

Python

Pour débutant...



2023/10/25 - Kévin Ta - Full Stack engineer - Proton

À l'issue de ce cours

- Comment installer Python.
- Choisir un éditeur de texte.
- Maîtriser les concepts de base de la programmation.
- Utiliser ces concepts sur Python.
- Création d'une suite de tests automatisés.

Notation

- QCM le vendredi avant la fin de la séance.
- 2 projets individuels.

**Teacher: Exam
will be easy.**

Orange is:

a. Fruit

b. Colour



Avant de démarrer

- Le but de mon intervention est de vous donner des fondations pour pouvoir être autonome et savoir chercher les informations dont vous avez besoin pour coder.
- N'hésitez pas à poser vos questions durant le cours.



Présentation très brève

Qu'est-ce que Python ?

- Langage de programmation généraliste, populaire dans le domaine de la Data Science et Machine Learning.
- Utilisé principalement pour développer le back end d'applications web ou de logiciels.
- Également utilisé pour des scripts d'automatisation.
- C'est un langage interprété (sans compilation nécessaire), dont l'interpréteur est écrit en C (nommé CPython).

Qui utilise Python ?

- Youtube
- Instagram
- Spotify
- Dropbox
- Pinterest
- Netflix
- BitTorrent



50.000 €

Salaire moyen annuel d'un développeur Python

28,05 %

Part de recherches concernant un tutoriel Python parmi tous les langages

Installer git et utiliser GitHub

- Tout au long de ce cours, vous aurez différent exercices. Sauvegardez-les dans votre repo GitHub.
- Le projet final sera à rendre sur GitHub. Entraînez-vous à utiliser git pendant le cours.

- Installer Git pour tout système : <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>

Créez un repo public sur GitHub pour les exercices:

- <https://docs.github.com/en/get-started/quickstart/create-a-repo>
- N'ajoutez pas de README.md.
- Ajoutez un .gitignore pour Python.
- Ne choisissez pas de license.

Installer Python et virtualenv

- Windows : <https://learn.microsoft.com/en-us/windows/python/beginners>
- macOS : <https://docs.python-guide.org/starting/install3/osx/>
- Linux : <https://docs.python-guide.org/starting/install3/linux/>

Vérifiez l'installation, les commandes ci-dessous ne devrait retourner aucune erreur:

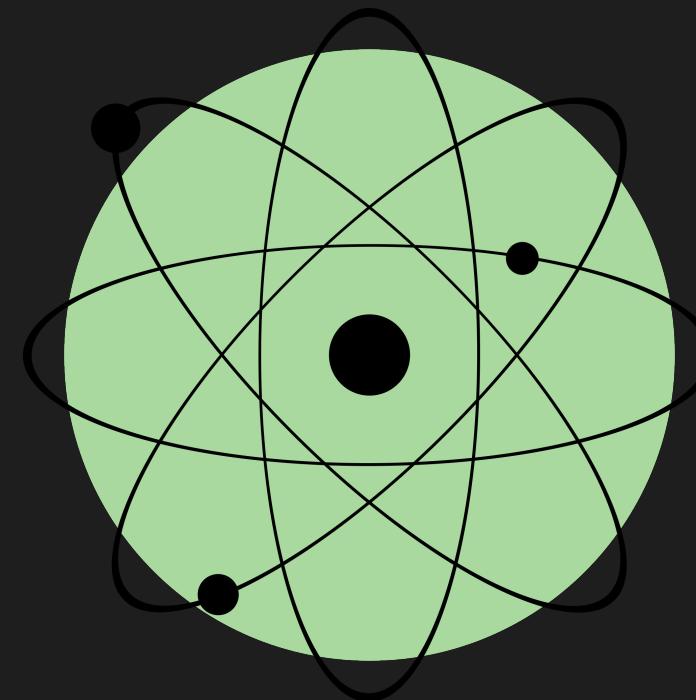
- `python3 --version`
- `pip3 --version`

Installez virtualenv avec la commande suivante:

- pip3 install virtualenv
- (Si la première commande ne fonctionne pas) sudo pip3 install virtualenv

Choisir un éditeur de texte

- PyCharm (avec Github student pack) : <https://www.jetbrains.com/pycharm/>
- Visual Studio Code : <https://code.visualstudio.com/>
- Atom : <https://github.com/atom/atom>



Initialisation de l'environnement

1. Clonez votre repo sur votre ordinateur : <https://docs.github.com/en/repositories/creating-and-managing-repositories/cloning-a-repository>
2. Entrez dans le nouveau dossier créé et créez un virtualenv : <https://python-guide-pt-br.readthedocs.io/fr/latest/dev/virtualenvs.html>
3. Ouvrez le dossier avec votre éditeur de code et créez un fichier app.py vide.

Exercice

Écrivez la ligne suivante dans app.py :

```
print ('Hello world')
```

Exécutez le fichier app.py avec la commande :

```
python3 app.py
```

Observez le résultat



Exercice

Écrire maintenant 2 phrases à la suite:

```
print('Hello world')
print('Welcome to Python')
```

Observer le résultat.

Python vous permet de définir une série d'instructions qui seront exécutées dans l'ordre.

Variables

Variables

- Les variables sont des conteneurs capable de stocker des valeurs (chiffre, mot, phrase, liste...).
- À la différence de C, Python est un langage interprété, il n'y a pas besoin d'avoir un type (int, string, array...).
- Une variable est créée de cette manière :

```
maVariable = 'contenu'  
maVariable = 123  
maVariable = ['Hello', 'World', 1]  
• ...
```

- Une variable est stockée dans le tas (heap), un garbage collector s'occupe de libérer la mémoire.

Exercice

- Créez une variable nommée 'world', assignez-y le chiffre 1 et affichez-la avec `print()`.
- Ensuite, stockez le mot 'World' dans la variable précédente et affichez-la.
- Créez une liste avec 'Hello', 'World' et 123, stockez cette liste dans la variable précédente et affichez cette liste avec `print()`.
- PS: Soyez malin avec 'World', vous pouvez réutiliser la variable...

Solution

```
#!/usr/bin/env python
world = 1
print(world)
world = "World"
print(world)
world = ["Hello", world, 123]
print(world)

1
World
['Hello', 'World', 123]
```

C++: Can not compare
float and int

Python:



Types de données

Type séquentiel - list, tuple, range

- Une liste est déclarée comme ceci: maVariable = [1, 2, 3].
- Un tuple est déclaré comme ceci : maVariable = (1, 'Hello', 3) ou tuple(1, 'Hello', 3).
- Une range est déclarée comme ceci : maVariable = range(10).
- Les listes et ranges servent généralement à stocker des données de même type.
- Les tuples servent généralement à stocker des données de type différent.
- Les types séquentiels prennent en charge les opérations standards :
 - maVariable.append('abc') pour insérer un élément à la fin de la liste ou range
 - maVariable.remove('abc') pour enlever le premier 'abc' de la liste ou range
 - maVariable[0] pour avoir le 1er élément
 - maVariable[-1] pour avoir le dernier élément
 - maVariable[0:2] pour extraire une sous-liste
 - len(maVariable) pour connaître la taille
 - maVariable + maVariable2 pour fusionner 2 variables de type séquentiel
 - <https://docs.python.org/3/library/stdtypes.html#common-sequence-operations>

Type séquence de texte - str

- Une chaîne de caractères peut être écrite de différentes manières :

```
'hello world' # guillemets simples  
"hello world" # guillemets simples (anglais)  
'''hello world''' # guillemets triples  
"""hello world"" # guillemets triples anglais
```

- Une chaîne de caractères implémente plusieurs méthodes comme la mise en majuscule/minuscule, la recherche de caractères :

- `var.uppercase()`, `var.replace('hello', 'world')`, `str(var)`, ...
- <https://docs.python.org/3/library/stdtypes.html#string-methods>

- La concaténation s'effectue avec un opérateur + :

- `maVariable + maVariable2` ou `maVariable + 'hello'`

- Une chaîne de caractères se comportent comme une liste:

- `maVariable[0]` pour avoir le 1er caractère
- `maVariable[-1]` pour avoir le dernier caractère
- `len(maVariable)` pour connaître la taille
- `maVariable[0:2]` pour extraire une sous-chaîne de caractère

Type numérique - int, float, complex

- Il existe 3 types numériques : entier (int), flottant (float), complexe (complex).
- Toutes les opérations mathématiques sont possibles (sauf avec les complexes) : +, -, *, ** (puissance), / (division), // (division entière), % (modulo)...
- Le type numérique implémente également plusieurs méthodes:
 - <https://docs.python.org/3/library/stdtypes.html#bitwise-operations-on-integer-types>
- Les opérations binaires sont également possibles : <<, >>, |, &, ^, ~
- Diviser un entier donnera un nombre flottant (même 1/1...), pour avoir un entier, il faut ‘caster’ la valeur de retour avec int().

Type boolean - bool

- Sous-type numérique qui ne contient que True ou False.
- Utile pour créer des conditions.

Type de correspondance - dict

- Type similaire à la liste mais permet d'utiliser une clé unique comme indice.
- Utile pour accéder rapidement aux données stockées.
- Déclaré comme ceci : maVariable = {'hello': 'world', 'python': 2, 'table': [1, 2]}
- Accès aux valeurs via les clés : maVariable['hello']
- Avoir toutes les clés : maVariable.keys()
- Avoir toutes les valeurs : maVariable.values()
- Il est impossible d'additionner des dicts tel que {'A'} + {'B'}

Exercice

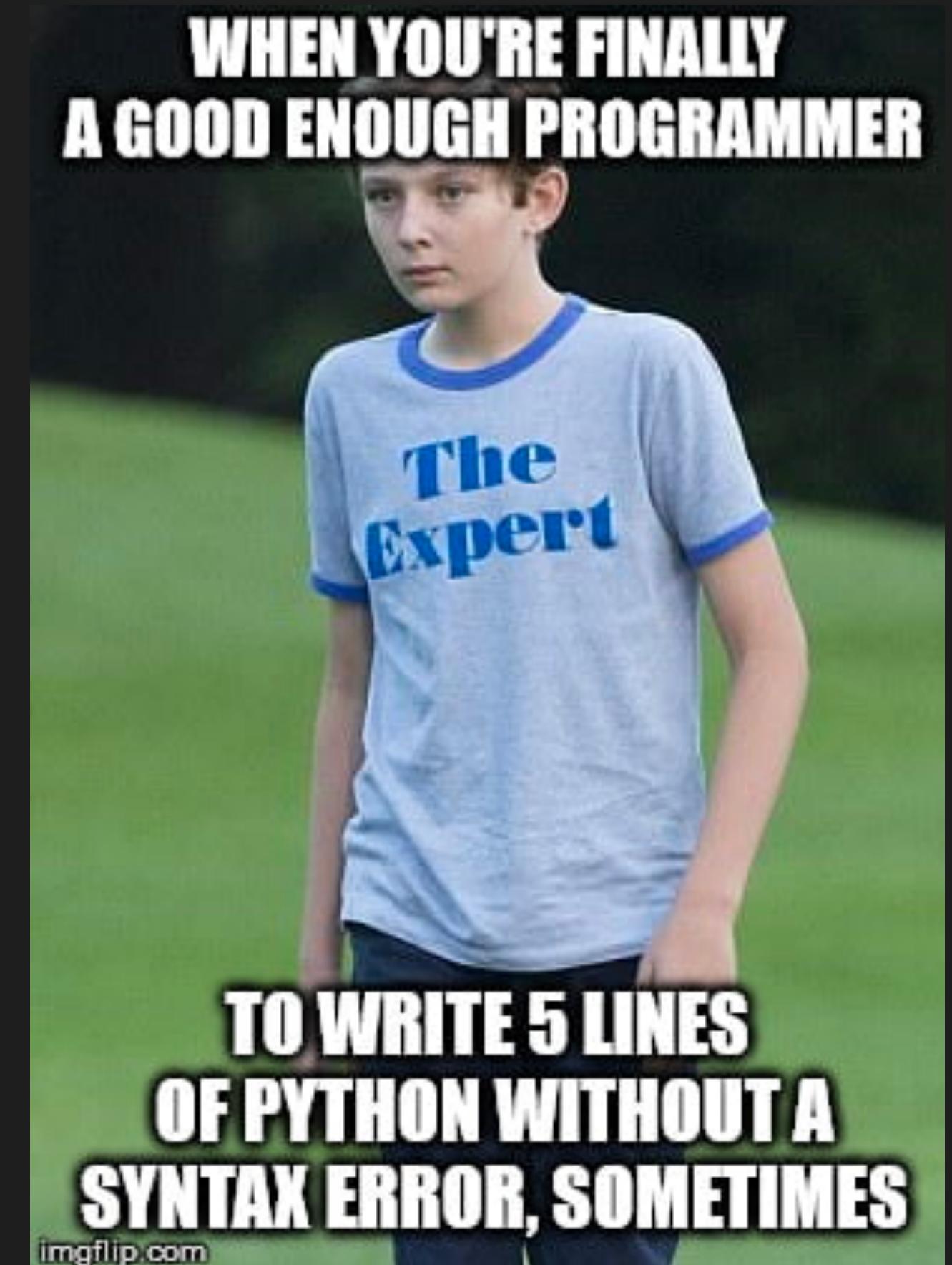
- Vous ne pouvez déclarer que 2 variables au total. Faites-en bon usage.
- Créez une liste contenant 'Bronze', 'Argent', 'Or'.
- Créez une liste contenant 'Bronze', 'Platine', 'Or'.
- Créez un dictionnaire avec comme clé 'rang' et valeur la liste précédente.
Affichez le dictionnaire
- Ajoutez dans ce dictionnaire une nouvelle clé 'niveau' et valeur 30.
- Divisez la valeur de niveau par 2, sans écrire $30 / 2$, ni 15. Affiche le dictionnaire.
- Vous devez obtenir `{'rang': ['Bronze', 'Platine', 'Or'], 'niveau': 15}`

Solution

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

```
niveau = ["Bronze", "Argent", "Or"]
niveau.remove("Argent")
niveau.insert(1, "Platine")
dict = { 'rang': niveau}
print(dict)
dict['niveau'] = 30
dict['niveau'] = int(dict['niveau'] / 2)
print(dict)

{ 'rang': ['Bronze', 'Platine', 'Or'] }
{ 'rang': ['Bronze', 'Platine', 'Or'], 'niveau': 15 }
```

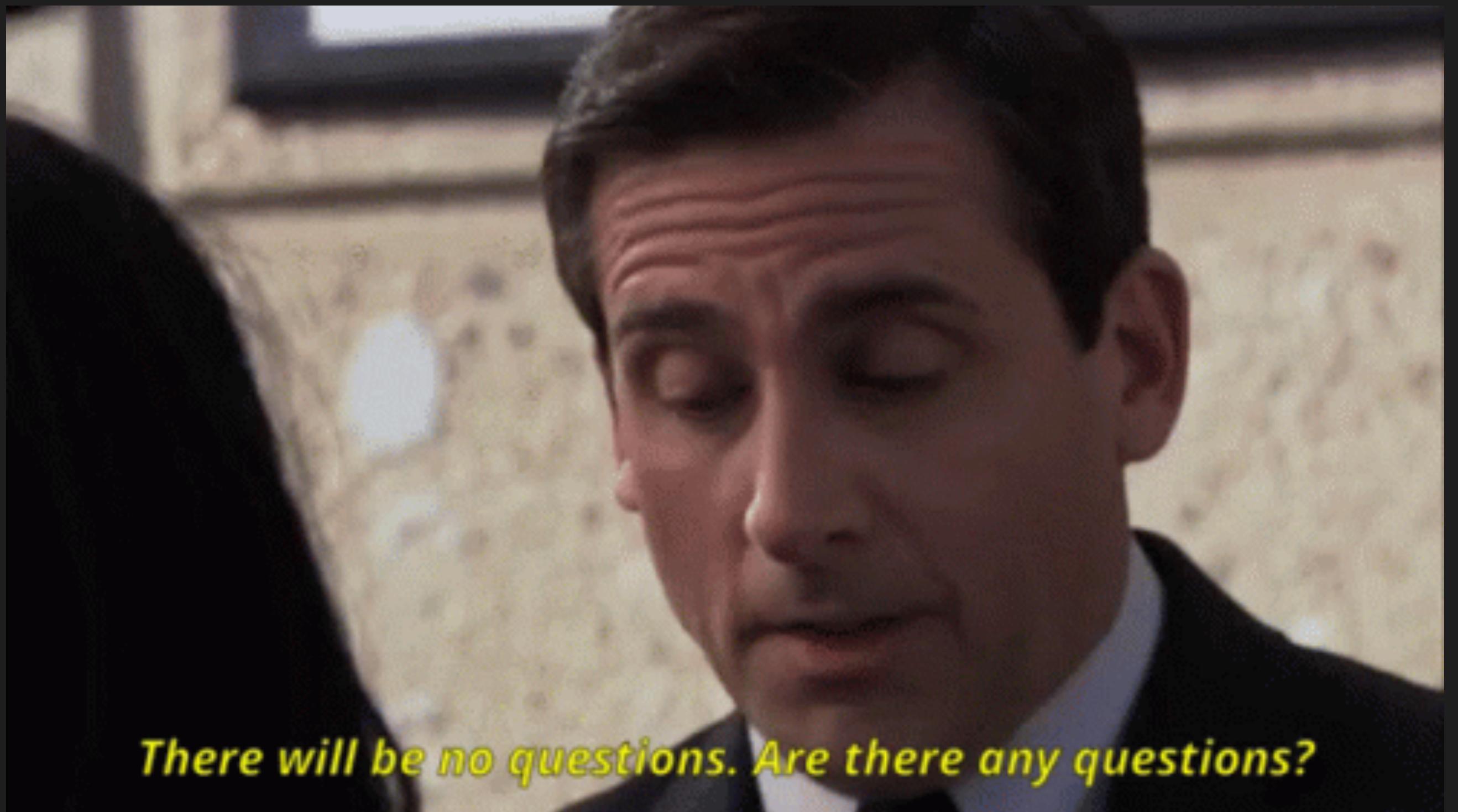


QCM 1



<https://app.wooclap.com/PDQUET>

Questions ?



There will be no questions. Are there any questions?

Interaction avec l'utilisateur

Input

- `Input()` permet d'interrompre le programme et de demander à l'utilisateur d'entrer une value.
- `maVariable = input('Entrez votre nom : ')`

Exercice

- Demandez à l'utilisateur d'entrer 2 nombres.
- Afficher ces 2 nombres dans une phrase en un seul print.
- Additionner ces 2 nombres et afficher le résultat.
- PS: La valeur de retour de input() est toujours une chaîne de caractères...

Solution

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

```
nombre1 = input('Entrez un premier nombre : ')
nombre2 = input('Entrez un deuxième nombre : ')
print('Vous avez entré ' + nombre1 + ' et ' + nombre2 + '.')
print(int(nombre1) + int(nombre2))
```

Entrez un premier nombre :

3

Entrez un deuxième nombre :

2

Vous avez entré 3 et 2.

5

Fonction

Fonction

- Une fonction est un ensemble de code dont le but est d'exécuter une tâche spécifique. Elle doit être déclarée en haut du script, avant le code principal.
- Elle permet de rendre le code plus lisible en divisant clairement les différentes fonctionnalités du programme.
- Python utilise un système d'indentation pour faire la différence entre le code appartenant à une fonction et le reste du code.
- Vous pouvez créer et utiliser des fonctions avec ou sans paramètres :

```
def maFonction(nombre1, nombre2):  
    print(nombre1 + nombre2)
```

```
maFonction(1, 2)
```

Fonction

- Une variable créée dans une fonction est locale à la fonction. Elle ne sera pas accessible depuis le script principal, à moins de la retourner.
- Il est possible de définir une variable avec une portée globale en utilisant l'instruction **global**. Dans le 1er exemple, la valeur de value sera “Local” car défini avec global. Dans le 2e exemple, la variable a est passé à solve par référence, la variable d'origine n'est pas modifié.

```
def func():  
    global value  
    value = "Local"  
  
    value = "Global"  
    func()  
print(value)
```

```
def solve(a):  
    a = [1, 3, 5]  
    a = [2, 4, 6]  
    print(a)  
    solve(a)  
    print(a)
```

Différence entre arguments et paramètres

- Un paramètre représente une valeur que la fonction s'attend à passer lorsque vous lappelez.
- Un argument représente la valeur que vous passez à un paramètre d'une fonction quand vous appelez la fonction.

The diagram shows a code snippet with annotations. An orange bracket labeled "Parameters" spans over the parameter names "param1" and "param2" in the function definition. Two orange arrows point from these labels to the corresponding arguments "5" and "6" in the function call below. A blue bracket labeled "Arguments" spans over the values "5" and "6".

```
function sum(param1, param2){  
    return param1 + param2;  
}  
  
sum(5, 6);
```

Parameters

Arguments

Valeur de retour

- Une fonction peut retourner ou non une valeur de n'importe quel type avec l'instruction `return` :

```
def maFonction(nombre1, nombre2):  
    return [nombre1, nombre2]
```

```
valeur = maFonction(1, 2)
```

- Toute instruction se situant après un `return` ne sera pas exécutée.

*args et **kwargs

- *args et **kwargs permettent de passer de multiples arguments et des arguments nommés à une fonction.
- Utiliser *args et **kwargs, dans les paramètres d'une fonction, permettent de récupérer les arguments, respectivement sous forme de **tuple** et de **dictionnaire**, de réaliser un **packing**.
- À l'inverse, utiliser *args et **kwargs, comme argument d'une fonction, permettent de réaliser un **unpacking**.

```
def sum(*args):  
    r = 0  
    for x in args:  
        r += x  
    return r
```

```
print(sum(10, 20, 30, 40))
```

```
def sum(**kwargs):  
    r = 0  
    for x in kwargs.values():  
        r += x  
    return r
```

```
print(sum(a=10, b=20, c=30, d=40))
```

Commentaires

Commentaires

- Les commentaires permettent de rendre votre code plus intelligible ou de générer de la documentation.
- Les lignes de commentaires commencent par le caractères #.
- Un bloc de commentaires peut être écrit entre triple guillemets anglais :

"""

Mon commentaire

"""

Mon commentaire

Docstring

- Les docstring permettent d'ajouter des commentaires aux fonctions, notamment pour renseigner les arguments attendus et la valeur de retour.

```
"""
```

Une description

:param nombre1 le premier nombre

:param nombre2 le deuxième nombre

:return une liste avec nombre1 et nombre2

```
"""
```

```
def maFonction(nombre1, nombre2):  
    return [nombre1, nombre2]
```

```
valeur = maFonction(1, 2)
```

Condition if - else et match

Condition if else

- Si une condition est remplie, alors exécuter une action :

```
var1 = 1
if var1 == 3:
    print('OK')
else:
    print('NOT')
```

- Il est possible de combiner des conditions avec les opérateurs and et or :

```
var1 = True
var2 = False
if var1 and not var2:
    print('OK')
else:
    print('NOT')
```

Condition if else

- Plusieurs opérateurs de comparaison sont disponibles :
 - == égal à
 - != différent de
 - > strictement supérieur à
 - >= supérieur ou égal à
 - < strictement inférieur à
 - <= inférieur ou égal à
- Il est possible de tester plusieurs conditions avec elif :

```
var1 = True  
var2 = False  
if var1 and not var2:  
    print('OK')  
elif not var1 or var2:  
    print('Alright')  
else:  
    print('NOT')
```



Match

- L'instruction **match** confronte la valeur d'une expression à plusieurs filtres successifs donnés par les instructions **case**.
- Si aucun des filtres dans les **case** ne fonctionne, aucune des branches indentées sous les **case** n'est exécutée.
- Pour match n'importe quelle valeur, vous pouvez créer une dernière instruction **case** avec un **_**.

```
match secret:  
    case 'abc':  
        return "OK"  
    case 123:  
        return "NOK"  
    case _:  
        return "Default"
```

Exercice

- Demandez à l'utilisateur d'entrer 2 nombres et un opérateur parmi +, -, *, /.
- Procédez à l'opération arithmétique entre ces 2 nombres en fonction du symbole passé en argument.
- Si l'opération est incorrect, affichez un message d'erreur.
- Écrivez votre code dans une fonction et appelez votre fonction.
- Affichez le résultat final.

Solution

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

def calculate(nombre1, nombre2, operation):
    if operation == '+':
        return nombre1 + nombre2
    elif operation == '-':
        return nombre1 - nombre2
    elif operation == '*':
        return nombre1 * nombre2
    elif operation == '/':
        return nombre1 / nombre2

nombre1 = input('Entrez un premier nombre : ')
nombre2 = input('Entrez un deuxième nombre : ')
operation = input('Entrez une opération : ')
if operation not in ['+', '-', '*', '/']:
    print('Opérateur non supporté')
else:
    print(calculate(int(nombre1), int(nombre2), operation))
```

Entrez un premier nombre :

3

Entrez un deuxième nombre :

2

Entrez une opération :

*

6



Using a
scientific
calculator



Using
Python as a
scientific
calculator

Boucles

Boucles

- Les boucles s'utilisent pour répéter plusieurs fois l'exécution d'une partie du programme.
- Il existe 2 types de boucle:
 - Boucle bornée
 - Boucle non bornée

Boucle bornée

- Quand on sait combien de fois doit avoir lieu la répétition, utilisez une boucle **for**.

```
for i in [0, 1, 2, 3]:  
    print("i a pour valeur", i)
```

```
liste = [1, 2, 3]  
for i in range(2):  
    print("i a pour valeur", liste[i])
```

Boucle non bornée

- Si on ne connaît pas à l'avance le nombre de répétitions, choisissez une boucle **while**.

```
x = 1
while x < 10:
    print("x a pour valeur", x)
    x = x * 2
print("Fin")
```

Boucle **for** ou **while** ?

- En général, si on connaît avant de démarrer la boucle le nombre d'itérations à exécuter, on choisit une boucle **for**.
- Au contraire, si la décision d'arrêter la boucle ne peut se faire que par un test, on choisit une boucle **while**.
- Il est toujours possible de remplacer une boucle **for** par une boucle **while**

L'instruction break et continue

- L'instruction **break** permet de casser l'exécution d'une boucle. Elle fait sortir de la boucle et passer à l'instruction suivante.
- L'instruction **continue** permet de passer prématulement au tour de boucle suivant. Elle fait continuer sur la prochaine itération de la boucle.

```
while True:  
    n = int(input("donnez un entier > 0 : "))  
    if n > 0:  
        break  
  
for i in range(4):  
    if i < 2:  
        continue
```

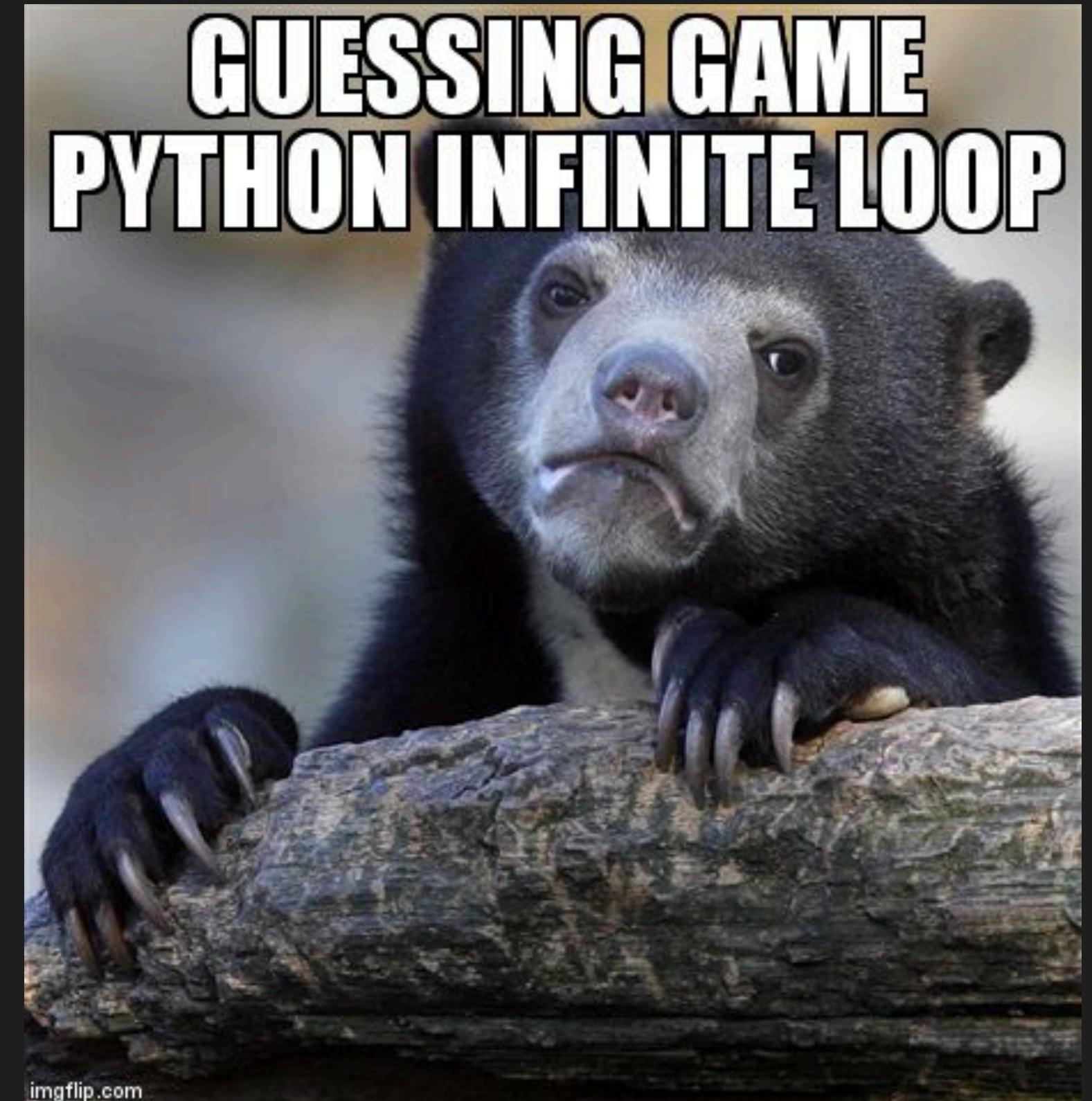
Exercice

- Créez un jeu où vous devez demander à l'utilisateur de deviner un mot secret.
- Vous devez définir le mot secret dans une variable de votre programme.
- Tant que l'utilisateur n'a pas deviné le mot, le programme doit continuer de demander à l'utilisateur le mot secret.

Solution

```
#!/usr/bin/env python  
  
secret = 'chien'  
guess = ''  
  
while guess != secret:  
    guess = input('Devinez le mot : ')  
print('Trouvé !')
```

```
Devinez le mot :  
chat  
Devinez le mot :  
chien  
Trouvé !
```



Exercice

- Créez une fonction permettant de calculer la puissance d'un nombre.
- Exemple : puissance(2, 3) est équivalent à calculer 2^3 et doit donc retourner 8.
- Vous ne devez pas utiliser l'opérateur exponent **.
- Affichez le résultat de la fonction.

Solution

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

def puissance(nombre, puissance):
    resultat = 1
    for i in range(puissance):
        resultat = resultat * nombre
    return resultat

print(puissance(2, 3))
```

Erreur et exceptions

Erreur et exceptions

- Vous pourriez rencontrer des erreurs de syntaxe, qui sont des erreurs d'analyse du code :

```
>>> while True print('Hello world')
File "<stdin>", line 1
    while True print('Hello world')
          ^
SyntaxError: invalid syntax
```

- Ou des erreurs durant l'exécution appelées exception :

```
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Gestion des exceptions

- Il est possible de prendre en charge les exceptions dans un programme à l'aide de l'instruction try - except :

```
while True:  
    try:  
        x = int(input('Entrez un nombre : '))  
        break  
    except ValueError:  
        print('Nombre incorrect')
```

- Les instructions dans la clause **try** sont exécutées en premier, si aucune exception n'est intervenue durant l'exécution, la clause **except** est sautée.
- Si une exception intervient durant l'exécution de la clause **try**, le reste de cette clause est sauté. Si le type d'exception levé correspond à un nom indiqué dans la clause **except**, la clause **except** correspondante est exécutée, puis le programme reprend après la clause **try**.
- Si aucune clause **except** n'existe pour le type d'exception levé, l'exception n'est pas gérée et l'exécution s'arrêtera avec un message d'erreur comme dans l'exemple ci-dessus.

Gestion des exceptions

- Une instruction **try** peut comporter plusieurs clauses **except** pour permettre la prise en charge de différentes exceptions. Une clause **except** peut également citer plusieurs exceptions :

```
try:  
    ...  
except (RuntimeError, TypeError, NameError):  
    pass
```

```
try:  
    ...  
except ValueError:  
    print("Mauvaise valeur")  
except TypeError:  
    print("Mauvais type")
```

Gestion des exceptions

- Une instruction **finally** sera toujours exécutée, qu'il y ait une exception survenue ou non :

```
try:  
    ...  
except ValueError:  
    print("Mauvaise valeur")  
finally:  
    print("Exécuté")
```

Gestion des exceptions

- Vous pouvez inspecter en détails une exception en spécifiant une variable à dans la clause **except** :

```
try:  
    ...  
except ValueError as e:  
    print(e)
```

Déclencher une exception

- Vous pouvez décider de déclencher de vous-même une exception spécifique dans votre programme à l'aide de l'instruction **raise** :

```
raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere
```

Exercice

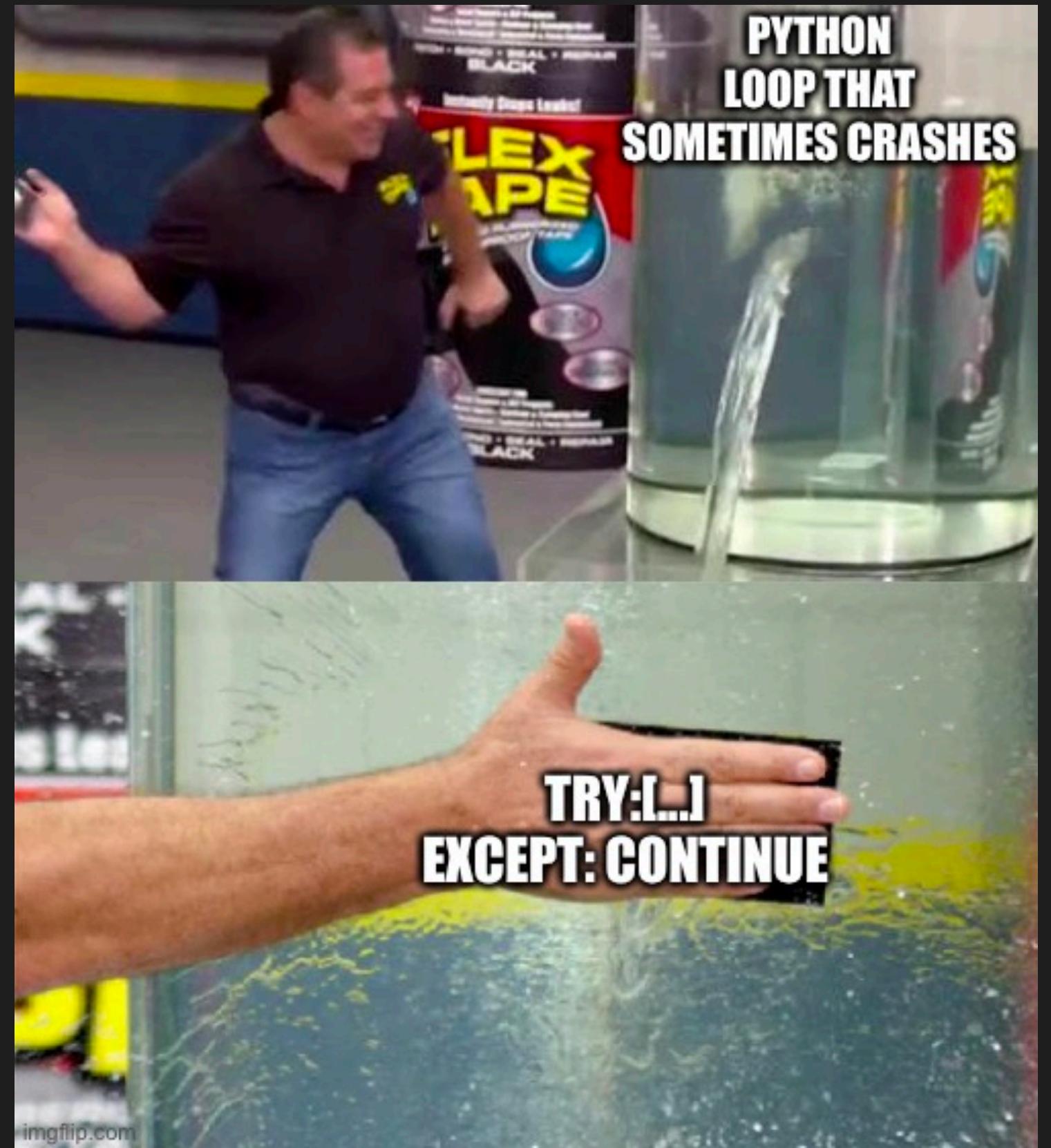
- Demandez à l'utilisateur d'entrer 2 nombres et un opérateur parmi +, -, *, /.
- Procédez à l'opération arithmétique entre ces 2 nombres en fonction du symbole passé en argument.
- Si l'opération est incorrect ou interdite, levez une exception et gérez-là avec une instruction try - except.
- Écrivez votre code dans une fonction et appelez votre fonction.
- Affichez le résultat final.

Solution

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

def calculate(nombre1, nombre2, operation):
    if operation == '+':
        return nombre1 + nombre2
    elif operation == '-':
        return nombre1 - nombre2
    elif operation == '*':
        return nombre1 * nombre2
    elif operation == '/':
        return nombre1 / nombre2
    raise ValueError('Opérateur non supporté')

nombre1 = input('Entrez un premier nombre : ')
nombre2 = input('Entrez un deuxième nombre : ')
operation = input('Entrez une opération : ')
try:
    print(calculate(int(nombre1), int(nombre2), operation))
except (ValueError, ZeroDivisionError) as e:
    print(e)
```

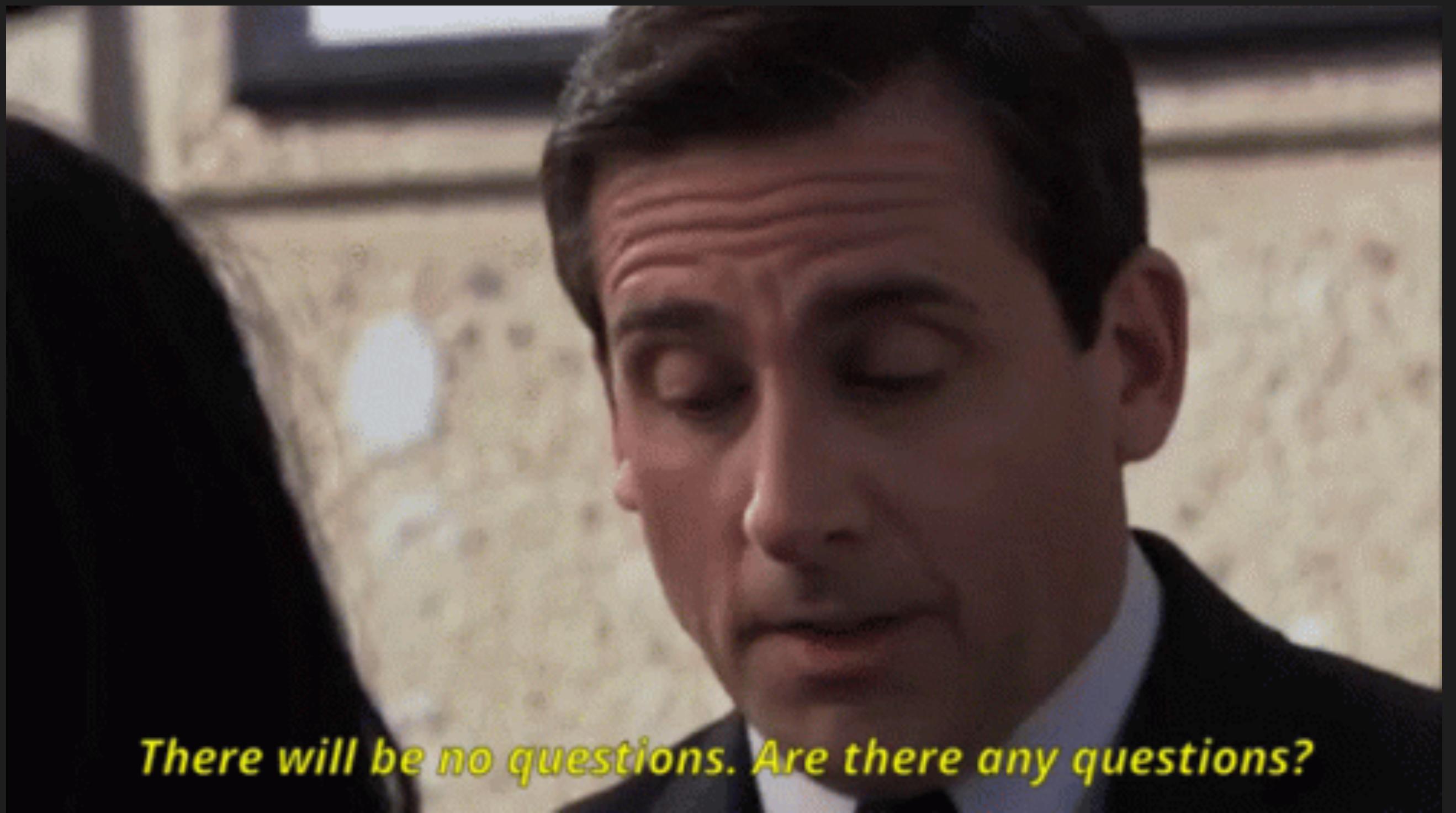


QCM 2



<https://app.wooclap.com/UHXYAD>

Questions ?



There will be no questions. Are there any questions?

Lire des fichiers

Ouvrir un fichier

- Python permet d'ouvrir des fichiers extérieurs à votre programme. Le plus souvent, ce seront des fichiers CSV, JSON, HTML...
- Pour cela, utilisez la fonction **open** :

```
open('monFichier.txt', 'r') # mode lecture seul  
open('monFichier.txt', 'w') # mode écriture (depuis le début du fichier)  
open('monFichier.txt', 'a') # mode écriture (depuis la fin du fichier)  
open('monFichier.txt', 'r+w') # mode lecture+écriture (depuis la début du fichier)
```

Lire un fichier

- Une fois ouvert, vous pouvez commencer à lire le contenu du fichier.
- Pour cela, utilisez la fonction **readlines** qui retourne toutes les lignes du fichiers dans une liste :

```
fichier = open('monFichier.txt', 'r')
for ligne in fichier.readlines():
    print(ligne)
```

Écrire dans un fichier

- Pour écrire dans un fichier, utilisez l'instruction **write**.
- L'instruction **write** ne revient pas à la ligne automatiquement, il faut ajouter manuellement le caractère **\n** entre guillemets doubles.

```
fichier = open('monFichier.txt', 'w')  
fichier.write("\nHello")  
fichier.write('\nWorld')  
fichier.close()
```

Fermer un fichier

- Une bonne pratique est de toujours fermer le fichier après utilisation.
- Si vous ne fermez pas un fichier, vous gaspillez des ressources et augmentez vos chances de perdre vos données...
- Soyez intelligent et utilisez la fonction **close** :

```
fichier = open('monFichier.txt', 'r')
for ligne in fichier.readlines():
    print(ligne)
fichier.close()
```

La version plus intelligente...

- Si vous manipulez des flux (fichiers, requêtes de base de données, téléchargement de fichiers...), il est plus intelligent d'utiliser l'instruction **with**.
- L'avantage de l'instruction **with** est que votre code sera plus lisible et gèrera automatiquement la fermeture du flux, même en cas d'exception.

La version plus intelligente...

```
# Nul, le fichier ne se fermera pas en cas d'exception
f = open('monFichier.txt', 'w')
f.write("Hello Python!\n")
f.close()

# Mieux, le fichier se fermera, même avec une exception
f = open("monFichier.txt", "w")
try:
    f.write("Hello Python!\n")
finally:
    f.close()

# Meilleur, le code est plus lisible
with open("monFichier.txt", "w") as f:
    f.write("Hello Python!\n")
```

Modules

Modules

- Python est un langage à la fois très puissant, modulable et évolutif, grâce à l'utilisation des modules.
- Un module est un fichier python que l'on importe dans un autre fichier ou script.
- Les modules permettent la séparation et donc une meilleure organisation du code. Une bonne pratique est de découper le code en différentes parties cohérentes.

Modules

- Il existe 3 grandes catégories de module :
 - Les modules standards qui sont intégrés automatiquement par Python.
 - Les modules développés par d'autres développeurs qu'on va pouvoir ré-utiliser.
 - Les modules qu'on va développer nous-mêmes
- Quelque soit la catégorie de module, la procédure pour utiliser un module reste la même.

Modules

- Il existe 3 grandes catégories de module :
 - Les modules standards qui sont intégrés automatiquement par Python.
<https://docs.python.org/3/py-modindex.html>
 - Les modules développés par d'autres développeurs qu'on va pouvoir ré-utiliser.
 - Les modules qu'on va développer nous-mêmes
- Quelque soit la catégorie de module, la procédure pour utiliser un module reste la même.

Module extérieur

- Contrairement aux modules intégrés à Python, les modules extérieurs doivent être téléchargés avant d'être utilisés.

- Vous pouvez utiliser le gestionnaire **pip** pour télécharger un module :

```
pip install requests
```

- Ensuite, il suffit d'importer le module comme ceci :

```
import requests  
r = requests.get('https://www.google.fr')
```

Exemple de module développé nous-même

- Considérons le fichier python suivant nommé fonction.py :

```
def fonction1(n):  
    print(n)  
  
def fonction2(n):  
    print(n * n)
```

```
import function  
function.function1(1)  
function.function2(2)
```

- Nous pouvons importer ce fichier dans un autre fichier avec l'instruction **import** et ré-utiliser nos 2 fonctions:

```
import fonction  
  
fonction.fonction1(1)  
fonction.fonction2(2)
```

Règle d'import

- Lorsqu'on importe un module, Python le recherche dans différents répertoires selon un ordre précis.
- En premier, le répertoire courant.
- Si le module est introuvable, Python recherche ensuite chaque répertoire listé dans la variable shell PYTHONPATH.
- Si tout échoue, Python vérifie le chemin par défaut. Sous UNIX, ce chemin par défaut est /usr/local/lib/python/.

Utiliser un alias

- Vous pouvez utiliser le mot clé **as** pour créer un alias de nom et obtenir des scripts plus courts et plus clairs dans le cas où le nom du module est très long.

```
import fonction as f
```

```
f.fonction1(1)  
f.fonction2(2)
```

Importer uniquement certaines fonctions

- Une bonne pratique consiste à n'importer que les fonctions dont vous avez besoin. Ceci permet d'avoir une vision claire de ce que l'on importe.

```
from fonction import fonction1, fonction2
```

```
fonction1(1)
```

```
fonction2(2)
```

Exercice

- Créez un programme renvoyant la date et l'heure actuelle (format YYYY-MM-DD HH:MM:SS) d'un fuseau horaire (exemple ‘Africa/Abidjan’), que l'utilisateur devra entrer.
- Utilisez cette API <https://worldtimeapi.org/api>.
- Vous devrez vérifier que le fuseau horaire existe dans WorldTimeApi, déclencher une exception si ce n'est pas le cas.
- Vous pouvez utiliser n'importe quel module pour interroger l'API et manipuler le résultat.

Solution

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

from datetime import datetime
from json import loads
from requests import get

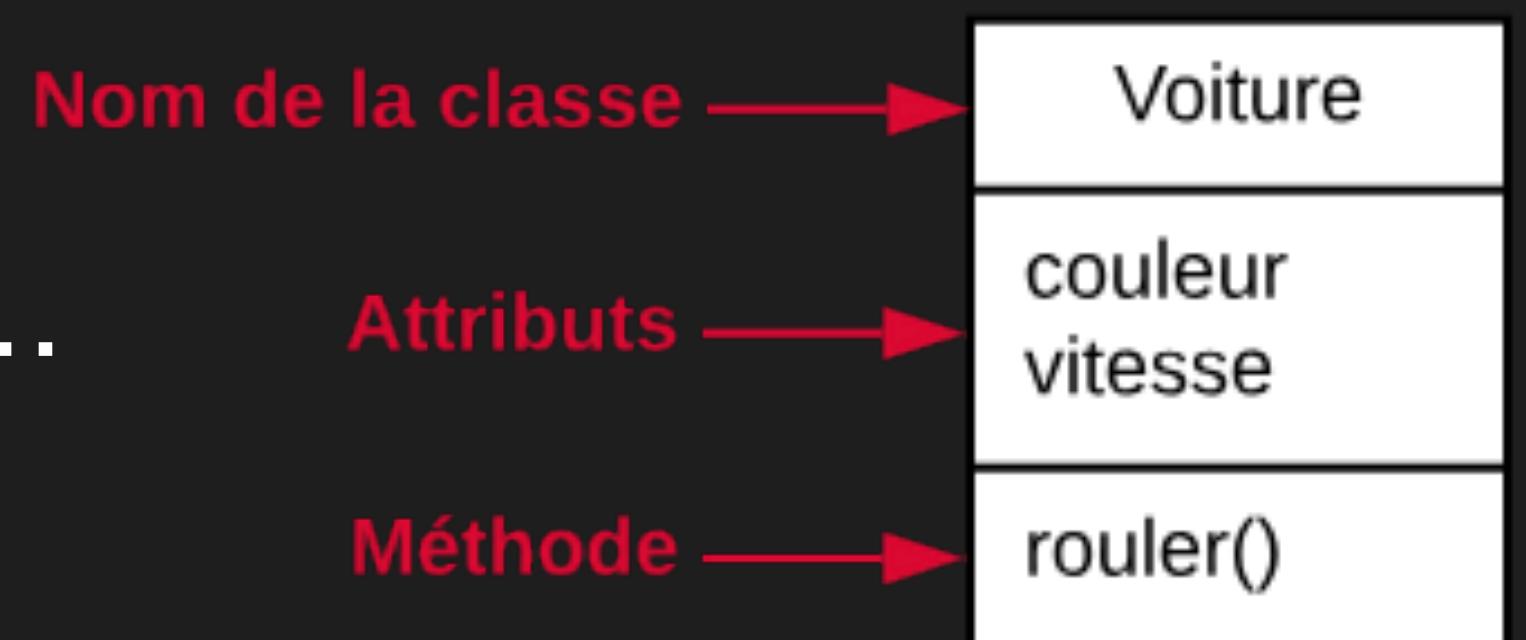
timezone = input("Entrez une capitale pour connaître l'heure local : ")
response = get('https://worldtimeapi.org/api/timezone/' + timezone)
if response.status_code != 200:
    raise ValueError('Fuseau horaire inconnu')
response_json = loads(response.text)
print(datetime.fromtimestamp(response_json['unixtime'] + response_json['raw_offset']))
```

Entrez une capitale pour connaître l'heure local : Asia/Taipei
2023-10-24 05:56:11

Classe et objet

Classe

- Vous avez vu que la plupart des données peut être stockée dans des types standards (str, int, list, ...). Y-a-t-il d'autres types de représentation pour des projets plus complexes ?
- Une classe va vous permettre de créer des “types” personnalisés à vos besoins, plus communément appelés structures de données.
- Une classe est constitué d'attributs et de méthodes.
- Exemple avec une voiture :
 - Attributs : Vitesse, couleur, jantes, moteur, taille, ...
 - Méthodes : rouler(), freiner(), changerJantes(), ...



Classe

- Créons une classe Voiture comme exemple :

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

class Voiture:
    def __init__(self, marque):
        self.marque = marque
        self.peinture = 'rouge'
        self.roue = 0

    def rouler(self):
        if (self.roue != 4):
            print('Il manque les roues..')
            return
        print('On accélère !')
```

<- Constructeur où les variables d'instances sont définis (valeurs différentes entre chaque instance de l'objet Voiture), exécuté chaque fois que l'objet est créé. **self** est un indicateur que la variable ou la méthode est propre à l'instance.

<- Méthodes

Objet

- Un objet est une instance de classe.
- Exemple : `maVoiture = Voiture()`
- L'objet `maVoiture` est maintenant une voiture dotée des caractéristiques (attributs) et fonctionnalités (méthodes) d'une voiture.
- Pour modifier les caractéristiques (attributs) de notre voiture :
`maVoiture.moteur = '2CV'`
- Pour utiliser les fonctionnalités (méthodes) de notre voiture : `maVoiture.rouler()`

Objet

- Créons un objet maVoiture, instance de la classe Voiture :

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

```
class Voiture:  
    def __init__(self, marque):  
        self.marque = marque  
        self.peinture = 'rouge'  
        self.roue = 0  
  
    def rouler(self):  
        if (self.roue != 4):  
            print('Il manque les roues..')  
            return  
        print('On accélère !')  
  
voiture = Voiture('Mercedes')  
voiture.rouler()  
voiture.roue = 4  
voiture.rouler()
```

Il manque les roues..
On accélère !

Héritage simple

- L'héritage permet de réutiliser du code existant et d'éviter de réinventer la roue...

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

```
class Voiture:  
    def __init__(self, marque):  
        self.marque = marque  
        self.peinture = 'rouge'  
        self.roue = 0  
  
    def rouler(self):  
        if (self.roue != 4):  
            print('Il manque les roues..')  
            return  
  
        print('On accélère !')
```

```
voiture = Voiture('Mercedes')  
voiture.rouler()
```

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

```
class Mercedes(Voiture):  
    def __init__(self, marque):  
        super().__init__('Mercedes')  
        self.roue = 4  
  
    voiture = Mercedes()  
    voiture.rouler()
```

La fonction **super** est un raccourci désignant la classe mère Voiture. Il est possible de redéfinir n'importe quel attribut de la classe mère depuis la classe fille (Mercedes)

Héritage multiple

- Même principe que l'héritage simple mais avec plusieurs classes mères.

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

```
class Voiture:  
    def __init__(self, peinture):  
        self.roue = 4  
        self.peinture = peinture  
  
    def rouler(self):  
        print('On accélère !')
```

```
class Mercedes:  
    marque = 'Mercedes'
```

```
class Amg(Voiture, Mercedes):  
    def __init__(self, peinture):  
        Voiture.__init__(self, peinture)  
        Mercedes.__init__(self)
```

```
a = Amg('Rouge')  
print(a.marque)  
print(a.peinture)  
print(a.roue)  
a.rouler()
```

Mercedes
Rouge
4

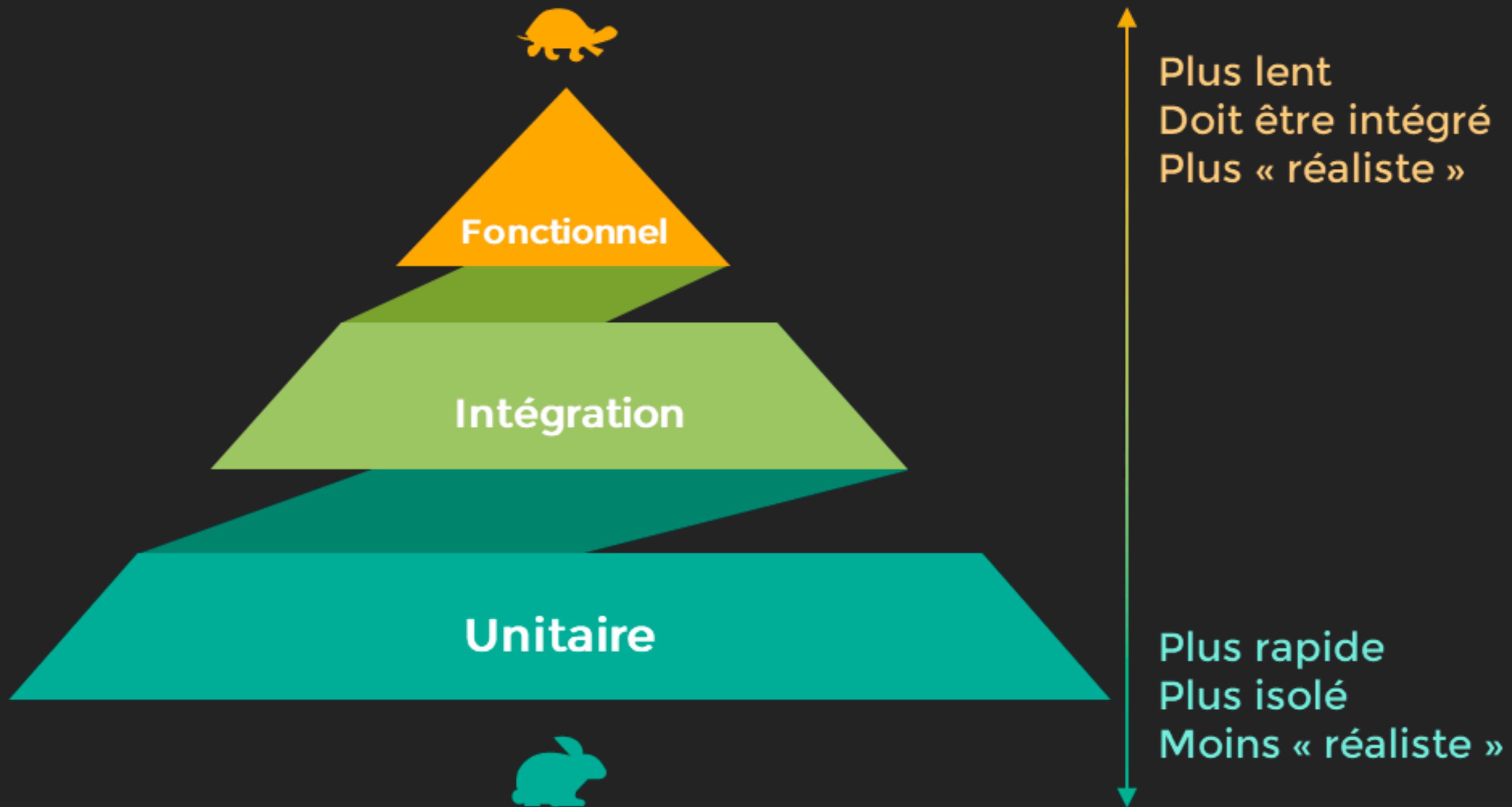
On accélère !

Tests

Tests

- Le but des tests est de confirmer que les différentes parties du programme correspondent aux spécifications techniques.
- Les tests automatisés permettent d'éviter de re-tester manuellement l'intégralité du projet à chaque fois qu'une modification a lieu.
- L'absence de tests entraîne un mauvais contrôle qualité de votre programme, ce qui aura forcément des répercussions sur l'expérience utilisateur.
- En conclusion, les tests automatisés sont quasiment obligatoire.

Pyramide de test



- 3 types de test :
- Unitaire
 - Intégration
 - Fonctionnel

Tests unitaires

- Les tests unitaires permettent de tester des unités de code, généralement des fonctions, des classes...
- Quand une application réussit tous ses tests unitaires, vous avez au moins la garantie que ses fonctionnalités secondaires sont correctes.
- Les tests unitaires sont la base de tout test, de ce fait, une règle importante est qu'un test unitaire ne peut pas dépendre de fonctionnalités extérieures à l'unité qui est testé. Exemple : Un test faisant appel à une base de données ou une API n'est pas un test unitaire.
- Étant donné qu'un test unitaire ne dépend d'aucun élément extérieur, il est le type de test le plus rapide à exécuter. Pratique pour déceler des bugs rapidement, tester des scénarios...

Tests d'intégration

- Les tests d'intégration ont pour objectif de vérifier qu'une nouvelle fonctionnalité ne va pas poser de problème lors de son intégration au sein de l'application.
- Les tests d'intégration vont vérifier que toutes ces fonctionnalités arrivent à travailler ensemble.
- Contrairement aux tests unitaires, les tests d'intégrations peuvent dépendre d'autres fonctionnalités extérieures. Ils sont donc plus lents à exécuter.

Tests fonctionnels

- Les tests fonctionnels vont vérifier qu'une fonctionnalité, dans son ensemble, marche comme nous le souhaitons du point de vue de l'utilisateur.
- Les tests fonctionnels testent un parcours utilisateur. Ils sont donc les plus lents à exécuter.

Écrire un test

- La librairie de tests la plus utilisée se nomme `unittest`. Toutes les méthodes de tests doivent commencer par le préfixe `test_`.

- Voici un exemple de test :

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

if __name__ == '__main__':
    unittest.main()
...
-----
Ran 3 tests in 0.000s
```

- Un scénario de test est créé comme classe-fille de `unittest.TestCase`.
- Les trois tests individuels sont définis par des méthodes dont les noms commencent par les lettres `test`. Cette convention de nommage signale au lanceur de tests quelles méthodes sont des tests.
- Chaque test est un appel à une fonction `assert`.
- Vous avez `assertEqual()` pour vérifier un résultat attendu, `assertTrue()` ou `assertFalse()` pour vérifier une condition, ou `assertRaises()` pour vérifier qu'une exception particulière est levée.

Écrire un test

- Pour initialiser des variables au début de chaque test, utilisez la fonction **setup**.
- Pour réinitialiser des variables à la fin de chaque test, utilisez la fonction **teardown**.

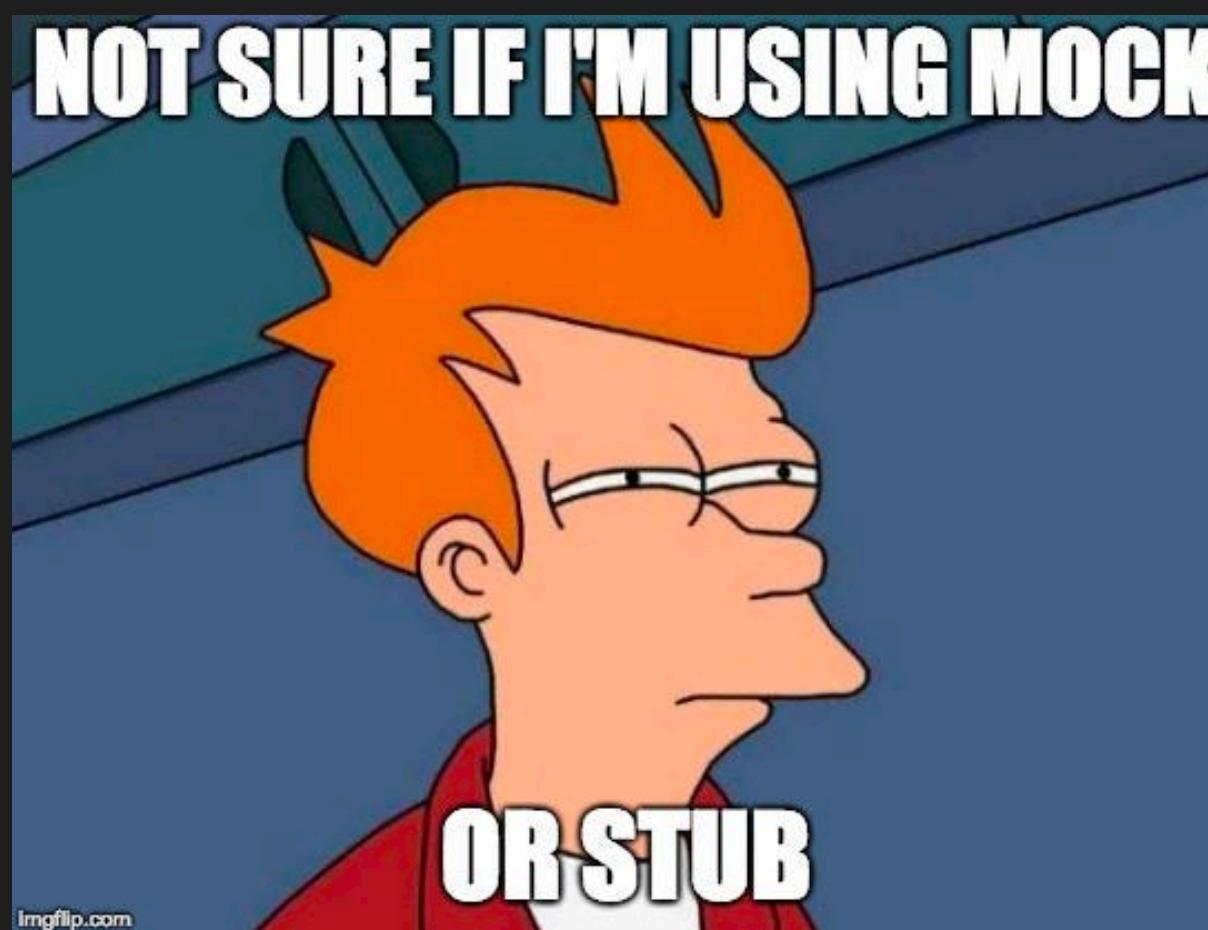
```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()
```

Mock vs Stub

- Pour les fonctions dépendant d'autres fonctionnalités extérieures, vous pouvez utiliser des stubs ou mocks afin de réaliser des tests unitaires.
- Un **stub** permet de simuler la valeur de retour de fonctions. Utile pour forcer un scénario ou ne pas bloquer votre test.
- Un **mock** permet de simuler la valeur de retour et de contrôler que l'appel à la fonction a été effectué.



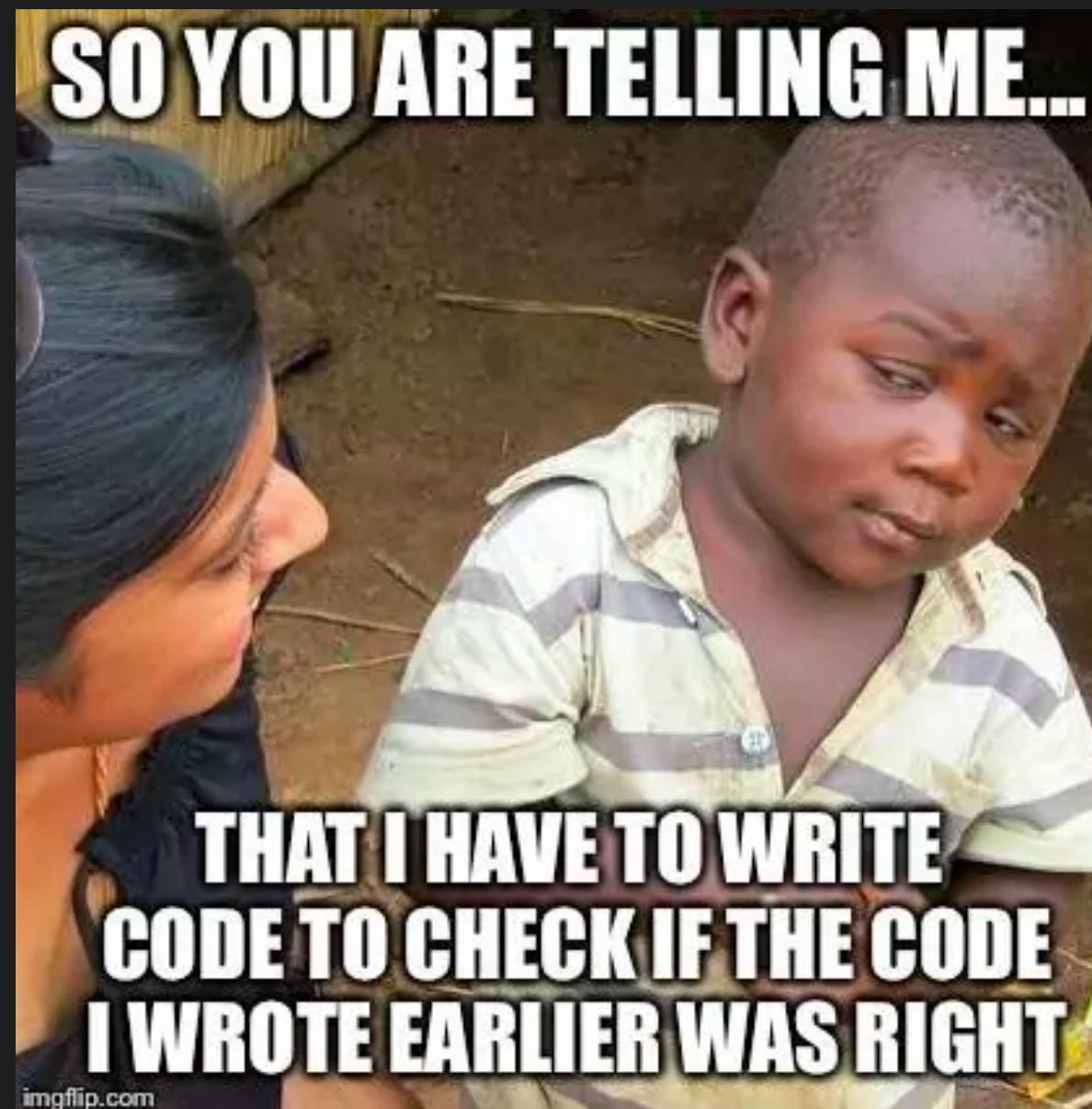
Interface en ligne de commande

- Vous pouvez utiliser unittest en ligne de commande pour tester un fichier, un module, un dossier entier de test... :

```
python -m unittest test_module1 test_module2
```

Quoi d'autres ?

- Il y a un nombre incroyable de ressources concernant les tests sur Python. Beaucoup trop pour être abordées dans ce cours pour débutant...
- Si vous souhaitez creuser le sujet, la documentation sera votre meilleure amie : <https://docs.python.org/3/library/unittest.html>

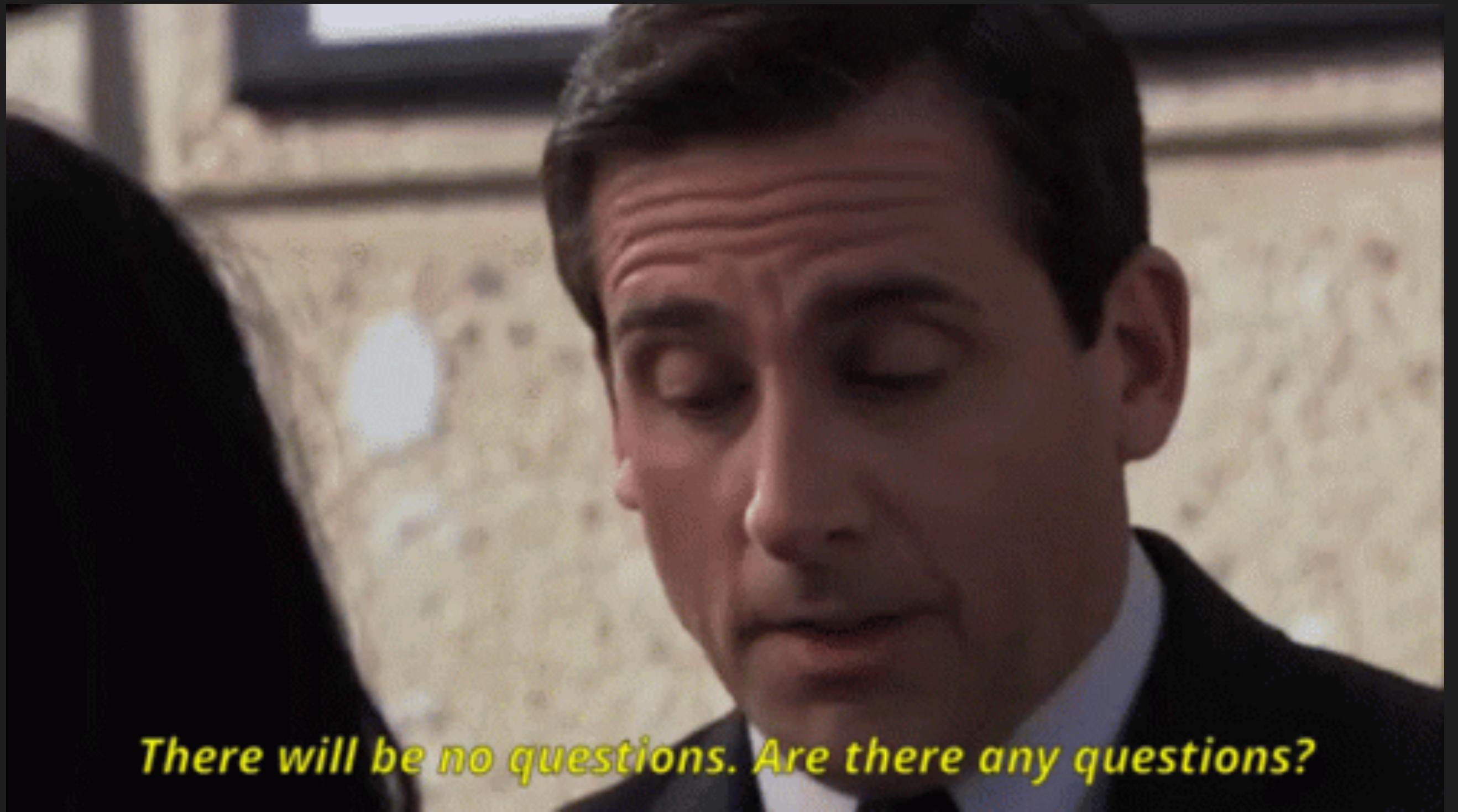


QCM 2



<https://app.wooclap.com/PDQUET>

Questions ?



There will be no questions. Are there any questions?

Projets

Projet 1 : QCM

- Créez un QCM de 10 questions avec 3 réponses possibles.
- Utilisez la notion de classes et objets pour construire les questions et les réponses.
- Séparer votre code en plusieurs fichiers et importez-les en tant que module.
- Les questions et les réponses doivent être montrées à l'utilisateur dans un ordre aléatoire.
- Demandez à l'utilisateur de choisir une des 3 réponses avec a, b ou c. Ne doit pas être sensible à la casse.
- Vous devrez afficher le score final ainsi que le corrigé à la fin du test.
- À rendre sur GitLab individuellement avec un README.md et un test unitaire.
- Pas de rapport écrit. Vous serez évalué sur la qualité de votre programme, sur le respect des consignes et s'il fonctionne comme attendu.

Projet 2 : Générateur et testeur de mot de passe

- Créez un programme embarquant des outils sécurité.
- Implémentez un testeur de mot de passe testant la force d'un mot de passe basé sur l'entropie respectant les critères de l'ANSSI : <https://www.ssi.gouv.fr/administration/precautions-elementaires/calculer-la-force-d-un-mot-de-passe/>
- Implémentez un générateur de mot de passe aléatoire en fonction de critères que l'utilisateur pourra sélectionner : nombre de minuscules, nombre de majuscules, nombre de chiffres, nombre de caractères spéciaux. Affichez le mot de passe généré avec son entropie et sa force.
- Implémentez un générateur de passphrase en vous basant sur la méthode de dés de l'EFF : <https://www.eff.org/fr/dice>.
- Utiliser les classes et objets. Séparer votre code en plusieurs fichiers et importez-les en tant que module.
- À rendre sur GitLab individuellement avec un README.md et un test unitaire.
- Pas de rapport écrit. Vous serez évalué sur la qualité de votre programme, sur le respect des consignes et s'il fonctionne comme attendu.