# Cheatsheet Python - Sommaire

## 1. Structures conditionnelles et répétitives

### 1.1. Instructions `if/else` et opérateurs logiques

### 1.2. Boucles `for` et `while`

### 1.3. Instructions de contrôle de boucle (`break`, `continue`, `else` dans les boucles)

## 2. Chaînes de caractères

### 2.1. Opérations de base sur les chaînes (`len`, concaténation, slicing)

### 2.2. Méthodes courantes (`split`, `join`, `replace`, `upper`, `lower`)

### 2.3. F-strings et formatage

## 3. Listes

### 3.1. Création et manipulation de listes

### 3.2. Méthodes courantes (`append`, `remove`, `sort`, `reverse`, `index`)

### 3.3. List comprehensions

## 4. Dictionnaires

### 4.1. Création et manipulation de dictionnaires

### 4.2. Méthodes courantes (`keys`, `values`, `items`, `get`)

### 4.3. Boucles sur les dictionnaires

## 5. Procédures et fonctions

### 5.1. Définir des fonctions avec `def`

### 5.2. Paramètres (obligatoires, optionnels, `\*args`, `\*\*kwargs`)

### 5.3. Valeurs de retour et portée des variables

## 6. Manipulation des fichiers

### 6.1. Ouverture et fermeture de fichiers (`open`, `close`)

### 6.2. Modes d'ouverture (`r`, `w`, `a`, `r+`)

### 6.3. Lecture et écriture dans les fichiers (`read`, `readline`, `write`)

### 6.4. Gestion des fichiers avec `with`

## 7. Outils pratiques

### 7.1. Fonctions intégrées (`range`, `enumerate`, `zip`, `map`, `filter`)

### 7.2. Gestion des erreurs avec `try/except`

### 7.3. Importation de modules (`math`, `os`, `json`)

# 1. Structures conditionnelles et répétitives

## 1.1. Instructions `if/else` et opérateurs logiques

```python

if condition1:

# Instructions si condition1 vraie

elif condition2:

# Instructions si condition2 vraie

else:

# Instructions si aucune condition vraie

```

Opérateurs logiques : `and`, `or`, `not`

## 1.2. Boucles `for` et `while`

```python

for element in sequence:

# Instructions exécutées pour chaque élément de la séquence

while condition:

# Instructions exécutées tant que la condition est vraie

```

## 1.3. Instructions de contrôle de boucle

- `break` : sortir de la boucle

- `continue` : passer à l'itération suivante

- `else` dans les boucles : instructions exécutées si la boucle se termine normalement (sans `break`)

```python

for element in sequence:

if condition:

break # Sortir de la boucle

if condition:

continue # Passer à l'itération suivante

else:

# Instructions si la boucle se termine normalement

```

# 2. Chaînes de caractères

## 2.1. Opérations de base sur les chaînes

```python

len(chaine) # Longueur de la chaîne

chaine1 + chaine2 # Concaténation

chaine[i:j] # Slicing (sous-chaîne de i à j exclu)

```

## 2.2. Méthodes courantes

```python

chaine.split(separateur) # Séparer la chaîne

separateur.join(liste\_chaines) # Joindre une liste de chaînes

chaine.replace(chaine1, chaine2) # Remplacer chaine1 par chaine2

chaine.upper() # Convertir en majuscules

chaine.lower() # Convertir en minuscules

```

## 2.3. F-strings et formatage

```python

nom = "Alice"

age = 25

f"Bonjour, je m'appelle {nom} et j'ai {age} ans."

"Bonjour, je m'appelle {} et j'ai {} ans.".format(nom, age)

"Bonjour, je m'appelle {0} et j'ai {1} ans.".format(nom, age)

```

# 3. Listes

## 3.1. Création et manipulation de listes

```python

liste = [element1, element2, element3]

liste[i] # Accéder à l'élément d'index i

liste[i:j] # Slicing (sous-liste de i à j exclu)

```

## 3.2. Méthodes courantes

```python

liste.append(element) # Ajouter un élément à la fin

liste.remove(element) # Supprimer la première occurrence de l'élément

liste.sort() # Trier la liste

liste.reverse() # Inverser la liste

liste.index(element) # Renvoyer l'index de la première occurrence

```

## 3.3. List comprehensions

```python

nombres\_carres = [i\*\*2 for i in range(10)]

nombres\_pairs = [i for i in range(10) if i % 2 == 0]

```

# 4. Dictionnaires

## 4.1. Création et manipulation de dictionnaires

```python

dico = {cle1: valeur1, cle2: valeur2, cle3: valeur3}

dico[cle] # Accéder à la valeur correspondant à la clé

```

## 4.2. Méthodes courantes

```python

dico.keys() # Renvoyer la liste des clés

dico.values() # Renvoyer la liste des valeurs

dico.items() # Renvoyer la liste des couples (clé, valeur)

dico.get(cle, valeur\_defaut) # Renvoyer la valeur ou la valeur par défaut

```

## 4.3. Boucles sur les dictionnaires

```python

for cle, valeur in dico.items():

# Instructions exécutées pour chaque paire clé-valeur

```

# 5. Procédures et fonctions

## 5.1. Définir des fonctions avec `def`

```python

def nom\_fonction(parametre1, parametre2=default):

# Instructions de la fonction

return valeur # Renvoyer une valeur

```

## 5.2. Paramètres

- Paramètres obligatoires : `def fonction(param1, param2):`

- Paramètres optionnels : `def fonction(param1, param2=default):`

- `\*args` : liste d'arguments positionnels

- `\*\*kwargs` : dictionnaire d'arguments nommés

## 5.3. Valeurs de retour et portée des variables

- `return` : renvoyer une valeur

- Variables locales : définies à l'intérieur d'une fonction

- Variables globales : définies à l'extérieur des fonctions

# 6. Manipulation des fichiers

## 6.1. Ouverture et fermeture de fichiers

```python

fichier = open("chemin/vers/fichier", mode)

fichier.close()

```

## 6.2. Modes d'ouverture

- `r` : lecture seule

- `w` : écriture (crée un nouveau fichier ou écrase le contenu)

- `a` : ajout (écriture à la fin du fichier)

- `r+` : lecture et écriture

## 6.3. Lecture et écriture dans les fichiers

```python

fichier.read() # Lire le contenu du fichier

fichier.readline() # Lire une ligne du fichier

fichier.write(chaine) # Écrire une chaîne dans le fichier

```

## 6.4. Gestion des fichiers avec `with`

```python

with open("fichier.txt", "r") as fichier:

contenu = fichier.read()

# Le fichier est automatiquement fermé à la fin du bloc with

```

# 7. Outils pratiques

## 7.1. Fonctions intégrées

- `range(n)` : générer une séquence de 0 à n-1

- `enumerate(sequence)` : itérer sur une séquence avec un index

- `zip(seq1, seq2)` : grouper les éléments de deux séquences

- `map(fonction, sequence)` : appliquer une fonction à chaque élément

- `filter(fonction, sequence)` : filtrer les éléments d'une séquence

## 7.2. Gestion des erreurs avec `try/except`

```python

try:

# Instructions susceptibles de provoquer une erreur

except TypeErreur as err:

# Instructions exécutées en cas d'erreur de type TypeErreur

except (Type1, Type2) as err:

# Instructions exécutées en cas d'erreur de type Type1 ou Type2

except Exception as err:

# Instructions exécutées en cas d'erreur quel que soit le type

else:

# Instructions exécutées si aucune erreur n'est levée

finally:

# Instructions exécutées à la fin, qu'il y ait eu une erreur ou non

```

## 7.3. Importation de modules

```python

import module

from module import fonction

from module import classe

import module as alias

from package.module import fonction

```

# 8. Exemples de code

## 8.1. Calcul de la somme des éléments d'une liste

```python

def somme\_liste(liste):

somme = 0

for element in liste:

somme += element

return somme

```

## 8.2. Compter le nombre d'occurrences de chaque élément dans une liste

```python

def compter\_occurrences(liste):

occurences = {}

for element in liste:

occurences[element] = occurences.get(element, 0) + 1

return occurences

```

## 8.3. Lire et écrire dans un fichier

```python

with open("fichier.txt", "r") as fichier:

contenu = fichier.read()

with open("fichier.txt", "w") as fichier:

fichier.write("Contenu à écrire dans le fichier")

```

## 8.4. Utilisation de modules

```python

import math

rayon = 5

surface = math.pi \* rayon\*\*2

```

## 8.5. Gestion des erreurs

```python

def diviser(a, b):

try:

resultat = a / b

except ZeroDivisionError:

print("Division par zéro !")

resultat = None

return resultat

```

## 8.6. Utilisation de list comprehensions

```python

nombres = [1, 2, 3, 4, 5]

carres\_pairs = [i\*\*2 for i in nombres if i % 2 == 0]

```

## 8.7. Création d'une liste de tuples

```python

noms = ["Alice", "Bob", "Charlie"]

ages = [25, 30, 35]

personnes = list(zip(noms, ages))

```

## 8.8. Utilisation de fonctions lambda

```python

f = lambda x: x\*\*2

resultat = f(5)

```

## 8.9. Utilisation de `map` et `filter`

```python

nombres = [1, 2, 3, 4, 5]

carres = list(map(lambda x: x\*\*2, nombres))

pairs = list(filter(lambda x: x % 2 == 0, nombres))

```

## 8.10. Utilisation de `enumerate`

```python

noms = ["Alice", "Bob", "Charlie"]

for index, nom in enumerate(noms):

print(f"Personne {index + 1} : {nom}")

```

## 8.11. Utilisation de `try/except/else/finally`

```python

def diviser(a, b):

try:

resultat = a / b

except ZeroDivisionError:

print("Division par zéro !")

resultat = None

else:

print("Division effectuée avec succès !")

finally:

print("Opération terminée.")

return resultat

```

## 8.12. Utilisation de `os` pour manipuler les fichiers

```python

import os

chemin = "chemin/vers/fichier"

if os.path.exists(chemin):

os.remove(chemin)

else:

print("Le fichier n'existe pas.")

```

## 8.13. Utilisation de `json` pour lire et écrire des fichiers JSON

```python

import json

donnees = {"nom": "Alice", "age": 25}

with open("donnees.json", "w") as fichier:

json.dump(donnees, fichier)

with open("donnees.json", "r") as fichier:

donnees = json.load(fichier)

```

## 8.14. Utilisation de `random` pour générer des nombres aléatoires

```python

import random

nombres = [1, 2, 3, 4, 5]

random.shuffle(nombres)

nombre\_aleatoire = random.choice(nombres)

```

## 8.15. Utilisation de `time` pour mesurer le temps d'exécution

```python

import time

debut = time.time()

# Instructions à mesurer

fin = time.time()

duree = fin - debut

```

## 8.16. Utilisation de `argparse` pour gérer les arguments de ligne de commande

```python

import argparse

parser = argparse.ArgumentParser()

parser.add\_argument("fichier", help="Chemin du fichier à lire")

parser.add\_argument("-o", "--option", help="Option à activer")

args = parser.parse\_args()

print(f"Fichier : {args.fichier}")

if args.option:

print(f"Option : {args.option}")

```

## 8.17. Utilisation de `unittest` pour écrire des tests unitaires

```python

import unittest

def somme(a, b):

return a + b

class TestSomme(unittest.TestCase):

def test\_somme\_positifs(self):

self.assertEqual(somme(2, 3), 5)

def test\_somme\_negatif\_positif(self):

self.assertEqual(somme(-2, 3), 1)

if \_\_name\_\_ == "\_\_main\_\_":

unittest.main()

```