Python3 Cheat sheet

Une feuille recapitulant les notions de Python3 vues par les étudiants ayant suivi le module jgb53d ("Bases de données et programmation").

Variables

Types

- **int** : un entier (e.g 1, -3, 19,...)
- **float** : un réel (0.23, -0.15, 5e-06)
- **str** : une chaîne de caractères.
- **bool** : un booléen (True, False).
- **NoneType** : pour une variable dont la valeur est **None** (*i.e.* non définie).

- type(objet) renvoie le type de l'objet (str, int,...).

Notation

- On préfèrera l'utilisation de "-" pour les variables dont le nom est long (e.g une_jolie_variable).
- Attention aux noms réservés en python (and, if, class, def,...).

Opérations sur les chaînes de caractères

Notez que la transformation d'une chaîne nécessite d'utiliser l'opérateur d'assignation (e.g. a = a.lower()).

```
a="Hello"
b="world\n"
v = "" # une chaîne de caractères vide.
type(a) # un objet str
help(str) # aide sur l'objet str
c = a + " " + b # concaténation → "Hello world"
c[0] # 'H'
c[0]*3 # 'multiplication'  → 'HHH'
len(b) # longueur → 6
b = b.capitalize # 'world\n' → 'World\n'
b = b.rstrip("\n") #  enlève le dernier caractère
b = b.lstrip("W") # enlève le premier caractère
a.count("l") # nombre d'occurrences → 2
a.endswith("o") # True
a.startswith("H") # True
a.find("o") # position du premier 'o' → 4
a.index("o") # position du premier 'o' → 4
a.lower() # minuscule ('lowercase')
a.upper()# majuscule ('uppercase')
a.replace("l","g") # 'Heggo'
conv = str.maketrans("He","Bu")
a.translate(conv) # Bullo
a.split("e") # → ['H', 'llo']
"3".zfill(4) # → '0003'
```

Opérations sur les numériques

```
a=3.2
b=4
c = a + b # addition
d = a * b  # multiplication
e = a / b # division
import math  # Opérations mathématiques avancées
math.sqrt(2) # racine carrée (square root)
a**2 # puissance
math.floor(3.14) # partie entière → 3
math.ceil(3.14)  # partie entière par excès → 4
math.ceil(3.14)  # fonctions trigonométriques (...)
math.log10(100)  # logarithme (base 10)
```

Opérations logiques

```
a == b  # True si a et b sont égaux.
a != b  # True si a et b sont différents.
a > b   # True si a est supérieure à b.
a < b   # True si a est inférieure à b.
a >= b  # True si a est supérieure ou égale à b.
a <= b  # True si a est inférieure ou égale à b.
c <= a <= b # True si a est supérieure ou égale à c
    et inférieure ou égale à b.
c < a < b # True si a est strictement supérieure à
    c et inférieure à b.
a < d  and b > c # ET logique.
a < d  or b > c # OU logique.
not a == b # Négation logique.
```

Conversions

```
int(3.0)     # conversion en entier
str(30)      # conversion en chaîne
float("30.2")  # conversion en réel
bool(0)      # conversion en booléen
```

Listes

Une liste permet de stocker un ensemble d'objets. On peut accéder aux éléments de la liste en désignant des positions (indices) dans celle-ci.

```
l = list() # déclare une liste vide
l = []     # déclare une liste vide
l = ["a", "b", "a", "c"] # une liste à 4 éléments
len(l) # taille de la liste → 3
l.count("a") # compter le nb. d'occurrences de "a"
l.append("e") # ajoute un élément (à la fin)
l.insert(p, "t") # ajoute "t"  avant la position p
e = l.pop(p)  # délète/renvoie l'élément en pos. p
l.index("c") # la position de l'élément "c"
l.reverse() # inverse l'ordre
l.sort() # tri alphanumérique
l + ["j", "k"] # concaténation
l += "e"  # ajoute à la fin
l * 3 # multiplication
"a" in l # test la présence de "a" dans l
```

Dictionnaires

Dans un dictionnaire on stocke des objets (valeurs) accessibles par une clef. On n'aura pas besoin de connaitre la position de l'objet mais le nom de sa clef pour y accéder.

```
prix = dict() # déclare un dictionnaire vide
prix = {} # déclare uun dictionnaire vide
# initialisation avec plusieurs éléments
prix = {"pomme": 3, "cerise": 8}
# ajoute une valeur (8) pour la clef 'poire'
prix['poire'] = 4
prix['poire']  # valeur pour la clef 'poire'
del prix['poire'] # délète le couple 'poire'/4
prix.keys()    # l'ensemble des clefs
len(prix.keys()) # le nombre de clefs
prix.values() # l'ensemble des valeurs
len(prix.values()) # le nombre de valeurs
```

Boucles

Boucles for

Il est souvent nécessaire de réaliser des itérations (boucles). Elles sont très utilisées notamment pour parcourir des listes ou les clefs/valeurs d'un dictionnaire. Les deux types de boucles que l'on rencontre le plus fréquemment sont les boucles for et les boucles while. Attention ! En python, l'indentation fait partie de la syntaxe. En particulier, le domaine d'application d'une boucle est défini par les lignes qui sont en retrait par rapport à l'instruction initiale.

```
### Effectuer une opération n fois
n = 100
for i in range(n): # 0..99
        print("je le fais")

# Parcourir des listes
chroms = ["Chr1", "Chr2", "Chr7"]

for c in chroms:
        print(c)

# parcourir des listes
# c contient la valeur en position p
for p,c in enumerate(chroms):
        print(p, c)

# Parcourir des chaînes de caractères
dna = 'ATGCTCGCTCGCTCGATGAAAATGTG'

# On passe en revue les nucléotides
# contenus dans la séquence
for  nuc in dna:
        print("La séquence contient "+ nuc)

#Parcourir un Dictionnaire
login2passwd = {"Alain":"abc123", "John":"qwerty"}

# Afficher les clefs
for key in login2passwd:
        print(key)

# Afficher les clefs et valeurs
# Il faut utiliser la méthode item()
for key, value in login2passwd.items():
        print(value, key)
```

Break et continue

Comme dans beaucoup d'autres langages, le mot clef **break**, permet de 'casser' une boucle, de l'arrêter avant qu'elle n'arrive à sa fin. Le mot clef **continue** permet de passer directement à la prochaine itération, sans effectuer la suite des instructions de cette itération.

```
## Exemple 1
# La sortie est:
# 0 1 2 3 4
for i in range(10):
        if i==5:
                break
        print(i)

## Exemple 2
# La sortie est
```

```
# 1 2 8 9 10
i = 0
while i < 10:
        i += 1
        if  2 < i < 8:
                continue
        print(i)
```

Boucle while

```
# Tant que la condition est vraie
# alors le jeu d'instructions est exécuté
a = 0
while a < 10:
    print(a)
    a+=1
```

Structures conditionelles

Une structure conditionnelle permet de tester si une condition est vraie ou fausse. Ces structures conditionnelles évaluent, de manière générale, une expression logique. Dans la structure if...else classique, on exécutera un certain nombre d'instructions si la condition est vraie, et un autre jeu d'intructions si celle-ci s'avère fausse.

```
# Une structure if simple:
if a > b :
      instructions

# Structure if .. else
if a > b :
      instructions
else:
      instructions

# la structure if...elif...else:
if a > b:
     instructions
elif c > d:
     instructions
else:
     instructions

# Structures if...else imbriquées

if a > b:
        instructions
        if c > d:
               instructions
else:
        instruction(s)
```

Entrées / sorties

La fonction input

```
# capture d'une entrée utilisateur
a = input("Entrez un nombre: ")
a = int(a) # input renvoie un objet str.
```

La fonction open

```
# ouverture d'un fichier en lecture.
# Parcours puis fermeture
f = open("file.txt", "r")

for line in f:
     instructions

f.close()

# ouverture d'un fichier en écriture
f = open("file.txt", "w")
f.write("blabla")
print("bidule", file=f)
f.close()
```

r	Ouvre un fichier en lecture. Défaut.
w	Ouvre un fichier en écriture.
a	Ajouter des lignes dans un fichier. Le pointeur est placé à la fin du fichier s'il existe, sinon un nouveau fichier est créé.
r+ ou w+ ou a+	Ouvre le fichier en lecture et en écriture.

Définition de fonctions

Définition et appel

La création d'une fonction se fait grâce au mot-clé **def**. Celui-ci est suivi du nom de la fonction. On décrit ensuite les opérations qui doivent être effectuées par la fonction en marquant l'indentation. Le programme python peut alors exécuter, ou appeler cette fonction. On peut alors demander à la fonction de renvoyer son résultat avec le mot clé **return**.

```
# Exemple de fonction simple
def exponential(x, y):
        return x ** y
result = exponential(2,4)
```

Passage d'arguments

```
#  Exemple d'appel
def printBreakfast(a, b):
        print("I'll have ", a , " toast(s) and ",
             b, " egg(s)")

printBreakfast(2, 3)
printBreakfast(a=2, b=3)

#  Arguments avec valeurs par défault
def printBreakfast(a=2, b=3):
        print("I'll have ", a , " toast(s) and ",
             b, " egg(s)")

printBreakfast()
```

Aide

Pour obtenir de l'aide sur un objet, utilisez la fonction **help**.

```
help(str)
# /start pour rechercher la chaîne de caractères
     'start'
# n (next) pour la prochaine occurence
# p (previous) pour l'occurence précédente
# q pour quitter
```

Divers

Caractères accentués

Par défaut, Python ne supporte pas les caractères accentués. Si l'on souhaite en utiliser il convient d'indiquer en première ligne du fichier :

```
# -*- coding: utf8 -*-
```

Lancer ipython3 dans un terminal

Dans un terminal taper :

```
ipython3
```

Executer le code Python

Pour executer le code python présent dans un fichier via un terminal, plusieurs solutions sont possibles.

```
# Solution 1
python3 leFichier.py
```

Dans la deuxième solution, indiquer en première ligne du fichier que le code doit être executer par python en utilisant un 'shebang' (#!).

```
#!/usr/bin/env python
...
...
```

Ensuite executez ce code après avoir rendu le fichier executable.

```
chmod u+x leFichier.py
./leFichier.py
```

Remerciements

Ce travail a été réalisé avec LaTeXd'après un patron proposé par Winston Chang.