pythonCourse.py

workshop.css

```
Programming ‘python’ {
  [Scripting Workshop]

     < Slides du cours >

}
```

pythonCourse.py

workshop.css

Table Of ‘Contents’ {

01 Les bases du python

< Comprendre et maîtriser
la syntaxe du langage >

02 Les Types complexes

< >

03 Réalisation de
scripts complexe

< >

}

Qu’est-ce que ‘python’ ? {

- [ ] **Langage cross plateforme**
  Fonctionne sur quasiment tous les OS
- [ ] **Langage haut niveau**
  Demande peu de connaissance hardware/os pour comprendre son fonctionnement
- [ ] **Langage interprété**
  Pas besoin d’être compilé
- [ ] **Langage orienté objet**
  On peut créer des entités représentatives du monde réel

}

De quoi avez-vous besoin ? {

Les ressources essentielles dont vous aurez besoin

* Python3
  * https://www.python.org/
* Un bon IDE
  * exemple VSCode : https://code.visualstudio.com/
* Des librairies et modules
  * Pip3
  * Virtualenv
  * Pandas, numpy etc.

}

Let's install! {

Place au terminal

* Sur Linux

```
$ python3 -V #si une version s'affiche, python est installé
$ sudo apt update && sudo apt install
```

* Sur MacOS

```
~> brew install python3
```

* Sur Windows
  - Aller sur le site de python.org et télécharger la dernière version

}

1
2
3
4
5
6
7
8
9
10
11
12
13
14

01

Les bases du Python

< Comprendre et maîtriser
la syntaxe du langage >

pythonCourse.py

workshop.css

Premier { Code;

Réalisez un ‘Hello World!’, dans la console Python, puis dans un script

```
>>> print(‘’Hello World!’’)
```

}

Appel de l'interpréteur {

La console représente vite une limite, dès que l'on travail à partir d'un fichier on parle de script

* ## Appel direct de l'interpréteur

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

* ## Appel direct du script

  1. Précisez au shell la localisation de l'interpréteur Python en indiquant dans la première ligne du script : (exemple pour linux) `#! /usr/bin/env python3`

  2. Donner une permission d'exécution avec : `$ chmod +x test.py`

  3. Puis exécuter avec : `$ ./test.py`

}

Commenter son code ! {

* Dans un script, tout ce qui suit le caractère # est ignoré par Python jusqu'à la fin de la ligne et est considéré comme un commentaire. Il est également possible d'écrire un gros block de commentaire avec

```
'''
 Ceci est un commentaire !
'''
```

* Les commentaires sont indispensables pour que vous puissiez annoter votre code. Il faut absolument les utiliser pour décrire les fonctions ou parties principales. Votre code doit être rapidement compréhensible par d'autre !

}

L'indentation en Python {

* En python, l'indentation est essentielle. En effet, une mauvaise indentation créera des bugs ou un mauvais comportement de votre code !

* Pratiquement, l'indentation en Python doit être homogène (soit des espaces, soit des tabulations, mais pas un mélange des deux). Une indentation avec 4 espaces est le style d'indentation le plus traditionnel et celui qui permet une bonne lisibilité du code.

}

Exemple de code

```python
import os
import sys
import subprocess

# Simple SSL/TLS python3 script vulnerabilities test

rc = subprocess.call(['which', 'testssl'], stdout=subprocess.PIPE)
if rc == 0:
    print ("[+] testssl.sh is installed on this machine")
else:
    print ("[-] testssl.sh is missing in path.")
    sys.exit()
# sys.argv[1] is the first argument given after the python script
# Check if argument was give, if not, print usage

if len(sys.argv) == 1:
    print ("[*] Simple SSL/TLS vulnerabilities tests Script")
    print ("[*] Usage       : " + sys.argv[0] + " <IP adresses file path> ")
    print ("[*] Note: IP addresses file must use the following format @ip:port ")
    sys.exit()

```

Les variables {

Une variable est une zone de la mémoire dans laquelle une valeur est stockée. Aux yeux du programmeur, cette variable est définie par un nom, alors que pour l'ordinateur, il s'agit en fait d'une adresse

* En Python, la déclaration d'une variable et son initialisation (c'est-à-dire la première valeur que l'on va stocker dedans) se font en même temps.

```
>>> x = 7
>>> x
7
```

* Que c'est il passé dans cet exemple ?

}

Les variables {

```
>>> x = 7
>>> x
7
```

* Python a deviné que la variable était un entier. On dit que Python est un langage au typage dynamique.

* Python a alloué l'espace en mémoire pour y accueillir un entier et a fait en sorte qu'on puisse retrouver la variable sous le nom x

* Python a assigné la valeur 2 à la variable x.

}

Les variables {

- Dans certains autres langages, il faut coder ces différentes étapes une par une (en C par exemple). Python étant un langage dit de haut niveau, la simple instruction x = 7 a suffi à réaliser les 3 étapes en une fois !

- Dernière chose, l'opérateur d'affectation = s'utilise dans un certain sens :

  - Si on a x = y - 3, l'opération y - 3 est d'abord évaluée et ensuite le résultat de cette opération est affecté à la variable x.

}

Les différents types de variables

{

* Le type d'une variable correspond à sa nature. Les trois types principaux sont les entiers (integer ou int), les réels (float) et les chaînes de caractères (string ou str). Il en existe plein d'autres !

* Python reconnaît certains types de variable automatiquement. Par contre, pour une chaîne de caractères, il faut l'entourer de guillemets (simples, doubles etc.) pour indiquer à Python le début et la fin de la chaîne.

}

Les noms de variables {

* Le nom des variable en Python peut-être constitué de lettres minuscules (a à z), de lettres majuscules (A à Z), de nombres (0 à 9) ou du caractère souligné (_).

* Un nom de variable ne doit pas débuter ni par un chiffre, ni par _ et ne peut pas contenir de caractère accentué. Il faut absolument éviter d'utiliser un mot "réservé" par Python comme nom de variable (par exemple : print, range, for, from, etc.).

* Python est sensible à la casse ! Donc les variables TesT, test ou TEST sont différentes. Enfin, vous ne pouvez pas utilisez d'espace dans un nom de variable.

}

Les opérateurs {

* +, -, *, **, /, et % sont des opérateurs. Ils permettent de faire des opérations sur les variables. (**) est l'opérateur de puissance. (%) le modulo renvoi le reste d'une division Euclidienne.

* Pour les chaînes de caractères, deux opérations sont possibles, l'addition et la multiplication :

    - L'opérateur d'addition + permet de concaténer (assembler) deux chaînes de caractères et l'opérateur de multiplication * permet de dupliquer plusieurs fois une chaîne.

}

Fonction type {

Pour obtenir le type d'une variable, utilisez la fonction type()

```
>>>x=2
>>> type(x)
<class 'int'>
```

Conversion de type

On est souvent amené à convertir les types avec les fonctions int(), float() et str().

```
>>> i=3
>>> str(i)
'3'
```

}

Fonction d’affichage {

Pour afficher du texte ou le contenu d’une variable utiliser la fonction print()

```
print(‘’Résultat de x :’’, x)
```

Fonction d’entrée

Grâce à la fonction input(), on peut demander une entrée à l’utilisateur puis affecter la valeur à une variable par exemple :

```
print(‘Quel âge as-tu ?’)
x = input()
print(‘Tu as {} ans !’.format(x))
```

}

Les listes {

* Une liste est une structure de données qui contient une série de valeurs. Python autorise la construction de liste contenant des valeurs de type différent (par exemple entier et chaîne de caractères), ce qui leur confère une grande flexibilité

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> tailles = [5, 2.5, 1.75, 0.15]
>>> mixte = ['girafe', 5, 'souris', 0.15]
```

* Lorsque l'on affiche une liste, Python la restitue telle qu'elle a été saisie.

```
>>> tailles
[5, 2.5, 1.75, 0.15]
```

}

Les listes {

* Un des gros avantages d'une liste est que vous pouvez appeler ses éléments par leur position. Ce numéro est appelé indice (ou index) de la liste.

```
liste : ['girafe', 'tigre', 'singe', 'souris']
indice:     0         1        2        3
```

/!\ Soyez très attentifs au fait que les indices d'une liste de n éléments commence à 0 et se termine à n-1. /!\

}

Opérations sur les listes {

* Tout comme les chaînes de caractères, les listes supportent l'opérateur + de concaténation, ainsi que l'opérateur * pour la duplication.

```
>>> ani1 = ['girafe','tigre']
>>> ani2 = ['singe','souris']
>>> ani1 + ani2
['girafe', 'tigre', 'singe', 'souris']
>>> ani1 * 3
['girafe', 'tigre', 'girafe', 'tigre', 'girafe', 'tigre']
```

}

Opérations sur les listes {

* L'opérateur + est très pratique pour concaténer deux listes. Vous pouvez aussi utiliser la fonction append() lorsque vous souhaitez ajouter un seul élément à la fin d'une liste.

```
>>> a = [] # liste vide
>>> a = a + [15]
>>>a
[15]
```

* Ou avec la fonction append() qui sera préférée pour ce genre d'opération

```
>>> a.append(-5)
>>> a
[15, -5]
```

}

Opérations sur les listes {

* La liste peut également être indexée avec des nombres négatifs selon le modèle suivant :

```
liste :          ['girafe', 'tigre', 'singe', 'souris']
indice positif :    0         1        2         3
indice négatif :   -4        -3       -2        -1
```

* Les indices négatifs reviennent à compter à partir de la fin. Leur principal avantage est que vous pouvez accéder au dernier élément d'une liste à l'aide de l'indice -1 sans pour autant connaître la longueur decette liste.

}

Fonction len() {

* L'instruction len() vous permet de connaître la longueur d'une liste, c'est-à-dire le nombre d'éléments que contient la liste. Voici un exemple d'utilisation :

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> len(animaux)
4
```

}

Time to practice {

Lancez votre IDE et réalisez les petits programmes suivants :

#1

* Créez un script permettant de créer une fiche client (Nom, prénom, téléphone, âge, etc.). Les variables sont remplies grâce aux input utilisateur.

#2

* Créez un script permettant de réaliser une commande au bar. Quel boissons veux l'utilisateur parmi une liste de choix possible (café, bière, etc.).

}

Boucles et comparaisons

Retour sur l’algorithmie

Boucles for {

* En programmation, on est souvent amené à répéter plusieurs fois une instruction. Incontournables à tout langage de programmation, les boucles vont nous aider à réaliser cette tâche de manière compacte. Imaginez par exemple que vous souhaitiez afficher les éléments d'une liste les uns après les autres.

```
animaux = ['girafe','tigre','singe','souris']
print(animaux[0])
print(animaux[1])
print(animaux[2])
print(animaux[3])
```

}

Boucles for {

* Si votre liste ne contient que quatre éléments, c'est encore faisable mais imaginez qu'elle en contienne 100 ou plus ! Bien développer c'est avant tout être feignant :

```
>>> animaux = ['girafe','tigre','singe','souris']
>>> for animal in animaux:
...  print(animal)
```

* La variable animal est appelée variable **d'itération**, elle prend successivement les différentes valeurs de la liste animaux à chaque itération de la boucle.

}

Boucles for {

```
for i in maListe :
    print(i)
```

* Le signe deux-points : à la fin de la ligne for. Cela signifie que la boucle for attend un bloc d'instructions.

* On appelle ce bloc d'instructions le corps de la boucle. Comment indique-t-on à Python où ce bloc commence et se termine ? Cela est signalé uniquement par l'indentation,

}

Fonction range() {

```
for i in range(3) :
   print(i)
…
0
1
2
```

* Python possède la fonction range(). Elle est utile pour générer des nombres entiers compris dans un intervalle.

* Pour Python, il s'agit d'un nouveau type, par exemple dans x = range(3) la variable x est de type *range* (tout comme on avait les types *int*, *float*, *str* ou *list*)

}

Boucles for - itération sur les indices {

* On peut utiliser un range et itérer sur les éléments d'une liste par ses indices

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> for animal in animaux:
...     print(animal)
...
girafe
tigre
singe
souris
```

}

Boucles for - itération sur les indices {

* Python possède également la fonction enumerate() qui vous permet d'itérer sur les indices et les éléments eux-mêmes.

```
>>> animaux = ['girafe', 'tigre', 'singe', 'souris']
>>> for i, animal in enumerate(animaux):
...     print("L'animal {} est un(e) {}".format(i, animal))
...
L'animal 0 est un(e) girafe
L'animal 1 est un(e) tigre
L'animal 2 est un(e) singe
L'animal 3 est un(e) souris
```

}

Comparaisons {

Python est capable d'effectuer toute une série de comparaisons entre le contenu de deux variables, telles que :

Syntaxe Python	Signification
==	égal à
!=	différent de
>	supérieur à
>=	supérieur ou égal à
<	inférieur à
<=	inférieur ou égal à

}

Comparaisons {

* Faites bien attention à ne pas confondre l'opérateur d'affectation = qui affecte une valeur à une variable et l'opérateur de comparaison == qui compare les valeurs de deux variables.

```
>>> animal = "tigre"
>>> animal == "tig"
False
```

}

While {

* Une autre alternative à l'instruction for couramment utilisée en informatique est la boucle while. Le principe est simple. Une série d'instructions est exécutée tant qu'une condition est vraie. Par exemple :

```
>>> i=1
>>> while i ⩽ 3:
...     print(i)
...     i=i+1
...
1
2
3
```

}

While {

* Une boucle while nécessite généralement trois éléments pour fonctionner correctement :
  - l'initialisation de la variable de test avant la boucle ;
  - le test de la variable associé à l'instruction while ;
  - la mise à jour de la variable de test dans le corps de la boucle.

}

Time to practice {

Lancez votre IDE et réalisez les petits programmes suivants :

#3

* Soit impairs la liste de nombres [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]. Écrivez un programme qui, à partir de la liste impairs, construit une liste pairs dans laquelle tous les éléments de impairs sont incrémentés de 1.

#4

* Écrivez un script qui dessine un triangle rectangle puis isocèle de 10 ‘*’ de haut

}

Les tests {

* Les tests sont un élément essentiel à tout langage informatique si on veut lui donner un peu de complexité car ils permettent à l'ordinateur de prendre des décisions si telle ou telle condition est vraie ou fausse. Pour cela, Python utilise l'instruction **if**

```
>>> x = 2
>>> if x == 2 :
...     print("Le test est vrai !")
...
Le test est vrai !
```

}

Les tests à plusieurs cas {

* Parfois, il est pratique de tester si la condition est **vraie ou si elle est fausse** dans une même instruction if. Plutôt que d'utiliser deux instructions if, on peut se servir des instructions if et else

```
>>> x = 5
>>> if x == 2 :
...     print("Le test est vrai !")
... else :
...     print("Le test est faux !")
Le test est faux !
```

}

Les tests à plusieurs cas {

* On peut utiliser une série de tests dans la même instruction if, notamment pour tester plusieurs valeurs d'une même variable.

```
>>> import random
>>> boisson = random.choice(['café', 'bière', 'soda'])
>>> if boisson == 'café' :
...     print('Voici un café')
... elif boisson == 'bière' :
...     print('Voici une bière')
... elif boisson == 'soda' :
...     print('Voici un soda')
...
Voici une bière
```

}

Tests multiples {

* En Python, on utilise le mot réservé **and** pour l'opérateur **ET** et le mot réservé **or** pour l'opérateur **OU**.

```
>>> x = 2
>>> y = 2
>>> if x == 2 and y == 2 :
...     print("le test est vrai")
...
le test est vrai
```

}

Tests multiples {

* Vous pouvez aussi tester directement l'effet de ces opérateurs à l'aide de True et False :

```
>>> True or False
True

>>> not True
False
>>> not False
True
```

}

Break & continue {

Ces deux instructions permet de modifier le comportement d'une boucle (for ou while) avec un test.

* L'instruction break stoppe la boucle.

```
>>> for i in range(5):
...     if i > 2 :
...         break
...     print(i)
```

* L'instruction continue saute à l'itération suivante.

```
>>> for i in range(5):
...     if i == 2:
...         continue
...     print(i)
```

}

Time to practice {

#5

A Fort Boyard, le père Fouras nous pose l'énigme suivante : Pour ouvrir le coffre où se trouve la clé, trouve la combinaison à trois chiffres sachant que :

- Le nombre est inférieur à 200.
- Deux de ses chiffres sont identiques.
- La somme de ses chiffres est égale à 5.
- C'est un nombre pair.

* On se propose d'utiliser une méthode dite **brute force**, c'est à dire d'utiliser une boucle et à chaque itération on teste les 4 conditions. Avez-vous réussi à trouver la même solution en raisonnant ?

}

Time to practice { La conjecture de Syracuse

#6

Soit un entier positif n. Si n est pair, alors le diviser par 2. Si il est impair, alors le multiplier par 3 et lui ajouter 1. En répétant cette procédure, la suite de nombres atteint la valeur 1 puis se prolonge indéfiniment par une suite de trois valeurs triviales appelée cycle trivial.

Par exemple, les premiers éléments de la suite de Syracuse si on prend comme point de départ 10 sont : 10, 5, 16, 8, 4, 2, 1. . .

Écrivez un script qui, partant d'un entier positif n (par exemple 10 ou 20), crée une liste des nombres de la suite de Syracuse. Avec différents points de départ (c'est-à-dire avec différentes valeurs de n), la conjecture de Syracuse est-elle toujours vérifiée ? Quels sont les nombres qui constituent le cycle trivial ?
Remarques

1. Pour cet exercice, vous avez besoin de faire un nombre d'itérations inconnu pour que la suite de Syracuse atteigne le chiffre 1 puis entame son cycle trivial. Vous pourrez tester votre algorithme avec un nombre arbitraire d'itérations, typiquement 20 ou 100, suivant votre nombre n de départ.
2. Un nombre est pair lorsque le reste de sa division entière (opérateur modulo %) par 2 est nul.

}

Les fichiers {

* Dans la plupart des programmes, on doit lire ou écrire dans un fichier. Python possède pour cela tout un tas d'outils qui vous simplifient la vie.

```
>>> file = open('zoo.txt', 'r')
>>> lignes = file.readlines()
>>> lignes
['girafe\n', 'tigre\n', 'singe\n', 'souris\n']
>>> for ligne in lignes:
...     print(ligne)
girafe
tigre
singe
souris
>>> file.close()
```

}

Les fichiers {

* Lors de l'ouverture d'un fichier, il faut spécifier le type de droits que l'on souhaite :
  - 'r' : pour l'ouverture en lecture seule (Read).
  - 'w' : pour l'ouverture en mode écriture (Write), si le fichier existe déjà son contenu est écrasé sinon il est créé.
  - 'a' : pour l'ouverture en mode ajout (Append), si le fichier existe déjà on écrit à la fin de ce dernier sans écraser son contenu sinon il est créé.

}

Read et readline {

* Il existe d'autres méthodes que readlines() pour lire (et manipuler) un fichier. Par exemple, la méthode read() lit tout le contenu d'un fichier et renvoie une chaîne de caractères unique.

* La méthode readline() lit une ligne d'un fichier et la renvoie sous forme d'une chaîne de caractères. À chaque nouvel appel de readline(), la ligne suivante est renvoyée. Associée à la boucle while, cette méthode permet de lire un fichier ligne par ligne.

}

L’écriture dans un fichier {

* Écrire dans un fichier est aussi simple que de le lire. Voyez l’exemple suivant :

```
>>> animaux = ['poisson', 'abeille', 'chat']
>>> filout = open('zoo.txt', 'w')
>>> for animal in animaux :
...     filout.write(animal + ‘\n’)

>>> filout.close()
```

}

Méthode optimisée d'ouverture et fermeture de fichier {

* Le mot-clé **with** permet d'ouvrir et de fermer un fichier de manière optimisée. Si pour une raison ou une autre l'ouverture conduit à une erreur (problème de droits, etc), l'utilisation de **with** garantit la bonne fermeture du fichier contrairement à la méthode **open()**

```
>>> with open('zoo.txt', 'r') as filin :
...     for ligne in filin:
...             print(ligne)
```

* Une fois sorti, Python fermera automatiquement le fichier. Vous n'avez donc plus besoin d'invoquer la méthode close().

}

Time to practice { Les fichiers

#3 bis

* Depuis un fichier ‘suite.txt’ contenant une suite de chiffres impairs [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21].

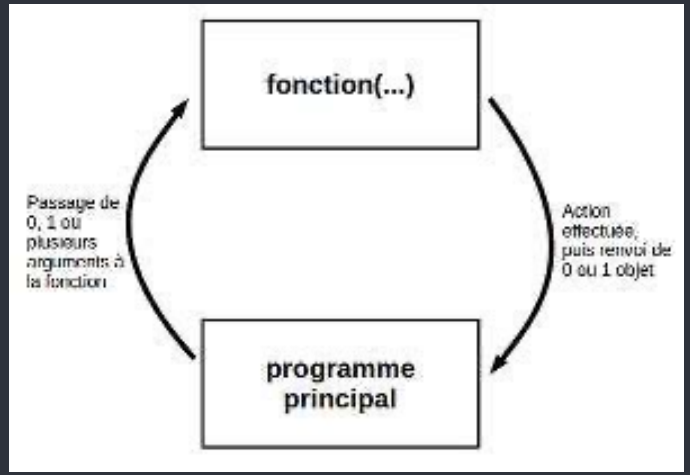
1. Écrivez un programme qui, à partir de du fichier ‘suite.txt’, construit une liste pairs dans laquelle tous les éléments de impairs sont incrémentés de 1.

2. Enfin, écrivez dans un nouveau fichier ‘suite_pair.txt’, les éléments de votre nouvelle liste

}

Les fonctions {

* En programmation, les fonctions sont très utiles pour réaliser plusieurs fois la même opération au sein d'un programme. Elles permettent également de rendre le code plus lisible et plus clair en le fractionnant en blocs logiques.



}

Les fonctions {

* Une fonction effectue une tâche. Pour cela, elle reçoit éventuellement des arguments et renvoie éventuellement un résultat. Ce qu'il se passe à l'intérieur de la fonction ne regarde que le programmeur. Seuls les paramètres d'entrée et les valeurs de sortie comptent pour l'utilisateur.

* Chaque fonction effectue en général une tâche unique et précise. Si cela se complique, il est plus judicieux d'écrire plusieurs fonctions (qui peuvent éventuellement s'appeler les unes les autres). Cette modularité améliore la qualité générale et la lisibilité du code.

}

Les fonctions {

* Pour définir une fonction, Python utilise le mot-clé def et si on veut que celle-ci renvoie une valeur, il faut utiliser le mot-clé return. Par exemple :

```
>>> def carre(x) :
...     return x ** 2
...
>>> print(carre(2))
4
```

* Nous avons passé un argument à la fonction carre() qui nous a retourné une valeur (stockable dans une variable) que nous avons affichée à l'écran.

}

Les fonctions {

* Pour définir une fonction, Python utilise le mot-clé def et si on veut que celle-ci renvoie une valeur, il faut utiliser le mot-clé return. Par exemple :

```
>>> def carre(x) :
...     return x ** 2
...
>>> print(carre(2))
4
```

* Nous avons passé un argument à la fonction carre() qui nous a retourné une valeur (stockable dans une variable) que nous avons affichée à l'écran.

}

Les fonctions – passage d'arguments {

* Le nombre d'argument(s) que l'on peut passer à une fonction est variable.

* Une particularité des fonctions en Python est que vous n'êtes pas obligé de préciser le type des arguments que vous lui passez, dès lors que les opérations que vous effectuez avec ces arguments sont valides. Python est connu comme étant un langage au typage dynamique, c'est-à-dire qu'il reconnaît pour vous le type des variables au moment de l'exécution, par exemple :

```
>>> def fois(x, y) :
...     return x * y
>>> fois(2,3)
6
>>> fois('to',2)
'toto'
```

}

Les fonctions – passage d’arguments {

* Les fonctions Python peuvent être définies avec des arguments nommés qui peuvent avoir des valeurs par défaut fournies. Lorsque les arguments de fonction sont passés à l'aide de leurs noms, ils sont appelés arguments de mot-clé.

* L'utilisation d'arguments de mots clés lors de l'appel d'une fonction permet de transmettre les arguments dans n'importe quel ordre – ne pas juste l'ordre dans lequel ils ont été définis dans la fonction. Si la fonction est invoquée sans valeur pour un argument spécifique, la valeur par défaut sera utilisée.

}

Les fonctions – passage d’arguments {

```
def findvolume(length=1, width=1, depth=1):
  print("Length = " + str(length))
  print("Width = " + str(width))
  print("Depth = " + str(depth))
  return length * width * depth;

findvolume(1, 2, 3)
findvolume(length=5, depth=2, width=4)
findvolume(2, depth=3, width=4)
```

}

Variables locales et variables globales {

* Lorsqu'on manipule les fonctions il est essentiel de bien comprendre comment se comportent les variables. Une variable est dite locale lorsqu'elle est créée dans une fonction, car elle n'existera et ne sera visible que lors de l'exécution de la dite fonction. Une variable est dite globale lorsqu'elle est créée dans le programme principal ; elle sera visible partout dans le programme.

}

Variables locales et variables globales {

```python
# définition d'une fonction carre()
def carre(x):
    y = x**2
    return y

# programme principal
z=5
resultat = carre(5)
print(resultat)
```

}

pythonCourse.py

workshop.css

Time to practice { Les fonctions

#7

* Un nombre premier est un entier naturel qui admet exactement deux diviseurs distincts entiers et positifs. Ces deux diviseurs sont 1 et le nombre considéré les nombres premiers étant ceux qui ne possèdent pas d'autre diviseur.

* Par exemple, le nombre entier 7 est premier car 1 et 7 sont les seuls diviseurs entiers et positifs de 7.

* Concevez une fonction is_prime() qui prend en argument un ombre entier positif n (supérieur à 2) et qui renvoie un booléen True si n est premier, et False si n n'est pas premier. Déterminez tous les nombres premiers de 2 à 100. On souhaite avoir une sortie comme ça :

```
2 est premier
3 est premier
4 n'est pas premier
```

}

Les modules {

* Il existe une série de modules que vous serez probablement amenés à utiliser si vous programmez en Python. En voici une liste non exhaustive. Pour la liste complète, reportez-vous à la page des modules 4 sur le site de Python :
  - math : fonctions et constantes mathématiques de base (sin, cos, exp, pi. . .).
  - sys : passage d'arguments, gestion de l'entrée/sortie standard. . .
  - os : dialogue avec le système d'exploitation (permet de sortir de Python, lancer une commande en shell...).
  - random : génération de nombres aléatoires.

* L'instruction import permet d'accéder à toutes les fonctions du module

}

Les modules {

* Avec les modules, on peut importer la totalité des fonctions ou juste certaines qui nous serait utile :

```
>>> from random import randint
>>> randint(0,10)
37
```

* À l'aide du mot-clé from, vous pouvez importer une fonction spécifique d'un module donné. Remarquez que dans ce cas il est inutile de répéter le nom du module, seul le nom de fonction est requis.

}

Les modules {

* Pour obtenir de l'aide sur un module rien de plus simple, il suffit d'invoquer la commande help() :

```
>>> import random
>>> help(random)
...
```

* Si on veut connaître d'un seul coup d'oeil toutes les méthodes ou variables associées à un objet, on peut utiliser la fonction dir()

}

Module sys {

* Le sys contient (comme son nom l'indique) des fonctions et des variables spécifiques au système, ou plus exactement à l'interpréteur lui-même. Par exemple, il permet de gérer l'entrée (stdin) et la sortie standard (stdout). Ce module est particulièrement intéressant pour récupérer les arguments passés à un script Python lorsque celui-ci est appelé en ligne de commande.

```
#!/usr/bin/env python3
import sys
print(sys.argv)
```

* Ensuite lancez test.py suivi de plusieurs arguments. Par exemple :

```
$ python3 test.py salut girafe 42
['test.py', 'salut', 'girafe', '42']
```

}

Module sys {

* Dans l'exemple précédent, $ représente l'invite du shell Linux, test.py est le nom du script Python, salut, girafe et 42 sont les arguments passés au script.

* La variable sys.argv est une liste qui représente tous les arguments de la ligne de commande, y compris le nom du script lui même qu'on peut retrouver dans sys.argv[0]. On peut donc accéder à chacun de ces arguments avec sys.argv[1], sys.argv[2]. . .

}

Module sys {

* On peut aussi utiliser la fonction sys.exit() pour quitter un script Python. On peut donner un argument à cette fonction (en général une chaîne de caractères) qui sera renvoyé au moment où Python quittera le script. Par exemple, si vous attendez au moins un argument en ligne de commande, vous pouvez renvoyer un message pour indiquer à l'utilisateur ce que le script attend comme argument :

```
import sys
if len(sys.argv) ≠ 2:
    sys.exit("ERREUR : il faut exactement un argument.")

# suite du script
```

}

Module os {

* Le module os gère l'interface avec le système d'exploitation. os.path.exists() est une fonction pratique de ce module qui vérifie la présence d'un fichier sur le disque.

```
>>> import sys
>>> import os
>>> if os.path.exists("toto.pdb"):
...   print("le fichier est présent")
... else:
...   sys.exit("le fichier est absent")
le fichier est absent
```

* Dans cet exemple, si le fichier n'est pas présent sur le disque, on quitte le programme avec la fonction exit() du module sys.

}

Module os {

* La fonction system() permet d'appeler n'importe quelle commande externe.

```
>>> import os
>>> os.system("ls -al")
total 40
drwxr-xr-x   7 lsgrge  staff  224  6 oct 11:50 .
drwxr-xr-x  16 lsgrge  staff  512  4 oct 16:11 ..
-rw-r--r--   1 lsgrge  staff  136  3 oct 14:13 files.py
-rw-r--r--   1 lsgrge  staff  172  6 oct 10:29 hello.py
-rw-r--r--   1 lsgrge  staff  182  6 oct 11:57 os.py
-rwxr-xr-x   1 lsgrge  staff   43 27 sep 09:57 test.py
-rw-r--r--   1 lsgrge  staff   21  3 oct 14:13 zoo.txt
```

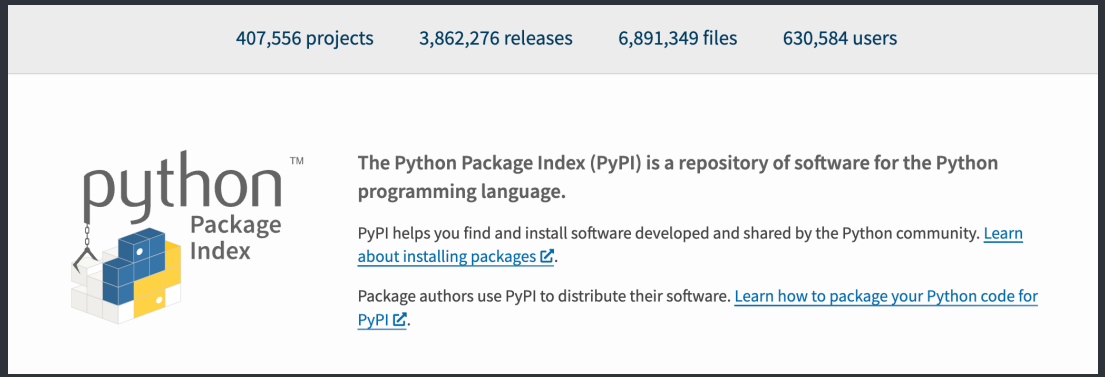
}

Les modules {

Python regorge de modules, il suffit de chercher sur PyPi ou sur des projets Github.

Vous pouvez également développer les vôtres !

}



Installer des modules {

* Grâce à l'utilitaire pip, on peut télécharger de nouveaux modules pour les importer dans notre code :

```
$ python3 -m pip install <module>
```

}

Bonne pratique – environnement virtualisé {

* Les imports de nombreux modules ou le fait de devoir travailler avec des versions précises pour certaines applications peuvent rendre notre installation de python instable.

* La solution ? Travailler dans des environnement virtuels :

```
$ python3 –m venv virtualenv/MonProjet # créer un environnement virtuel

$ source virtualenv/MonProjet/bin/activate # activer l'environnement virtuel
}

$ deactivate # sortir de l'environnement virtuel
```

Bonne pratique – environnement virtualisé {

* Cette technique permet de conserver un environnement de travail propre et de pouvoir travailler avec plusieurs versions de python différentes mais aussi des versions de module différentes.

* Dorénavant, lorsque vous travaillez sur différents programmes complexe faisant appel à des modules externe, il faudra le faire dans un Virtualenv !

}

Retour sur les chaînes de caractères {

* On a vu que les chaînes de caractères peuvent être considérées comme des listes

```
>>> animaux = "girafe tigre"
>>> animaux[0:4]
'gira'
```

* Mais la différence avec les listes, les chaînes de caractères présentent toutefois une différence notable, ce sont des listes **non modifiables**. Une fois définie, vous ne pouvez plus modifier un de ses éléments. Python renvoie un message d'erreur :

```
>>> animaux = "girafe tigre"
>>> animaux[4]
'f'
>>> animaux[4] = "F"
Traceback (most recent call last):
```

}

Retour sur les chaînes de caractères {

* Voici quelques méthodes spécifiques aux objets de type string :

```
>>> x = "girafe"
>>> x.upper()
'GIRAFE'
>>> 'TIGRE'.lower()
'tigre'
>>> x.capitalize()
'Girafe'
```

}

Retour sur les chaînes de caractères {

* Il existe une méthode associée aux chaînes de caractères qui est particulièrement pratique, la fonction split() :

```
>>> animaux = "girafe tigre singe"
>>> animaux.split()
['girafe', 'tigre', 'singe']
```

* La fonction split() découpe la ligne en plusieurs éléments appelés champs, en utilisant comme séparateur les espaces ou les tabulations. Il est possible de modifier le séparateur de champs, par exemple :

```
>>> animaux = "girafe:tigre:singe"
>>> animaux.split(":")
['girafe', 'tigre', 'singe']
```

}

Retour sur les chaînes de caractères {

* La fonction find() recherche une chaîne de caractères passée en argument.

```
>>> animal = "girafe"
>>> animal.find('i’)
1
>>> animal.find('tig')
-1
```

* Si l’élément recherché est trouvé, alors l’indice du début de l’élément dans la chaîne de caractères est renvoyé. Si l’élément n’est pas trouvé, alors la valeur -1 est renvoyée.

}

Retour sur les chaînes de caractères {

* On trouve aussi la fonction replace(), qui serait l'équivalent de la fonction de substitution de la commande Unix sed :

```
>>> animaux = "girafe tigre"
>>> animaux.replace("tigre", "singe")
'girafe singe'
>>> animaux.replace("i", "o")
'gorafe togre'
```

* Enfin, la fonction count() compte le nombre d'occurences d'une chaîne de caractères passée en argument :

```
>>> animaux = "girafe tigre"
>>> animaux.count("i")
2
```

}

Conversion d'une liste de chaînes de caractères en unechaîne de caractères {

* La conversion d'une liste de chaînes de caractères en une chaîne de caractères est un peu particulière puisqu'elle fait appelle à la fonction join().

```
>>> seq = ["A", "T", "G", "A", "T"]
>>> seq
['A', 'T', 'G', 'A', 'T']
>>> "-".join(seq)
'A-T-G-A-T'
>>> " ".join(seq)
'A T G A T'
>>> "".join(seq)
'ATGAT'
```

}

* Attention, la fonction join() ne s'applique qu'à une liste de chaînes de caractres !

Time to practice { Palindrome

#7

Un palindrome est un mot ou une phrase dont l'ordre des lettres reste le même si on le lit de gauche à droite ou de droite à gauche. Par exemple, ressasser et Engage le jeu que je le gagne sont des palindromes.

Écrivez la fonction palindrome() qui prend en argument une chaîne de caractères et qui affiche xxx est un palindrome si la chaîne de caractères est un palindrome et xxx n'est pas un palindrome sinon (bien sur, xxx est ici le palindrome en question). Pensez à vous débarasser au préalable des majuscules et des espaces.

Testez ensuite si les expressions suivantes sont des palindromes :

- Radar
- Never odd or even
- Karine alla en Iran
- Un roc si biscornu

}

Retour sur les listes {

* Comme pour les chaînes de caractères, les listes possèdent de nombreuses méthodes (on rappelle, une méthode est une fonction qui agit sur l'objet à laquelle elle est attachée par un .) qui leur sont propres et qui peuvent se révéler très pratiques.

* append() que nous avons déjà vue permet d'ajouter un élément à la fin d'une liste existante.

* insert() pour insérer un objet dans une liste avec un indice déterminé.

}

Retour sur les listes {

* del pour supprimer un élément d'une liste à un indice déterminé.
* remove() pour supprimer un élément d'une liste à partir de sa valeur.
* sort() pour trier une liste.
* reverse() pour inverser une liste.
* count() pour compter le nombre d'éléments (passé en argument) dans une liste.

}

Time to practice { Triangle de Pascal

#8 Voici le début du triangle de Pascal :

```
1
11
121
1331
14641
 ...
```

Comprenez comment une ligne est construite à partir de la précédente. A partir de l'ordre 1 (ligne 2, 11), générez l'ordre suivant (121). Vous pouvez utiliser une liste préalablement générée avec range(). Généralisez à l'aide d'une boucle.
Ecrivez dans un fichier pascal.out les lignes du triangle de Pascal de l'ordre 1 jusqu'à l'ordre 10.

}

Plus sur les fonctions - Fonctions récursives {

* Une fonction récursive est une fonction qui s'appelle elle même. Les fonctions récursives permettent d'obtenir une efficacité redoutable dans la résolution de certains algorithmes comme le tri rapide

* Oublions la recherche d'efficacité pour l'instant, et concentrons nous sur un exemple de base, la fonction factorielle, qui illustre à merveille comment sont gérés des fonctions récursives en Python.

}

Plus sur les fonctions - Fonctions récursives {

* Regardez ce code et tenter de comprendre ce qu'il va faire :

```python
def calc_factorielle(nb):
    if nb == 1:
        return 1
    else:
        return nb * calc_factorielle(nb - 1)


# prog principal
print(calc_factorielle(4))
```

}

Plus sur les fonctions - Fonctions récursives {

* Même si les fonctions récursives peuvent être compliqué à comprendre, le but est de vous illustrer qu'une fonction qui en appelle une autre reste "figée" dans le même état, jusqu'à temps que la fonction appelée lui renvoie une valeur.

}

Plus sur les fonctions – Portée des variables {

* Il est très important lorsque l'on manipule des fonctions de connaître la portée des variables. On a vu que les variables créées au sein d'une fonction ne sont pas visibles à l'extérieur de celle-ci car elles étaient locales à la fonction. Observez le code suivant :

```
>>> def mafonction():
...     x=2
...     print('x vaut {} dans la fonction'.format(x))
...
>>> mafonction()
x vaut 2 dans la fonction
>>> print(x)
Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
NameError: name 'x' is not defined
```

}

Plus sur les fonctions – Portée des variables {

* Lorsqu'une variable déclarée à la racine du module (c'est comme cela que l'on appelle un programme Python), elle est visible dans tout le module. Dans ce cas, la variable x est visible dans le module principal et dans toutes les fonctions du module. Cependant, Python ne permet pas la modification d'une variable globale dans une fonction :

```
>>> def maFonction():
...     x=x+1
...
>>> x = 1
>>> maFonction()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in fct
UnboundLocalError: local variable 'x' referenced before assignment
```

}

Plus sur les fonctions – Portée des variables {

* L'erreur renvoyée montre que Python pense que x est une variable locale qui n'a pas été encore assignée. Si on veut vraiment modifier une variable globale dans une fonction, il faut utiliser le mot-clé global :

```
>>> def mafonction():
...     global x
...     X = x + 1
...
>>> x = 1
>>> mafonction()
>>>x
2
```

}

02

Les Types complexes

< Comprendre et maîtriser les dictionnaires et les classes >

Dictionnaires et tuples {

* Les dictionnaires se révèlent très pratiques lorsque vous devez manipuler des structures complexes à décrire et que les listes présentent leurs limites. Les dictionnaires sont des collections non ordonnées d'objets, c'est à dire qu'il n'y a pas de notion d'ordre (i.e. pas d'indice). On accède aux valeurs d'un dictionnaire par des clés.

```
>>> ani1 = {}
>>> ani1['nom'] = 'girafe'
>>> ani1['taille'] = 5.0
>>> ani1['poids'] = 1100
>>> ani1
{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}
```

}

Dictionnaires et tuples {

* En premier, on définit un dictionnaire vide avec les symboles {} (comme pour les listes avec []). Ensuite, on remplit le dictionnaire avec différentes clés ('nom', 'taille', 'poids') auxquelles on affecte des valeurs ('girafe', 5.0, 1100). Vous pouvez mettre autant de clés que vous voulez dans un dictionnaire (tout comme vous pouvez ajouter autant d'éléments que vous voulez dans une liste).

```
>>> ani2 = {'nom':'singe', 'poids':70, 'taille':1.75}
```

* Mais rien ne nous empêche d'ajouter une clé et une valeur supplémentaire :

```
>>> ani2[age] = 15
```

}

Dictionnaires et tuples - Méthodes keys() et values() {

* Les méthodes .keys() et .values() renvoient, comme vous vous en doutez, les clés et les valeurs d'un dictionnaire :

```
>>> ani2.keys()
dict_keys(['poids', 'nom', 'taille'])
>>> ani2.values()
dict_values([70, 'singe', 1.75])
```

* Les mentions dict_keys et dict_values indiquent que nous avons à faire à des objets un peu particulier. Si besoin, nous pouvons les transformer en liste avec la fonction list() (par exemple : list(ani2.values())).

}

Dictionnaires et tuples - Liste de dictionnaires {

* En créant une liste de dictionnaires qui possèdent les mêmes clés, on obtient une structure qui ressemble à une base de données :

```
>>> animaux = [ani1, ani2]
>>> animaux
[{'nom': 'girafe', 'poids': 1100, 'taille': 5.0}, {'nom': 'singe',
'poids': 70, 'taille': 1.75}]
>>>
>>> for ani in animaux:
...     print(ani['nom'])
...
girafe
singe
```

}