# Table des matières

# Chapitre A

# Révisions

Dans ce premier chapitre, nous n'allons pas programmer de jeu, mais revoir à la vitesse d'un grand 8 quelques exercices de base de programmation en Python. Si vous manipulez déjà avec aisance les tests, les boucles et les manipulations de listes, vous pouvez directement passer au chapitre suivant.

## 1 - Variables et fonctions

### La console

Dans la zone **console** de Python, on peut saisir des instructions qui seront exécutées immédiatement. Cette saisie se fait après les >>> que l'on appelle des **chevrons**. Cela est très pratique pour tester en direct les fonctions que vous aurez programmées. Voici un rappel de quelques opérations possibles sur les nombres :

```
>>> 3+1
4
>>> len("Python")
6
>>> (15%4) ** 2
9
```

| Opérateur | Effet |
|---|---|
| +, -, * | Respectivement la somme, la différence et le produit |
| $a$ / $b$ | Quotient décimal de $a$ par $b$ |
| $a$ // $b$ | Quotient entier de $a$ par $b$ |
| $a$ % $b$ | Reste entier dans la division de $a$ par $b$ |
| $a$ ** $b$ | Résultat de $a^b$ ( et non ^) |

Souvenez-vous que dans la console, vous pouvez utiliser les touches ↑ et ↓ pour vous déplacer dans l'historique de ce que vous avez déjà tapé au fur et à mesure.

Python respecte les priorités mathématiques des opérations. En informatique, ce concept porte le nom de **précédence** des opérateurs. Dans le cas du langage Python, les précédences sont définies ainsi (dans l'ordre croissant) :

| | |
|---|---|
| +, - | Addition et soustraction |
| *, /, //, % | Multiplication, division, quotient et reste |
| $+x$, $-x$ | Positif, négatif |
| ** | Exponentiation |

## Les variables

Pour programmer nos jeux, il faudra conserver un certain nombre d'informations en mémoire ; on utilise alors des variables pour stocker les valeurs. Le fait de donner un nom à un calcul, un texte ou un autre objet s'appelle **l'affectation**.

| En Algorithmique | On lit ... | En Python |
|---|---|---|
| $a \leftarrow 3$ | $a$ reçoit la valeur 3 | `a = 3` |

Dans la console Python, lorsque l'on affecte une valeur à une variable, le contenu de celle-ci n'est pas affiché. Il faut taper le nom de la variable pour afficher sa valeur.

Lors d'une affectation, la quantité à droite du signe = est évaluée et stockée dans la variable de gauche. L'écriture `a = a + 1` a donc un sens en informatique. Cela signifie simplement que la variable `a` est augmentée de 1.

Le symbole = joue donc un rôle d'affectation et non d'égalité. D'ailleurs, l'égalité `b = a * 2` n'est plus vérifiée à la fin de notre exemple.

```
>>> a = 3
>>> b = a * 2
>>> b
6
>>> a + 1
4
>>> a
3
>>> a = a + 1
>>> a
4
>>> b
6
```

Voici donc un petit exercice permettant de jouer avec les nombres et les opérations avec Python...

**EX A1** *Si $n$ est un entier positif, relier les expressions Python aux expressions françaises correspondantes :*

- `n + 1`
- `n // 10 % 10`
- `n // 10`
- `n // 2 * 2 + 1`
- `n ** 2`
- `n % 10`
- `n * 2`

- *Le chiffre des unités de $n$*
- *Le premier nombre impair qui suit $n$*
- *Le chiffre des dizaines de $n$*
- *Le nombre de dizaines de $n$*
- *Le carré de $n$*
- *Le double de $n$*
- *L'entier qui suit $n$*

## Affectations simultanées

Commençons par un premier exercice pour nous faire une idée.

**EX A2** *On considère la suite d'instructions ci-contre :*

```
>>> x = 17
>>> y = 32
>>> x = x + y
>>> y = x - y
>>> x = x - y
```

1. *Que contiennent les variables $x$ et $y$ à la fin ?*
2. *Constate-t-on le même phénomène pour des valeurs quelconques de $x$ et $y$ ?*
3. *Écrire une suite d'instructions produisant le même effet, mais sans effectuer de calcul (on pourra utiliser une troisième variable temporaire).*

Python propose une manière simple d'affecter simultanément des variables en utilisant une virgule : les deux premières lignes de l'exercice précédent peuvent être réduites en `x, y = 17, 32`. Comme pour l'affectation classique, lorsque l'on procède à une affectation simultanée, les quantités de droite sont évaluées dans un premier temps, puis dans un second temps elles sont stockées dans les variables de gauche. Ainsi les 3 dernières lignes de l'exercice précédent peuvent tout simplement être transformées en : `x, y = y, x`.

**EX A3** ***La carte vitale.***

*En France, le numéro de sécurité sociale présent sur la carte vitale est un numéro "signifiant" (c'est-à-dire non aléatoire) composé de 13 chiffres permettant d'identifier une personne, suivi d'une clé de contrôle de 2 chiffres pour déceler d'éventuelles erreurs de saisie.*


*Image par Giesesamvitale sur Wikipédia.*

*Par exemple, le premier chiffre indique le sexe (1 pour un homme, 2 pour une femme), les deux suivants l'année de naissance, suivis de deux chiffres pour indiquer le mois et ainsi de suite. Pour la carte fictive ci-dessus, le code est donc* 2690549588157 *et la clé* 80*. Cette clé correspond à la différence entre* 97 *et le reste de la division du code par* 97*. Réaliser une suite d'instructions qui, connaissant le code* N *à 13 chiffres, calcule la clé de sécurité.*

## Les types d'objets

Il existe de nombreux types d'objets en Python, nous apprendrons même dans ce livre à créer nos propres types ! En voici quelques-uns que nous allons régulièrement utiliser.

Dans la console (ou plus tard dans un programme), on peut demander de quel type est une variable à l'aide de la fonction **`type`** comme le montre l'exemple de droite.

```
>>> n = 5
>>> type(n)
<class 'int'>
>>> type(2 ** 2020)
<class 'int'>
>>> type(1/2)
<class 'float'>
>>> A=(2,3)
>>> type(A)
<class 'tuple'>
>>> n < 1
False
>>> type(n < 1)
<class 'bool'>
```

• **Les entiers** : en Python, les entiers ont la particularité de ne pas être limités en taille (contrairement à la majorité des autres langages).

• **Les flottants** : pour simplifier, on peut considérer que ce sont les nombres décimaux ou même réels. En réalité ce n'est pas si simple, comme l'illustre le 3e exemple.

```
>> type(3)
<class 'int'>
>>> type(3.0)
<class 'float'>
```

Pour indiquer à Python qu'un nombre entier est un flottant, on ajoute `.0` ou juste un point.

```
>>> a = 45 / 9
>>> a
5.0
>>> type(a)
<class 'float'>
```

Une division décimale renvoie toujours un flottant même si le résultat est entier.

```
>>> 1 - 0.8 - 0.2
-5.551115123125783e-17
```

Enfin, il faut se méfier des "erreurs d'arrondis" avec les nombres flottants même simples.

Ici le résultat affiché est environ $-5,6.10^{-17}$. Nous n'irons pas plus loin dans les explications mais il faut retenir qu'il peut se produire des erreurs d'arrondis sur les nombres flottants et qu'il est préférable d'utiliser des entiers, dans la mesure du possible bien entendu.

• **Les tuples** : ils représentent des couples ou triplets ou $n$-uplets. On peut récupérer chacune des composantes via leurs indices (ces derniers commencent à 0) ou directement par une affectation simultanée :

```
>>> A = (4, 2, 5)
>>> x = A[0]
>>> y = A[1]
>>> x
4
>>> y
2
```

ou

```
>>> A = (4, 2, 5)
>>> x, y, z = A
>>> x
4
>>> y
2
```

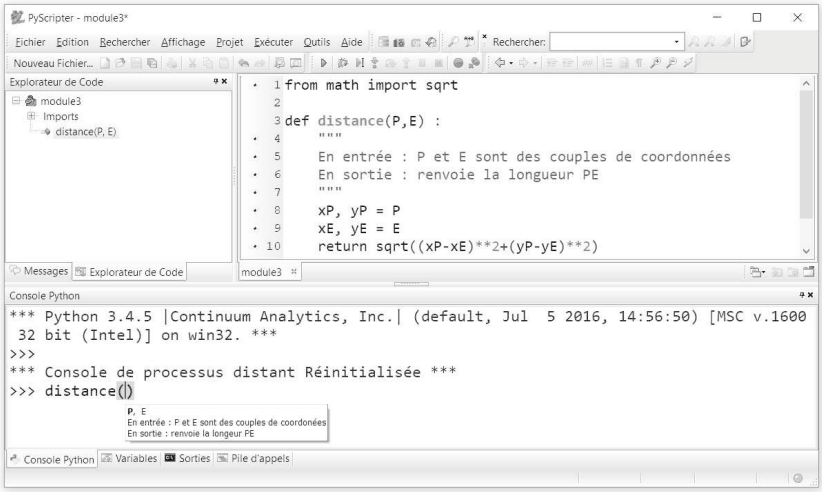
• **Les booléens** : c'est le dernier type dont nous parlerons rapidement. Ces variables ne peuvent prendre que deux valeurs : `True` ou `False` (Vrai ou Faux pour les non-anglophones !). Vous aurez sûrement remarqué la majuscule devant ces deux mots, pensez-y.

```
>>> b = 2 < 3
>>> b
True
>>> type(b)
<class 'bool'>
```

## Les fonctions

Python propose déjà un certain nombre de fonctions déjà programmées (on les appelle les fonctions **built-in** ou fonctions **natives** en français). Nous avons déjà rencontré par exemple la fonction `type` : elle reçoit une variable en entrée (un objet) et renvoie son type (sa classe). Dans nos projets, nous utiliserons très souvent des fonctions : elles permettent de séparer une tâche à réaliser en tâches plus simples et donnent ainsi de la clarté au code.

L'exemple suivant calcule et renvoie la distance entre un personnage $P(x_P ; y_P)$ et un ennemi $E(x_E ; y_E)$ à l'aide de la formule issue du théorème de Pythagore, $PE = \sqrt{(x_E - x_P)^2 + (y_E - y_P)^2}$ :




À noter que l'on a dû ici importer en première ligne la fonction racine carrée (**`sqrt`**) depuis le module `math` : il s'agit d'une bibliothèque contenant un grand nombre de fonctions mathématiques.

La déclaration d'une fonction est ainsi structurée :

- La déclaration commence par le mot clef **`def`** suivi du nom de la fonction (ici `distance`), puis des paramètres (ici `P` et `E`) et enfin deux points pour déclarer le bloc correspondant à la fonction.
- Juste en dessous, un texte placé entre des triples guillemets qui donne la description de la fonction. Ce texte appelé **docstring** est facultatif. S'il est renseigné, il s'affiche en info-bulle lorsque l'on tape le nom de la fonction, comme sur la capture précédente dans la console. Ceci est très pratique pour retrouver l'ordre des paramètres dès lors que l'on a créé un grand nombre de fonctions.
- L'ensemble des instructions à réaliser dans la fonction est **indenté**, c'est à dire décalé vers la droite pour structurer le programme.
- Enfin, dès que le programme rencontre l'instruction **`return`**, il quitte la fonction et renvoie la valeur indiquée.

Quelques remarques supplémentaires sur les fonctions :

- Comme dans l'exemple précédent, une fonction peut dépendre de plusieurs paramètres, il suffit de les lister en les séparant par une virgule.
- Si vous définissez une fonction dont le nom existe déjà, seule la dernière définition est prise en compte en écrasant la précédente.
- Une fois le programme exécuté, les fonctions et les variables en cours sont accessibles dans la console comme on peut le voir sur la capture d'écran précédente. Cependant, gardez en tête que les paramètres de la fonction sont des **variables muettes** : les variables `P` et `E` ne sont connues qu'à l'intérieur même de la fonction et vous pouvez très bien appeler `distance(A, (-1,4))` si `A` a été défini auparavant.
- De la même manière, toutes les nouvelles variables créées à l'intérieur de la fonction sont appelées **variables locales** et n'existent qu'à l'intérieur de la fonction. En particulier, elles n'interfèrent heureusement pas avec d'autres variables qui pourraient porter le même nom dans le reste du programme. Voici ce qui se passe si on demande la valeur de `xP` : elle n'est pas connue.

```
>>> distance((7,6),(10,2))
5.0
>>> xP
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
NameError: name 'xP' is not defined
```

- Une fonction peut aussi renvoyer plusieurs valeurs, il suffit de les séparer par une virgule au moment du `return`.
- Enfin, une fonction peut à son tour en appeler une autre. L'exemple suivant propose de détecter une collision et renvoie un booléen indiquant si la distance $PE$ est inférieure à $10$.

```
def touche(P,E) :
    return (distance(P,E) < 10)

>>> touche((10,2),(12,1))
True
```