

# Algorithmique Appliquée

BTS SIO SISR

## Les bases du langage Python

# Plan

- Conditions
- Chaînes de caractères et encodage de caractères
- Entrée et sortie standard
- Boucles "Tant Que"
- Boucles "Pour"
- Discussion sur les différences entre Scratch et Python
- Style, commentaires et PEP 8

# Conditions

# Branche

- Un algorithme doit souvent prendre des **décisions**.
- En fonction de la valeur d'une **expression Booléenne**, l'interpréteur va suivre une **branche** ou une autre.

# Exemple de branchement

- On peut visualiser graphiquement les branches.
- Pseudo-code équivalent :

```
Si la valeur de l'expression Test renvoie Vrai:  
    Exécute le Bloc de code 1  
Sinon:  
    Exécute le Bloc de code 2
```



- En anglais :
  - si ➡ if
  - sinon ➡ else

# Conditions en Python

- La forme de base est la suivante :

```
if Test:  
    Bloc de code 1  
else:  
    Bloc de code 2
```

- Attention aux `:` et à l'**indentation** de 4 espaces.
- Attention à la casse : les mots clés `if` et `else` sont en **minuscule**.

# Le `else` est facultatif

- Une condition peut prendre tout simplement la forme :

```
if Test:  
    Bloc de code # Exécuté si Test == True
```

# Quelques exemples

```
texte = ""  
taille = 175  
  
if taille > 180:  
    texte = "grand"  
else:  
    texte = "petit"  
  
print(texte)
```



petit



# Quelques exemples

```
texte = "petit"  
taille = 175  
  
if taille > 180:  
    texte = "grand"  
  
print(texte)
```



petit

# if imbriqués

```
texte = ""
taille = 175

if taille > 180:
    if taille > 200:
        texte = "très grand"
    else:
        texte = "grand"
else:
    if taille > 155:
        texte = "moyen"
    else:
        texte = "petit"

print(texte)
```

# Plus lisible avec les expressions Booléennes composées

```
texte = ""
taille = 175

if taille > 200:
    texte = "très grand"
elif taille > 180:
    texte = "grand"
elif taille > 155:
    texte = "moyen"
else:
    texte = "petit"

print(texte)
```



moyen

# Expression conditionnelle dense

```
taille = 175  
texte = "grand" if taille > 180 else "petit"  
print(texte)
```



petit

# Chaînes de caractères et encodage de caractères

# Type `str`

- En anglais, **string** signifie **chaîne de caractères**.
- C'est une **liste de caractères**.
- Cette liste commence et termine par `"` (ou `'`).
- Cela n'a *rien* à voir avec certains vêtements... 🙄
- `str` est la contraction de string.
- Exemple :

```
taille = "petit"
```

# Concaténation

```
entree = "avocat"  
plat = "riz"  
dessert = "chocolat"  
espace = " "  
repas = entree + espace + plat + espace + dessert  
  
print(repas)
```



avocat riz chocolat

# Multiplication scalaire

```
a = "a"  
trois_a = 3 * a  
print(trois_a)
```



aaa



# Multiplication ?

```
a = "a"  
a_voir = a * a
```



```
TypeError: can't multiply sequence by non-int of type 'str'
```

# Longueur

`len` est un diminutif de **length**, qui signifie longueur.

```
chaîne = "abcde"  
len(chaîne)
```



5

```
chaîne = "papillon"  
len(chaîne)
```



8

# Indexation

```
chaine = "abcde"  
chaine[0]
```



"a"

```
chaine = "abcde"  
chaine[2]
```



"c"

# Indexation négative

```
chaine = "abcde"  
chaine[-1]
```



"e"

```
chaine = "abcde"  
chaine[-3]
```



"c"

# Indexation en dehors des limites

```
chaîne = "abcde"  
chaîne[5]
```



```
IndexError: string index out of range
```

# Tranche entre 2 bornes

```
debut = 1  
fin = 4  
chaine = "abcde"  
tranche = chaine[debut:fin]  
print(tranche)
```



"bcd"

*Note* : en anglais, on parle de **slicing**.

*Note 2* : la borne de fin est exclue.

# Tranche à partir d'un index

```
debut = 2  
chaine = "abcdefg"  
tranche = chaine[debut:]  
print(tranche)
```



"cdefg"

# Tranche jusqu'à un index

```
fin = 5  
chaine = "abcdefg"  
tranche = chaine[:fin]  
print(tranche)
```



"abcde"



# Tranche non contiguë

```
debut = 1  
fin = 10  
saut = 2  
chaine = "abcdefghijklmnop"  
tranche = chaine[debut:fin:saut]  
print(tranche)
```



"bdfhj"

# Caractères spéciaux

- On utilise le **caractère d'échappement** `\` en préfixe des caractères spéciaux dans les chaînes de caractères.
- Quelques exemples :
  - Guillements : `\"`
  - Tabulation : `\t`
  - Retour à la ligne : `\n`
  - Retour chariot : `\r`
  - Backslash : `\\`

# Exemples de caractères spéciaux

```
print("lapin\rLu")
```



```
"Lupin"
```

```
print("C:\\Users\\mikado\\Documents")
```



```
"C:\Users\mikado\Documents"
```

```
print("\tValeur : \"Zorro\"")
```



```
" Valeur : \"Zorro\""
```

# Conversion vers des nombres

```
chaine = "1234"  
entier = int(chaine)  
entier += 8765  
print(entier)
```



9999

```
chaine = "3.1415"  
reel = float(chaine)  
reel *= 2  
print(reel)
```



6.283

# Conversion depuis des nombres

```
entier = 123  
chaine = str(entier)  
chaine *= 2  
print(chaine)
```




"123123"

# Encodage de caractères : ASCII

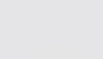




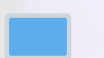
- **ASCII** : American Standard Code for Information Interchange.
- Encodage simple et **compact** : sur 7 bits (moins de 1 octet), on peut avoir jusqu'à  $2^7 = 128$  caractères.
- Chaque caractère est représenté par un nombre entre 0 et 127.
- La **table ASCII** offre une correspondance entre les nombres et leurs caractères associés.

*USASCII code chart*



		Column					Row							
		b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	b <sub>0</sub>	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Row	Column	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	NUL	DLE	SP	0	@	P	\	p					
0	1	SOH	DC1	!	1	A	Q	a	q					
0	2	STX	DC2	"	2	B	R	b	r					
0	3	ETX	DC3	#	3	C	S	c	s					
0	4	EOT	DC4	\$	4	D	T	d	t					
0	5	ENQ	NAK	%	5	E	U	e	u					
0	6	ACK	SYN	&	6	F	V	f	v					
0	7	BEL	ETB	'	7	G	W	g	w					
1	0	BS	CAN	(	8	H	X	h	x					
1	1	HT	EM	)	9	I	Y	i	y					
1	2	LF	SUB	*	:	J	Z	j	z					
1	3	VT	ESC	+	;	K	[	k	{					
1	4	FF	FS	,	<	L	\	l						
1	5	CR	GS	-	=	M	]	m	}					
1	6	SO	RS	.	>	N	^	n	~					
1	7	SI	US	/	?	O	_	o	DEL					

# Limites du ASCII

-  : Bonjour (*Japonais*)
-  : Comment allez-vous ? (*Mandarin*)
-  : شكرا لك : Merci (*Arabe*)
-    : Je ne comprends rien au cours ! (*Emoji*)

# Unicode

- **Unicode** : Universal Coded Character Set.
- L'objectif est de **normaliser** l'encodage de caractères, d'**inclure** un maximum de langues et autres besoins et d'**optimiser** la représentation numérique.
- L'**UTF-8** (pour Unicode Transformation Format - 8 bit) est le plus répandu.
- UTF-8 utilise entre 1 et 8 octets pour représenter jusqu'à 1 112 064 caractères.



# Encodage en Python

- L'encodage par défaut d'un script Python est **UTF-8**.
- On peut changer cela en ajoutant au tout début d'un fichier Python :

```
# -*- coding: ascii -*-
```

- Toutes les `str` d'un script Python utilisent cet encodage par défaut.

# Changer l'encodage à la volée

```
texte_en_utf8 = "Bonjour !"
texte_en_ascii = texte_en_utf8.encode("ascii")
print(texte_en_ascii)
```



b'Bonjour !'

Le préfixe `b` signifie qu'il s'agit d'une chaîne binaire.

# Entrée et sortie standard

# Introduction aux I/O

- En français, on parle d'Entrées/Sorties, soit **E/S**.
- La langue anglaise prédomine en informatique.
- La traduction en anglais d'E/S est **I/O** pour Input/Output.
- On enlève en général le *slash*, ce qui donne **IO**.

# Les fichiers standards

- Les Systèmes d'Exploitation classiques comportent 3 fichiers standards :
  - `stdin` (index 0) : entrée texte standard.
  - `stdout` (index 1) : sortie texte standard.
  - `stderr` (index 2) : sortie d'erreur standard.

# Lire dans `stdin`

- `stdin` est une redirection vers le périphérique clavier.
- Une lecture dans `stdin` signifie donc que l'on va lire ce que l'utilisateur écrit.
- En Python, on utilise la fonction `input` pour lire dans `stdin` :

```
nom = input("Votre nom : ")
```

# Ecrire dans `stdout`

- `stdout` est une redirection vers la console.
- Une écriture dans `stdout` va afficher le contenu dans la console.
- En Python, on utilise la fonction `print` pour écrire dans `stdout` :

```
nom = input("Votre nom : ")  
print(nom)
```

# Ecrire dans `stderr`

- `stderr`, tout comme `stdout`, est une redirection vers la console.
- Une écriture dans `stderr` va afficher le contenu dans la console.
- En Python, lorsqu'une exception est levée, un message d'erreur est affiché dans `stderr` par défaut.
- On peut également utiliser `sys.stderr.write` :

```
import sys
sys.stderr.write("Oh mince ! dit Shipper")
```



# Formattage simple

```
age_en_texte = input("Votre age : ")  
age = int(age_en_texte)  
print("Vous avez", age, "ans")
```



"Vous avez 42 ans"  
(si age == 42 )

# Limites du formattage simple

```
ht = 69.5
tva = 1/5
taxe = round(ht * tva * 100) / 100
ttc = ht + taxe
label_ht = "Prix (HT) : "
label_ttc = "Prix (TTC) : "
print(label_ht, ht, "€\n", label_ttc, ttc, "€")
```



```
Prix (HT) : 69.5 €
Prix (TTC) : 83.4 €
```

# Chaîne de caractères littérale formatée

```
ht = 69.5
tva = 1/5
taxe = round(ht * tva * 100) / 100
label_ht = "Prix (HT) :"
label_ttc = "Prix (TTC) :"
print(f"{label_ht:>12} {ht:.2f}€\n{label_ttc:>12} {ht + taxe:.2f}€")
```



```
Prix (HT) : 69.50€
Prix (TTC) : 83.40€
```

# **TP 03 - Initiation aux Environnements de Développement Intégrés avec pour but de manipuler des chaînes de caractères**



- **Visual Studio Code** est un Environnement de Développement Intégré.
- Edité par Microsoft en JavaScript/Electron.
- Gratuit et Open Source.
- Linux, macOS, Windows.
- Nombreuses extensions.
- **Lien vers le site officiel**

# TP : Usage d'un IDE et manipulation de chaînes

Lien vers le sujet de TP.



```
compressing.py - algo-appliquee - Visual Studio Code
File Edit Selection View Go Run Terminal Help
1d .../00-avant-propos slides.md .../01-intro-programmation slides.md .../02-bases-python M compressing.py x
bin > compressing.py
You, 2 weeks ago | 1 author (You)
1 """ Compress and resize a folder of png images to a folder of jpeg images. You, 2 weeks ago
2 Usage: python3.9 bin/compressimg.py
3 --input=./cours/01-intro-programmation/work-assignment-01/steps
4 --output=./cours/01-intro-programmation/work-assignment-01/distrib
5 """
6
7 import os
8 import click
9 import numpy as np
10 from skimage import io, color, transform
11
12 MAX_HEIGHT=600
13 MAX_WIDTH=800
14 JPEG_QUALITY=75
15
16 def get_target_filenames(files):
17     filenames_no_ext = [ os.path.splitext(file)[0] for file in files ]
18     target_files = [ file + ".jpeg" for file in filenames_no_ext ]
19     return target_files
20
21 def process_one_file(input_dir, input_file, output_dir, output_file):
22     full_input_path = os.path.join(input_dir, input_file)
23     image = io.imread(full_input_path).astype(np.uint8)
24     height, width, _ = image.shape
25     print(f"height: {height}; width: {width}")
26
27     if width > MAX_WIDTH:
28         scale = MAX_WIDTH / width
29         height = round(height * scale)
30         width = MAX_WIDTH
```

# Boucles "Tant que"



# Pourquoi des boucles ?

- Imaginons un programme qui prend en entrée un nombre et doit calculer la somme de 1 à ce nombre :

```
nombre = int(input("Entrez un nombre positif : "))
resultat = 0
if nombre == 1:
    resultat = 1
elif nombre == 2:
    resultat = 1 + 2
elif nombre == 3:
    resultat = 1 + 2 + 3
elif nombre == 4:
    resultat = 1 + 2 + 3 + 4
# etc.
print(resultat)
```



# Pourquoi des boucles ?

On souhaite en fait ici exprimer :

$$\sum_{i=0}^{nombre} i = 0 + 1 + 2 + \dots + (nombre - 1) + nombre$$

Le commentaire `# etc.` dans la diapositive précédente ou l'ellipse `...` dans la formule ci-dessus expriment tous les 2 une **répétition**.

# Définition

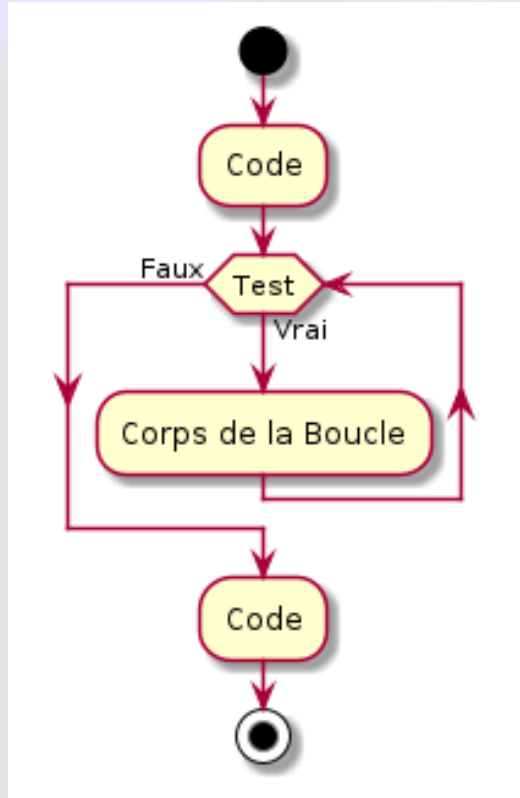
- Une **boucle** permet de répéter un ensemble d'instructions.
- Une répétition de cet ensemble d'instructions s'appelle une **itération**.
- L'ensemble d'instructions à répéter s'appelle le **corps de la boucle**.
- Une boucle s'arrête lorsque sa **condition de fin** devient vraie.

# Boucle "Tant Que"

Une boucle "Tant Que" peut s'exprimer ainsi en pseudo-code :

Tant Que la valeur de l'expression Test est Vraie:  
Exécute le Corps de la Boucle

En anglais : tant que ➡ while



# Exemple en Python

```
nombre = int(input("Entrez un nombre : "))  
  
resultat = 0  
i = 0  
while i <= nombre:  
    resultat += i  
    i += 1  
  
print(resultat)
```



45 si nombre == 9

# Evolution des valeurs

Itération	i	resultat	nombre
0	0	0	9
1	1	1	9
2	2	3	9
3	3	6	9
4	4	10	9
5	5	15	9
6	6	21	9
7	7	28	9
8	8	36	9
9	9	45	9

# Boucle infinie 🦴

```
nombre = 9  
resultat = 0  
i = 0  
while i <= nombre:  
    resultat += i  
    i -= 1
```

# Continue

```
from math import sin

pi = 3.14159265
epsilon = 1e-6
actuel = -2 * pi
fin = 2 * pi
increment = pi / 6
while actuel < fin:
    sin_actuel = sin(actuel)
    precedent = actuel
    actuel += increment

    if abs(sin_actuel) < epsilon: # si sin(actuel) est proche de 0
        continue                # évite les instructions suivantes

    valeur = 1 / sin_actuel
    print(f"x = {precedent:>9.6f} ;"
          f" sin(x) = {sin_actuel:>9.6f} ;"
          f" 1 / sin(x) = {valeur:>9.6f}")
```

# Continue

```
x = -5.759587 ; sin(x) = 0.500000 ; 1 / sin(x) = 2.000000
x = -5.235988 ; sin(x) = 0.866025 ; 1 / sin(x) = 1.154701
x = -4.712389 ; sin(x) = 1.000000 ; 1 / sin(x) = 1.000000
x = -4.188790 ; sin(x) = 0.866025 ; 1 / sin(x) = 1.154701
x = -3.665191 ; sin(x) = 0.500000 ; 1 / sin(x) = 2.000000
x = -2.617994 ; sin(x) = -0.500000 ; 1 / sin(x) = -2.000000
x = -2.094395 ; sin(x) = -0.866025 ; 1 / sin(x) = -1.154701
x = -1.570796 ; sin(x) = -1.000000 ; 1 / sin(x) = -1.000000
x = -1.047198 ; sin(x) = -0.866025 ; 1 / sin(x) = -1.154701
x = -0.523599 ; sin(x) = -0.500000 ; 1 / sin(x) = -2.000000
x = 0.523599 ; sin(x) = 0.500000 ; 1 / sin(x) = 2.000000
x = 1.047198 ; sin(x) = 0.866025 ; 1 / sin(x) = 1.154701
x = 1.570796 ; sin(x) = 1.000000 ; 1 / sin(x) = 1.000000
x = 2.094395 ; sin(x) = 0.866025 ; 1 / sin(x) = 1.154701
x = 2.617994 ; sin(x) = 0.500000 ; 1 / sin(x) = 2.000000
x = 3.665191 ; sin(x) = -0.500000 ; 1 / sin(x) = -2.000000
x = 4.188790 ; sin(x) = -0.866025 ; 1 / sin(x) = -1.154701
x = 4.712389 ; sin(x) = -1.000000 ; 1 / sin(x) = -1.000000
x = 5.235988 ; sin(x) = -0.866025 ; 1 / sin(x) = -1.154701
x = 5.759587 ; sin(x) = -0.500000 ; 1 / sin(x) = -2.000000
```



# Break

```
x = 1000 * 1000 * 1000
while True:
    if (x % 11 == 0) and (x % 27 == 0):
        break
    x -= 1
print(f"{x} est dans la table des 11 et des 27")
```



"999999891 est dans la table des 11 et des 27"

# Boucles imbriquées

```
i = 1
while i < 4:
    line = ""
    j = 1
    while j < 4:
        line += f"{i * j:>3}"
        j += 1
    print(line)
    i += 1
```



1	2	3
2	4	6
3	6	9

# Boucles "Pour" et "Bornes"

# Intérêt

Admettons que nous ayons une liste de noms que nous souhaitons afficher dans la console :

```
liste = [ "Alan", "Ada", "Donald" ]
taille = len(liste)
i = 0
while i < taille:
    nom = liste[i]
    print(nom)
    i += 1
```



```
Alan
Ada
Donald
```

# Simplification

```
for nom in [ "Alan", "Ada", "Donald" ]:  
    print(nom)
```



```
Alan  
Ada  
Donald
```

# Boucle "Pour"

Une boucle "Pour" peut s'exprimer ainsi en pseudo-code :

Pour chaque élément du conteneur :  
Exécute le Corps de la Boucle sur cet élément

En anglais : pour ➡ for



# Autre exemple

```
for i in { 1, 2, 3 }:  
    print(i)
```



```
1  
2  
3
```

# Retour sur les sommes

```
somme = 0  
for i in { 1, 2, 3, 4, 5, 6, 7, 8, 9 }:  
    somme += i  
print(somme)
```



45



## Mais si on ne connaît pas la limite supérieure ?

```
taille = int(input("Taille : "))
somme = 0
# for i in { 1, 2, 3, 4, 5, ..., taille }: # ce code est commenté
#     somme += i                          # comment fait-on ?
print(somme)
```

# Solution : bornes

La fonction `range` permet de résoudre ce problème.

En anglais : `borne` ➡ `range` .

```
taille = int(input("Taille : "))
somme = 0
for i in range(taille + 1):
    somme += i
print(somme)
```

# Autre exemple

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



```
0  
1  
2
```

Autrement dit :  $[0; 3[$

# Bornes de début et de fin

```
debut = 1  
fin = 3  
for i in range(debut, fin):  
    print(i)
```



```
1  
2
```

Autrement dit :  $[debut; fin[$

# Pas

```
debut = 1  
fin = 6  
pas = 2  
for i in range(debut, fin, pas):  
    print(i)
```



```
1  
3  
5
```

# TP 04 - Quelques algorithmes simples pour prendre en main les fondamentaux de l'algorithmique

- *Jupyter Notebook* est supporté par Binder dans votre *navigateur web* et par *Visual Studio Code*.
- Vous avez maintenant pu essayer les deux dans le cadre des précédents TP.
- Vous êtes libres d'utiliser l'environnement de **votre choix** à partir de ce TP.



# **TP : Algorithmes pour l'arithmétique simple**

[Lien vers le sujet de TP.](#)



# Différences entre Python et Scratch

# **Différences concernant les conditions**

# Différences concernant les boucles

Comment feriez-vous pour ré-implémenter le TP 01  
Anjou Vélo Vintage en Python ?

# Style, commentaires et PEP 8

# Que pensez-vous du code suivant ?

```
from math import sin
var1=3.14159265;var2=1e-6;var3=-2*var1;var4=2*var1;var5=var1/6
while var3<var4:
    var6=sin(var3);var7=var3;var3+=var5
    if abs(var6)<var2:continue
    var8=1/var6
    print(f"var7={var7};var6={var6};var8={var8}")
```

➡ Que fait-il ?

# Vous avez déjà vu ce code !

```
from math import sin

pi = 3.14159265
epsilon = 1e-6
actuel = -2 * pi
fin = 2 * pi
increment = pi / 6
while actuel < fin:
    sin_actuel = sin(actuel)
    precedent = actuel
    actuel += increment

    if abs(sin_actuel) < epsilon: # si sin(actuel) est proche de 0
        continue                # évite les instructions suivantes

    valeur = 1 / sin_actuel
    print(f"x = {precedent:>9.6f} ;"
          f" sin(x) = {sin_actuel:>9.6f} ;"
          f" 1 / sin(x) = {valeur:>9.6f}")
```

# Le style, ça compte

- On écrit le code d'abord pour les **êtres humains**.
- Ensuite, on écrit le code pour la machine.
- Le code doit être **simple**, et si possible, **évident**.



# Maintenance

- Un développeur passe **plus de 80%** de son temps à **lire du code** existant.
- Des projets de plus de **100 000 lignes de code** sont courant.
- Imaginez-vous devoir lire, comprendre et corriger des problèmes dans un code **que vous n'avez pas écrit**.
- Imaginez que le développeur initial a quitté l'équipe, voire l'entreprise...

# Une histoire de coûts

- Un code simple à comprendre prendra **moins de temps à faire évoluer.**
- Un code simple comporte en général **moins de problèmes.**
- Le temps passé à comprendre du code et corriger des problèmes génère **un coût pour les entreprises.**
- La mauvaise qualité d'un logiciel impacte **l'image de marque** d'une entreprise.

# Mais les performances ?

- Un code difficile à comprendre ne s'exécute **pas** plus rapidement.
- En général, **80% du temps d'exécution est passé dans 20% du code.**
- On optimise uniquement les 20%...
- **Même les parties optimisées doivent être maintenable.**

# Qu'est-ce qu'un code lisible ?

- **Commentaires** : ils aident à comprendre les parties non triviales.
- **Variables** : elles doivent être bien nommées.
- **Indentation** : Python vous y oblige !
- **Espacement** :
  - *Vertical* pour séparer les blocs de code.
  - *Horizontal* pour séparer les composantes d'une expression.
  - Une expression par ligne.

# Commentaires

```
# Les commentaires commencent par le caractère "#".  
# Ils ne sont pas exécutés par l'interpréteur Python.  
une_ligne = "peut commencer" # par des instructions  
                                # qui s'exécutent et  
                                # se terminer par des  
                                # commentaires.
```

# Bonnes pratiques

- L'une des difficultés du développement logiciel est d'écrire du code simple à comprendre.
- De nombreux outils et techniques visent notamment cet objectif.
- Il existe **de nombreuses autres bonnes pratiques**.
- Nous en mentionnerons quelques unes dans le reste du cours.

# Bonnes pratiques en Python

- Python est un langage piloté par une communauté.
- La communauté écrit des propositions : **Python Enhancement Proposals (PEP)**.
- Les propositions sont discutées puis intégrées dans le langage, afin de l'améliorer.
- L'une des propositions est **PEP 8** et discute du style en Python.



# PEP 8

[Lien vers PEP 8](#)

*Grandes lignes dans les prochaines diapositives*



# Rester consistant

- La règle la plus importante est de **rester consistant**.
- Une base de code écrite par N développeurs doit donner le sentiment qu'elle a été écrite par une seule et même personne.
- Les règles d'entreprise sont prioritaires sur celles de PEP 8.

# Espaces ou Tabulations ?

On utilise 4 espaces pour l'indentation.

# Longueur maximale de ligne

Maximum 79 caractères par ligne au total.

# Règles pour une expression sur plusieurs lignes

- 79 caractères, c'est peu.
- On doit souvent revenir à ligne pour les expressions complexes.
- Exemple :

```
revenus = (salaire_net  
           + interets_comptes_bancaires  
           - impots_revenus  
           - remboursements_emprunt)
```

# Nombreuses autres règles...

- N'hésitez pas à consulter [PEP 8](#).
- Regardez du code écrit en suivant ces règles, par exemple sur [Python.org](#).

# Devoir à la Maison 01

# DM : Retours sur Scratch et Python

[Lien vers le sujet de DM.](#)