

# INTRODUCTION AU LANGUAGE **PYTHON**

EN CLASSE DE SECONDE GT





# DÉFINITION

[HTTPS://FR.WIKIPEDIA.ORG/WIKI/PYTHON\\_\(LANGAGE\)](https://fr.wikipedia.org/wiki/Python_(langage))



PYTHON EST UN LANGAGE DE PROGRAMMATION  
**INTERPRÉTÉ** ET **MULTIPLATEFORMES** (...), IL EST DOTÉ  
D'UN **TYPAGE DYNAMIQUE** FORT (...).

LE LANGAGE PYTHON EST PLACÉ SOUS UNE **LICENCE**  
**LIBRE** (...).

IL EST CONÇU POUR OPTIMISER LA PRODUCTIVITÉ  
DES PROGRAMMEURS EN OFFRANT DES OUTILS DE  
**HAUT NIVEAU** ET UNE SYNTAXE **SIMPLE** À UTILISER.

# DÉFINITION (SUITE)

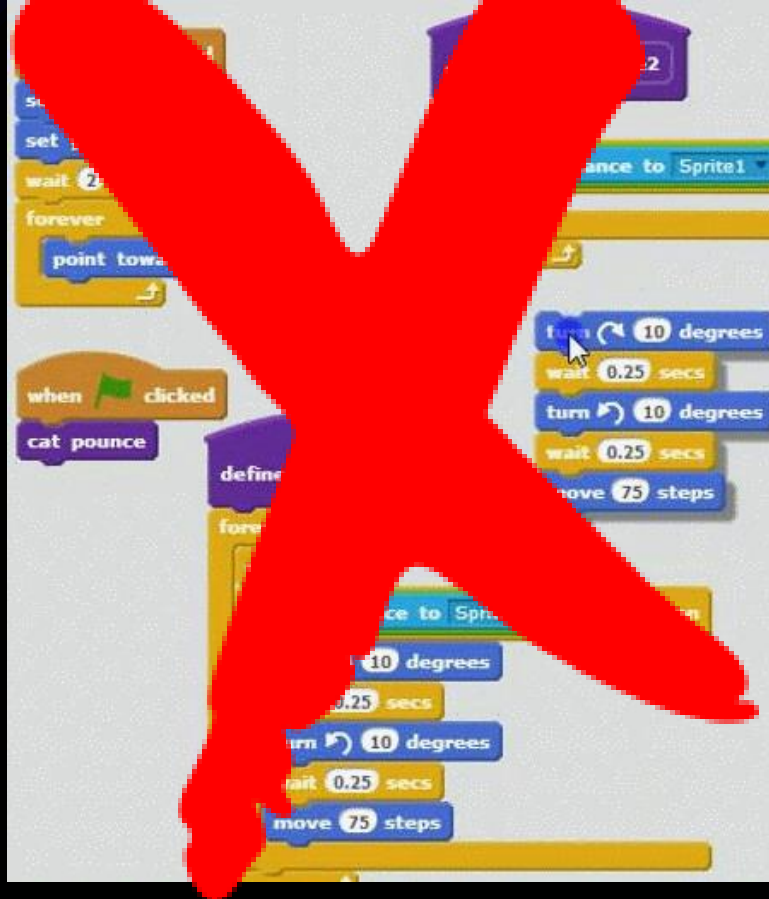
[HTTPS://FR.WIKIPEDIA](https://fr.wikipedia.org/wiki/Python_(langage))

LANGAGE)



Python	
	
<b>Date de première version</b>	20 février 1991 <sup>1</sup>
<b>Paradigmes</b>	Objet, impératif et interprété
<b>Auteur</b>	Guido van Rossum
<b>Développeurs</b>	Python Software Foundation
<b>Dernière version</b>	2.7.17 (19 octobre 2019) 3.8.1 (19 décembre 2019)
<b>Typage</b>	Fort, dynamique, duck typing
<b>Influencé par</b>	ABC, C, Eiffel, ICON, Modula-3, Java, Perl, Smalltalk, Tcl
<b>A influencé</b>	Ruby, Groovy, Boo, Julia
<b>Implémentations</b>	CPython, Jython, IronPython, PyPy
<b>Écrit en</b>	C pour CPython, Java pour Jython, C# pour IronPython et en Python lui-même pour PyPy
<b>Système d'exploitation</b>	Multiplateforme
<b>Licence</b>	Licence libre : Python Software Foundation License
<b>Site web</b>	<a href="http://www.python.org">www.python.org</a>  [archive]
<b>Extension de fichier</b>	py, pyc, pyd, pyo, pyw et pyz 

# « VRAIE » PROGRAMMATION



```
31 def
32 self.file = None
33 self.fingerprints = set()
34 self.logdups = True
35 self.debug = debug
36 self.logger = logging.getLogger(__name__)
37 if path:
38     self.file = open(os.path.join(job_dir, path), 'w')
39     self.file.seek(0)
40     self.fingerprints.update(self.request_fingerprint(request))
41
42 @classmethod
43 def from_settings(cls, settings):
44     debug = settings.getbool('debug')
45     return cls(job_dir(settings), debug)
46
47 def request_seen(self, request):
48     fp = self.request_fingerprint(request)
49     if fp in self.fingerprints:
50         return True
51     self.fingerprints.add(fp)
52     if self.file:
53         self.file.write(fp + os.linesep)
54
55 def request_fingerprint(self, request):
56     return request_fingerprint(request)
```

# PYTHON ET PROGRAMMES SCOLAIRES



Programme de sciences numériques et technologie de  
seconde générale et technologique

## Notions transversales de programmation

Un langage de programmation est nécessaire pour l'écriture des programmes : un langage simple d'usage, interprété, concis, libre et gratuit, multiplateforme, largement répandu, riche de bibliothèques adaptées aux thématiques étudiées et bénéficiant d'une vaste communauté d'auteurs dans le monde éducatif est nécessaire. Au moment de la conception de ce programme, le langage choisi est **Python** version 3 (ou supérieure).

Contenus	Capacités attendues
Affectations, variables Séquences Instructions conditionnelles Boucles bornées et non bornées Définitions et appels de fonctions	Écrire et développer des programmes pour répondre à des problèmes et modéliser des phénomènes physiques, économiques et sociaux.

# PYTHON ET PROGRAMMES SCOLAIRES (SUITE)



 Les publications du Conseil supérieur des programmes (CSP)

Élaboration des projets de programme des enseignements généraux du no  
lycée professionnel

programmes - contenus des enseignements - Publication du CSP - 19/11/2018

## Mathématiques

Classe de seconde, voie professionnelle

En seconde, les élèves passent progressivement de l'utilisation du langage de programmation visuel qu'ils ont utilisé dans les classes antérieures au langage interprété Python. Ce dernier a été choisi pour sa concision, sa simplicité, son implémentation dans de multiples environnements et son utilisation dans l'enseignement supérieur. On ne vise pas la maîtrise d'un langage de programmation ni une virtuosité technique ; la programmation est un outil au service de la formation des élèves à la pensée algorithmique.

### Exemples d'algorithmes et d'activités numériques

- Traduire un programme de calcul à l'aide d'une fonction en Python.
- Calculer les images de nombres par une fonction.
- Déterminer l'équation réduite d'une droite non parallèle à l'axe des ordonnées.
- Rechercher un extremum par balayage sur un intervalle donné.
- Rechercher un encadrement ou une valeur approchée d'une solution d'une équation du type  $f(x) = 0$  par balayage sur un intervalle donné.





# PHILOSOPHIE : « ZEN OF PYTHON »

**Beautiful** is better than ugly.

**Explicit** is better than implicit. **Simple** is better than complex. **Complex** is better than complicated. **Flat** is better than nested. **Sparse** is better than dense.

**Readability** counts. *Special cases* aren't special enough to break the rules.

Although **practicality** beats purity. *Errors* should never pass silently. Unless **explicitly** silenced. In the face of *ambiguity*, **refuse** the temptation to guess. There should be **one** — and preferably only one — obvious way to do it. Although that way may not be obvious at first *unless you're Dutch*. **Now** is better than never. Although never is **often** better than *right* now. If the implementation is *hard* to explain, it's a **bad** idea. If the implementation is *easy* to explain, it may be a **good** idea. **Namespaces** are one *honking great* idea — let's do more of those!

**Beautiful** is better than ugly.  
**Explicit** is better than implicit. **Simple** is better than complex. **Complex** is better than complicated. **Flat** is better than nested. **Sparse** is better than dense. **Readability** counts. *Special cases* aren't special enough to break the rules. Although **practicality** beats purity. *Errors* should never pass silently. Unless **explicitly** silenced. In the face of *ambiguity*, **refuse** the temptation to guess. There should be **one** — and preferably only one — obvious way to do it. Although that way may not be obvious at first *unless you're Dutch*. **Now** is better than never. Although never is **often** better than *right* now. If the implementation is *hard* to explain, it's a **bad** idea. If the implementation is *easy* to explain, it may be a **good** idea. **Namespaces** are one *honking great* idea — let's do more of those!

idea. If the implementation is *easy* to explain, it may be a **good** idea. **Namespaces** are one *honking great* idea — let's do more of those!





# PHILOSOPHIE : « ZEN OF PYTHON » (SUITE)

## PEP 20 : “The Zen of Python”

Ensemble de 19 principes qui influencent le design du langage de programmation Python

1. « Beautiful is better than ugly »  
(...)

➔ indentation comme syntaxe



# INDENTATION COMME SYNTAXE

Définition du terme « **indentation** » :  
**Décalage** d'une partie de texte à droite ou à gauche, par rapport au texte environnant.

Elle se produit en insérant des **espaces** ou des **tabulations**.



NB :

- pas de mélange des deux (espaces/tabulations)
- toujours le même nombre d'espaces

Habitude : **4 espaces**



# INDENTATION COMME SYNTAXE (SUITE)

Dans certains langages, l'indentation est un facteur d'esthétique et de **lisibilité** (*exemple ci-dessous en PHP*) :

```
1 if (victor(human)) { human_wins++; printf("I am  
your humble servant.\n"); } else {  
computer_wins++; printf("Your destiny is under my  
control!\n"); }
```

```
1 if (victor(human)) {  
2     human_wins++;  
3     printf("I am your humble servant.\n");  
4 } else {  
5     computer_wins++;  
6     printf("Your destiny is under my control!\n");  
7 }
```



# INDENTATION COMME SYNTAXE (SUITE)

En Python, le choix a été fait **d'obliger** le développeur à utiliser l'indentation afin de former des blocs, et ainsi écrire du **code lisible** :

```
1 down = 0
2 up = 100
3 for i in range(1,10):
4     guessed_age = int((up+down)/2)
5     answer = input('Are you ' + str(guessed_age) + " years old?")
6     if answer == 'correct':
7         print("Nice")
8         break
9     elif answer == 'less':
10        up = guessed_age
11    elif answer == 'more':
12        down = guessed_age
13    else:
14        print('wrong answer')
```



# INDENTATION COMME SYNTAXE (SUITE)

En Javascript :

```
for (i=0; i<10; i=i+1) {  
    document.write(i);  
}
```

En Python :

```
for i in range(0,10):  
    print(i)
```



# VARIABLES & OPÉRATEUR D'AFFECTATION



Une **variable** est un espace mémoire dans lequel il est possible de placer une **valeur**.

L'opérateur « **=** » permet d'affecter une valeur à une variable. Si la variable n'existe pas, elle est créée lors de cette opération.

exemple : **age = 28**

NB : Python est sensible à la casse (min/MAJ) :  
**age ≠ Age ≠ AGE**

# COMMENTAIRES



Dans le code source d'un programme, il est possible et **conseillé** d'ajouter des commentaires, afin de **décrire** ce code, **faciliter sa compréhension** et justifier certains choix.

En Python, sont considérés comme un commentaire et ne sont donc pas interprétés :

# COMMENTAIRES



Sur une ligne :

tout ce qui suit le caractère #

Sur plusieurs lignes :

tout ce qui se trouve entre les marqueurs  
''' (3 symboles guillemets simples)

et ''' (idem)

# COMMENTAIRES



```
1 age=input("Entrez votre âge ")
2 # on demande la saisie de la variable age
3
4 print("Vous avez", age, "ans")
5
6
7 '''
8 Ce programme permet de demander l'âge
9 d'une personne et de l'afficher dans une phrase
10 '''
```



# LES PRINCIPAUX TYPES DE DONNÉES

types numériques :

type **int** : entier positif ou négatif      3

type **float** : nombre réel      4.5

*NB : le séparateur décimal est le **point***

type **str** : chaine de caractères      “Oui”

type **bool** : “booléen”      True ; False



# LES PRINCIPAUX TYPES DE DONNÉES (SUITE)



type **list** : collections ordonnées d'objets  
séparés par des virgules

```
L=[3, "Bonjour", 4.5, True]
```

type **dict** : dictionnaire

```
d={ "nom": "Durand", "prénom": "Eva",  
    "age": 22, "sexe": 2 }
```

# LES PRINCIPAUX TYPES DE DONNÉES (SUITE)



Utilisation de la fonction Python **type()** pour tester le type d'une donnée :

```
type(d)  
      <class 'dict'>
```

```
type(4.5)  
      <class 'float'>
```

```
type(3)  
      <class 'int'>
```

# INTERACTION AVEC L'UTILISATEUR : LA FONCTION INPUT()



La fonction **input()** permet de demander à l'utilisateur de saisir une valeur.

Depuis Python3, la fonction renvoie toujours un type **string**. Il est parfois nécessaire de convertir cette saisie pour l'exploiter correctement.

```
nom=input("Entrez votre nom :")  
age=int(input("Entrez votre âge :"))
```

# INTERACTION AVEC L'UTILISATEUR : LA FONCTION INPUT() (SUITE)



```
>>> x=input("Entrez une valeur : ")
Entrez une valeur : azerty
>>> type(x)
<class 'str'>
```

```
>>> x=input("Entrez une valeur : ")
Entrez une valeur : 12
>>> type(x)
<class 'str'>
```

```
>>> x=int(input("Entrez une valeur : "))
Entrez une valeur : 12
>>> type(x)
<class 'int'>
```



# OPÉRATEURS DE COMPARAISON

égal à : ==

différent de : !=

inférieur à : <

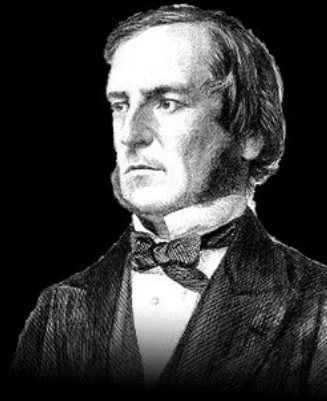
supérieur à : >

inférieur ou égal à : <=

supérieur ou égal à : >=



# OPÉRATEURS LOGIQUES



George BOOLE,  
mathématicien et philosophe britannique,  
19<sup>e</sup> siècle

# OPÉRATEURS LOGIQUES



Un booléen est un type de données qui ne peut prendre que deux valeurs : **vrai** ou **faux**. En Python, elles sont notées **True** et **False**.

opérateur ET : **and**

opérateur OU : **or**

opérateur NON : **not** (contraire de)





# OPÉRATEURS ARITHMÉTIQUES

les quatre opérations élémentaires

+ - x ÷                      + - \* /

quotient de la  
division euclidienne : //

reste de la  
division euclidienne : %

puissance : \*\*

- NB :
- ✓ règles usuelles de priorités
  - ✓ + et \* fonctionnent avec des **str**
  - ✓ **bibliothèques** vite nécessaires

# OPÉRATEURS ARITHMÉTIQUES (SUITE)



```
>>> 6 + 4
```

```
10
```

```
>>> 6 * 5
```

```
30
```

```
>>> 6.2 * 5
```

```
31.0
```

```
>>> 30 / 5
```

```
6.0
```

```
>>> 2 ** 3
```

```
8
```

```
>>> 13 / 5
```

```
2.6
```

```
>>> 13 // 5
```

```
2
```

```
>>> 13 % 5
```

```
3
```

```
>>> sqrt(5)
```

```
NameError: name  
'sqrt' is not defined
```

13		5
		2
3		

NB : espaces non obligatoires





## STRUCTURES ALTERNATIVES (**SI**)

La structure conditionnelle “si” (**if**) permet d'exécuter un bloc d'instructions si et *seulement si* une condition est vérifiée.

```
if condition:  
    instruction1  
    instruction2 (etc.)  
else:  
    instruction3
```

NB : la partie **else** est optionnelle.



# STRUCTURES ALTERNATIVES (SI) (SUITE)

Exemple :

```
1 chaine = input("Note sur 20 : ")
2 note = float(chaine)
3 if note >= 10.0:
4     print("J'ai la moyenne")
5 else:
6     print("C'est en-dessous de la moyenne")
7 print("Fin du programme")
```



# STRUCTURES ITÉRATIVES (**POUR**)

La structure itérative “pour” (**for**) permet de répéter certaines instructions pour chaque élément d'une liste.

```
for i in range(3):  
    print(i)
```

```
>>> for i in range(3):  
    print(i)  
  
0  
1  
2
```

NB1 : la variable peut être quelconque (ici : *i*)

NB2 : variable entière dans l'intervalle [ 0 ; 3 [

# STRUCTURES ITÉRATIVES (POUR)



```
for x in range(200,209):  
    print(x)
```

```
>>> for x in range(200,209):  
      print(x)  
  
200  
201  
202  
203  
204  
205  
206  
207  
208
```

NB : l'intervalle est : [ 200 ; 209 [

# STRUCTURES ITÉRATIVES (**POUR**)



```
for x in range(200,209,2):  
    print(x)
```

```
>>> for x in range(200,209,2):  
    print(x)  
  
200  
202  
204  
206  
208
```

NB : le pas est de 2

# STRUCTURES ITÉRATIVES (POUR)



```
for y in "bonjour":  
    print(y)
```

```
>>> for y in "bonjour":  
        print(y)  
  
b  
o  
n  
j  
o  
u  
r
```

NB : valable pour une chaîne ou une liste

# STRUCTURES ITÉRATIVES (TANT QUE)



La structure itérative “tant que” (**while**) permet de répéter certaines instructions tant qu’une condition est respectée.

```
i=0
while i < 10:
    print(i)
    i=i+1
```

```
>>> i=0
>>> while i < 10:
>>>     print(i)
>>>     i=i+1
0
1
2
3
4
5
6
7
8
9
```

# STRUCTURES ITÉRATIVES : **FOR** OU **WHILE** ?



## **for** :

nombre d'itérations déterminé dès  
l'écriture du programme ;  
incrémentation automatique

## **while** :

condition à vérifier pour continuer les  
itérations ; incrémentation gérée par le  
développeur



# STRUCTURES ITÉRATIVES : **FOR** OU **WHILE** ?



```
>>> for x in range(10):  
        print(x)
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

```
>>> i=0  
>>> while i < 10:  
        print(i)  
        i=i+1
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

# STRUCTURES ITÉRATIVES : COMMENT LES INTERROMPRE ?



**break :**

interrompt une boucle **for** ou **while**

**continue :**

interrompt **une** iteration, mais la boucle continue

*NB : il existe aussi en Python une instruction... qui ne fait rien : **pass** (et elle n'est pourtant pas inutile).*

# STRUCTURES ITÉRATIVES : COMMENT LES INTERROMPRE ?



```
>>> for x in range(10):  
    if x==5:  
        break  
    print(x)
```

```
0  
1  
2  
3  
4
```

```
>>> for x in range(10):  
    if x==5:  
        continue  
    print(x)
```

```
0  
1  
2  
3  
4  
6  
7  
8  
9
```

# FONCTIONS



En programmation, une fonction désigne un « **sous-programme** » permettant d'effectuer des opérations répétitives.

Cela permet d'**alléger** le code et d'éviter les recopies et la maintenance difficile.

On parle de **factorisation** de code.

# FONCTIONS



Syntaxe :

```
def carre(x):  
    return x*x
```

Appel :

```
print(carre(5))
```

*NB : une fonction peut ne prendre aucun paramètre, ou ne renvoyer aucun résultat...*