INTRODUCTION À PYTHON

INF5190 - AUTOMNE 2019

JEAN-PHILIPPE CAISSY

CAISSY.JEAN-PHILIPPE@UQAM.CA

HTTPS://CAISSY.DEV/INF5190

QU'EST-CE QUE PYTHON?

- Langage interprété comme Node, Ruby, PHP, etc
- C'est une norme, plusieurs implémentations (CPython, Jython, IronPython, PyPy)
- Orienté objet
- Mature : plus de 25 d'âge
- Multiplateforme
- Garbage Collector
- Console interactive!
- *Batteries-included*
- Utilisé par toute sorte d'industrie

PYTHON 2 VS. PYTHON 3

- Pas de rétro-compabilité
- Support unicode complet
- Plusieurs nouvelles fonctionalités
- Migration lente, communauté divisée
- Même s'il reste de gros projets sous Python 2, en 2019 c'est Python 3!

CONSOLE

Python vient avec une console interactive.

Démonstration rapide!

EXÉCUTION D'UN SCRIPT

```
$ cat hello.py
text = "Hello World!"
print(text)
```

```
$ python3 hello.py
Hello World!
```

SYNTAXE

- Aucun *curly-braces { ou }*!
- L'indentation sert à délimiter les blocs de code, comme en CoffeeScript.

```python
if taille > 50:
    print("Plus grand que 50")
else:
    while taille <= 50:
        if taille % 2 == 0:
            print(taille)

        taille -= 1
```

```java
if(taille > 50) {
    System.out.println("Plus grand que 50");
} else {
    while(taille <= 50) {
        if(taille % 2 == 0) { System.out.println(taill
        taille -= 1
    }
}
```

INDENDATION

Il faut faire attention avec l'identation! On ne peut pas mélanger des tabluations et des espaces.

```
$ cat indendation.py
if True:
        print("Cette ligne est indentée avec un tab")
        print("Cette ligne est indentée avec des espaces")
```

```
$ python3 indendation.py
  File "indendation.py", line 3
    print("Cette ligne est indentée avec des espaces")
                                                     ^
TabError: inconsistent use of tabs and spaces in indentation
```

FLOT DE CONTÔLE

- if/elif/else
- for
- while

Une déclaration se termine avec un deux-point **:**

IF/ELIF/ELSE

```
if condition1:
    #faire quelquechose
elif condition2:
    #faire autre chose
elif condition3:
    #faire 3e chose
else:
    #sinon ...
```

WHILE

```
i = 0
while i < 10:
    i += 1

print(i) #affiche 10
```

FOR

Il n'y pas de boucle for comme en C ou java avec déclaration d'entier et incrément (`for i = 0; i < 10; ++i`)

On utilise `for` pour naviguer dans des listes.

Avec la méthode `range`, on peut créer une liste de 0 à x éléments :

```
for i in [0, 1, 2, 3, 4]:
    print(i)
    #va afficher les chiffres de 0 à 4

for i in range(10, 20):
    print(i)
    #va afficher les chiffres de 10 à 19
```

Depuis Python 3, plusieurs valeurs de retour sont devenus des itérateurs au lieu de listes.

Python 3

```
>>> range(10)
range(0, 10)
```

Python 2

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Pour forcer le retour d'une liste, on doit convertir l'objet

```
>>> list(range(0, 10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

TYPES DE DONNÉES

En Python, tout est objet.
Chaque type de donnée est en fait un objet.

- Bool: True/False
- Integer: 1, 2, 3, etc
  Aucune limite sur les entiers.
- Float (IEEE 754)
- Decimal: nombre flottants à valeurs fixes
- String : chaîne de caractère unicode
- Bytes: un tablea d'octets

TYPES DE DONNÉES

- Liste/Tableau : `[1, True, "123"]`
- Dictionnaire : `{'cle 1': 123, 123: 'valeur 2'}`
- Tuples : `('A', 'B', 'C', 12)`

STRING

Depuis Python 3, les strings sont des strings encodés avec unicode.

Une chaîne de caractère est en fait un tableau de caractère.

```
chaine = "INF5190-30"
print(chaine[2]) #Affiche la lettre 'F'
print(chaine[-1]) #Affiche le chiffre '0'

print(chaine.lower()) #affiche "inf5190-30"
print("foobar".upper()) #affiche "FOOBAR"
```

STRING

La concaténation de chaîne peut se faire de plusieurs manière. Depuis Pyhton 3.7, il existe un interpolation litérale.

```
prenom = "Jean-Philippe"
print(f"Bonjour {prenom}")

nom = "Caissy"
print(f"Re bonjour {prenom} {nom}")
```

STRING

On peut aussi utiliser la méthode `format`

```
prenom = "Jean-Philippe"
print("Bonjour {0}".format(prenom))

nom = "Caissy"
print("Re bonjour {0} {1}".format(prenom, nom))
```

LIST

```
ma_liste = [1, 'b', 'c']
if 'b' in ma_liste:
  print("'b' se trouve dans ma_liste")

ma_liste.append('2')
print(ma_liste) #[1, 'b', 'c', '2']

ma_liste.pop() # enlève et retourne le dernier élément
print(ma_liste) #[1, 'b', 'c']
print(ma_liste[1]) #Affiche 'b'
```

LIST

```
ma_liste = [0, 2, 4]
i = ma_liste[-1] + 1 # i = 4 + 1
while i < 20:
    if i % 2 == 0:
        ma_liste.append(i)
    i += 1
```

```
print(ma_liste) # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

COUPAGE (*SLICE*) DE LISTE

```
ma_liste = [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

print(ma_liste[2:5]) # [4, 6, 8]

print(ma_liste[:5]) # [0, 2, 4, 6, 8]

print(ma_liste[5:]) # [10, 12, 14, 16, 18]

print(ma_liste[:]) # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

DICTIONNAIRES

```
cle = 567
mon_dict = {
  'abc': True,
  123: 'def',
  cle: ['1', '2', '3']
}

print(mon_dict['abc']) #retourne True
mon_dict[123] = 'fed'

del mon_dict['abc'] #supprime la clé 'abc'

print('abc' in mon_dict) # True
print('123' in mon_dict) # False (la clé est un entier)
print(123 in mon_dict) # True
```

TUPLE

Un tuple est une liste immuable : une fois défini on ne peut pas rajouter ou supprimer d'éléments.

```
valeurs = ('patate', 'foobar', 123)
print(valeurs[1]) #affiche 'foobar'

valeurs[3] = 'abc' #Erreur
valeurs[2] = 'abc' #Erreur, impossible de modifier un élément
```

MÉTHODES

```
def ma_methode(param_1):
    if param_1:
        print("param_1 est défini")

def autre_methode():
    ma_methode(true)
    ma_methode(param_1='Patate')
```

Arguments nommés et arguments indexés par position

```python
def ma_methode(param_1, param_2):
    print(f"{param_1}, {param_2}")
```

```python
ma_method("toto", "tata")

ma_method(param_1="toto", param_2="tata")

ma_method("toto", param_2="tata")

ma_method(param_2="tata", "toto")
#         ^------ Génère une exception.
#                 Les arguments nommés doivent être à la fin.
```

Une méthode peut avoir un nombre indéfini de paramètres

```python
def faire_qqchose(param1, *args):
    print(param1, args)

faire_qqchose('foobar', 'patate', 123)
#Affiche 'foobar' ['patate', 123]
```

`args` va contenir une liste de tous les paramètres supplémentaires que la méthode reçoit.

Une méthode peut avoir un nombre indéfini de paramètres nommés !

```python
def faire_qqchose(param1, **kwargs):
    print(param1, kwargs)

faire_qqchose('foobar', cours='web', groupe='20')
#Affiche 'foobar' {'cours': 'web', 'groupe'='20'}
```

`kwargs` va contenir un dictionnaire de tous les paramètres nommées passé à la méthode.

On peut mélanger les deux !

```python
def faire_qqchose(param1, *args, **kwargs):
    print(param1, args, kwargs)

faire_qqchose('foobar', 42, 'hello', cours='web', groupe='20')
#Affiche 'foobar' [42, 'hellp'] {'cours': 'web', 'groupe'='20'
```

LES CLASSES/OBJETS

```
class Personne(object):
    nom = "Jean-Philippe"
    courriel = "caissy.jean-philippe@uqam.ca"

    def afficher_nom(self):
        print("{0} <{1}>".format(self.nom, self.courriel))
```

Vous remarquerez qu'on passe l'instance de la classe comme premier paramètre (self). La norme est d'utilisé `self` mais ça peut être n'importe quel nom de paramètre.

```
class Personne(object):
    nom = "Jean-Philippe"
    courriel = "caissy.jean-philippe@uqam.ca"

    def afficher_nom(moi_meme):
        print("{0} <{1}>".format(moi_meme.nom, moi_meme.courri
```

```python
class Personne(object):
    nom = "Jean-Philippe"
    courriel = "caissy.jean-philippe@uqam.ca"

    def afficher_nom(self):
        print("{0} <{1}>".format(self.nom, self.courriel))
```

```python
moi = Personne()
moi.afficher_nom() #Affiche Jean-Philippe <caissy.jean-philipp
```

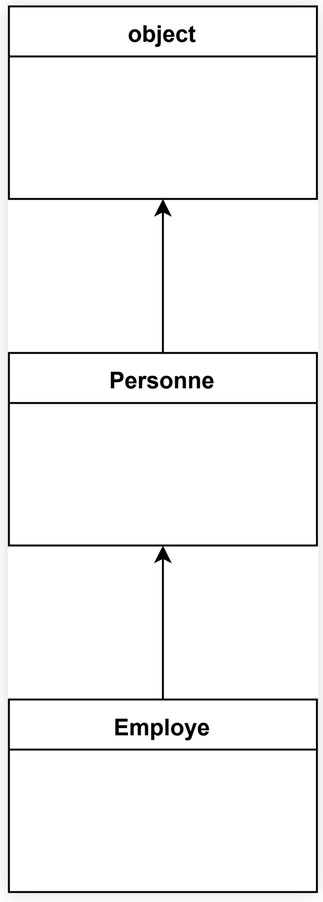
```python
moi.nom = "Éric Beaudry"
moi.courriel = "beaudry.eric@uqam.ca"
moi.afficher_nom()
```

```python
class Personne(object):
#                 ^------ héritage explicite de la classe objec
    nom = "Jean-Philippe"
    courriel = "caissy.jean-philippe@uqam.ca"

    def afficher_nom(self):
        print("{0} <{1}>".format(self.nom, self.courriel))
```

```python
class Personne:
#               ^------ héritage implicite de la classe object
    nom = "Jean-Philippe"
    courriel = "caissy.jean-philippe@uqam.ca"

    def afficher_nom(self):
        print("{0} <{1}>".format(self.nom, self.courriel))
```

HÉRITAGE DE CLASSE

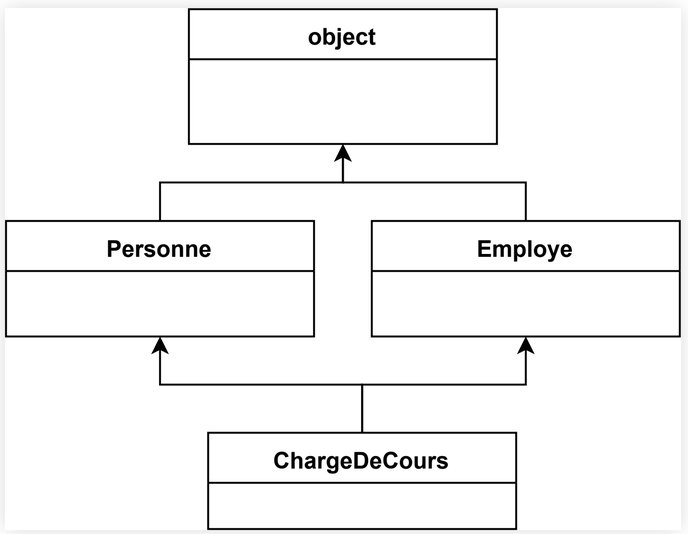
```python
class Personne(object):
    def afficher_nom(self):
        print(self.nom)

class Employe(Personne):
    def afficher_status(self):
        print("Employé")
```



```python
class Personne(object):
    def afficher_nom(self):
        print(self.nom)

class Employe(Personne):
    def afficher_status(self):
        print("Employé")

emp = Employe()
emp.nom = "Jean-Philippe"
emp.afficher_nom()
emp.afficher_status()
```

HÉRITAGE MULTIPLE

```python
class Personne(object):
    def afficher_nom(self):
        print(self.nom)

class Employe(object):
    def afficher_poste(self):
        print("Employé")

class ChargeDeCours(Personne, Employe):
    def afficher_cours(self):
        print(self.cours)
```



```python
class Personne(object):
    def afficher_nom(self):
        print(self.nom)

class Employe(object):
    def afficher_poste(self):
        print("Employé")

class ChargeDeCours(Personne, Employe):
    def afficher_cours(self):
        print(self.cours)
```

```python
emp = ChargeDeCours()
emp.nom = "Jean-Philippe"
emp.cours = "INF5190"
emp.afficher_nom()
emp.afficher_poste()
emp.afficher_cours()
```

MÉTHODES MAGIQUES

Méthodes spéciales d'une classe. utilisés comme constructeurs de classe, surcharge d'opérateurs. Le format des méthodes est toujours entourées de deux *underscores*.

Nom	Description
`__init__(self, autre)`	Constructeur de classe
`__del__(self)`	Destructeur de classe (rarement utilisé)
`__str__(self)`	Cast vers un string
`__eq__(self, autre)`,`__ne__(self, autre)`	Surcharge d'opérateurs `==` et `!=`
`__ge__(self, autre)`,`__le__(self, autre)`,`__lt__(self, autre)`	Surcharge d'opérateurs `>=`, `<=` et `&t;`
`__add__(self, autre)`,`__sub__(self, autre)`,`__mul__(self, autre)`,`__div__(self, autre)`	Surcharge d'opérateurs `+`, `-`, `*` et `/`

```python
class Employe(Personne):
    def __init__(self, code_ms, nom):
        self.code_ms = code_ms
        self.nom = nom

    def __str__(self):
        return f"Employé {self.nom}"

    def __eq__(self, other)
        return self.code_ms == other.code_ms
```

```python
employe_a = Employe("caij", "Jean-Philippe Caissy")
employe_b = Employe("beae", "Eric Beaudry")

print(employe_a) # retourne "Employé Jean-Philippe Caissy"
print(employe_a == employe_b) # retourne False
```

QUESTIONS?

LA SEMAINE PROCHAINE :

- SUITE DE L'INTRODUCTION À PYTHON (MODULES, PACKAGING, ENVIRONEMENT DE DEVELOPEMENT, WSGI)
- FONCTIONNEMENT D'UNE APPLICATION WEB