

<http://deptinfo.unice.fr/~lozes/ISNPython>

# Introduction au langage Python<sub>3</sub>



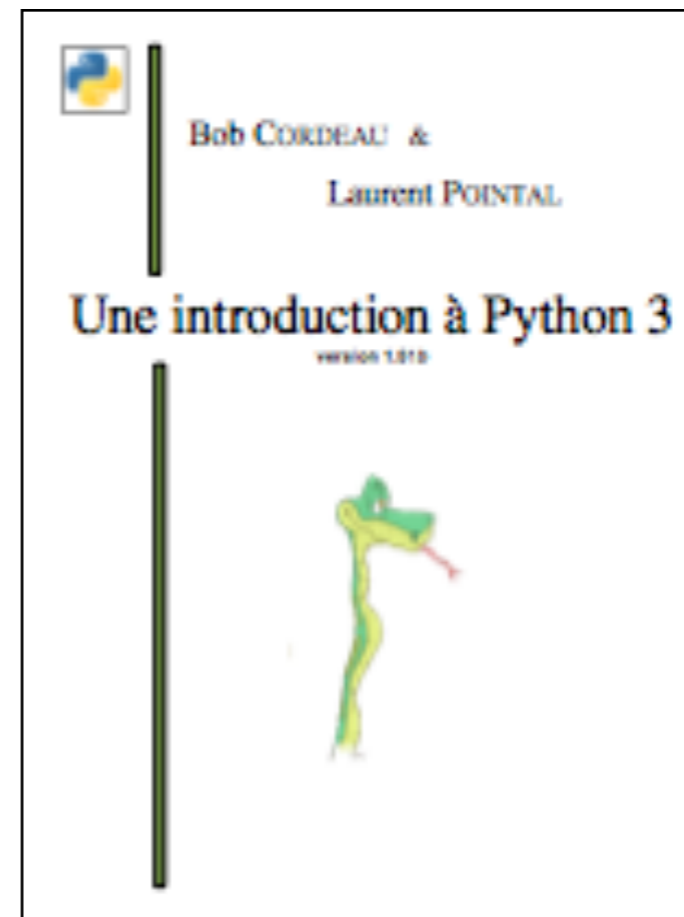
<http://docs.python.org/py3k>



La maison mère



livre gratuit en ligne,  
payant en librairie



polycopié Orsay

# Ressources sur cet enseignement

- Les livres cités à la page précédente, entre autres.
- Le langage support est **Python**, dans sa version **3.6**
- La page Web de la formation :  
`http://deptinfo.unice.fr/~lozes/~ISNPython`
- La documentation exhaustive du langage chez la maison mère :  
`http://docs.python.org/3.6/`
- Les logiciels à télécharger : voir la page Web de la formation...

# Qu'est-ce que la Programmation ?

- L'art de rédiger des textes dans une langue artificielle.



*la science ?*



*les programmes*



*un langage de  
programmation*

- Que fait un programme?

- il **calcule** (simulation, analyse statistique, etc)

- Quel langage?

Fortran, R,...

- il recherche, trie, traite des **informations**  
(bases de données)

Cobol, SQL, SPARQL, ...

- il dialogue avec l'**utilisateur**  
(interfaces graphiques, ampli web)

Java, Javascript, Ruby

- il pilote le **matériel** (système d'exploitation)

C, C++

- il échange avec le **réseau** les 4 « intergiciels » (MPI, RMI, CORBA,...)

# Jouer avec des nombres entiers

- Comme dans toutes les sciences, les nombres jouent un rôle important.

- Python se présente comme une calculatrice interactive à travers son **toplevel**.

Il présente son prompt `>>>` pour que vous lui soumettiez un calcul...

```
Python 3.4.1 Shell
Python 3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 00:54:21)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> 10 + 3
13
>>> 10 + 3 * 5
25
>>> 16 / 3
5.333333333333333
>>> 16 // 3
5
>>> 16 % 3
1
>>> |
```

- Opérateurs de base dans les entiers : `+` `-` `*` `//` `%` `**`

`a // b` fournit le **quotient** de l'entier `a` par l'entier `b`  $\neq 0$ .

`a % b` fournit le **reste** de la division de l'entier `a` par l'entier `b`  $\neq 0$ .

- Si `a` et `b` sont entiers avec `b`  $\neq 0$ , alors :

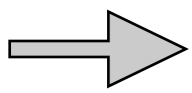
$$a == b * (a // b) + (a \% b) \quad \textit{division euclidienne}$$

- L'opérateur **\*\*** permet de calculer une puissance :

```
>>> 2 ** 10           # le "kilo informatique"
1024
```

- Mal utilisée, une opération peut provoquer une **ERREUR** :

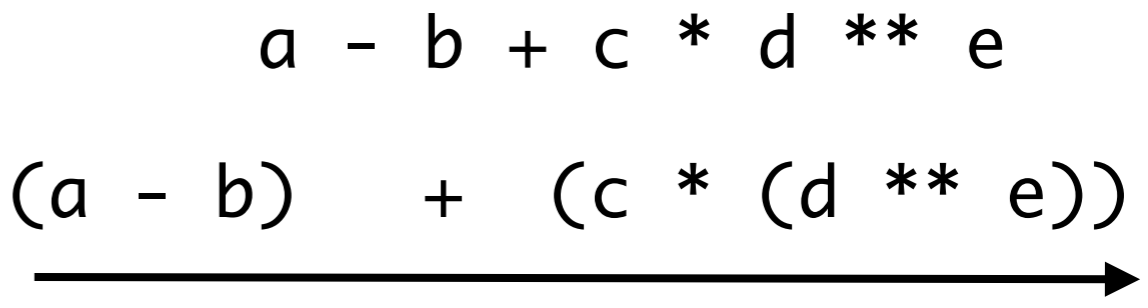
```
>>> 16 % 0
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    16 % 0
ZeroDivisionError: integer division or modulo by zero
```



- L'ordre du calcul d'une expression arithmétique tient compte de la **priorité** de chaque opérateur.

+ -	* // %	**
même priorité (faible)	même priorité (haute)	(très haute)

```
>>> 5 - 8 + 4 * 2 ** 3
29
```





- La taille des entiers n'est limitée que par la mémoire de la machine :

```
>>> 874121921611478384371591419 ** 10  
260447454985660585245180084757921014509252146181223003  
455270044550605023694309556482234857745408080127776885  
578607664767325495386096105286268494775055260671252317  
663749619931275002342815836733596266389781703659821356  
427940431697254607426485624871372306773584664397463801
```

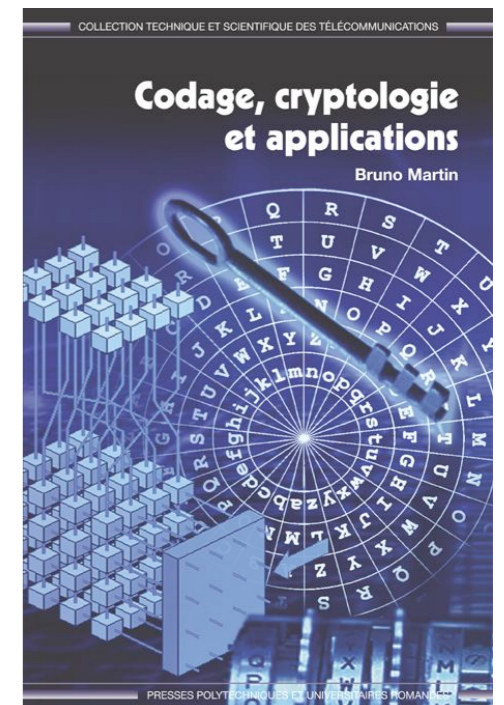
- On dit improprement que l'on travaille avec des nombres entiers *en précision infinie* ! On dit aussi que *le calcul entier est exact*.

- Cette propriété est importante pour les problèmes de cryptologie par exemple (codes secrets). Il est difficile de décomposer :

1527347820637235623261830898781760040741

en produit de facteurs premiers !

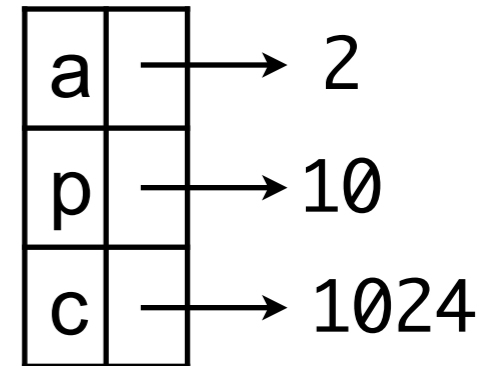
*Et pourtant indication : il n'y en a que deux...*



# Programmer avec des variables

- Une variable est une boîte dans laquelle on peut stocker une valeur

```
>>> a = 2          # lire : a prend pour valeur 2
>>> p = 10         # p prend pour valeur 10
>>> c = a ** p     # c prend pour valeur celle de ap
>>> c
1024
```



*variable = expression*

- Ce signe = non commutatif n'a rien à voir avec celui des maths !
- L'écriture  $2 + 3 = a$  n'aura donc aucun sens !!

```
>>> 2 + 3 = a
SyntaxError: can't assign to operator
```

```
>>> a = 2 + 3
>>> a
5
```

- On dit que  $a = 2 + 3$  est une **instruction d'affectation**.



- Ne confondez pas les **EXPRESSIONS** et les **INSTRUCTIONS**, qui toutes deux vont contenir des variables. Une expression sera *calculée*, tandis qu'une instruction sera *exécutée* !

**Expression**

a vaut 5  
x vaut 3

$a * x ** 2 \longrightarrow 45$

```
>>> a * x ** 2
45
```

**Instruction**

a vaut 5  
x vaut 3

$c = a * x ** 2 \longrightarrow$  *Aucun résultat*

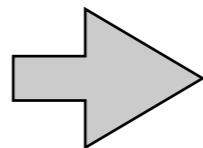
```
>>> c = a * x ** 2
>>>
```

- Une **variable** a le droit de **changer de valeur** !

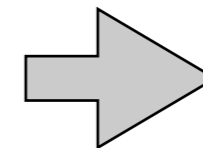
```
>>> a = 2
>>> b = a # la valeur de a est calculée puis c'est elle qui est transmise à b
>>> a = a + 1 # qui se lit : a devient égal à la valeur de a + 1
>>> b
2 # et non 3, ok ?
```

- Exemple : **échange de deux variables**. On utilise une variable temporaire.

```
>>> a = 2
>>> b = 3
```



```
>>> temp = a
>>> a = b
>>> b = temp
```



```
>>> a
3
>>> b
2
```

- L'opérateur d'**égalité** mathématique se note ==

```
>>> a == 3
True
```

```
>>> b == 3
False
```

```
>>> True == False
False
```

- Les valeurs True et False sont les **valeurs booléennes** :

```
>>> a > 3
False
```

```
>>> a + 1 >= 3
True
```

```
>>> a + 1 < 3
False
```

- Dans le contexte d'une expression arithmétique, True == 1 et False == 0. Attention: Python vous laisse mélanger logique et arithmétique!

```
>>> 5 + True
6
```

```
>>> True * False
0
```

```
>>> True == False + 1
True
```

# Affichage de résultats avec `print(...)`

- La fonction `print(...)` permet d'afficher une suite d'expressions :

```
>>> a = 2
>>> print('Le carré de a vaut', a*a, 'et non 5')
Le carré de a vaut 4 et non 5
>>>
```

*espace automatique*

- Ce qui est affiché en bleu n'est pas le résultat de la fonction `print`, mais l'effet de cette fonction. La fonction `print` n'a aucun résultat ! Ou plutôt son résultat est **None**, qui ne s'affiche pas...

```
>>> print(5) == None
5
True
>>> x = None
>>> x
>>>
```

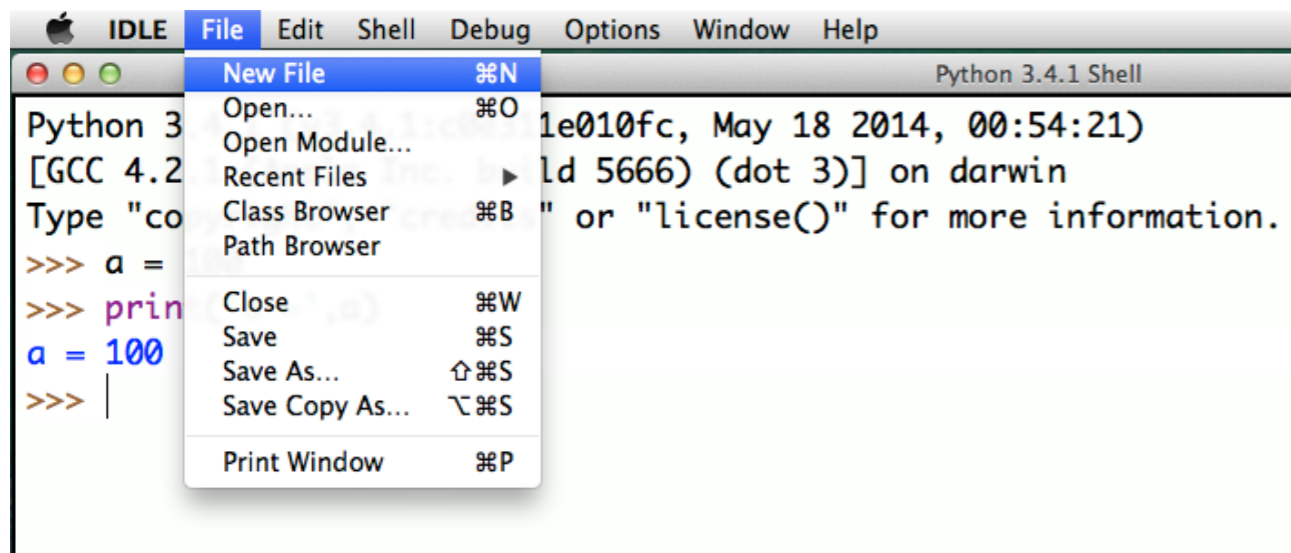
l'effet de `print(5)`

le résultat du test d'égalité

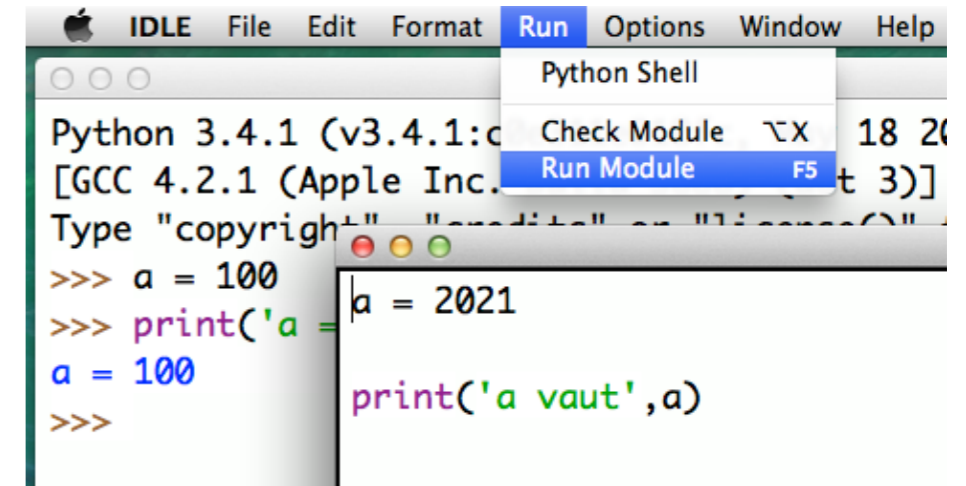
**None** est une valeur spéciale signifiant "*rien*"...

# Rédaction de textes dans l'éditeur IDLE

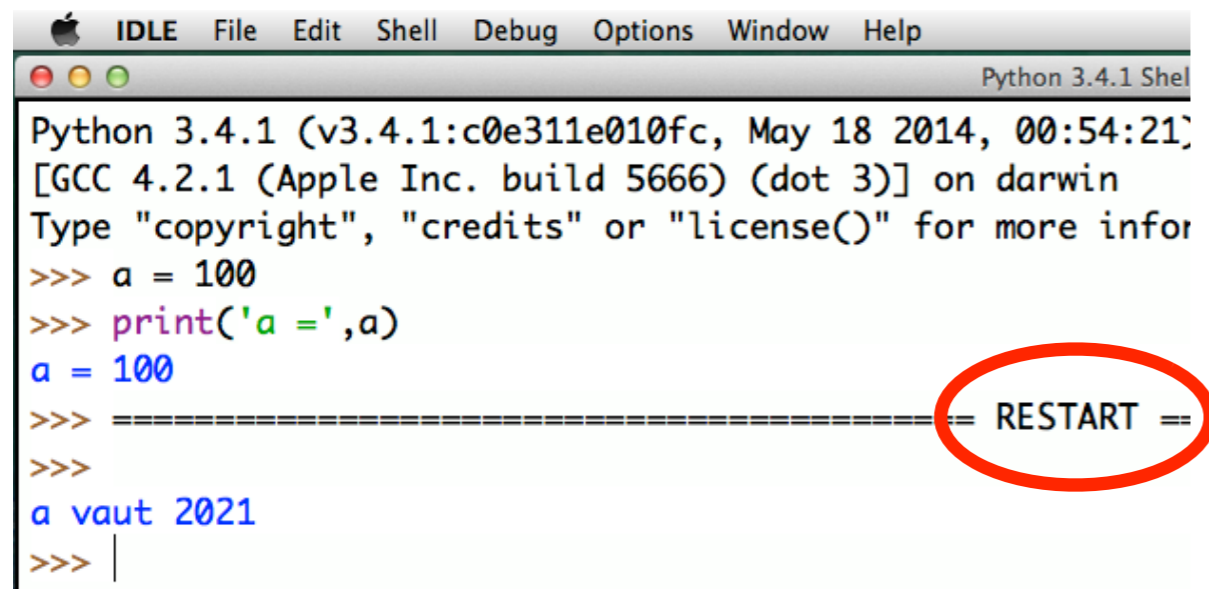
- Le travail interactif au *toplevel* est pratique pour de petits calculs. Mais mieux vaut en général travailler dans *l'éditeur intégré* IDLE.



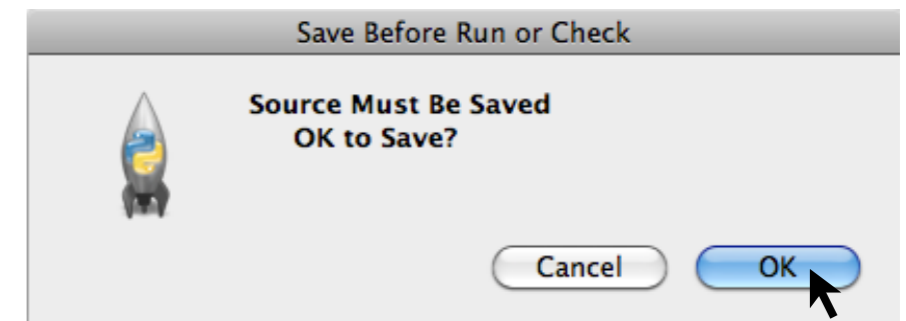
→  
New File



↓  
Run Module



←  
OK  
essai.py

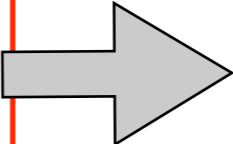


# Conditionnelle : la prise de décisions `if`

- L'instruction conditionnelle `if` permet de prendre une décision :

```
a = -2
if a > 0 :
    print(a, 'est positif')
else :
    print(a, 'est négatif ou nul')
```

*IDLE*



*Run*

-2 est négatif ou nul

- Au toplevel, c'est moins agréable, il faut terminer par une ligne vide.

```
>>> a = -2
>>> if a > 0 :
    print(a, 'est positif')
else :
    print(a, 'est négatif ou nul')
----- ligne vide, OVER, à toi !
-2 est négatif ou nul
```

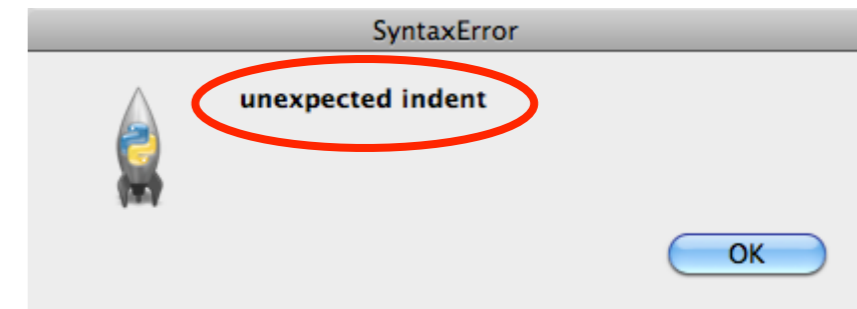
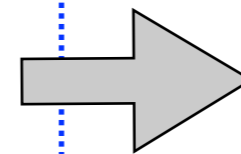
- La bonne distance à la marge d'une ligne (**indentation**) doit être respectée. Elle permet de structurer et de comprendre un programme :

```
| a = -2  
| if a > 0 :  
| ↔ print(a, 'est positif')  
| else :  
| ↔ print(a, 'est négatif ou nul')
```

- Un bloc d'instructions est une suite d'instructions alignées à la verticale. Vous voyez ci-dessus un bloc formé de deux instructions.

```
a = -2  
if a > 0 :  
print(a, 'est positif')  
else :  
    print(a, 'est négatif ou nul')
```

*Run*



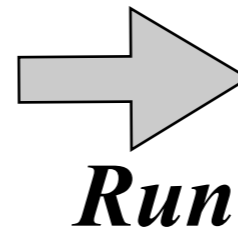
- UNE BONNE **INDENTATION** EST OBLIGATOIRE EN PYTHON.

# Comment définir une nouvelle fonction ?

- Soit à définir la fonction  $f : n \mapsto 2n - 1$

```
def f(n) :  
    return 2 * n - 1  
print('f(5) vaut', f(5))
```

*IDLE*



f(5) vaut 9

- Notez l'indentation (automatique en principe)...
- Le mot **return** signifie "*le résultat est...*". On dit que la fonction **retourne un résultat** (à celui qui a demandé le calcul).
- Il n'est nulle part dit que  $n$  est un entier. On peut aussi utiliser  $f$  sur les réels approchés voire même sur les complexes :

```
>>> f(5)  
9
```

```
>>> f(5.2)  
9.4
```

```
>>> f(2+3j)      # j au lieu de i  
(3+6j)
```



# Les fonctions prédéfinies de Python

- Tous les langages de programmation fournissent un large ensemble de **fonctions** prêtes à être utilisées. Exemple dans les entiers :

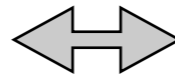
```
>>> 5 * 2 ** 3      # les opérateurs arithmétiques sont des fonctions cachées
40
>>> abs(-5)        # la fonction "valeur absolue" est prédéfinie
5
```

- Certaines fonctions résident dans des **modules** spécialisés, comme `fractions`, `math`... Il faut consulter la documentation en ligne !

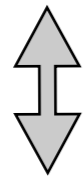
```
>>> gcd(18,12)      # je veux calculer un PGCD
NameError: name 'gcd' is not defined
>>> import fractions # importation du module fractions
>>> fractions.gcd(18,12) # mais le mot gcd reste inconnu !
6
>>> from fractions import gcd # importation du seul mot gcd
>>> gcd(18,12)      # le mot fractions reste inconnu !
6
```

- Comment savoir si deux entiers n'ont que 1 comme diviseur commun ?

```
def premiers_entre_eux(p,q) .IDLE  
    if gcd(p,q) == 1 :  
        return True  
    else :  
        return False
```



```
def premiers_entre_eux(p,q) :  
    return gcd(p,q) == 1
```



```
>>> premiers_entre_eux(21,6)  
False  
>>> premiers_entre_eux(21,8)  
True
```

```
def premiers_entre_eux(p,q) :  
    if gcd(p,q) == 1 :  
        return True  
    return False
```

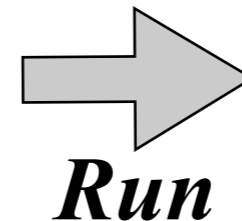
*Le mot clé **return** provoque un échappement. Le reste du texte de la fonction est abandonné !*

- Vous voyez qu'il existe différentes manières de coder une fonction. Elles se distinguent par leur **efficacité**, mais aussi leur **élégance**.

- Comment définir la valeur absolue si elle n'existait pas ?

```
def val_abs(n) :  
    if n > 0 :  
        return n  
    else :  
        return -n  
  
print('|-5| vaut',val_abs(-5))
```

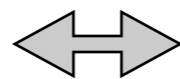
*IDLE*



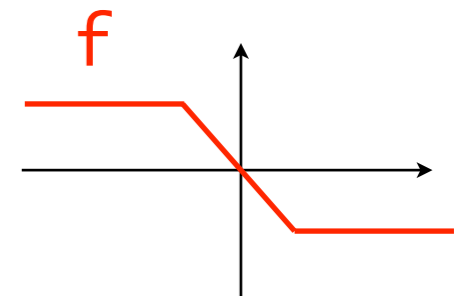
**| -5 | vaut 5**

- Les choix multiples avec **if ... elif ... elif ... elif ... else ...** :

```
def f(x) :  
    if x < -1 :  
        return 1  
    elif x <= 1 :  
        return -x  
    else :  
        return -1
```



```
def f(x) :  
    if x < -1 :  
        return 1  
    else :  
        if x <= 1 :  
            return -x  
        else :  
            return -1
```

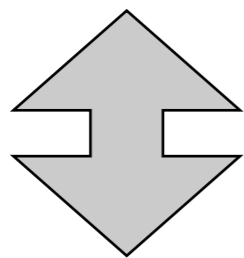


# Le mot-clé **not** désigne la négation logique

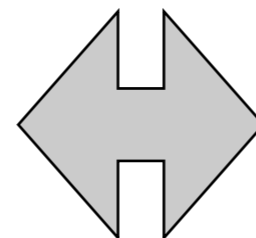
p	True	False
not p	False	True

```
if a > 0 :  
    pass  
else :  
    print('a n'est pas positif')
```

L'instruction qui ne fait rien!



```
if not (a > 0 ) :  
    print('a n'est pas positif')
```



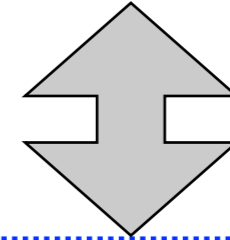
```
if a <= 0 :  
    print('a n'est pas positif')
```

# Les mots-clés **and** et **or** désignent la conjonction et la disjonction

p	True	True	False	False
q	True	False	True	False
p and q	True	False	False	False

conjonction

```
if a > 0 :  
    if b > 0 :  
        print('a et b sont positifs')
```

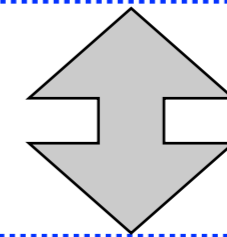


```
if (a > 0) and (b > 0) :  
    print('a et b sont positifs')
```

p	True	True	False	False
q	True	False	True	False
p or q	True	True	True	False

disjonction

```
if a > 0 :  
    print('a ou b est positif')  
elif b > 0 :  
    print('a ou b est positif')
```



```
if (a > 0) or (b > 0) :  
    print('a ou b est positif')
```

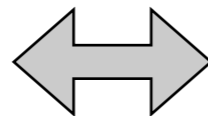
# Évaluation paresseuse de **and** et **or**

```
>>> a = -2
>>> x == 3
NameError: name 'x' is not defined
>>> (a > 0) and (x == 3)
False
```

L'expression `x == 3` n'a pas été évaluée!

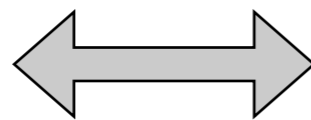
Pour évaluer un **and** ou un **or**, Python commence par évaluer le terme gauche, et ensuite, si c'est nécessaire, le terme droit.

```
>>> (a < 0) or (x == 3)
True
```



```
>>> if a < 0 :
    True
elif x == 3 :
    True
else :
    False
```

```
>>> a < 0 or x == 3
True
```



```
True
```

Note: parenthèses optionnelles! Dans le doute, les mettre...

# Générer des entiers au hasard avec `randint`

- Il faut importer la fonction `randint(...)` du module `random` :

```
from random import randint
```

- Avec  $a \leq b$  entiers, `randint(a,b)` retourne un entier aléatoire de l'intervalle  $[a,b]$ .

↓  
*pseudo-aléatoire !*

- Ex: `2 * randint(0,10)` retourne un entier pair aléatoire de  $[0,20]$ .
- Une fonction `jet35()` sans arguments qui tire 3 ou 5 au hasard :

```
def jet35() :  
    if randint(0,1) == 0 :  
        return 3  
    else :  
        return 5
```

*IDLE*

```
>>> jet35()  
3  
>>> jet35()  
5  
>>> jet35()  
5  
.....
```



# Saisie utilisateur : la fonction `input`

On peut demander à l'utilisateur de saisir un texte, et le sauver dans une variable

```
def perroquet() :  
    txt = input()  
    print(txt)
```

*IDLE*

```
>>> perroquet()  
dis bonjour!  
dis bonjour!  
>>> perroquet()  
ffrfrfrfrf!  
ffrfrfrfrf!  
>>>
```

Si on veut demander un nombre, il faut convertir le résultat de la fonction `input` en un nombre avec `int(...)`

```
def metaucarre() :  
    txt = input('quel n?')  
    n = int(txt)  
    print('n*n =', n*n)
```

*IDLE*

```
>>> metaucarre()  
quel n? 3  
n*n = 9
```

# Jouer à pile ou face

*IDLE*

```
def pile_ou_face() :  
    x = input('choix?')  
    y = randint(0,1)  
    if (x == 'pile' and y == 0) or (x == 'face' and y == 1) :  
        print('gagné!')  
    else :  
        print('perdu!')
```

```
>>> pile_ou_face()  
choix? pile  
perdu!  
>>> pile_ou_face()  
choix? pile  
gagné!  
>>>
```