TYPES DE DONNÉES

Avertissement : *Ce chapitre peut être sauté en première lecture pour être lu de façon fractionnée en fonction des besoins.*

Sous Python, chaque objet possède un **type** qui désigne la nature de cette objet. Une des premières choses que vérifie Python lorsque vous lui soumettez un code est que vous ne mélangez pas les torchons et les serviettes. N'essayez donc pas d'additionner des entiers avec des chaînes de caractères : il refusera.

En Python, une variable est **typée**. Son **type** correspond à sa nature : un entier, un flottant, une chaîne de caractère. Lors de l'exécution du programme, le langage vérifie d'abord les types de données avant d'effectuer les opérations.
Python possède un *typage dynamique*, cela signifie que le type de chaque variable est deviné par la machine au moment de l'exécution et vous n'avez pas besoin de le déclarer vous même. C'est plus rapide et moins lourd à écrire, mais demande de faire plus attention, d'être rigoureux et de toujours **se poser la question de la nature des objets que l'on manipule** (comme en mathématiques, vous le verrez plus tard).
Le type d'une variable peut être modifié au cours du programme comme on l'a montré dans l'exemple plus haut.

A contrario, un langage comme C++ utilise un *typage statique* qui est déclaré par l'utilisateur dans le code source. Cela permet au compilateur de détecter les erreurs de typage plus tôt et d'optimiser l'allocation mémoire. En revanche cela donne plus de rigidité au code et alourdit la programmation. Le type d'un objet s'obtient par la commande `type(objet)`.

Nous allons faire ici la revue des principaux types proposés par Python.

A Les types numériques

Les entiers relatifs : `int`
Depuis Python 3, il n'y a pas de limite de taille pour l'entier[1] : l'entier peut être aussi grand que l'on veut dès lors que cela tient dans la mémoire.

Les nombres décimaux : `float`
La virgule est représentée par un point (notation anglo-saxonne). Les flottants possèdent un nombre limité de chiffres après la virgule. N'imaginez donc pas que vous faites des calculs exacts[2].

Les nombres complexes : `complex`
Les nombres complexes sont codés sous forme cartésienne (couple de flottants). Le nombre imaginaire i est désigné par la lettre `j`. On les écrit sous la forme `a+bj`.
N'oubliez pas le `b` *même s'il vaut* 1.

```
type(5)
type(-3)
type(0)
type(5.)
type(5.09934)
type(4,34)      #renvoie une erreur
type(5+3j)
type(2j)
type(3+j)       #renvoie une erreur
type(2-1j)
```

Python 1: Types numériques

Méthodes sur les complexes
Python est un langage orienté objet. Chaque objet possède donc des méthodes qu'il hérite de sa classe.
Nous n'allons pas rentrer dans les détails, mais vous reconnaîtrez les méthodes à la syntaxe utilisée : un point suivi du nom de la méthode.

[1] Ce n'est pas le cas dans beaucoup d'autres langages.
[2] En fait, Python fait une approximation du flottant en fraction binaire avant de réaliser les calculs. Ceci peut conduire a des résultats surprenants même pour des valeurs avec peu de décimales.

```
a=3+2j
a.real
a.imag
a.conjugate()
```

Python 2: Méthodes sur les complexes

Remarque : Lorsque dans Pyzo,vous tapez le nom de la variable suivi du point, il vous propose les méthodes disponibles (beaucoup plus nombreuses que celles présentées ici).

Opérations usuelles
Les opérations usuelles sont proposées par Python :

Opérateur Python	Opération mathématique
`x+y`	Somme de x et y
`x-y`	Différence de x et y
`x*y`	Produit de x et y
`x/y`	Quotient de x par y
`x**y`	x à la puissance y
`abs(x)`	Valeur absolue de x
`x//y`	Division *entière* de x par y
`x%y`	Reste de la division *entière* de x par y

Table 1: Opérations numériques usuelles

Remarque : Python ne fait pas de calculs exacts, il introduit donc des erreurs d'approximation dans certains calculs. Il faut en tenir compte.
Dans le dernier exemple du listing qui suit, on demande à Python d'écrire 0,1 avec 20 décimales dernière la virgule. Si vous essayez de taper ce code, vous verrez que Python a introduit des décimales non nulles plus loin dans le développement décimal. Cela vient du fait qu'il enregistre les nombres en format binaire. Lorsque Python convertit 0,1 en binaire, comme l'écriture ne "tombe pas juste", cela provoque une petite erreur d'approximation.

- Pour la puissance, n'utilisez pas le caractère "^". Par convention, $0^0 = 1$.
- Les *puissances de* 10 sont codées avec la lettre `e` en minuscule. Ainsi 5000 peut s'écrire $5e3$.
  La puissance est entière.
- Python accepte de faire des opérations entre des éléments numériques qui n'ont pas le même type. Il prendra alors le type le plus général pour le résultat.

```
1e-20      #renvoie 1e-20
1e-20+1-1 #renvoie 0, cela vient des erreurs d'arrondis
1e-20+(1-1) #renvoie 1e-20, l'addition n'est pas associative chez les serpents
5+5        #int
5+3.       #float
3.*(5-2j) #complex
5/3        #float
13//4      #int
13.0//4.0 #float
13.2%4.0 #float

#Le code suivant n'est pas à connaître.
#écrit l'approximation Python de 0.1 avec 20 décimales
"{0:.20f}".format(0.1)
```

Python 3: Opérations sur les nombres

Incrémenter une variable
Supposons à présent que dans votre programme, vous savez que la variable "a" contient un nombre. Vous cherchez simplement à augmenter sa valeur de k, on dit incrémenter. Par exemple, pour incrémenter de 1 :

- si la variable "a" pointe vers 5 au début, vous voulez qu'elle pointe $5+1=6$ ensuite,
- si elle pointe vers -12 au début, vous voulez qu'elle pointe vers $-12+1=-11$ ensuite.
- ...

Il existe plusieurs manières de faire la manipulation. La plus naturelle consiste à remplacer la valeur de `a` par `a+1`.

```
a=1
a=a+1
print(a)
```

Python 4: Incrément simple

On peut aussi utiliser une écriture plus ramassée

```
a=1
a+=1
print(a)
a+=5
print(a)
a-=4
print(a)
a=+10          #Attention
print(a)
```

Python 5: Incrément compact

B Les booléens

Le type `bool` correspond aux booléens : `True` (Vrai) et `False` (Faux). `True` est synonyme de 1 et `False` est synonyme de 0.
Nota : Python fait la distinction entre les majuscules et les minuscules. `true` n'est pas la même chose que `True`.

Les comparaisons entre éléments renvoient un booléen.

Opérateur	Signification
==	Est égal
!=	Est différent
<	Est strictement inférieur
>	Est strictement supérieur
>=	Est supérieur ou égal
<=	Est inférieur ou égal

Table 2: Opérateurs de comparaison

⚠ Le test d'égalité se fait avec un double égal ==, le simple égal est réservé à l'affectation (le test d'égalité ne modifie pas les variables, contrairement à l'affectation).
Nota : Il existe aussi les opérateurs `is` et `is not` pour déterminer s'il s'agit du même objet. C'est à dire si les deux variables pointent vers le même objet en mémoire.

On peut construire des expressions logiques à partir des liens logiques :

Opérateur	Signification
`or`	OU logique
`and`	ET logique
`not`	NON logique

Table 3: Opérateurs logiques

a	b	not a	a or b	a and b
True	True	False	True	True
True	False	False	True	False
False	True	True	True	False
False	False	True	False	False

Table 4: Tables de vérité

Un python, c'est fainéant (et nous aussi) :
Les opérateurs `or` et `and` ne lisent que le minimum d'information et s'arrêtent dès qu'ils peuvent conclure.
Ainsi, dans l'expression "`a or b`", Python ne lit l'expression `b` que si `a` vaut `False`. Sinon, il renvoie directement `True`.
De même, pour "`a and b`", Python ne lit l'expression `b` que si `a` vaut `True`. Sinon, il renvoie directement `False`.
Cette subtilité est à garder en mémoire, car cela permet parfois de simplifier des écritures. Par exemple "`a==0 or abs(b/a)>100`" est une expression valide car la division n'est effectuée que si `a==0` renvoie `False`.

Remarque : L'opérateur `not` n'est pas prioritaire par rapport aux opérateurs de comparaisons. N'hésitez pas à utiliser des parenthèses pour écrire les expressions logiques et éviter des erreurs dans les priorités.

On peut mettre des comparaisons les unes à la suite des autres, le mot clef `and` est alors implicite.
Par exemple `a<b<c!=d` veut dire `a<b and b<c and c!=d`.

```
1==0       #False
1==1       #True


(1==1)+1   #convertit True en 1
1==1+1

5.3>2      #convertit tout en float
2.0==2

3<4<6>4
3<4<6>(5/0)
3<4>6>(5/0)

#Le not n'est pas prioritaire et calculé à la fin
not (1==1)*(2==5)

#ici le résultat diffère car not est dans la parenthèse
(not 1==1)*(2==5)

#Seul le 0 vaut False
bool(0)    #convertit en booléen
bool(1)
bool(-2.5)
```

Python 6: Tests logiques

C Les chaînes de caractères

Les chaînes de caractères servent au texte, elles sont désignées par le type `str`.

Guillemets
Elles sont encadrés par des guillemets simples ou doubles. Ils ont le même rôle, vous devez simplement ouvrir et fermer la chaîne avec les mêmes types de guillemets.

```
type("Hello World !")
type('Hello World !')
```

```
"trop"=='trop'   #trop, c'est trop
```

Python 7: Chaînes de caractères

Si on choisit des guillemets doubles, on peut mettre des apostrophes (guillemets simples) dans la chaîne de caractères sans que cela ne pose de problèmes. Par contre, si on veut mettre des doubles guillemets, il faut les *échapper* avec l'antislash : \".
De même si on encadre par des guillemets simples.

```
"J'aime Python"
'J'aime Python'   #erreur
"J'ai dit \"J'aime Python\""
'J\'ai dit "J'aime Python"'
```

Python 8: Guillemets et échappement

Opérations sur les chaînes de caractères

Opération Python	Manipulation de la chaîne
`a+b`	Concaténation des chaînes `a` `b` (Colle deux chaînes bout à bout)
`len(s)`	Longueur de la chaîne `s` (nombre de caractères)
`n*s`	Répète n fois la chaîne `s`
`a in b`	appartenance de `a` à la chaîne `b`
`a not in b`	le contraire

```
serpent="Python"
poisson='thon'
len(serpent)     #Pas si long, finalement
poisson in serpent
'Boum'=='boum'    #attention à la casse majuscule/minuscule !
5*"idiot"         #restera différent de ''intelligent''
serpent=="Py"+poisson
```

Python 9: Opérations sur les chaînes de caractères

Indexation et slicing (tranchage)
Dans une chaîne de caractères, les caractères sont numérotés **en commençant par** 0. On peut ainsi faire référence à certains caractères en particulier.
À la suite de la session précédente ,

```
serpent[0]
lg=ln(serpent)
serpent[lg-1]
serpent[lg]   #erreur, on est sorti de la chaîne car l'index commence par 0
```

On peut aussi faire référence à une partie de la chaîne entre deux indices `i` et `j` (`j` est exclu) : `[i,j]`.

```
serpent[0:2]    #caractères 0 et 1

#''l'oubli'' du premier indice le met automatiquement à 0
serpent[:2]     #caractères 0 et 1

serpent[2:5]    #caractères 2,3 et 4

serpent[2:lg]==poisson
#''l'oubli'' du deuxième indice le met automatiquement au maximum
serpent[2:]

serpent[:]

planete=serpent[:1]+'lut'+serpent[4:]
```

Python 10: Slicing des chaines de caractères

Pour prendre qu'une lettre sur deux, on rajoute un troisième argument qui vaut 2, pour une lettre sur trois, ce sera 3.

```
serpent[0:lg:2]
serpent[::2]
serpent[1:lg:2]
serpent[1::3]
```

Python 11: Slicing des chaines de caractères - pas

Enfin, on peut aussi compter à partir de la fin avec les nombres négatifs.

```
serpent[-1]      #dernier élément
serpent[-2]      #avant-dernier élément
serpent[::-1]    #lit à l'envers
```

Python 12: Slicing des chaines de caractères - indices négatifs

Syntaxe	Signification
`chaine[i]`	$i^{\text{ème}}$ élément, on compte à partir de 0 Si i est négatif, compte à partir de la fin.
`chaine[i:j]`	Sous liste des éléments i (inclus) à j (exclu) Si i est vide, alors il est remplacé par 0 (début de la chaîne) Si j est vide, alors il est remplacé par `len(chaine) (fin de la chaîne)`
`chaine[i:j:k]`	Sous chaîne des éléments i (inclus) à j (exclu) avec un pas de k k peut être négatif pour compter à rebours.

Table 5: Parcours de la chaîne et extraction

D Les listes

Les listes sont du type `list`. Dans le principe, elles sont très semblables à des chaînes de caractères, ou chaque caractère est indexé par sa position, mais possède un type quelconque (`int`, `float`, `str`,..., ou même `str`). Les éléments d'une liste n'ont pas tous forcément le même type.

Une liste s'écrit entre crochets avec des éléments séparés par des virgules.
Les éléments sont indexés et on peut y avoir accès comme pour les chaînes de caractères.

```
liste=[1,'deux']
liste
type(liste)
liste[0]       #élément d'indice 0
liste+=3       #erreur : on ne peut ajouter un entier à une liste
liste+=[3]     #c'est mieux !
liste[1:]      #sous liste
```

Python 13: Listes

Attention : Il faut faire attention à distinguer deux notations :

- `liste[2]` donne l'élément d'indice 2 de la liste,
- `liste[2:3]` donne la sous liste qui contient l'élément d'indice 2.
- `liste[2:2]` donne une sous-liste vide (placée en indice 2). On peut utiliser cette syntaxe pour insérer une sous-liste en position 2).

```
liste=[1,'deux',3]
liste
type(liste)
```

```
liste[2]         #renvoie l'entier 3
liste[2:3]       #renvoie la sous liste qui contient 3
type(liste[2])     #entier
type(liste[2:3])   #liste
liste=[]           #liste vide
liste
```

Python 14: Élément d'une liste versus sous liste

Modification d'une liste
Contrairement à une chaîne de caractères, une liste peut être modifiée de l'intérieure sans être recopiée.

```
liste=[1,'deux',3]
liste[2]=2
liste
liste[1:3]=5       #erreur, ce doit être une sous-liste
liste[1:3]=[5]
liste
liste[1:3]=[2,3,4,5,6]
liste
liste[0]=['do','re','mi'] #liste dans la liste
```

Python 15: Modification d'une liste

Pour montrer la proximité entre les listes et les chaînes de caractères, on peut voir qu'une chaîne peut être interprétée comme une liste. Comparez les deux instructions :

```
liste=[1,2]
liste[0]='chaine'
liste
liste[0:1]='chaine'
liste
```

Python 16: Les chaînes et les listes sont *iterables*

Dans le premier cas, on change l'élément 0 de la liste pour le remplacer par la chaîne de caractères `chaine`.
Dans le deuxième cas, on change une sous liste, il faut donc la remplacer par une sous-liste. Python interprète alors la chaîne de caractères comme une liste de caractères qu'il rentre les uns à la suite des autres.

Méthodes sur les listes
Ces méthodes **modifient** la liste.

Méthode	Action correspondante
`liste.append(a)`	Ajoute l'élément "a" en fin de liste
`liste.pop()`	Enlève le dernier élément et le renvoie en résultat
`liste.pop(i)`	Enlève l'élément i de la liste et le renvoie en résultat
`liste.index(x)`	Renvoie l'indice de la première occurrence de x dans la liste. Renvoie une erreur s'il n'y est pas.
`liste.count(x)`	Compte le nombre d'occurences de x dans la liste.
`liste.reverse()`	Retourne la liste.
`liste.sort()`	Trie la liste (utiliser pour les types numériques).

Table 6: Méthodes sur les listes

```
liste=[1,2]
liste.append(3)
liste
liste.pop()
liste
liste.append(3)
liste.reverse()
liste
liste.append(7)
liste.sort()
liste
```

Python 17: Méthodes sur les listes

Difficultés avec l'affectation

```
liste=[1,2,3]
liste2=liste
liste[0]='modif'
liste
liste2
liste2 is liste
```

Python 18: Listes et affectations

On constate que l'élément 0 de **liste2** a aussi été modifié !
Ces subtilités nous amènent doucement vers les difficultés de l'affectation.
Dans l'exemple, lorsque l'on effectue l'affectation `liste2=liste`, Python fait pointer les deux éléments vers la même liste en mémoire (comme nous n'avons vu plus haut pour les types plus simples). C'est ce qui est confirmé par le test logique : `liste2 is liste`.
Lorsque `liste[0]` est modifiée, ce n'est pas tant l'adresse mémoire de la liste qui est modifiée, qu'un élément à l'intérieur de cette liste. On modifie ainsi le pointeur d'indice 0 dans la liste, mais la liste elle-même garde le même emplacement mémoire. `liste2` qui pointe vers le même emplacement est donc aussi modifiée.
Cela reste vrai si on ajoute ou supprime des éléments de la liste.

Si on veut imposer à Python de faire une copie de la liste sur un autre emplacement mémoire, on utilise la méthode `copy`. (On peut aussi le faire avec une boucle **for**).

```
liste=[1,2,3]
liste2=liste.copy()
liste2 is liste
liste[0]='modif'
liste
liste2
```

Python 19: Copie d'une liste

E Les dictionnaires

Dans une liste, les éléments sont indexés par des entiers de 0 à n.
Un dictionnaire correspond à une liste dont on choisit les indexes (qui ne sont pas nécessairement entier) que l'on appelle clef (key). Un dictionnaire est de type `dict`.
Les dictionnaires sont entrés entre accolades. Chaque élément est représenté par un couple de la forme `clef:valeur`.

```
dico={"un":1,"deux":2,"trois":3,55:[5,'cinq']}
dico[1]          #erreur, ce n'est pas un indice.
dico[55]
dico[55][1]
dico['un']=0       #modification
dico
del(dico['un'])    #suppression
dico
dico['dix']=10     #ajout
dico
len(dico)          #nombre d'éléments
dico.keys()        #liste des clefs
dico.values()      #liste des valeurs
```

Python 20: Dictionnaires

Exemple d'utilisation concrète d'un dictionnaire :

```
coord={'abscisse':3.2,'ordonnee':-4.3}
coord[abscisse]
coord[ordonnee]
```

Python 21: Dictionnaire : exemple concret

F Les tuples

Les tuples sont désignés par le type `tuple`. Ils fonctionnent comme des listes figées (que l'on ne peut pas modifier une fois construites : les tuples sont **non mutables**). On les distingue des listes par l'utilisation de parenthèses à la place des crochets.

```
t1=()                    #tuple vide
type(t1)
t2=(1,2,'trois')
type(t2)
t2[0]
t2[2]
len(t2)
t2[1:]                   #slicing
t2[2]=3                  #erreur : tuple non mutable
t2+t2                    #fonctionne car le tuple n'est pas modifié
t2
t2.append(4)             #erreur : tuple non mutable
t2.reverse()             #erreur : tuple non mutable
t2[::-1]                 #lit à l'envers mais ne modifie pas
t2
```

Python 22: Tuples

Attention, les parenthèses servent aussi à séparer les expressions, à modifier les ordres de priorité... comme en maths. Ainsi, il n'existe pas de tuples avec un seul élément.

```
t3=(17)
type(t3)                 #entier
t3
```

Python 23: Tuples à un élément ?

Type	Nature
`int`	Entier *de taille arbitraire*
`float`	Nombre décimal *dit nombre à virgule flottante*
`complex`	Nombre complexe
`str`	Chaine de caractères (texte)
`bool`	Booléen *(Valeur logique)*
`tuple`	Liste de longueur fixe
`list`	Liste de longueur variable

Table 7: Quelques types de données en Python

G Conversion et lecture au clavier

Lorsque les expressions sont compatibles, on peut les convertir d'un type à un autre.

```
lettre='1'
type(lettre)
entier=int(lettre)
type(entier)
flottant=float(entier)
```

```
type(flottant)
list(lettre)

#Attention
print(str(5))
print(str(float(5))
```

Python 24: Conversion du type

Pour lire une saisie au clavier, on utilise la commande `input`. Par défaut, `input` renvoie une chaîne de caractères.

```
entree=input('entrez un entier')
entree+1                              #erreur car...
type(entree)
int(entree)+1
entree=float(input('entrez un nombre'))
type(float)
```

Python 25: Lecture au clavier