LE MENU PYTHON

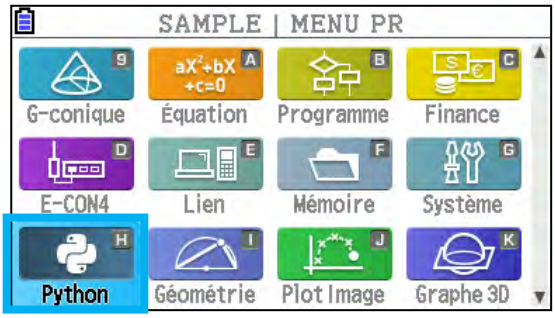
Python est un langage de programmation interprété permettant une initiation aisée aux concepts de base de la programmation structurée. **Python** désigne également l'interpréteur qui permet de lire les scripts qui sont écrits en langage Python. Les calculatrices **Graph 35+E II** et **Graph 90+E** utilise ***MicroPython*** une version adaptée de **Python 3** pour les microcontrôleurs.

Nous aborderons dans ce chapitre le menu **Python** des calculatrices graphiques **CASIO Graph 35+E II et Graph90+E** en nous servant de différents exercices en accord avec le programme du lycée ainsi que d'exemples traités dans le supérieur.

1. Menu Python, manipulations à partir de l'écran d'accueil et création d'un premier programme

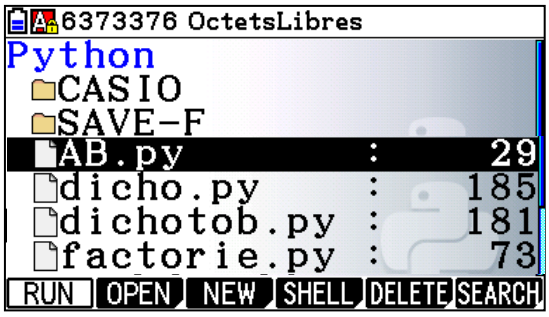
1. Accès au menu PYTHON :

Appuyer sur la touche [MENU]
Sélectionner le menu **PYTHON** à l'aide des flèches.
Valider par la touche [EXE]



Appuyer sur le raccourci [ALPHA] [S⇔D] correspondant à la lettre **{H}** en haut à droite de l'icône.

On accède ainsi à l'écran d'accueil du menu où l'on trouve l'ensemble des fichiers au format **.py** mémorisés dans la calculatrice ainsi qu'à la taille des fichiers en Ko (kilo octet).

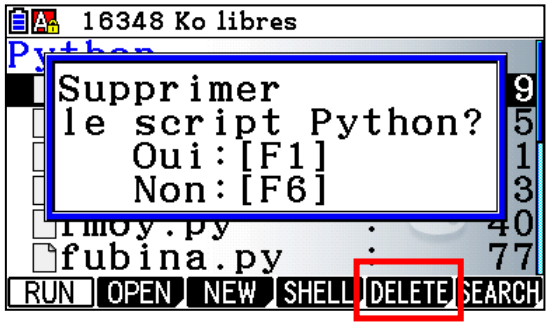


On observe la taille de chaque fichier, par exemple AB.py est un programme Python de taille 29 Ko.

2. Suppression d'un fichier

Application : Effacer le fichier *AB.py*

A partir de l'écran d'accueil, il est possible d'effacer un fichier en se positionnant sur le fichier concerné à l'aide des flèches ▲▼ et en sélectionnant [F5] **{DELETE}**.



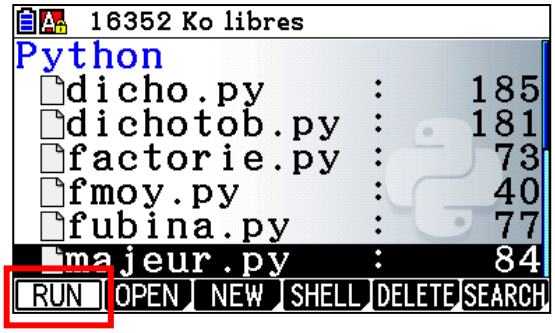
Il faut confirmer avec la touche [F1] **{Oui}**

Le fichier AB.py est alors supprimé de la liste.

3. Exécution d'un programme

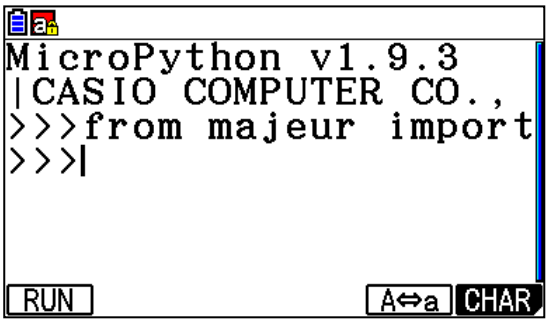
Application : Exécuter le programme *majeur.py*

A partir de l'écran d'accueil, il est possible d'exécuter un programme en se positionnant sur le fichier concerné à l'aide des flèches ▲▼ et en sélectionnant [F1] **{RUN}**.



Le programme est exécuté et il apparaît une ligne de commande **>>> from majeur import ***

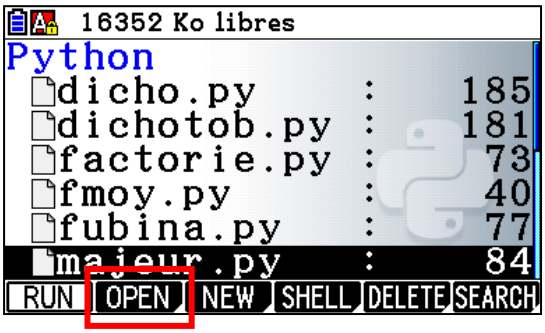
On a ainsi téléchargé le programme **majeur** et on se retrouve dans le **SHELL** c'est à dire l'interpréteur des commandes Python.
Pour sortir du SHELL appuyer sur [EXIT].



4. Accès au contenu d'un programme existant

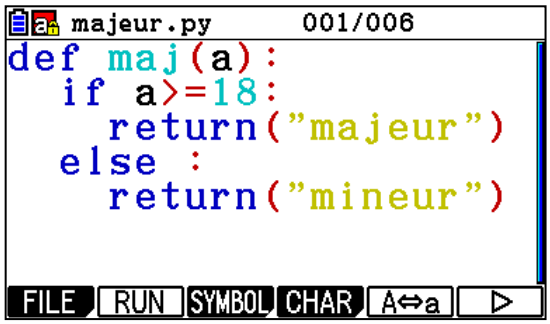
Application : Accéder au contenu du programme Python *majeur.py*

A partir de l'écran d'accueil, il est possible d'accéder au contenu d'un programme en se positionnant sur le fichier concerné à l'aide des flèches ▲▼ et en sélectionnant [F2] **{OPEN}**.



On peut alors modifier le programme et le sauvegarder avec la touche [F1] **{FILE}** puis [F1] **{SAVE}** ou [F2] **{SAVE.AS}.** On peut aussi l'exécuter avec la touche [F2] **{RUN}**.

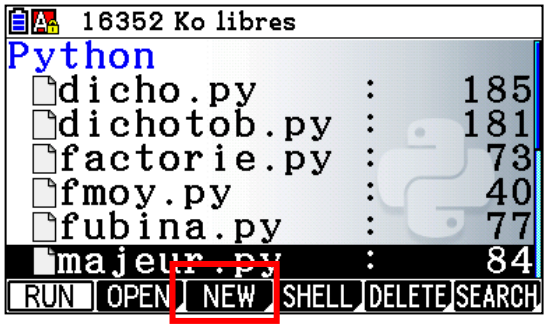
On peut également sauvegarder les modifications en utilisant la touche [EXIT] et en confirmant l'enregistrement [F1] **{Oui}.**



5. Création et exécution d'un premier programme - Catalogue – Mode alphabétique

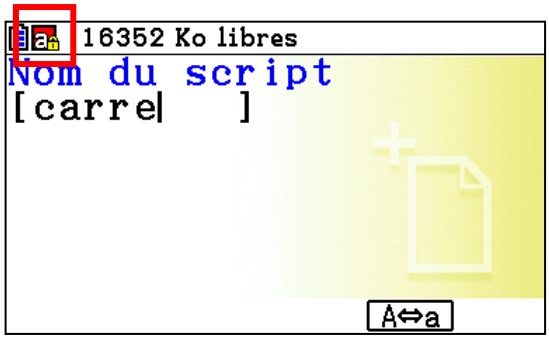
Application : Créer un programme (script) Python appelé *carre.py* qui définit la fonction carrée.

A partir de l'écran d'accueil, il est possible de créer un nouveau programme en sélectionnant [F3] **{NEW}**.



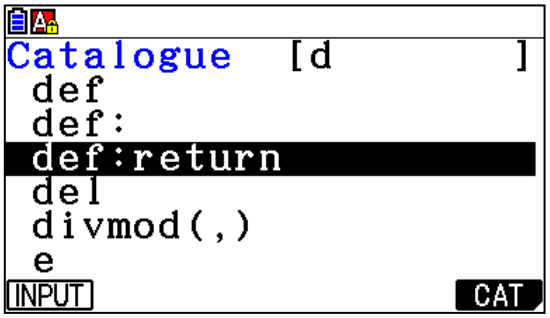
Entrer le nom du script : **carre** avec les touches alphanumériques et valider avec [EXE].

Par défaut, le clavier est bloqué en mode alphanumérique minuscule. Le symbole apparait en haut à gauche de l'écran de la Graph 90+E. Sur la Graph 35+E II, c'est le curseur qui indique le mode alphabétique minuscule.
Si on souhaite écrire en majuscule, il suffit d'appuyer sur la touche [F5] $\{A \Leftrightarrow a\}$.

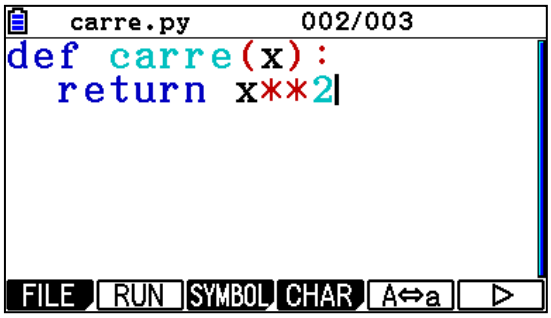


CATALOG:

Dans le SHELL ou l'éditeur de programme, on peut utiliser le catalogue [SHIFT] [4] (CATALOG) puis taper dans la barre de recherche les premières lettres de ce que l'on cherche, ici « d ». Il suffit ensuite de se déplacer à l'aide des flèches [▲] [▼] sur l'expression souhaitée et terminer par [EXE].



On complète ensuite avec le nom de la fonction **carre** (voir le mode alphabétique ci-dessous) en précisant son argument ici « **x** » et le calcul à effectuer **x** [^] [2].



On remarque que le symbole puissance du clavier est remplacé par l'écriture de la puissance ** en Python.
On remarque aussi la **coloration syntaxique**.
def et **return** en **bleu foncé** sont des expressions réservées à l'écriture d'instructions en Python.

MODE ALPHABETIQUE:

Dans le SHELL ou l'éditeur de programme, les touches alphanumériques s'activent avec [ALPHA], dans ce mode il est possible d'écrire une seule lettre.
Les touches [SHIFT] [ALPHA] permettent de bloquer le clavier en mode alphabétique, le cadenas apparaît sur l'icône de la Graph 90+E en haut à gauche de l'écran. Sur la Graph 35+E II c'est le curseur qui change de forme et qui permet de voir si l'on est en mode alphabétique, majuscule, minuscule ou en mode numérique.
[F5] **{A ⇔ a}** permet de passer des majuscules aux minuscules.

Graph 90+E

- 1 minuscule: [ALPHA] a
- Plusieurs minuscules: [SHIFT] [ALPHA] a
- 1 majuscule: [ALPHA] [F5] A
- Plusieurs majuscules: [SHIFT] [ALPHA] [F5] A

Graph 35+E II

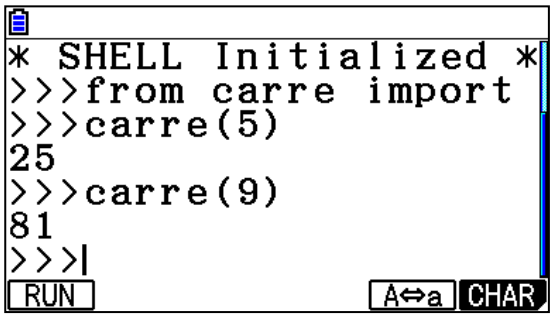
- 1 minuscule: [ALPHA] a
- Plusieurs minuscules: [SHIFT] [ALPHA] a a
- 1 majuscule: [ALPHA] [F5] A
- Plusieurs majuscules: [SHIFT] [ALPHA] [F5] A A

On exécute enfin le programme avec [F2] **{RUN}**.
Il faut confirmer l'enregistrement du programme avec [F1] **{Oui}.** Le programme s'exécute et on se retrouve dans le **SHELL**.

On voit que le programme **carre** a été importé :
from carre import *
Pour activer le programme dans le SHELL il faut appeler **carre** en passant une valeur en paramètre.

Par exemple, on tape dans le **SHELL** ***carre(5)***
On voit alors l'affichage du résultat souhaité.
On peut recommencer avec d'autres valeurs.

On sort du SHELL avec [EXIT].



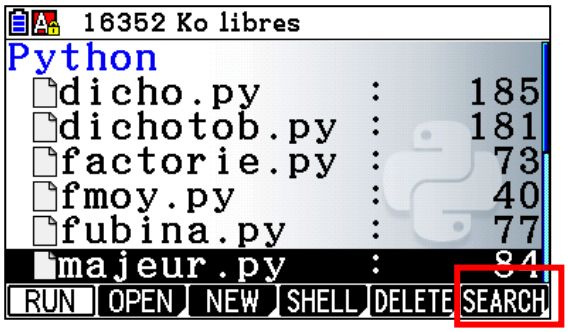
6. Rechercher d'un fichier

Application : On souhaite trouver un programme nommé *mystere.py*

A partir de l'écran d'accueil, il est possible de chercher un fichier en sélectionnant la touche [F6] **{SEARCH}**.

Le nom du fichier recherché va être rentré en majuscule en validant avec [EXE].

Si le fichier n'existe pas le message « **Non trouvé** »



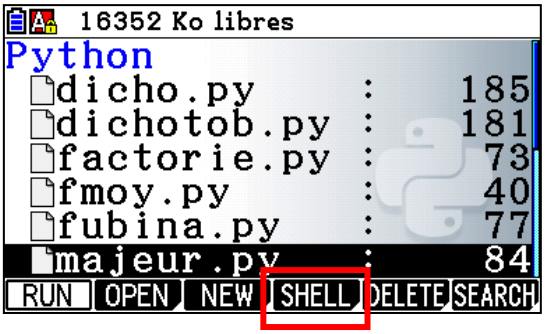
2. Le SHELL: l'interpréteur

Le **SHELL** correspond à l'interpréteur Python.

On peut exécuter des instructions Python dans le **SHELL** sans écrire un programme avec un éditeur.
A partir de l'écran d'accueil, on peut ouvrir le SHELL en sélectionnant la touche [F4] **{SHELL}**.

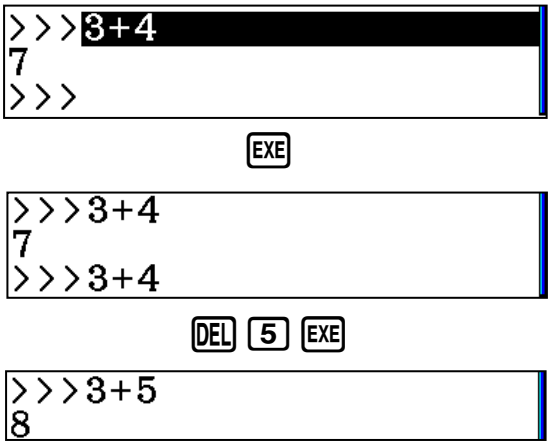
On voit apparaître le message ***** **SHELL Initialized** ***** ainsi que le prompte (invite de commande) **>>>**

A partir de l'écran d'accueil, il est possible d'accéder au **SHELL** qui est un interpréteur pour exécuter des instructions Python en sélectionnant la touche [F4] **{SHELL}**.



1. Le Copier/Coller dans le SHELL

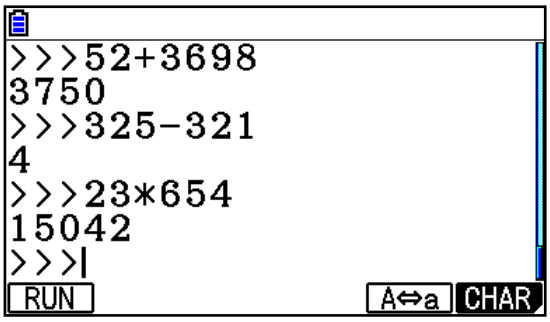
A l'aide des flèches ▲▼ on peut se placer sur une ligne du SHELL à copier puis appuyer sur EXE. Cela permet de recopier la ligne sur la dernière invite de commandes puis de la modifier.



2. Calculs simples dans le SHELL, quotient, reste

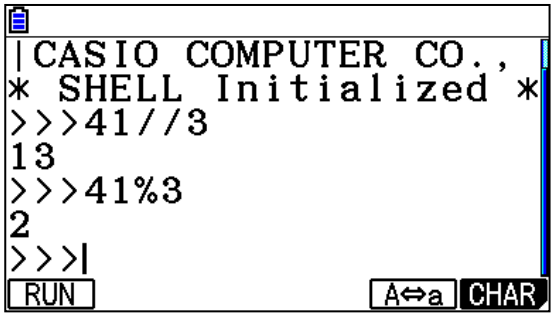
Dans le SHELL on peut effectuer toutes les opérations algébriques classiques : addition, soustraction, multiplication et division.

On peut faire d'autres calculs comme une division euclidienne chercher le quotient et le reste.

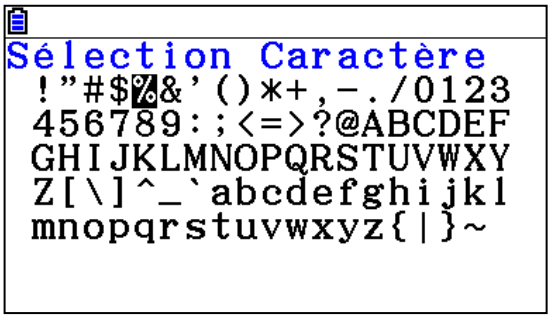


Application *: On souhaite obtenir le quotient et le reste de la division de 41 par 3.*

Le quotient s'obtient à l'aide des touches (÷ ÷) qui donnent *//* et le reste à l'aide de l'instruction %.



Pour afficher le symbole %, on utilise F6 **{CHAR}** et on sélectionne % avec les flèches directionnelles ▶ puis faire EXE.



41//3 donne un quotient de 13

41 %3 donne un reste de 3

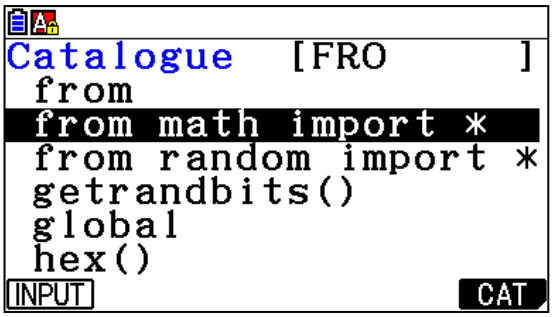
3. Les modules et les variables du menu python

Ce qui est présenté ci-dessous est aussi valable dans l'éditeur de programmes.

1. Le module math

Pour certains calculs avec des fonctions classiques comme la racine carrée ou les fonctions trigonométriques, il faut importer le module **math** avec l'instruction **from math import ***

On peut utiliser le catalogue pour écrire l'instruction avec les touches [SHIFT] [4] CATALOG
et taper dans la barre de recherche les lettres « FRO » puis se déplacer à l'aide des flèches ▲▼ sur la ligne souhaitée et terminer par [EXE]. Enfin, faire à nouveau [EXE] pour exécuter l'instruction.

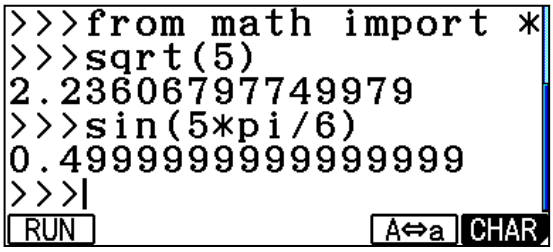


Application *: On souhaite obtenir une valeur approchée à 0.001 près des réels* $\sqrt{5}$ *et* $sin(\frac{5\pi}{6})$ *.*

En tapant les touches [SHIFT] [x^2] on obtient l'affichage **sqrt()** qui désigne la fonction **racine carrée** en Python. Ce qui donne $\sqrt{5} \approx 2.236$

En tapant [sin] [(] [5] [×] [SHIFT] [×10ˣ] [÷] [6] [)]

On obtient l'affichage $\boldsymbol{sin(5 * pi/6)}$ ***pi*** désigne le nombre π en Python.
On trouve alors $sin(\frac{5\pi}{6}) = 0.5$

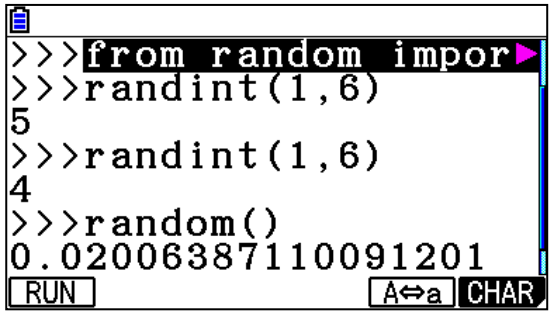


2. Nombres aléatoires, module random

Pour générer des nombres aléatoires, il faut utiliser le module random : **from random import ***

Application *: On souhaite générer un nombre entier entre 1 et 6 ainsi qu'un nombre réel compris strictement entre 0 et 1 .*

Pour obtenir un nombre entier entre 1 et 6 on utilise l'instruction **randint(,)** en passant par le catalogue.



Pour obtenir un nombre réel nombre compris strictement entre 0 et 1 on utilise l'instruction **random( )**

3. Les différents types de variables

Nous allons aborder à l'aide du **SHELL** les différents types de variables que peut gérer Python.

On dit que le langage Python a un **« typage dynamique»** car le type d'une variable est défini au moment de l'affectation.

Application *: Déterminer le type des variables a, b, c, d, et f lorsque l'on fait les affectations suivantes :*

$a = 2$
$b = 3.14$
$c = "c"$
$d = [1,2,3]$
$e = (1,2,3)$
$f = \{1,2,3\}$

```
>>>a=2
>>>b=3.14
>>>c="c"
>>>d=[1,2,3]
>>>e=(1,2,3)
>>>f={1,2,3}
```

Le signe = s'obtient avec les touches: SHIFT ⊙
Les crochets [] avec SHIFT + // SHIFT −
Les accolades {} avec SHIFT × // SHIFT ÷

On effectue les affectations pour les différentes variables avec = et à l'aide de l'instruction **type()** on obtient le type de chaque variable :

```
>>>type(a)
<class 'int'>
>>>type(b)
<class 'float'>
>>>type(c)
<class 'str'>
```

Pour a le type est **« entier »** (int)
Pour b le type est **« flottant »** (float)
Pour c le type est **« caractère »** (str)
Pour d le type est **« liste »** (list)
Pour b le type est **« n-uplet »** (tuple)
Pour c le type est **« ensemble »** (set)

```
>>>type(d)
<class 'list'>
>>>type(e)
<class 'tuple'>
>>>type (f)
<class 'set'>
```

Il existe également d'autres types comme par exemple le type booléen (bool) pour une variable qui peut prendre deux valeurs True ou False.

```
>>>g=True
>>>type (g)
<class 'bool'>
```

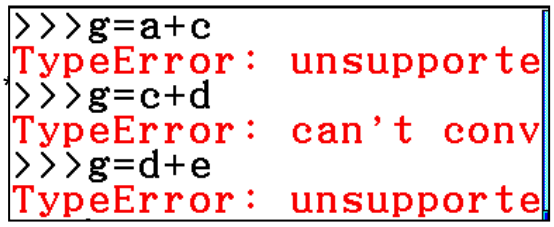
Si l'on fait un calcul avec une variable du type « entier » et une variable de type « flottant » et que l'on sauvegarde le résultat dans une autre variable celle-ci va avoir le type « flottant ».

```
>>>g=a+b
>>>type (g)
<class 'float'>
```

Si l'on fait des opérations avec des variables du même type on conserve le type.

```
>>>g=c+c
>>>type (g)
<class 'str'>
>>>g
'cc'
```

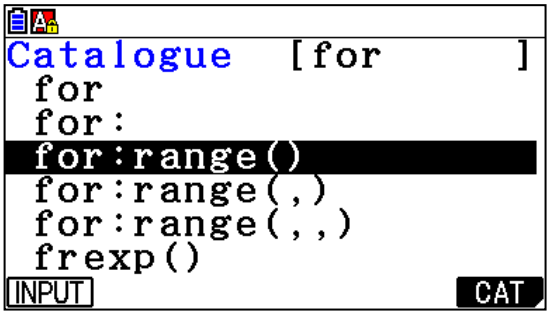
Par exemple, si on fait $g = c + c$ cela revient à concaténer 2 chaînes de caractères et on obtient alors que la variable g a pour contenu « cc » et qui est de type caractère « str ».



Par contre, en général on ne peut pas faire d'opérations avec des variables qui n'ont pas le même type, des messages d'erreurs s'affichent :
TypeError : unsupported types for - -add - - : 'int', 'str'
TypeError : can't convert 'list' object to str implicitlly

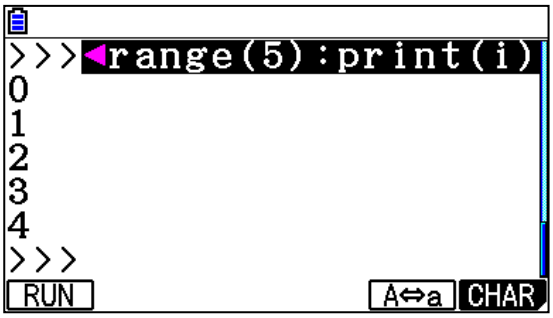
4. Les instructions de base: for, print, input, if

Il est possible de tester une instruction dans le SHELL à condition de l'écrire sur une seule ligne.
On retrouve ses instructions en sélectionnant le catalogue avec les touches SHIFT 4 (CATALOG)



Par exemple si on veut tester la boucle **for** pour se souvenir où elle commence et où elle s'arrête, on pourra saisir:

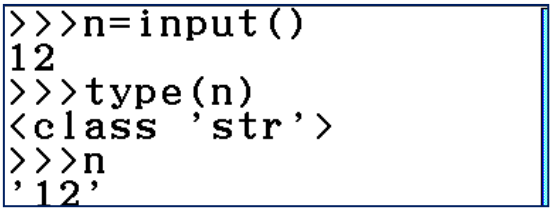
for i in range(5) : print(i)



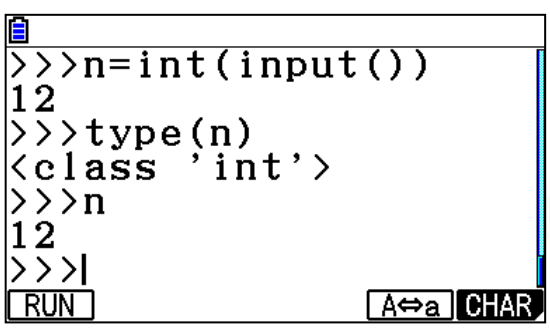
Pour trouver l'instruction **for**, sélectionner le catalogue avec les touches SHIFT 4 (CATALOG)
Puis taper dans la barre de recherche les lettres « FOR ». Se déplacer ensuite à l'aide des flèches ▲▼ sur la ligne **for : range()** et terminer par EXE.

Compléter la ligne avec **print(i)** que l'on trouve de la même manière dans le (CATALOG) puis appuyer à nouveau sur EXE pour valider l'instruction.

Essayons maintenant d'exécuter **n=input().** La calculatrice se met en attente d'une saisie par l'opérateur. Si on saisit le nombre 12 on voit que la variable **n** est alors du type **« str »** et non « **int** ».
L'affectation a enregistré la chaîne de caractères « 12 ».



Pour affecter un nombre entier il faut donc spécifier le type de variable en transformant la chaîne de caractère récupérée à l'aide de l'instruction **input()** en un entier. Pour cela on va utiliser l'instruction **int()** pour imposer le typage « **int** ».



L'instruction à exécuter est alors **n=int(input())** et on voit alors que pour la saisie du nombre 12 la variable n est bien du type « **int** » .

On peut utiliser l'instruction float() pour avoir un type « flottant » et str() pour transformer un nombre en chaîne de caractères.

```
>>>n=12
>>>float(n)
12.0
>>>str(n)
'12'
```

Pour réaliser des tests on fait appel aux instructions **if** ou **if else**. Les conditions du test sont par exemple de la forme **==** , **>=** ou **<=** pour comparer le contenu d'une expression avec une autre.

```
>>>if n==12:print("ok
ok
>>>
```

4. Edition de programmes

A la différence du SHELL, les instructions doivent ici être saisies sur plusieurs lignes. L'indentation (l'espace en début de ligne) est primordiale pour Python. Cela permet de marquer le début et la fin d'un bloc d'instructions (plus besoin de **IfEnd** / **WhileEnd** etc…). Tout ce qui est indenté par exemple après **def** fait partie de la fonction créée. L'indentation sur la calculatrice se fait automatiquement lors de la création d'une nouvelle ligne avec EXE après le symbole **:**
Cette indentation est par défaut égale à deux espaces mais il est possible d'en mettre

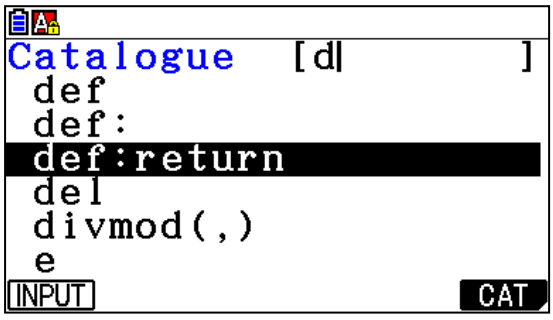
1. Création d'une fonction **def / return**, correction d'une erreur, différence entre **print** et **return**

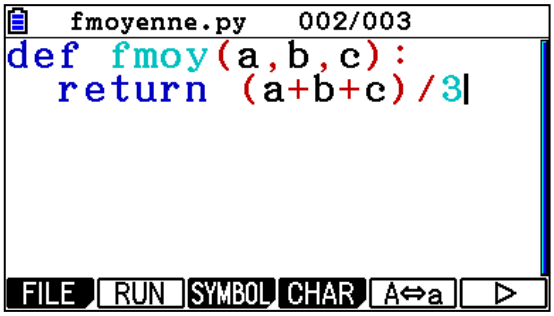
Application : Nous voulons créer une fonction qui calcul la moyenne de 3 nombres réels.

On créé le programme ***fmoyenne***

On utilise SHIFT 4 (**CATALOG**) pour aller chercher **def:return**. Le **return** est alors indenté automatiquement.

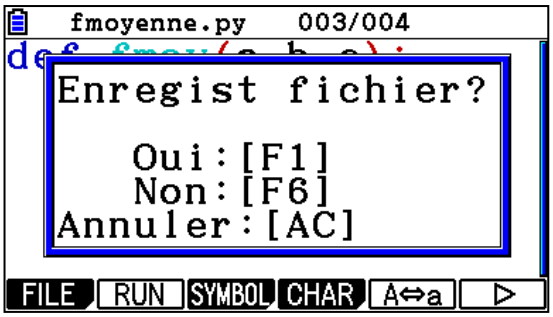
On définit la fonction nommée **fmoy** qui fait la moyenne de 3 réels a, b et c.



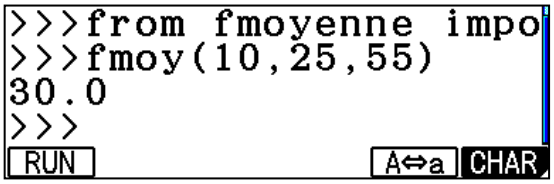


On lance le programme avec F2 **{RUN}**.

Il faut confirmer l'enregistrement du programme F1 **{Oui}**.

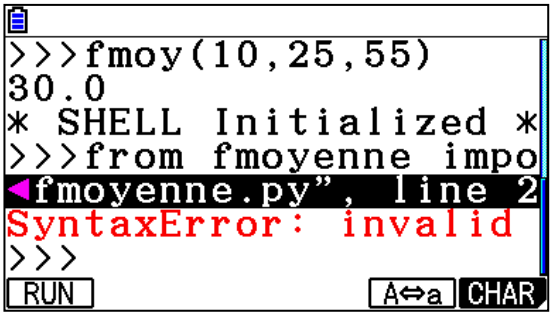


Après ce lancement on se retrouve dans le SHELL.
On peut voir que l'interpréteur Python a téléchargé le nouveau programme avec l'instruction :
from fmoyenne import *

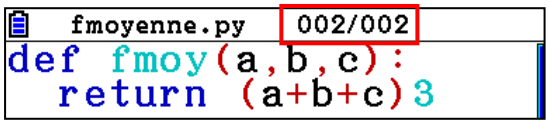


On peut tester la fonction en tapant par exemple en ligne de commande :
>>> fmoy(10, 25 , 55)
On voit que la valeur retournée par la fonction est un flottant **30.0** .
Supposons que nous ayons commis une erreur lors de l'écriture du script, par exemple, si on a oublié le symbole de la division dans le calcul de la moyenne.

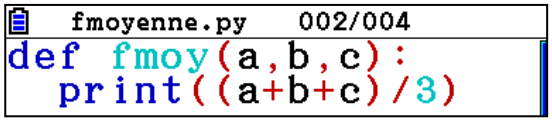
Lors du lancement du programme, le **SHELL** indique qu'il y a **une erreur de syntaxe** et on peut voir que l'erreur est **à la ligne 2** du programme.



En faisant EXIT on se retrouve dans l'éditeur de script et le numéro de ligne s'affiche en haut à droite.

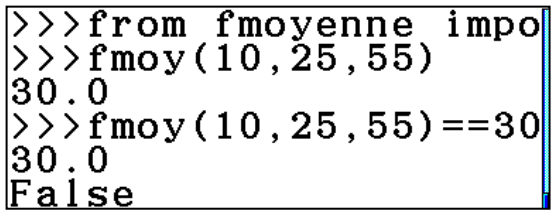


Si par exemple, on utilise l'instruction **print** au lieu de **return.**



Lors de l'utilisation de la fonction, l'affichage sera le même mais si on fait un test du contenu de **fmoy(10,25,55)** avec la valeur **30.0**, on voit que la réponse est « fausse » (**False).**

Le même test avec l'instruction r**eturn** donne une réponse « vraie » (**True**).



En utilisant l'instruction **print** on fait juste un affichage mais on ne donne pas de valeur à **fmoy(a,b,c)** le résultat ne peut donc pas être réutilisé, par exemple **fmoy(1,2,3)+2** donnera une erreur car **fmoy(1,2,3)** sera dans ce cas un **NoneType**

```
>>>from fmoyenne impo
>>>fmoy(10,25,55)
30.0
>>>fmoy(10,25,55)==30
True
```

2. Instructions conditionnelles if / else / elif

Application : Créer une fonction qui indique si une personne est majeure ou mineure en fonction de son âge.

On créé le programme ***fmajeur.***

On peut commencer ici à écrire **def** lettre par lettre (les trois lettres sont côte à côte sur le clavier), il nous faudra ici deux **return** indentés que l'on mettra après avoir inséré l'instruction conditionnelle.

```
fmajeur.py     005/006
def fmaj(a):
  if a>=18:
    return("majeur")
  else:
    return("mineur")
```

On définit la fonction nommée **fmaj** qui s'applique au nombre a qui est l'âge de la personne et on fait un test pour savoir si ce nombre est plus grand ou plus petit que 18.

On utilise SHIFT 4 (**CATALOG**) pour aller chercher **if:else** et **return** et F4 **{CHAR}** pour sélectionner ≥.

On peut aussi sélectionner les instructions conditionnelles avec les touches F6 F1 **{COMMAND}** puis F2 **{if.else}**.
Pour revenir aux onglets principaux, appuyer sur EXIT F6.
On sélectionne ensuite F6 F2 **{OPERAT}** puis F3 **{>}** F1 **{=}** pour comparer a avec 18.

```
* SHELL Initialized *
>>>from fmajeur impor
>>>fmaj(21)
'majeur'
>>>fmaj(12)
'mineur'
>>>
RUN                A⇔a CHAR
```

On lance le programme et on applique la fonction **fmaj** à 21 et 12. On obtient bien 'majeur' pour 21 et 'mineur' pour 12.

Application : Créer une fonction qui indique le prix unitaire d'un article en fonction de la quantité commandée N : si $0 \leq N \leq 50$ le prix sera de 15 euros, si $50 < N \leq 100$ le prix sera de 13 euros et si $N > 100$ le prix sera de 10 euros.

On créé un nouveau programme nommé **prix**.

On définit la fonction **prix** qui donne le prix unitaire de l'article en fonction de la quantité commandée N.

On sélectionne cette fois **if:elif** pour pouvoir appliquer les 3 conditions.

La fonction va retourner **10** si $N > 100$,
13 si $50 < N \leq 100$ et **15** dans les autres cas.

On peut ensuite lancer le programme et appliquer la fonction à plusieurs valeurs : 25, 60 et 150 articles.

On pourrait ajouter autant de « elif » que l'on veut pour tester différentes possibilités.
On n'est pas obligé d'utiliser « else » ou « elif ».

```
prix.py        007/008
def prix(N):
  if N>100:
    return(10)
  elif 50<N<=100:
    return(13)
  else:
    return(15)
```

```
>>>prix(25)
15
>>>prix(60)
13
>>>prix(150)
10
>>>
RUN                A⇔a CHAR
```

3. Boucle for

Application : Créer une fonction qui affiche tous les diviseurs d'un entier naturel N.

On créé un nouveau programme nommé **diviseur.**

On définit la fonction **div** qui va afficher tous les diviseurs de l'entier naturel N.

On utilise SHIFT 4 **(CATALOG)** pour aller chercher **for:range(,)** .

On peut aussi sélectionner la boucle Pour avec les touches F6 F1 **{COMMAND}** puis F4 **{for:range}**.
Pour revenir aux onglets principaux, appuyer sur EXIT F6.

L'instruction ***i in range(a,b,n)*** fait prendre à l'itérateur ***i*** les valeurs entières de ***a*** jusqu'à ***b-1*** avec un pas de ***n***. Si on utilise ***i in range(b)*** alors ***i*** prend les valeurs de 0 à b-1 avec un pas de 1.

On peut tester le programme par exemple avec la valeur 12.

On voit l'affichage des diviseurs de 12.

```
def div(N):
  for i in range(1,N
    if N%i==0:
      print(i)
```

```
for i in range(1,N+1)
```

```
>>>div(12)
1
2
3
4
6
12
RUN                A⇔a CHAR
```

On pourrait aussi envisager de modifier le programme pour que les diviseurs de N soient sauvegardés dans une liste nommée **divis**.

Il faut initialiser la liste avec divis= [ ] pour avoir une liste vide à l'initialisation.

On peut utiliser le raccourci SHIFT + SHIFT − pour saisir les crochets [ ]

On obtient la liste de tous les diviseurs de 12.

```
diviseur.py     006/007
def div(N):
  divis=[]
  for i in range(1,N+
    if N%i==0:
      divis=divis+[i]
  return(divis)
FILE RUN SYMBOL CHAR A⇔a ▷
```

```
6
12
* SHELL Initialized *
>>>from diviseur impo
>>>div(12)
[1, 2, 3, 4, 6, 12]
>>>
RUN A⇔a CHAR
```

4. Boucle while - module random

Application : Réaliser un programme qui génère aléatoirement un nombre entier entre 1 et 100. Le but sera pour l'utilisateur de trouver ce nombre. Pour l'aider, le programme lui donnera les indications « Trop petit » ou « Trop grand » à chaque fois qu'il testera un nombre. Enfin le programme donnera le score, c'est-à-dire le nombre d'essais qui auront été nécessaires à l'utilisateur pour trouver le nombre.
On créé un nouveau programme nommé **nombremy**.

On créé la fonction **mys** qui va générer un nombre aléatoire **n** que l'utilisateur va devoir trouver.
La variable **s** va compter le nombre d'essais.
La variable **a** va enregistrer le nombre choisi par l'utilisateur.

Avec SHIFT 4 CATALOG on sélectionne les différentes instructions et avec F4 **{CHAR}** les symboles:

- La première ligne d'instructions **from random import*** permet d'importer des fonctions en lien avec les calculs de probabilités.
- La variable **s** est initialisée à 1.
- La variable **n** contient un nombre entier aléatoire entre 1 et 100 : **n=randint(1,100)**
- On demande à l'opérateur de choisir un nombre avec l'instruction **a=int(input(« nombre »))**
- On utilise une boucle while pour continuer tant que a est différent de n (**!=** est la syntaxe python pour "différent de")

```
from random import *
def mys():
  s=1
  n=randint(1,100)
  a=int(input("nombre
```

On peut aller chercher **while** avec F6 F1 **{COMMAND}** puis F3 **{while}**.
On peut ensuite sélectionner EXIT F2 **{OPERAT}** puis F2 **{!}** F1 **{=}** ou F3 **{>}** pour faire le test.
Pour revenir aux onglets principaux, appuyer sur EXIT F6.

- On incrémente le score de 1 dans chaque passage dans la boucle **s=s+1**
- On utilise l'instruction **if:else** pour afficher le message **trop grand** ou **trop petit** selon les cas.
- L'opérateur doit saisir un nouveau nombre tant que la valeur n'est pas celle de **n**.
  a=int(input(« nombre »))
- Si la valeur saisit est égale à **n** alors le programme se termine et on retourne le score.
  return(« bravo, score = », s)

```
a=int(input("nombre
while a!=n:
  s=s+1
  if a>n:
    print("trop gra
  else:
    print("trop pet
```

```
  a=int(input("nomb
return("bravo, scor
```

On lance le programme et on teste des valeurs pour trouver le nombre mystère.

On voit, dans la copie d'écran ci-contre, qu'au bout de 3 essais (10, 5 et 7) le nombre mystère a été trouvé.

```
nombre10
trop grand
nombre5
trop petit
nombre7
('bravo, score :', 3)
>>>
RUN                A⇔a CHAR
```

5. Les listes:

Application *:* Réaliser un programme qui génère ***la suite de Syracuse*** à partir d'un entier naturel N non nul saisi par l'utilisateur. Cette suite définie par récurrence consiste à réitérer le processus suivant :

Si le nombre est pair on le divise par 2.
Si le nombre est impair on le multiplie par 3 et on ajoute 1.

La conjecture de Syracuse, non encore démontrée à ce jour, est l'hypothèse mathématique selon laquelle l'algorithme de Syracuse appliqué à n'importe quel entier strictement positif atteint 1 au bout d'un nombre fini d'itération.
Nous allons créer un nouveau programme nommé **suitesyr**. Nous allons utiliser une liste **LIST** pour enregistrer les différentes valeurs prises par la suite jusqu'à atteindre la valeur 1.

On créé la fonction **syra** qui affiche la liste des valeurs de la suite ainsi que le nombre de termes appelé ***temps de vol*** de la valeur **N** saisie en paramètre par l'utilisateur.

On peut utiliser le raccourci SHIFT + SHIFT − pour saisir les crochets **[ ]**

```
suitesyr.py    003/006
def syra(N):
  LIST=[N]
```

Au départ la liste **LIST** contient la valeur **N**.
On continue tant que N est différent de 1:
while N!=1

Pour faire le test de parité on utilise le reste de la division euclidienne de N par 2: **N%2**
Si celui-ci est égal à 0 c'est adire si **N%2==0** alors N est paire et sinon impair.

On peut utiliser le raccourcis SHIFT • pour saisir =
On peut trouver % dans l'onglet **{CHAR}** F4

Dans le cas où le nombre est pair on affecte à la variable N la valeur **int(N/2)** de type entier. Sinon on affecte à N la valeur **3*N+1.**
On ajoute le dernier élément à la liste avec **LIST=LIST+[N]**
On termine par retourner la longueur **len(LIST)**
la liste **LIST: return(len(LIST),LIST)**

```
def syra(N):
  LIST=[N]
  while N!=1:
    if N%2==0:
      N=int(N/2)
    else:
      N=3*N+1
```

```
  LIST=[N]
  while N!=1:
    if N%2==0:
      N=int(N/2)
    else:
      N=3*N+1
    LIST=LIST+[N]
```

```
turn(len(LIST),LIST)
FILE RUN SYMBOL CHAR A⇔a ▷
```

On lance le programme et on teste la fonction **syra** avec plusieurs valeurs. On voit que pour 5 la liste comporte 6 valeurs, pour 15 il y a 18 valeurs et pour 127 la liste est constituée de 47 entiers.
On remarquera que l'on peut obtenir une liste aussi grande que l'on veut avec des puissances de 2.
En effet, l'entier $N = 2^P$ a pour tant de vol $p + 1$.

```
>>>syra(5)
[5, 16, 8, 4, 2, 1])
>>>syra(15)
(18, [15, 46, 23, 70,
>>>syra(127)
(47, [127, 382, 191,
>>>
RUN A⇔a CHAR
```

Pour aller plus loin on pourrait chercher à déterminer la valeur la plus élevée atteinte par la suite, appelée altitude de la suite, en introduisant une variable M initialisée à N.

Il faut ajouter un test **if** sur la valeur de N par rapport à la mémoire **M**.

Si **N>M** alors **M** prend la valeur de **N**.
A la fin de la boucle **while**, M contiendra l'altitude de la suite.

```
def syra(N):
  LIST=[N]
  M=N
  while N!=1:
    if N%2==0:
      N=int(N/2)
```

```
    else:
      N=3*N+1
    LIST=LIST+[N]
    if N>M:
      M=N
return(len(LIST),M,LI
```

On constate pour une valeur initiale de 127 que **l'altitude maximale** est de **4372.**

```
>>>syra(127)
(47, [127, 382, 191,
* SHELL Initialized *
>>>from suitesyr impo
>>>syra(127)
(47, 4372, [127, 382,
>>>
RUN                A⇔a CHAR
```

6. Programmation récursive

Application *:* Réaliser un programme qui calcule n ! la factorielle d'un nombre entier n :
Si $n = 0$ alors $n ! = 0$
Sinon $n ! = n \times (n-1) \times \ldots \times 2 \times 1 = n \times (n-1)!$

On dit qu'un **programme est récursif** s'il s'appelle lui-même dans le programme.
On parle également de **fonction récursive**.

On créé la fonction récursive **fac** qui calcule la factorielle d'un entier naturel n entré en paramètre.

Si n est nul alors la fonction renvoie **1** sinon elle renvoie $\boldsymbol{n \times fac(n-1)}$. Cette dernière instruction de calcul fait apparaître la récursivité.

```
def fac(n):
  if n==0:
    return(1)
  else:
    return(n*fac(n-1))
```

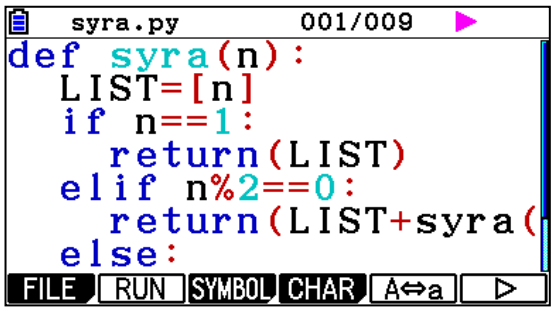
```
def fac(n):
if n==0:
  return(1)
else:
  return(n*fac(n-1))
```

On peut lancer le programme et tester sur plusieurs valeurs de n. on obtient, par exemples :
$3 ! = 3 \times 2 \times 1 = 6$ $5 ! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
$20 ! = 2432902008176640000$

```
>>>fac(3)
6
>>>fac(5)
120
>>>fac(20)
2432902008176640000
```

On pourrait introduire une fonction récursive pour calculer les éléments de la suite de Syracuse abordée dans le paragraphe précédent.

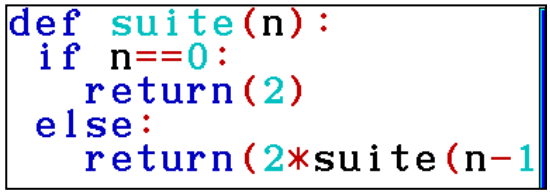
```
def syra(n):
  LIST=[n]
  if n==1:
    return(LIST)
  elif n%2==0:
    return(LIST+syra(int(n/2))
  else:
    return(LIST+syra(3*n+1))
```



Application *:* On considère la suite (u_n) définie par récurrence par $u_0 = 2$ et $u_{n+1} = 2 * u_n + 4$.
Créer un programme qui détermine le terme de rang n de la suite (u_n).

On créé la fonction récursive **suite** qui calcule le terme de rang n. Si $n = 0$ le résultat est 2 et sinon on calcule tous les termes récursivement jusqu'à n.

```
def suite(n):
 if n==0:
   return(2)
 else:
   return(2*suite(n-1)+4)
```

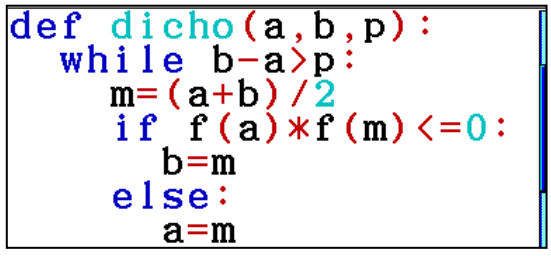


Application *:* Réaliser un programme récursif qui détermine à l'aide *de* ***la méthode de dichotomie,*** un encadrement avec une précision p, de l'unique solution de l'équation $f(x) = 0$ où la fonction f est continue et strictement monotone sur un intervalle $[a; b]$ avec a et b des réels tels que : $f(a) \times f(b) \leq 0$

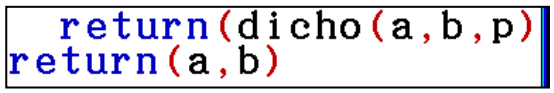
On créer un programme nommé **dichoto** dans lequel est définie la fonction f suivante :
$f(x) = x^3 + 3x - 5$

On utilise également une fonction récursive *dicho* qui prend comme argument a, b et p.

Avec les touches une boucle **while** on fait un test sur la précision obtenue.
On utilise l'instruction **if:else** pour modifier a ou b en fonction du signe de $f(a) \times f(m)$ on fait un calcul récursif avec *dicho.*



Lorsque la précision est atteinte la fonction *dicho* renvoie les valeurs du dernier calcul pour a et b.



```
def dicho(a,b,p):
 while b-a>p:
  m=(a+b)/2
  if f(a)*f(b)≤0:
   b=m
  else:
   a=m
  return(dicho(a,b,p))
return(a,b)
```

On exécute alors le programme pour tester, par exemple : **dicho(1,2,0.001)** alors on obtient que la solution est entre **1.153** et **1.154**

```
>>>from dichoto impor
>>>dicho(1,2,0.001)
(1.1533203125, 1.1542
```

7. Programmation avec une variable globale

Application *:* On veut construire une fonction qui affecte une constante à une variable x.

La fonction f n'a pas d'argument on utilise alors des parenthèses vides ***f()***
On affecte 12 à la variable x : **x = 12**

```
constant.py    006/006
def f():
  x=12

x=0
f()
print(x)
```

Dans le programme principal, on initialise la variable x à 0 et on utilise la fonction **f()**. Ensuite on affiche la valeur contenue dans la variable x.

Le SHELL nous affiche la valeur 0 alors que nous devrions avoir 12.
La fonction f a utilisé la variable x comme une variable locale qui n'a pas de lien avec la variable x du programme principal.

```
>>>from constant impo
0
```

On utilise pour corrigé cela la mention **global x** à l'aide du catalogue.

```
constant.py    002/007
def f():
  global x
  x=12

x=0
f()
print(x)
```

La variable x devient alors globale.
C'est la même variable dans le programme principal et la fonction.
Lorsque l'on exécute le programme on obtient bien la valeur 12.

```
* SHELL Initialized *
>>>from constant impo
12
```

La fonction **f()** a bien modifié le contenu de la variable x.

Par défaut une variable est toujours locale.

8. Programmation avec variables booléennes

Application : On veut créer un script nommé **premier** qui détermine si un nombre N est un nombre premier.

On créé pour cela 3 fonctions.

La première, divis(N,i) renvoie **True** si le nombre i divise N et **False** sinon.
Le test qui permet de savoir si i est un diviseur de N s'écrit en python: N**%**i **==** 0 ce qui correspond à "Le reste de la division euclidienne de N par i est égal à 0"

```
def divis(N,i):
  return(N%i==0)
```

La deuxième fonction $nbdivis(N)$ compte le nombre de diviseurs de N.
On initialise le compteur c à 0 : c = 0
On utilise une boucle **for** qui se répéter N fois :
for i in range(1,N+1)
On utilise une instruction conditionnelle pour savoir si le nombre i divise N.
Si la réponse est ***True*** alors on incrémente le compteur c : **c = c** + 1
A la fin de la boucle la fonction va renvoier le nombre de diviseurs soit la valeur de c.

```
def nbdivis(N):
  c=0
```

```
for i in range(1,N+1)
  if divis(N,i)==True
    c=c+1
return(c)
```

```
def nbdivis(N):
  c=0
  for i in range (1,N+1):
    c=c+1
  return(c)
```

La dernière fonction premier(N) utilise un test sur le nombre de diviseur nbdivis(N)==2 pour retourner **True** si N est premier et **False** sinon.

```
def premier(N)
  return(nbdivis(N)==2)
```

```
premier(N):
eturn(nbdivis(N)==2)
```

Les fonctions doivent être définies avant de les utiliser dans d'autres fonctions. L'ordre de création des fonctions est donc important.

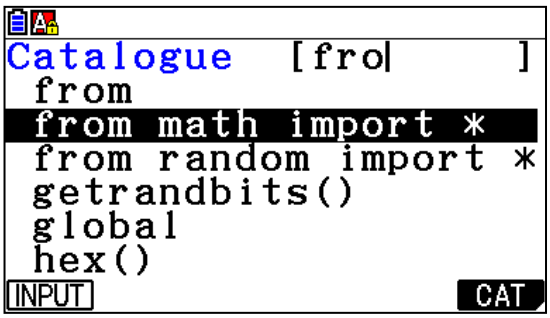
Lorsque l'on exécute le programme par exemples :
premier(1) donne False
premier(17) donne True
premier(533) donne False

```
>>>premier(1)
False
>>>premier(17)
True
>>>premier(533)
False
```

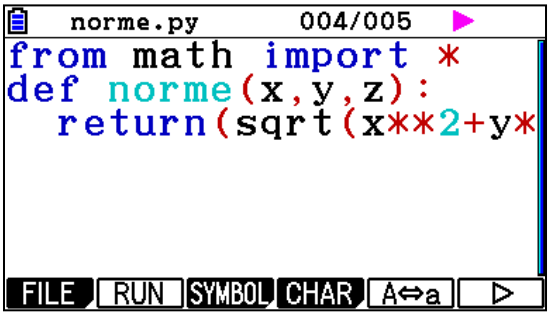
9. Programmation avec la bibliothèque math

Application : On veut créer un programme qui calcul la norme d'un vecteur dans l'espace muni d'un repère orthonormé.

Pour certains calculs utilisant des fonctions classiques comme la racine carrée ou les fonctions trigonométriques, il faut importer le module **math** avec l'instruction **from math import** *

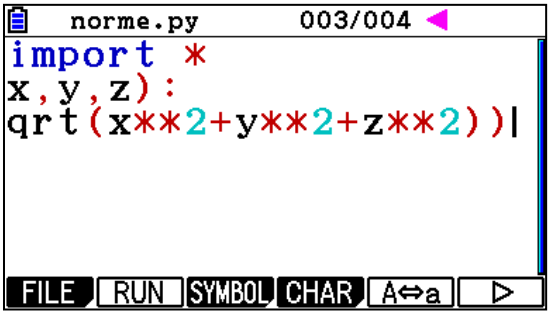


On créé une fonction $norme(x, y, z)$
où les coordonnées du vecteur dans le repère orthonormé sont des paramètres.



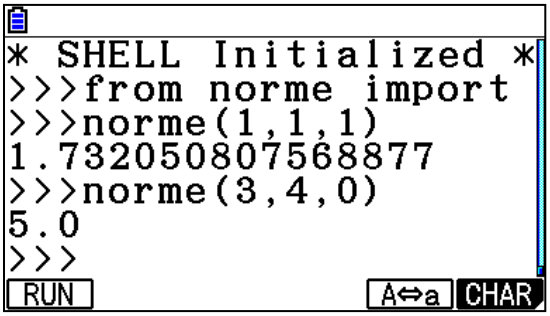
La fonction racine carrée en Python s'écrit **sqrt().**
Il suffit de sélectionner la touche racine carrée de la calculatrice (SHIFT x^2) pour obtenir **sqrt.**

Pour la norme on rentre la formule:
$sqrt(x\text{^}2 + y\text{^}2 + z\text{^}2)$



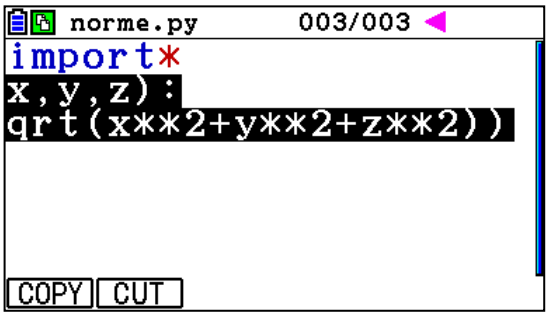
On remarque que le symbole puissance du clavier est remplacé par l'écriture en Python de la puissance ** : $sqrt(x ** 2 + y ** 2 + z ** 2)$

Lorsque l'on exécute le programme par exemples :
norme(1,1,1) donne approximativement 1.732
norme(3,4,0) donne la valeur exacte 5.0



10. Le Copier/Coller dans l'éditeur de programmes

Pour Copier/Coller du texte on utilise [SHIFT] [8] **CLIP** pour sélectionner la zone à copier ensuite avec les flèches ▶. Sélectionner ensuite [F1] **{COPY}** pour copier dans le presse papier puis déplacer le curseur à l'endroit où coller et taper [SHIFT] [9] **PASTE** pour coller le contenu du presse papier. On peut l'utiliser ici pour Copier/Coller la fonction **norme**.

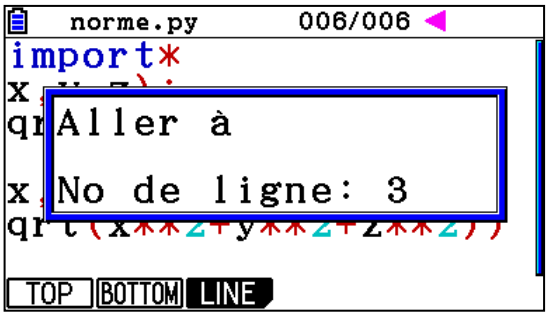


11. Aller directement à une ligne dont le numéro est donné - JUMP:

Si l'on souhaite par exemple aller rapidement à un ligne donnée d'un programme, il suffit de sélectionner [F6] puis [F3] **{JUMP}.**
On a alors 3 possibilités:

- **{TOP}** pour aller à la première ligne
- **{BOTTOM}** pour aller à la dernière ligne
- **{LINE}** pour aller à une ligne de numéro donné

Dans ce dernier cas il suffit ensuite d'u=indiquer le numéro de la ligne souhaitée comme ci-contre.



5. Liste des commandes les plus utiles:

A=3	Affecte la valeur 3 à la variable A
print(A) **print**("CASIO")	Affiche la valeur de A Affiche le texte **CASIO**
A=**input**("A=") → même si une valeur est saisie, A contiendra une chaîne de caractères A=**int**(**input**("A=") → pour un entier A=**float**(**input**("A=")→ pour un flottant	Demande à l'utilisateur de saisir A et affiche le texte **A=**
A==3	Teste si A est égal à 3
A!=3	Teste si A est différent de 3
A>=3	Teste si A est supérieur ou égal à 3
A%3	Renvoie le reste de la division de A par 3 (si A%3=0 alors A est divisible par 3).

def fct(a,b): …. **return**(*résultat*)	Définit une fonction nommée *fct* de deux arguments *a* et *b*.
if *condition*: … **else**: …	Instruction conditionnelle
for i **in range**(0,5,2): …	Pour *i* allant de 0 à 4 inclus avec un pas de 2.
while *condition*: …	Tant que la condition est vraie répéter les instructions.

l[3] c[3]	Le 3ème élément de la liste *l* ou de la chaîne de caractères *c* (les listes et les chaînes de caractères commencent à l'indice 0)
l+[4,2] c+"casio"	Concaténer la liste *l* avec la liste [4,2] Concaténer la chaîne *c* avec la chaîne "casio"
len(l) **len**(c)	Longueur d'une liste ou d'une chaîne de caractère

from random **import***	Importe tout le module random
randint(a,b)	Renvoie un nombre entier aléatoire entre *a* et *b* inclus.