IFT-1004 - Introduction à la programmation

Module 2 : Expressions et séquencement

Honoré Hounwanou, ing.

Département d'informatique et de génie logiciel
Université Laval

Table des matières

Les variables en Python

Objectif

Savoir déclarer des variables qui soient représentatives des données qu'elles symbolisent, tout en respectant les règles de syntaxe inhérentes.

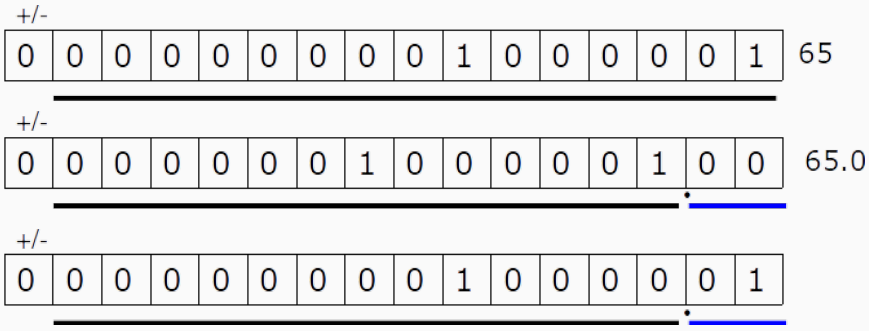
Une variable en Python

- Représente une donnée ;
- Porte un nom, idéalement significatif (qui désigne le contenu d'une case mémoire à une adresse donnée);
- Est associée à une case de la mémoire (statique);
- A donc une adresse mémoire (statique);
- Est du type de la dernière donnée qu'on lui a affectée (dynamique);
- Ce type est nécessaire pour **interpréter** le contenu de la variable.

Nom	Adresse	Contenu	Type
...	1000	...	...
...	1001	...	...
age_retraite	1002	65	entier
age_embauche	1003	60.0	réel
	1004		

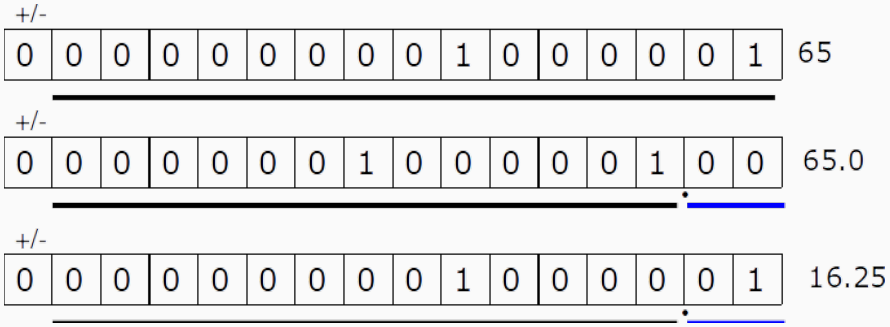
Interprétation du contenu d'une variable

Le type d'une variable sert à son interprétation (ex. 16 bits)



Interprétation du contenu d'une variable

Le type d'une variable sert à son interprétation (ex. 16 bits)



Un nom de variable en Python

- Séquence de lettres (a-z, A-Z) et de chiffres (0-9), qui doit toujours débuter par une lettre ;
- Seul le caractère spécial _ (souligné) est permis ;
- Nous éviterons les noms débutant ou se terminant par un _ (ceux-ci ont une signification particulière en Python, nous y reviendrons) ;
- Les majuscules et minuscules représentent différentes lettres ;
- Il faut éviter les 33 mots réservés qui ont une signification prédéfinie ;
- Par convention, les variables en Python utilisent le format suivant : `ma_variable`.

Les mots réservés en Python

Les mots suivants sont réservés par le langage, et ne peuvent donc pas être utilisés comme nom de variable.

`False`	`class`	`finally`	`is`	`return`
`None`	`continue`	`for`	`lambda`	`try`
`True`	`def`	`from`	`nonlocal`	`while`
`and`	`del`	`global`	`not`	`with`
`as`	`elif`	`if`	`or`	`yield`
`assert`	`else`	`import`	`pass`	
`break`	`except`	`in`	`raise`	

Les types de données de base et leurs opérateurs

Les types de données de base et leurs opérateurs

Objectif

Savoir comment bien utiliser les données typées dans des calculs divers afin de garantir la validité du résultat.

Les types numériques : entier ou réel

- Entier : `int`, sans partie décimale
- Réel : `float`, avec partie décimale
- La représentation interne varie, l'interprétation (l'utilisation) aussi
- Opérateurs arithmétiques : **+**, **-**, *****, **/**, **//**, **%**, ******
- Fonctions mathématiques : `abs(arg)`, `int(arg)`, `float(arg)`, etc.

Nom	Adresse	Contenu	Type
age_retraite	1002	**65**	int
age_embauche	1003	**60.0**	float
...	1004	...	...

Les opérateurs arithmétiques

Opérateur	Exemple	x=9 et y=4
+	x + y	13
-	x - y	5
*	x * y	36
/	x / y	2.25
//	x // y	2
%	x % y	1
**	x ** y	6561

Conversion de données

- `age_retraite - age_embauche ?`
- `age_retraite / 10 ?`
- `age_retraite / 10.0 ?`
- `age_retraite // 10 ?`
- `age_retraite // 10.0 ?`
- `age_retraite // float(10) ?`

Nom	Adresse	Contenu	Type
age_retraite	1002	**65**	int
age_embauche	1003	**60.0**	float
...	1004	...	...

Attention : Le résultat aurait été différent sous Python 2 !

Priorité d'évaluation

1. Les parenthèses d'abord : `( )`
2. Les fonctions : `abs(arg)`, etc.
3. L'exposant : `**`
4. Les opérateurs unaires : `-`, `+`
5. Les opérateurs binaires : `*`, `/`, `//`, `%`
6. Les opérateurs binaires : `+`, `-`

Quelle est la valeur de `x` après l'exécution de l'expression suivante ?

```
x = -8 + 20 + 3 * 12 // (4 + 2) * 5 - abs(-3) + 5**2 % 4
```

Le type Booléen : `True` ou `False`

- Opérateurs de comparaison : <, >, <=, >=, ==, !=
- Moins prioritaires que les opérateurs arithmétiques
- `False` correspond à la valeur 0
- `True` correspond à toute valeur différente de 0, généralement 1
- `erreur_emb = age_embauche > age_retraite`

Nom	Adresse	Contenu	Type
age_retraite	1002	**65**	int
age_embauche	1003	**60.0**	float
...	1004	...	...

Le type Booléen : `True` ou `False`

- Opérateurs de comparaison : <, >, <=, >=, ==, !=
- Moins prioritaires que les opérateurs arithmétiques
- `False` correspond à la valeur 0
- `True` correspond à toute valeur différente de 0, généralement 1
- `erreur_emb = age_embauche > age_retraite`

Nom	Adresse	Contenu	Type
age_retraite	1002	**65**	int
age_embauche	1003	**60.0**	float
erreur_emb	1004	...	...

Le type Booléen : `True` ou `False`

- Opérateurs de comparaison : <, >, <=, >=, ==, !=
- Moins prioritaires que les opérateurs arithmétiques
- `False` correspond à la valeur 0
- `True` correspond à toute valeur différente de 0, généralement 1
- `erreur_emb = age_embauche > age_retraite`

Nom	Adresse	Contenu	Type
age_retraite	1002	**65**	int
age_embauche	1003	**60.0**	float
erreur_emb	1004	False	...

Le type Booléen : `True` ou `False`

- Opérateurs de comparaison : <, >, <=, >=, ==, !=
- Moins prioritaires que les opérateurs arithmétiques
- `False` correspond à la valeur 0
- `True` correspond à toute valeur différente de 0, généralement 1
- `erreur_emb = age_embauche > age_retraite`

Nom	Adresse	Contenu	Type
age_retraite	1002	**65**	int
age_embauche	1003	**60.0**	float
erreur_emb	1004	False	bool

Les opérateurs de comparaison

Opérateur	Exemple	x=9 et y=4
==	x == y	False
!=	x != y	True
>	x > y	True
>=	x >= y	True
<	x < y	False
<=	x <= y	False

Les opérateurs logiques

Opérateur	Exemple	cond1 = True cond2 = False
not	**not** cond1	False
not	**not** cond2	True
and	cond1 **and** cond2	False
or	cond1 **or** cond2	True

Table de vérité : `not`

Opérateur	Valeur	Résultat
not	True	False
not	False	True

Table de vérité : and

Valeur 1	Opérateur	Valeur 2	Résultat
True	and	True	True
True	and	False	False
False	and	True	False
False	and	False	False

Table de vérité : or

Valeur1	Opérateur	Valeur2	Résultat
True	or	True	True
True	or	False	True
False	or	True	True
False	or	False	False

Priorité d'évaluation

1. Les parenthèses : `()`
2. Les fonctions : `abs(arg)`, etc.
3. L'exposant : `**`
4. Les opérateurs *unaires* : `-`, `+`
5. Les opérateurs *binaires* : `*`, `/`, `//`, `%`
6. Les opérateurs binaires : `+`, `-`
7. Les opérateurs de comparaison : `<`, `<=`, `>`, `>=`
8. Les opérateurs de comparaison : `==`, `!=`
9. L'opérateur logique : `not`
10. L'opérateur logique : `and`
11. L'opérateur logique : `or`

Coffre à outils

Question : A-t-on tous les opérateurs requis pour effectuer :

1. Des calculs mathématiques simples ?
2. Des calculs mathématiques complexes ?
3. L'évaluation de conditions d'exécution de séquences d'instructions ?

Le séquencement d’instructions

Le séquencement d'instructions

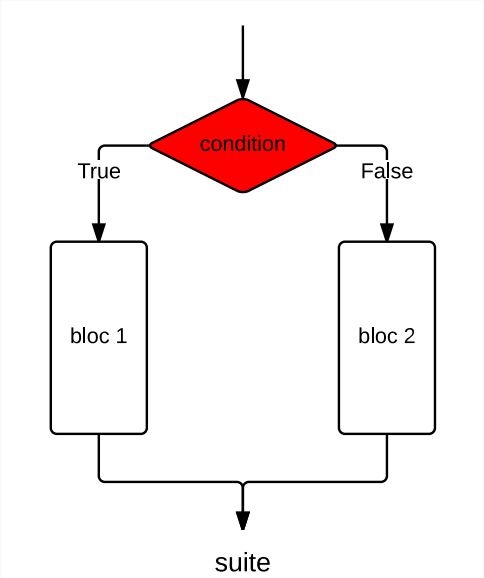
Objectif

Pouvoir élaborer un bloc d'instructions implémentant un algorithme

Les instructions conditionnelles

Structure *si - sinon*

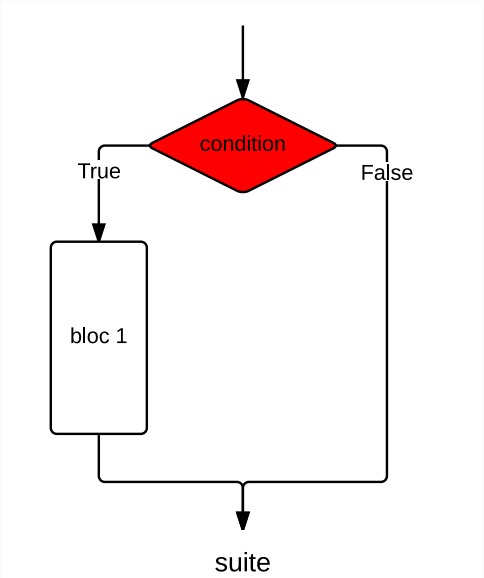
- Si (une **condition** est vraie) alors
  - On exécute le bloc 1
- Sinon
  - On exécute le bloc 2
- Suite



Les instructions conditionnelles

Structure *si*

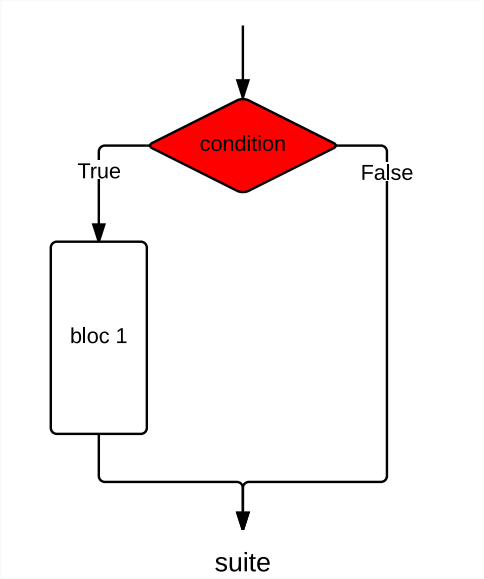
- Si (une **condition** est vraie) alors
  - On exécute le bloc 1
- Sinon
  - Ne rien faire dans ce cas
- Suite



Les instructions conditionnelles

Structure *si*

- Si (une **condition** est vraie) alors
  - On exécute le bloc 1
- Suite



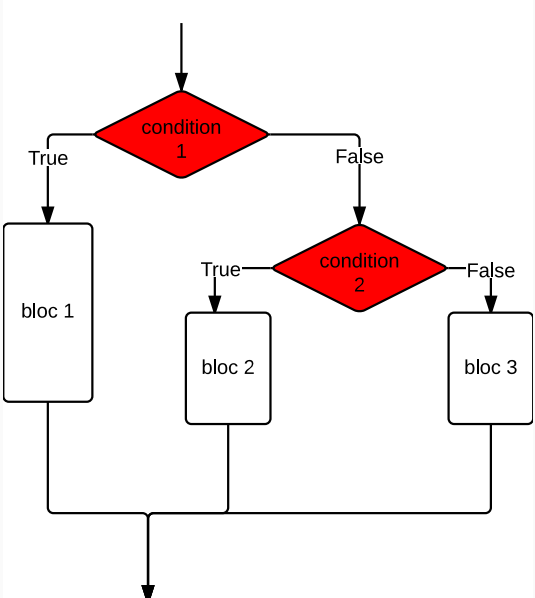
Les instructions conditionnelles

```
age_embauche = int(input("Âge à l'embauche : "))
age_retraite = int(input("Âge à la retraite : "))
erreur_emb = age_retraite < age_embauche

if not erreur_emb:
    anciennete = age_retraite - age_embauche
    print("années de travail : ", anciennete)
else:
    nb_an_sans_cotiser = age_embauche - age_retraite
    print("nb d'années à racheter : ", nb_an_sans_cotiser)
```

Structure *si-sinon* imbriquées

- Si `condition1` alors
  - On exécute le bloc 1
- Sinon
  - Si `condition2` alors
    - On exécute le bloc 2
  - Sinon
    - On exécute le bloc 3
- Suite

Les instructions conditionnelles



Les instructions conditionnelles

```
if anciennete >= 30:
    cadeau = "voyage"
elif anciennete >= 25:
    cadeau = "bague"
elif anciennete >= 20:
    cadeau = "gâteau"
elif anciennete >= 15:
    cadeau = "ballons"
else:
    cadeau = "merci"

print(cadeau)
```

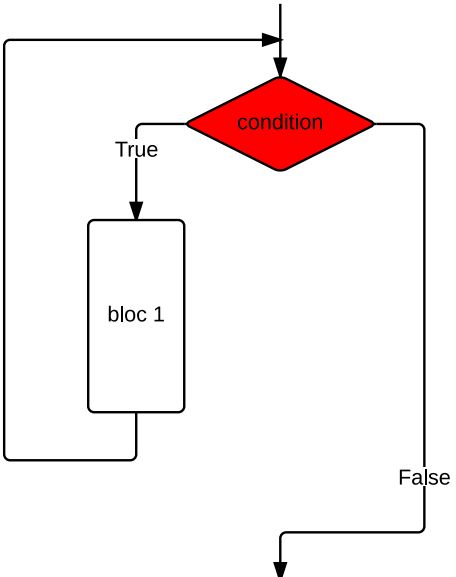
Les boucles

Nous distinguons deux types de boucles : les boucles `while` et les boucles `for`

- Boucle `while` : on exécute un bloc d'instructions tant que la condition est vraie
  - On ne peut pas (facilement) prédire le nombre d'exécutions du bloc d'instructions, car cela dépend d'une condition à évaluer
- Boucle `for` : on exécute un bloc d'instructions un certain nombre de fois
  - On peut prédire (généralement par calcul) le nombre d'exécutions du bloc d'instructions
  - On exécute systématiquement le bloc d'instructions ce nombre de fois
  - Utilisée pour **parcourir** les éléments d'une structure *itérable*.

La boucle `while`

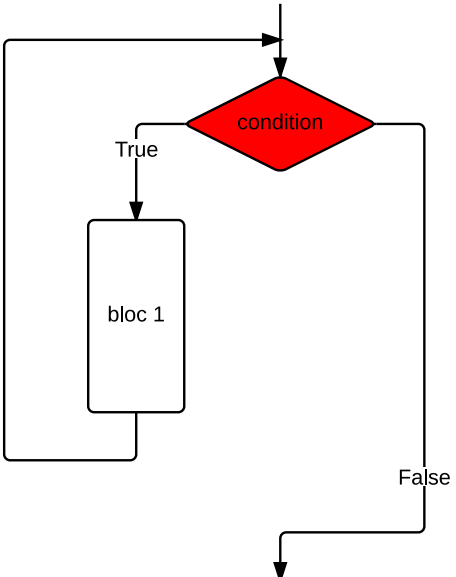
- Tant que `condition` :
  - On exécute le bloc 1
- Suite



La boucle `while`

- Tant que `condition` :
  - On exécute le bloc 1
- Suite

Qu'arrive-t-il si la condition est toujours vraie ?



Exemple : calculer x^y

$$x^y = x \cdot x \cdot x \cdot \ldots \cdot x \qquad (y - 1 \text{ multiplications})$$

$$x^4 = x \cdot x \cdot x \cdot x$$

$$x^3 = x \cdot x \cdot x$$

$$x^2 = x \cdot x$$

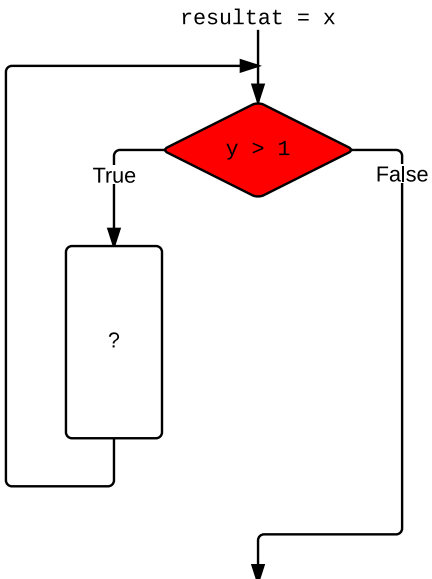
$$x^1 = x$$

$$x^0 = 1$$

Exemple : calculer x^y

$$x^y = x \cdot x \cdot x \cdot \ldots \cdot x \quad (y - 1 \text{ fois})$$

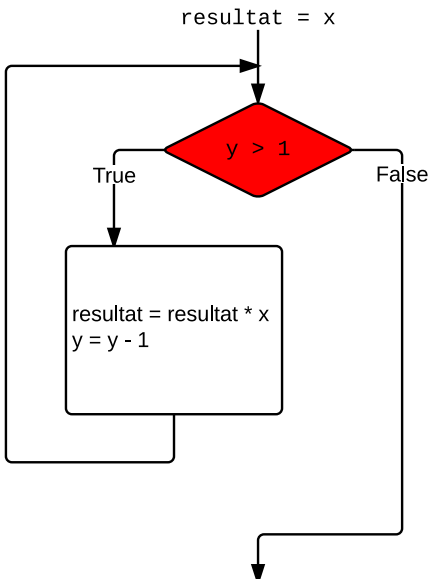
```
resultat = x

...

print(resultat)
```



Exemple : calculer x^y

$$x^y = x \cdot x \cdot x \cdot \ldots \cdot x \quad (y - 1 \text{ fois})$$

```
resultat = x

while y > 1:
    resultat = resultat * x
    y = y - 1

print(resultat)
```

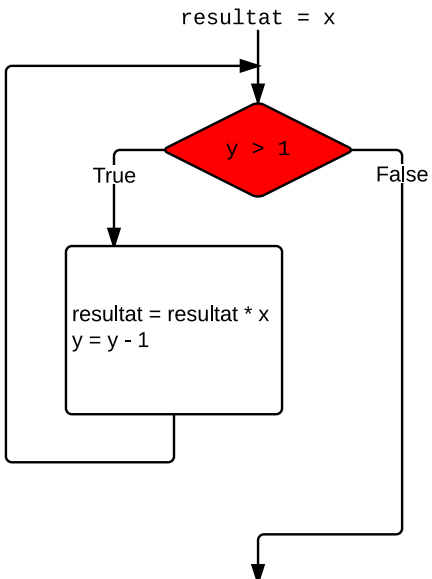


Exemple : calculer x^y

$$x^y = x \cdot x \cdot x \cdot \ldots \cdot x \quad (y - 1 \text{ fois})$$

```
resultat = x

while y > 1:
    resultat = resultat * x
    y = y - 1

print(resultat)
```

Est-ce que cette boucle termine dans tous les cas ?

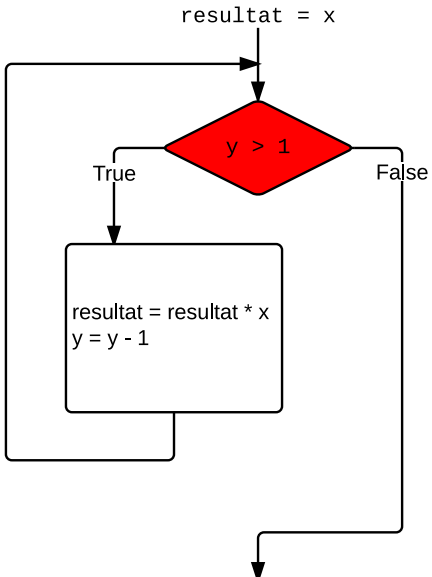


Exemple : calculer x^y

$$x^y = x \cdot x \cdot x \cdot \ldots \cdot x \quad (y - 1 \text{ fois})$$

```
resultat = x

while y > 1:
    resultat = resultat * x
    y = y - 1

print(resultat)
```

Est-ce que la boucle produit toujours le bon résultat ?

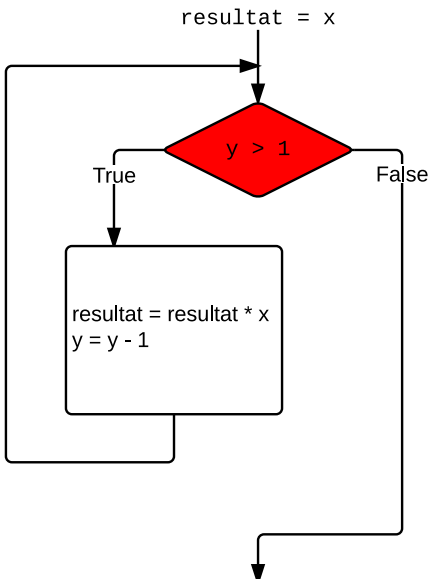


Exemple : calculer x^y

$$x^y = x \cdot x \cdot x \cdot \ldots \cdot x \quad (y - 1 \text{ fois})$$

```
resultat = x

while y > 1:
    resultat = resultat * x
    y = y - 1

print(resultat)
```

Et si y est négatif ou nul ?

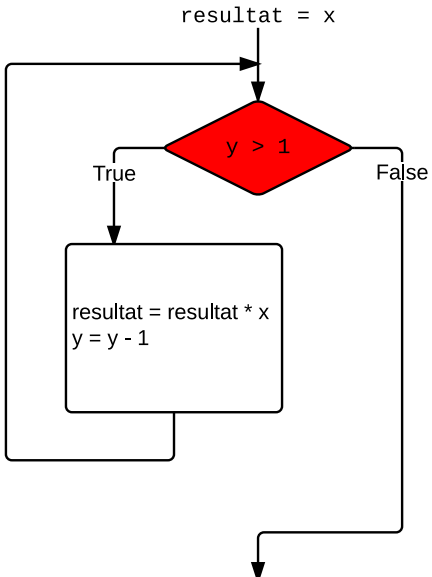


Exemple : calculer x^y

$$x^y = x \cdot x \cdot x \cdot \ldots \cdot x \quad (y - 1 \text{ fois})$$

```
resultat = x

while y > 1:
    resultat = resultat * x
    y = y - 1

print(resultat)
```

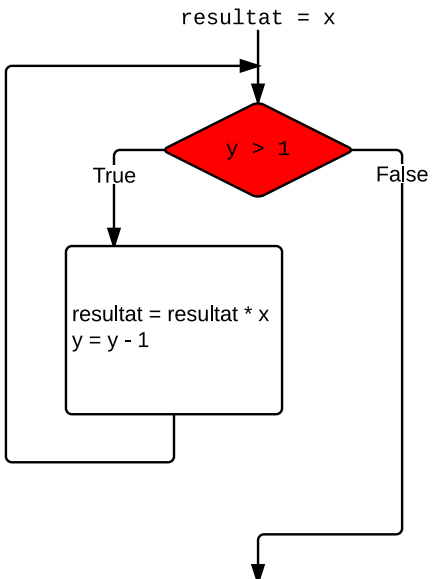
Comment modifier le code pour y inclure la situation où `y = 0` ?



Exemple : calculer x^y

$$x^y = x \cdot x \cdot x \cdot \ldots \cdot x \quad (y - 1 \text{ fois})$$

```
# On suppose y >= 0

if y == 0:
  resultat = 1

else:
  resultat = x
  while y > 1:
    resultat = resultat * x
    y = y - 1

print(resultat)
```



Boucles `while` imbriquées

Exemple d'utilisation : calculer une table de multiplication `[1 à x]` par `[1 à y]`, où x et y sont deux paramètres tels que `x >=1` et `y >= 1`.

1	1	2	3	4	5	...	y
1	1	2	3	4	5	...	y
2	2	4	6	8	10	...	$2 \cdot y$
3	3	6	9	12	15	...	$3 \cdot y$
...	...	...	...	...	...	...	...
x	x	$x \cdot 2$	$x \cdot 3$	$x \cdot 4$	$x \cdot 5$	...	$x \cdot y$

Boucles `while` imbriquées

```
# On suppose que x >= 1 et y >= 1

multiplicateur = 1
while multiplicateur <= x:  # création d'une ligne
    print("Les multiples de ", multiplicateur, "sont : ", end="")

    ...

    multiplicateur = multiplicateur + 1  # on passe à la
                                         # ligne suivante
                                         # du tableau
```

Boucles `while` imbriquées

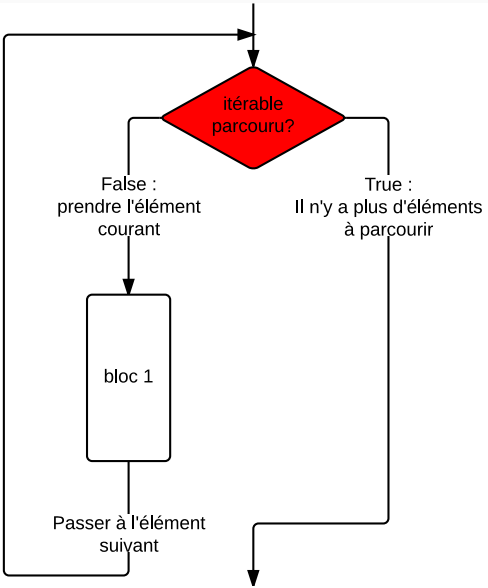
```
# On suppose que x >= 1 et x >= 1

multiplicateur = 1
while multiplicateur <= x:  # création d'une ligne
    print("Les multiples de ", multiplicateur, "sont : ", end="")
    multiplicande = 1

    while multiplicande <= y:
        print(multiplicateur * multiplicande, end=" ")
        multiplicande = multiplicande + 1

    print("")
    multiplicateur = multiplicateur + 1  # on passe à la
                                         # ligne suivante
                                         # du tableau
```

La boucle `for`

- Pour cible dans itérable :
  - On exécute le bloc 1
- Suite

Une *liste* est un objet itérable

```
for nb in [1, 2, 3]:
    print(nb)
print("terminé")
```



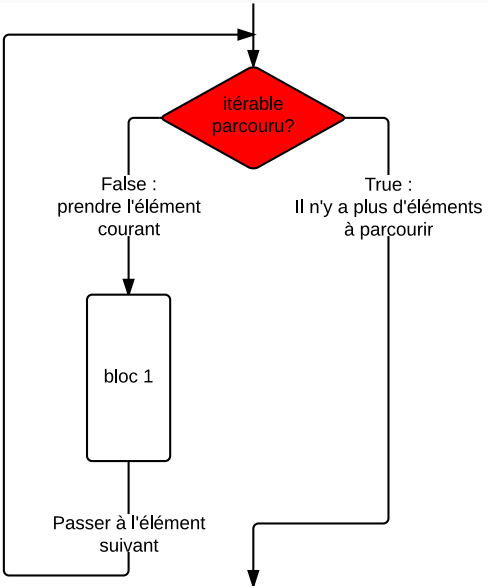
La fonction range(start, stop, step)

Par défaut, `start=0` et `step=1`

```
intervalle = list(range(5))
intervalle = list(range(-1, 5))

intervalle = list(range(5, -1, -1))
```

```
for nb in intervalle:
    print(nb)
print("terminé")
```

On y reviendra !



En résumé

- Les expressions, les structures sélectives et itératives permettent :
  - D'effectuer des calculs et des tests sur les données ou le résultat des calculs
  - De choisir des blocs d'instructions à exécuter conditionnellement à l'évaluation de ces tests
  - De répéter des instructions à exécuter un certain nombre de fois
- Est-ce que ces instructions, ensemble, offrent les mêmes capacités de traitement que le cerveau humain ?

Lectures, travaux et exercices

Plan des activités du module 2

- Documents à lire
  - Chapitres 3 et 4 de G. Swinnen
- Travaux dirigés
  - Entrées, sorties, types de données

Questions ?