, on a `l[:5]` est une liste qui contient `[2,3,5,7,11]`.

Tranches (*slices*)

- On peut extraire une tranche d'une liste en donnant entre crochets l'indice du premier élément puis l'indice du dernier (qui sera exclu) séparé par un `:`.
  Par exemple si la liste est `l=[2,3,5,7,11,13,17,19]` alors `l[2:4]` est une liste qui contient `[5,7]`.
- Si l'indice du premier est omis alors la tranche commmence à l'indice 0.
  Avec la même liste `l`, on a `l[:5]` est une liste qui contient `[2,3,5,7,11]`.
- Si l'indice du dernier est omis alors la tranche va jusqu'à la fin de la liste.

Tranches (*slices*)

- On peut extraire une tranche d'une liste en donnant entre crochets l'indice du premier élément puis l'indice du dernier (qui sera exclu) séparé par un `:`.
  Par exemple si la liste est `l=[2,3,5,7,11,13,17,19]` alors `l[2:4]` est une liste qui contient `[5,7]`.
- Si l'indice du premier est omis alors la tranche commmence à l'indice 0.
  Avec la même liste `l`, on a `l[:5]` est une liste qui contient `[2,3,5,7,11]`.
- Si l'indice du dernier est omis alors la tranche va jusqu'à la fin de la liste.
  Avec la même liste `l`, on a `l[7:]` est une liste qui contient `[19]`.

Tuples

- Les tuples sont le pendant non mutables des listes. Ils se notent entre parenthèses (et), les éléments sont aussi séparés par des virgules.

Exemple

```
1 anniv = (31,"Janvier",1956)
2 print("Mois de naissance = ",anniv[1])
3 anniv[2] = 1970 #provoque une erreur
```

Tuples

- Les tuples sont le pendant non mutables des listes. Ils se notent entre parenthèses `(` et `)`, les éléments sont aussi séparés par des virgules.
- De même que pour les listes, on peut accéder à la longueur avec `len`, aux éléments avec la notation crochet et le parcours avec une boucle `for` est aussi possible.

Exemple

```
1 anniv = (31,"Janvier",1956)
2 print("Mois de naissance = ",anniv[1])
3 anniv[2] = 1970 #provoque une erreur
```

Tuples

- Les tuples sont le pendant non mutables des listes. Ils se notent entre parenthèses `(` et `)`, les éléments sont aussi séparés par des virgules.
- De même que pour les listes, on peut accéder à la longueur avec `len`, aux éléments avec la notation crochet et le parcours avec une boucle `for` est aussi possible.
- La modification par contre n'est pas possible

Exemple

```
1 anniv = (31,"Janvier",1956)
2 print("Mois de naissance = ",anniv[1])
3 anniv[2] = 1970 #provoque une erreur
```

Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaines de caractères.

Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaines de caractères.
  Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre `"n"`

Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaines de caractères.
  Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre `"n"`
- Le parcours par élément peut aussi se faire sur une chaine de caractères.

Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaines de caractères.
  Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre `"n"`
- Le parcours par élément peut aussi se faire sur une chaine de caractères.
  Pour afficher chaque lettre du mot "Génial", on peut donc écrire :

Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaines de caractères.
  Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre `"n"`
- Le parcours par élément peut aussi se faire sur une chaine de caractères.
  Pour afficher chaque lettre du mot "Génial", on peut donc écrire :

```
1 for lettre in mot:
2     print(lettre)
```

Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaines de caractères.
  Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre `"n"`
- Le parcours par élément peut aussi se faire sur une chaine de caractères.
  Pour afficher chaque lettre du mot "Génial", on peut donc écrire :

```
1 for lettre in mot:
2     print(lettre)
```

- Comme les tuples, les chaines de caractères sont non mutables.

Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaines de caractères.
  Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre `"n"`
- Le parcours par élément peut aussi se faire sur une chaine de caractères.
  Pour afficher chaque lettre du mot "Génial", on peut donc écrire :

```
1 for lettre in mot:
2     print(lettre)
```

- Comme les tuples, les chaines de caractères sont non mutables.
- ❗ Les variables lues au clavier (instruction `input`) ou issus de la lecture d'un fichier sont des chaines de caractères. On doit les convertir dans le type approprié pour les utiliser comme nombre.

Chaines de caractères

- La notation avec les crochets permettant d'accéder aux éléments d'une liste s'utilise aussi avec les chaines de caractères.
  Par exemple si `mot = "Génial"` alors `mot[2]` contient la lettre `"n"`
- Le parcours par élément peut aussi se faire sur une chaine de caractères.
  Pour afficher chaque lettre du mot "Génial", on peut donc écrire :

```
1 for lettre in mot:
2     print(lettre)
```

- Comme les tuples, les chaines de caractères sont non mutables.
- ❗ Les variables lues au clavier (instruction `input`) ou issus de la lecture d'un fichier sont des chaines de caractères. On doit les convertir dans le type approprié pour les utiliser comme nombre.
- La fonction `split` permet de renvoyer une liste de sous chaines en utilisant le séparateur donné en argument.

Exemple

Ecrire une fonction `check_date` qui prend en argument une chaine de caractères et renvoie `True` si cette chaine est une date valide au format JJ/MM/AAAA et `False` sinon. Pour simplifier on testera simplement que le jour est entre 1 et 31 et le mois entre 1 et 12.

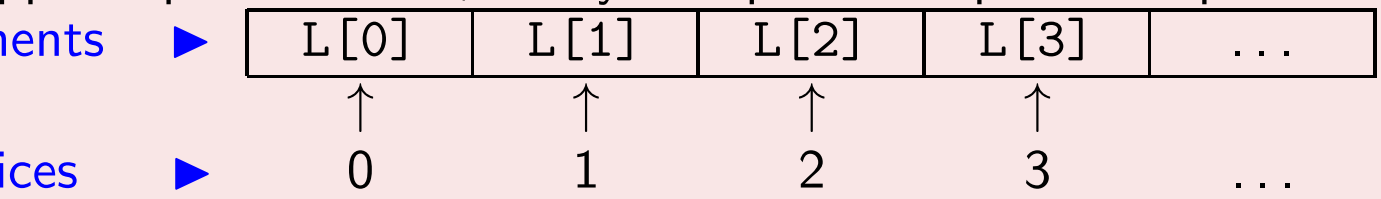
Exemple

Ecrire une fonction `check_date` qui prend en argument une chaine de caractères et renvoie `True` si cette chaine est une date valide au format JJ/MM/AAAA et `False` sinon. Pour simplifier on testera simplement que le jour est entre 1 et 31 et le mois entre 1 et 12.

```
1  def check_date(date):
2      ldate = date.split("/")
3      if len(ldate)!=3:
4          return False
5      jour,mois =  int(ldate[0]),int(ldate[1])
6      if jour<1 or jour>31 or mois<1 or mois>12:
7          return False
8      return True
```

Parcours d'une liste

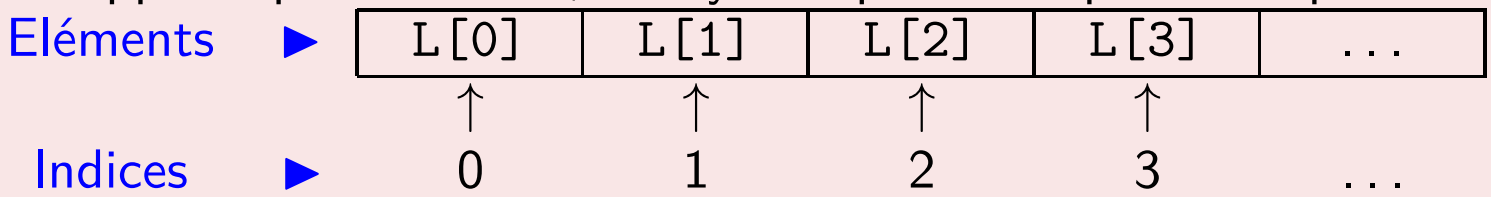
On rappelle qu'une liste `L`, en Python peut se représenter par le schéma suivant :



On peut parcourir cette liste :

Parcours d'une liste

On rappelle qu'une liste `L`, en Python peut se représenter par le schéma suivant :

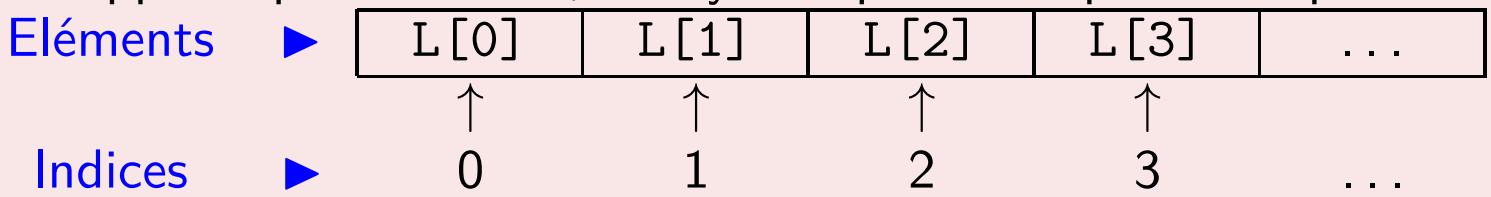


On peut parcourir cette liste :

- Par indice (on se place sur la seconde ligne du schéma ci-dessus) et on crée une variable (un entier) qui va parcourir la liste des indices :
  `for indice in range(len(L))`
  Il faut alors accéder aux éléments en utilisant leurs indices.

Parcours d'une liste

On rappelle qu'une liste `L`, en Python peut se représenter par le schéma suivant :



On peut parcourir cette liste :

- Par indice (on se place sur la seconde ligne du schéma ci-dessus) et on crée une variable (un entier) qui va parcourir la liste des indices :
  `for indice in range(len(L))`
  Il faut alors accéder aux éléments en utilisant leurs indices.
- Par élément (on se place sur la première ligne du schéma ci-dessus) et on crée une variable qui va parcourir directement la liste des éléments :
  `for element in L`
  La variable de parcours (ici `element`) contient alors directement les éléments).

Exemple 1

Ecrire une fonction `est_dans` qui prend en argument un entier `n` et une liste d'entiers `l` et renvoie `True` si `n` est dans `l` et `False` sinon. On écrira une version utilisant un parcours par valeur et une version utilisant un parcours par indice.

Exemple 1

Ecrire une fonction `est_dans` qui prend en argument un entier `n` et une liste d'entiers `l` et renvoie `True` si `n` est dans `l` et `False` sinon. On écrira une version utilisant un parcours par valeur et une version utilisant un parcours par indice.

- Parcours par élément :

```
1  def est_dans(n,l):
2      for x in l:
3          if x==n:
4              return True
5      return False
```

Exemple 1

Ecrire une fonction `est_dans` qui prend en argument un entier `n` et une liste d'entiers `l` et renvoie `True` si `n` est dans `l` et `False` sinon. On écrira une version utilisant un parcours par valeur et une version utilisant un parcours par indice.

- Parcours par élément :

```
1  def est_dans(n,l):
2      for x in l:
3          if x==n:
4              return True
5      return False
```

- Parcours par indice :

```
1  def est_dans_ind(n,l):
2      for i in range(len(l)):
3          if l[i]==n:
4              return True
5      return False
```

Exemple 2

Ecrire une fonction `max_liste` qui prend en argument une liste non vide d'entiers `l` et renvoie le maximum des éléments de cette liste

Exemple 2

Ecrire une fonction `max_liste` qui prend en argument une liste non vide d'entiers `l` et renvoie le maximum des éléments de cette liste

```
1  def max_liste(l):
2      # la liste doit être non vide
3      assert len(l)!=0
4      current_max = l[0]
5      for elt in l:
6          if elt>current_max:
7              current_max = elt
8      return current_max
```

Gestions des fichiers en Python

En python, on peut ouvrir un fichier présent sur l'ordinateur à l'aide de l'instruction `open`. Cette instruction renvoie une variable appelée descripteur de fichier et prend un paramètre indiquant le mode d'ouverture du fichier :

Gestions des fichiers en Python

En python, on peut ouvrir un fichier présent sur l'ordinateur à l'aide de l'instruction `open`. Cette instruction renvoie une variable appelée descripteur de fichier et prend un paramètre indiquant le mode d'ouverture du fichier :

- `"r"` (read) pour ouvrir le fichier en lecture. C'est le mode par défaut.

Gestions des fichiers en Python

En python, on peut ouvrir un fichier présent sur l'ordinateur à l'aide de l'instruction `open`. Cette instruction renvoie une variable appelée descripteur de fichier et prend un paramètre indiquant le mode d'ouverture du fichier :

- "r" (read) pour ouvrir le fichier en lecture. C'est le mode par défaut.
- "w" (write) pour ouvrir le fichier en écriture. Attention, le contenu initial du fichier est alors perdu.

Gestions des fichiers en Python

En python, on peut ouvrir un fichier présent sur l'ordinateur à l'aide de l'instruction `open`. Cette instruction renvoie une variable appelée descripteur de fichier et prend un paramètre indiquant le mode d'ouverture du fichier :

- "r" (read) pour ouvrir le fichier en lecture. C'est le mode par défaut.
- "w" (write) pour ouvrir le fichier en écriture. Attention, le contenu initial du fichier est alors perdu.
- "a" (append) pour ouvrir le fichier en ajout.

Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier crée à l'aide de l'instruction `open` :

Exemples

Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier crée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`

Exemples

Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier crée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`

Exemples

Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier crée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`

Exemples

Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier crée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`
- Fermeture avec `close`

Exemples

Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier crée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`
- Fermeture avec `close`

Exemples

Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier crée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`
- Fermeture avec `close`

Exemples

Ouvrir le fichier "truc.txt", lire sa première ligne puis le refermer.

Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier crée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`
- Fermeture avec `close`

Exemples

Ouvrir le fichier "truc.txt", lire sa première ligne puis le refermer.

```
fic = open("truc.txt","r")
```

Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier crée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`
- Fermeture avec `close`

Exemples

Ouvrir le fichier "truc.txt", lire sa première ligne puis le refermer.

```
fic = open("truc.txt","r")
lig1 = fic.readline()
```

Opérations sur les descripteurs de fichiers

Les opérations suivantes sont possibles sur un descripteur de fichier crée à l'aide de l'instruction `open` :

- Lecture du contenu complet du fichier avec `read`
- Lecture du contenu ligne par ligne avec `readline`
- Ecriture avec de `write`
- Fermeture avec `close`

Exemples

Ouvrir le fichier "truc.txt", lire sa première ligne puis le refermer.

```
fic = open("truc.txt","r")
lig1 = fic.readline()
fic.close()
```

Algorithmique

Dans l'étude d'un algorithme, on s'intéresse aux notions suivantes :

Algorithmique

Dans l'étude d'un algorithme, on s'intéresse aux notions suivantes :

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes avec des boucles non bornées)

Algorithmique

Dans l'étude d'un algorithme, on s'intéresse aux notions suivantes :

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes avec des boucles non bornées)
2. correction : peut-on garantir que l'algorithme fournit la réponse attendue ?

Algorithmique

Dans l'étude d'un algorithme, on s'intéresse aux notions suivantes :

1. terminaison : peut-on garantir que l'algorithme se termine en un temps fini ? (ne concerne que les algorithmes avec des boucles non bornées)
2. correction : peut-on garantir que l'algorithme fournit la réponse attendue ?
3. complexité : evolution du temps d'exécution de l'algorithme en fonction de la taille des données.

Preuve de la terminaison d'un algorithme

- Seules les boucles non bornées (de type boucle `while`) peuvent se répéter indéfiniment. Par conséquent, prouver la terminaison d'un algorithme revient à prouver la terminaison de ses boucles non bornées.

Preuve de la terminaison d'un algorithme

- Seules les boucles non bornées (de type boucle `while`) peuvent se répéter indéfiniment. Par conséquent, prouver la terminaison d'un algorithme revient à prouver la terminaison de ses boucles non bornées.
- Pour prouver la terminaison d'une boucle non bornée on utilise la notion de variant de boucle. Il s'agit de mettre en évidence une variable qui décroît strictement à chaque passage dans la boucle.

Preuve de la terminaison d'un algorithme

- Seules les boucles non bornées (de type boucle `while`) peuvent se répéter indéfiniment. Par conséquent, prouver la terminaison d'un algorithme revient à prouver la terminaison de ses boucles non bornées.
- Pour prouver la terminaison d'une boucle non bornée on utilise la notion de variant de boucle. Il s'agit de mettre en évidence une variable qui décroît strictement à chaque passage dans la boucle. Comme, il n'existe pas de suite d'entiers naturels strictement décroissante cela prouve que la boucle termine.

Exemple

On considère la fonction ci-dessous :

```
1  def quotient(a,b):
2      '''Renvoie le quotient dans la division euclidienne de a
   ↪   par b avec a et b deux entiers naturels'''
3      q=0
4      while a-b>=0:
5          a=a-b
6          q=q+1
7      return q
```

Exemple

On considère la fonction ci-dessous :

```
1  def quotient(a,b):
2      '''Renvoie le quotient dans la division euclidienne de a
   ↪   par b avec a et b deux entiers naturels'''
3      q=0
4      while a-b>=0:
5          a=a-b
6          q=q+1
7      return q
```

En trouvant un variant de boucle, prouver la terminaison de ce programme.

Correction de l'exemple

Montrons que la variable a est un variant de boucle, c'est à dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque passage dans la boucle.

Correction de l'exemple

Montrons que la variable a est un variant de boucle, c'est à dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque passage dans la boucle.

- La valeur initiale de a est un entier naturel (précondition)

Correction de l'exemple

Montrons que la variable a est un variant de boucle, c'est à dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque passage dans la boucle.

- La valeur initiale de a est un entier naturel (précondition)
- A chaque tour de boucle la valeur de a diminue (de b> 0).

Correction de l'exemple

Montrons que la variable a est un variant de boucle, c'est à dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque passage dans la boucle.

- La valeur initiale de a est un entier naturel (précondition)
- A chaque tour de boucle la valeur de a diminue (de $b > 0$).
- La nouvelle valeur de a est a-b qui est garantie positive par condition d'entrée dans la boucle

Correction de l'exemple

Montrons que la variable a est un variant de boucle, c'est à dire qu'elle prend ses valeurs dans $\mathbb{N}$ et décroît strictement à chaque passage dans la boucle.

- La valeur initiale de a est un entier naturel (précondition)
- A chaque tour de boucle la valeur de a diminue (de $b > 0$).
- La nouvelle valeur de a est `a-b` qui est garantie positive par condition d'entrée dans la boucle

Les trois éléments ci-dessus prouvent que la variable a est un variant de la boucle `while` de ce programme, par conséquent cette boucle se termine.

Correction d'un algorithme

On dira qu'un algorithme est correct

Correction d'un algorithme

On dira qu'un algorithme est correct lorsqu'il renvoie la réponse attendue pour n'importe quel donnée en entrée.

Correction d'un algorithme

On dira qu'un algorithme est correct lorsqu'il renvoie la réponse attendue pour n'importe quel donnée en entrée.
On parle de correction partielle quand le résultat est correct lorsque l'algorithme s'arrête. Et de correction totale lorsque la correction est partielle et que l'algorithme se termine.

Tests et correction

Correction d'un algorithme

On dira qu'un algorithme est correct lorsqu'il renvoie la réponse attendue pour n'importe quel donnée en entrée.
On parle de correction partielle quand le résultat est correct lorsque l'algorithme s'arrête. Et de correction totale lorsque la correction est partielle et que l'algorithme se termine.

Tests et correction

Des tests ne permettent pas de prouver qu'un algorithme est correct.

Correction d'un algorithme

On dira qu'un algorithme est correct lorsqu'il renvoie la réponse attendue pour n'importe quel donnée en entrée.
On parle de correction partielle quand le résultat est correct lorsque l'algorithme s'arrête. Et de correction totale lorsque la correction est partielle et que l'algorithme se termine.

Tests et correction

Des tests ne permettent pas de prouver qu'un algorithme est correct. En effet, ils ne permettent de valider le comportement de l'algorithme que dans quelques cas particuliers et jamais dans le cas général

Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme on utilise la notion d'invariant de boucle. C'est une propriété du programme qui

Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme on utilise la notion d'invariant de boucle. C'est une propriété du programme qui

- est vraie à l'entrée dans la boucle.

Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme on utilise la notion d'invariant de boucle. C'est une propriété du programme qui

- est vraie à l'entrée dans la boucle.
- reste vraie à chaque itération si elle l'était à l'itération précédente.

Preuve de la correction d'un algorithme

Pour prouver la correction d'un algorithme on utilise la notion d'invariant de boucle. C'est une propriété du programme qui

- est vraie à l'entrée dans la boucle.
- reste vraie à chaque itération si elle l'était à l'itération précédente.

Trouver un invariant de boucle c'est prouver qu'un algorithme fournit la réponse attendue quelque soient les données.

Exemple

On considère la fonction ci-dessous :

```
1  def compte(elt,liste):
2      '''compte le nombre de fois où elt apparaît dans liste'''
3      compteur=0
4      for x in liste:
5          if x==elt:
6              compteur=compteur+1
7      return compteur
```

Exemple

On considère la fonction ci-dessous :

```
1  def compte(elt,liste):
2      '''compte le nombre de fois où elt apparaît dans liste'''
3      compteur=0
4      for x in liste:
5          if x==elt:
6              compteur=compteur+1
7      return compteur
```

En trouvant un invariant de boucle, montrer qu'à la sortie de la boucle, la variable compteur contient le nombre de fois où `elt` apparaît dans `liste`

Correction de l'exemple

On note $[e_1, e_2, \ldots, e_n]$ la liste et k le nombre de tours de boucle

Correction de l'exemple

On note $[e_1, e_2, \ldots, e_n]$ la liste et k le nombre de tours de boucle
Montrons que la propriété :
« `compteur` contient le nombre de de fois où `elt` apparaît dans les k premiers éléments de la liste »est un invariant de boucle.

Correction de l'exemple

On note $[e_1, e_2, \ldots, e_n]$ la liste et k le nombre de tours de boucle
Montrons que la propriété :
« `compteur` contient le nombre de de fois où `elt` apparaît dans les k premiers éléments de la liste »est un invariant de boucle.

Correction de l'exemple

On note $[e_1, e_2, \ldots, e_n]$ la liste et k le nombre de tours de boucle
Montrons que la propriété :
« `compteur` contient le nombre de de fois où `elt` apparaît dans les k premiers éléments de la liste »est un invariant de boucle.

- En entrant dans la boucle ($k = 0$) la propriété est vraie car `compteur` vaut 0.

Correction de l'exemple

On note $[e_1, e_2, \ldots, e_n]$ la liste et k le nombre de tours de boucle
Montrons que la propriété :
« `compteur` contient le nombre de de fois où `elt` apparaît dans les k premiers éléments de la liste »est un invariant de boucle.

- En entrant dans la boucle ($k = 0$) la propriété est vraie car `compteur` vaut 0.
- Supposons la propriété vraie au k-ième tour de boucle alors, au tour suivant elle reste vraie puisqu'on ajoute 1 au compteur lorsque `elt`$= e_{k+1}$

Correction de l'exemple

On note $[e_1, e_2, \ldots, e_n]$ la liste et k le nombre de tours de boucle
Montrons que la propriété :
« `compteur` contient le nombre de de fois où `elt` apparaît dans les k premiers éléments de la liste »est un invariant de boucle.

- En entrant dans la boucle ($k = 0$) la propriété est vraie car `compteur` vaut 0.
- Supposons la propriété vraie au k-ième tour de boucle alors, au tour suivant elle reste vraie puisqu'on ajoute 1 au compteur lorsque `elt`$= e_{k+1}$

Cette propriété est donc bien un invariant de boucle.

Correction de l'exemple

On note $[e_1, e_2, \ldots, e_n]$ la liste et k le nombre de tours de boucle
Montrons que la propriété :
« `compteur` contient le nombre de de fois où `elt` apparaît dans les k premiers éléments de la liste »est un invariant de boucle.

- En entrant dans la boucle ($k = 0$) la propriété est vraie car `compteur` vaut 0.
- Supposons la propriété vraie au k-ième tour de boucle alors, au tour suivant elle reste vraie puisqu'on ajoute 1 au compteur lorsque `elt`$= e_{k+1}$

Cette propriété est donc bien un invariant de boucle. L'invariant de boucle reste vraie en sortie de boucle ce qui prouve que l'algorithme est correct.