



LES FONCTIONS (et gestion des erreurs)

1. LA GESTION DES ERREURS:

✓ **ACTIVITE:** Créez le fichier *erreurs.py* puis enregistrez-le dans le répertoire THONNY. Commençons par inviter l'utilisateur à entrer un entier...

```
x=int(input("Entier: "))
```

A l'exécution du programme, entrons autre chose qu'un entier...

```
x=int(input("Entier: "))  
ValueError: invalid literal for int() with base 10: ''
```

Extrait de la documentation officielle de PYTHON:

A **TypeError** occurs when an operation or function is applied to an object of inappropriate type (exemple déjà rencontré: "3"+3).

ValueError is thrown when a function's argument is of an inappropriate type (c'est le cas ici).

Impossible de concevoir un programme qui n'anticipe pas les erreurs de saisie (**ValueError**). PYTHON, comme les autres langages permet de gérer les erreurs avec les blocs **try** et **except** (**try** et **catch** dans les autres langages):

```
# syntaxe:  
try:  
    instructions  
except ValueError:  
    instructions en cas  
    d'erreur sur le type de  
    variable dans le bloc try
```

```
# tapez ce code dans erreurs.py  
try:  
    x=int(input("Entier: "))  
except ValueError:  
    print("Pas un entier...")
```

L'erreur est maintenant prise en compte par le programme

C'est mieux, le programme est quitté sans erreur mais quitté quand même! Comment forcer l'utilisateur à saisir une entrée du type attendu? Une solution s'impose: la boucle **while**.

```
# tapez et testez ce code dans erreurs.py  
while True: # cette condition est toujours réalisée!  
    try:  
        x=int(input("Entier: "))  
        break # tout va bien, on peut sortir de la boucle  
    except ValueError:  
        print("Ce n'est pas un entier, recommencez...")  
print("Sortie de boucle, l'entier saisi vaut ",x)
```

✓ AIDE MEMOIRE, GESTION DES ERREURS:

QUELQUES TYPES D'ERREURS	
Type d'erreur :	exemples:
ValueError	<pre>>>> int("NSI") >>> from math import * >>> sqrt(-4)</pre>
ZeroDivisionError	<pre>>>> 10/0</pre>
TypeError	<pre>>>> "3"+3</pre>
SYNTAXE DES BLOCS TRY ET EXCEPT	
Pour:	on tape en Python:
exécuter des instructions en cas d'erreur	<pre>try: instructions except TypeErreur: instructions en cas d'erreur dans le bloc try</pre>
forcer le programme à recommencer tant que l'erreur n'est pas levée (raised)	<pre>while True: try: instructions break except TypeErreur: instructions en cas d'erreur dans le bloc try</pre>

2. LES FONCTIONS:

Pour arrondir le nombre π on peut procéder ainsi:

- Multiplier π par 10 à la puissance du nombre de décimales désiré puis ajouter 0,5
- Ne garder que la partie entière de ce nombre
- Diviser cette partie entière par 10 à la puissance du nombre de décimales désiré

Pour arrondir le nombre π deux chiffres après la virgule on peut donc taper le code suivant (après importation de pi du module math): `int(pi*10**2+0.5)/(10**2)`

Téléchargez le fichier *arrondis.py* sur le site, placez-le dans le dossier THONNY puis ouvrez-le dans le logiciel.

✓ **ACTIVITE:** Observez attentivement la syntaxe de la déclaration des trois premières fonctions. Dans la zone consacrée au corps du programme, testons successivement ces différents appels de fonctions (`\u03c0` est le caractère π en Unicode UTF-8).

```
arrondir_pi()
```

Appel de la fonction qui affiche la variable x dans la console

```
a=arrondir_pi_2(5)  
print("\u03c0=",a)
```

La tentative d'affectation puis d'affichage échoue car la fonction ne prend aucune valeur.

```
a=arrondir_pi_3(5)  
print("\u03c0=",a)
```

L'affectation fonctionne, la troisième fonction prend une valeur (de type float) avec l'instruction return

La dernière fonction doit nous fournir l'arrondi de **n'importe quel nombre** (*float*) avec le **nombre de décimales** voulu (*int*) passés tous deux en paramètres. Par exemple, l'appel/affectation de la fonction dans le corps du programme: `a=arrondir_nombre(5.765832,3)` doit afficher après `print("arrondi:",a)`: **arrondi: 5.766** dans la console.

- Terminez dans le fichier *arrondis.py* l'écriture de la fonction **arrondir_nombre** puis testez si son appel/ affectation puis son affichage fonctionnent.
- Modifiez le corps du programme de manière à ce que ce soit l'utilisateur qui saisisse lui-même le nombre à arrondir et le nombre de décimales désiré. Testez...
- Coupez cette partie du code (le corps du programme) du fichier *arrondis.py* puis enregistrez. Créez un nouveau fichier *principal.py* dans le même répertoire puis tapez:

```
from arrondis import arrondir_nombre  
  
# Collez ici le code du b)
```

Si tout fonctionne, vous venez de créer votre premier module!

✓ AIDE MEMOIRE, FONCTIONS:

SYNTAXE DES DECLARATIONS ET APPELS DE FONCTIONS	
Déclaration et appel d'une fonction sans argument en entrée ni valeur retournée en sortie (on parle de procédure).	<pre>def ma_fonction(): instructions # appel: ma_fonction()</pre>
Déclaration et appel d'une fonction sans argument en entrée avec valeur retournée en sortie. <i>ma_fonction() est maintenant une variable de type int, float, str, bool...</i>	<pre>def ma_fonction(): instructions return variable # appel: ma_fonction()</pre>
Déclaration d'une autre fonction avec arguments (arg) en entrée puis appel avec paramètres (param) correspondants. <i>Les paramètres et arguments peuvent être de type int, float, str, bool..</i>	<pre>def ma_fonction(arg1,arg2,arg3,..): instructions return variable # appel: ma_fonction(param1,param2,param3,..)</pre>
Appeler une fonction définie dans un autre programme (module): <i>fichier.py</i>	<pre>from fichier import ma_fonction # appel: ma_fonction(param1,param2,param3,..)</pre>
QUELQUES REGLES A CONNAITRE	
<ul style="list-style-type: none">+ Le programme ignore la déclaration d'une fonction (définie avec le mot clé def) tant qu'elle n'est pas appelée.+ En PYTHON, les fonctions se déclarent avant leur appel.+ L'instruction return permet à la fonction d'être utilisée comme variable de type, <i>int, float, str, bool...</i>+ Dans le corps d'une fonction, on peut appeler une autre fonction.+ Une fonction peut être passée en paramètre d'une autre fonction.+ Une variable définie à l'intérieur du corps d'une fonction est locale, elle est invisible du corps du programme.	