Cours python_220

Types

int #nombre entier
float #nombre décimal
str #chaine de caractères
bool #booléen (vrai ou faux)

renverser une chaine :
chaine[::-1]
Ce n'est pas une méthode. Il faut penser à entrer le résultat dans une variable.

Opérateurs de base

Les opérateurs arithmétiques sont : +, -, *, /
Puissance : **
Division entière : //
Reste de la division entière : %

== *#Pour tester une égalité il faut utiliser un double égal*

!= *#Pour tester une différence*
<, >, <=, >=

Les opérateurs logiques :
and **&**
or |
not ~
xor ^

Opérateur sur les chaines de caractères
'bon'+'jour' *#vaut 'bonjour'*
'Ha'*3 *#vaut 'HaHaHa'*
'bon'<'jour' *#vaut True*
'na'=='Ha' *#vaut False*
'ou' in 'jour' *#vaut True*
'on' not in 'bon' *#vaut False*

Variables, affectation et expression

Les noms de **variables** ne peuvent contenir que des signes alphanumériques (a-z, A-Z, et 0-9) et des underscore (blancs soulignés). Le nom ne peut pas commencer par un chiffre.

L'affectation attribue une valeur à une variable à l'aide du symbole =
Syntaxe : nom_variable=expression

Types structurés

Une fonction qui renvoie plusieurs valeurs les renvoie dans un tuple.

Les listes :
Initialisation : l=[val1, val2,…]
Accès : l[rang]
Ajout : l.append(valeur)
Suppression : del l[rang]
Modification : l[rang]=valeur
Taille : len(l)

Compréhension de liste :
Liste=[expression for i in conteneur]

Les dictionnaires :
Initialisation : d={clé1 :val1, clé2 :val2,…}
Accès : d[clé]
Ajout : d[cléN]=valN (ajoute à la fin)
Suppression : del d[clé]
Modification : d[clé]=val
Taille : len(d)

Compréhension de dictionnaire :
dico={k :v for k,v in liste de tuple}

Instructions

input('texte permettant de savoir ce qui doit être entré') *#le résultat est une chaine de caractères*

Fonctions

Une **fonction** permet de donner un nom à un bloc de code.

#affecter le résultat à une variable après avoir éventuellement modifié le type.

try et except *# Pour gérer les erreurs de saisie*

exemple :

```
ok=False
while not ok:
    try:
        a= int(input("Entrer un entier :  "))
        ok=True
    except ValueError:
        print("Erreur de saisie ! Entrer un entier")
```

print('texte',variable) ***#Le texte est entre guillemets, si il y a plusieurs éléments à afficher, les séparer par une virgule.***

Un **conteneur** est une variable d'un type structuré ou un intervalle défini par la fonction **range**
Syntaxe :
range(4) **#** *conteneur d'entiers de 0 à 3*
range(d, n, p) *#conteneur d'entiers de d inclus à n exclu par pas de p.*
range(0,4,1) est identique à **range(4)**
range(d,n) est identique à **range(d,n,1)**

Une **boucle bornée (for)** exécute un bloc un nombre prédéfini de fois en changeant la valeur d'une variable.
Syntaxe :

```
for var in conteneur :  #attention aux :
    Bloc                #attention à l'indentation
```

```
if condition :          #attention aux :
  bloc                  #attention à l'indentation
elif condition :        #C'est le sinon quand il y
  bloc                    en a plusieurs
:
:
else :                  #C'est le dernier sinon
  bloc                  Le else n'est pas obligatoire
```

Une **boucle non bornée (while)** exécute un bloc tant que la condition est vérifiée.
Syntaxe :

```
while condition :    #attention aux :
  bloc               #attention à l'indentation
```

Syntaxe :

```
def nom_fonction(para1,para2,…): #att aux :
  corps                          #att indentation
```

Pour des raisons de lisibilité, l'on peut indiquer le type des paramètres.
Syntaxe :

```
def nom_fonction(para1 :int, para2 :float,…) :
```

L'appel d'une fonction à la forme suivante :
Syntaxe :

```
nom_fonction(exp1,exp2,…)
```

para1, … sont les paramètres : des noms de variables qui peuvent être utiliser dans le bloc corps, et qui sont initialisés par les expressions exp1, … passées en arguments lorsque la fonction est appelée.

La fonction **return** permet à une fonction de renvoyer un ou plusieurs résultats.

Lorsque la fonction return est utilisée, l'on peut préciser le type de retour.
Syntaxe exemple :

```
def nom_fonction(para1 :int, …)->bool :
```

Lorsqu'avec return, l'on renvoie plusieurs valeurs :

```
return val1,val2,val3
```

en réalité, c'est un **tuple** qui est renvoyé.
On accède aux valeurs de la façon suivante :

```
v1,v2,v3=fonction
```

L'instruction **assert** permet de tester une fonction.
Syntaxe :

```
assert fonction()==valeur
assert not fonction()   #Résultat attendu False
assert fonction()       #Résultat attendu True
```

Les méthodes

Les méthodes sont des instructions que l'on appelle de la façon suivante :
var.méthode
La méthode est appliquée sur la variable var.

append() : permet d'ajouter à une liste ce qui est défini dans les parenthèses.
items() : permet d'obtenir la liste des tuples (clé, valeur) d'un dictionnaire.
keys() : permet d'obtenir la liste des clés d'un dictionnaire (l'ordre peut être modifié)

values() : permet d'obtenir la liste des valeurs d'un dictionnaire (l'ordre est le même que pour keys())
reverse() : permet de renverser une liste

Bibliothèques

Une bibliothèque contient des fonctions qui ne sont pas directement accessibles. Il faut importer celle-ci en début de programme pour y avoir accès.
Syntaxe :

from turtle import* *#ici, nous importons toutes les fonctions contenues dans la bibliothèque turtle.*

Pour avoir accès à la liste de ces fonctions, il suffit de taper dans le shell :

import turtle
help(turtle)

Pour des raisons de capacités mémoires, l'on peut souhaiter de n'importer qu'une fonction contenue dans une bibliothèque. Il suffit de la spécifier. L'on peut également la renommer pour plus de lisibilité.
Syntaxe :

from bibliothèque import fonction as nouveau_nom

Exemple :

from math import sqrt as racine #si l'on souhaite importer plusieurs fonction, les séparer par une virgule.

math

```
sqrt()                  #racine carrée
round()                 #arrondi
floor()                 #arrondi inférieur
ceil()                  #arrondi supérieur
pi
cos(), sin(), tan()
degrees(), radians()
```

random

```
randint(a,b)    #entier au hasard  entre a et b au sens large
random()        #un réel au hasard entre 0 compris et 1 non compris
```

turtle

```
forward() ou fd()       #avance de ... pixels
backward() ou back()    #recule de ... pixels
left()                  #tourne à gauche de ... °
right()                 #tourne à droite de ... °
clear()                 #efface
penup()                 #lève le crayon
pendown()               #descend le crayon
setx()                  #déplace le crayon horizontalement de ... pixels
sety()                  #déplace le crayon verticalement de ... pixels
```

nsi_ui

matplotlib.pyplot

fractions

```
Fraction(entier_a,entier_b)   #est égale à la fraction simplifiée de a/b

méthodes :
    numerator denominator
    exemple :
      fr.numerator            #prend le numérateur de la fraction fr
```