



Révisions

- Premiers pas sur l'interpréteur de commandes
- Les variables
- Fonctions de base
- Tests conditionnels

Objectifs :

Le but de cette séance est de revenir sur les fondamentaux du langage Python et de voir le fonctionnement de la chaîne de développement.

Les éléments principaux sont :

- l'éditeur de texte, le terminal et les messages d'information python.
- l'interaction avec Python.
- les variables.
- les tests conditionnels.



Premiers pas sur l'interpréteur de commandes

Python présente la particularité de pouvoir être utilisé de plusieurs manières différentes.

1-directement dans l'interpréteur de commandes

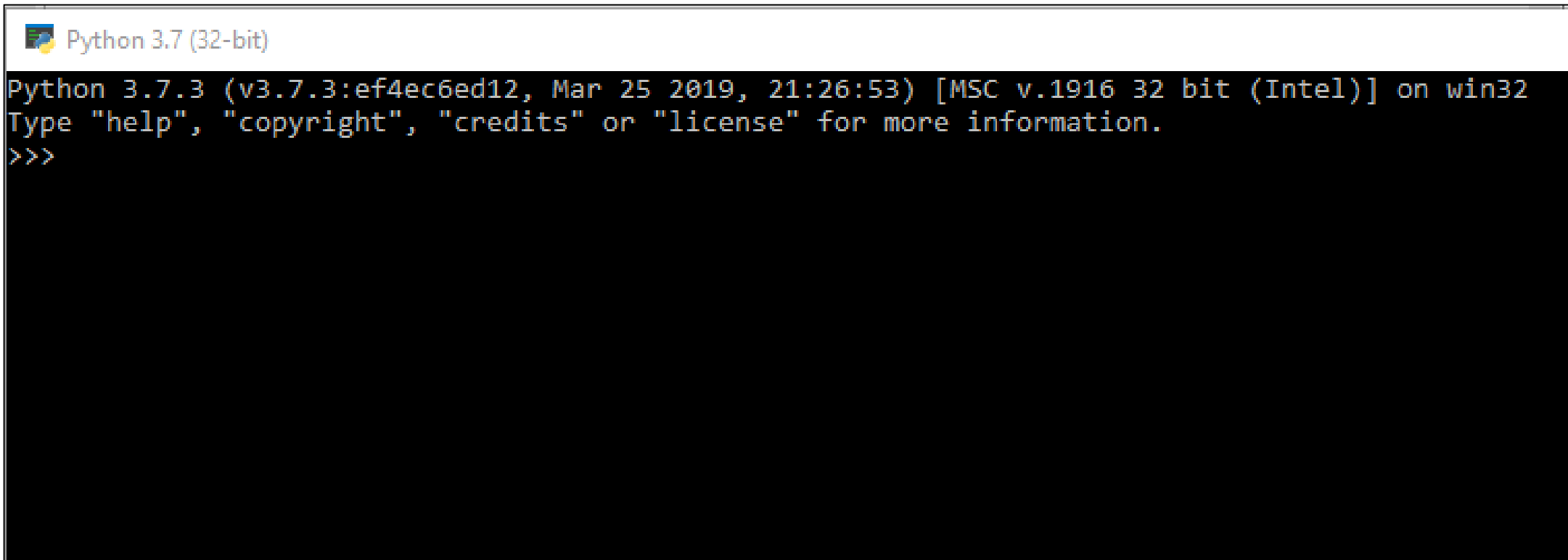
C'est le **mode interactif**, c'est-à-dire que vous pouvez dialoguer avec l'interpréteur directement depuis le clavier.

2-dans un fichier .py que nous exécuterons avec l'interpréteur de commandes
vous apprendrez comment créer vos premiers programmes (scripts) et les sauvegarder sur disque.

Lancer Python sous Windows

Nous allons désormais utiliser Python sous Windows.

Vous avez plusieurs façons d'accéder à la ligne de commande Python
menu Démarrer>Tous les programmes>Python
vous obtenez une console d'interprétation de Python (interpréteur).



```
Python 3.7 (32-bit)

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Interpréteur de commandes de Python

```
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

L'interpréteur de commandes va nous permettre de tester directement du code.

- Je saisis une ligne d'instructions,
- j'appuie sur la touche Entrée de mon clavier,
- je regarde ce que me répond Python (s'il me dit quelque chose),
- puis j'en saisis une deuxième,
- une troisième...

la série de trois chevrons (>>>) signifient : « je suis prêt à recevoir tes instructions ».

Interpréteur de commandes de Python

Saisissez une phrase
(test avec python)

```
>>> test avec python
      File "<stdin>", line 1
        test avec python
            ^
SyntaxError: invalid syntax
>>>
```

l'interpréteur vous indique qu'il a trouvé un problème dans votre ligne d'instruction :

- Il vous indique le numéro de la ligne (line 1)
- il vous répète la ligne (ceci est utile quand on travaille sur un programme de plusieurs centaines de lignes)
- Puis il vous dit ce qui l'arrête, ici : `SyntaxError: invalid syntax` (ce que vous avez saisi est incompréhensible pour Python)
- il vous affiche à nouveau une série de trois chevrons

On voit que l'interpréteur parle en anglais et les instructions que vous saisirez seront également en anglais.

Première instruction

Saisir un nombre

Vous avez pu voir sur notre premier test que Python n'aimait pas les suites de lettres qu'il ne comprend pas. Par contre, l'interpréteur aime les nombres :

```
>>> 7  
7  
>>>
```

ce retour indique que l'interpréteur a bien compris et que votre saisie est en accord avec sa syntaxe

vous pouvez saisir des nombres à virgule : on utilise ici la notation anglo-saxonne, c'est-à-dire que le point remplace la virgule :

```
>>> 9.5  
9.5  
>>>
```


Première instruction

Opérations courantes

Addition, soustraction, multiplication, division

Pour effectuer ces opérations, on utilise respectivement les symboles +, -, * et /.

```
>>> 3 + 4
```

```
7
```

```
>>> -2 + 93
```

```
91
```

```
>>> 9.5 + 2
```

```
11.5
```

```
>>> 3.11 + 2.08
```

```
5.18999999999999995
```

```
>>>
```

Première instruction

Opérations courantes

Addition, soustraction, multiplication, division

Pour effectuer ces opérations, on utilise respectivement les symboles +, -, * et /.

```
>>> 3 + 4
```

```
7
```

```
>>> -2 + 93
```

```
91
```

```
>>> 9.5 + 2
```

```
11.5
```

```
>>> 3.11 + 2.08
```

```
5.18999999999999995
```

```
>>>
```

Première instruction

Opérations courantes

Division entière

Il existe un opérateur qui permet de connaître le résultat d'une division entière.

Cet **opérateur** utilise le symbole « `//` ». Il permet d'obtenir la partie entière d'une division

```
>>> 10 // 3
```

```
3
```

```
>>>
```

La partie entière de la division de 10 par 3 est le résultat de cette division, sans tenir compte des chiffres au-delà de la virgule (en l'occurrence, 3)

Première instruction Opérations courantes modulo

Il existe un opérateur qui permet de connaître le reste d'une division.

L'opérateur « % », que l'on appelle le « modulo », permet de connaître le reste de la division

```
>>> 10%3  
1  
>>>
```

Pour obtenir le modulo d'une division, on « récupère » son reste. Dans notre exemple, $10/3 = 3$ et il reste 1

Première instruction

Opérations courantes

Puissance

```
>>> 5**2  
25  
>>> 2 ** 7  
128
```

En Python, il est possible de calculer des puissances avec l'opérateur **
seule limite, la capacité de la machine (à $2^{1000000}$ on la plante!)

Première instruction

Opérations courantes

Hiérarchie des opérations

```
>>> 7 + 3 * 4
```

```
>>> (7+3)*4
```

Lorsqu'il y a plus d'un opérateur dans une expression, l'ordre dans lequel les opérations doivent être effectuées dépend de règles de priorité.

Sous Python, les règles de priorité sont les mêmes que celles qui vous ont été enseignées au cours de mathématique.



Les variables

Une variable

une variable est une petite **information temporaire** qu'on stocke dans la mémoire vive (RAM).

On dit qu'elle est « **variable** » car c'est une valeur qui peut changer pendant le déroulement du programme.

Par exemple, un nombre de vies restant au joueur (5) risque de diminuer au fil du temps. Si ce nombre atteint 0, on saura que le joueur a perdu.

En langage Python, une **variable** est constituée de deux choses :

- une **valeur** : c'est le nombre qu'elle stocke (par exemple 5) ;
- un **nom** : c'est ce qui permet de la reconnaître.

En programmant en Python, on n'aura pas à retenir l'adresse mémoire : à la place, on va juste indiquer des noms de variables.

Python fera la **conversion entre le nom de la variable et l'adresse mémoire**.

Noms des variables

En langage Python, chaque variable doit donc avoir un **nom**

Contraintes

- il ne peut y avoir que des **lettres minuscules, majuscules** ($a \rightarrow z$, $A \rightarrow Z$) et des **chiffres** ($0 \rightarrow 9$)
- Seules les lettres ordinaires sont autorisées. Les lettres accentuées, les cédilles, les caractères spéciaux tels que \$, #, @, etc. sont interdits
- votre nom de variable doit **commencer par une lettre**
- les **espaces** sont **interdits**. À la place, on peut utiliser le caractère « **underscore** » _ (qui ressemble à un trait de soulignement). C'est le seul caractère différent des lettres et chiffres autorisé
- **passer en majuscule le premier caractère de chaque mot**, à l'exception du premier mot constituant la variable.

La variable contenant mon âge se nommerait alors **monAge**.

- le langage Python fait la différence entre les majuscules et les minuscules, il **respecte la « casse »**, ce qui signifie que des lettres majuscules et minuscules ne constituent pas la même variable (la variable AGE est différente de aGe, elle-même différente de age).

Noms des variables

Voici quelques exemples de noms de variables corrects

nombreDeVies, prenom, nom, numero_de_telephone, numeroDeTelephone

Quoi que vous fassiez, faites en sorte de donner des **noms clairs** à vos variables :

On aurait pu abrégé nombreDeVies, en l'écrivant par exemple ndv.

C'est peut-être plus court, mais c'est beaucoup moins clair pour vous quand vous relisez votre code.

Choisissez la convention qui vous plaît, puis essayez de rester cohérent et de **n'utiliser qu'une seule convention d'écriture pour tout votre projet.**

Ainsi, vous pourrez vos variables facilement dès que vous commencez à travailler sur des programmes volumineux.

Mot-clés réservés

Enfin, certains mots-clés de Python sont **réservés**, c'est-à-dire que vous ne pouvez pas créer des variables portant ce nom
voici la liste pour Python 3 :

<code>and</code>	<code>del</code>	<code>from</code>	<code>none</code>	<code>true</code>
<code>as</code>	<code>elif</code>	<code>global</code>	<code>nonlocal</code>	<code>try</code>
<code>assert</code>	<code>else</code>	<code>if</code>	<code>not</code>	<code>while</code>
<code>break</code>	<code>except</code>	<code>import</code>	<code>or</code>	<code>with</code>
<code>class</code>	<code>false</code>	<code>in</code>	<code>pass</code>	<code>yield</code>
<code>continue</code>	<code>finally</code>	<code>is</code>	<code>raise</code>	
<code>def</code>	<code>for</code>	<code>lambda</code>	<code>return</code>	

Affecter une valeur à une variable

En Python, pour donner une valeur à une variable, il faut écrire **nom_de_la_variable = valeur**.

```
>>> mon_age = 21
```

On dit en effet qu'on a **affecté** la valeur 21 à la variable mon_age

NB : Les espaces séparant « = » du nom et de la valeur de la variable sont facultatifs

afficher la valeur d'une variable

On peut afficher la valeur d'une variable en la saisissant simplement dans l'interpréteur de commandes

puis <Enter>.

Python répond en affichant la valeur correspondante

```
>>> mon_age  
21  
>>>
```

afficher la valeur d'une variable

Si vous modifiez la variable puis que vous la réaffichez, la valeur de la variable aura changé :

```
>>> mon_age = mon_age + 2  
>>> mon_age  
23  
>>>
```

Les types de données

Python a besoin de connaître quels types de données sont utilisés pour savoir quelles opérations il peut effectuer avec.

Python associe à chaque donnée un type, qui va définir les opérations autorisées sur cette donnée.

Les types de données incontournables :

- Les nombres entiers
- Les nombres flottants
- Les booléens
- Les chaînes de caractères

Les types de données

Les nombres entiers

Python différencie les entiers, des nombres à virgule flottante.

Pour un ordinateur, les opérations que l'on effectue sur des nombres à virgule ne sont pas les mêmes que celles sur les entiers, et cette distinction reste encore d'actualité.

Le type entier se nomme **int** (qui correspond à l'anglais « integer », c'est-à-dire entier).

La forme d'un entier est un nombre sans virgule

Les types de données

Les nombres flottants

Les flottants sont les nombres à virgule.

Ils se nomment **float** (ce qui signifie « flottant » en anglais).

La syntaxe d'un nombre flottant est celle d'un nombre à virgule (n'oubliez pas de remplacer la virgule par un point).

Si ce nombre n'a pas de partie flottante mais que vous voulez qu'il soit considéré par le système comme un flottant, vous pouvez lui ajouter une partie flottante de 0.

3.152

52.0

Les types de données

Les booléens

Un booléen est une information Vraie ou Fausse.

Ce type sera très utile lorsque nous commencerons à comparer des valeurs entre elles.

vrai s'écrit **True** et faux s'écrit **False** (pensez aux majuscules !).

Exemple :

Un est inférieur à deux

```
>>> 1 < 2  
True
```

Un est supérieur à 2

```
>>> 1 > 2  
False
```

le type booléen est très économique en termes de place mémoire occupée, puisque pour stocker une telle information binaire, un seul bit suffit (1 ou 0).

Les types de données

Les chaînes de caractères

La chaîne de caractères permet de stocker une série de lettres, une phrase. Elle se nomme **str** (abréviation de « string » qui signifie chaîne en anglais).

On peut écrire une chaîne de caractères de différentes façons :

- entre **guillemets** ("ceci est une chaîne de caractères") ;
- entre **apostrophes** ('ceci est une chaîne de caractères') ;
- entre **triples guillemets** ("""ceci est une chaîne de caractères""") ou entre **triples apostrophes** ('''')

On peut stocker une chaîne de caractères dans une **variable** :

```
>>> ma_chaine = "Bonjour !"
>>> ma_chaine
'Bonjour!'
>>>
```

Les types de données

Les chaînes de caractères

Echappement

Si vous utilisez les délimiteurs simples (le guillemet ou l'apostrophe) pour encadrer une chaîne de caractères, il se pose le problème des guillemets ou apostrophes que peut contenir ladite chaîne.

```
>>> chaine = 'J'aime le Python!'
SyntaxError: invalid syntax
```

vous obtenez une message d'erreur `SyntaxError`

Ceci est dû au fait que l'apostrophe de « J'aime » est considérée par Python comme la fin de la chaîne et qu'il ne sait pas quoi faire de tout ce qui se trouve au-delà.

Pour pallier ce problème, il faut **échapper** les apostrophes se trouvant au cœur de la chaîne. On **insère** un **caractère anti-slash** « **** » **avant** les apostrophes contenues dans le message

```
>>> chaine = 'J\'aime le Python!'
>>> chaine
"J'aime le Python!"
```

Les types de données

Les chaînes de caractères

Echappement

Pour pallier ce problème, on peut aussi alterner intelligemment entre les **guillemets** et les **apostrophes** :

```
>>> chaine2 = "J'aime le Python!"
```

```
>>> chaine2
```

```
"J'aime le Python!"
```

Les types de données

Les chaînes de caractères

Saut de ligne « \n »

Le caractère « \n » symbolise un saut de ligne.

```
>>> chaine="essai\nsur\nplusieurs\nlignes"  
>>> chaine  
'essai\nsur\nplusieurs\nlignes'  
>>>
```

Pour l'instant, l'interpréteur affiche les sauts de lignes comme on les saisit, c'est-à-dire sous forme de « \n ».

Nous verrons plus tard comment afficher réellement ces chaînes de caractères.

affecter à d'autres variables

On peut également attribuer à une variable la valeur d'une autre variable

Par exemple :

```
>>> toto = mon_age  
>>> toto  
23  
>>>
```

Signifie que l'on attribue la valeur de la variable `mon_age` à la variable `toto`

Signifie que la *valeur* de `toto` est maintenant celle de `mon_age`.

Notez bien que cette instruction n'a en rien modifié la valeur de `mon_age`

Règles :

- Si la variable n'est pas créée, Python s'en charge automatiquement.
- Si la variable existe déjà, l'ancienne valeur est supprimée et remplacée par la nouvelle

attribuer à d'autres variables

Vous pouvez aussi attribuer à d'autres variables des valeurs obtenues en effectuant des calculs sur la première variable

essayons d'attribuer une valeur à une autre variable d'après la valeur de mon_age

```
>>> toto = mon_age * 2  
>>> toto  
46  
>>>
```

Comme mon_age contenait 23, toto vaut maintenant 46.
De même que précédemment, mon_age vaut toujours 23.

Incrémentation

L'incrémentation désigne l'augmentation de la valeur d'une variable d'un certain nombre

`variable = variable + 1` Cette syntaxe est claire et intuitive mais assez longue

`variable += 1`

L'opérateur += revient à ajouter à la variable la valeur qui suit l'opérateur.

Les opérateurs -=, *= et /= existent également, bien qu'ils soient moins utilisés

```
>>> a = 2
>>> a += 1
>>> a
3
>>> a += 3
>>> a
6
```

Permutation

Python propose un moyen simple de permuter deux variables (échanger leur valeur)

```
>>> a = 5
>>> b = 32
>>> a,b = b,a # permutation
>>> a
32
>>> b
5
>>>
```

après l'exécution de la ligne 3, les variables a et b ont échangé leurs valeurs.

Affectation à plusieurs variables

On peut aussi affecter une même valeur à plusieurs variables

```
>>> x = y = 3
>>> x
3
>>> y
3
>>>
```

On peut aussi effectuer des affectations parallèles à l'aide d'un seul opérateur

```
>>> a,b,c = 4, 8.33, "bonjour"
>>> a
4
>>> b
8.33
```

Dans cet exemple, les variables **a** et **b** prennent simultanément les nouvelles valeurs **4** et **8,33** et la variable **c** prend la valeur "**bonjour**"

Commentaires

Les commentaires sont des messages qui sont ignorés par l'interpréteur et qui permettent de donner des indications sur le code.

En Python, un commentaire

- débute par un dièse (« # »)
- se termine par un saut de ligne.

Tout ce qui est compris entre ce # et ce saut de ligne est ignoré.

Un commentaire peut donc occuper

- la totalité d'une ligne (on place le # en début de ligne)
- une partie seulement, après une instruction (on place le # après la ligne de code pour la commenter plus spécifiquement)

```
>>> # Premier exemple
>>> a = 5
>>> b = 6 # Deuxieme exemple
>>>
```



Les fonctions de base

Utiliser une fonction

Une fonction exécute un certain nombre d'instructions déjà enregistrées.

C'est donc un groupe d'instructions écrites pour faire une action précise et auquel vous donnez un nom.

Vous n'avez plus ensuite qu'à appeler cette fonction par son nom, autant de fois que nécessaire (cela évite les répétitions).

La plupart des fonctions ont besoin d'au moins un paramètre pour travailler sur une donnée ; ces paramètres sont des informations que vous passez à la fonction afin qu'elle travaille dessus.

Les fonctions s'utilisent en respectant la syntaxe suivante :

nom_de_la_fonction(parametre_1,parametre_2,...,parametre_n).

- Vous commencez par écrire le **nom** de la fonction.
- Vous placez entre **parenthèses** les **paramètres** de la fonction. Si la fonction n'attend aucun paramètre, vous devrez quand même mettre les parenthèses, sans rien entre elles.

La fonction « type »

La fonction « type » permet de savoir de quel type est une variable

Syntaxe : `type(nom_de_la_variable)`

La fonction renvoie le type de la variable passée en paramètre

Si vous saisissez dans l'interpréteur les lignes suivantes

```
>>> a = 6  
>>> type(a)
```

Vous obtenez

```
<class 'int'>
```

Python vous indique donc que la variable a appartient à la classe des entiers (la notion de classe sera étudiée plus tard)

La fonction « print »

La fonction print est dédiée à l'**affichage** uniquement. Le **nombre de ses paramètres** est **variable**, c'est-à-dire que vous pouvez lui demander d'afficher une ou plusieurs variables

```
>>> a = 3
>>> print(a)
3
>>> a = a + 3
>>> b = a - 2
>>> print("a =", a, "et b =", b)
a = 6 et b = 4
```

Le premier *appel* à print affiche la valeur de la variable a, c'est-à-dire « 3 »

Dans le deuxième appel à print, on passe 4 paramètres :

- deux chaînes de caractères
- et les variables a et b.

Quand Python interprète cet appel de fonction, il va afficher les paramètres dans l'ordre de passage, en les **séparant par un espace**

Fonction pour convertir une variable vers un autre type

Pour convertir une variable vers un autre type, il faut utiliser le nom du type comme une fonction

int() : Permet de convertir une valeur en un nombre entier.

```
>>> str = "1234"  
>>> nb = int(str)  
>>> str  
'1234'  
>>> nb  
1234  
>>>
```

str() : Permet de convertir une valeur en chaîne de caractère

float() : Permet de convertir une valeur en nombre flottant

La fonction « input »

input() est une fonction qui va permettre **d'interagir avec l'utilisateur**.

input() accepte un **paramètre** facultatif : **le message à afficher à l'utilisateur**.

```
>>> # Test de la fonction input
>>> nombre = input("Saisissez un nombre : ")
Saisissez un nombre : 9
```

Dès que le programme rencontre une instruction **input**, l'exécution s'interrompt, affiche le message ("Saisissez un nombre") et attend que l'utilisateur saisisse une valeur sur son clavier. L'interruption peut durer quelques secondes, quelques minutes ou plusieurs heures : **la seule chose qui fera exécuter la suite des instructions, c'est que la touche Entrée ait été enfoncée**. Aussitôt que c'est le cas, il se passe deux choses.

- **tout ce qui a été saisi** avant la touche Entrée (une suite de lettres, de chiffres, ou un mélange des deux) **est rentré dans la variable qui précède** l'instruction **input** (ici, *nombre* récupère la valeur 9).
- Et ensuite, immédiatement, la machine exécute la suite des instructions (s'il y en a).

La fonction « input »

La fonction **input()** renvoie toujours une chaîne de caractères.

Si vous souhaitez que l'utilisateur entre une valeur numérique, vous devrez donc convertir la valeur entrée (qui sera donc de toute façon de type string) en une valeur numérique, par l'intermédiaire des fonctions intégrées **int()** (si vous attendez un entier) ou **float()**.

```
>>> annee = input("Entrez une année : ")
Entrez une année : 2019
>>> type(annee)
<class 'str'>
>>> annee
'2019'
>>> # On veut convertir la variable en un entier, on utilise donc la fonction int
>>> annee = int(annee)
>>> type(annee)
<type 'int'>
>>> annee
2019
```



Écrire les programmes Python dans des fichiers sous Windows

l'interpréteur est très pratique :
il propose une manière interactive d'écrire un programme, qui permet de tester le résultat de chaque instruction.

Mais l'interpréteur a aussi un défaut :
le code que vous saisissez est effacé à la fermeture de la fenêtre (pas de sauvegarde).
Or, si on rédiger des programmes relativement complexes, devoir réécrire le code entier de son programme à chaque fois qu'on ouvre l'interpréteur de commandes est assez lourd...

La solution ?

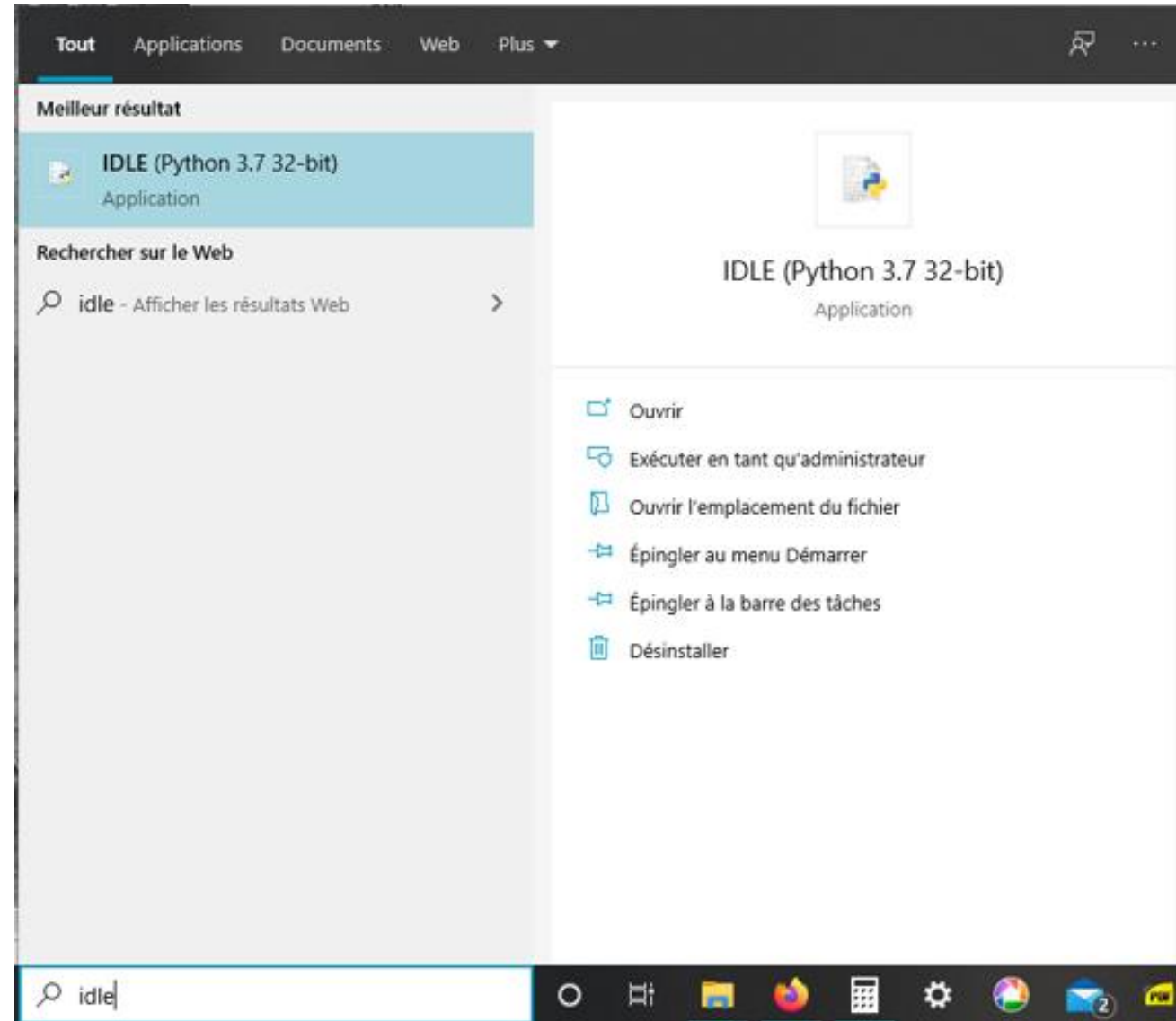
Mettre notre code dans un fichier que nous pourrions lancer à volonté, comme un véritable programme

Programme dans un fichier

On peut placer notre code Python dans un fichier, que nous pourrions ensuite exécuter

Voici la démarche :

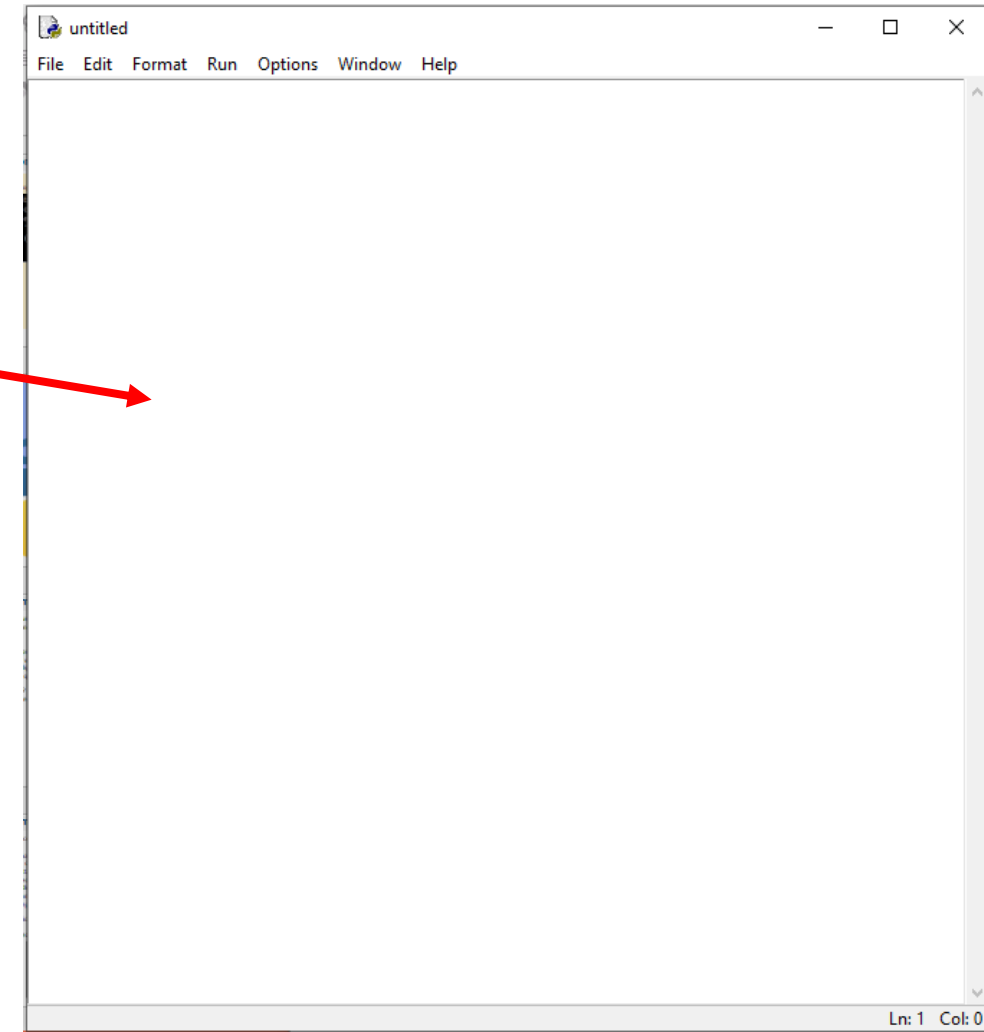
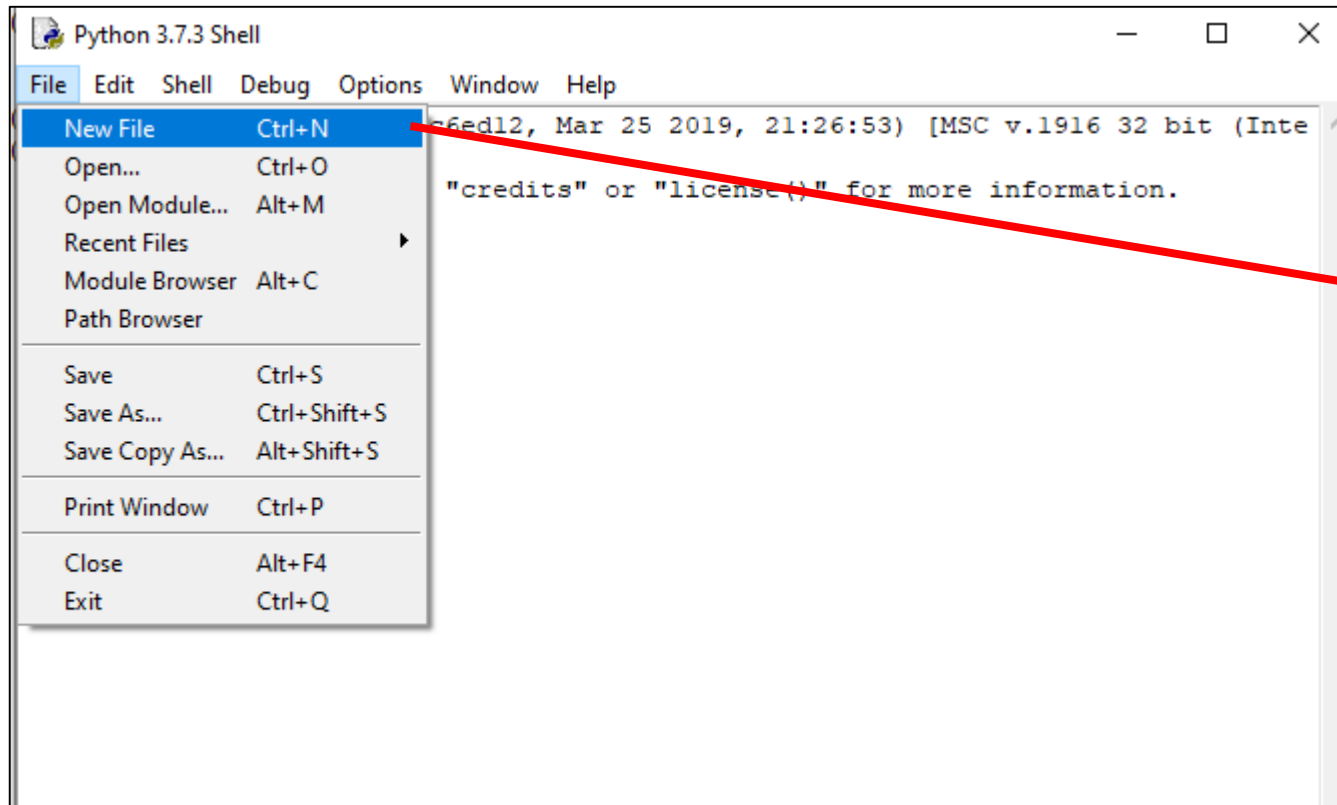
1. Ouvrez la console IDLE



Programme dans un fichier

Voici la démarche :

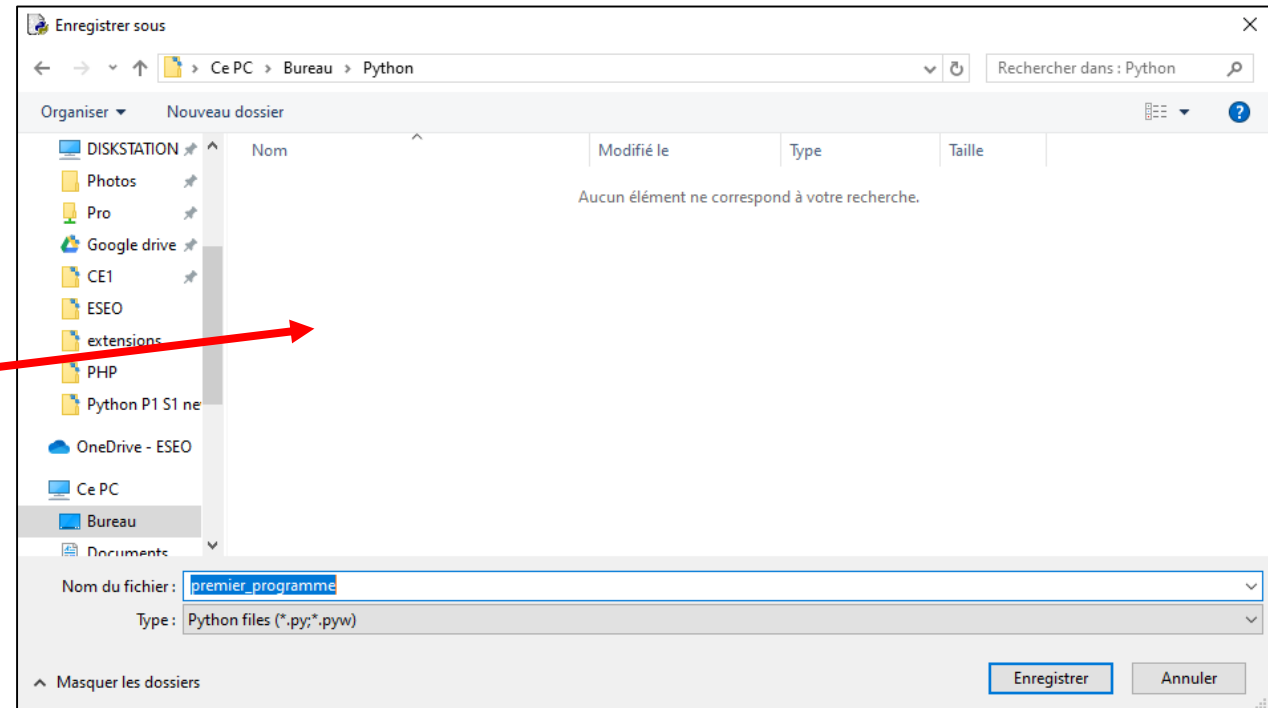
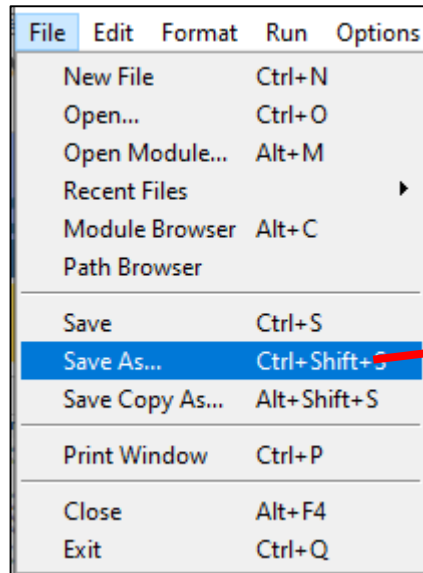
2. Dans le menu File, on choisit l'entrée *New File* (raccourci: Ctrl+N). Une nouvelle fenêtre s'ouvre.



Programme dans un fichier

Voici la démarche :

3. On commence par sauvegarder ce fichier script : Dans le menu File, on choisit l'entrée *Save AS* (raccourci: Ctrl+Shift+S). L'enregistrer dans **votre dossier Python** sous le nom que vous avez choisi : *premier_programme.py* par exemple . **Veillez à organiser votre travail ! (C:/Python/3-input-print/ex1.py)**



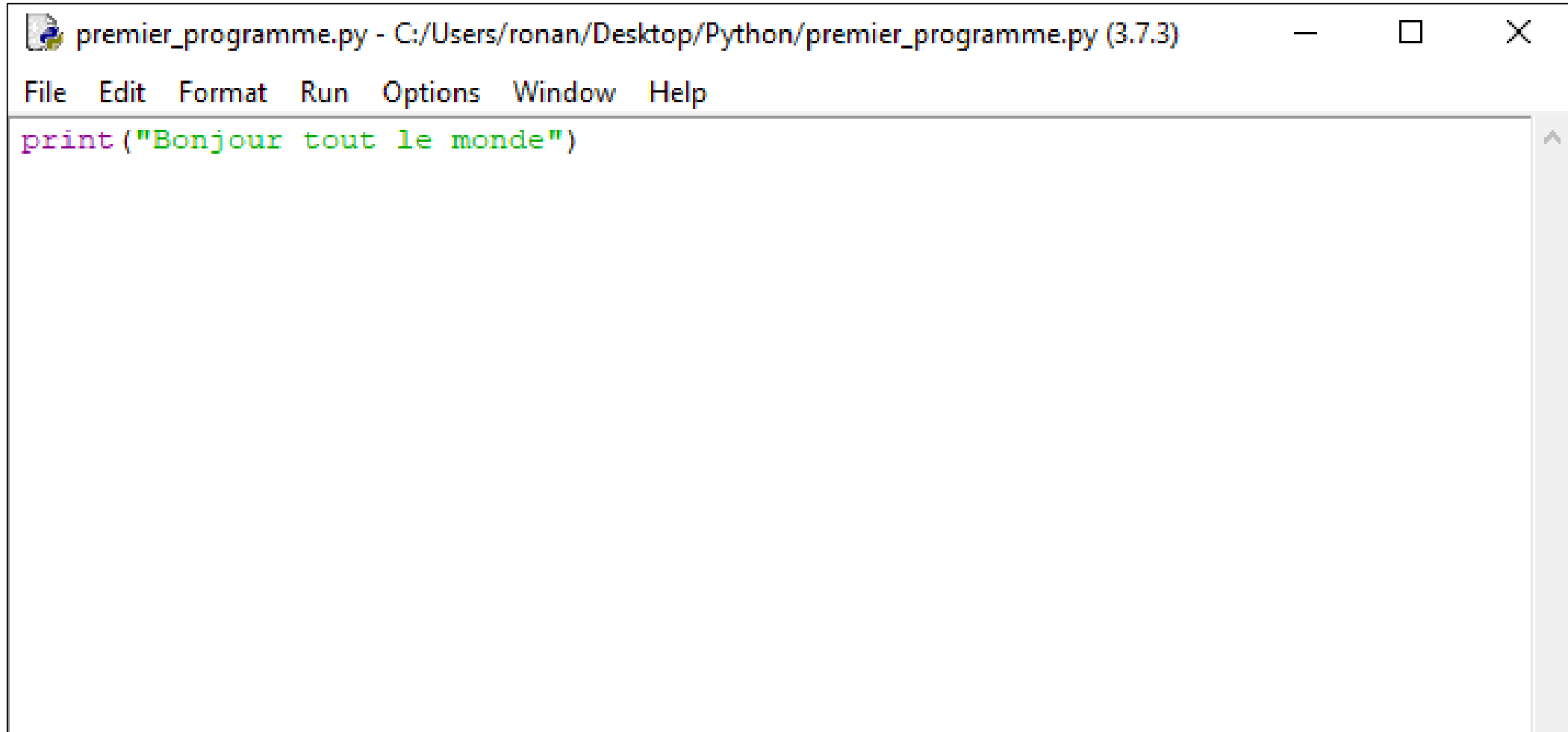
Et veillez à choisir un dossier de sauvegarde que vous pourrez retrouver d'une séance à l'autre ! (Cloud, clé USB, etc...)

Programme dans un fichier

Voici la démarche :

4. On peut alors écrire les instructions dans la fenêtre script.

Exemple: Taper dans la fenêtre script l'instruction *`print("bonjour tout le monde")`*

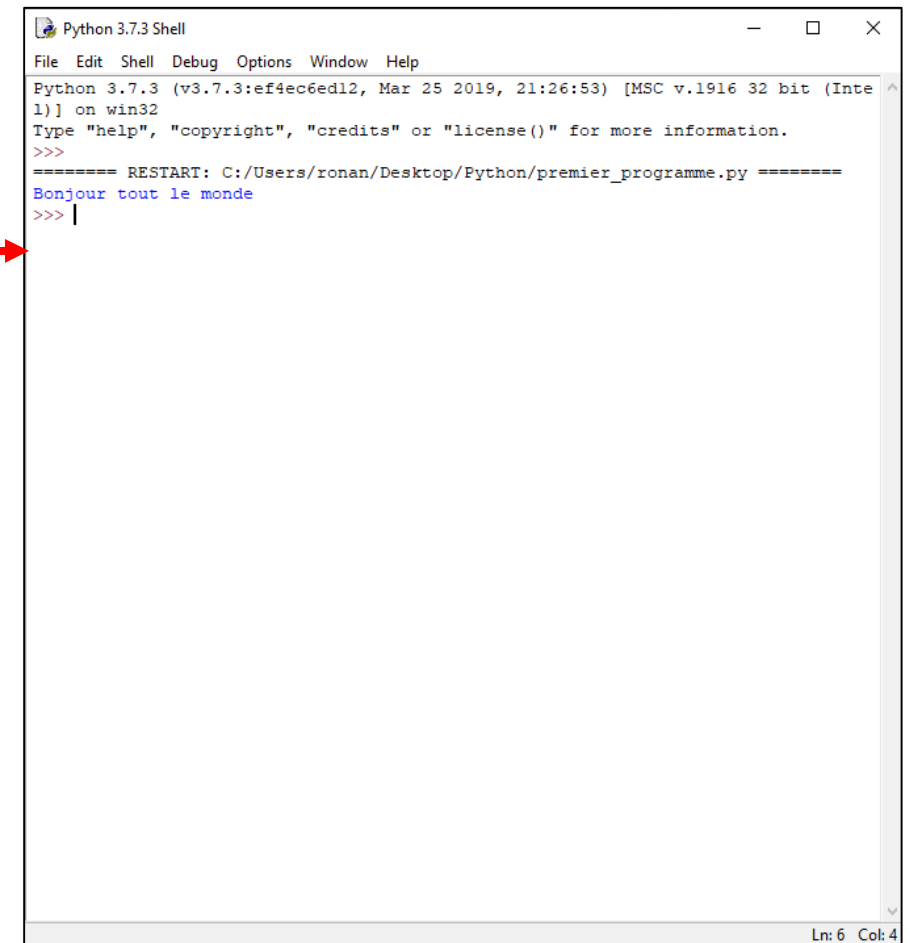
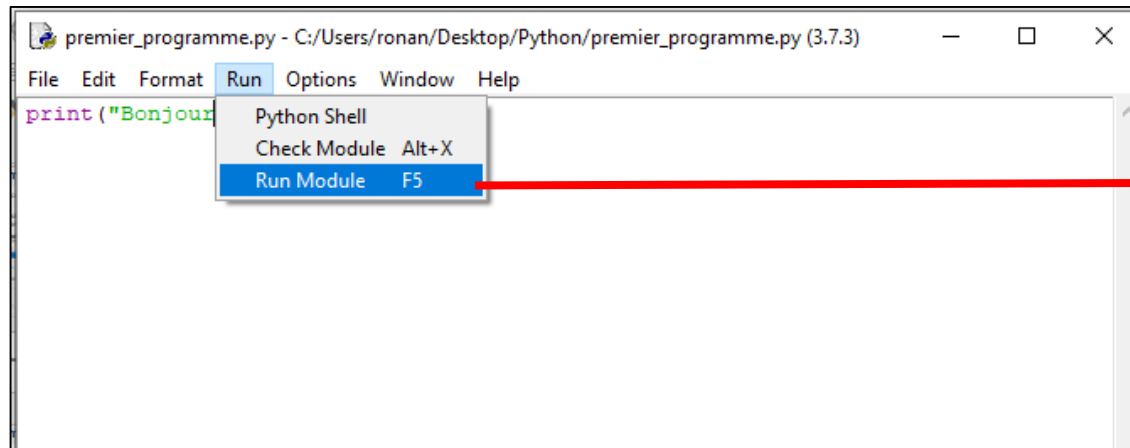


```
premier_programme.py - C:/Users/ronan/Desktop/Python/premier_programme.py (3.7.3)
File Edit Format Run Options Window Help
print("Bonjour tout le monde")
```

Voici la démarche :

5. Dans le menu Run, on choisit l'entrée ***Run Module (Raccourci F5)***. On bascule alors dans la console Shell.

Le fichier script est alors exécuté et la console reste ouverte pour que vous puissiez voir le résultat ou les **erreurs éventuelles**.



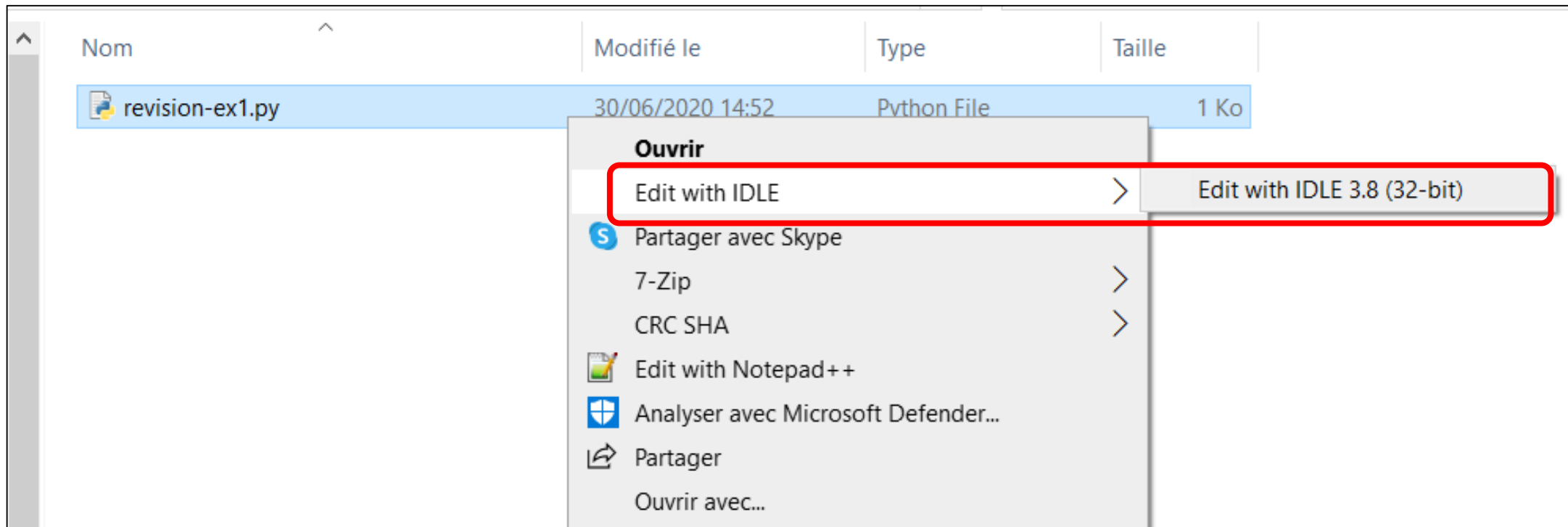
Programme dans un fichier

Une fois que vous avez fini de travailler sous Python, vous clôturez toutes les fenêtres de IDLE

Il vous reste votre fichier premier_programme.py enregistré sous Z:\ (par exemple)

Pour le ré-ouvrir votre fichier sous IDLE :

Clic droit > Edit with IDLE





Comment structurer correctement son programme

Comment structurer correctement son programme

1/ Un programme commence par le nom du programme et/ou une courte description

Programme calcul TVA

2/ on initialise les constantes

TAUXTVA = 0.20

3/ on initialise les variables , c'est-à-dire qu'elles reçoivent une valeur initiale

nb = 0

4/ on écrit les instructions du programme : l'ordre des instructions est primordial car elles sont exécutées dans l'ordre dans lequel elles apparaissent dans le programme. On dit que l'exécution est **séquentielle**.

Une fois que le programme a fini une instruction, il passe à la suivante.

Programme calcul TVA

initialisation

TAUXTVA = 0.20

nb = 0

instructions

Comment débbugger son programme : la trace

La trace d'un programme représente la valeur des différentes informations d'un programme durant son exécution.

Il est indispensable de vérifier la trace d'un programme afin de vérifier qu'il fonctionne, afin de le tester.

1- Choisir les variables sur lesquelles on va effectuer le test

2- effectuer la trace : pour chaque variable testée, calculer la valeur obtenue après chaque instruction du programme

- Soit à la main (sur une feuille)
- Soit en utilisant différents `print()` sur la variable à tester, tout au long du programme

```
print("test nb : ", nb)
```

3- analyser les résultats obtenus : correspondent-ils à ce que l'on attendait ?



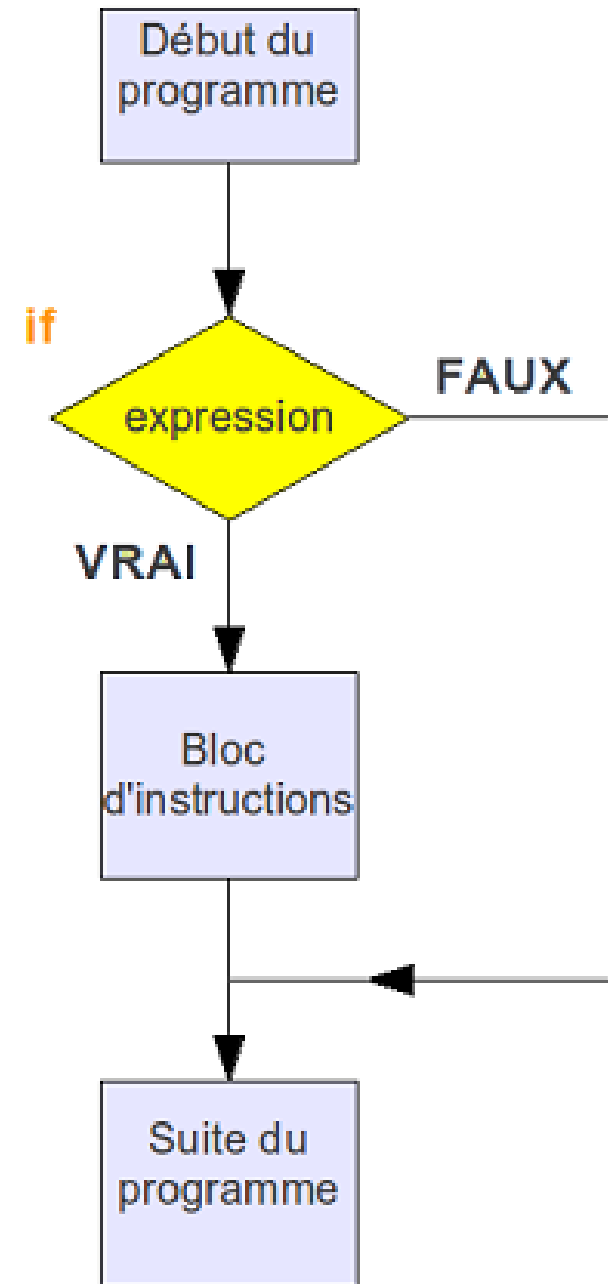
Structures conditionnelles

Condition minimale if

La condition if permet d'exécuter

- une ou plusieurs instructions dans un cas,
- d'autres instructions dans un autre cas

if va servir d'aiguillage et permettre au programme de suivre des chemins différents selon les circonstances



Condition minimale if

Syntaxe :

```
if a > 0:
```

le test conditionnel se compose, dans l'ordre :

- du mot clé if qui signifie « si » en anglais ;
- de la condition proprement dite, $a > 0$ dans l'exemple;

les conditions qui se trouvent entre if et les deux points sont appelés des **prédicats**.

- du signe **deux points**, « : », qui termine la condition et est indispensable : Python affichera une erreur de syntaxe si vous l'oubliez.

Les opérateurs de comparaison

Les conditions doivent nécessairement introduire de nouveaux opérateurs, dits **opérateurs de comparaison**

Opérateur	Signification littérale
<	Strictement inférieur à
>	Strictement supérieur à
<=	Inférieur ou égal à
>=	Supérieur ou égal à
==	Égal à
!=	Différent de

l'égalité de deux valeurs est comparée avec l'opérateur « == » et non « = ».

« = » est en effet l'opérateur d'affectation et ne doit pas être utilisé dans une condition

Condition minimale if

```
# Programme testant le if
```

```
a = 5
```

```
if a > 0:
```

```
    print("a est supérieur à 0.")
```

Code dans un fichier essai.py

1. commentaire décrivant qu'il s'agit du premier test de condition.
2. Cette ligne affecte la valeur 5 à la variable a.
3. Ici se trouve le test conditionnel (if a > 0:)
4. Ici se trouve l'instruction à exécuter dans le cas où a est supérieur à 0.
Cette instruction (et les autres instructions à exécuter s'il y en a) est indentée (décalée vers la droite)

Et nous allons exécuter le fichier en appelant le fichier dans l'exécuteur

```
a est supérieur à 0.
```

```
>>>
```

Condition minimale if

Exemple avec plusieurs instructions

précédemment, notre bloc n'était constitué que d'une seule instruction.

Rien ne nous empêche de mettre plusieurs instructions dans ce bloc, par exemple :

```
a = 5
b = 8
if a > 0:
    # On incrémente la valeur de b
    b += 1
    # On affiche les valeurs des variables
    print("a =", a, "et b =", b)
```

Code dans un fichier
essai.py

```
a = 5 et b = 9
>>>
```

Résultat dans la
console

Condition complète if, elif, else

L'instruction else:

Le mot-clé else, qui signifie « sinon » en anglais, permet de définir une première forme de complément à notre instruction if

```
a = 5
if a > 0:
    print("a est supérieur à 0.")
else:
    print("a est inférieur ou égal à 0.")
```

Et nous allons exécuter le fichier en appelant le fichier dans l'exécuteur

```
a est supérieur à 0.
```

```
>>>
```

Python exécute soit l'un, soit l'autre, et jamais les deux.

Notez 4 points importants sur cette instruction else:

- Elle doit se trouver **au même niveau d'indentation** que l'instruction if qu'elle complète.
- Elle **se termine** par **deux points**.
- Elle est **facultative**
- Elle ne peut figurer **qu'une fois**, clôturant le bloc de la condition. Deux instructions else dans une même condition ne sont pas envisageables.

Condition complète if, elif, else

L'instruction elif:

Le mot clé elif est une contraction de « else if », que l'on peut traduire par « sinon si ».

```
a=0
if a > 0: # Positif
    print("a est positif.")
elif a < 0: # Négatif
    print("a est négatif.")
else: # Nul
    print("a est nul.")
```

- Si a est strictement supérieur à 0, on dit qu'il est positif ;
- sinon si a est strictement inférieur à 0, on dit qu'il est négatif ;
- sinon, (a ne peut qu'être égal à 0), on dit alors que a est nul.

a est nul.

>>>

Notez 5 points importants sur cette instruction elif

- Elle est sur le même niveau **d'indentation** que le if initial.
- Elle se termine aussi par **deux points**.
- Entre le elif et les deux points se trouve une **nouvelle condition**
- Elle est **facultative**
- Vous pouvez mettre **autant de elif que vous voulez** après une condition en if.

Le mot-clé and

Il arrive souvent que nos conditions doivent tester plusieurs prédicats.

- le mot clé **and** qui signifie « et » en anglais, permet de tester plusieurs prédicat à la fois
- Le ET a le même sens en informatique que dans le langage courant. Pour que "Condition1 ET Condition2" soit VRAI, il faut impérativement que Condition1 soit VRAI et que Condition2 soit VRAI. Dans tous les autres cas, "Condition 1 et Condition2" sera faux.

```
a = 5
# on cherche à tester à la fois si a est supérieur ou égal à 2 et
# inférieur ou égal à 8
if a >= 2 and a <= 8:
    print("a est dans l'intervalle.")
else:
    print("a n'est pas dans l'intervalle.")
```

Code dans un fichier .py

```
a est dans l'intervalle.
>>>
```

Résultat dans la
console

Le mot-clé or

Il arrive souvent que nos conditions doivent tester plusieurs prédicats.

- le mot clé **or** qui signifie « ou » permet de tester un prédicat ou un autre prédicat
- Il faut se méfier un peu plus du OU. Pour que "Condition1 OU Condition2" soit VRAI, il suffit que Condition1 soit VRAIE ou que Condition2 soit VRAIE, ou que les deux conditions soient VRAIES.

```
a = 5
#on cherche à savoir si a n'est pas dans l'intervalle. La variable ne se trouve pas dans
l'intervalle si elle est inférieure à 2 ou supérieure à 8
if a<2 or a>8:
    print("a n'est pas dans l'intervalle.")
else:
    print("a est dans l'intervalle.")
```

Code dans un fichier .py

a est dans l'intervalle.

>>>

Résultat dans la
console

Le mot-clé not

Le **not** inverse une condition : `not(Condition1)` est VRAI si `Condition1` est FAUX, et il sera FAUX si `Condition1` est VRAI.

C'est l'équivalent pour les booléens du signe "moins" que l'on place devant les nombres.

```
a = 5
#on cherche à savoir si a n'est pas inférieur à 2 -> donc on cherche à savoir si a est
supérieur à 2 !
if not(a<2):
    print("a n'est pas dans l'intervalle.")
else:
    print("a est dans l'intervalle.")
```

Code dans un fichier .py

```
a n'est pas dans l'intervalle.
>>>
```

Résultat dans la
console