PREMIERE GENERALE

NUMERIQUE ET SCIENCES DU NUMERIQUE

DECOUVERTE DU LANGAGE PYTHON

▷ **Histoire de l'informatique**
▷ Représentation des données : types et valeurs de base
▷ Représentation des données : types construits
▷ Traitement de données en tables
▷ Interactions entre l'homme et la machine sur le Web
▷ Architectures matérielles et systèmes d'exploitation
▷ **Langages et programmation**
▷ Algorithmique

Le langage Python été développé par **Guido Von Rossum en 1989** à l'Université d'Amsterdam. Ce langage a été nommé ainsi en référence à la série télévisée Monthy Python's Flying Circus.

1 – PRESENTATION

Le langage Python est un **langage de programmation objet interprété**. Il offre un **environnement complet de développement** comprenant un interpréteur performant et de nombreux modules. Python est un langage open source. **Libre et gratuit**, il est supporté, développé et utilisé par une large communauté : 300 000 utilisateurs et plus de 500 000 téléchargements par an.

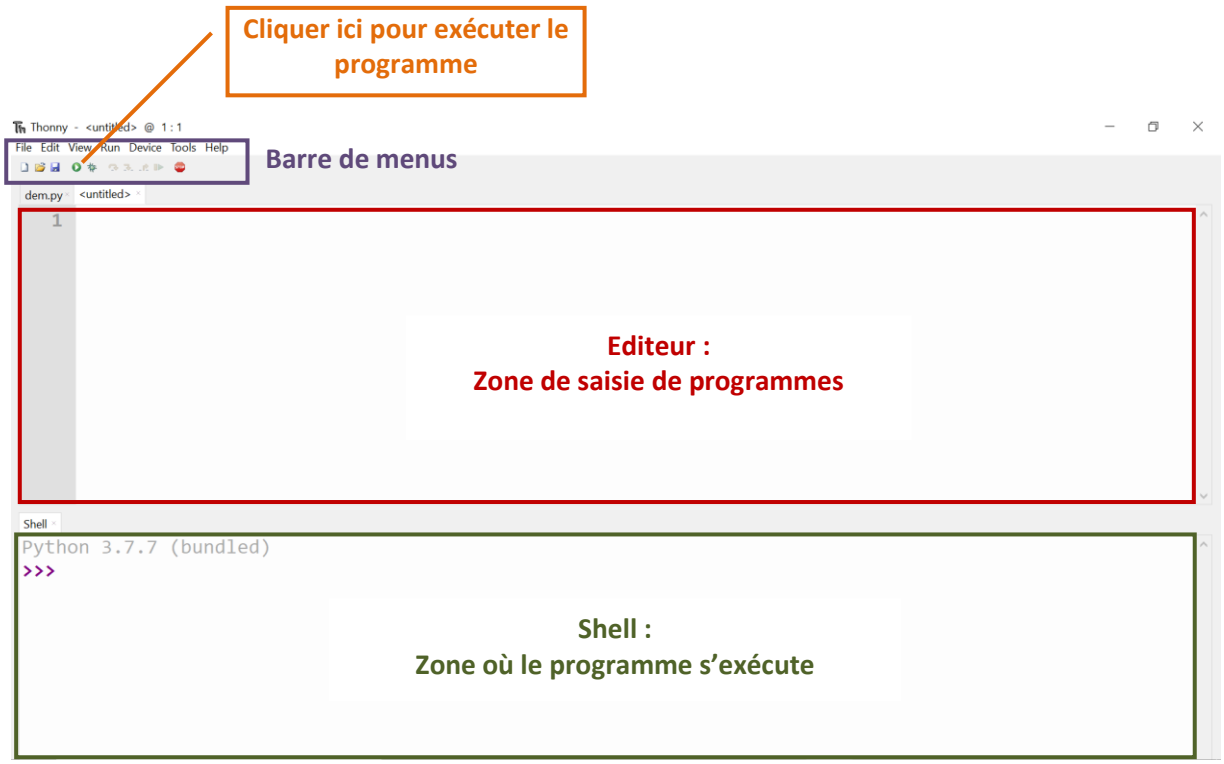
Un atout indéniable est sa **disponibilité sur la grande majorité des plates-formes informatiques** courantes : Mac OS X, Unix, Windows ; Linux, Android, IOS….

Avec le langage Python il est possible de faire :

- du calcul scientifique (librairie **NumPy**) ;
- des graphiques (librairie **matplotlib**) ;
- du traitement du son ;
- du traitement d'image (librairie **PIL**) ;
- des applications avec interface graphique GUI (librairies **Tkinter**, **PyQt**, **wxPython**, **PyGTK** ...)
- des jeux vidéo en temps réel (librairie **Pygame**)
- des applications Web (serveur Web **Zope** ; framework Web **Django**, **Karrigell** ; framework JavaScript **Pyjamas**)
- interfacer des systèmes de gestion de base de données (librairie **MySQLdb** ...) ;
- ...

2 – IDE THONNY

L'interface de développement **Thonny** est constituée d'une barre de menu et deux panneaux : l'**éditeur** et le **shell**.



2.1 – Le Shell

Python est un **langage interprété**. L'interpréteur convertit les instructions Python en langage machine. Le **shell** est la zone dans laquelle l'utilisateur interagit avec l'interpréteur Python. Pour cette raison le shell est souvent nommé interpréteur. Le shell est facilement reconnu. C'est lui qui contient le triple chevron `>>>` qui est l'invite de Python (**prompt** en anglais) et qui signifie que l'interpréteur attend une commande. L'utilisation du shell ressemble à **l'utilisation d'une calculatrice**.

Exercice 1

Taper, dans le shell, la ligne ci-dessous. **Vérifier** le résultat.

```
>>>  (7 + 3) * 10 / 2
```

2.2 – L'éditeur

Un programme Python est appelé **script** L'éditeur permet **d'éditer des scripts**. L'extension des scripts python est **.py**.

Exercice 2

Taper, dans l'éditeur, les lignes ci-dessous. **Enregistrer** le script.

```
1  res = (7 + 3) * 10 / 2
2  print(res)
```

Cliquer sur l'icône Run . **Vérifier** le résultat affiché dans le shell.

2.3 –Quelques recommandations

Un **commentaire** commence par le caractère # et s'étend jusqu'à la fin de la ligne.

```
#-------------------------------------------------
# Ceci est un commentaire
#-------------------------------------------------
res = (7 + 3) * 10 / 2    # En voici un autre
```

Le langage Python compte 33 mots clés :

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

Les **instructions Python** sont éditées en **minuscules**. Les noms de variables **peuvent contenir des majuscules** mais il faudra les écrire toujours de la même façon en respectant les minuscules et majuscules.

3 – TYPES ET VARIABLES

Une variable est un **espace mémoire** dans lequel il est possible de **stocker une valeur** (une donnée). Il s'agit donc d'un **identifiant associé à une valeur**. La notion de variable n'existe pas dans le langage Python. On parle plutôt de **référence d'objet**. Il s'agit donc d'une **référence d'objet située à une adresse mémoire**.

Contrairement à certains langages comme le langage C ou C++ (arduino), sous Python, il n'est pas nécessaire de **définir le type des variables avant de pouvoir les utiliser**. Il d'affecter une valeur à un nom de variable pour que celle-ci soit **automatiquement créée avec le type qui correspond au mieux à la valeur fournie**. On parle alors de **typage dynamique**.

Les types primitifs des variables du langage Python sont : **booléen** (bool), **entier** (int), **entier long** (long), **nombre flottant** (float) et **chaîne de caractères** (str).

Exercice 3

Taper, dans le shell, les lignes ci-dessous. **Décrire** ce qui se passe à l'exécution de chacune des lignes.

```
>>> r, pi = 12, 3.14159
>>> r**2
>>> s = pi * r**2
>>> s
```

Taper, à la suite, les lignes suivantes. **Préciser** l'utilité de la fonction `type()`.

```
>>> type(r)
>>> type(pi)
>>> type(s)
```

Exercice 4

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous. **Exécuter** le script

```
1  a = 18
2  nombre = 34
3  valeur = 2.718
4  sp1 = "Informatique"
5  sp2 = "Sciences du Numérique"
6  y = nombre / a
7  z = nombre / valeur
8  TR = sp1 + " et " + sp2
9  f = sp1 * 2
10 e = sp1 + 2
```

Indiquer quelle ligne du script déclenche une erreur lors de son exécution. **Justifier** l'erreur **Donner** le type et la valeur des variables y, z, TR et f après exécution des instructions ci-dessus.

4 – OPERATEURS ARITHMETIQUES

Les opérateurs arithmétiques sont utilisés pour effectuer des opérations mathématiques comme des additions, soustractions, multiplication, …. entre différentes variables contenant des valeurs numériques.

Exercice 5

Taper, dans le shell, les lignes ci-dessous. **Justifier**, pour chacune des lignes le résultat obtenu et **nommer** l'opération réalisée.

```
>>> 20 + 1
>>> 20 / 3
>>> 20 // 3
>>> 20 % 3
>>> 3.14 * 10
>>> 2 ** 4
>>> (3 + 2) * 5
>>> 3 + 2 * 5
>>> int(7.6)
>>> round(3.1415, 2)
```

5 – CHAINE DE CARACTERES

On appelle caractère tout symbole qui peut être écrit avec les **lettres de l'alphabet** latin, : `abcd...xyz ABCD...XYZ`, les **chiffres décimaux**, `0123456789`, les **symboles de ponctuation** (y compris l'espace), `.,;:!?`, les **symboles de parenthésage**, `(){}[]` et bien d'autres caractères comme les **lettres accentuées**, àéùèÀÈ… et les **lettres d'autres alphabets** ou **caractères spéciaux**, ∓ʒ☿©§.

Une **chaîne de caractère** (**string** en anglais) est une séquence de caractères, c'est-à-dire des caractères qui se suivent les uns derrière les autres. Une chaîne vide est une chaîne qui ne contient aucun caractère. Les chaînes de caractères sont généralement encadrées par des guillemets « " » ou des apostrophes « ' ».

Exercice 6

Taper, dans le shell, les lignes ci-dessous. **Justifier** pourquoi certaines lignes provoquent des erreurs.

```
>>> 'Il apprend à programmer en Python'
>>> "'Il dit : "J'apprends à programmer en Python"'
>>> 'Il dit : "J'apprends à programmer en Python""
>>> "Il dit : J'apprends à programmer en Python"
```

Le caractère « **** » permet d'indiquer à l'interpréteur que le caractère qui le suit doit être conversé tel quel et ne doit pas être interprété.

Exercice 7

Taper, dans le shell, les lignes ci-dessous. **Justifier** pourquoi ces lignes ne provoquent plus d'erreurs.

```
>>> "Il dit : \"J'apprends à programmer en Python\""
>>> 'Il dit : "J\'apprends à programmer en Python\"'
```

Il existe d'autres caractères particuliers qui n'entrainent aucun affichage. On trouve entre autres : **\n** qui permet d'insérer des **sauts à la ligne**, **\t** permet d'insérer des **marques de tabulation** dans une chaîne.

Comme tout type de donnée, une chaîne de caractères peut être **stockée dans une variable**.

Exercice 8

Taper, dans le shell, les lignes ci-dessous.

```
>>> chaine = "J'apprends à programmer en Python"
>>> chaine
```

La **multiplication** de chaînes de caractères est réalisée à l'aide de l'opérateur « ***** ». La **concaténation** de chaînes de caractères est réalisée à l'aide de l'opérateur « **+** ».

Exercice 9

Taper, dans le shell, la ligne ci-dessous.

```
>>> "Coucou" * 3
```

Taper, dans le shell, les lignes ci-dessous.

```
>>> commentaire = "#"
>>> commentaire * 20
```

Taper, dans le shell, les lignes ci-dessous.

```
>>> chaine = "Bonjour, " + "C'est " + "moi."
>>> chaine
```

Pour connaître la taille d'une chaîne de caractères, il est possible d'utiliser, sous Python, la fonction `len()`.

Exercice 10

Taper, dans le shell, les lignes ci-dessous. **Justifier** les résultats obtenus.

```
>>> len('')
>>> len('a')
>>> chaine = "Bienvenue en NSI !"
>>> len(chaine)
```

Dans Python, une chaîne est manipulée comme une **séquence indexée de caractères**. Ainsi, chaque caractère est **accessible directement par son index ou indice**. Le premier caractère (le plus à gauche) d'une chaîne, de longueur `n`, a pour indice `0` et le dernier l'indice `n – 1`.

Exercice 11

Taper, dans le shell, les lignes ci-dessous. **Justifier** les résultats obtenus.

```
>>> chaine = "Bienvenue en NSI !"
>>> chaine[0]
>>> chaine[5]
>>> chaine[17]
>>> chaine[20]
```

En plus de cet accès unitaire aux caractères, il est possible d'accéder à des **sous-chaînes** en précisant la tranche souhaitée par l'indice du premier caractère inclus et l'indice du dernier caractère non inclus.

Exercice 12

Taper, dans le shell, les lignes ci-dessous. **Justifier** les résultats obtenus.

```
>>> chaine = "Bienvenue en NSI !"
>>> chaine[0:8]
>>> chaine[:8]
>>> chaine[13:]
>>> chaine[-1]
```

6 – SAISIR DES DONNEES : INSTRUCTION « INPUT »

L'instruction `input` permet d'obtenir des données de l'utilisateur. Cette instruction **interrompt** le programme, **affiche un message** dans le shell et **attend** que l'utilisateur **saisisse des caractères**. La saisie est validée par l'appui sur la touche ↵ du clavier.

Exercice 13

Taper, dans le shell, la ligne ci-dessous. **Justifier** pourquoi cette ligne provoque une erreur.

```
>>> "Score :" + 35
```

Taper, dans le shell, la ligne ci-dessous. **Préciser** le rôle de l'instruction `str`.

```
>>> "Score :" + str(35)
```

Exercice 14

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous. **Exécuter** le script. **Tester-le**. **Justifier** pourquoi le programme génère une erreur.

```
1 nombre = input("Entrer un nombre entier : ")
2 calcul = nombre + 2
3 print("Le résultat est :", calcul)
```

Modifier le script de la manière suivante. **Enregistrer** puis **exécuter** le script. **Tester-le**.

```
1 valeur = int(input("Entrer un nombre entier : ")
2 calcul = nombre + 2
3 print("Le résultat est :", calcul)
```

Donner le rôle de l'instruction `int`. **Donner** le type des variables `valeur`, `nombre` et `calcul`.

Exercice 15

Editer, un script qui :
- demande les coordonnées x et y de 2 points A et B ;
- affiche les coordonnées x et y du milieu de [AB].

7 – TEST ET STRUCTURES ALTERNATIVES

7.1 – Logique Booléenne

En informatique, tous les mécanismes de tests sont basés sur la **logique booléenne** : Une expression est soit **vraie** (`True`) soit **fausse** (`False`). Les opérateurs de comparaison permettent de comparer deux valeurs et d'indiquer si la condition énoncée est vérifiée ou non (a est plus grand que b, a et b ont la même valeur, etc…). `True` correspond au niveau logique **1** et `False`, au niveau logique `0`. Il existe 6 opérateurs de comparaisons.

`>`	Strictement supérieur à
`>=`	Supérieur ou égal à
`<`	Strictement supérieur à
`<=`	Supérieur ou égal à
`==`	Egal à
`!=`	Différent de

Pour obtenir des tests plus complexes il est possible de combiner plusieurs tests à l'aide des trois **opérateurs booléens** : **and** (ET), **or** (OU), **not** (NON).

Exercice 16

Taper, dans le shell, les lignes ci-dessous. **Vérifier** les résultats des tests.

```
>>>  a = 1
>>>  b = 2
>>>  a == 1 and b < 3
>>>  a == 1 and b == 3
>>>  a == 1 or b > 3
>>>  a < 1 or b > 3
```

7.2 – Structures alternatives

La **structure alternative** `if` permet d'exécuter un bloc d'instruction si une condition est vraie, sinon le bloc d'instruction n'est pas exécuté.

```
if Condition:
    Bloc d'Instructions
Suite du programme
```

Exercice 17

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous. **Exécuter** le script. **Vérifier** le résultat obtenu.

```
1  a = 1
2  if a == 1:
3      print("La valeur est bien 1")
```

Modifier le script de la manière suivante. **Exécuter** le script. **Vérifier** le résultat obtenu.

```
1  a = 3
2  if a == 1:
3      print("La valeur est bien 1")
```

La **structure alternative** `if - else` permet d'exécuter un bloc d'instruction si une condition est vraie, sinon un autre bloc d'instruction est exécuté.

```
if Condition:
    Bloc d'Instructions 1
else:
    Bloc d'Instructions 2
Suite du programme
```

Exercice 18

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous. **Exécuter** le script. **Vérifier** le résultat obtenu.

```
1  a = 1
2  if a == 1:
3      print("La valeur est bien 1")
4  else:
5      print("La valeur est différente de 1")
```

Modifier le script de la manière suivante. **Exécuter** le script. **Vérifier** le résultat obtenu.

```
1  a = 3
2  if a == 1:
3      print("La valeur est bien 1")
4  else:
5      print("La valeur est différente de 1")
```

Dans le cas de **structures alternatives imbriquées**, il est possible d'utiliser une instruction `elif` (contraction de `else if`).

```
if Condition 1:
    Bloc d'Instructions 1
elif Condition 2:
    Bloc d'Instructions 2
else:
    Bloc d'Instructions 3
Suite du programme
```

Exercice 19

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous. **Exécuter** le script. **Vérifier** le résultat obtenu.

```
1  a = 10
2  if a <= 0:
3      print("La valeur est nulle ou négative")
4  elif a == 100:
5      print("La valeur est égale à 100")
6  elif a < 100:
7      print("La valeur est inférieure à 100")
8  else:
9      print("La valeur est supérieure à 100")
```

Donner les valeurs de la variable **a** permettant d'exécuter chacun des blocs d'instructions. **Tester**-les

8 – STRUCTURES ITERATIVES (BOUCLES)

Une **structure itérative** ou **boucle** permet de **répéter une portion de code**.

8.1 – Boucle « while » (tant que)

Tant que la condition est **vraie** (**True**) le bloc d'instructions est exécuté. Le cycle continu jusqu'à ce que la condition soit fausse (**False**) : on passe alors à la suite du programme.

```
while Condition:
    Bloc d'Instructions
Suite du programme
```

Exercice 20

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous. **Exécuter** le script. **Vérifier** le résultat obtenu.

```
1  nombre = int(input("Quelle table faut-il afficher ? "))
2  print("Tableau de multiplication de ", nombre)
3  compteur = 1
4  while compteur < 10:
5      print(nombre, " * ", compteur, " = ", nombre*compteur)
6      compteur = compteur + 1
7  print("C'est fini")
```

8.2 – Boucle « for »

Une boucle **for** permet de **répéter n fois** une instruction ou un bloc d'instructions. La variable **i prend successivement les valeurs 0, 1,…, n–1**.

```
for élément in range(n):
    Bloc d'Instructions
Suite du programme
```

La fonction range qui renvoie une séquence d'entiers successifs peut être utilisée ainsi :

`range(F)`	Renvoie la séquence d'entiers de **0 inclus à F exclu**.
`range(D, F)`	Renvoie la séquence d'entiers de **D inclus à F exclu**.
`range(D, F, P)`	Renvoie la séquence d'entiers de **D inclus à F exclu avec un pas égal à P**.

Exercice 21

Editer, dans l'éditeur, les lignes de code ci-dessous. **Compléter** les paramètres de fonction afin de faire varier la variable compteur de 1 à 9. **Exécuter** le script. **Vérifier** le résultat obtenu.

```
1  nombre = int(input("Quelle table faut-il afficher ? "))
2  print("Tableau de multiplication de ", nombre)
3  for compteur in range (  ,  ):
4      print(nombre, " * ", compteur, " = ", nombre*compteur)
5  print("C'est fini")
```

La boucle **for** permet de parcourir une **séquence itérable** (chaîne de caractère, tableau) **élément par élément**. L'élément peut être de tout type : entier, caractère, élément d'un tableau…

```
for élément in séquence:
Bloc d'Instructions
Suite du programme
```

Exercice 22

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous. **Exécuter** le script. **Justifier** le résultat obtenu.

```
1  chaine = "Python"
2  for lettre in chaine:
3      print(lettre)
4  # on sort de la boucle
5  print("Fin de la boucle")
```

8.3 – Instruction « Break »

L'instruction « **break** » provoque une sortie immédiate d'une boucle « **`while`** » ou d'une boucle « **`for`** ».

Exercice 23

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous. **Exécuter** le script. **Vérifier** le résultat obtenu.

```
1  compteur = 0
2  while True:    # la condition est toujours vraie
3      print("Compteur : ", compteur)
4      quitter = input("Voulez-vous quitter (o/n) ? ")
5      if quitter == 'o':
6          break
7      compteur = compteur + 1
8  # on sort de la boucle
9  print("Fin de la boucle")
```

L'expression « **`True`** » **est toujours vraie** : il s'agit d'une **boucle sans fin**. L'instruction « **`break`** » est donc le seul moyen de sortir de la boucle.

9 – FONCTIONS

Une fonction est une **portion de code** (sorte de sous-programme) que l'on peut appeler au besoin. L'utilisation des fonctions permet : d'**éviter la répétition** ; de **mettre en relief les données et les résultats** (entrées et sorties de la fonction) ;la **réutilisation dans d'autres scripts** par l'intermédiaire du mécanisme de l'import ; de **décomposer une tâche complexe** en tâches plus simples. On obtient ainsi des **programmes plus courts et plus lisibles**.

9.1 – Définition d'une fonction

Pour définir une fonction il faut utiliser la syntaxe suivante :

```
def nom_fonction(parametres1, parametre2, parametre3, …):
    """ Documentation de la fonction.
        ……"""
    bloc_instructions
    return resultat1, resultat2, …
```

Exercice 24

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous. **Exécuter** le script.

```
1  def conv_celsius_kelvin(degres_celsius):
2      """Cette fonction permet de convertir des
3          degrés Celsius en degrés Kelvin"""
4      degres_kelvin = degres_celsius + 273
5      return degres_kelvin
```

Taper, dans le shell, les lignes ci-dessous pour vérifier le fonctionnement.

```
>>> conv_celsius_kelvin(0)
>>> conv_celsius_kelvin(-273)
>>> conv_celsius_kelvin(30)
```

9.2 – Passage de paramètres

Le **passage de paramètres** permet de **fournir les données** utiles à la fonction. Ce passage s'effectue **lors de l'appel** de la fonction. Il est possible de fournir **plusieurs paramètres** à la fonction. Les paramètres passés en arguments peuvent de **types simples** (`int`, `float`, `str`…) mais aussi de **types plus complexes** (tableaux par exemple). Il est également possible de passer en argument **d'autres fonctions**.

Exercice 25

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous. **Exécuter** le script. **Indiquer** de quelle ligne à quelle ligne est définie la fonction et le programme principal. **Préciser** à quelle ligne est appelée la fonction et combien de paramètres lui sont passés.

```
1  def table_mul(table, debut, fin):
2      """----------------------------------------
3      table : table de multiplication attendue
4      debut : à partir de quelle valeur
5      fin : jusqu'à quelle valeur
6      ----------------------------------------"""
8      n = debut
9      while n <= fin :
10         print(n,"*",table,"=",n*table)
11         n = n + 1
12
13 # programme principal
14 num_table = int(input("Quelle table voulez-vous ?"))
15 num_debut = int(input("A partir de quelle valeur ?"))
16 num_fin = int(input("Jusqu'à quelle valeur ?"))
17 table_mul(num_table,num_debut,num_fin)  # appel de la fonction
```

9.3 – Retour des résultats

Le corps d'instruction de la fonction `table_mul()` de l'exercice précédent, **ne contient pas de** `return`, c'est-à-dire qu'elle ne **retourne pas de résultat**. On appelle ce type de fonction, **procédure**. L'instruction `return` **stoppe l'exécution** de la fonction et **retourne une ou plusieurs données**.

Exercice 26

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous.

```
1  def surface_volume_sphere(R):
2      """ Cette fonction calcule et retourne à partir du rayon R,
3      la surface S et le volume V d'une sphère """
4      S = 4.0 * 3.14 * R**2
5      V = S * R / 3
6      return S, V
8
9  # programme principal
10 rayon = float(input("Rayon (en cm): "))
11 s,v = surface_volume_sphere(rayon)
12 print("Sphère de rayon",rayon,"cm")
13 print("Sphère de surface",s,"cm²")
14 print("Sphère de volume",v,"cm3")
```

Indiquer de quelle ligne à quelle ligne est définie la fonction et le programme principal. **Préciser** à quelle ligne est appelée la fonction et combien de paramètres lui sont passés. **Préciser** combien de données sont retournées par la fonction.

9.4 – Portée des variables : variables locales ou globales

La portée d'une variable dépend de **l'endroit du programme où on peut y accéder**. Une **variable globale** est visible et utilisable dans **tout le programme**. Une **variable locale** est créée par une fonction et **n'est visible que par cette fonction**. Lors de la **sortie de la fonction**, la variable est **détruite**.

Exercice 27

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous. **Exécuter** le script. **Justifier** pourquoi les deux valeurs affichées sont différentes.

```
1  x = 10            # variable globale
2
3  def ma_fonction():
4      x = 20        # variable locale
5      print("La variable locale est",x)
6
8  # programme principal
9  print("La variable globale est",x)
10 ma_fonction()
```

10 – MODULES

Exercice 28

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous qui doivent permettre de calculer la racine carrée de n. **Exécuter** le script. **Indiquer** quelle erreur est déclenchée.

```
1  def racine_carree(n):
2      n2 = sqrt(n)          # sqrt : racine carrée
3      return n2
4
5  # programme principal
6  print(racine_carree(2))
```

Lorsque l'IDE est lancée, il ne charge qu'un minimum de fonctionnalité. Pour ajouter d'autres fonctionnalités à Python, il faut **importer des modules**. Python contient de nombreux modules dans sa bibliothèque standard, c'est à dire des modules qui sont disponibles juste en les appelant et sans les télécharger sur le web.

La fonction **sqrt** appartient au module **math** de la librairie standard. Pour pouvoir utiliser cette fonction, il est nécessaire d'importer le module **math** dans le programme. Pour cela il faut la commande **import**. Il y a 3 possibilités d'utilisation de la commande **import**.

– On peut **importer tout le module** :

```
import nom_module
```

Dans notre exemple il faut importer tout le module **math**. Dans ce cas, il faut préciser le nom du module devant le nom de la fonction lors de son appel.

Exercice 29

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous **Exécuter** le script.

```
1 import math
2 def racine_carree(n):
3     n2 = math.sqrt(n)         # sqrt : racine carrée
4     return n2
5 # programme principal
6 print(racine_carree(2))
```

– On peut **importer tous les objets contenus dans le module** :

```
from nom_module import *
```

Exercice 30

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous **Exécuter** le script.

```
1 from math import *
2 def racine_carree(n):
3     n2 = sqrt(n)          # sqrt : racine carrée
4     return n2
5 # programme principal
6 print(racine_carree(2))
```

Avec cette méthode l'appel de la fonction s'écrit **sqrt(n)**. Cela simplifie l'écriture, cependant si une autre fonction **srqt** est présente dans le fichier principal, elle sera automatiquement écrasée par la fonction **srqt** appartenant au module. Le programme reste beaucoup plus lisible dans le premier exemple car l'écriture **math.sqrt(n)** indique bien que la fonction **sqrt(n)** appartient au module **math**.

– On peut **importer seulement la ou les fonctions utiles** :

```
from nom_module import nom_fonction1, nom_fonction2
```

Exercice 31

Ouvrir un nouveau script. **Editer**, dans l'éditeur, les lignes de code ci-dessous **Exécuter** le script.

```
1 from math import sqrt
2 def racine_carree(n):
3     n2 = sqrt(n)          # sqrt : racine carrée
4     return n2
5 # programme principal
6 print(racine_carree(2))
```

Avec cette méthode l'appel de la fonction s'écrit également **`sqrt(n)`**. Cependant **l'accès aux autres fonctions et à certaines variables** (`pi` par exemple) n'est pas possible.

11 – EXERCICES

Exercice 32 : Structures alternatives

Ecrire un script qui demande l'âge d'une personne et affiche « `La personne a X an(s)` », où `X` est la valeur entrée, et en ne mettant un « `s` » à « `ans` » que si nécessaire

Exercice 33 : Structures alternatives

Ecrire un script qui :

- Demande l'âge d'une personne.
- Affiche « `C'est une personne âgée` », si l'âge est supérieur à `60`.
- Affiche « `C'est un enfant` », si l'âge est inférieure à `15`.
- Affiche « `C'est un adulte` », sinon.

Tester votre programme avec le mode debugger de thonny.

Exercice 34 : Structures alternatives

Ecrire un script qui demande un nombre entier `N` à l'utilisateur et affiche si celui-ci est un nombre pair ou impair.

Exercice 35 : Structures itératives (boucle for)

Ecrire un script qui :

- Demande un nombre entier `N` à l'utilisateur.
- Calcul la somme des `N` entiers.
- Affiche cette somme..

Exercice 36 : Structures itératives (boucle for)

Ecrire un script qui permet de reproduire le dessin ci-contre.

```
*
**
***
****
*****
******
*******
********
*********
**********
```

Exercice 37 : Structures itératives (boucle for)

Ecrire un script qui permet d'implémenter l'algorithme ci-contre.

```
Début
    Afficher « Entrer la valeur de N »
    Attendre la saisie de N
    S ← 0
    I ← 1
    Tant que I<=N faire
        S ← S+I
        I ← I+1
    Fin Tant Que
    Afficher « La somme des », N, « premiers entiers est : », S
Fin
```

Exercice 38 : Opérateurs booléens

Ecrire un script qui permet à l'utilisateur de saisir deux nombres et d'afficher « `Le plus petit est ….` ».

Exercice 39 : Opérateurs booléens

Ecrire un script qui permet à l'utilisateur de saisir trois nombres et d'afficher « `Le plus grand est ….` ».

Exercice 40 : Opérateurs arithmétiques

Ecrire un script qui permet d'afficher les entiers de `1` à `100` qui sont des multiples de `7` ; s'ils sont en plus des multiples de `4` alors on affichera une étoile derrière. On doit obtenir l'affichage suivant :

`7 14 21 28 * 35 42 49 56 * 63 70 77 84 * 91 98`

Exercice 41 : Fonctions

Ecrire le script ci-dessous et **exécuter**-le. **Indiquer** ce qui se passe.

```
1  #ma_fonction
2  def ma_fonction(x):
3      y= 3*x + 2
4      return y
```

Taper dans le shell l'instruction suivante. **Justifier** le résultat.

```
>>>  ma_fonction(4)
```

Rajouter à la suite du script précédent les lignes ci-dessous. **Exécuter** le script et **justifier** le résultat.

```
5  #programme principal
6  print(ma_fonction(5))
```

Exercice 42 : Fonctions

Editer une fonction permettant de coder $y = x^2 + 2x + 10$. **Tester-le**.

Exercice 43 : Fonctions

Editer une fonction « **`aire_rec`** » qui prend comme paramètres la longueur et la largeur d'un rectangle et qui retourne l'aire du rectangle.

Exercice 44 : Fonctions

Editer une fonction « **`plus_grand`** » qui prend comme paramètres deux nombres et qui retourne le plus grand des deux.

Exercice 45 : Fonctions

Editer une fonction « **`pyramide`** » qui prend comme paramètre un nombre entier `N` et qui affiche un triangle rectangle dont la hauteur et la base sont égales à `N`.

```
pyramide(4)
*
**
***
****
```