INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Leçon sur la notion de fonction

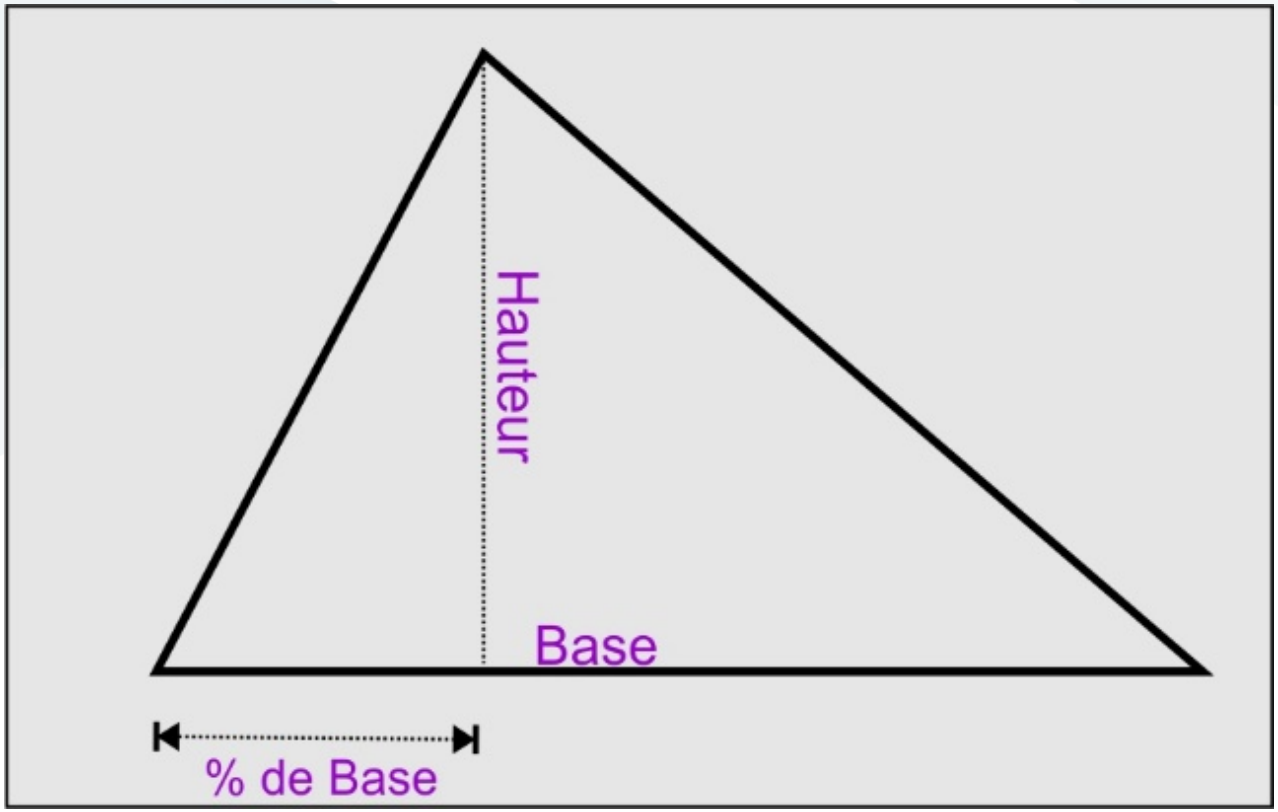
Université de Toulouse

Du script à la fonction..

- **Script : ensemble de commande mise dans un fichier**
- **Pourquoi un simple script peut poser problème :**
  - Multiplication des variables
  - Effet de bord : réutilisation ou modification de variables portant le même nom dans plusieurs scripts…
  - Réutilisation à plusieurs reprise et dans des conditions différentes d'une partie de code
- **Solution : mettre le tout dans un « boîte étanche » :**
  - ⇒ Notion de **FONCTION**
  - ⇒ Un ensemble de fonctions forme un **MODULE**
  - ⇒ Exemple le module Math contient les fonctions log, sqrt, trunc,…

Exemple

- **Problème à résoudre :**
  - Calculer le rapport Surface/Périmètre du triangle dont on connait la hauteur, la base et la position relative de la hauteur sur le base (en %)



La solution (sans fonction)

Saisie des valeurs nécessaires

Calcul assez simple

Affichage

```
import math


Hauteur = input("Hauteur du triangle :")
Hauteur = float(Hauteur)
Base  = input("Base du triangle :")
Base= float(Base)
ratio = input("Position en % de la hauteur sur la base  :")
ratio = float(ratio)/100


L1 = math.sqrt((ratio*Base)**2 + Hauteur**2)
L2 = math.sqrt(((1-ratio)*Base)**2 + Hauteur**2)
Perimetre = L1 + L2 + Base
Surface = Hauteur * Base /2
Rapport   = Surface /Perimetre


print("Le rapport Surface/Base de ce triangle vaut", Rapport)
```

Création d'une fonction (1)

- Pour en faire une fonction, il suffit de rajouter

```
def Rapport_Tri():
```

- Mais l'exécution pose souci :

```
Hauteur = float(Hauteur)
                ^
IndentationError:
unexpected indent
```

Func_Rapport.py

```
def Rapport_Tri():
import math
Hauteur = input("Hauteur du triangle :")
Hauteur = float(Hauteur)

Base  = input("Base du triangle :")
Base= float(Base)

ratio = input("Position en % de la hauteur sur la base  :")
ratio = float(ratio)/100

L1 = math.sqrt((ratio*Base)**2 + Hauteur**2)
L2 = math.sqrt(((1-ratio)*Base)**2 + Hauteur**2)

Perimetre = L1 + L2 + Base
Surface = Hauteur * Base /2
Rapport   = Surface /Perimetre

print("Le rapport Surface/Base de ce triangle vaut ",Rapport)
```

Création d'une fonction (2)

- L'indentation est crucial en Python
- Il est obligatoire de mettre une tabulation pour délimiter les structures algorithmiques
- Une fonction est une structure algorithmique

```
import math

def Rapport_Tri():
    Hauteur = input("Hauteur du triangle :")
    Hauteur = float(Hauteur)

    Base = input("Base du triangle :")
    Base= float(Base)

    ratio = input("Position en % de la hauteur sur la base :")
    ratio = float(ratio)/100/100

    L1 = math.sqrt((ratio*Base)**2 + Hauteur**2)
    L2 = math.sqrt(((1-ratio)*Base)**2 + Hauteur**2)

    Perimetre = L1 + L2 + Base
    Surface = Hauteur * Base /2
    Rapport = Surface /Perimetre

    print("Le rapport Surface/Base de ce triangle vaut",Rapport)

print("Le rapport Surface/Base de ce triangle vaut ", Rapport)
print("Bonjour le monde!!!")
```

appartient à la fonction Rapport_Tri

n'appartient à la fonction

Création d'une fonction (2)

- Maintenant que la fonction existe correctement on peut utiliser le fichier.
- L'exécution ne donne rien à part « bonjour le monde »
- => pas d' appel à la fonction
- Ajout de l'appel

Func_Rapport.py

```
import math
def Rapport_Tri():
    Hauteur = input("Hauteur du triangle :")
    Hauteur = float(Hauteur)

    Base  = input("Base du triangle :")
    Base= float(Base)

    ratio = input("Position en % de la hauteur sur la base  :")
    ratio = float(ratio)/100

    L1 = math.sqrt((ratio*Base)**2 + Hauteur**2)
    L2 = math.sqrt(((1-ratio)*Base)**2 + Hauteur**2)

    Perimetre = L1 + L2 + Base
    Surface = Hauteur * Base /2
    Rapport   = Surface /Perimetre

    print("Le rapport Surface/Base de ce triangle vaut",Resu)
print("Bonjour le monde!!!")
Rapport_Tri()
```

Quel intérêt ?

- Je peux appeler plusieurs fois ma fonction dans mon programme sans avoir à dupliquer le code
- Je peux mettre mes fonctions dans un fichier séparé pour en faire un bibliothèque (ex : bibliothèque math = ensemble de fonctions)
  - Nécessité d' « import » le fichier bibliothèque
  - Mon fichier peut être commun à plusieurs autres fichiers
- Les variables internes à ma fonction ne sont plus visibles de l'extérieur
  - Notion de variables locales

Du passage d'arguments

- **Où est le problème :**
  - La fonction est bloquante…je ne peux pas l'utiliser sans demander à l'utilisateur de saisir les valeurs
  - J'ai « perdu » les informations importantes dont le résultat
- **Comment le résoudre ?**
  - Se passer des fonctions

    si je dois faire 10 fois le calcul, je réécris 10 fois le code (inconcevable!)

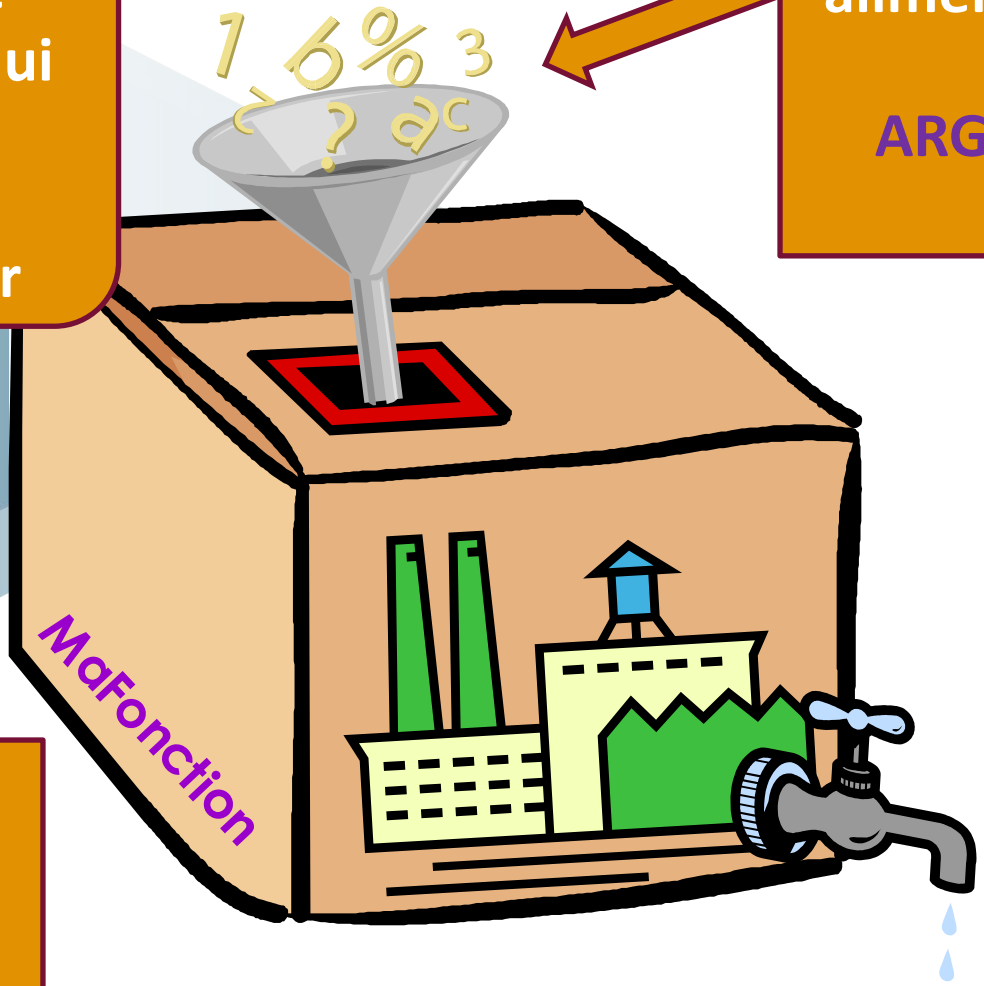
    je vais vite m'encombrer avec des tonnes de variables locales avec des effets de bords néfastes
  - Utiliser la **technique du passage d'arguments**

Une fonction c'est quoi ?

- **Ceci est une fonction :**

C'est une « usine » qui rend un service particulier

La fonction peut être alimentée par des données en entrée : ARGUMENTS d'ENTREE



La fonction peut fournir des données en sortie : ARGUMENTS de SORTIE

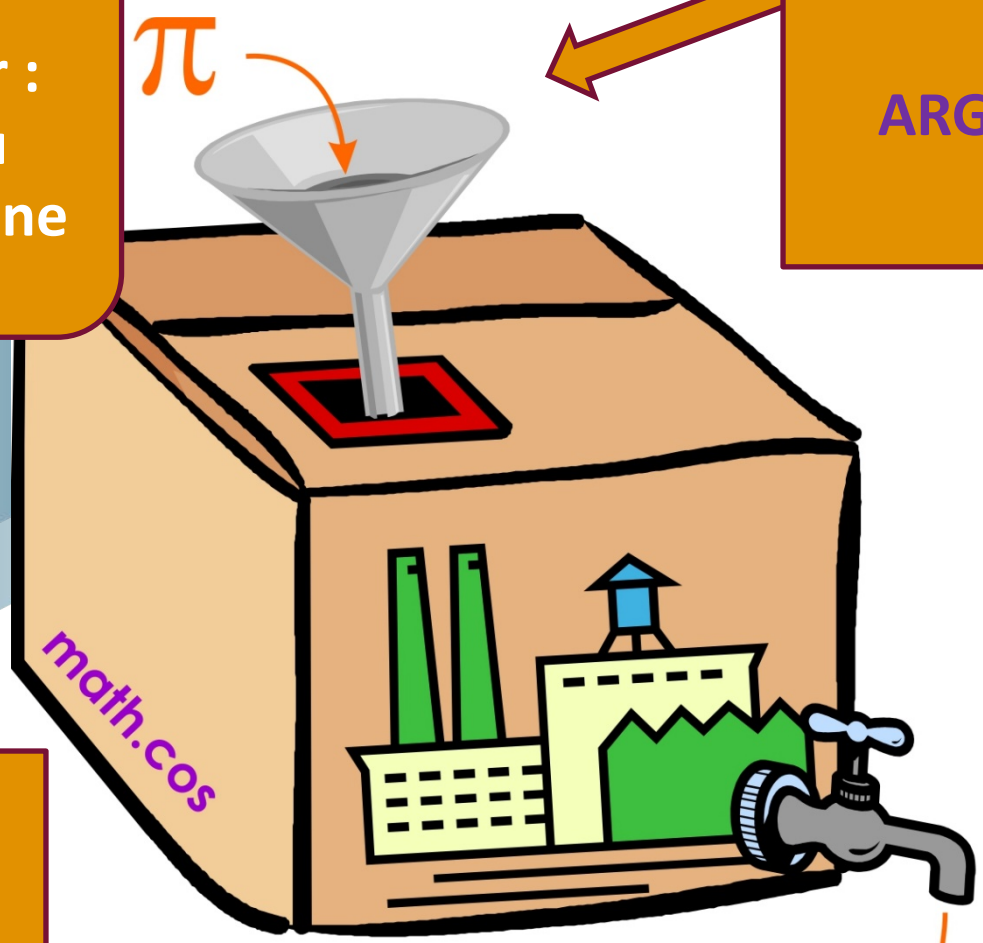
Ce qui se passe dans l'usine est « invisible »

Université de Toulouse

Exemple : la fonction COS

Service particulier : Calcul du cosinus d'une valeur

On donne une valeur (ici π) comme ARGUMENTS d'ENTREE



On récupére le cosinus de cette valeur en ARGUMENTS de SORTIE

On ne sait pas comment il fait ce calcul

Utilisation de math.cos ()

```
import math
math.cos(3.14)
```

Utilisation directe => le résultat est perdu (mais affiché)

```
X = math.cos(3.14)
```

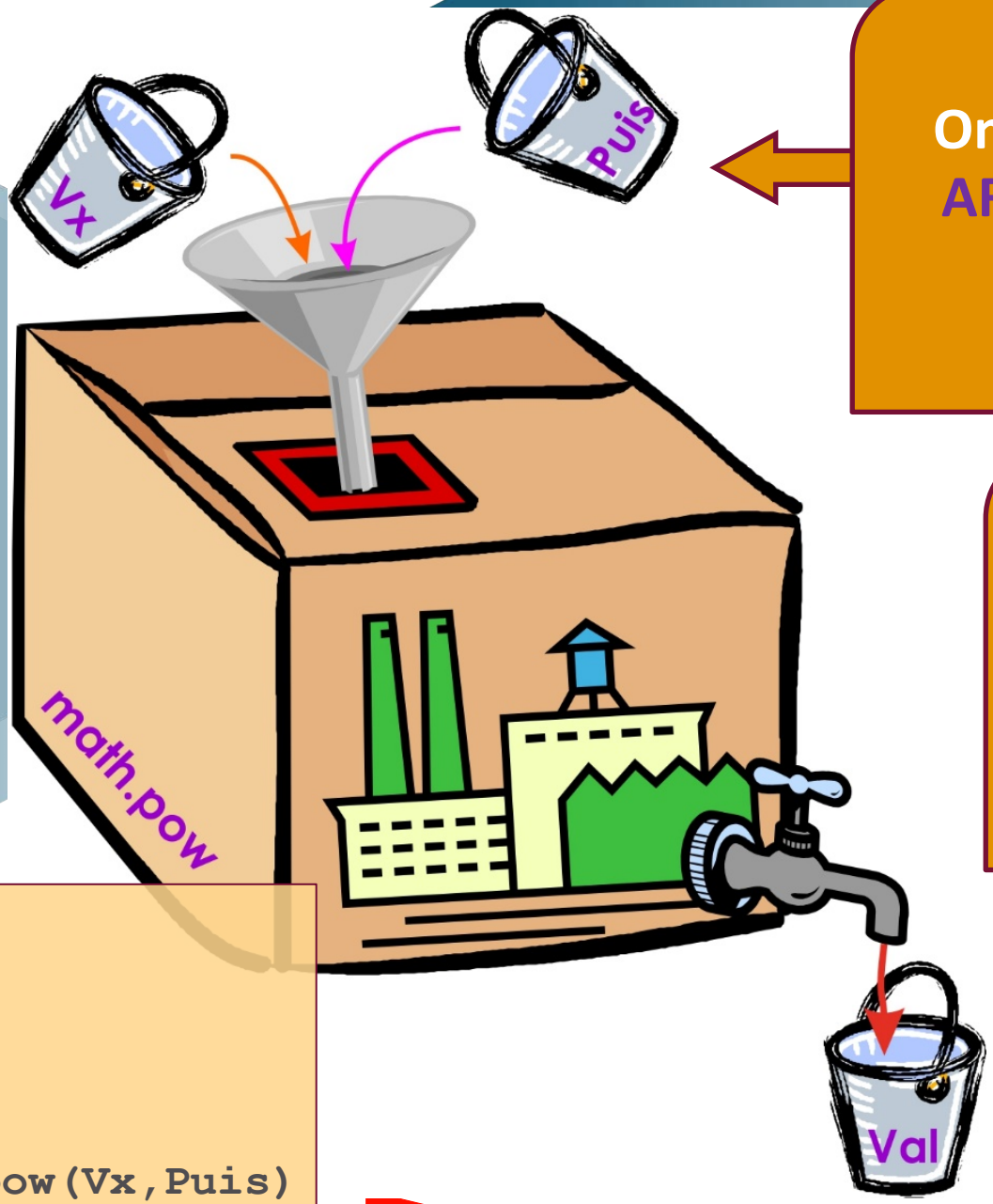
Utilisation directe
Le résultat est stocké dans la variable X

```
Angle = input(' Donnez un angle :')

Y = math.cos(Angle)
```

Utilisation avec une variable fournie en argument d'entrée .
Le résultat est stocké dans la varaible Y

```
print('Le cosinus de ',Angle,' est ,'Y)
```

Exemple : la fonction POW



On verse le contenu des **ARGUMENTS d'ENTREE** dans la fonction

On récupère l' **ARGUMENTS de SORTIE** dans la variable d'affectation

```
Puis = 3

Vx = 12.2

Val = math.pow(Vx,Puis)
```


Université de Toulouse

La Fonction vue de l'intérieur (1)

- Supposons la fonction :

```
def Test(ArgA,ArgB) :

    Calcul = ArgA ** ArgB
    Ret = Calcul
    return(Rel)
```

Ne pas oublier les tabulations

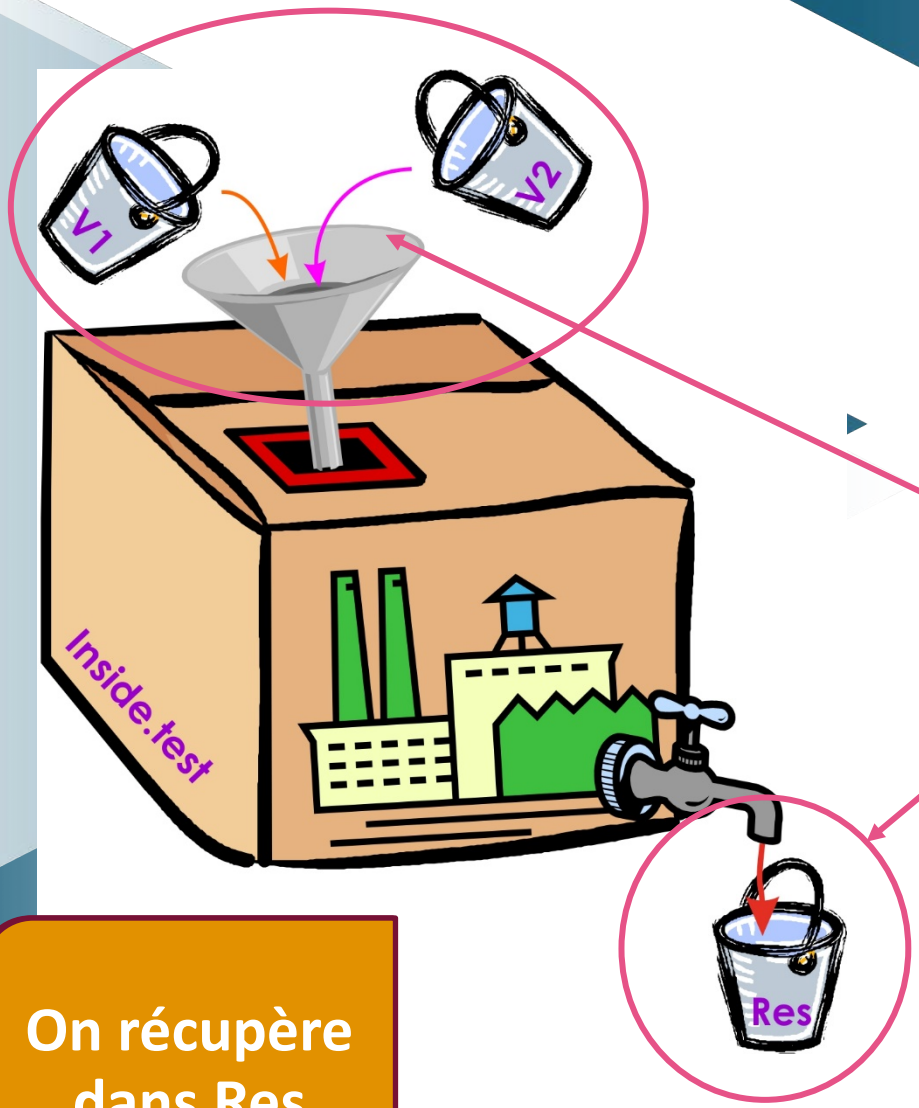
Fichier Inside.py

- Et son utilisation

```
V1 = input('Donnez le nombre réel : ')
V1 = float (V1)

V2 = input('Donnez un nombre entier: ')
V2 = int(V2)

Res = Test(V1,V2)

print('Avec V1=',V1,'et V2=',V2)
print('Ma fonction test me donne ',Res)
```

Fichier TestInside.py

La Fonction vue de l'intérieur (2)



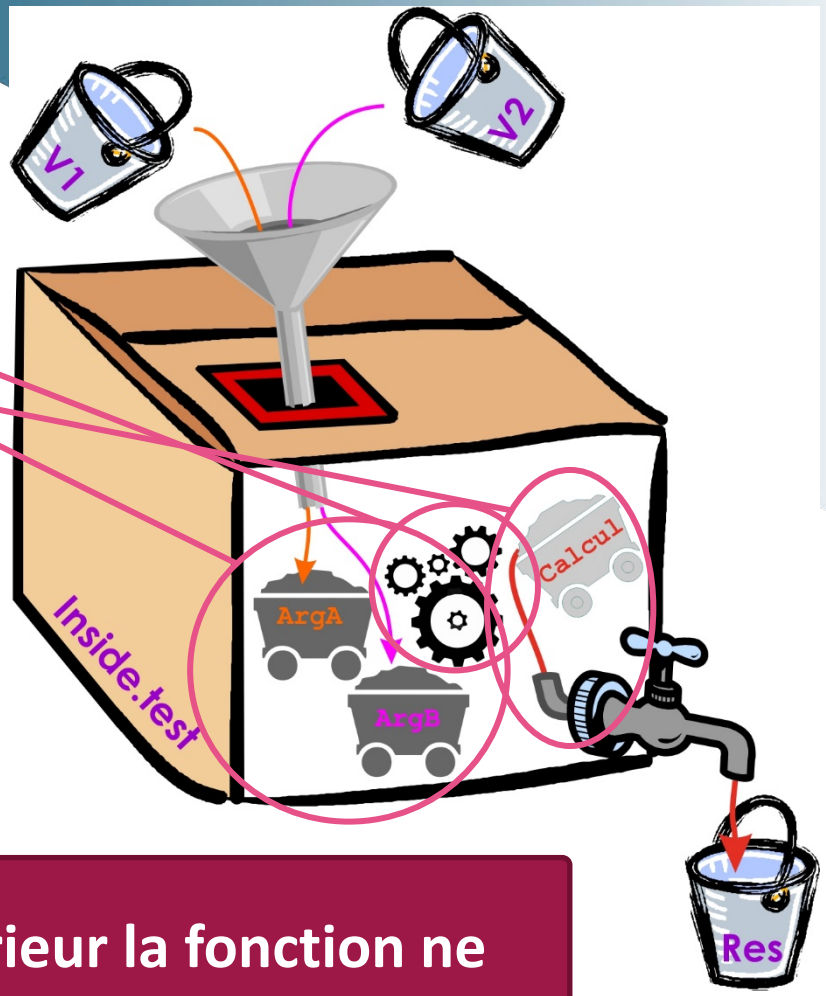

```
import Inside



V1 = input('Donnez le nombre réel : ')
V1 = float (V1)

V2 = input('Donnez un nombre entier: ')
V2 = int(V2)

Res = Test(V1,V2)

print('Avec V1=',V1,'et V2=',V2)
print('Ma fonction test me donne ',Res)
```

Fichier TestInside.py

On récupère dans Res

La Fonction vue de l'intérieur (3)

```
def test(ArgA,ArgB) :

    Calcul = ArgA ** ArgB
    Ret = Calcul
    return(Ret)
```



Le résultat du calcul est envoyé vers la sortie de « l'usine »

A l'intérieur la fonction ne connait ni V1, ni V2
Elle ne voit pas plus Res

Fonctions : aspect syntaxe (1)

- **Un fonction débute par :**

```
def NomdeLaFonction (Arg1,Arg2,..) :
```

  - **def** : ouverture de la structure
  - **NomdeLaFonction** : libre mais si possible *intelligent*
  - **(Arg1,Arg2,..)** : liste et nommage des arguments
  - **:** : fin de la déclaration de la fonction

- **Ensuite on écrit l'ensemble des instructions de la fonction avec tabulation à chaque début de ligne**

- **Le fonction se termine dès qu'il n'y a plus de tabulation**

Fonctions : aspect syntaxe (2)

- **Un fonction « retourne » par :**

```
return(val)
```

  - **val** : une des variables locales ; a priori le résultat du calcul
  - **return** : n'importe où mais logiquement à la fin

- **Une fonction ne retourne qu'une seule variable…mais cette valeur peut être une liste de valeur**

Ce n'est pas interdit par la syntaxe mais il est conseillé de ne faire ni INPUT ni PRINT dans une fonction

Fonctions : aspect syntaxe (3)

- Pour retourner une liste de variable :

```
return(val1,val2,…)
```

- Attention il n'y a qu'un seul return. Les donnée sont alors « regroupées » dans une liste.
- La récupération suite à l'appel se fait évidement dans la même ordre :

```
X,Y,… = mafonction(…)
```

- X recevra la valeur de val1, Y celle de V2,…

Exemple de retour multiple

- Dans cet exemple,

  Rac2 sera affecté

  avec Demi,

- Rac3 avec Tiers

  Rac4 avec Quart

```
def puissant(Nombre):
    Demi=Nombre**(1/2)
    Tiers = Nombre**(1/3)
    Quart = Nombre**(1/4)
    return (Demi,Tiers,Quart)


XValue=input("Quel nombre dois-je traiter :")
XValue = float(XValue)
Rac2,Rac3,Rac4 = puissant(XValue)

print("La racine carrée est ",round(Rac2,2))
print("La racine cubique est ",round(Rac3,2))
print("La racine quatrième est ",round(Rac4,2))
```

Il est important de faire la différence entre les variables internes (Demi,Tiers,Quart) et les externes (Rac2,Rac3,Rac4)

Fonctions : règle d'usage

- Il sera **interdit** d'utiliser le même nom pour les variables à l'intérieur et à l'extérieur de la fonction
  - Pour les arguments d'appel *(Arg1,Arg2)* différent de *Var1* et *Second*
  - Pour les arguments de sortie *Val* différent de Ret1
- Conseil : utiliser systématique *Arg1,Arg2,…Ret1,Ret2,…*

```
#...déclaration de la fonction
def Mafonc (Arg1,Arg2,..)
          …
          …

          Return (Ret1)



# Utilisation de la fonction

Var1 = …
Second = …
…
Val = Mafonc (Var1,Second)

Var2 = …
Arg1 = …
Ret1 = Mafonc (Arg1,Var2)
```

Import d'un module

- **Rappel sur les possibilités d'import**
  - Import du module dans l'environnement

```
import Mon_module
```

  - Import du module dans l'environnement et renommage

```
import Mon_module as Mod
```

  - Import d'une fonction d'un module dans l'environnement

```
from Mon_module import Ma_Fonc
```

  - Import de toutes les fonctions d'un module dans l'environnement

```
from Mon_module import *
```

Il peut y avoir plusieurs fonctions dans un module

Qu'est ce qui ne va pas ?

- Trouvez les erreurs

```
Def racine12(Nombre)

Nombre = input('Donnez un nombre ')
Nb = float (Nombre)/12

Rac = math.sqrt(Nb)
```

Fichier Jeu.py

```
import Jeu


Valeur = input('Donnez la valeur: ')
Valeur = float (Valeur)

Resu = Jeu.racine12(Valeur)

print(Valeur,'divisé par 12 vaut',Nb)

print('La racine de',Valeur,'est',Resu)
```

Premier essai

- Ecrire un module (Firstmod.py) qui contient la fonction BigDouble qui multiplie par 2.1 un nombre passé en argument d'entrée et qui retourne le résultat trouvé
- Importer directement le module dans la console et essayer BigDouble(12). Tester les différentes formes d'import
- Ecrire un script (Test_Firstmod.py) qui demande à l'utilisateur de saisir un nombre, qui le divise par 2.1 puis qui appelle BigDouble avec cette variable. Vérifier que le nombre issu de BigDouble est bien le même que celui saisi par l'utilisateur en affichant la différence entre ces 2 variables