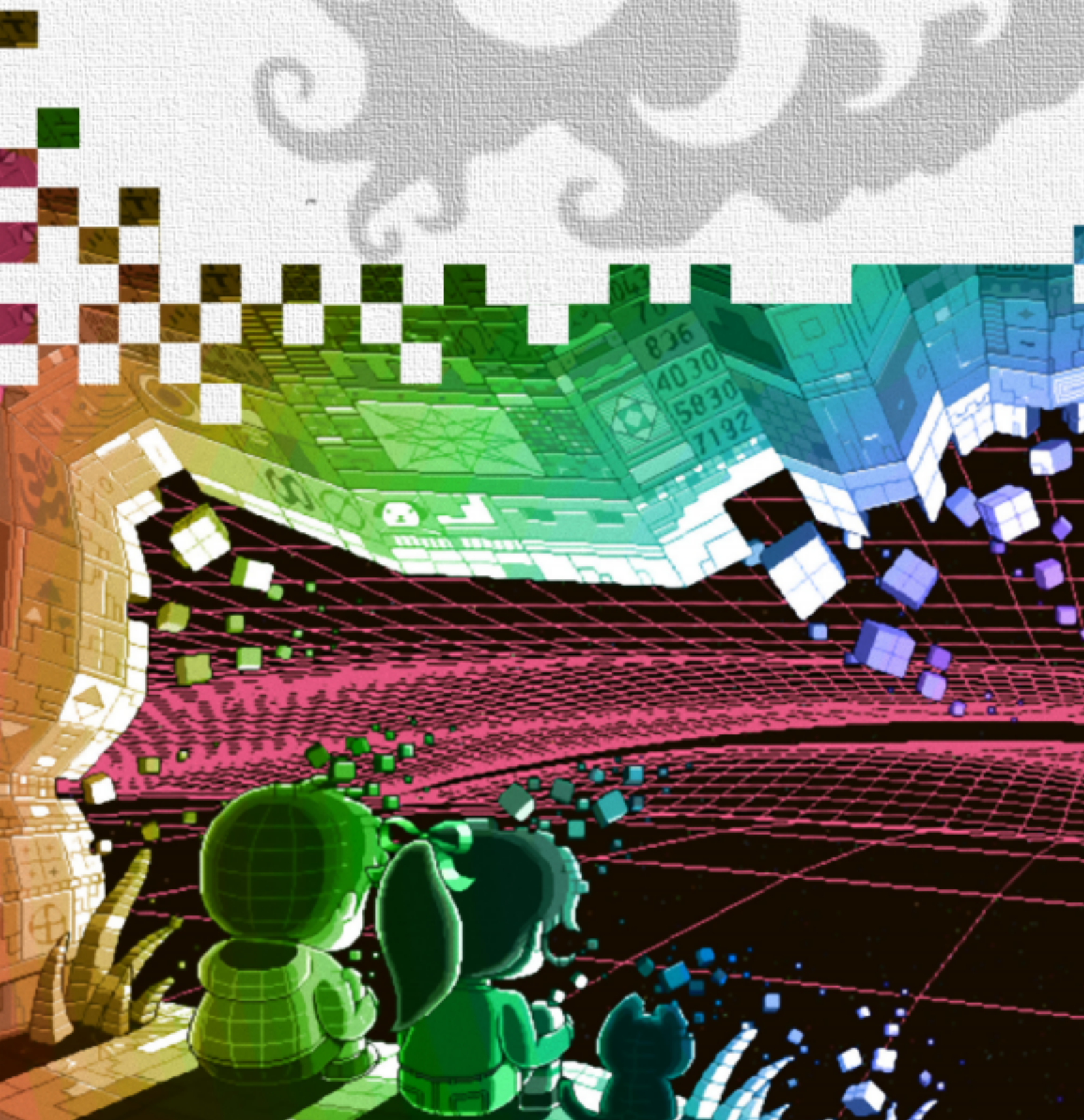


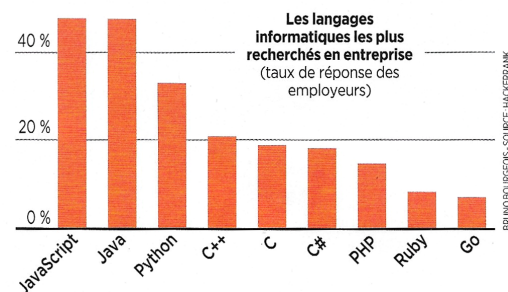
Introduction pour Python 3



Chapitre 1

Introduction à Python

Le langage Python a été créé en 1991 par Guido Von Rossum. C'est un langage de programmation utilisé par les développeurs de "haut niveau" : cela signifie qu'il est proche de la langue naturelle et donc plus facile à appréhender. Il est gratuit et on peut utiliser un interpréteur en ligne ou installé sur le disque dur. Nous utiliserons l'interface de développement IDLE ou THONNY. Les instructions de bases sont données en annexe de ce chapitre.



1.1 Affectation

Affectation Une variable est l'association entre un nom et une valeur. L'affectation est une instruction qui permet de donner une valeur à une variable. Par exemple, vous écrivez $A \leftarrow 5$ en pseudo-code (type BAC). Python utilise le = pour cela : `A=5`.

Typage Toute variable possède un type : entier, chaîne de caractère, entier long, flottant ou booléen pour les **types simples**. Python gère le typage et attribue de façon dynamique ce type à chaque variable.

Toute variable est *in fine* représentée pour le processeur par des octets, et c'est grâce au type de cette variable que le processeur saura si ces octets représentent un entier, un réel, une chaîne de caractères etc. Python est un langage **typé dynamiquement**, le type des variables est déterminé lors de l'interprétation et pourra éventuellement être changé. A contrario, un langage **typé statiquement** comme C/C++ ou Java force à définir le type des variables et à le conserver au cours de la vie de la variable.

Exemples :

```
>>> A = 3
>>> type(A)
<class 'int'>
>>> B = 3.14
>>> type(B)
<class 'float'>
```

```
>>> L= [1, 2, 5,7]
>>> type(L)
<class 'list'>
>>> texte = "bonjour"
>>> type(texte)
<class 'str'>
>>> type(5>4)
<class 'bool'>
```

Cette facilité de Python peut également être source de confusion et il convient d'avoir bien réfléchi aux type des données que l'on utilise. Une expression est une suite de caractères définissant une valeur. Notez qu'**une expression est toujours évaluée pour produire ou retourner une nouvelle valeur**, celle-ci peut-être stockée dans une variable. Un peu comme en maths, les expressions sont évaluées en suivant les priorités des opérateurs, y compris logique comme **not**, **and** et **or** donnés ici dans l'ordre de leur priorité.

A contrario, **une affectation est une instruction** : elle n'a pas de valeur. Voir 1.8.2 pour un complément sur le typage.

1.2 Utiliser des fonctions

Python utilise des fonctions dont la syntaxe est très proche de celle utilisée en math. Une bonne habitude consiste à vérifier toute fonction nouvellement programmée dans le **shell** -appelé également **console** : la fenêtre avec trois chevrons `>>>`). Pour d'autres fonctions voir le Projet Fractales.

1.2.1 Exemple 1

Voici la syntaxe utilisée pour définir une fonction nommée f : le mot clé **def** suivi du nom de la fonction et le mot **return** suivi de l'expression retournée. f prend en paramètre une valeur x et retourne $3x^2$:

```
>>> def f(x):
...     return 3*x**2
...
>>> f(5)
75
```

La **console** ou **shell** permet de visualiser l'interprétation du code. Noter la présence du symbole de multiplication obligatoire hors calcul symbolique (Pour le calcul formel, voir module **sympy**).

Remarque : Les fonctions très courtes comme celle-ci, nommons-la *fonction1*, peuvent-être définies **en une seule ligne** avec le mot clé *lambda*. En voici la syntaxe :

```
fonction1 = lambda x:3*x**2
```

1.2.2 Exemple 2

Dans une micro-entreprise, le bénéfice pour la vente de q objets identiques est donné en centaines d'euro par la fonction B définie par $B(q) = q^3 - 27q^2 + 135q$ sur $[0; +\infty[$. Calculer le bénéfice pour 3 objets, 10 objets.

1. Programmer une fonction B prenant en paramètre le nombre d'objets vendus et renvoyant le bénéfice correspondant.
2. Vérifier les résultats obtenus pour $q = 3$, et $q = 10$.
Un des avantages de passer par une fonction est de pouvoir la réutiliser ailleurs dans le programme.
3. Modifier votre programme précédent pour obtenir la quantité q à partir de laquelle l'entreprise ne fait plus de bénéfice (à cent objets près). *Pour les boucles avec condition d'arrêt, voir 1.4.1.*

1.2.3 Fonction et paramètre

Une fonction peut avoir plusieurs paramètres et retourner plusieurs valeurs.

Exemple :

```
def somme_produit(a, b):
    """
    Calcule la somme et le produit de deux réels a et b
    :param a,b (float):
    :return (float):
    """
    s = a+b
    p = a*b
    return s, p

somme_produit(5, 7)
>>>12,35
```

On peut passer un nombre variable de paramètre à une fonction en utilisant la syntaxe :

```
def f(*param):
```

Les paramètres sont placés dans un **tuple**(voir 1.6.1 :Séquences).

```
>>> def f(*nb_variable_param):
...     print(f'param:{nb_variable_param}')
...
>>> f(5)
param:(5,)
>>> f(5, "chatons")
param:(5, 'chatons')
```

On peut appliquer une valeur par défaut à un paramètre, dans ce cas il peut être omis lors de l'appel de la fonction.

```
>>> def f(x,a=1,b=1,c=2):
...     return a*x**2+b*x+c
...
>>> f(5)
32
>>> f(5, 1,0,0)
25
```

Paramètre ou argument ? On confond souvent les deux bien que les arguments soient en réalité les valeurs passées à la fonction lors de son appel. On utilise parfois les dénominations *paramètre formel* pour le paramètre et *paramètre effectif* pour l'argument .

1.2.4 Variable locale, variable globale

On distingue pour les variables différents espaces : un espace global dans lequel le programme et ses variables globales sont définies et un espace local à chaque fonction, distinct des autres, contenant les paramètres qui lui sont passés et les variables qui y sont définies.

Une variable locale est une variable qui ne peut être utilisée que dans une fonction ou une classe. Celle-ci n'existe que dans la fonction.

Une variable globale est une variable déclarée à l'extérieur d'une fonction ou d'une classe et peut-être utilisée n'importe où dans le programme. En Python, il vous faudra toutefois utiliser le mot clé **global** si vous voulez modifier une variable globale au sein d'une fonction. Il est conseillé d'éviter au maximum les variables globales dans un programme.

Accès et modification

Accès : Les instructions se trouvant à l'intérieur d'une fonction peuvent accéder aux variables définies au niveau principal, mais en consultation seulement : elles peuvent utiliser les valeurs de ces variables, mais pas les modifier (à moins de faire appel à l'instruction **global**). L'utilisation des variables globales si elle est possible est cependant fortement déconseillée (Exemple 1).

Modification : Une fonction ne peut pas modifier par affectation une variable extérieure à son espace (local).

La variable locale n'existe plus après l'appel de la fonction.

Ces variables locales définies à l'intérieur d'une fonction sont inaccessibles aux instructions qui se trouvent à l'extérieur de cette fonction. Cela vous permet d'utiliser les mêmes noms de variables dans différentes parties d'un programme, sans risque d'interférence. L'accès aux variables se fait en priorité par l'espace de nom local, puis par l'espace global (Exemple 2).

Pour décrire la même chose en d'autres termes, nous pouvons dire que **chaque fonction possède son propre espace de noms**, indépendant de l'espace de noms principal.

A l'intérieur d'une fonction, ce sont donc bien les variables locales qui ont la priorité. Il existe donc une sorte de hiérarchie entre les espaces de noms. Nous constaterons la même chose à propos des classes et des objets en classe de *Terminale NSI*.

Exemple 1

```
gravitation = 9.81
def sur_la_lune(masse):
    """
    renvoie le poids sur la lune
    """
    global gravitation
    gravitation = 1.625
    poids= masse * 1.625
    return poids
```

Nous avons bien :

```
print(gravitation)
>>>9.81
```

mais :

```
sur_la_lune(10)
print(gravitation)
>>> 1.625
```

Exemple 1 :Accès à une variable globale dans le corps d'une fonction

```
a = 10
def double():
    return 2*a

>>>double()
20
```

La variable `a` est globale : elle est déclarée à l'extérieur de la fonction `double`. La fonction utilise pourtant bien la valeur de `a`.

Ce code est bien sûr à éviter : il faudrait passer par un paramètre pour utiliser la variable `a`.

Exemple 2 : Modification d'une variable globale Que donne les scripts suivants ?

```
x = 1
def incremente(x):
    x = x+1
    return x
incremente(x)
print(x)
```

```
x = 1
def incremente():
    global x
    x = x+1
    return x
incremente(x)
print(x)
```

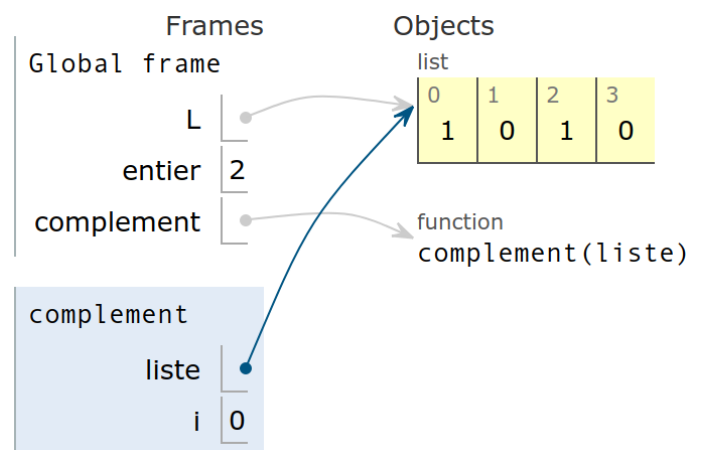
Il serait préférable pour la lisibilité du code de changer le nom du paramètre dans le script 1. Le script 2 est à éviter.

1.2.5 Effet de bord

L'*effet de bord* ou *side effect* est une modification provoquée par une fonction sur une dépendance partagée (variable ou périphérique) autre que celle explicitement spécifiée lors de l'appel de cette fonction.

Exemple :

```
entier = 2
def complement(liste):
    for i in range(len(L)):
        liste[i] = 1-liste[i]
    return liste
L = [1, 0, 1, 0]
>>>complement(L)
[0, 1, 0, 1]
>>>L
[0, 1, 0, 1]
```



La liste `L` est une variable globale, elle a cependant été modifiée par la fonction `complement`. En Python, les listes sont modifiables (ou *mutables*) car elles pointent vers un emplacement mémoire contenant l'objet, ici `[1, 0, 1, 0]`. On peut observer ci-dessus les objets utilisés dans ce script : `L` pointe vers une adresse en mémoire contenant la liste `[1, 0, 1, 0]`. Lorsqu'on utilise `L` comme argument lors de l'appel de la fonction `complement`, le paramètre `liste` pointe également vers cette adresse et va modifier l'objet `[1, 0, 1, 0]` contenu à cet emplacement mémoire. `L` est alors modifiée. Remarquons que `entier` ou `i` sont de type `int` et ne sont pas modifiables : on n'observe pas de pointeur.

Ce comportement n'était cependant pas attendu dans la spécification de la fonction. On indiquera les éventuels effets de bord dans la *docstring*.

1.3 Modularité

Quand on doit élaborer un programme, il est nécessaire de le découper en un maximum de bloc afin de le rendre plus lisible et fonctionnel.

- Concevoir un programme avec plusieurs fonctions permet de vérifier son code au fur et à mesure avec des **jeux de test**. De plus ces fonctions pourront être réutilisées ailleurs dans le programme. On parle de *factorisation* du code.
- Les fonctions les plus utilisées pourront être placées dans des fichiers appelés modules ou bibliothèques.

1.4 Traceback et erreurs

```
>>> def f(x):
...     return 3*x**2
...
>>> g(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'g' is not defined
```

Une erreur s'est produite lors de l'appel de la fonction g , à la ligne 1 : le nom g n'est en effet pas défini. Cela arrive aussi si le code n'a pas été interprété. Le **Traceback** indique la ligne qui a provoqué l'erreur et la nature de l'erreur. Cela permet de détecter rapidement le problème : ne négligez pas la lecture de ce message !

1.5 Instructions conditionnelles

1.5.1 Structure et syntaxe

L'instruction conditionnelle **if** permet de réaliser l'exécution d'un bloc d'instruction à une condition.

Exemples On peut avoir plusieurs conditions, le **else** n'est pas obligatoire.

Exemple 1

```
if n > 0:
    print("positif")
else:
    print("négatif")
```

Exemple 2

```
if x > 0:
    y = 2*x+1
elif x > 3:
    y = 3x-2
else:
    y = 1-x
```

Exemple 3

```
if n%2:
    print("n est pair")
```

- Il peut se présenter plusieurs conditions, dans ce cas, on les enchaîne (Exemple 2).
- On peut imbriquer les conditions si les conditions se cumulent.

Exercices

1. Que réalise les algorithmes ci-dessous ? Programmer ces algorithmes en python en utilisant . éventuellement la tableau des instructions en annexe.

Algorithme_1

```

1 Saisir A et B
2 Si  $A > B$  alors
3   | retourner A
4 sinon
5   | retourner B
6 FinSi

```

Algorithme_2

```

1 Saisir a, b, c, d
2 Si  $a \neq c$  alors
3   |  $S \leftarrow \frac{d-b}{a-c}$ 
4 sinon
5   |  $S \leftarrow \emptyset$ 
6 FinSi

```

2. Programmez une fonction `max2` en Python prenant en paramètre deux nombres A et B, elle retournera la maximum de A et B. Utilisez cette fonction pour déterminer le maximum du bénéfice de **1.2.2**.
En réutilisant votre fonction `max2`, programmez une fonction `max3` prenant en paramètre trois réels A,B,C et retournant le maximum des 3 nombres.
3. Expliquer le rôle de la structure conditionnelle puis programmez une fonction `equation_1_degre` prenant en paramètre quatre réels a,b,c,d et retournant la solution de $ax + b = cx + d$.
4. Programmez une fonction `divise` en Python prenant en paramètre deux nombres A et B, elle retournera `True` si B divise A et `False` sinon.

1.5.2 Compléments sur les booléens**Opérateur logique and, not et or**

Les opérations logiques peuvent être combinées pour écrire une condition avec l'ordre de priorité suivant : `not`, `and`, `or`.

Si P et Q sont deux booléens :

ET logique : `P and Q` est vrai si et seulement si P est vrai et Q est vrai.

OU logique : `P or Q` est vrai si et seulement si P est vrai ou Q est vrai (ou les deux).

Autrement dit, l'un des deux peut-être faux, l'évaluation de P ou Q sera `True`

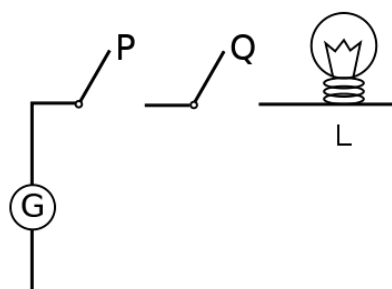
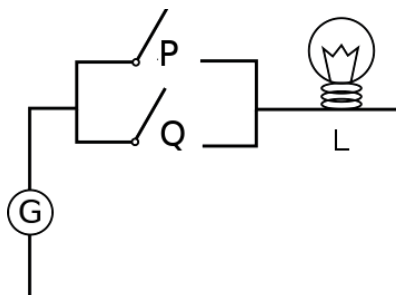
Négation : `not P` est vrai si et seulement si P est faux.

Exemples :

```
>>> 5 != 10 and 8 > 3
False
```

```
>>> 5 != 10 or 8 > 3
True
```

Analogie avec les circuits électriques Voici une représentation sous forme de schéma électrique des fonctions logiques **OU** et **ET** :



Dans quels cas la lampe s'allume-t-elle ?

Tolérance pour les flottants

Les approximations de la représentation des flottants empêchent l'utilisation de l'égalité :

```
>>> 0.3 == 3 * 0.1
False
```

L'égalité sur les flottants est donc avantageusement remplacée par une comparaison à ε près.

Proposez un prédicat (fonctions renvoyant un booléen) qui vérifie si deux valeurs flottantes sont proches. En plus des deux valeurs, le prédicat acceptera donc un troisième paramètre représentant la tolérance acceptable.

1.6 Boucles

1.6.1 Répétition avec condition d'arrêt : la boucle *Tant que*

L'algorithme ci-dessous, écrit en langage naturel renvoie une valeur y pour tout entier naturel n .

1	$i \leftarrow n$
2	$y \leftarrow 0$
3	Tant que $i > 0$
4	$y \leftarrow y + n$
5	$i \leftarrow i - 1$
6	Fin tant que
7	Renvoyer y

1. Tester cet algorithme pour $n=5$ à l'aide d'un tableau. Qu'affiche-t-il en sortie ?

i				
y				

2. Voici sa version en langage *Python*. Programmez-le.

```
i = n
y = 0
while i > 0:
    y = y+n
    i = i-1
print(y)
```

3. Ecrire ce programme avec une fonction `mystere` prenant n en paramètre et retournant y .

1.6.2 Le nombre de répétition est donné : boucle *pour*.

Lorsque le nombre d'itération est connu, on utilise une **boucle bornée** *pour*. La syntaxe de Pour i allant de 1 à n est `for i in range (1, n+1):` .

Exemple :

```
...
1 2 3 4 5 6 7 8 9
>>> for elmt in range (1, 10):
...     print(elmt, end=' ')
```

```
>>> for elmt in range (7):      ...
...     print(elmt, end=' ')    0 1 2 3 4 5 6
```

Exemple :Tableau de valeur d'une fonction L'algorithme suivant affiche une table de valeurs de la fonction carrée pour un pas P .

1	X est un nombre réel et P un entier
2	Saisir P
3	$X \leftarrow 0$
4	Pour i allant de 1 à 10
5	$X \leftarrow X + P$
6	Afficher X^2
7	Fin Pour

Implémentez ce programme en *Python*.

Exemple 2 :Tracer un carré avec le module turtle Tapez le code suivant :

```
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
```

Utilisez une boucle pour factoriser le code de cet algorithme. Programmez une fonction *polygone* prenant en paramètre le nombre de côté et affichant le polygone régulier correspondant.

Exemple 3 :Que dessine le programme suivant ?

```
from turtle import *
for i in range (1, 13):
    forward(150)
    left(150)
exitonclick()
```

Pour *turtle*, voir les annexes 1.7.

Note sur l'importation de module L'instruction `from turtle import *` importe toutes les méthodes du module `turtle`. Cette façon de faire n'est pas recommandée alors que bien souvent seules quelques-unes vous sont utiles.

Préférez dans ce cas : `import turtle`, vous devrez alors préfixer vos méthodes par `turtle` mais pour préserverez l'*espace de nom*.

Un *alias* vous permettra de raccourcir le nom du module à préfixer :

```
import turtle as t
```

Et pour importer une classe ou une fonction de cette lib : `from math import sqrt`.

Exemples

```
>>>import math          >>>import math as m          >>>from math import sqrt
>>>math.sqrt(36)        >>>m.sqrt(36)              >>>sqrt(36)
6                        6                        6
```

Voir 1.8.2

1.7 Conteneurs

Les **conteneurs** sont utilisés pour stocker des objets sous une forme organisée qui suit des règles d'accès spécifiques. Une **séquence est une collection ordonnée d'objets**. Ce sont donc des conteneurs dont les éléments sont toujours dans le même ordre et on peut y accéder grâce à leur *index*. Nous utiliserons les **listes**, les **chaines de caractères** et les **tuples** parfois appelés n-uplets.

Propriétés communes aux séquences

Toutes ces séquences ont des propriétés en commun. Si **seq** est une séquence, alors :

1. **seq[k]** désigne l'élément d'indice k de cette séquence (la numérotation commençant à 0) , l'accès à cet élément se fait en coût constant; il n'est pas plus coûteux d'accéder à **seq[1]** ou **seq [10000]** en Python.
2. on peut effectuer des coupes à l'aide de la technique du slicing;
3. la fonction **len** donne la longueur de la séquence;
4. la concaténation et la duplication se notent respectivement **+** et *****;

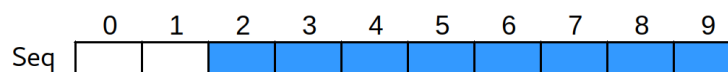
Cette dernière permet de créer des listes vides rapidement :

Les fonctions **str**, **list**, **tuple** permettent de convertir un type de séquence en un autre; par exemple :

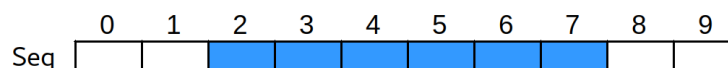
```
>>> list('Commodore')
['C', 'o', 'm', 'm', 'o', 'd', 'o', 'r', 'e']
```

Tranchage (Hors programme) Le tranchage ou *slicing* permet de découper les séquences. Il suffit d'indiquer les indices de début **i**(inclus) et de fin **j** (exclus) : **seq[i:j]**

seq[2:]



seq[2:8]



1.7.1 Listes

Leur **type** est **list**. Les listes sont des séquences **mutables** : on peut leur ajouter des éléments, les modifier. On crée une liste avec des crochets, et on sépare les éléments par une virgule. Les éléments sont indexés à partir de l'index 0. La longueur d'une liste est son nombre d'élément, elle s'obtient avec la fonction **len()**.

Exemple 1

```

>>> L1 = [3, 1, 4, 1, 5, 9, 6, 5, 4]
>>> L1[4]
5
>>> L1.remove(4)
>>> L1
[3, 1, 1, 5, 9, 6, 5, 4]
>>> L2 = ['linux', 'android', 'unix', 'windows', 'macos']
>>> L2[0]
'linux'
>>> L2.pop(3)
'windows'
>>> L2
['linux', 'android', 'unix', 'macos']
>>> L = L1+L2
>>> L
[3, 1, 1, 5, 9, 6, 5, 4, 'linux', 'android', 'unix', 'macos']
>>> L3 = [[1, 2, 3, 4], [0, 0, 0, 0], [-1, 7, 8, 2]]
>>> L3[2][1]
7
>>> L3[0][0]=-5
>>> L3
[[-5, 2, 3, 4], [0, 0, 0, 0], [-1, 7, 8, 2]]
>>> len(L3)
3
>>> len(L3[1])
4
>>> L1.append(10)
>>> L1
[3, 1, 1, 5, 9, 6, 5, 4, 10]
>>> L1=[]
>>> L1
[]

```

Notez la différence entre la modification en place pour les méthodes `pop()`, `append()` ou `remove()` et la concaténation qui nécessite d'utiliser une nouvelle affectation.

Exemple 2 Voici une liste de couleurs pour le module `tkinter`, sur lequel s'appuie `turtle`.

```
couleur = ['blue', 'red', 'green', '#F4FA58']
```

Utiliser cette liste pour construire une frise de **polygone** avec des couleurs choisies de façon aléatoire.



: La dernière couleur est donnée avec son code hexadécimal, aussi utilisé par HTML (voir projet Fractales ou OEDR).

Exemple 3 On peut construire une liste *en compréhension* : c'est une construction propre à Python qui rend le code plus lisible.

```
liste = [<fonction sur item> for <item> in <liste> if <condition>]

>>> Liste_carre = [i*i for i in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
>>> Liste_carre
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> Liste2 = [x**2 for x in [0, 1, 2, 3, 4, 25] if x**2>10]
>>> Liste2
[16, 625]
```

Les listes sont modifiables

Que donne la séquence suivante ?

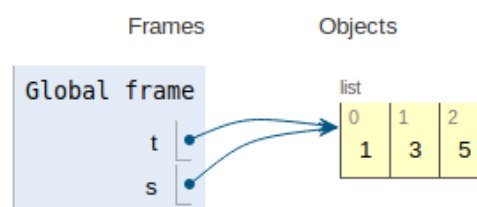
```
t = [1, 2, 3]
s = t
t[1] = 5
>>>t
[1, 5, 3]
>>>s
[1, 5, 3]
```

Le résultat peut paraître surprenant, cela tient au fait que les listes sont des objets modifiables (ou **mutables** mais c'est un anglicisme), si une liste est passée en paramètre, c'est par adresse. Autrement dit, elle va être modifiée par la fonction et rester modifiée en dehors de cette fonction.

Les listes sont des pointeurs ou alias : elles pointent vers une adresse mémoire. Cela permet notamment d'utiliser des objets dont on ne connaît pas la taille.

Note : Lorsqu'une fonction modifie des éléments du programme en dehors de son environnement local, cela s'appelle l'**effet de bord**.

```
s est égal à [1, 3, 5]
t est égal à [1, 3, 5]
```



Les listes s et t pointent vers la même adresse

On utilisera le module `deepcopy` ou la syntaxe `s = t[:]` dans le cas de liste d'objets non mutables pour obtenir une "vraie" copie de `t`.

```
t = [1, 2, 3]
s = t[:]
t[1] = 5
print(t)
print(s)

from copy import *
t = [1, 2, 3]
s = deepcopy(t)
t[1] = 5
print(t)
print(s)
```

Conséquences directes :

Programme principal : Dans une fonction, toute modification sur une liste existante se répercute dans le programme principal.

Valeur de retour : Il est donc inutile de renvoyer la liste.

Variable globale : Il est inutile de préciser dans une fonction qu'une liste est une variable globale.

Exercices

1. Ecrire une fonction `maximum(L)` prenant une liste de nombres en paramètre et retournant le plus grand nombre de la liste.
2. Ecrire une fonction `loto` simulant le tirage du loto : 5 boules parmi les numéros 1 à 49 et un complémentaire entre 1 et 10. Créer une liste de 49 boules, elle sera passée en paramètre à la fonction `loto`.
3. Ecrire une fonction `tableau_aleatoire(n , a, b)` qui retourne un tableau de taille `n` contenant des entiers tirés au hasard entre `a` et `b`.
4. Ecrire une fonction `echange(tab, i, j)` qui échange dans le tableau `tab` les éléments d'indice `i` et `j`.
5. Reprendre la fonction `maximum` pour qu'elle retourne un tuple constitué du maximum et de son indice dans la liste.
6. Gérer avec une assertion le cas où la liste est vide (programmation défensive).

1.7.2 Chaîne de caractères

Leur **type** est `str`. Elles ressemblent aux listes mais ne sont pas modifiables, ainsi toutes les méthodes modifiant les listes ne leur sont pas applicables. Elles sont déclarées entre guillemets (double quotes) ou apostrophes (simple quote). La concaténation `+`, la multiplication `*` et le slicing `[index1:index2]` s'effectuent comme pour les listes. `n` permet d'aller à la ligne.

```
>>> Texte1 = "Atari"
>>> Texte1.lower()
'atari'
>>> type(Texte1)
<class 'str'>
>>> Texte1[3]
'r'
>>> Texte1[1:4]
'tar'

>>> Texte2 = 'Commodore'
>>> Texte2.upper()
'COMMODORE'
>>> Texte = Texte1+Texte2
>>> Texte
'AtariCommodore'
>>> Texte = Texte*2
>>> Texte
'AtariCommodoreAtariCommodore'
```

Remarque Les *f-string* depuis Python 3.5 sont très commodes pour formater les chaînes de caractères. Leur syntaxe est la suivante :

```
f"texte entre guillemets ou quote, {variables entre accolades}"
```

Exemple :

```
>>> var = 3
>>> print(f'Je viens vers {var} heures' )
Je viens vers 3 heures
```

1.7.3 Tuples ou n-uplets

Leur **type** est **tuple**. Les tuples sont également des séquences mais ne sont pas **modifiables**, on dit qu'ils sont **non mutables**.

La déclaration d'un tuple se fait à l'aide de parenthèses. On accède à ses éléments de la même manière qu'avec une liste.

Exemples :

```
>>> T = ('blue', 'red', 'green', '#F4FA58')
>>> T[2]
'green'
>>> T.append('purple')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Les tuples sont souvent utilisés comme résultat d'une fonction.

1.7.4 Parcourir une séquence

Listes, chaîne de caractères ou tuples sont *itérables* : on peut les parcourir avec une boucle **for**.

Exemple Avec le mot clé **in** :

```
>>> liste_pi = [3, 1, 4, 1, 5]
>>> for i in liste_pi:
...     print(i, end=" ")
...
3 1 4 1 5
```

En utilisant les indices :

```
>>> tuple_pi = (3, 1, 4, 1, 5)
>>> for i in range(len(tuple_pi)):
...     print(tuple_pi[i], end=" ")
...
3 1 4 1 5
```



: On *préférera* utiliser des éléments de même type dans une liste. Dans un tuple, ceux-ci pourront avoir une nature différente.

1.7.5 Dictionnaires

Les dictionnaires appelés aussi tables de hachage sont des conteneurs mais pas des séquences : **chaque élément est indexé par une clé. Cette clé ne pas peut-être mutable**. Comme les listes, les dictionnaires sont des **objets** mutables.

Exemples

```
>>> DICT = {'abscisse':2, 'ordonnee':5}
>>> DICT['ordonnee']
5
>>> perso={"prenom": "Rick", "age": 70}
>>> perso
{'prenom': 'Rick', 'age': 70}
>>> perso['nom'] = 'Sanchez'
>>> perso
{'prenom': 'Rick', 'age': 70, 'nom': 'Sanchez'}
```

On peut itérer sur les clés, les valeurs ou les couples clé/valeur d'un dictionnaire :

Itérer sur les clés :

<pre>>>> for cle in perso.keys(): ... print(cle) ... prenom age nom</pre>	<pre>>>> for cle in perso: ... print(cle) ... prenom age nom</pre>
--	---

Itérer sur les valeurs ou les couples :

<pre>>>> for valeur in perso.values(): ... print(valeur) ... Rick 70 Sanchez</pre>	<pre>>>> for (cle,valeur) in perso.items(): ... print(cle,">", valeur) ... prenom > Rick age > 70 nom > Sanchez</pre>
---	--

L'instruction `len(dictionnaire)` donne le nombre de clé/couple. `del('cle')` efface le couple correspondant.

```
>>> len(perso)
3
>>> del(perso['nom'])
>>> perso
{'prenom': 'Rick', 'age': 70}
```

Exemples Les langages de programmation proposent souvent des instructions pour gérer les cas où se présentent plusieurs conditions. Par exemple, l'instruction `switch` en *C* ou `case` en *Pascal*. Cela n'existe pas en *Python*, mais on peut avantageusement utiliser les dictionnaires. Rappelons que :

`dico.get(key, autre_fonction)`

retourne la valeur associée à la clé `key`, et permet d'exécuter la fonction `autre_fonction` si la clé n'existe pas (on peut éventuellement écrire n'importe quelle valeur qui sera alors une valeur par défaut).

Voici un exemple où l'utilisateur entre un nom de système d'exploitation, le programme affiche la machine associée.

```
def MacOS():
    print("C'est un Mac")

def PC():
    print("C'est un PC")

def mobile():
    print("C'est un appareil mobile")

dico_systeme = {'MacOs':fonctionMacOS, 'Windows':fonctionPC, 'Linux':fonctionPC}
dico_systeme.get(entree_utilisateur, fonction_mobile)()
entree_utilisateur = 'Linux'
```

>>>C'est un PC

Notez l'absence de parenthèses pour ne pas appeler les fonctions à cet endroit. Les lignes :

```
dico_systeme = {'MacOs':fonctionMacOS, 'Windows':fonctionPC, 'Linux':fonctionPC}
dico_systeme.get(entree_utilisateur, fonction_mobile)()
```

remplace la séquence d'instructions :

```
if entree_utilisateur == "Linux":
    fonctionPC()
elif entree_utilisateur == "Windows":
    fonctionPC()
else:
    fonction_mobile()
```

1.8 ANNEXES

1.8.1 Instructions basiques

Pseudo-code	PYTHON
Saisir A (non entier)	<code>A = float(input("A ?"))</code>
Saisir A (entier)	<code>A = int(input("A ?"))</code>
Afficher A	<code>print(A)</code>
$A \leftarrow 3$	<code>A = 3</code>
Pour i allant de 1 à 10 <i>Bloc d'instructions</i> Fin Pour	for i in range (1,11) : Bloc d'instructions
Si <i>condition</i> Alors <i>Action 1</i> Sinon <i>Action 2</i> Fin Si	if <i>condition</i> : Bloc d'instructions else : Bloc d'instructions
Tant que <i>condition</i> <i>Action</i> Fin Tant que	while <i>condition</i> : Bloc d'instructions
Définition d'une fonction <i>f</i> de paramètre <i>x</i> qui renvoie <i>y</i>	def fonction(x) : y = return y
N entier aléatoire entre 1 et 6	import random as r N = r.randint(1,6)
Test $A \neq B$	<code>A != B</code>
Test $A = B$	<code>A == B</code>
Quotient entier de A par B	<code>A // B</code>
x^y	<code>x ** y</code>
reste de la division de a par b	<code>a % b</code>



: *Condition* et *Action* doivent être remplacées par les conditions et actions à réaliser dans chaque algorithme.

1.8.2 Typage des variables

Python ne connaît pas à priori le type des variables qui sont définies. Les variables ne sont pas typées : elles pointent vers des valeurs qui sont associées à un type. Chaque objet en Python est représenté sous la forme d'une structure C, cela prend quatre fois plus de place en mémoire pour représenter un double flottant en Python qu'en C.

- On peut dire que la seule "déclaration" de type en python est la chaîne de caractère à l'aide de guillemets :

```
texte = "Bonjour !"
```

Cela explique pourquoi la saisie d'une valeur nécessite de préciser le type de la variable. Par exemple si n est un nombre entier (integer) : La bonne méthode reste l'utilisation d'une fonction.

```
n=int(input(\Entrez une valeur pour n": ))
```



:Depuis la version 3.5 de Python, vous avez la possibilité de fournir des annotations de type lors de la définition de fonctions. Cela n'a rien d'obligatoire mais peut rendre le code plus lisible.

```
def ajouter_deux(x:int) -> int:
    return x+2
```

— En Python on utilise `**` pour le symbole puissance, par exemple x^3 s'écrit `x**3`.

Nom de variables Les noms de variables doivent obéir à quelques règles :

- Le nom doit être explicite.
- Il ne doit pas comporter d'espace.
- On peut utiliser les lettres non accentuées, les chiffres, le tiret du 8 _ .
- Le premier caractère du nom ne doit pas être un chiffre et sauf cas particulier, éviter le tiret du 8.
- Python est sensible à la casse : `Mvariable` et `mvariable` ne sont pas la même variable.
- Ne pas utiliser de mots clé du langage.

Mot clés en Python 3 : `and`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `while`, `yield`, `True`, `False`, `None`.

1.8.3 Modules

Des *modules* peuvent être utiles comme `math`, `turtle` ou `random`. Un module est une collection de fonctions déjà programmées. Il suffit d'écrire au début du code :

```
>>>from math import *      >>>from math import sqrt >>>import math as m
>>>sqrt(25)                >>>sqrt(25)              >>>m.sqrt(25)
5                          5                          5
```

Import complet de la librairie.	Import de la fonction <code>sqrt</code> uniquement.	<code>math</code> est importé avec un alias
---------------------------------	---	---

Privilégier la dernière version pour préserver l'espace de nom. NOTer que :

- tout le code à l'intérieur d'un module est exécuté lors du premier import ;
- ce code n'est pas ré-exécuté si le module a déjà été importé auparavant.

1.8.4 Module Turtle

Le module *turtle* est une librairie de tracé de figures simples, il a l'avantage de pouvoir être utilisé avec très peu de connaissances. Un curseur (la tortue) effectue le tracé à l'écran en se déplaçant selon les instructions codées par l'utilisateur. L'entête de votre code Python devra comporter l'importation de ce module :tapez la ligne

```
from turtle import*
```

- `forward(100)` avance le curseur de 100 pixels.
- `backward(90)` recule le curseur de 90 pixels.
- `left(90)` tourne le curseur de 90° vers la gauche.
- `goto(x,y)` déplace le curseur au point de coordonnées (x,y) .
- `color('blue')` affiche le tracé en bleu.
- `width(2)` affiche le tracé avec un trait de largeur 2 pixels.
- L'adresse suivante vous fournira l'ensemble des instructions disponibles :
http://fr.wikibooks.org/wiki/Programmation_Python/Turtle

