LANGAGE ET PROGRAMMATION (1)

NSI | 04

LES FONCTIONS (et gestion des erreurs)

[Frédéric PEURIERE]

1. LA GESTION DES ERREURS:

✔ **ACTIVITÉ:** Créez le fichier *erreurs.py* puis enregistrez-le dans votre dossier de travail. 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 Thonny puis
entrez autre chose qu´un entier:
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	>>> .....
	
	>>> from math **import** *
	>>> .....

ZeroDivisionError	>>> .....
TypeError	>>> .....
SYNTAXE DES BLOCS TRY ET EXCEPT	
Pour:	on tape en Python:

exécuter des instructions en cas d´erreur	`try:`
	`  instructions`
	`except TypeErreur:`
	`  instructions en cas d´erreur`
	`  dans le bloc try`

forcer le programme à recommencer **tant que** l´erreur n´est pas levée (raised)	`while True:`
	`  try:`
	`  instructions`
	`  break`
	`  except TypeErreur:`
	`  instructions en cas d´erreur`
	`  dans le bloc try`

2. LES FONCTIONS :

✔ FAIRE UN CALCUL :

Nous prendrons comme exemple le calcul de l'énergie cinétique d'un véhicule afin d'illustrer quelques notions utiles sur l'utilisation des fonctions en `python`.

L'énergie cinétique d'un objet de masse m se déplaçant à la vitesse v est: $Ec = \frac{1}{2}mv^2$

Nous allons créer une fonction `energie_cinetique` qui calcule sa valeur à partir des paramètres masse et vitesse.

Définissons la fonction :

```
def energie_cinetique(m, v):
    return 0.5*m*v**2
```

Appelons ensuite cette fonction :

```
energie_cinetique(2, 3)
```

Une fonction ne peut exécuter qu'un `return`, tout `return` est **définitif** :

```
def exemple():
    return "Bonjour"
    print("Ce code ne sera pas exécuté(dead code)")

# appel
exemple()
```

Tout ce qui suit le `return` n'est pas exécuté, on parle de code *mort* (*dead code* en anglais).

✔ PROCEDURES :

Il est possible de définir des fonctions qui ne renvoient pas de valeur, on les nomme alors parfois procédure. Ce type de fonction n'utilise pas l'instruction `return`. On peut s'en servir par exemple pour faire des affichages.

```
def affiche_ec(m, v):
    ec = 0.5*m*v**2
    print("L'énergie cinétique est:", ec, "J")
```

Appel de la procédure : `affiche_ec(1,10)`

✔ PORTÉE DES VARIABLES :

```
def double(x):
    calcul=2*x
    return calcul

# retourne 14
double(7)
# retourne une erreur
print(calcul)
```

Les variables définies dans les fonctions, ses paramètres ou autres, sont appelés des **variables locales**, par opposition aux **variables globales**, qui sont définies dans flot d'exécution du programme.

`calcul` est une **variable locale** accessible seulement dans le corps de la fonction.

✓ 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).	`def ma_fonction():`
	`  instructions`
	
	`# appel:`
	`ma_fonction()`

Déclaration et appel d´une fonction sans argument en entrée avec valeur retournée en sortie.
 	
ma_fonction() est maintenant une variable de type int, float, str, bool...
	`def ma_fonction():`
	`  instructions`
	`  return variable`
	
	`# appel:`
	`ma_fonction()`

Déclaration d´une autre fonction avec arguments (arg) en entrée puis appel avec paramètres (param) correspondants.
 	
Les paramètres et arguments peuvent être de type int, float, str, bool..
	`def ma_fonction(arg1,arg2,arg3,..):`
	`  instructions`
	`  return variable`
	
	`# appel:`
	`ma_fonction(param1,param2,param3,..)`

QUELQUES REGLES A CONNAITRE

+ On utilise des fonction lorsque une même tâche doit être réalisée plusieurs fois dans un programme.
+ 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, *int, float, str, bool...*
+ 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.