

Branchement conditionnel

-

Structures indicées immuables

Branchement conditionnel

besoin : distinguer différents cas et appliquer à chacun d'eux un traitement spécifique.

Branchement conditionnel

besoin : distinguer différents cas et appliquer à chacun d'eux un traitement spécifique.

réponse : branchement conditionnel (if)

syntaxe

```
if condition:
    instruction01
    ...
    instruction0p
else:
    instructionN1
    ...
    instructionNq
```

```
if condition:
    instruction01
    ...
    ...
    ...
    instruction0p
```

- la partie **else** et le bloc d'instructions associé disparaissent si aucun traitement quand la clause **condition** est non vérifiée.

syntaxe

```
if condition:
    instruction01
    ...
    instruction0p
else:
    instructionN1
    ...
    instructionNq
```

```
if condition:
    instruction01
    ...
    ...
    ...
    instruction0p
```

- la partie **else** et le bloc d'instructions associé disparaissent si aucun traitement quand la clause **condition** est non vérifiée.
- instruction composée : " : " obligatoire (ligne d'en-tête); indentation significative (définit le bloc à exécuter)

syntaxe

```
if condition:
    instruction01
    ...
    instruction0p
else:
    instructionN1
    ...
    instructionNq
```

```
if condition:
    instruction01
    ...
    ...
    ...
    instruction0p
```

- la partie **else** et le bloc d'instructions associé disparaissent si aucun traitement quand la clause **condition** est non vérifiée.
- instruction composée : " : " obligatoire (ligne d'en-tête); indentation significative (définit le bloc à exécuter)
- **condition** : expression **booléenne** (valeur True/False)

Exemples d'expressions booléennes

- l'entier n est pair :

Exemples d'expressions booléennes

- l'entier n est pair : $(n\%2)==0$:

Exemples d'expressions booléennes

- l'entier n est pair : $(n\%2)==0$:
- l'entier n est multiple de 3 et impair :

Exemples d'expressions booléennes

- l'entier n est pair : $(n\%2)==0$:
- l'entier n est multiple de 3 et impair :
 $(n\%3)==0$ and $(n\%2==1)$

Exemples d'expressions booléennes

- l'entier n est pair : $(n\%2)==0$:
- l'entier n est multiple de 3 et impair :
 $(n\%3)==0$ and $(n\%2==1)$
- si le booléen `trouve` est faux

Exemples d'expressions booléennes

- l'entier `n` est pair : `(n%2)==0` :
- l'entier `n` est multiple de 3 et impair :
`(n%3)==0 and (n%2==1)`
- si le booléen `trouve` est faux `if not(trouve):`

Exemples d'expressions booléennes

- l'entier n est pair : $(n\%2)==0$:
- l'entier n est multiple de 3 et impair :
 $(n\%3)==0$ and $(n\%2==1)$
- si le booléen `trouve` est faux `if not(trouve):`
- si le flottant x est compris entre -1 et 10

Exemples d'expressions booléennes

- l'entier n est pair : $(n\%2)==0$:
- l'entier n est multiple de 3 et impair :
 $(n\%3)==0$ and $(n\%2==1)$
- si le booléen `trouve` est faux `if not(trouve):`
- si le flottant x est compris entre -1 et 10
`if (x>=-1) and (x<=10):`

Exemples d'expressions booléennes

- l'entier `n` est pair : `(n%2)==0` :
- l'entier `n` est multiple de 3 et impair :
`(n%3)==0 and (n%2==1)`
- si le booléen `trouve` est faux `if not(trouve):`
- si le flottant `x` est compris entre `-1` et `10`
`if (x>=-1) and (x<=10):`
- attention évaluation paresseuse de `and` et `or`
faux et - - - : toujours faux
d'où en python, pour `and` (et `or`), évaluation de gauche à droite, si `False` (resp. `True`) à gauche, l'évaluation s'arrête et on renvoie `False` (resp. `True`)

```
>>> (20<2) and nimportequoi
False
```

Branchements imbriqués

Une des instructions à exécuter lors du traitement d'un des cas précédents, peut nécessiter elle-même un nouveau branchement conditionnel.

Situation naturelle en présence de sous-cas

Branchements imbriqués

Une des instructions à exécuter lors du traitement d'un des cas précédents, peut nécessiter elle-même un nouveau branchement conditionnel.

Situation naturelle en présence de sous-cas

Pas de difficultés particulières => **respect de la syntaxe**
les deux points, et l'indentation des lignes d'instructions vis-à-vis de la ligne d'en-tête de ce deuxième branchement (`if`).

Branchements imbriqués - exemple

On dispose d'une variable montant qui correspond à un prix à payer. Si le client dispose d'une carte de fidélité (booléen fidele), il a droit à une remise de 5% qui passe à 15% si le montant dépasse 100 euros, sinon il a une remise de 10% si son achat dépasse 120 euros. On cherche le montant final dont doit s'acquitter l'acheteur.

Branchements imbriqués - exemple

On dispose d'une variable `montant` qui correspond à un prix à payer. Si le client dispose d'une carte de fidélité (booléen `fidele`), il a droit à une remise de 5% qui passe à 15% si le montant dépasse 100 euros, sinon il a une remise de 10% si son achat dépasse 120 euros. On cherche le montant final dont doit s'acquitter l'acheteur.

```
if fidele : # il y a une carte fidélité
    if montant > 100:
        montant = montant * (1 - 0.15)
    else :
        montant = montant * (1 - 0.05)
else : # pas de carte fidélité
    if montant > 120 :
        montant = montant * (1 - 0.10)
    # pas de else ...
```

Disjonctions de cas

Disjonctions de cas multiples

- structure précédente utilisable

mais peu naturelle : certains cas arbitrairement privilégiés par rapport à d'autres.

=> à éviter

Disjonctions de cas

Disjonctions de cas multiples

- structure précédente utilisable

mais peu naturelle : certains cas arbitrairement privilégiés par rapport à d'autres.

=> à éviter

- structure plus lisible et avec même indentation des blocs de chaque cas :

```
if condition1:
    BlocInstructions1
elif condition2:
    BlocInstructions2
...
elif conditionK:
    BlocInstructionsK
...
elif conditionN:
    BlocInstructionsN
else:
    BlocInstructionsElse
```

Disjonctions de cas

- `condition1, ..., conditionN` : expressions booléennes, conditionnent l'exécution du bloc `instructionsK`

`BlocInstructionsK` = suite d'instructions (**bloc**). Un bloc peut lui-même contenir des instructions conditionnelles avec des sous-blocs.

Disjonctions de cas

- `condition1, ..., conditionN` : expressions booléennes, conditionnent l'exécution du bloc `instructionsK`

`BlocInstructionsK` = suite d'instructions (**bloc**). Un bloc peut lui-même contenir des instructions conditionnelles avec des sous-blocs.

Attention aux indentations (elles permettent de distinguer à quel bloc une instruction appartient)

Disjonctions de cas

- `condition1, ..., conditionN` : expressions booléennes, conditionnent l'exécution du bloc `instructionsK`

`BlocInstructionsK` = suite d'instructions (**bloc**). Un bloc peut lui-même contenir des instructions conditionnelles avec des sous-blocs.

Attention aux indentations (elles permettent de distinguer à quel bloc une instruction appartient)

- ordre des conditions important : si une condition `conditionK` est satisfaite, aucune des conditions suivantes n'est testée.

Disjonctions de cas

- `condition1, ..., conditionN` : expressions booléennes, conditionnent l'exécution du bloc `instructionsK`

`BlocInstructionsK` = suite d'instructions (**bloc**). Un bloc peut lui-même contenir des instructions conditionnelles avec des sous-blocs.

Attention aux indentations (elles permettent de distinguer à quel bloc une instruction appartient)

- ordre des conditions important : si une condition `conditionK` est satisfaite, aucune des conditions suivantes n'est testée.
- **elif** = contraction de **else**: **if** mais l'indentation différente et les usages des deux structures sont utiles pour des cas différents (**else**: **if** pour l'introduction de sous-cas, **elif** pour des disjonctions de cas multiples).

Types indicés immuables

chaînes de caractères et tuples : structures indicées immuables

- impossible de modifier directement un des éléments (valeur)
- éléments accessibles par un indice
=> ordre implicite
=> **en python, on part toujours de l'indice 0**
- en python 'str' (string) et 'tuple'

Création directe (str)

- contenu textuel, entre deux apostrophes, ou deux guillemets (= délimiteurs de la chaîne)

"Avec des guillemets, j'ai le droit"
'mais une citation, " " est préférable'
-> échapper un caractère : 'j\'ai le droit'

Création directe (str)

- contenu textuel, entre deux apostrophes, ou deux guillemets (= délimiteurs de la chaîne)

"Avec des guillemets, j'ai le droit"

'mais une citation, " " est préférable'

-> échapper un caractère : 'j\'ai le droit'

- `print(chaine)` = affiche la chaîne de caractères à l'écran et passe à la ligne

```
texte="j'adore l'informatique"
```

```
print(texte)
```

```
print('voilà ce que je veux écrire :')
```

Création directe (str)

- contenu textuel, entre deux apostrophes, ou deux guillemets (= délimiteurs de la chaîne)

"Avec des guillemets, j'ai le droit"

'mais une citation, " " est préférable'

-> échapper un caractère : 'j\'ai le droit'

- `print(chaine)` = affiche la chaîne de caractères à l'écran et passe à la ligne

```
texte="j'adore l'informatique"
```

```
print(texte)
```

```
print('voilà ce que je veux écrire :')
```

- `len(texte)` = longueur de texte (str)

Création directe (tuple)

- **tuple** = concept mathématique de n -uplet
contrairement à la structure de liste, un tuple est immuable
- écrire ses éléments (pas forcément tous du même type) **dans l'ordre** en les **séparant par des virgules** et en entourant le tuple par des parenthèses.

```
>>> a=('Je','suis',True,45)
>>> type(a)
<class 'tuple'>
```

Attention : tuple à un seul élément => ne pas oublier la virgule séparatrice indiquant qu'il s'agit d'un tuple

```
>>> type( (45) )
<class 'int'>
>>> type( (45,))
<class 'tuple'>
```

Concaténation

- T1+T2 fabrique le nouveau tuple/chaîne de caractères (si T1 et T2 sont des tuples/chaînes de caractères) obtenu par concaténation de T1 suivi de T2.

```
>>> deb='Bien'  
>>> mot=deb+"venue"  
>>> mot  
'Bienvenue'
```

- mettre le résultat dans une variable pour le réutiliser.

Concaténation

- $n * T1$ fabrique le tuple/chaîne de caractères obtenu en concaténant $T1$ avec lui-même un nombre entier $n > 0$ de fois.

```
>>> 4*deb
```

```
'BienBienBienBien'
```

```
>>> t3=(2.1,True)
```

```
>>> 3*t3
```

```
(2.1, True, 2.1, True, 2.1, True)
```

Accès à un élément

- éléments d'un tuple/chaîne de caractères **numérotés à partir de l'indice 0**

premier élément : indice 0,

second élément : indice 1,

dernier élément : indice $n - 1$ avec n longueur du tuple/chaîne de caractères.

Accès à un élément

- éléments d'un tuple/chaîne de caractères **numérotés à partir de l'indice 0**

premier élément : indice 0,

second élément : indice 1,

dernier élément : indice $n - 1$ avec n longueur du tuple/chaîne de caractères.

- accès à la valeur de l'élément d'indice p du tuple ou chaîne de caractères contenu dans la variable T

$T[p]$

-> Attention crochets indispensables

-> valeur de p doit être valide (entre 0 inclus et la longueur exclue.)

Parcours par for

T variable contenant un type énumérable
for elmt un T : = parcours de T dans l'ordre

```
for lettre in 'informatique':  
    print(lettre)
```

```
T=(4,5,7)
```

```
S=0
```

```
for x in T:  
    S=S+x
```

Notion de tranche

- fabriquer un nouveau tuple/chaîne de caractères par une tranche d'un tuple/chaîne de caractères (T), préexistant.

T[debut:fin]

-> tuple/chaîne de caractères constitué des éléments/caractères du tuple/chaîne initial entre les indices deb **inclus** et fin **exclu**.

-> attention : deb entre 0 et la longueur moins 1,
et fin entre 1 et la longueur.

Si deb est supérieur ou égal à fin : tuple/chaîne vide.

```
>>> mot='informatique'  
>>> mot[0:4]  
'info'
```

Dépaquetage d'un tuple

tuples (avec peu d'éléments ...)

-> récupérer les éléments par dépaquetage au lieu des indices

-> autant de variables que d'éléments dans le tuple et principe d'affectation

```
>>> uplet = (45,True,'info')
```

```
>>> a,b,c= uplet
```

```
>>> (e,f,g) = uplet
```

```
>>> print(c,' puis',f)
```

```
info  puis True
```

```
>>> a,b=uplet
```

```
Traceback (most recent call last):
```

```
  File "<console>", line 1, in <module>
```

```
ValueError: too many values to unpack (expected 2)
```

nombre de variables pour accueillir les éléments du tuple incorrect

=> une erreur détectée (avec message explicite)