[Python] Kit de base

Introduction

Python est un langage interprété très permissif et souple.

Il supporte une multitude de modules rendant le langage ultra puissant pour divers types d'utilisations.

Variables

- Entier

```
#Affectation d'un entier
myVar=3
```

- Flottant

```
#Affectation d'un flottant
myVar=2.8
```

- Chaîne de caractères

```
#Affectation d'une chaîne de caractères
myVar="Hello world"
```

- Liste

```
#Affectation d'une liste
myVar=[1, 2, 3]
```

- Dictionnaire

```
#Affectation d'un dictionnaire
myVar={"clé1": "valeur1", "clé2": "valeur2"}
```

- Tuple (liste de constantes) :

```
#Affectation d'un tuple
myVar = (1, 2, 3, 4, 5)
```

- Set (liste sans doublon) :

```
#Affectation d'un set
myVar = {1, 2, 3, 4, 5}
```

- Booléen

```
#Affectation d'un booléen
myVar=True
```

Print

Pour afficher une variable dans la console :

```
print(myVar)
```

Ou la version formatée :

```
print(f"Hello {name} !")
```

Input

Il peut être intéressant de récupérer une saisie de la part de l'utilisateur.

Dans ce cas, la fonction input le permet :

```
entree_utilisateur = input("Entrez votre nom : ")
```

Remarque : La fonction renvoie une chaîne de caractère.

Il peut donc être intéressant de transtyper la valeur de retour.

Transtypage (casting)

L'intérêt du **transtypage** est de passer d'un type de variable à un autre (dans la mesure du possible).

Cela peut s'avérer utile pour transformer une chaîne de caractères saisie par l'utilisateur et la transformer en type entier afin de pouvoir effectuer des calculs par exemple.

On peut aussi s'en servir pour convertir un **float** en **int** et ne récupérer ainsi que la partie entière d'un nombre.

En reprenant l'exemple de l'entrée utilisateur ci-dessus, voici comment transformer l'input **string** vers **int** :

```
nombre_entier = int(entree_utilisateur)
```

La variable **nombre_entier** pourra ensuite être traitée dans un algorithme de calcul.

Commentaires

En Python, il existe 2 types de commentaires :

- Commentaire sur une seule ligne :

  ```
  # Voici un commentaire sur une seule ligne
  ```

- Commentaire sur plusieurs lignes :

```
"""
Voici un commentaire
sur plusieurs
lignes
"""
```

Opérations mathématiques

```
a = 10
b = 5

addition = a + b          # Addition
soustraction = a - b      # Soustraction
multiplication = a * b    # Multiplication
division = a / b          # Division
modulo = a % b            # Modulo (reste de la division entière)
exposant = a ** b         # Exposant
```

Conditions

Il est possible de réaliser certaines instructions seulement sous certaines conditions.

Pour cela on utilise les structures conditionnelles.

Ces structures conditionnelles vérifient des conditions grâce aux opérateurs conditionnels.

Opérateurs conditionnels

Opérateur conditionnels	Description
==	Égalité
!=	Différence
>	Supérieur strict
<	Inférieur strict
>=	Supérieur ou égal
<=	Inférieur ou égal

If

La structure conditionnelle la plus simple est le **if** . Voici sa syntaxe :

```
if a > b:
    print("a est plus grand que b")
```

Remarque : Attention aux tabulations qu'il faut placer dans le bloc d'instructions où la condition est vérifiée.

If / Else

Le **if** / **else** reprend la même syntaxe en ajoutant le cas où la condition n'est pas respectée :

```
if password == "p@ssW0rD":
    print("Mot de passe correct")
else:
    print("Mot de passe incorrect")
```

If / Elif / Else

Le **if** / **elif** / **else** reprend la même syntaxe en ajoutant une nouvelle condition :

```
if a > b:
    print("a est plus grand que b")
elif a < b:
    print("a est plus petit que b")
else:
    print("a est égal à b")
```

Match / case :

Le **match** / **case** est semblable au switch case que l'on peut retrouver en langage C avec un syntaxe adaptée à Python.

Cette structure conditionnelle permet de tester uniquement des égalités :

```
match a:
  case 'coucou':
    print("Hello")
  case : 'a plus':
    print("Au revoir")
```

Dans l'exemple ci-dessus, on teste d'abord le cas où si la variable a=='coucou' (si oui, on affiche "Hello").

Dans un deuxième temps, on teste le cas où si la variable a=='a plus' (si oui, on affiche "Au revoir").

Boucles

While

La boucle **while** ou *tant que*, permet d'exécuter un bloc d'instructions tant qu'une condition est vérifiée :

```
while a > 0:
    print(a)
    a = a - 1
```

For

La boucle for est plus puissante et peut être utilisée de différentes manières.

- Range : de 0 à X (ex: 5)

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

- Liste : Affiche les éléments d'une liste

  ```
  for item in list:
    print(item)
  ```

- Liste avec incrément : Affiche les éléments d'une liste en conservant l'incrément

  ```
  for i, item in enumerate(list):
    print(f"ID n°{i} --> {item}"
  ```

Modules

Avec Python, il est possible d'utiliser des modules qui sont des bibliothèques ajoutant des fonctionnalités supplémentaires.

Pour installer un module, il suffit d'utiliser **pip** avec la commande suivante :

```
python -m pip install <MODULE>
```

Ainsi, le module sera présent sur votre système mais pas utilisable dans votre code.

Pour cela, il faut importer le module à la première ligne de votre code grâce à cette instruction :

```
import <MODULE>
```

La syntaxe suivante permet d'importer seulement une ou plusieurs classes spécifiques d'un module :

```
from <MODULE> import <CLASS>, <FUNCTION>
```

Liste des modules les plus courants

Modules	Description
time	Gestion du temps et des dates.
os	Diverses interfaces pour le système d'exploitation.
sys	Paramètres et fonctions propres à des systèmes.
requests	Utilisation des requêtes
matplotlib	Création de graphique.
numpy	Opérations mathématiques.
random	Fonctions liées à l'aléatoire.
tkinter	GUI pour Python.

Fonctions

Les fonctions permettent de créer un bloc d'instructions réutilisable à souhait.

Elle réalise généralement accomplir une tâche spécifique.

Les fonctions peuvent prendre ce qu'on appelle des paramètres, qui sont des variables fournis à la fonction, qu'elle pourra utiliser :

```
def addition(a, b):
    resultat = a + b
    return resultat

"""
Ici, la fonction addition() a comme paramètres 'a' et 'b'.
Elle stocke dans une variable locale le résultat de l'addition des deux valeurs contenues dans 'a' et 'b'.
Puis elle renvoie le résultat pour qu'on puisse l'utiliser en dehors de la fonction.
"""
```

On peut donc appeler cette fonction et afficher le résultat de l'opération :

```
a=4
b=5

resultat=addition(a, b)
print(resultat)
```

Main

```
def main():
    print("Hello world!")

if __name__ == "__main__":
    main()
```

Fichiers

En Python, on peut nativement ouvrir des fichiers pour traiter leur contenu :

```
fichier = open("mon_fichier.txt", "r")
contenu = fichier.read()
fichier.close()
```

La fonction open() prend en premier paramètre le chemin du fichier à ouvrir, et en deuxième paramètre il prend le mode d'ouverture. Voici un bref résumé des modes d'ouverture :

Modes d'ouverture	Descriptions
r	Ouvre un fichier en lecture seule.
w	Ouvre un fichier en écriture.
a	Ouvre un fichier en ajout.

Exceptions

Les exceptions permettent de prendre en compte qu'il y a une possibilité d'erreur lors du traitement d'une bloc d'instructions et d'agir en conséquent.

```
try:
    resultat = 10 / 0
except ZeroDivisionError:
    print("Division par zéro impossible")
```

Remarque : Ici, le cas de l'erreur **ZeroDivisionError** est traité mais le mot clé **except** sans précision du type d'erreur capture toutes les erreurs éventuelles.

POO

La programmation orientée objet (POO) est une physionomie de programmation où l'on travaille non pas avec des variables simples, mais avec des **objets** qui ont des **attributs** (caractéristiques) et des **méthodes** (savoir-faire).

Un objet est définis par une **classe** que l'on pourrait représenter comme étant le schéma descriptif de l'objet.

Classes

Voici un exemple de classe :

```
class MaClasse:
    def __init__(self, nom):
        self.nom = nom

    def afficher_nom(self):
        print("Mon nom est", self.nom)
```

On dit que la classe **MaClasse** a comme attribut **self.nom** et comme méthode **self.__init__()** ainsi que **self.afficher_nom()** .

La méthode **__init__()** est présente dans toutes les méthodes et est exécutée lors de l' **instanciation** (la création) de l'objet.

Elle doit définir les attributs de la classe et prend souvent en paramètre un ou plusieurs attributs de la classe (ici, c'est le cas avec le **nom**) .

Le mot-clé **self** désigne l'objet lui-même.

Instanciation d'objet

Maintenant que la classe est définie, nous pouvons instancier (créer) un ou plusieurs objets à partir de la classe :

```
objet = MaClasse("Objet")
```

Méthodes

Les méthodes sont des fonctions propres à la classe :

```
objet.afficher_nom()
```

La particularité des méthodes par rapport aux fonctions est la présence de l'objet (self) qui est implicitement passé en premier paramètre à la méthode lors de l'appel de celle-ci.

Remarque : Puisque self ne doit pas être passé, il faut considérer le premier paramètre de l'appel de la méthode comme étant le deuxième décrit dans la classe.

Getter / Setter

Les décorateurs permettent de définir des **getters** et des **setters** pour vos attributs.

Pour rappel, un getter est une méthode qui retourne la valeur d'un attribut et un setter permet de définir la valeur d'un attribut depuis l'extérieur de la classe.

L'implémentation de ces deux types de fonction doit se faire avec des **décorateurs** @ en python.

D'ailleurs, il est obligatoire de définir un getter et un setter même si vous n'en souhaitez qu'un des deux.

Voici comment implémenter les guetters et setters :

```
class MaClasse:
    def __init__(self):
        self._ma_valeur = None  # Attribut privé, convention avec un préfixe _

    @property
    def ma_valeur(self):
        return self._ma_valeur

    @ma_valeur.setter
    def ma_valeur(self, nouvelle_valeur):
        if nouvelle_valeur < 0:
            raise ValueError("La valeur ne peut pas être négative")
        self._ma_valeur = nouvelle_valeur

# Utilisation de la classe
objet = MaClasse()

# Accès à l'attribut via le getter
print(objet.ma_valeur)  # Affiche None car aucune valeur n'a été définie encore

# Utilisation du setter pour définir une nouvelle valeur
objet.ma_valeur = 42

# Accès à la nouvelle valeur via le getter
print(objet.ma_valeur)  # Affiche 42

# Essayons de définir une valeur négative, cela lèvera une exception
objet.ma_valeur = -5  # Cela lève une ValueError
```

Héritage

Une classe en python peut donner naissance à une ou plusieurs classes dérivées qui bénéficieront des attributs et des méthodes de la classe mère ainsi que de leur nouveaux potentiels attributs et méthodes.

Voici un exemple avec un classe mère Animal et deux classes filles Chien et Chat :

```
# Classe mère (superclasse)
class Animal:
```

```
    def __init__(self, nom, age):
        self.nom = nom
        self.age = age

    def courir(self):
        print("Je cours")



# Classe enfant (dérivée)
class Chien(Animal):
    def __init__(self, nom, age, race):
        # Appeler le constructeur de la classe parente
        super().__init__(nom, age)
        self.race = race

    def parler(self):
        return "Woof!"

# Classe enfant (sous-classe) différente
class Chat(Animal):
    def __init__(self, nom, age, couleur):
        # Appeler le constructeur de la classe parente
        super().__init__(nom, age)
        self.couleur = couleur

    def chasser(self):
        return "RIP la souris..."

# Créer des instances d'objets
mon_chien = Chien("Rex", 3, "Labrador")
mon_chat = Chat("Misty", 2, "Gris")

# Utiliser les méthodes héritées
print(mon_chien.nom)  # Affiche "Rex"
print(mon_chat.age)   # Affiche 2

# Appeler les méthodes spécifiques à chaque classe enfant
print(mon_chien.parler())  # Affiche "Woof!"
print(mon_chat.chasser())   # Affiche "RIP la souris..."
```

Polymorphisme

Le polymorphisme permet de redéfinir les méthodes de la classe mère dans les classes filles afin de modifier le comportement de cette méthode lors de son exécution.

Voici un exemple concret :

```
# Classe parente (superclasse)
class Animal:
    def __init__(self, nom):
        self.nom = nom

    def parler(self):
        pass  # Cette méthode sera redéfinie dans les classes filles

# Classe fille
class Chien(Animal):
    def parler(self):
        return "Woof!"

# Classe fille différente
class Chat(Animal):
    def parler(self):
        return "Miaou!"

# Fonction qui utilise le polymorphisme
def faire_parler(animal):
    print(f"{animal.nom} dit: {animal.parler()}")

# Créer des instances d'objets
mon_chien = Chien("Rex")
mon_chat = Chat("Misty")

# Utiliser la fonction avec différentes instances
faire_parler(mon_chien)  # Affiche "Rex dit: Woof!"
faire_parler(mon_chat)   # Affiche "Misty dit: Miaou!"
```

Arguments

Lors de l'exécution de vos scripts, il est possible de passer au script des arguments pour s'en servir comme variables :

```
python monScript.py <ARG_1> <ARG_2> <ARG_3>
```

À l'aide du module standard **sys**, il est possible de récupérer ces arguments directement depuis une liste :

```
import sys


for arg in sys.argv[]:
  print(arg)
```

Si nous faisons cela, nous verrons apparaître tous les arguments ainsi que le nom du fichier du script dans **sys.argv[0]** puisqu'il est considéré comme le premier paramètre.

Pour corriger cela et ne récupérer que les vrais arguments, nous pouvons faire comme cela :

```
for arg in arguments[1:]:
  print(arg)
```

Revision #19
Created 11 September 2023 17:10:11 by Elieroc
Updated 1 August 2024 13:08:31 by Elieroc