Python

Base

CONSULTING BSTORM


CONSULTING
BSTORM

Table des matières

- Variable
- Interaction avec la console
- Opérateur
  - Arithmétique
  - Comparaison
  - Logique
- Structure conditionnel
  - If
- Structure itérative
  - while
  - for
- Collection
  - Tuple
  - List
  - Dictionaries
  - Set
- Les méthodes
- Gestion des erreurs

Les variables

Qu'est ce que c'est ?

Une variable permet de stocker une information en mémoire, celle-ci se compose :

- Un nom
- Un type
- Une valeur
- Une référence mémoire

Le concept de variable constante n'existe pas en python.

Le nom d'une variable

En python, la convention de nommage pour les variables est le « snake_case »

- Peut être composé de lettre, chiffre et underscore
- Commence toujours par une lettre
- En minuscule
- Les caractères spéciaux sont interdits (lettres accentuées, $, #, …)

Les mots réservés

Attention, il y a des mots-clefs que vous ne pouvez pas utiliser!

Voici la liste des 33 mots réservés en python :

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

Le type d'une variable

Le python est un langage à typage dynamique.

- Le type adapté est automatiquement utilisé lors de l'affectation de la variable.
- Il est conseillé de ne pas changer le type de variable en cours de programme !

La fonction « type(...) » permet de récupérer le type de la variable

Les types de base

- Numérique
  - int : Valeur entière
  - float : Valeur approximative d'un nombre réel

- Texte
  - str : Chaîne de caractères
  - unicode : Chaîne de caractères encodés en Unicode

Une valeur de type texte doit être en simple ou double quote.
Pour du texte multiligne sans caractère d'échappement, il faut tripler les quotes.

Les types de base

- Booléen
  - bool : Valeur « True » ou « False » *(Attention, en python, il faut une Majuscule !)*

- Autres types
  - vide : Valeur « None » *(Équivalent au « null » de C#, Java, ...)*
  - type : Type basique, renvoyé à l'aide de la méthode type(...)
  - object : Objet basique
  - function : Fonction

Liste complète des différents types :
https://fr.wikiversity.org/wiki/Python/Les_types_de_base

Affectation

Contrairement au langage classique... il n'est pas nécessaire de déclarer les variables.

L'affectation des variables se réalise avec l'opérateur « = ».

```
entier = 42
reel = 3.14
texte = "Hello World"
booleen = True
```

Python autorise également l'affectation multiple

```
nb1, nb2 = 13, 42
```

Conversion

Il existe des méthodes permettant de convertir une variable vers un type donné

- int(...) : Conversion vers un entier de type “int”
- float(...) : Conversion vers un nombre à virgule
- str(...) : Transformation vers du texte
- unicode(...) : Transformation vers du texte encodé en Unicode

Interaction avec la console

Ecrire dans la console

La fonction a utiliser pour afficher du texte dans la console est « print(...) »

Lorsqu'il y a plusieurs valeurs, un séparateur (espace par défaut) est ajouté.
En fin de ligne, un saut de ligne est ajouté.

```
print("Hello World")

nb = 42
print("Le nombre est", nb)

# "sep" modifie le séprateur
# "end" modifie la fin de ligne
print("Le nombre est", nb, sep="_", end=" > ")
```

Lire dans la console

Pour interagir avec l'utilisateur, il est possible de lire la saisie de la console.

Pour cela, il faut utiliser la méthode « input("message") », qui va permettre de récupérer la valeur entrée par l'utilisateur en format texte.

```
valeur = input("Veuillez entrer une valeur...")
```

Attention, la valeur récupérée sera toujours de type texte,
une conversion est nécessaire pour la transformer en entier, réel,...

```
nombre = int(input("Entrez un nombre !"))
```

Les opérateurs

Arithmétique

Les opérateurs arithmétiques sont :

+	Addition
-	Soustraction
*	Multiplication
/	Division réel
//	Division entière
%	Modulo (Reste de la division entière)

Comparaison

Les opérateurs de comparaison sont :

==	Egalité strict	**!=**	Différence
<	Strictement supérieur	**<=**	Supérieur ou égal
>	Strictement inférieur	**>=**	Inférieur ou égal

in Test d'adhésion : *Si “a” est contenu dans “b”.*

is Test d’identité : *Si “a” et “b” contiennent sur le même contenu.*

Logique

Les opérateurs logiques sont :

and Vrai, si les 2 valeurs sont vraies.

or Vrai, si au moins une des 2 valeurs est vraie.

not L'inverse de la valeur.

ET

A ∧ B	Vrai	Faux
Vrai	V	F
Faux	F	F

OU

A v B	Vrai	Faux
Vrai	V	V
Faux	V	F

NON

	¬A
Vrai	F
Faux	V

Affectation et opération

Il existe une écriture raccourcie qui permet de réaliser une opération arithmétique combinée avec une affection

a += b → a = a + b

a -= b → a = a - b

a *= b → a = a * b

...

La structure conditionnelle

Le « if, elif, else »

La structure « if » permet d'exécuter des opérations en fonction de condition.

- if Condition à tester *(obligatoire)*
- elif Condition si la précédente est fausse *(optionnel et répétable)*
- else Traitement si aucune condition validée *(optionnel)*

```
if condition_if:
    print("Condition \"if\" valide")
elif condition_elif:
    print("Condition \"elif\" valide")
else:
    print("Si aucune condition remplie")
```

Les structures itératives

La structure « while »

La boucle « while » permet de répéter un ensemble d'opérations tant que la condition vérifiée est vraie.

```
compteur = 0
while compteur <= 10:
    print(compteur)
    compteur += 1
```

Attention, vous devez toujours maîtriser la condition de la boucle!

Les différentes structures « for »

Les structures « for » sont les boucles permettant de parcourir les collections.

En python, il en existe 3 cas différents :

- Utiliser directement avec une collection
- A l'aide de la méthode « enumerate(...) »
- A l'aide de la méthode « range(...) »

Les différentes collections du python seront abordées par la suite

Les boucles « for »

- Avec une collection

```
collection = ["A", "B", "C"]

for element in collection:
    print(element)
```

- Méthode « enumerate(*collection*) »

```
collection = ["X", "Y", "Z"]

for i, val in enumerate(collection):
    print("Index :", i)
    print("Valeur :", val)
```

Les boucles « for »

- Méthode « range(start, stop, step) »

Elle est utilisable de 3 manières différentes :

➔ Avec tous les paramètres

➔ Avec le “start” et le “stop”
Le “step” par défaut est 1

➔ Avec uniquement le “stop”
Le “start” par défaut est 0

```
collection = ["A", "B", "C", "D", "E"]
limite = len(collection)


for i in range(0, limite, 1):
    print(i, " - ", collection[i])


for i in range(3, limite):
    print(i, " - ", collection[i])


for i in range(3):
    print(i, " - ", collection[i])
```

Les collections

Les différentes collections du python

Les collections permettent de regrouper dans une seule variable un ensemble d'informations.

En python, il en existe plusieurs :

Nom	Indexé	Ordonnée	Modifiable	Membre dupliqué
List	Oui	Oui	Oui	Oui
Tuple	Oui	Oui	Non	Oui
Set	Non	Non	Oui	Non
Dictionaries	Oui	Non	Oui	Non

La collection « list »

En python, une liste s'écrit entre crochet “[ ]”

```
# Creation de la liste
my_list = ["Riri", "Fifi", "Loulou"]

# Utilisation du constructeur pour créer la liste
my_list_2 = list(("Riri", "Fifi", "Loulou"))      # /!\ Double parenthèse

# Creation d'une liste vide
my_list_vide_1 = []
my_list_vide_2 = list()

# Récuperation d'un élément
elem = my_list[1]
```

La collection « list »

```
# Modification d'une valeur
my_list[2] = "Zaza"

# Ajouter d'un élément
my_list.append("Donald")
my_list.insert(1, "Daisy")

# Supprimer un élémént
my_list.remove("Fifi")
my_list.pop(2)   # Si l'index n'est pas défini, le dernier est supprimé

# Supprimer tous les éléments
my_list.clear()
```

Récupérer un range

Le langage python permet de récupérer un ensemble d'éléments au sein d'une liste.

```
tab = [4,7,9,3,8,2,6,5]
#      0 1 2 3 4 5 6 7

# récupérer de l'indice 2 à l'indice 5 (non compris)
tab[2:5] # => [9,3,8]
# récupérer jusqu'à l'indice 3
tab[:3] # => [4,7,9]
# récupérer à partir de l'indice 4
tab[4:] # => [8,2,6,5]
# moyen rapide pour créer un clone
tab[:] # => [4,7,9,3,8,2,6,5]
# le troisième argument représente le pas
tab[2:6:2] # => [9,8]
```

La collection « tuple »

En python, un tuple s'écrit entre parenthèse “( )”

```
# Creation du tuple
my_tuple = ("Riri", "Fifi", "Loulou")

# Utilisation du constructeur pour créer le tuple
my_tuple_2 = tuple(("Riri", "Fifi", "Loulou"))     # /!\ Double parenthèse

# Récuperation d'un élément
elem = my_tuple[1]

# Modification impossible

# Ajout ou suppression non autorise!
```

La collection « set »

En python, un set s'écrit entre accolade “{ }”

```
# Creation du set
my_set = {"Riri", "Fifi", "Loulou"}

# Utilisation du constructeur pour créer le set (déconseillé)
my_set_2 = set(("Riri", "Fifi", "Loulou"))     # /!\ Double parenthèse

# Creation d'un set vide
my_set_vide_1 = set()    # Attention, l'ecriture {} ne créer pas un "set" !

# Récuperation ou modification d'un élément impossible
```

La collection « set »

```
# Ajouter d'un élément
my_set.add("Balthazar")

# Ajouter de plusieurs éléments
my_set.update(["Donald", "Daisy"])

# Supprimer un élément
my_set.remove("Fifi")
my_set.discard("Riri")   # Ne provoque pas d'erreur si l'element est manquant
my_set.pop()             # Supprime le dernier élément

# Supprimer tous les éléments
my_set.clear()
```

La collection « Dictionaries »

En python, un dictionnaire s'écrit entre accolade avec des combinaisons “clef/valeur”.

```
# Creation du dictionnaire
my_dictionaries = {
  "firstname": "John",
  "lastname": "Smith",
  "year_result": 15
}

# Utilisation du constructeur pour créer le dictionnaire
my_dictionaries_2 = dict(firstname="John", lastname="Smith", year_result=15)

# Creation d'un dictionnaire vide
my_dictionaries_vide_1 = {}
my_dictionaries_vide_2 = dict()
```

La collection « Dictionaries »

```
# Récuperation d'une valeur sur base de la clef
f_name = my_dictionaries["firstname"]
l_name = my_dictionaries.get("lastname")

# Modification d'une valeur
my_dictionaries["year_result"] = 15

# Ajouter d'une clef / valeur
my_dictionaries["matricule"] = "JOSM042"

# Supprimer un élémént
my_dictionaries.pop("year_result")
my_dictionaries.popitem()    # Supprime le dernier élément ajouté

# Supprimer tous les éléments
my_dictionaries.clear()
```

Les méthodes

Définition d'une méthode

L'utilisation des méthodes permet d'éviter la répétition de codes.

Pour déclarer une méthode, il faut utiliser le mot clef « def ».
Il en existe 2 types :

- Procédure → Ne retourne aucune valeur
- Fonction → Retourne une valeur

La convention de nommage pour les méthodes est également le « snake_case »

Exemple d'une procédure

```
def afficher_tableau(tableau):
    for elem in tableau:
        print(elem, end=" > ")
    print()


tab = ["Riri", "Fifi", "Loulou"]
afficher_tableau(tab)
```

Exemple d'une fonction

```
def calculer_moyenne(tableau):
    nb_elem = len(tableau)
    total = 0
    for elem in tableau:
        total += elem

    return total / nb_elem


results = [12, 20, 15, 7]
moy = calculer_moyenne(results)
print(moy)
```

Les paramètres

Les valeurs reçues en paramètres sont copiées dans des variables locales à la méthode.

```
def demo_modif(val):
    val += 1
    print(val)   # Affichage de la valeur local


a = 42
print(a)          # Affiche 42
demo_modif(a)     # Affiche 43
print(a)          # Affiche 42
```

Variable locale VS globale

Les variables créées au sein d'une méthode n'existe que dans celle-ci.

En python, il est possible d'utiliser le mot clef « global » pour créer ou utiliser une variable en dehors de la zone mémoire de la méthode.

⚠️ Attention, ceci n'est pas conseillé ! ⚠️

```
def demo_variable():
    var_local = 42

    global var_global
    var_global = 1337


var_global = 10
print(var_global)    # Affiche 10

demo_variable()
print(var_global)    # Affiche 1337
print(var_local)     # Erreur !
```

Gestion des erreurs

Try... Except

La gestion des erreurs permet d'intercepter les exceptions pour les traiter.

En python, il y a quatre types de blocs prévu pour cela : **try**, **except**, **else** et **finally**.

```
try:
    print("Code à risque")
except:
    print("Une exception a été levée")
else:
    print("Aucune exception rencontrée")
finally:
    print("Code toujours executé")
```

Multiples « Except »

Le bloc « Except » est spécialisable pour gérer uniquement un type d'erreur.

Il est possible de gérer plusieurs types d'erreurs, à l'aide de multiples **except**.

Dans ce cas, l'exception va passer de **except** en **except**, à la recherche du premier bloc compatible avec l'exception.

```
print("Calcul d'une division")
try:
    nb = int(input("Nombre : "))
    div = int(input("Diviseur : "))

    res = nb / div
    print("Le resultat est", res)
except ValueError:
    print("Valeur incorrecte!")
except ZeroDivisionError:
    print("Division par Zero!")
```

Merci pour votre attention.