

Cours 2 : Variables et fonctions



Ing-info — Ingénierie informatique

Christophe Gonzales

- 1 Les types des variables
- 2 Les fonctions
- 3 Les structures de données

Types des variables

Rappel : en python, toute variable a un type

Types des variables

Rappel : en python, toute variable a un type

- ▶ Type défini automatiquement par la valeur affectée à la variable

Rappel : en python, toute variable a un type

- ▶ Type défini automatiquement par la valeur affectée à la variable
- ▶ Définir des variables de différents types :

code python	type	type python
<code>a = "toto"</code>	chaîne de caractères	<code>str</code>
<code>a = 4</code>	nombre entier	<code>int</code>
<code>a = 3.4</code>	nombre réel	<code>float</code>
<code>a = True</code>	booléen	<code>bool</code>

Types des variables

Rappel : en python, toute variable a un type

- ▶ Type défini automatiquement par la valeur affectée à la variable
- ▶ Définir des variables de différents types :

code python	type	type python
<code>a = "toto"</code>	chaîne de caractères	<code>str</code>
<code>a = 4</code>	nombre entier	<code>int</code>
<code>a = 3.4</code>	nombre réel	<code>float</code>
<code>a = True</code>	booléen	<code>bool</code>

- ▶ Connaître le type d'une variable :

```
a = 3.4
type(a)
```

```
float
```

Types des variables

Rappel : en python, toute variable a un type

- ▶ Type défini automatiquement par la valeur affectée à la variable
- ▶ Définir des variables de différents types :

code python	type	type python
<code>a = "toto"</code>	chaîne de caractères	<code>str</code>
<code>a = 4</code>	nombre entier	<code>int</code>
<code>a = 3.4</code>	nombre réel	<code>float</code>
<code>a = True</code>	booléen	<code>bool</code>

- ▶ Connaître le type d'une variable :

```
a = 3.4
type(a)
```

```
float
```



Il existe d'autres types : complexes, listes, dictionnaires, *etc.*

Les nombres (int et float)

► Opérateurs arithmétiques :

Opérateur	Signification	Particularité
$x + y$	addition	
$x - y$	soustraction	
$x * y$	multiplication	
x / y	division	résultat toujours en float
$x \% y$	reste de la division	
$x ** y$	x puissance y	
$x // y$	division entière par défaut	$9 // 2 = 4$, $-11.0 // 3 = -4.0$

Les nombres (int et float)

► Opérateurs arithmétiques :

Opérateur	Signification	Particularité
$x + y$	addition	
$x - y$	soustraction	
$x * y$	multiplication	
x / y	division	résultat toujours en float
$x \% y$	reste de la division	
$x ** y$	x puissance y	
$x // y$	division entière par défaut	$9 // 2 = 4$, $-11.0 // 3 = -4.0$

► Types des valeurs obtenues :

`int op int \implies int`

`int op float OU float op int \implies float`

`float op float \implies float`

Les nombres (int et float)

► Opérateurs arithmétiques :

Opérateur	Signification	Particularité
$x + y$	addition	
$x - y$	soustraction	
$x * y$	multiplication	
x / y	division	résultat toujours en float
$x \% y$	reste de la division	
$x ** y$	x puissance y	
$x // y$	division entière par défaut	$9 // 2 = 4$, $-11.0 // 3 = -4.0$

► Types des valeurs obtenues :

`int op int` \implies `int`

`int op float` OU `float op int` \implies `float`

`float op float` \implies `float`

► Opérateurs d'affectations :

► `x = y` : affectation déjà vue

► `x += y` \iff `x = x + y` (idem avec `-`, `*`, `/`, `%`, `**`, `//`)

Exemples de manipulation d'int et de float

▶ Programme Python :

```
1 a = 33 + 4
```

▶ Effets des instructions :

Exemples de manipulation d'int et de float

▶ Programme Python :

```
1 a = 33 + 4
```

▶ Effets des instructions :

- 1 Création de l'int `a` et affectation de la valeur 37

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
```

► Effets des instructions :

- 1 Création de l'int a et affectation de la valeur 37

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4  
2 b = 1.0
```

► Effets des instructions :

- 1 Création de l'int a et affectation de la valeur 37
- 2 Création du float b et affectation de la valeur 1.0

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
3 a += b
```

► Effets des instructions :

- 1 Création de l'int a et affectation de la valeur 37
- 2 Création du float b et affectation de la valeur 1.0

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
3 a += b
```

► Effets des instructions :

- 1 Création de l'int `a` et affectation de la valeur 37
- 2 Création du float `b` et affectation de la valeur 1.0
- 3 Équivalent à `a = a + b`

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
3 a += b
```

► Effets des instructions :

- 1 Création de l'int `a` et affectation de la valeur 37
- 2 Création du float `b` et affectation de la valeur 1.0
- 3 Équivalent à `a = a + b`. `a = int`, `b = float` \implies `a + b = float`

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
3 a += b
```

► Effets des instructions :

- 1 Création de l'int `a` et affectation de la valeur 37
- 2 Création du float `b` et affectation de la valeur 1.0
- 3 Équivalent à `a = a + b`. $a = \text{int}$, $b = \text{float} \implies a + b = \text{float}$
 $\implies a$ devient un float et vaut $a + b = 37 + 1.0 = 38.0$

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
3 a += b
4 print(a)
```

► Effets des instructions :

- 1 Création de l'int `a` et affectation de la valeur 37
- 2 Création du float `b` et affectation de la valeur 1.0
- 3 Équivalent à `a = a + b`. `a = int`, `b = float` $\implies a + b = \text{float}$
 $\implies a$ devient un float et vaut `a + b = 37 + 1.0 = 38.0`

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
3 a += b
4 print(a)
```

► Effets des instructions :

- 1 Création de l'int `a` et affectation de la valeur 37
- 2 Création du float `b` et affectation de la valeur 1.0
- 3 Équivalent à `a = a + b`. `a = int`, `b = float` $\implies a + b = \text{float}$
 $\implies a$ devient un float et vaut `a + b = 37 + 1.0 = 38.0`
- 4 Affiche 38.0

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
3 a += b
4 print(a)
5 print(a * b + 3)
```

► Effets des instructions :

- 1 Création de l'int a et affectation de la valeur 37
- 2 Création du float b et affectation de la valeur 1.0
- 3 Équivalent à $a = a + b$. $a = \text{int}$, $b = \text{float} \implies a + b = \text{float}$
 $\implies a$ devient un float et vaut $a + b = 37 + 1.0 = 38.0$
- 4 Affiche 38.0

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
3 a += b
4 print(a)
5 print(a * b + 3)
```

► Effets des instructions :

- 1 Création de l'int `a` et affectation de la valeur 37
- 2 Création du float `b` et affectation de la valeur 1.0
- 3 Équivalent à `a = a + b`. `a = int`, `b = float` $\implies a + b = \text{float}$
 $\implies a$ devient un float et vaut `a + b = 37 + 1.0 = 38.0`
- 4 Affiche 38.0
- 5 `a * b + 3` $\iff (a * b) + 3$

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
3 a += b
4 print(a)
5 print(a * b + 3)
```

► Effets des instructions :

- 1 Création de l'int a et affectation de la valeur 37
- 2 Création du float b et affectation de la valeur 1.0
- 3 Équivalent à $a = a + b$. $a = \text{int}$, $b = \text{float} \implies a + b = \text{float}$
 $\implies a$ devient un float et vaut $a + b = 37 + 1.0 = 38.0$
- 4 Affiche 38.0
- 5 $a * b + 3 \iff (a * b) + 3$
 $\iff (38.0 * 1.0) + 3$

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
3 a += b
4 print(a)
5 print(a * b + 3)
```

► Effets des instructions :

- 1 Création de l'int `a` et affectation de la valeur 37
- 2 Création du float `b` et affectation de la valeur 1.0
- 3 Équivalent à `a = a + b`. `a = int`, `b = float` $\implies a + b = \text{float}$
 $\implies a$ devient un float et vaut `a + b = 37 + 1.0 = 38.0`
- 4 Affiche 38.0
- 5 $a * b + 3 \iff (a * b) + 3$
 $\iff (38.0 * 1.0) + 3 = 41.0$

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
3 a += b
4 print(a)
5 print(a * b + 3)
```

► Effets des instructions :

- 1 Création de l'int `a` et affectation de la valeur 37
- 2 Création du float `b` et affectation de la valeur 1.0
- 3 Équivalent à `a = a + b`. `a = int`, `b = float` $\implies a + b = \text{float}$
 $\implies a$ devient un float et vaut `a + b = 37 + 1.0 = 38.0`
- 4 Affiche 38.0
- 5 $a * b + 3 \iff (a * b) + 3$
 $\iff (38.0 * 1.0) + 3 = 41.0$
 \implies affiche 41.0

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
3 a += b
4 print(a)
5 print(a * b + 3)
6 a = 3 / 4
```

► Effets des instructions :

- 1 Création de l'int a et affectation de la valeur 37
- 2 Création du float b et affectation de la valeur 1.0
- 3 Équivalent à $a = a + b$. $a = \text{int}$, $b = \text{float} \implies a + b = \text{float}$
 $\implies a$ devient un float et vaut $a + b = 37 + 1.0 = 38.0$
- 4 Affiche 38.0
- 5 $a * b + 3 \iff (a * b) + 3$
 $\iff (38.0 * 1.0) + 3 = 41.0$
 \implies affiche 41.0

Exemples de manipulation d'int et de float

► Programme Python :

```
1 a = 33 + 4
2 b = 1.0
3 a += b
4 print(a)
5 print(a * b + 3)
6 a = 3 / 4
```

► Effets des instructions :

- 1 Création de l'int a et affectation de la valeur 37
- 2 Création du float b et affectation de la valeur 1.0
- 3 Équivalent à $a = a + b$. $a = \text{int}$, $b = \text{float} \implies a + b = \text{float}$
 $\implies a$ devient un float et vaut $a + b = 37 + 1.0 = 38.0$
- 4 Affiche 38.0
- 5 $a * b + 3 \iff (a * b) + 3$
 $\iff (38.0 * 1.0) + 3 = 41.0$
 \implies affiche 41.0
- 6  / : division \implies float. Donc a devient un float qui vaut 0.75

Transtype

Transformation d'une valeur d'un type dans un autre type.

Transtypage

Transformation d'une valeur d'un type dans un autre type.

► Programme Python :

```
1 a = 5 / 4
```

Transtypage

Transformation d'une valeur d'un type dans un autre type.

► Programme Python :

```
1 a = 5 / 4
```

► Effets des instructions :

① division \implies a de type float et vaut 1,25

Transtypage

Transformation d'une valeur d'un type dans un autre type.

► Programme Python :

```
1 a = 5 / 4
2 b = int(5 / 4)
```

► Effets des instructions :

- ① division \implies a de type float et vaut 1,25

Transtypage

Transformation d'une valeur d'un type dans un autre type.

► Programme Python :

```
1 a = 5 / 4
2 b = int(5 / 4)
```

► Effets des instructions :

- 1 division \implies a de type float et vaut 1,25
- 2 on calcule $5/4 = 1,25$ (float) puis on transforme en int \implies b est un int et vaut 1

Transtypage

Transformation d'une valeur d'un type dans un autre type.

► Programme Python :

```
1 a = 5 / 4
2 b = int(5 / 4)
3 c = 3 * 4
```

► Effets des instructions :

- 1 division \implies a de type float et vaut 1,25
- 2 on calcule $5/4 = 1,25$ (float) puis on transforme en int \implies b est un int et vaut 1

Transtypage

Transformation d'une valeur d'un type dans un autre type.

► Programme Python :

```
1 a = 5 / 4
2 b = int(5 / 4)
3 c = 3 * 4
```

► Effets des instructions :

- 1 division \implies a de type float et vaut 1,25
- 2 on calcule $5/4 = 1,25$ (float) puis on transforme en int \implies b est un int et vaut 1
- 3 c est un int (multiplication de 2 int) et vaut 12

Transtypage

Transformation d'une valeur d'un type dans un autre type.

► Programme Python :

```
1 a = 5 / 4
2 b = int(5 / 4)
3 c = 3 * 4
4 d = float(3 * 4)
```

► Effets des instructions :

- 1 division \implies a de type float et vaut 1,25
- 2 on calcule $5/4 = 1,25$ (float) puis on transforme en int \implies b est un int et vaut 1
- 3 c est un int (multiplication de 2 int) et vaut 12

Transtypage

Transformation d'une valeur d'un type dans un autre type.

► Programme Python :

```
1 a = 5 / 4
2 b = int(5 / 4)
3 c = 3 * 4
4 d = float(3 * 4)
```

► Effets des instructions :

- 1 division \implies a de type float et vaut 1,25
- 2 on calcule $5/4 = 1,25$ (float) puis on transforme en int \implies b est un int et vaut 1
- 3 c est un int (multiplication de 2 int) et vaut 12
- 4 on calcule $3 * 4 \implies$ int égal à 12, puis on transforme en float \implies d est un float et vaut 12.0

Transtypage

Transformation d'une valeur d'un type dans un autre type.

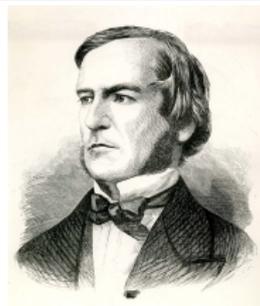
► Programme Python :

```
1 a = 5 / 4
2 b = int(5 / 4)
3 c = 3 * 4
4 d = float(3 * 4)
```

► Effets des instructions :

- 1 division \implies a de type float et vaut 1,25
- 2 on calcule $5/4 = 1,25$ (float) puis on transforme en int \implies b est un int et vaut 1
- 3 c est un int (multiplication de 2 int) et vaut 12
- 4 on calcule $3 * 4 \implies$ int égal à 12, puis on transforme en float \implies d est un float et vaut 12.0

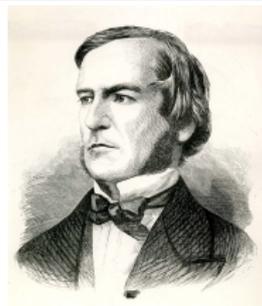
- ▶ George Boole (1815-1864)



- ▶ George Boole (1815-1864)
- ▶ Créateur de l'algèbre de Boole \implies logique



*The Mathematical Analysis of Logic :
being an Essay towards a Calculus of
Deductive Reasoning (1854)*

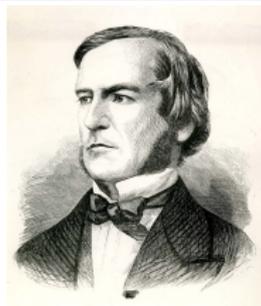


- ▶ George Boole (1815-1864)
- ▶ Créateur de l'algèbre de Boole \implies logique



*The Mathematical Analysis of Logic :
being an Essay towards a Calculus of
Deductive Reasoning (1854)*

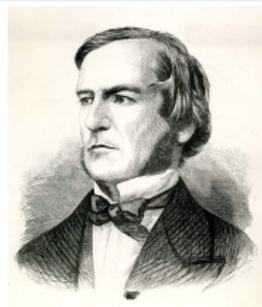
- ▶ 2 valeurs de vérité : **True** et **False**



- ▶ George Boole (1815-1864)
- ▶ Créateur de l'algèbre de Boole \implies logique



*The Mathematical Analysis of Logic :
being an Essay towards a Calculus of
Deductive Reasoning* (1854)

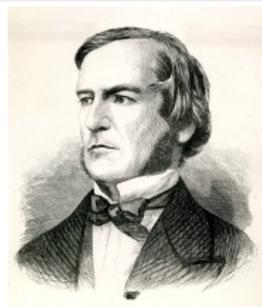


- ▶ 2 valeurs de vérité : **True** et **False**
- ▶ Permet de faire du raisonnement :
 - ▶ Si telle propriété est vraie (True) alors faire. . .

- ▶ George Boole (1815-1864)
- ▶ Créateur de l'algèbre de Boole \implies logique



*The Mathematical Analysis of Logic :
being an Essay towards a Calculus of
Deductive Reasoning* (1854)



- ▶ 2 valeurs de vérité : **True** et **False**
- ▶ Permet de faire du raisonnement :
 - ▶ Si telle propriété est vraie (True) alors faire. . .
 - ▶ Si clic sur bonbon alors griser le bonbon



Comment créer des booléens ?

- Utiliser les valeurs True et False :

```
a = True
```

```
a = False
```

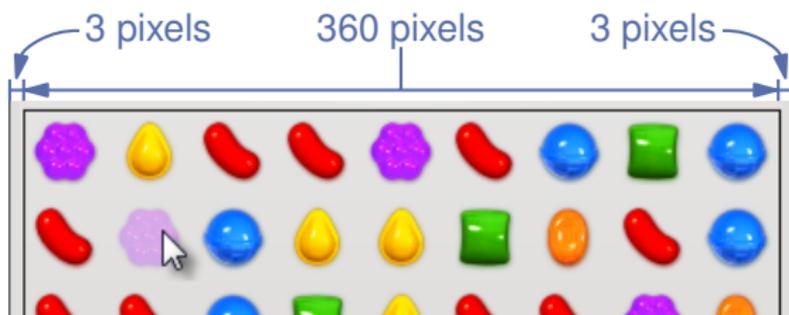
Comment créer des booléens ?

- Utiliser les valeurs True et False :

```
a = True
```

```
a = False
```

- Utiliser une expression booléenne :



`x_cursor` : position du curseur souris

curseur dans l'espace de jeu si $x_cursor \geq 3$ et $\leq 360 + 3$ pixels

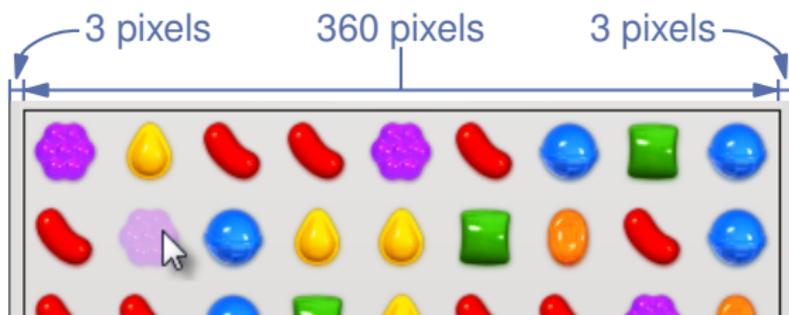
Comment créer des booléens ?

- Utiliser les valeurs True et False :

```
a = True
```

```
a = False
```

- Utiliser une expression booléenne :



`x_cursor` : position du curseur souris

curseur dans l'espace de jeu si `x_cursor` ≥ 3 et $\leq 360 + 3$ pixels

```
t1 = (x_cursor >= 3)
```

```
t2 = (x_cursor <= 360 + 3)
```

\implies t1 et t2 booléens = True

Opérateurs de comparaison

Opérateur	Valeur de l'expression
$x \geq y$	vaut True si et seulement si x supérieur ou égal à y
$x \leq y$	vaut True si et seulement si x inférieur ou égal à y
$x > y$	vaut True si et seulement si x strictement supérieur à y
$x < y$	vaut True si et seulement si x strictement inférieur à y

Opérateurs de comparaison

Opérateur	Valeur de l'expression
$x \geq y$	vaut True si et seulement si x supérieur ou égal à y
$x \leq y$	vaut True si et seulement si x inférieur ou égal à y
$x > y$	vaut True si et seulement si x strictement supérieur à y
$x < y$	vaut True si et seulement si x strictement inférieur à y
$x == y$	vaut True si et seulement si x est égal à y
$x != y$	vaut True si et seulement si x différent de y



Test d'égalité : `< == >` et non `< = >` !!!

Opérateurs de comparaison

Opérateur	Valeur de l'expression
<code>x >= y</code>	vaut True si et seulement si x supérieur ou égal à y
<code>x <= y</code>	vaut True si et seulement si x inférieur ou égal à y
<code>x > y</code>	vaut True si et seulement si x strictement supérieur à y
<code>x < y</code>	vaut True si et seulement si x strictement inférieur à y
<code>x == y</code>	vaut True si et seulement si x est égal à y
<code>x != y</code>	vaut True si et seulement si x différent de y



Test d'égalité : `< == >` et non `< = >` !!!

```
a = 3 == 2
print(a)
```

Opérateurs de comparaison

Opérateur	Valeur de l'expression
<code>x >= y</code>	vaut True si et seulement si x supérieur ou égal à y
<code>x <= y</code>	vaut True si et seulement si x inférieur ou égal à y
<code>x > y</code>	vaut True si et seulement si x strictement supérieur à y
<code>x < y</code>	vaut True si et seulement si x strictement inférieur à y
<code>x == y</code>	vaut True si et seulement si x est égal à y
<code>x != y</code>	vaut True si et seulement si x différent de y



Test d'égalité : `< == >` et non `< = >` !!!

```
a = 3 == 2  
print(a)
```

affectation de `(3 == 2)` à a

Opérateurs de comparaison

Opérateur	Valeur de l'expression
<code>x >= y</code>	vaut True si et seulement si x supérieur ou égal à y
<code>x <= y</code>	vaut True si et seulement si x inférieur ou égal à y
<code>x > y</code>	vaut True si et seulement si x strictement supérieur à y
<code>x < y</code>	vaut True si et seulement si x strictement inférieur à y
<code>x == y</code>	vaut True si et seulement si x est égal à y
<code>x != y</code>	vaut True si et seulement si x différent de y



Test d'égalité : `< == >` et non `< = >` !!!

```
a = 3 == 2  
print(a)
```

affectation de `(3 == 2)` à a

```
False
```

Opérateurs de comparaison

Opérateur	Valeur de l'expression
<code>x >= y</code>	vaut True si et seulement si x supérieur ou égal à y
<code>x <= y</code>	vaut True si et seulement si x inférieur ou égal à y
<code>x > y</code>	vaut True si et seulement si x strictement supérieur à y
<code>x < y</code>	vaut True si et seulement si x strictement inférieur à y
<code>x == y</code>	vaut True si et seulement si x est égal à y
<code>x != y</code>	vaut True si et seulement si x différent de y



Test d'égalité : `< == >` et non `< = >` !!!

```
a = 3 == 2
print(a)
b = a = 3
print(a,b)
```

affectation de `(3 == 2)` à a

False

Opérateurs de comparaison

Opérateur	Valeur de l'expression
<code>x >= y</code>	vaut True si et seulement si x supérieur ou égal à y
<code>x <= y</code>	vaut True si et seulement si x inférieur ou égal à y
<code>x > y</code>	vaut True si et seulement si x strictement supérieur à y
<code>x < y</code>	vaut True si et seulement si x strictement inférieur à y
<code>x == y</code>	vaut True si et seulement si x est égal à y
<code>x != y</code>	vaut True si et seulement si x différent de y



Test d'égalité : `< == >` et non `< = >` !!!

```
a = 3 == 2
print(a)
b = a = 3
print(a, b)
```

affectation de `(3 == 2)` à a

affectation de 3 à a puis à b

False

Opérateurs de comparaison

Opérateur	Valeur de l'expression
<code>x >= y</code>	vaut True si et seulement si x supérieur ou égal à y
<code>x <= y</code>	vaut True si et seulement si x inférieur ou égal à y
<code>x > y</code>	vaut True si et seulement si x strictement supérieur à y
<code>x < y</code>	vaut True si et seulement si x strictement inférieur à y
<code>x == y</code>	vaut True si et seulement si x est égal à y
<code>x != y</code>	vaut True si et seulement si x différent de y



Test d'égalité : `< == >` et non `< = >` !!!

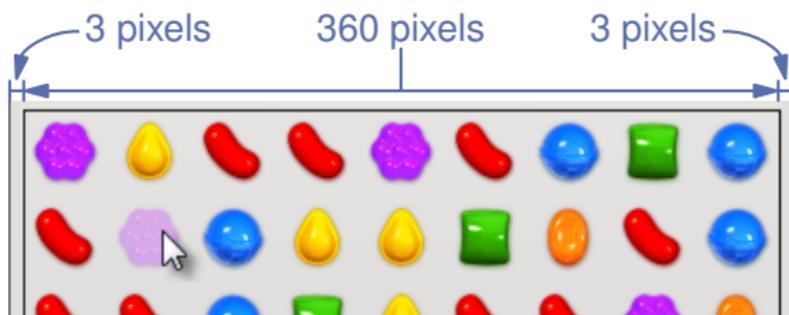
```
a = 3 == 2
print(a)
b = a = 3
print(a, b)
```

affectation de `(3 == 2)` à a

affectation de 3 à a puis à b

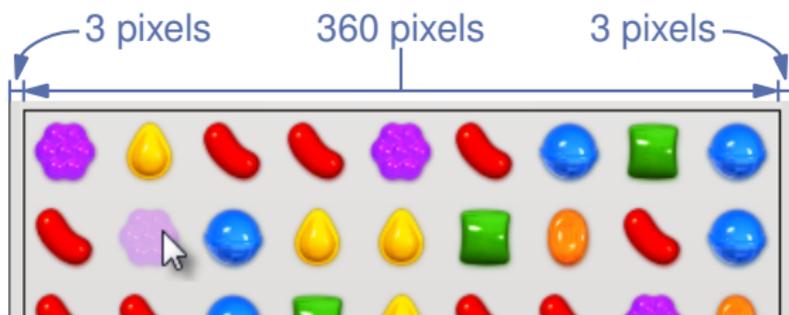
```
False
3 3
```

Opérateurs booléens (1/3)



► **OU logique** : `expr1 or expr2`

Opérateurs booléens (1/3)

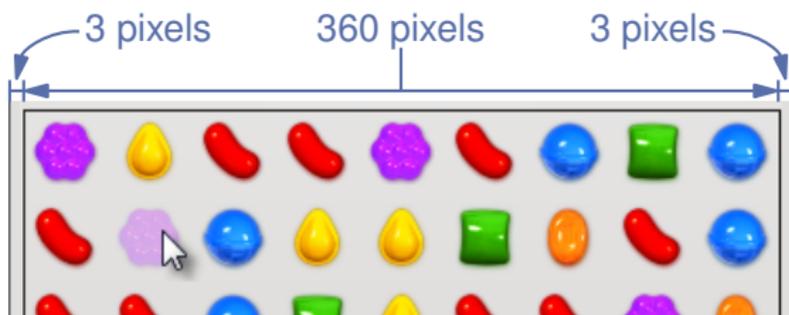


► **OU logique** : `expr1 or expr2`

vrai si au moins une des expressions est vraie

curseur en dehors du jeu : `x_cursor < 3 or x_cursor > 363`

Opérateurs booléens (1/3)



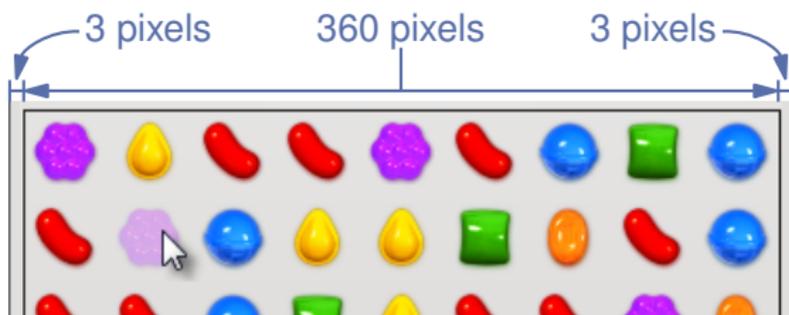
► **OU logique** : `expr1 or expr2`

vrai si au moins une des expressions est vraie

curseur en dehors du jeu : `x_cursor < 3 or x_cursor > 363`

► **ET logique** : `expr1 and expr2`

Opérateurs booléens (1/3)



► **OU logique** : `expr1 or expr2`

vrai si au moins une des expressions est vraie

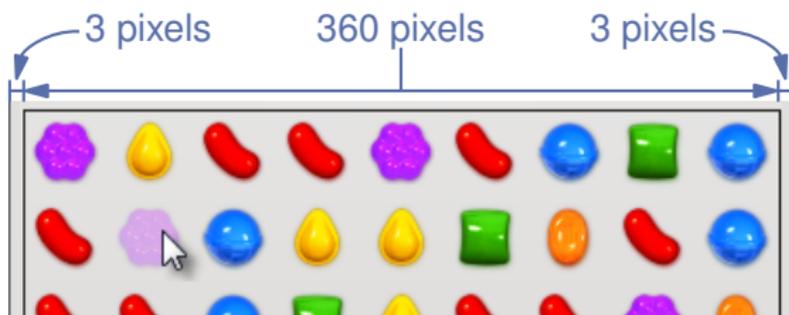
curseur en dehors du jeu : `x_cursor < 3 or x_cursor > 363`

► **ET logique** : `expr1 and expr2`

vrai si toutes les expressions sont vraies

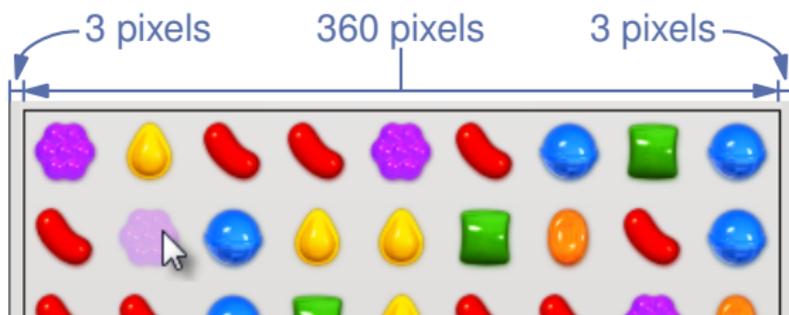
curseur dans le jeu : `x_cursor >= 3 and x_cursor <= 363`

Opérateurs booléens (2/3)



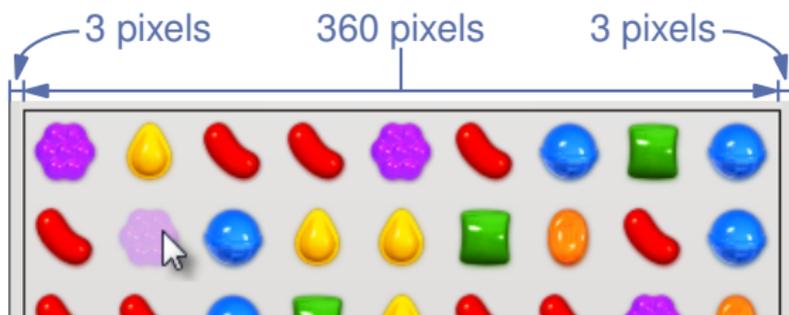
► **NON logique** : `not expr`

Opérateurs booléens (2/3)



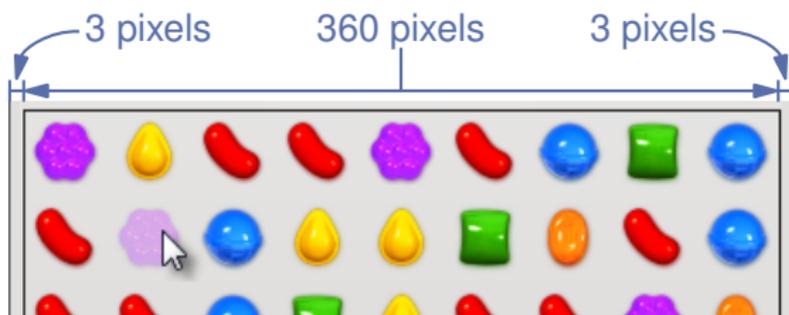
- ▶ **NON logique** : `not expr`
vrai si l'expression est fausse
 curseur à gauche de l'espace de jeu : `not (x_cursor >= 3)`

Opérateurs booléens (2/3)



- ▶ **NON logique** : `not expr`
vrai si l'expression est fausse
 curseur à gauche de l'espace de jeu : `not (x_cursor >= 3)`
- ▶ **OU exclusif** : `expr1 ^ expr2`

Opérateurs booléens (2/3)



► **NON logique** : `not expr`

vrai si l'expression est fausse

curseur à gauche de l'espace de jeu : `not (x_cursor >= 3)`

► **OU exclusif** : `expr1 ^ expr2`

vrai si une des deux expressions est vraie et l'autre est fausse

curseur en dehors du jeu : `x_cursor >= 3 ^ x_cursor <= 363`

Opérateurs booléens (3/3)

- ▶ **Table de vérité du OU logique :** `expr1 or expr2`

<code>expr1\expr2</code>	<code>False</code>	<code>True</code>
<code>False</code>	<code>False</code>	<code>True</code>
<code>True</code>	<code>True</code>	<code>True</code>

Opérateurs booléens (3/3)

- ▶ **Table de vérité du OU logique :** `expr1 or expr2`

<code>expr1\expr2</code>	False	True
False	False	True
True	True	True

- ▶ **Table de vérité du ET logique :** `expr1 and expr2`

<code>expr1\expr2</code>	False	True
False	False	False
True	False	True

Opérateurs booléens (3/3)

- **Table de vérité du OU logique :** `expr1 or expr2`

<code>expr1\expr2</code>	False	True
False	False	True
True	True	True

- **Table de vérité du ET logique :** `expr1 and expr2`

<code>expr1\expr2</code>	False	True
False	False	False
True	False	True

- **Table de vérité du NON logique :** `not expr`

<code>expr</code>	False	True
	True	False

Opérateurs booléens (3/3)

- ▶ **Table de vérité du OU logique** : `expr1 or expr2`

<code>expr1\expr2</code>	False	True
False	False	True
True	True	True

- ▶ **Table de vérité du ET logique** : `expr1 and expr2`

<code>expr1\expr2</code>	False	True
False	False	False
True	False	True

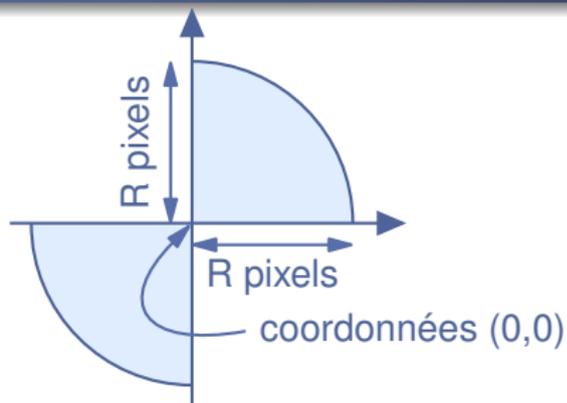
- ▶ **Table de vérité du NON logique** : `not expr`

<code>expr</code>	False	True
	True	False

- ▶ **Table de vérité du OU exclusif** : `expr1 ^ expr2`

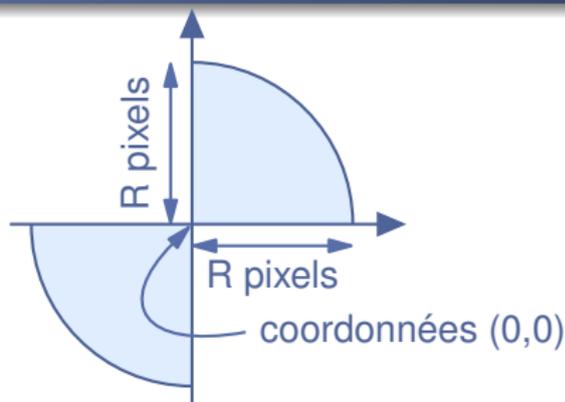
<code>expr1\expr2</code>	False	True
False	False	True
True	True	False

Expression booléenne plus complexe



- ▶ Test si le curseur dans la zone bleutée

Expression booléenne plus complexe



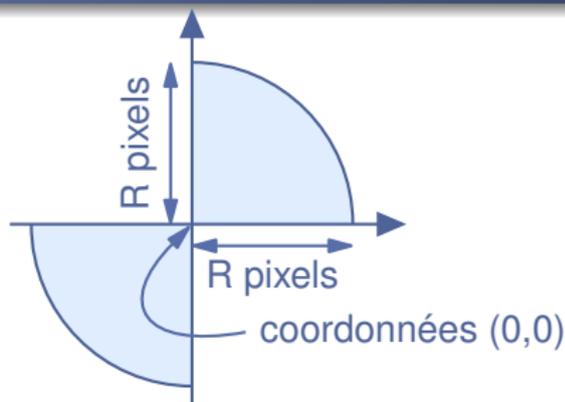
► Test si le curseur dans la zone bleutée :

```
x curseur_2 = x curseur * x curseur
y curseur_2 = y curseur * y curseur
rayon_2 = R * R

test_zone = (x curseur_2 + y curseur_2 <= rayon_2) and \
  ( ( x curseur >= 0 ) and ( y curseur >= 0 ) ) or \
  ( ( x curseur <= 0 ) and ( y curseur <= 0 ) ) )
```

⚠ « \ » : permet de faire tenir une instruction sur plusieurs lignes

Expression booléenne plus complexe



► Test si le curseur dans la zone bleutée :

```
x curseur_2 = x curseur * x curseur
y curseur_2 = y curseur * y curseur
rayon_2 = R * R
```

```
test_zone = (x curseur_2 + y curseur_2 <= rayon_2) and \
  ( ( x curseur >= 0 and y curseur >= 0 ) or \
    ( x curseur <= 0 and y curseur <= 0 ) )
```

⚠ « \ » : permet de faire tenir une instruction sur plusieurs lignes

⚠ **Conseil** : parenthésez les expressions !

Un bon programmeur commente toujours son code !

Commentaires

- ▶ **Ultra importants** pour bien comprendre son code !
- ▶ Commentaire en Python : débute par # et va jusqu'à la fin de la ligne
- ▶ Mais un # à l'intérieur d'une chaîne de caractères \neq commentaire (Exemple : "ce # n'est pas un commentaire")

Un bon programmeur commente toujours son code !

Commentaires

- ▶ **Ultra importants** pour bien comprendre son code !
- ▶ Commentaire en Python : débute par # et va jusqu'à la fin de la ligne
- ▶ Mais un # à l'intérieur d'une chaîne de caractères \neq commentaire (Exemple : "ce # n'est pas un commentaire")

Test du transparent précédent :

```
x curseur_2 = x curseur * x curseur # carre de x curseur
y curseur_2 = y curseur * y curseur
rayon_2 = R * R

# test que le curseur est dans une des zones bleutées
test_zone = \
    # on teste ici que le curseur est dans le cercle
    (x curseur_2 + y curseur_2 <= rayon_2) and \
    # on teste qu'il est dans le quadrant supérieur droit
    ( ( x curseur >= 0 ) and ( y curseur >= 0 ) ) or \
    # on test qu'il est dans le quadrant inférieur gauche
    ( ( x curseur <= 0 ) and ( y curseur <= 0 ) ) )
```

Constante

Valeur qui ne pourra jamais être modifiée durant l'exécution du programme.

Constante

Valeur qui ne pourra jamais être modifiée durant l'exécution du programme.

► Exemple dans notre candy crush :

```
NB_COLONNES_JEU    = 9  # nb colonnes de l'espace de jeu
NB_LIGNES_JEU      = 9  # nb lignes de l'espace de jeu
NB_MOUVEMENTS_JEU = 20 # nb mouvements que le joueur
                    # est autorisé à faire
```

Constante

Valeur qui ne pourra jamais être modifiée durant l'exécution du programme.

► Exemple dans notre candy crush :

```
NB_COLONNES_JEU    = 9  # nb colonnes de l'espace de jeu
NB_LIGNES_JEU      = 9  # nb lignes de l'espace de jeu
NB_MOUVEMENTS_JEU = 20 # nb mouvements que le joueur
                    # est autorisé à faire
```



Python ne connaît pas la notion de constante

Constante

Valeur qui ne pourra jamais être modifiée durant l'exécution du programme.

► Exemple dans notre candy crush :

```
NB_COLONNES_JEU    = 9  # nb colonnes de l'espace de jeu  
NB_LIGNES_JEU      = 9  # nb lignes de l'espace de jeu  
NB_MOUVEMENTS_JEU = 20 # nb mouvements que le joueur  
                    # est autorisé à faire
```



Python ne connaît pas la notion de constante

⇒ On utilise des variables à la place

Convention : constantes symbolisées par des variables dont les noms sont uniquement avec des majuscules

Pourquoi utiliser des constantes ?

2 raisons principalement :

- ▶ Rend le code source plus simple à lire

Pourquoi utiliser des constantes ?

2 raisons principalement :

- ▶ Rend le code source plus simple à lire
- ▶ Permet de rendre vos programmes « paramétrables »
⇒ modifs programmes plus simples et moins bugguées !!!

Pourquoi utiliser des constantes ?

2 raisons principalement :

- ▶ Rend le code source plus simple à lire
- ▶ Permet de rendre vos programmes « paramétrables »
⇒ modifs programmes plus simples et moins bugguées!!!

▶ **Extrait de candy crush :**

```
def getXCoord(x_pixel):  
    return (x_pixel - 3) // 40
```

```
def getXPixel(x_coord):  
    return x_coord * 40 + 3
```

```
def getXCoord(x_pixel):  
    return (x_pixel - X_TABLEAU_JEU) // TAILLE_BONBON
```

```
def getXPixel(x_coord):  
    return x_coord * TAILLE_BONBON + X_TABLEAU_JEU
```

② Les fonctions

- ▶ **Programme** : Demander à l'utilisateur de saisir un nombre (positif) x et afficher la valeur de \sqrt{x} , $\sqrt{x+1}$ et $\sqrt{x+2}$

- ▶ **Programme** : Demander à l'utilisateur de saisir un nombre (positif) x et afficher la valeur de \sqrt{x} , $\sqrt{x+1}$ et $\sqrt{x+2}$
- ▶ **Problème** : Comment calculer la racine carrée de x ?

- ▶ **Programme** : Demander à l'utilisateur de saisir un nombre (positif) x et afficher la valeur de \sqrt{x} , $\sqrt{x+1}$ et $\sqrt{x+2}$
- ▶ **Problème** : Comment calculer la racine carrée de x ?
- ▶ **Solution** : Calculer les suites u_n et v_n jusqu'à convergence :
 - ▶ $u_0 = 1, v_0 = x$
 - ▶ $u_{n+1} = \frac{2}{\frac{1}{u_n} + \frac{1}{v_n}}$
 - ▶ $v_{n+1} = \frac{u_n + v_n}{2}$
 - ▶ Convergence $\implies u_n \approx v_n \approx \sqrt{x}$

- ▶ **Programme** : Demander à l'utilisateur de saisir un nombre (positif) x et afficher la valeur de \sqrt{x} , $\sqrt{x+1}$ et $\sqrt{x+2}$
- ▶ **Problème** : Comment calculer la racine carrée de x ?
- ▶ **Solution** : Calculer les suites u_n et v_n jusqu'à convergence :
 - ▶ $u_0 = 1, v_0 = x$
 - ▶ $u_{n+1} = \frac{2}{\frac{1}{u_n} + \frac{1}{v_n}}$
 - ▶ $v_{n+1} = \frac{u_n + v_n}{2}$
 - ▶ Convergence $\implies u_n \approx v_n \approx \sqrt{x}$
- ▶ Code python = écrire ces itérations
- ▶ Si convergence en 10 itérations \implies code de plus $10 \times 4 = 40$ lignes

- ▶ **Programme** : Demander à l'utilisateur de saisir un nombre (positif) x et afficher la valeur de \sqrt{x} , $\sqrt{x+1}$ et $\sqrt{x+2}$
- ▶ **Problème** : Comment calculer la racine carrée de x ?
- ▶ **Solution** : Calculer les suites u_n et v_n jusqu'à convergence :
 - ▶ $u_0 = 1, v_0 = x$
 - ▶ $u_{n+1} = \frac{2}{\frac{1}{u_n} + \frac{1}{v_n}}$
 - ▶ $v_{n+1} = \frac{u_n + v_n}{2}$
 - ▶ Convergence $\implies u_n \approx v_n \approx \sqrt{x}$
- ▶ Code python = écrire ces itérations
- ▶ Si convergence en 10 itérations \implies code de plus $10 \times 4 = 40$ lignes
- ▶ Code pas très lisible : dans l'énoncé, pas de suite u_n, v_n

Code de la mort qui tue

- ▶ Code pour afficher \sqrt{x} et $\sqrt{x+1}$ (convergence en 2 itérations) :

```
x = input("Saisissez un nombre positif : ")

u_n = 1.0          # pour n = 0, u_n = u_0
v_n = float(x)    # on garantit que v_n sera un float
u_n_1 = 2 / ((1 / u_n) + (1 / v_n)) # calcule u_{n+1}
v_n_1 = (u_n + v_n) / 2
u_n = u_n_1       # pour se réserver des 2 lignes du dessus,
v_n = v_n_1       # on affecte à u_n la valeur de u_{n-1}
u_n_1 = 2 / ((1 / u_n) + (1 / v_n))
v_n_1 = (u_n + v_n) / 2
print("sqrt(", x, ") =", u_n_1)

u_n = 1.0
v_n = float(x+1)
u_n_1 = 2 / ((1 / u_n) + (1 / v_n))
v_n_1 = (u_n + v_n) / 2
u_n = u_n_1
v_n = v_n_1
u_n_1 = 2 / ((1 / u_n) + (1 / v_n))
v_n_1 = (u_n + v_n) / 2
print("sqrt(", x+1, ") =", u_n_1)
```

- ▶ **2 inconvénients majeurs de ce code :**
 - ▶ illisible \implies on ne voit pas ce qu'il fait en le lisant
 - ▶ nécessité de savoir comment calculer une racine carrée

► 2 inconvénients majeurs de ce code :

- illisible \implies on ne voit pas ce qu'il fait en le lisant
- nécessite de savoir comment calculer une racine carrée

► **Solution** : utiliser la fonction `sqrt`

```
x = input("Saisissez un nombre positif : ")  
  
print("sqrt(", x, ") =", sqrt(x))  
print("sqrt(", x+1, ") =", sqrt(x+1))  
print("sqrt(", x+2, ") =", sqrt(x+2))
```

\implies code lisible et simple !

On peut faire mieux

▶ 2 inconvénients majeurs de ce code :

- ▶ illisible \implies on ne voit pas ce qu'il fait en le lisant
- ▶ nécessite de savoir comment calculer une racine carrée

▶ **Solution** : utiliser la fonction `sqrt`

```
x = input("Saisissez un nombre positif : ")  
  
print("sqrt(", x, ") =", sqrt(x))  
print("sqrt(", x+1, ") =", sqrt(x+1))  
print("sqrt(", x+2, ") =", sqrt(x+2))
```

\implies code lisible et simple !

Appel de fonction : `nom_fonction(paramètres)`

▶ **Exemples de fonctions** : `print`, `input`, `sqrt`



Plusieurs paramètres \implies séparés par des `<< , >>`

Pourquoi les fonctions ?

- ▶ Pas besoin de savoir comment elles sont écrites
(ex : affichage d'une image de bonbon de candy crush)

Pourquoi les fonctions ?

- ▶ Pas besoin de savoir comment elles sont écrites
(ex : affichage d'une image de bonbon de candy crush)
- ▶ Réutilisabilité d'un code déjà écrit
(10 appels de certaines fonctions dans candy crush)

Pourquoi les fonctions ?

- ▶ Pas besoin de savoir comment elles sont écrites (ex : affichage d'une image de bonbon de candy crush)
- ▶ Réutilisabilité d'un code déjà écrit (10 appels de certaines fonctions dans candy crush)
- ▶ Simplifie la maintenance du code (correction de bugs)

Pourquoi les fonctions ?

- ▶ Pas besoin de savoir comment elles sont écrites (ex : affichage d'une image de bonbon de candy crush)
- ▶ Réutilisabilité d'un code déjà écrit (10 appels de certaines fonctions dans candy crush)
- ▶ Simplifie la maintenance du code (correction de bugs)
- ▶ Améliore la lisibilité de vos programmes

Pourquoi les fonctions ?

- ▶ Pas besoin de savoir comment elles sont écrites (ex : affichage d'une image de bonbon de candy crush)
- ▶ Réutilisabilité d'un code déjà écrit (10 appels de certaines fonctions dans candy crush)
- ▶ Simplifie la maintenance du code (correction de bugs)
- ▶ Améliore la lisibilité de vos programmes
- ▶ Permet de structurer les programmes en les découpant en petits morceaux

Quelques conseils pour écrire vos fonctions

- ▶ Une fonction doit être **courte** et **simple**.
⇒ éviter des fonctions de plus de 30-40 lignes
- ▶ Les lignes ne doivent **pas être trop longues**
⇒ éviter les lignes de plus de 80 caractères

Quelques conseils pour écrire vos fonctions

- ▶ Une fonction doit être **courte** et **simple**.
⇒ éviter des fonctions de plus de 30-40 lignes
- ▶ Les lignes ne doivent **pas être trop longues**
⇒ éviter les lignes de plus de 80 caractères

⇒ La fonction se voit entièrement sur un écran

Quelques conseils pour écrire vos fonctions

- ▶ Une fonction doit être **courte** et **simple**.
⇒ éviter des fonctions de plus de 30-40 lignes
- ▶ Les lignes ne doivent **pas être trop longues**
⇒ éviter les lignes de plus de 80 caractères

⇒ La fonction se voit entièrement sur un écran

- ▶ Dans le code, **séparer les fonctions** par deux lignes vides

⇒ Code plus facile à lire

Quelques conseils pour écrire vos fonctions

- ▶ Une fonction doit être **courte** et **simple**.
⇒ éviter des fonctions de plus de 30-40 lignes
- ▶ Les lignes ne doivent **pas être trop longues**
⇒ éviter les lignes de plus de 80 caractères

⇒ La fonction se voit entièrement sur un écran

- ▶ Dans le code, **séparer les fonctions** par deux lignes vides

⇒ Code plus facile à lire

- ▶ Une fonction ne fait qu'**une et une seule chose**
⇒ pas de fonction qui calcule les match3 de candy crush, déplace des bonbons, en transforme certains et affiche du texte

Quelques conseils pour écrire vos fonctions

- ▶ Une fonction doit être **courte** et **simple**.
⇒ éviter des fonctions de plus de 30-40 lignes
- ▶ Les lignes ne doivent **pas être trop longues**
⇒ éviter les lignes de plus de 80 caractères

⇒ La fonction se voit entièrement sur un écran

- ▶ Dans le code, **séparer les fonctions** par deux lignes vides

⇒ Code plus facile à lire

- ▶ Une fonction ne fait qu'**une et une seule chose**
⇒ pas de fonction qui calcule les match3 de candy crush, déplace des bonbons, en transforme certains et affiche du texte
- ▶ Le nom de la fonction doit **indiquer ce qu'elle fait**

Quelques conseils pour écrire vos fonctions

- ▶ Une fonction doit être **courte** et **simple**.
⇒ éviter des fonctions de plus de 30-40 lignes
- ▶ Les lignes ne doivent **pas être trop longues**
⇒ éviter les lignes de plus de 80 caractères

⇒ La fonction se voit entièrement sur un écran

- ▶ Dans le code, **séparer les fonctions** par deux lignes vides

⇒ Code plus facile à lire

- ▶ Une fonction ne fait qu'**une et une seule chose**
⇒ pas de fonction qui calcule les match3 de candy crush, déplace des bonbons, en transforme certains et affiche du texte
- ▶ Le nom de la fonction doit **indiquer ce qu'elle fait**

Fonction complexe ⇒ à découper en plusieurs morceaux

Quelques conseils pour écrire vos fonctions

- ▶ Une fonction doit être **courte** et **simple**.
⇒ éviter des fonctions de plus de 30-40 lignes
- ▶ Les lignes ne doivent **pas être trop longues**
⇒ éviter les lignes de plus de 80 caractères

⇒ La fonction se voit entièrement sur un écran

- ▶ Dans le code, **séparer les fonctions** par deux lignes vides

⇒ Code plus facile à lire

- ▶ Une fonction ne fait qu'**une et une seule chose**
⇒ pas de fonction qui calcule les match3 de candy crush, déplace des bonbons, en transforme certains et affiche du texte
- ▶ Le nom de la fonction doit **indiquer ce qu'elle fait**

Fonction complexe ⇒ à découper en plusieurs morceaux

Quelques conseils pour écrire vos fonctions

- ▶ Une fonction doit être **courte** et **simple**.
⇒ éviter des fonctions de plus de 30-40 lignes
- ▶ Les lignes ne doivent **pas être trop longues**
⇒ éviter les lignes de plus de 80 caractères

⇒ La fonction se voit entièrement sur un écran

- ▶ Dans le code, **séparer les fonctions** par deux lignes vides

⇒ Code plus facile à lire

- ▶ Une fonction ne fait qu'**une et une seule chose**
⇒ pas de fonction qui calcule les match3 de candy crush, déplace des bonbons, en transforme certains et affiche du texte
- ▶ Le nom de la fonction doit **indiquer ce qu'elle fait**

Fonction complexe ⇒ à découper en plusieurs morceaux

...extrait du Linux kernel coding style



Simplicité

+

lisibilité

Moins de bugs!!!

Comment définir des fonctions ?

Création d'une fonction :

```
def nom_fonction(param1, param2):  
    """ce que fait la fonction  
  
    La doc sur plusieurs lignes  
    """  
  
    instruction_1  
    ...  
    instruction_n  
  
    return valeur
```

Comment définir des fonctions ?

Création d'une fonction :

mot clef



```
def nom_fonction(param1, param2):  
    """ce que fait la fonction  
  
    La doc sur plusieurs lignes  
    """  
  
    instruction_1  
    ...  
    instruction_n  
  
    return valeur
```

Comment définir des fonctions ?

Création d'une fonction :

mot clef

paramètres

```
def nom_fonction(param1, param2):  
    """ce que fait la fonction  
  
    La doc sur plusieurs lignes  
    """  
  
    instruction_1  
    ...  
    instruction_n  
  
    return valeur
```

Comment définir des fonctions ?

Création d'une fonction :

parenthèses obligatoires, même si pas de paramètre

mot clef

paramètres

```
def nom_fonction(param1, param2):  
    """ce que fait la fonction
```

```
    La doc sur plusieurs lignes  
    """
```

```
    instruction_1
```

```
    ...
```

```
    instruction_n
```

```
    return valeur
```

Comment définir des fonctions ?

Création d'une fonction :

parenthèses obligatoires, même si pas de paramètre

mot clef

paramètres

```
def nom_fonction(param1, param2):
```

```
    """ce que fait la fonction
```

```
    La doc sur plusieurs lignes
```

```
    """
```

documentation de la
fonction : dans console
Spyder, help(nom_fonction)

```
    instruction_1
```

```
    ...
```

```
    instruction_n
```

```
    return valeur
```

Comment définir des fonctions ?

Création d'une fonction :

parenthèses obligatoires, même si pas de paramètre

mot clef

paramètres

```
def nom_fonction(param1, param2):
```

```
    """ce que fait la fonction
```

```
    La doc sur plusieurs lignes
```

```
    """
```

documentation de la
fonction : dans console
Spyder, help(nom_fonction)

```
    instruction_1
```

```
    ...
```

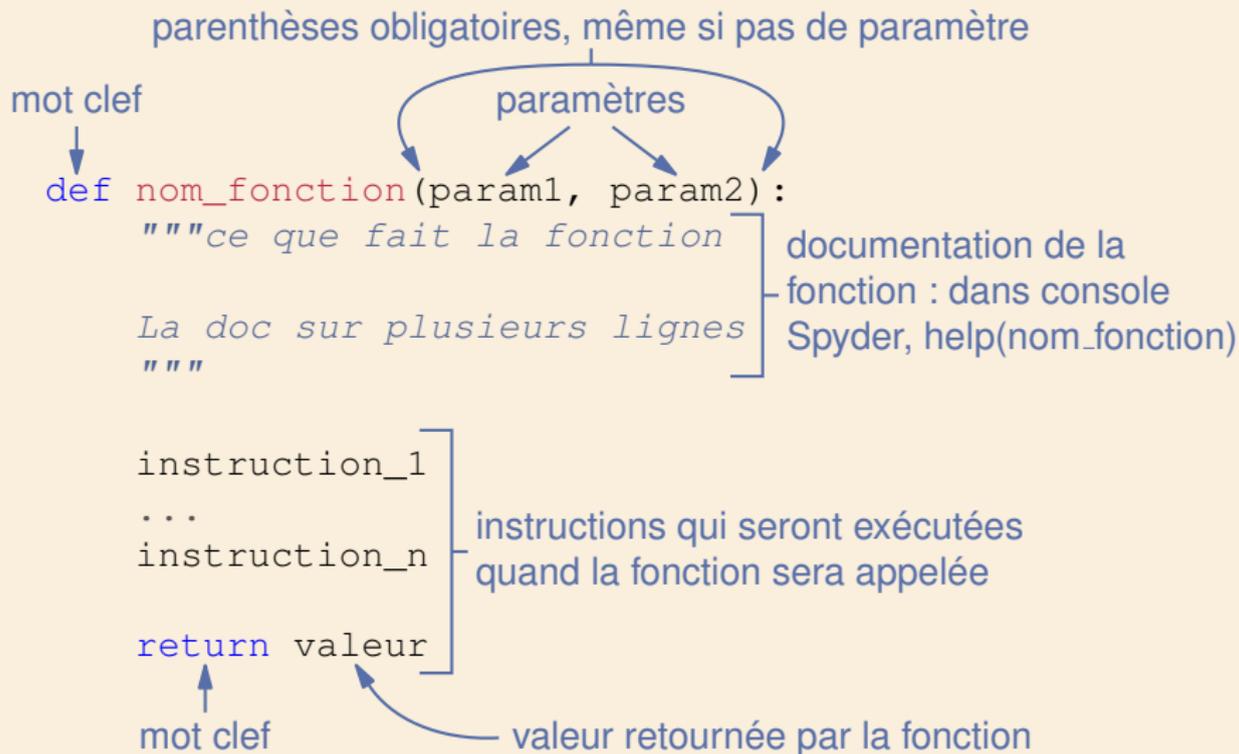
```
    instruction_n
```

```
    return valeur
```

instructions qui seront exécutées
quand la fonction sera appelée

Comment définir des fonctions ?

Création d'une fonction :



Comment définir des fonctions ?

Création d'une fonction :

parenthèses obligatoires, même si pas de paramètre

mot clef

paramètres

```
def nom_fonction(param1, param2):
```

```
    """ce que fait la fonction
```

```
    La doc sur plusieurs lignes
```

```
    """
```

documentation de la
fonction : dans console
Spyder, help(nom_fonction)

```
    instruction_1
```

```
    ...
```

```
    instruction_n
```

instructions qui seront exécutées
quand la fonction sera appelée

```
    return valeur
```

mot clef

valeur retournée par la fonction

indentation

Exemples de définitions de fonctions

```
1 def carre(x) :  
2     """Renvoie le carré du nombre passé en argument"""  
3     return x * x
```

Exemples de définitions de fonctions

```
1 def carre(x) :  
2     """Renvoie le carré du nombre passé en argument"""  
3     return x * x  
4  
5 def puissance(x, y):  
6     """puissance(x,y) renvoie la valeur de x puissance y"""  
7     return x ** y
```

Exemples de définitions de fonctions

```
1 def carre(x) :  
2     """Renvoie le carré du nombre passé en argument"""  
3     return x * x  
4  
5 def puissance(x, y):  
6     """puissance(x,y) renvoie la valeur de x puissance y"""  
7     return x ** y  
8  
9 def affiche_trois_fois(x):  
10    """Affiche 3 fois son paramètre"""  
11    print(x)  
12    print(x)  
13    print(x)
```

- ▶ `affiche_trois_fois` : ne renvoie rien !
⇒ s'appelle une **procédure** plutôt qu'une fonction

Exemples de définitions de fonctions

```
1 def carre(x) :  
2     """Renvoie le carré du nombre passé en argument"""  
3     return x * x  
4  
5 def puissance(x, y):  
6     """puissance(x,y) renvoie la valeur de x puissance y"""  
7     return x ** y  
8  
9 def affiche_trois_fois(x):  
10    """Affiche 3 fois son paramètre"""  
11    print(x)  
12    print(x)  
13    print(x)  
14  
15 def affiche_cadre():  
16    print("=====  
17    print("# cadre #")  
18    print("=====")
```

- ▶ `affiche_trois_fois` : ne renvoie rien !
⇒ s'appelle une **procédure** plutôt qu'une fonction
- ▶ `affiche_cadre` : ne prend pas d'argument (paramètre)



il faut quand même mettre les parenthèses !

Appel de fonctions

Comment appeler une fonction

Écrire le nom de la fonction et spécifier les valeurs des paramètres entre parenthèses.

Comment appeler une fonction

Écrire le nom de la fonction et spécifier les valeurs des paramètres entre parenthèses.

```
1  # définition de la fonction
2  def puissance(x, y):
3      return x ** y
4
5  # appel de la fonction
6  a = puissance(1+1+1, 2) # affecte 3 puissance 2 = 9 à a
```

Comment appeler une fonction

Écrire le nom de la fonction et spécifier les valeurs des paramètres entre parenthèses.

```
1 # définition de la fonction
2 def puissance(x, y):
3     return x ** y
4
5 # appel de la fonction
6 a = puissance(1+1+1, 2) # affecte 3 puissance 2 = 9 à a
```



Code de la fonction exécuté **seulement** quand on appelle la fonction, pas quand on la définit !

Appel de fonctions

Comment appeler une fonction

Écrire le nom de la fonction et spécifier les valeurs des paramètres entre parenthèses.

```
1 # définition de la fonction
2 def puissance(x, y):
3     return x ** y
4
5 # appel de la fonction
6 a = puissance(1+1+1, 2) # affecte 3 puissance 2 = 9 à a
```



Code de la fonction exécuté **seulement** quand on appelle la fonction, pas quand on la définit !

Que se passe-t-il lors de l'appel ?

- 1 Évaluation des valeurs passées en argument de la fonction (ici : 3 et 2)

Appel de fonctions

Comment appeler une fonction

Écrire le nom de la fonction et spécifier les valeurs des paramètres entre parenthèses.

```
1 # définition de la fonction
2 def puissance(x, y):
3     return x ** y
4
5 # appel de la fonction
6 a = puissance(1+1+1, 2) # affecte 3 puissance 2 = 9 à a
```



Code de la fonction exécuté **seulement** quand on appelle la fonction, pas quand on la définit !

Que se passe-t-il lors de l'appel ?

- 1 Évaluation des valeurs passées en argument de la fonction (ici : 3 et 2)
- 2 Affectation de ces valeurs aux paramètres (ici x et y)

Appel de fonctions

Comment appeler une fonction

Écrire le nom de la fonction et spécifier les valeurs des paramètres entre parenthèses.

```
1 # définition de la fonction
2 def puissance(x, y):
3     return x ** y
4
5 # appel de la fonction
6 a = puissance(1+1+1, 2) # affecte 3 puissance 2 = 9 à a
```



Code de la fonction exécuté **seulement** quand on appelle la fonction, pas quand on la définit !

Que se passe-t-il lors de l'appel ?

- 1 Évaluation des valeurs passées en argument de la fonction (ici : 3 et 2)
- 2 Affectation de ces valeurs aux paramètres (ici x et y)
- 3 Exécution du code de la fonction (avec $x = 3$ et $y = 2$)

Comment appeler une fonction

Écrire le nom de la fonction et spécifier les valeurs des paramètres entre parenthèses.

```
1 # définition de la fonction
2 def puissance(x, y):
3     return x ** y
4
5 # appel de la fonction
6 a = puissance(1+1+1, 2) # affecte 3 puissance 2 = 9 à a
```



Code de la fonction exécuté **seulement** quand on appelle la fonction, pas quand on la définit !

Que se passe-t-il lors de l'appel ?

- 1 Évaluation des valeurs passées en argument de la fonction (ici : 3 et 2)
- 2 Affectation de ces valeurs aux paramètres (ici x et y)
- 3 Exécution du code de la fonction (avec $x = 3$ et $y = 2$)
- 4 Récupération de la valeur retournée (si elle existe – ici : 9)

Appel de fonctions (suite)

Quelques conséquences du mécanisme d'appel de fonction :

- ▶ Passer le même nombre de valeurs que le nombre de paramètres attendus
⇒ puissance (x) ?

Quelques conséquences du mécanisme d'appel de fonction :

- ▶ Passer le même nombre de valeurs que le nombre de paramètres attendus
⇒ `puissance(x)` ?
- ▶ Passer les valeurs des paramètres dans le même ordre que dans la définition de la fonction :

`puissance(3, 2)` : $x = 3$ et $y = 2 \implies$ renvoie 3^2

`puissance(2, 3)` : $x = 2$ et $y = 3 \implies$ renvoie 2^3

Quelques conséquences du mécanisme d'appel de fonction :

- ▶ Passer le même nombre de valeurs que le nombre de paramètres attendus
 \implies puissance(x) ?
- ▶ Passer les valeurs des paramètres dans le même ordre que dans la définition de la fonction :

puissance(3, 2) : x = 3 et y = 2 \implies renvoie 3^2

puissance(2, 3) : x = 2 et y = 3 \implies renvoie 2^3

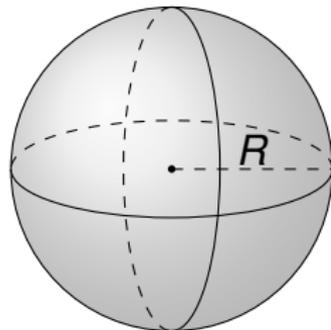
- ▶ passer des expressions comme valeurs des paramètres :

puissance(1 + 2, 2)

puissance(a + 3, b - 1)

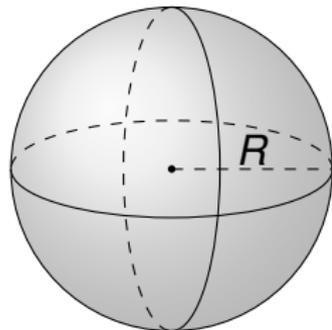
Quelques conséquences du mécanisme d'appel de fonction :

- ▶ Passer le même nombre de valeurs que le nombre de paramètres attendus
 \implies puissance(x) ?
- ▶ Passer les valeurs des paramètres dans le même ordre que dans la définition de la fonction :
puissance(3, 2) : x = 3 et y = 2 \implies renvoie 3²
puissance(2, 3) : x = 2 et y = 3 \implies renvoie 2³
- ▶ passer des expressions comme valeurs des paramètres :
puissance(1 + 2, 2)
puissance(a + 3, b - 1)
- ▶ Passer une variable en argument et lui affecter le résultat de la fonction :
a = puissance(a + 3, b - 1)



▶ Aire de la surface : $4\pi \times R^2$

▶ Volume de la boule : $\frac{4\pi \times R^3}{3}$



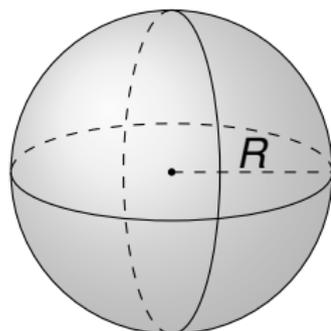
► Aire de la surface : $4\pi \times R^2$

► Volume de la boule : $\frac{4\pi \times R^3}{3}$

```
pi = 3.1415926536
```

```
def aire_surface(R):  
    return 4 * pi * R**2
```

```
def volume_boule(R):  
    return 4 * pi * R**3 / 3
```



► Aire de la surface : $4\pi \times R^2$

► Volume de la boule : $\frac{4\pi \times R^3}{3}$

```
pi = 3.1415926536
```

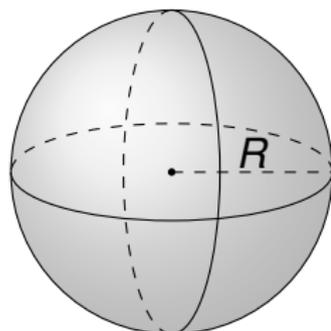
```
def aire_surface(R):  
    return 4 * pi * R**2
```

```
def volume_boule(R):  
    return 4 * pi * R**3 / 3
```

```
# Rayon Terre = 6371km
```

```
aire_terre = aire_surface(6371)  
volume_terre = volume_boule(6371)
```

```
print("Aire =", aire_terre)  
print("Volume =", volume_terre)
```



▶ Aire de la surface : $4\pi \times R^2$

▶ Volume de la boule : $\frac{4\pi \times R^3}{3}$

```
pi = 3.1415926536
```

```
def aire_surface(R):  
    return 4 * pi * R**2
```

```
def volume_boule(R):  
    return 4 * pi * R**3 / 3
```

```
# Rayon Terre = 6371km
```

```
aire_terre = aire_surface(6371)  
volume_terre = volume_boule(6371)
```

```
print("Aire =", aire_terre)  
print("Volume =", volume_terre)
```

```
Aire = 510064471.91144544  
Volume = 1083206916849.273
```

Est-ce que 2 segments $[A, B]$ et $[C, D]$ ont une intersection ?

Est-ce que 2 segments $[A, B]$ et $[C, D]$ ont une intersection ?

- ▶ 2 vecteurs \vec{U} et \vec{V} colinéaires \iff produit vectoriel = 0
- ▶ $\vec{U} = (ux, uy)$, $\vec{V} = (vx, vy) \implies$ prod. vectoriel = $ux \times vy - uy \times vx$

Est-ce que 2 segments $[A, B]$ et $[C, D]$ ont une intersection ?

- ▶ 2 vecteurs \vec{U} et \vec{V} colinéaires \iff produit vectoriel = 0
- ▶ $\vec{U} = (ux, uy)$, $\vec{V} = (vx, vy) \implies$ prod. vectoriel = $ux \times vy - uy \times vx$

```
def prod_vectoriel(ux, uy, vx, vy):  
    return ux * vy - uy * vx  
  
def is_collinear(ux, uy, vx, vy):  
    return prod_vectoriel(ux, uy, vx, vy) == 0
```

Est-ce que 2 segments $[A, B]$ et $[C, D]$ ont une intersection ?

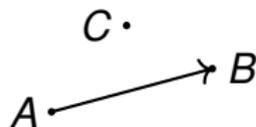
- ▶ 2 vecteurs \vec{U} et \vec{V} colinéaires \iff produit vectoriel = 0
- ▶ $\vec{U} = (ux, uy)$, $\vec{V} = (vx, vy) \implies$ prod. vectoriel = $ux \times vy - uy \times vx$

```
def prod_vectoriel(ux, uy, vx, vy):  
    return ux * vy - uy * vx  
  
def is_collinear(ux, uy, vx, vy):  
    return prod_vectoriel(ux, uy, vx, vy) == 0
```

- ▶ 3 points A, B, C

C strictement à gauche du vecteur \vec{AB}

\iff prod. vectoriel de \vec{AB} et $\vec{AC} > 0$



Est-ce que 2 segments $[A, B]$ et $[C, D]$ ont une intersection ?

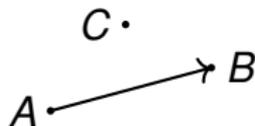
- ▶ 2 vecteurs \vec{U} et \vec{V} colinéaires \iff produit vectoriel = 0
- ▶ $\vec{U} = (ux, uy)$, $\vec{V} = (vx, vy) \implies$ prod. vectoriel = $ux \times vy - uy \times vx$

```
def prod_vectoriel(ux, uy, vx, vy):  
    return ux * vy - uy * vx  
  
def is_collinear(ux, uy, vx, vy):  
    return prod_vectoriel(ux, uy, vx, vy) == 0
```

- ▶ 3 points A, B, C

C strictement à gauche du vecteur \vec{AB}

\iff prod. vectoriel de \vec{AB} et $\vec{AC} > 0$

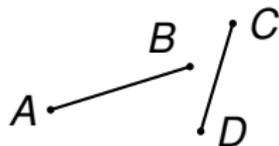
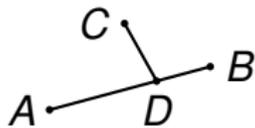
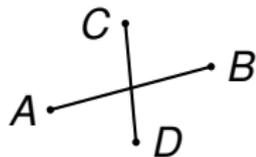


```
def is_left(ax, ay, bx, by, cx, cy):  
    return prod_vectoriel(bx-ax, by-ay, cx-ax, cy-ay) > 0
```

► $[A, B]$ et $[B, C]$ ont une intersection \iff ① et ②

① Exactement un des 2 points C ou D est strictement à gauche de \overrightarrow{AB}

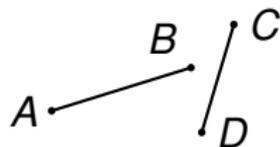
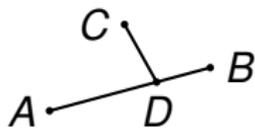
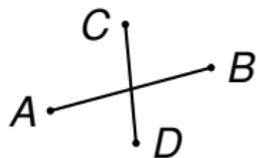
② Exactement un des 2 points A ou B est strictement à gauche de \overrightarrow{CD}



► $[A, B]$ et $[B, C]$ ont une intersection \iff ① et ②

① Exactement un des 2 points C ou D est strictement à gauche de \overrightarrow{AB}

② Exactement un des 2 points A ou B est strictement à gauche de \overrightarrow{CD}

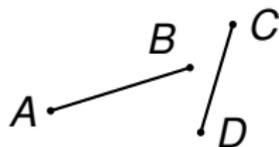
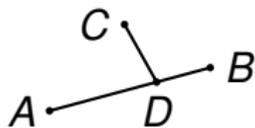
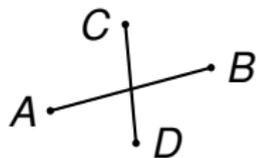


► Exactement un des 2 points \implies XOR

► $[A, B]$ et $[B, C]$ ont une intersection \iff ❶ et ❷

❶ Exactly one of the 2 points C or D is strictly to the left of \overrightarrow{AB}

❷ Exactly one of the 2 points A or B is strictly to the left of \overrightarrow{CD}



► Exactly one of the 2 points \implies XOR

```
def has_intersection(ax, ay, bx, by, cx, cy, dx, dy):  
    left_abc = is_left(ax, ay, bx, by, cx, cy)  
    left_abd = is_left(ax, ay, bx, by, dx, dy)  
    left_cda = is_left(cx, cy, dx, dy, ax, ay)  
    left_cdb = is_left(cx, cy, dx, dy, bx, by)  
  
    return (left_abc ^ left_abd) and (left_cda ^ left_cdb)
```

Est-ce qu'un point C appartient au segment $[A, B]$?

► 3 point $A = (ax, ay)$, $B = (bx, by)$, $C = (cx, cy)$

$$\text{Alors } C \in [A, B] \iff \begin{cases} \vec{AB} \text{ et } \vec{AC} \text{ colinéaires} \\ cx \in [ax, bx] \text{ ou } cx \in [bx, ax] \\ cy \in [ay, by] \text{ ou } cy \in [by, ay] \end{cases}$$

Est-ce qu'un point C appartient au segment $[A, B]$?

► 3 point $A = (ax, ay)$, $B = (bx, by)$, $C = (cx, cy)$

$$\text{Alors } C \in [A, B] \iff \begin{cases} \vec{AB} \text{ et } \vec{AC} \text{ colinéaires} \\ cx \in [ax, bx] \text{ ou } cx \in [bx, ax] \\ cy \in [ay, by] \text{ ou } cy \in [by, ay] \end{cases}$$

```
def is_between(ax, ay, bx, by, cx, cy):  
    cx_between_ax_bx = (ax <= cx <= bx) or (bx <= cx <= ax)  
    cy_between_ay_by = (ay <= cy <= by) or (by <= cy <= ay)  
  
    return (is_collinear(bx-ax, by-ay, cx-ax, cy-ay)  
            and cx_between_ax_bx  
            and cy_between_ay_by)
```

Appel de fonctions (presque fin)

Quel affichage produit ce code ?

```
1
2
3
4 def puissance( x , y ) :
5     return x ** y
6
7 x = puissance( 2 , 5 )
8 print(x)
```

Appel de fonctions (presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance( x , y ) :
5     return x ** y
6
7 x = puissance( 2 , 5 )
8 print(x)
```

Appel de fonctions (presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance( x , y ) :
5     return x ** y
6
7 x = puissance( 2 , 5 )
8 print(x)
```

⇒ On veut que `puissance(2, 5)` renvoie toujours 2^5 !!!

Appel de fonctions (presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance ( x , y ) :
5     return x ** y
6
7 x = puissance ( 2 , 5 )
8 print ( x )
```

⇒ On veut que `puissance (2 , 5)` renvoie toujours 2^5 !!!

Appel de fonctions (presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance ( x , y ) :
5     return x ** y
6
7 x = puissance ( 2 , 5 )
8 print ( x )
```

The diagram illustrates the execution of the code. It shows the following steps:

- Line 1: Variable `x` is assigned the value 3.
- Line 2: Variable `y` is assigned the value 4.
- Line 4: A function `puissance` is defined with parameters `x` and `y`. The function body (line 5) returns the result of `x ** y`.
- Line 7: The function `puissance` is called with arguments 2 and 5. The result of this call is assigned to the variable `x`.
- Line 8: The value of `x` is printed.

⇒ On veut que `puissance (2 , 5)` renvoie toujours 2^5 !!!

Appel de fonctions (presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance ( x , y ) :
5     return x ** y
6
7 x = puissance ( 2 , 5 )
8 print ( x )
```

⇒ On veut que `puissance (2 , 5)` renvoie toujours 2^5 !!!

Visibilité des variables

- ▶ À l'intérieur d'une fonction, les paramètres de celle-ci *masquent* les variables de même nom déclarées en dehors de la fonction (on n'y accède plus).

Appel de fonctions (presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance ( x , y ) :
5     return x ** y
6
7 x = puissance ( 2 , 5 )
8 print ( x )
```

⇒ On veut que `puissance (2 , 5)` renvoie toujours 2^5 !!!

Visibilité des variables

- ▶ À l'intérieur d'une fonction, les paramètres de celle-ci *masquent* les variables de même nom déclarées en dehors de la fonction (on n'y accède plus).
- ▶ Moyen d'y accéder quand même : hors périmètre Ing-info.

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance_x(y):
5     return y ** x
16 print(puissance_x(2))
```

Appel de fonctions (presque presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance_x(y):
5     return y ** x
16 print(puissance_x(2))
```

← x défini ligne 1

→ 8

Appel de fonctions (presque presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance_x(y):
5     return y ** x
6
7 def puissance_x_bis(y):
8     x = 2
9     return y ** x
10
11
12
13
14
15
16 print(puissance_x(2))
17 print(puissance_x_bis(2))
```

← x défini ligne 1

→ 8

Appel de fonctions (presque presque fin)

Quel affichage produit ce code ?

```
1  x = 3
2  y = 4
3
4  def puissance_x(y) :
5      return y ** x
6
7  def puissance_x_bis(y) :
8      x = 2
9      return y ** x
16 print(puissance_x(2))
17 print(puissance_x_bis(2))
```

← x défini ligne 1

← x défini ligne 8 ≠ ligne 1

→ 8

→ 4

Appel de fonctions (presque presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance_x(y):
5     return y ** x
6
7 def puissance_x_bis(y):
8     x = 2
9     return y ** x
10
11
12
13
14
15
16 print(puissance_x(2))
17 print(puissance_x_bis(2))
18 print(x)
```

← x défini ligne 1

← x défini ligne 8 ≠ ligne 1

→ 8

→ 4

Appel de fonctions (presque presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance_x(y):
5     return y ** x
6
7 def puissance_x_bis(y):
8     x = 2
9     return y ** x
10
11
12
13
14
15
16 print(puissance_x(2))
17 print(puissance_x_bis(2))
18 print(x)
```

← x défini ligne 1

← x défini ligne 8 ≠ ligne 1

→ 8

→ 4

→ 3 x défini ligne 1

Appel de fonctions (presque presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance_x(y):
5     return y ** x
6
7 def puissance_x_bis(y):
8     x = 2
9     return y ** x
10
11 def puissance_x_ter(y):
12     print(x)
13     x = 4
14     return y ** x
15
16 print(puissance_x(2))
17 print(puissance_x_bis(2))
18 print(x)
19 print(puissance_x_ter(2))
```

← x défini ligne 1

← x défini ligne 8 ≠ ligne 1

→ 8

→ 4

→ 3 x défini ligne 1

Appel de fonctions (presque presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance_x(y):
5     return y ** x
6
7 def puissance_x_bis(y):
8     x = 2
9     return y ** x
10
11 def puissance_x_ter(y):
12     print(x)
13     x = 4
14     return y ** x
15
16 print(puissance_x(2))
17 print(puissance_x_bis(2))
18 print(x)
19 print(puissance_x_ter(2))
```

← x défini ligne 1

← x défini ligne 8 ≠ ligne 1

← x défini ligne 1

→ 8

→ 4

→ 3 x défini ligne 1

Appel de fonctions (presque presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance_x(y) :
5     return y ** x
6
7 def puissance_x_bis(y) :
8     x = 2
9     return y ** x
10
11 def puissance_x_ter(y) :
12     print(x)
13     x = 4
14     return y ** x
15
16 print(puissance_x(2))
17 print(puissance_x_bis(2))
18 print(x)
19 print(puissance_x_ter(2))
```

← x défini ligne 1

← x défini ligne 8 ≠ ligne 1

← x défini ligne 1

← nouvel x ou x ligne 1 ?

→ 8

→ 4

→ 3 x défini ligne 1

Appel de fonctions (presque presque fin)

Quel affichage produit ce code ?

```
1 x = 3
2 y = 4
3
4 def puissance_x(y):
5     return y ** x
6
7 def puissance_x_bis(y):
8     x = 2
9     return y ** x
10
11 def puissance_x_ter(y):
12     print(x)
13     x = 4
14     return y ** x
15
16 print(puissance_x(2))
17 print(puissance_x_bis(2))
18 print(x)
19 print(puissance_x_ter(2))
```

← x défini ligne 1

← x défini ligne 8 ≠ ligne 1

← x défini ligne 1

← nouvel x ou x ligne 1 ?

→ 8

→ 4

→ 3 x défini ligne 1

→ erreur

Variables globales et locales

Variable globale

- ▶ définie en dehors d'une fonction (via une affectation)
- ▶ existe jusqu'à la fin du programme

Variables globales et locales

Variable globale

- ▶ définie en dehors d'une fonction (via une affectation)
- ▶ existe jusqu'à la fin du programme

Variable locale

- ▶ définie à l'intérieur d'une fonction (via une affectation)
- ▶ créée lors de cette affectation
- ▶ existe jusqu'à la fin de l'exécution de la fonction

Variables globales et locales

Variable globale

- ▶ définie en dehors d'une fonction (via une affectation)
- ▶ existe jusqu'à la fin du programme

Variable locale

- ▶ définie à l'intérieur d'une fonction (via une affectation)
- ▶ créée lors de cette affectation
- ▶ existe jusqu'à la fin de l'exécution de la fonction

Quelle variable x à l'intérieur d'une fonction ?

- ▶ Si x paramètre de la fonction : utiliser ce x
- ▶ Sinon si x variable locale : utiliser ce x
- ▶ Sinon si x variable globale : utiliser ce x
- ▶ Sinon erreur

⇒ **Portée des variables**

Rappel : appel de fonction

- 1 Évaluation des valeurs passées en argument de la fonction
- 2 Affectation de ces valeurs aux paramètres
- 3 Exécution du code de la fonction et retour

Rappel : appel de fonction

- 1 Évaluation des valeurs passées en argument de la fonction
- 2 Affectation de ces valeurs aux paramètres
- 3 Exécution du code de la fonction et retour

⇒ les valeurs en argument sont *copiées dans de nouvelles variables !!!*

Rappel : appel de fonction

- 1 Évaluation des valeurs passées en argument de la fonction
- 2 Affectation de ces valeurs aux paramètres
- 3 Exécution du code de la fonction et retour

⇒ les valeurs en argument sont *copiées dans de nouvelles variables!!!*

```
def ma_fonction(x):  
    x = x+2  
    return x  
  
x = 7  
ma_fonction(x)  
print(x)
```

Rappel : appel de fonction

- 1 Évaluation des valeurs passées en argument de la fonction
- 2 Affectation de ces valeurs aux paramètres
- 3 Exécution du code de la fonction et retour

⇒ les valeurs en argument sont *copiées dans de nouvelles variables!!!*

```
def ma_fonction(x):  
    x = x+2  
    return x  
  
x = 7  
ma_fonction(x)  
print(x)
```

7

Rappel : appel de fonction

- 1 Évaluation des valeurs passées en argument de la fonction
- 2 Affectation de ces valeurs aux paramètres
- 3 Exécution du code de la fonction et retour

⇒ les valeurs en argument sont *copiées dans de nouvelles variables !!!*

```
def ma_fonction(x):  
    x = x+2  
    return x  
  
x = 7  
ma_fonction(x)  
print(x)
```

7

▶ Passage de paramètres ⇒ pas de modif des variables globales

Pourquoi indenter le code de la fonction ?

Qu'affiche le code suivant ? 4 1 1 ou 1 3 1 3 ?

```
1 y = 4
2
3 def ma_fonction(x, y):
4     print(x)
5
6     print(y)
7
8 ma_fonction(1, 3)
9 ma_fonction(1, 3)
```

⇒ Où se trouve la fin de la fonction ? Ligne 4 ou 6 ?

Pourquoi indenter le code de la fonction ?

Qu'affiche le code suivant ? 4 1 1 ou 1 3 1 3 ?

```
1 y = 4
2
3 def ma_fonction(x, y):
4     print(x)
5
6     print(y)
7
8 ma_fonction(1, 3)
9 ma_fonction(1, 3)
```

⇒ Où se trouve la fin de la fonction ? Ligne 4 ou 6 ?

⇒ Sans indentation :
ligne vide = fin

⇒ Code pas lisible

Pourquoi indenter le code de la fonction ?

Qu'affiche le code suivant ? 4 1 1 ou 1 3 1 3 ?

```
1 y = 4
2
3 def ma_fonction(x, y):
4     print(x)
5
6     print(y)
7
8 ma_fonction(1, 3)
9 ma_fonction(1, 3)
```

⇒ Où se trouve la fin de la fonction ? Ligne 4 ou 6 ?

⇒ Sans indentation :
ligne vide = fin

⇒ Code pas lisible

Règle simple

- ▶ Le corps de la fonction est indentée.
- ▶ Fin de l'indentation = fin de la fonction.

③ Les structures de données

- ▶ programmes \implies manipuler des ensembles d'« objets »

- ▶ programmes \implies manipuler des ensembles d'« objets »
- ▶ **Problème** : comment les stocker ?

- ▶ programmes \implies manipuler des ensembles d'« objets »
- ▶ **Problème** : comment les stocker ?
- ▶ **1ère solution** : utiliser plein de variables
 \implies nombre d'objets connus à l'avance
 \implies pas pratique

- ▶ programmes \implies manipuler des ensembles d'« objets »
- ▶ **Problème** : comment les stocker ?
- ▶ **1ère solution** : utiliser plein de variables
 \implies nombre d'objets connus à l'avance
 \implies pas pratique
- ▶ **2ème solution** : structures de données
 \implies solution efficace

Vous avez dit structures de données ?

- ▶ **Problème** : programmes rapides \implies accès rapides aux éléments
 \implies dépend des opérations que l'on exécute

Vous avez dit structures de données ?

- ▶ **Problème** : programmes rapides \implies accès rapides aux éléments
 \implies dépend des opérations que l'on exécute
- ▶ **Opérations usuelles** :

Vous avez dit structures de données ?

- ▶ **Problème** : programmes rapides \implies accès rapides aux éléments
 \implies dépend des opérations que l'on exécute
- ▶ **Opérations usuelles** :
 - ▶ ajouter des éléments

Vous avez dit structures de données ?

- ▶ **Problème** : programmes rapides \implies accès rapides aux éléments
 \implies dépend des opérations que l'on exécute
- ▶ **Opérations usuelles** :
 - ▶ ajouter des éléments
 - ▶ supprimer des éléments

Vous avez dit structures de données ?

- ▶ **Problème** : programmes rapides \implies accès rapides aux éléments
 \implies dépend des opérations que l'on exécute
- ▶ **Opérations usuelles** :
 - ▶ ajouter des éléments
 - ▶ supprimer des éléments
 - ▶ accéder à la valeur d'éléments consécutifs

Vous avez dit structures de données ?

- ▶ **Problème** : programmes rapides \implies accès rapides aux éléments
 \implies dépend des opérations que l'on exécute
- ▶ **Opérations usuelles** :
 - ▶ ajouter des éléments
 - ▶ supprimer des éléments
 - ▶ accéder à la valeur d'éléments consécutifs
 - ▶ accéder à la valeur d'éléments non consécutifs

Vous avez dit structures de données ?

- ▶ **Problème** : programmes rapides \implies accès rapides aux éléments
 \implies dépend des opérations que l'on exécute
- ▶ **Opérations usuelles** :
 - ▶ ajouter des éléments
 - ▶ supprimer des éléments
 - ▶ accéder à la valeur d'éléments consécutifs
 - ▶ accéder à la valeur d'éléments non consécutifs
 - ▶ trier les éléments

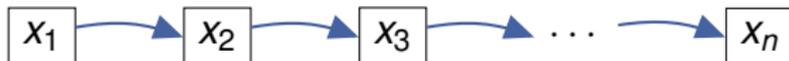
Vous avez dit structures de données ?

- ▶ **Problème** : programmes rapides \implies accès rapides aux éléments
 \implies dépend des opérations que l'on exécute
- ▶ **Opérations usuelles** :
 - ▶ ajouter des éléments
 - ▶ supprimer des éléments
 - ▶ accéder à la valeur d'éléments consécutifs
 - ▶ accéder à la valeur d'éléments non consécutifs
 - ▶ trier les éléments

\implies Peu de structures de données différentes en pratique

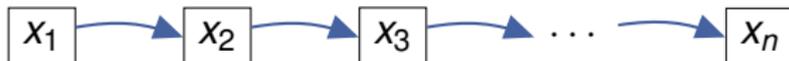
Structures classiques :

► Listes

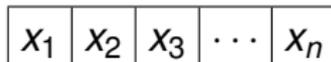


Structures classiques :

► Listes



► Tableaux (1D appelés `List` en Python3)



Structures classiques :

- ▶ Listes
- ▶ Tableaux (1D appelés `List` en Python3)
- ▶ Piles et files

Structures classiques :

- ▶ Listes
- ▶ Tableaux (1D appelés `List` en Python3)
- ▶ Piles et files
- ▶ Tables de hachage (dictionnaires en Python)

Structures classiques :

- ▶ Listes
- ▶ Tableaux (1D appelés `List` en Python3)
- ▶ Piles et files
- ▶ Tables de hachage (dictionnaires en Python)
- ▶ Tuples (n -uplets)

Structures classiques :

- ▶ Listes
- ▶ Tableaux (1D appelés `List` en Python3)
- ▶ Piles et files
- ▶ Tables de hachage (dictionnaires en Python)
- ▶ Tuples (n -uplets)
- ▶ Arborescences, arbres binaires de recherche, les tas

Structures classiques :

- ▶ Listes
- ▶ Tableaux (1D appelés `List` en Python3)
- ▶ Piles et files
- ▶ Tables de hachage (dictionnaires en Python)
- ▶ Tuples (n -uplets)
- ▶ Arborescences, arbres binaires de recherche, les tas

Structures classiques :

- ▶ Listes
- ▶ Tableaux (1D appelés `List` en Python3)
- ▶ Piles et files
- ▶ Tables de hachage (dictionnaires en Python)
- ▶ Tuples (n -uplets)
- ▶ Arborescences, arbres binaires de recherche, les tas

Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein
« Introduction à l'algorithmique », Dunod, 1996

