Conditions

13.1

Booléens

13.1.1 Booléen voir Type simple 4.3

13.1.2 Vocabulaire voir Type simple 6.2.3

Expression - Instruction - Affectation

13.1.3 Créer un booléen avec bool() voir Type simple 4.4.5

Un contenu VIDE/NUL est évalué à False si on veut un booléen. Sinon, le résultat est False.

13.2

Instruction conditionnelle SI, SINON SI, SINON

13.2.1 Instruction conditionnelle voir Intro 1.4.4

13.2.2 Structure générale

Quelque soit le langage, on retrouve toujours une structure qui ressemble à ceci :

Un contenu VIDE/NUL est évalué à False si on veut un booléen. Sinon, le résultat est False.

```
1  Instructions Avant 1
2  Instructions Avant 2
3
4  # BLOC A (un seul, obligatoire)
5  SI une expr. booléenne A est vraie
6      Instruction A1
7      Instruction A2
8
9  # BLOC B (0 ou + : optionnel)
10 SINON SI une expr. bool. B est vraie
11     Instruction B1
12     Instruction B2
13
14 # BLOC C (0 ou 1 : optionnel unique)
15 SINON (dans tous les autres cas)
16     Instruction C1
17     Instruction C2
18
19 Instructions Après 1
20 Instructions Après 2
```

Nombre de blocs

- Un unique bloc SI et il doit toujours être en première position
- 0, 1, 2 ou autant de blocs SINON SI que vous voulez
- 0 ou 1 bloc SINON et il doit toujours être en dernière position : c'est le bloc par défaut. Si aucun autre n'a été activé, on prend celui-ci.

13.2.3 Un seul bloc exécuté

A SAVOIR IMPERATIVEMENT L'interpréteur Python n'exécute que **l'un des blocs** A, B ou C d'une structure if-elif-else.

Il exécute les instructions du bloc de la première expression évaluée à True puis quitte la structure **même si les autres expressions sont vraies également**.

Un exemple de déroulement :

```
1  Instructions Avant 1
2  Instructions Avant 2
3
4  # BLOC A (un seul, obligatoire)
5  if note > 11 :
6      Instruction A1
7      Instruction A2
8      Instruction A3
9
10 # BLOC B (0 ou + : optionnel)
11 elif note < 9 :
12     Instruction B1
13     Instruction B2
14     Instruction B3
15
16 # BLOC C (0 ou 1 : optionnel unique)
17 else :
18     Instruction C1
19     Instruction C2
20
21 Instructions Après 1
22 Instructions Après 2
```

Exemple avec note valant 15 .

C'est le bloc A qui va s'activer car note > 11 est évaluée à True.

Il exécute L1-L2-L5-L6-L7-L8-L21-L22.

Exemple avec note valant 5 .

C'est le bloc B qui va s'activer car le bloc A ne sera pas activé et note < 9 est évaluée à True.

Il exécute L1-L2-L5-L11-L12-L13-L14-L21-L22.

Exemple avec note valant 9 .

C'est le bloc C qui va s'activer car le bloc A ne sera pas activé, le bloc B non plus (9 n'est pas strictement inférieur à 9) et le else ne possède jamais de condition : si on y arrive, on le déclenche toujours.

Il exécute L1-L2-L5–L11-L17-L8-L19-L21-L22.

13.2.4 Instruction conditionnelle et fonction

Version propre avec une seule "porte de sortie"

— AVANTAGE : plus facile de surveiller la correction de la fonction (le fait que la fonction réponde toujours correctement)

— DESAVANTAGE : il faut stocker la réponse dans une variable et lire ensuite séquentiellement la fonction jusqu'à trouver l'unique sortie.

```
def appreciation(note :int) -> str :
    if note > 13 :
        app = "Bien"
    elif note < 9 :
        app = "Insuffisant"
    else :
        app = "Moyen"
    return app
```

Version avec plusieurs "portes de sortie"

— AVANTAGE : on n'a pas besoin en tant que lecteur humain de continuer à lire le reste du code lorsqu'on rencontre une sortie.

— DESAVANTAGE : moins facile à surveiller (plusieurs portes de sortie à contrôler).

```
def appreciation(note :int) -> str :
    if note > 13 :
        return "Bien"
    elif note < 9 :
        return "Insuffisant"
    else
        return "Moyen"
```

```
def appreciation(note :int) -> str :
    if note > 13 :
        return "Bien"
    elif note < 9 :
        return "Insuffisant"
    return "Moyen"
```

Exemple de vérification avant envoi

Une seule sortie : une seule vérification.

```
def beaucoup_trop_gentil(note :int) -> int :
    if note < 10 :
        note = note + 5
    else :
        note = note + 3
    # Vérification avant envoi
    if note > 20 :
        note = 20
    return note
```

Deux sorties : deux vérifications...

```
def beaucoup_trop_gentil(note :int) -> int :
    if note < 10 :
        note = note + 5
        # Vérification avant envoi
        if note > 20 :
            note = 20
        return note
    else :
        note = note + 3
        # Vérification avant envoi
        if note > 20 :
            note = 20
        return note
```

13.3

Prédicat : fonction renvoyant un booléen

13.3.1 Opérateurs == et =

Attention à ne pas confondre

— l'opérateur d'égalité de contenu **==**

— l'opérateur d'affectation **=**

Avec **a == b**, l'interpréteur comprend qu'il doit tester l'égalité des variables **a** et **b**. Il va donc évaluer cette expression et obtenir vrai si les deux contenus sont les mêmes.

Avec **a = b**, l'interpréteur comprend qu'il faut stocker **b** dans la variable **a**. On génère donc un alias.

13.3.2 FONCTION : Prédicat

Forme typique de prédicat

On peut nommer de cette façon une fonction qui renvoie toujours un booléen : elle répondra soit **True** soit **False**.

On parle de prédicat puisque la fonction prédit une propriété à partir des paramètres reçus.

Ce cas de figure est très courant en informatique : créer un prédicat ne requiert donc forcément pas d'utiliser

un if : il suffit de fournir l'expression qu'on veut évaluer derrière votre return.

Exemple avec une fonction qui évalue si une note de 10 permet de valider un semestre :

```
def est_valide(note :int) -> bool :
    """Renvoie True si la note >= 10"""
    return note >= 10
```

La version à éviter tant que c'est possible

La version ci-dessous est valide également mais est à éviter si possible : elle revient à dire "si c'est vrai, dis que c'est vrai, sinon, dis que c'est faux". Un peu lourd. Et l'oubli du else est très courant : on finit par avoir une fonction qui dit bien quand c'est vrai mais qui ne répond rien quand c'est faux.

```
def est_valide(note :int) -> bool :
    """Renvoie True si la note >= 10"""
    if note >= 10 :
        return True
    else :
        return False
```

13.4

If sans condition booléenne ?

13.4.1 Condition non booléenne

Cas d'une expression booléenne

Lorsque l'interpréteur Python lit un **`if`** ou un **`elif`**, il s'attend à trouver une expression booléenne derrière.

Exemple : **`if a >= 0:`**

Ici, l'expression est bien une expression booléenne.

Cas d'une expression non booléenne

Il peut arriver par contre que l'expression fournie ne soit pas une expression booléenne.

Exemple : **`if a:`**

Python ne va alors pas s'embêter : il va chercher à l'interpréter comme un booléen, comme si vous aviez tapé ceci :

Exemple : **`if bool(a):`**

Ainsi :

Si **`a`** est nul, vide ("" ou [] ou () ou) ou None, **`a`** sera équivalent à un **`False`**.

Dans tous les autres cas, **`a`** sera équivalent à **`True`**.

13.4.2 Cas du tableau vide

On peut agir en fonction de l'état VIDE ou NON VIDE d'un type construit en utilisant une simple instruction conditionnelle.

Voici comment agir de façon différentes**i le tableau est NON VIDE** et sinon s'il est VIDE :

```
if tab :  # si tab est NON VIDE
    actions_si_le_tableau_n_est_pas_vide

else :  # sinon, c'est que tab est VIDE
    actions_si_le_tableau_est_vide
```

Le mot-clé **`not`** permet d'inverser la valeur booléenne : vrai devient faux, et faux devient vrai. On peut alors demander ainsi d'agir **`si le tableau est VIDE`**, et sinon s'il est NON VIDE :

```
if not tab :  # tab est vide
    actions_si_le_tableau_est_vide

else :  # sinon, c'est que tab est vide
    actions_si_le_tableau_est_vide
```

Dans le dernier cas, notez que si **`tab`** est évalué à **`False`**, alors **`not tab`** sera évaluée à True.