**python**

**INTRODUCTION**

Créé par Guido van Rossum d’après Monthy Python

L’indentation correspond à 4 « espace », faisable avec un seul « Tab » est essentiel à la lecture du code ce qui lui permet de connaitre le début et la fin du code. Cela remplace d'autres types de clé de lecture du code (exemple: **[ { (** ))

Le texte qui suit « **>>** » correspond à l’**output** of the **interactive shell**

La police la plus communément utilisée en programmation est « **Courrier New** » car chaque caractère à la même taille ce qui facilite la lecture notamment vis-à-vis de l’indentation.

Pour écrire des commentaires, utiliser le symbole « **#** »

Il n’est possible d’écrire et de copier-coller qu’une ligne à la fois dans le module d'exécution

Un code entouré par trois **" " "** permet d'étendre l'**output** sur plusieurs lignes

Un code peut s'écrire sur plusieurs lignes à l'aide du symbole **\**

**Data types :** Chaque donnée Python est regroupée dans des catégories appelées Data Types  
Chaque valeur dans Python est appelé un **object.**

* **Str : Strings** 🡺 Chaine de caractères
* **Int : Integer 🡺** Nombres entiers
* **Float : Floating point numbers 🡺** Décimaux
* **Bool : Boolean 🡺** Résultat True ou False
* **NoneType 🡺** N'ont pas de valeur et renvoie toujours comme résultat None

Un **object** dans Python a 3 propriétés :

* **Identity :** emplacement dans la mémoire de l'ordinateur (ne change jamais)
* **Data type :** Catégorie de donnée auquel appartientl'**object** ce qui détermine ses propriétés de l'objet (ne change jamais)
* **Value :** Correspond à la donnée qu'il représente

**Constantes, variables, incrémenter ou décrémenter**

* **Constant** : Valeur prédéterminée qui ne variera pas
* **Variable** : Valeur qui peut changer. Il suffit d'affecter un ou plusieurs caractères à une valeur en utilisant un **assignment operator** (**=**). On peut assigner n'importe quelle valeur à une variable (texte, chiffre, etc…)  
  Règles pour nommer les variables :
  + Seulement des chiffres, lettres et underscore
  + Ne peut commencer par un chiffre
  + Ne peut s'appeler comme un **Keywords** de Python
* **Increment/Decrement**:  
  x = 10  
  x += 1 ou x = x + 1 🡺 11  
  x -= 1 ou x = x – 1 🡺 9

**Errors and Exceptions**

Il existe 2 types d'erreurs dans Python:

* **Syntax error =** Non-respect des règles de syntaxe de Python (une chaine de caractères commence par " et finit par ")

**Cette erreur bloque totalement l'exécution du code**

* **Exceptions** = Tout sauf erreur de syntaxe
  + ZeroDivisionError = Division par zéro
  + IndentationError = Erreur d'indentation du texte dans le code
  + ValueError = Erreur dans le type de donnée (str en int par exemple)
  + NameError = variable non définie
  + IndexError = fait appel à une position d'index de liste supérieur à la taille de la liste

**Ce type d'erreur n'est pas nécessairement bloquant**

**Operators**

**Operands:** Les valeurs de chaque côté de l'opérateur

**Expression**: L'opérateur plus le **Operands**

3 + 2:

3 et 2 sont les **Operands**

+ est le **Operator**

L'ensemble constitue une **Expression**

Ordre de calcul: Please Excuse My Dear Aunt Sally

Parentheses  
Exponents  
Multiplication / Division  
Addition / Substraction

En cas d'égalité, on effectuera les opérations de gauche à droite (15 / 3 \*2) 🡺 15/3 = 5 🡺 5\*2 = 10

**Arithmetic Operators:**

\*\* Exponents 3 \*\*2 9 Exposant/Puissance

% Modulo /Remainder 14%4 2 Le restant de la division

// Integer division/floored quotient 13 // 8 1 Entier/quotient de la division

/ Division 13/8 1.625

\* Multiplication

- Substraction

+ Addition

**Comparison Operators** 🡺 Résultat booléen: True or False

> Greater than 100>10 True

< Less than 100<10 False

>= idem ou égal 2>=2 True

<= idem ou égal 1<=4 True

== Equal 6 ==9 False

!= Not Equal 3!=2 True

Attention, ne pas confondre "=" (**assignment operator**) qui signifie "affecter la valeur de … à …" et "==" qui signifie bien "vérifier que … est égal à …"

**Logical Operators**🡺 Résultat booléen: True or False

**And** Renvoie le résultat True si toutes les **expressions** comparées renvoient le résultat **True**  
Peut se cumuler 1 == 1 and 10 != 2 and 2 < 10 🡺 Renvoie **True**

**Or** Renvoie le résultat True si au moins une des **expressions** comparées renvoie le résultat **True**  
Peut se cumuler 1 == 1 or 1 == 2 or 1 == 3 🡺 Renvoie **True**

**Not** Renvoie le résultat inverse   
Se place devant l'expression à affecter not 1 == 1 🡺 Renvoie **False**

**Conditional Statements** 🡺 ce sont des types de **control structure** (fonctionne comme sur VBA)

x = 100

if x == 10:

print("10!")

elif x == 20:

print("20!")

else:

print("I don't know!")

if x == 100:

print("x is 100!")

if x % 2 == 0:

print("x is even!")

else:

print("x is odd!")

>>I don't know !

>>x is 100!

>>x is even!

La partie avec le if et son résultat s'appelle : **if-statement**La partie avec le elif et son résultat s'appelle : **elif-statement**La partie avec le else et son résultat s'appelle : **else-statement**L'ensemble s'appelle **if-else statement.**

Un if n'est pas obligé de se finir par un else.  
Le code s'exécute uniquement si l'expression testée renvoie **True**

Fonctionnement :  
If : keyword -> Expression -> : ->indentation -> Code to execute  
Else : keyword -> : -> indentation -> Code to execute  
Elif : Fonctionne comme un if (else if) pour un même ensemble **if-else statement**

Attention cependant, si le **if** renvoie **False**, il testera le **Elif** suivant dans l'ordre et s'arrêtera au premier **True**, si aucun **True** n'apparait, il ira finalement dans le **Else**

**Statements :**

Il existe 2 type de **statements :**

* **Simple statements :**

Tient sur une ligne de code : exemple print ("Hello, World !")

* **Compound statements :**

S'exprime sur plusieurs lignes de code (if-else statements). Ils sont composés de plusieurs **clauses**

Une **clause** est constituée d'un **header** suivi par une ou plusieurs **suites**.  
**Header** = **keyword** -> : -> **suites**  
**Suites** = ligne de code d'une **clause**

Exemple de **clause** constituée d'une seule **suite**

For i in range(100):  
 print("Hello, World !")

Exemple plus complexe repris de la partie ci-dessus :

x = 100

if x == 10:

print("10!")

elif x == 20:

print("20!")

else:

print("I don't know!") 1er **Compound statement** constitué de 3 **clauses**

if x == 100:

print("x is 100!") 2ème **compound statement** constitué d'une **clause**

if x % 2 == 0:

print("x is even!")

else:

print("x is odd!") 3ème **compound statement** constitué de 2 **clauses**

>>I don't know !

>>x is 100!

>>x is even!

Chacune des **clauses** ci-dessus sont constituées d'un **header** et d'une seule **suite**

NB : les **statements** peuvent être espacés sans impact sur la lecture du code

**FUNCTIONS**

**Calling** consiste à donner à une **function** un ou plusieurs **inputs** qu'on appelle **parameter** pour qu'il nous retourne un **output.**

Function (parameter 1, parameter 2, …) 🡺 Call a function: function name -> ( -> parameter -> , ->… )

**Defining Functions**

Il est possible de définir n'importe quelle fonction. Par convention une fonction ne contient pas de lettre majuscule et doit respecter les règles des variables (chiffres lettre et underscore).

Pour définir une fonction:

**def** nom\_de\_la\_fonction (paramètre 1, paramètre 2,…) **:**

**return** statement is used to exit a function and go back to the place from where it was called.

Il n'est pas obligatoire de saisir de paramètre dans ce cas laisser les parenthèses vides

Il n'est obligatoire de saisir un **return** dans ce cas le résultat renverra toujours **None**

**def** f(x, y, z):

**return** x + y +z

result = f(1, 2, 3)

**print**(result)

>> 6

**Built-In Functions**

Les **built-in functions** sont les fonctions disponibles dans la **library** de Python.

**print** =renvoie le résultat

**len** = renvoie le nombre de caractère du paramètre

**str** =transforme un paramètre en string

**int** =transforme un paramètre en integer

**float** =transforme un paramètre en float

**input** =Fais appel à une donnée à saisir par l'utilisateur (par défaut résultat sorti en string)

**Reusing functions**

Une fois une fonction définie elle peut être appelée autant de fois que l'on veut. Ceci permet de raccourcir le code

def even\_odd():

n = input ("Type a number: ")

n = int(n)

if n % 2 == 0:

print ("n is even")

else:

print ("n is odd")

even\_odd()

even\_odd()

even\_odd()

**Required and Optional Parameters**

Les **parameters** peuvent être soit **Required** soit **Optional**

S'ils sont **required** la **function** ne s'exécutera pas et retournera une **exception** dans le code dans le cas où ils n'auraient pas été saisis. 🡺 f(x) est **required**

Un **parameter** peut être **optional** en définissant une valeur par défaut dans la définition initiale du code. Pour définir sa valeur par défaut il suffit d'affecter une valeur avec un signe = dans la définition de la fonction. 🡺 f(x=2) est **optional**

Exemple :

def f(x, y=10):

return x + y

Cas n°1: Required non saisi

result = f()

print(result)

>> TypeError: f() missing 1 required positional argument: 'x'

Cas n°2: required saisi mais optional non saisi

result = f(2)

print(result)

>> 12

Cas n°3: required saisi et optional saisi

result = f(2, 5)

print(result)

>> 7

**Scope**

**Reading** une **variable** signifie trouver sa valeur  
**Writing** une **variable** signifie modifier sa valeur

Les variables peuvent être définie soit en dehors d'une fonction (sans indentation) auquel cas elle est une **global variable** soit à l'intérieur de la fonction et à un **local scope**.

**Global variable** : peut-être **Read** depuis n'importe quelle partie du code (dans une fonction ou en dehors). Pour le **Write** en dehors d'une fonction, pas de soucis par contre pour le **Write** dans une fonction il faut utiliser le **keyword global**  
**Local variable:** les variables locales ne peuvent être Read et Write que dans la fonction qui l'utilise

Cas n°1: variable utilisée dans une fonction mais non définie dans cette fonction ou globale

if x > 100:  
 print("x is > 100")

>> NameError: name x is not defined

Cas n°2: variable utilisée dans une fonction et définie dans celle-ci

def f():  
 x = 1  
 print (x)

f()

>> 1

Cas n°3: variable globale définie, utilisée dans une fonction et Write dans cette fonction

x = 100

def f():  
 global x se positionne avant le write  
 x += 1  
 print (x)

f()

>> 101

f()

>> 102 Dans ce cas x s'incrémente bien à chaque occurence

**Exception Handling**

Afin de gérer les erreurs il faut utiliser le **compound statement** constitué de **try** et **except.**

la **clause** du **try** contient l'erreur qui peut se produire. S'il n'y a pas d'erreur il va exécuter le code normalement, en revanche dès que l'erreur apparait il va passer à la clause **except**. La **function** **except** contient en **parameter** les **exceptions** à gérer (séparées par des virgules)

Exemple :

try:  
 a = input ("saisir un nombre")  
 b = input ("un autre")  
 a = int(a)  
 b = int(b)  
 print(a/b)  
except (ZeroDivisionError, ValueError):  
 print("invalid input")

NB: ne pas utiliser dans la partie except de variable définie dans la partie try puisqu'elle ne sera probablement jamais définie puisque le code dans la partie try s'interrompt dès que l'erreur apparait (mais conserve les variables définies avant l'erreur)

Exemple:

try:

c = "invalid input"

10/0

except ZeroDivisionError:

print(c)

>> invalid input

En revanche

try:

c = "invalid input"

10/0

except ZeroDivisionError:

print(c)

>> NameError: name 'c' is not defined

**Docstrings**

Les docstrings sont des commentaires placés au début d'une fonction pour expliquer son fonctionnement : Elle définit les **data types**, les **parameters** nécessaires et ce que fait la **function.**

Exemple

def add(x, y)

"""

Returns x + y

: param x: int

: param y: int

: return: int sum of x and y

"""

return x + y

Indispensable pour faciliter la lecture. **A APPLIQUER**

**CONTAINERS**

**List = [ ]**

Les **lists** sont un type de **container** qui permet de **store** des **objects** dans un ordre spécifique.

Pour créer une nouvelle liste:

* Utiliser la **function** **list** pour créer une liste vide  
  random = list() liste vide
* Utiliser des brackets (vide ou séparé par des virgules pour y attribuer des éléments)  
  random = [] liste vide  
  random = ["apple", 1984, True] liste de 3 éléments

Pour ajouter un nouvel élément à la liste il faut utiliser la **Method** **append** qui va venir rajouter un élément à la fin de la liste.

* random.append("banana")  
  random  
  >> ["apple", 1984, True, "banana"]

Les **objects** dans les **lists** sont **iterables**. Cela signifie qu'ils sont indexés et tous accessibles via un **loop**  
Les **objects** sont classé selon leur ordre dans la liste en commençant par 0

Il est donc possible de retrouver un des objects de la list via son numéro (nom de la liste [index])

* random[0]
* random[1]
* random[2]
* random[3]

>>'apple'  
>>1984  
>>True  
>>'banana'

Les **lists** sont **mutables**. Cela signifie qu'on peut ajouter, retirer ou modifier des **objects** dans le **container.**

Changer un **object** par un autre:  
random  
random[2] = False  
random  
>> ["apple", 1984, True, "banana"]  
>> ["apple", 1984, False, "banana"]

Supprimer le dernier élément de la **list** avec la **Method pop**:

* suppr = random.pop() ceci permet de conserver l'élément supprimé
* random possibilité d'utiliser le pop sans variable
* suppr

>> ["apple", 1984, True]  
>> 'banana'

**Combine** 2 **list** avec l'**operator** **addition:**

* colors1 = ["blue", "green"]
* colors2 = ["yellow", "purple"]
* colors1 + colors2

>> ["blue", "green", "yellow", "purple"]

Vérifier la présence avec le **keyword** **in** (et par extension l'absence en ajoutant **not**)

* colors = ["blue", "green"]
* "green" in colors

>> True

* "yellow" not in colors

>> True

Taille d'une liste: **len**

* len(colors)  
  >> 2

Exemple

colors = ["purple", "orange", "green"]  
guess = input("Guess a color:")

if guess in colors:  
 print("you guessed correctly!")  
else:  
 print("Wrong ! Try again.")  
>> Guess a color:

**Tuples = ( )**

Les Tuples sont simplement des listes qui ne sont pas **mutables**. On ne peut pas changer de valeur, ajouter un objet ou en supprimer un. Il reste indexé et utilisé avec le **keyword in**

Pour créer un **Tuples**:

* my\_tuples = tuple() créé un tuple vide
* my\_tuples = () créé un tuple vide
* my\_tuples = ("R2D2", 1984, True) créé un tuple avec 3 objets
* my\_tuples = ("un seul objet",) **Bien mettre la virgule quand y en a qu'un**

**Dictionaries = { }**

Les **dictionaries** est une liste qui utilise des **keys** associées à des **values** créent un **key-value pair**. Faire un **mapping** consiste à associer une clé à une valeur.

Les **dictionaries** ne sont pas indexé, il n'y a donc pas d'ordre dans la liste  
Les **dictionaries** sont mutables, on peut donc ajouter/supprimer des **key-value pairs**On peut vérifier qu'une **key** est dans un **dictionarie** mais pas si une **value** est présente

#Créer un dictionnaire:

dico = dict() utiliser la fonction **dict**

dico = {} accolades

# Créer une liste

livre = {"Dracula":"Stroker"} Pas besoin de mettre une virgule quand il est seul

#Ajouter un élément

livre["1984"] = "Orwell"

#Regarder ma clé

livre["1984"]

#Regarder l'existence d'une clé dans mon dictionnaire

"Le procès" **in** livre

#Supprimer une paire

**del** livre["1984"]

**Containers in Containers**

On peut stocker des **lists** dans des **lists**

liste\_centrale = []

liste1 = ["Bob", "Alfred"]

liste2 = ["Jean", "Batman"]

liste\_centrale.append(liste1)

liste\_centrale.append(liste2)

print liste\_centrale

>> [["Bob", "Alfred"],["Jean", "Batman"]]

* les listes sont indexées
* un ajout/suppression dans la liste1 se reflètera comme il se doit dans la liste\_centrale

On peut stocker

* un **tuple** dans une **list**
* une **list** dans un **tuple**
* un **dictionary** dans une **list** ou un **tuple**

Par ailleurs une **list**, **tuple** ou **dictionary** peuvent être des **value** dans un **dictionary**

**STRING MANIPULATION**

**Outils d'utilisation des chaines de caractères**

* Trois guillemets : écrire un texte sur plusieurs lignes
* Les caractères d'un texte son numérotés comme pour les listes (premier caractère 0, second 1 etc…) en lecture de gauche à droite
* Pour une lecture de droite à gauche il faut partir de -1 (dernière lettre) puis -2 etc…

ff = tic

ff[0]  
>>'t'

ff[-1]  
>>'c'

**Strings** sont **immutables** = on ne peut pas changer un caractère d'un **string.**

* **Concatenate**: en utilisant le symbole +

"cha" + "chat"  
>> 'chachat'

* **Multiplication**: en utilisant le symbole \*

"cha" \*2  
>> 'chacha'

* **Case** en utilisant les **Method**
  + "chacha".upper() Majuscule  
    >> 'CHACHA'
  + "CHACHA".lower() Minuscule  
    >> 'chacha'
  + "chacha est magique".capitalize() Première lettre de la phrase  
    >>'Chacha est magique'
  + "chacha est magique. chacha !".title() Première lettre de chaque phrase.
* **Format** est une **Method** qui permet de remplacer le/les **{ }** d'un texte par un/des mot ou variables

Exemple:

prenom = input("Saisir votre prénom ")

age = input("Saisir votre âge ")

passion = input("Saisir votre passion ")  
phrase = "Je m'appelle {}, j'ai {} et j'aime le {}".format(prenom, age, passion)  
print (phrase)

* **Split** est une **Method** qui permet de séparer un texte selon un **parameter**. Le résultat devient une **list.**
  + "Hello.How.Are.You".split(".")  
    >>['Hello', 'How', 'Are', 'You'] Bien noter que le **parameter** disparait du texte
* **Join** est une **Method** qui permet d'ajouter de nouveaux caractères entre chaque caractère d'un **string**
  + "+".join("abc")  
    >>'a+b+c'
  + mots = ["les" ,"mots", "sans" , "espace"]  
    phrase = " ".join(mots)  
    phrase  
    >>'les mots sans espace'
* **Strip Space** est une **Method** qui permet de supprimer les espaces
  + " the ".strip()  
    >>'the'
* **Replace** est une **Method** qui permet de remplacer un caractère par un autre
  + "J'aime pas la lettre a".replace("a", "@")  
    >>'J'@ime p@s l@ lettre @"
* **Index** est une **Method** qui permet de trouver la position du premier caractère recherché (attention renvoi une **exception** (ValueError) s'il n'y en a pas)
  + "Animals".index("i")  
    >> 2
* **In** pour savoir si "cat" est contenu dans "cat in the hat" >> True
* **Newline** permet de sauter une ligne dans le résultat affiché, il suffit d'écrire **\n**
  + print("line1\nline2")  
    >>line1  
    >>line2

**Escaping** consiste à mettre un **\** devant un caractère qui signifie normalement quelque chose en code Python pour qu'il comprenne qu'il ne doit pas l'interpréter en tant que code mais comme un **string**

* "She said "Surely"." 🡺 >>SyntaxError
* "She said \"Surely\"." 🡺 >> 'She said "Surely" NB: Alterner les **"** et les **'** est possible.

**Slicing** consiste à prendre dans un contenu indexé (liste, tuple, string, …) des bornes de ce contenu via [début(inclus):fin(exclus)].

* base="0123456"
* base[0:3]  
  >> '012'
* base[:5]  
  >> '01234'
* base[2:]  
  >> '23456'
* base[:]  
  >> '0123456'

On peut également rajouter un 3ème chiffre pour déterminer de combien doit on itérer (par défaut 1)

* base[0:3:2]  
  >> '02'

**LOOPS**

**For-loops**

Une **for-loop** est une boucle appliquée sur un **iterable.** ce process s'appelle **iterating**.  
Comment ça marche ?  
**for** variable **in** iterable:  
 instruction

variable sera la variable utilisée par la boucle pour stocker les données au fur et à mesure de la boucle. A chaque passage sont contenu sera remplacé par le nouveau.

iterable est n'importe quelle **iterable** à passer en revue.

list tuple dictionary

tv = ["Got","Himym","Ncis"] ("Got","Himym","Ncis") {"Got","Himym","Ncis"}

for show in tv: for show in tv: for show in tv:  
 print(show) print(show) print(show)  
>> Got >> Got >> Got   
>> Himym >> Himym >> Himym   
>> Ncis :2 >> Ncis >> Ncis

Changer les objets dans un **mutable iterable:**

tv = ["Game of Thrones", "How i met", "Daredevil"]

i = 0

for show in tv:

new = tv[i]

new = new.upper()

tv[i] = new

i += 1

print(tv)

['GAME OF THRONES', 'HOW I MET', 'DAREDEVIL']

On a utilisé ici un **index variable** c’est-à-dire une variable qui contient l'**integer** représentant l'**index** de l'**iterable.** On a ensuite remplacé l'objet de la liste une fois modifié et **increment** à chaque passage de loop.

Comme passer en revue un iterable est fréquent, on va utiliser la **Method** **enumerate** qui va faire ça.

tv = ["Game of Thrones", "How i met", "Daredevil"]

for i, show in enumerate(tv):

tv[i] = show.upper()

print(tv)

['GAME OF THRONES', 'HOW I MET', 'DAREDEVIL']

On a donc 2 variables: **i** qui va contenir l'index donné par **enumerate** et **show** qui sera l'objet de l'index actuel.

Enfin, il est possible d'utiliser 2 boucles pour joindre 2 listes en une :

tv = ["Game of Thrones", "How i met", "Daredevil"]

cinema = ["Fight Club", "Snatch", "Iron Man"]

ecran = []

for i, movie in enumerate(tv):

ecran.append(movie.upper())

for i, movie in enumerate(cinema):

ecran.append(movie.upper())

print(ecran)

['GAME OF THRONES', 'HOW I MET', 'DAREDEVIL', 'FIGHT CLUB', 'SNATCH', 'IRON MAN']

**Range**

Range est une **built-in function** qui permet de créer une suite d'entier.  
Range(start number(inclus),end number(exclus))

for i in range(1,11):  
 print(i)  
>> 1

…

>> 9

>> 10

**While-Loop:**

Boucle qui s'exécute tant que l'expression renvoie **True**.

x = 3  
while x >0:  
 print("{}".format(x))  
 x -= 1  
Print("Partez!)  
>> 3  
>> 2  
>> 1  
>> Partez !  
Attention aux **infinite loop** (ctrl+c pour interrompre l'exécution)

**Break**:

C'est un **break-statement** qui vient interrompre une boucle

qs = ["nom ?", "prénom ?", "sexe ?)"] Liste de 3.

n = 0 Pour démarrer la liste à partir de 0

while True:

print("type q to quit")

a = input(qs[n])

if a == "q":

break

n = (n + 1) % 3 Astuce permettant de faire le tour de la liste sans tomber sur un IndexError en revenant à quand on a atteint la taille de la liste (ici 3)

**Continue**

C'est un **continue-statement** qui fait revenir au début de la boucle (fait une itération dans le cas d'un range mais pas naturellement dans le cas d'un while)

i = 0 for i in range (1, 6):

while i <= 5: if I == 3:

if i == 3: continue

continue print(i)

print(i) Dans ce cas itération du **range** par **continue**

i += 1

**Infinite** **loop** car pas de i+=1 dans **if**

**Nested Loops**

Consiste à intégrer une boucle dans une autre boucle (pas de limite d'intégration). Celui qui héberge s'appelle le **outer-loop** et celui hébergé s'appelle le **inner-loop.**

for nested in for

list1 = [1,2]

list2 = [2,3]

somme = []

for i in list1:

for j in list2:

somme.append(i+j)

print(somme)

>>[3, 4, 4, 5]

for nested in while

while input("y or n? ") != "n":

for i in range(1,3):

print(i)

>> y or n? y

>> 1

>> 2

>> y or n? y

>> 1

>> 2

>> y or n? n

**MODULES**

Pour importer un module 🡺 import module\_name  
Pour exécuter une fonction ou une variable du module 🡺 module\_name.function(parameters)

**Built\_in Modules:**

Exemple de modules préexistants avec leurs fonctions:

* math math.pow(2,3) >> 8.0
* random random.randint(0,100) >> 42
* statistics statistics.mean(list) >> Moyenne chiffres liste
* keywords keyword.iskeyword("for") >> True

**Importing other modules**

Pour importer un module de son cru, il faut qu'il soit enregistré dans le même fichier.

Attention, à l'import d'un module tout le code s'exécute

**FILES**

**Writing to Files**

**Reading** = Accéder aux données   
**Writing** = Ajouter ou modifier les données

Pour ouvrir un fichier il faut utiliser la **built-in function** **open** (**file path**, "**r**" ou "**w**" ou "**w+**")  
 **r = reading w = writing w+ = reading & writing**

Pour le **file** **path**, ne pas écrire le chemin d'accès soi-même, utiliser le **built-in** **module** **os** et utiliser la **function** **path.** En effet, tous les os n'utilisent pas le même langage pour accéder au fichier.

Ouvrir un fichier en **r** = lecture seule; pour **w** et **w+** cela écrasera ou créera le fichier existant.

Exemple

import os

path\_track = os.path.join("C:\\","Users","yannt","Desktop","PYTHON","st.txt")

st = open(path\_track,"w+")

st.write("Hi from Python!")

st.close()

Cette méthode implique d'ouvrir et de fermer le fichier.

**Automatically closing files**

Pour éviter d'avoir à penser à refermer le fichier, il faut utilise le **with-statement**   
Pour ce faire: **open**(file\_path,mode) **as** variable\_name:  
 code

import os

path\_track = os.path.join("C:\\","Users","yannt","Desktop","PYTHON","st.txt")

with open(path\_track,"w+") as fff:

fff.write("Salut nounou !!")

**Reading from files**

Pour lire un fichier il suffit de mettre le **mode** "r" puis d'appeler la **Method** **read**.

Attention, on ne peut appeler la sur **Method** **read** un fichier qu'une seule fois par ouverture 🡺 Sauvegarder son contenu dans une **variable** ou un **container**

import os

path\_track = os.path.join("C:\\","Users","yannt","Desktop","PYTHON","st.txt")

my\_list = []

with open(path\_track,"r") as fff:

my\_list.append(fff.read())  
print(my\_list)

>> 'Salut nounou !!'

**CSV Files** = Comma Separated Values)

Dans les fichiers CSV, une virgule = une cellule différente (décalage colonne) et une ligne représente une ligne (décalage ligne). Ces séparateurs s'appellent des **delimiter**.

Pour pouvoir utiliser le csv il faut importer le **module csv.** Ce module contient une **Method writer** et ce **writer** contient une **Method writerow** 🡺 Cette **Method** accepte une **list** comme **parameter** mais seulement une ligne à la fois.

import os

import csv

path\_track = os.path.join("C:\\","Users","yannt","Desktop","PYTHON","st.csv")

liste1 = ["one","two","three"]

liste2 = ["a","b","c"]

with open(path\_track,"w",newline='') as fff:

w = csv.writer(fff, delimiter=",")

w.writerow(liste1)

w.writerow(liste2)

Pour lire un fichier csv

import csv

with open(path\_track,"r") as fff: Ouvrir l'objet st.csv

w = csv.reader(fff, delimiter=",") Convertir l'objet en csv avec reader

for row in w: Boucle pour parcourir les lignes

print(",".join(row)) Pour l'avoir en string

print(row) Pour l'avoir en liste

Exercice:

import os

import csv

file = [

["Top Gun", "Risky Business", "Minority Report"],

["Titanic","The Revenant", "Inception"],

["Training Day", "Man on Fire", "Flight"]

]

file\_path = os.path.join("C:\\","Users","yannt","Desktop","PYTHON","monlapin.csv")

with open(file\_path, mode="w", newline='') as lapin:

fichier = csv.writer(lapin, delimiter=",")

for acteur in file:

fichier.writerow(acteur)

ASTUCES

**List Comprehension:**

Ca a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

b = [i for i in a if i % 2 == 0]

Correspond à ça

a = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

b = []

for i in a:

if i % 2 ==0:

b.append(i)

Pareil, ça :

a = [1,1,2,3,5,8,13,21,34,55,89,6,9]

b = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,19,54,55,56]

c = []

for i in a:

for j in b:

if j == i:

if j not in c:

c.append(j)

Correspond à ça:

a = [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13]

c = []

c = [j for i in a for j in b if j == i if j not in c]

def loop(x):

y = []

for i in x:

if i not in y:

y.append(i)

return y

def loop2(x):

y = []

[y.append(i) for i in x if i not in y]

return y

**Set**

a = [1,1,1,5,3,2,2,8,9,10]

d= set(a)

print(d)

>> {1, 2, 3, 5, 8, 9, 10}

Semble renvoyer un dictionnaire trié et sans doublon d'une liste. Passage de list a set très facilement.  
list(set(a)) set(list(a))

**Reversed**

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

y = list(reversed(x))

>> [5, 4, 3, 2, 1]

**if \_\_name\_\_ == '\_\_main\_\_'**

Tous les fichiers Python ont une variable appelée \_\_name\_\_ qui a pour valeur "\_\_main\_\_" lorsqu'il s'agit du script principal (le fichier qu'on a lancé) mais pas s'il s'agit d'un import.

Sans ce if avant l'exécution, un import lancerait la totalité du code inscrit dans l'import, alors qu'on ne le souhaite pas forcément.

**Object-oriented programming:**

Une **class** d'objet est un regroupement et une classification d'objets similaires.  
Chaque objet dans une classe est une **instance** de la **class**.

**class** [name]: [suites]

Par convention le [name] des classes commence par une majuscule et s'écrit en camelCase = ParExempleCommeCa.

Les [suites] sont des statement qu'on appelle **method**. Ces **method** sont comme des **functions** mais qui ne peuvent s'appliquer qu'aux objets de la **class.** Par convention les **method** s'écrivent comme les fonctions : minuscule\_séparées\_par\_des\_underscores

Ces method doivent obligatoirement (sauf exception) appeler au moins **un parameter**. Le premier **parameter** de la **method** s'appelle toujours self. Ceci est lié au fait que quand tu appelles une méthode sur un objet, python va automatiquement passer l'objet appelé par la **method** en **parameter** de cette **method.**

Une **instance variable** est une variable qui appartient à un objet. Pour créer des instance variable  
self.[variable\_name] = [variable\_value]  
Les **instance variables** se définissent dans une **method** \_\_init\_\_ (= intialize)

class Orange:

def \_\_init\_\_(self, w, c):

self.weight = w

self.color = c

print("Created!") ce sont les caractéristiques des objets de la classe Orange

les **method** entourées par deux underscore (\_\_init\_\_) sont des **magic method** une méthode utilisée par Python pour des buts spécifiques, comme créer un objet.

Créer un **object/instance** dans une **class** s'appelle = **instantiating** **a** **class**

Pour créer un objet dans une class:

or1 = Orange(10,"dark orange")

Il suffit d'indiquer les paramètres acceptés dans \_\_init\_\_. Pas la peine d'indiquer le paramètre self, puisqu'il s'agit de l'objet créé. En imprimant l'objet de la classe

print(or1)

<\_\_main\_\_.Orange object at 0x04127EB0>

Il va indiquer de quelle classe il appartient et son emplacement dans la mémoire  
Pour obtenir les valeurs de l'**instance variable** il suffit d'écrire :

>>> print(or1.weight)

10

>>> print(or1.color)

dark orange

Pour redéfinir la valeur d'une instance variable comme ceci :

>>> or1.weight = 11

Exemple de classe avec méthode

class Rectangle():

def \_\_init\_\_(self, w, l):

self.width = w

self.len = l

def area(self):

return self.width \* self.len

def change\_size(self, w, l):

self.width = w

self.len = l

rectangle = Rectangle(10,20)

print(rectangle.area())

rectangle.change\_size(20,40)

print(rectangle.area())

>> 200

>> 800

**4 Pillars of object-oriented programming**:

Encapsulation:

* Les **group variables** et les **method** sont toutes deux regroupées dans un seul élémet: **object**
* les données internes à une classe sont inaccessibles au **client** qui ne peut donc venir modifier les variables.

Dans Python, tout est public. Pour rendre inaccessibles les données dans une **method** d'une **class**, la convention implique de nommer les **private** **variables** ou **private** **method** avec un \_underscore en premier caractère.

Abstraction

Consiste simplement à réduire les caractéristiques d'un objet à ce qui est nécessaire, pas être exhaustif

Polymorphism

Capacité en programmation à utiliser la même interface qu'elle que soit le **data type** des variables  
Print marche quel que soit le type de donnée

Inheritance

Une **child class** hérite des method et variables contenues dans le **parent class.** Pour associer une **class** à une autre, il faut mettre en **paramètre** de la **class** le nom de la **parent class**

class Shape():

def \_\_init\_\_(self, w, l):

self.width = w

self.len = l

def print\_size(self):

print("{} by {}".format(self.width, self.len))

class Square(Shape):

def area(self):

return self.width \* self.len

carré = Square(20,20)

print(carré.area())

>> 400

Une **child class** peut outrepasser la **method** du **parent class** en nommant la nouvelle **method** exactement comme celle du **parent** **class** : c'est la **method** **overriding**

class Shape():

def \_\_init\_\_(self, w, l):

self.width = w

self.len = l

def print\_size(self):

print("{} by {}".format(self.width, self.len))

class Square(Shape):

def area(self):

return self.width \* self.len

def print\_size(self):

print("I am {} by {}".format(self.width, self.len))

carré = Square(20,20)

carré.print\_size()

🡺 Composition

La **composition** consiste à stocker un objet en tant que variable d'un autre objet

class Dog():

def \_\_init\_\_(self, name, breed, owner):

self.name = name

self.breed = breed

self.owner = owner

class Person():

def \_\_init\_\_(self, name):

self.name = name

mick = Person("Mick Jagger")

stan = Dog("Stanley", "Bulldog", mick)

print(stan.owner.name)

>> Mick Jagger

Grâce à cette **composition** l'objet stan de la class Dog a un propriétaire, un objet de la class Person stocké dans la **instance variable** owner de la class Dog

**Attention:**

* Bien mettre ces putains de parenthèses quand on appelle une méthode d'une classe

>>> carré.what\_am\_i

<bound method Shape.what\_am\_i of <\_\_main\_\_.Square object at 0x03F17ED0>>

>>> carré.what\_am\_i()

I am a shape

* Dans le cas d'une **composition** l'objet stocké dans l'instance variable doit être appelé dans sa source

jolly = Horse("Jolly Jumper")

patrick = Rider("Patoche le beau", jolly)

print(patrick.nom)

print(patrick.cheval)

print(patrick.cheval.nom)

>> Patoche le beau

>> <\_\_main\_\_.Horse object at 0x035D7F30>

>> Jolly Jumper

**More Object Oriented Programming**

Class Variables vs Instance Variables

Dans Python, les **class** sont des **objects**

Les **class** ont 2 types de variables:

* **Instance variables** = celles qui sont définies par self
* **Class variables** = variables définies à l'intérieur d'une **class** et accessibles via des objets de la class et via des objets créés avec un objet de class. On y accède comme pour les instances variables: avec le préfixe self

class Rectangle():

recs = [] class variable (liste)

def \_\_init\_\_(self, w, l):

self.width = w

self.len = l

self.recs.append((self.width,self.len)) ajout de tuples dans la liste à chaque création d'instances variables

r1 = Rectangle(10,20) création d'une instance variable qui appelle

r2 = Rectangle(20,40) la méthode \_\_init\_\_

r3 = Rectangle(40,80)

print(Rectangle.recs)

>> [(10, 20), (20, 40), (40, 80)]

Magic Method

Toutes les class dans Python sont en fait des child class d'une class appelée Object

Python utilise des **method** **inherited** de la class Object dans différentes situations.  
Par exemple quand on utilise la fonction print, il va utiliser la **method** appelée \_\_repr\_\_ issue de la class Object

class Lion:

def \_\_init\_\_(self,name):

self.name = name

lion = Lion("Mufasa")

print(lion)

>> <\_\_main\_\_.Lion object at 0x03D98530>

Par défaut il va imprimer l'objet son lieu de stockage dans la mémoire

class Lion:

def \_\_init\_\_(self,name):

self.name = name

def \_\_repr\_\_(self):

return self.name

lion = Lion("Mufasa")

print(lion)

>> Mufasa

Grâce à **l'override** de la **Magic Method** \_\_repr\_\_ on peut redéfinir le **return** de print

Même principe pour des opérateurs comme "+" qui appellent une **Magic Method** \_\_add\_\_

class AlwaysPositive:

def \_\_init\_\_(self,n):

self.number = n

def \_\_add\_\_(self, other):

return abs(self.number + other.number)

x = AlwaysPositive(-20)

y = AlwaysPositive(10)

print(x+y)

>> 10

Is

**Is** renvoi **True** si deux **objects** sont les mêmes objets et **False** dans le cas contraire

class Person:

bob = []

def \_\_init\_\_(self):

self.name = "Bob"

self.bob.append(self.name)

bob = Person()

same\_bob = bob

print(bob is same\_bob)

print(Person.bob)

>> True

>>['Bob']

another\_bob = Person()

print(bob is another\_bob)

print(Person.bob)

>> False

>> ['Bob', 'Bob']

L'objectif de **is** est de savoir si l'objet est le même, pas sa valeur. Dans le premier cas, le same\_bob est exactement le même objet que bob. Dans le second cas, bien que la valeur soit identique, l'objet est différent. D'où le True et le False

**BASH**

Il s'agit d'une **command-line interface** afin pour que l'os l'exécute.

le $ indique qu'il attend une commande

echo = print  
python3 = permet de lancer du python dans cette interface  
exit() = permet de quitter python  
history = permet de voir les dernières commandes tapées  
↑↓ = permet d'accéder aux dernières commandes tapées

Relative vs Absolute Paths

Les os sont faits de **directories** et de **files.** Directory = fichiers sur l'ordinateur qui correspond à une adresse où il existe dans l'os.Dans Bash on est dans un **directory** situé dans un **path** particulier.

En utilisant la commande pwd (=Print Working Directory) il imprime le directory dans lequel tu te trouves actuellement.

En science informatique, un "arbre" est un concept important qu'on appelle **data structure**. Il y a le **root** tout en haut et chaque dossier/branche peut avoir plusieurs sous-dossiers etc…

**Absolute path** : chemin d'accès direct de la forme /home/bernie le premier / représente le "root" et ensuite ça descend l'arbre au fur et à mesure

**Relative path** : au lieu de commencer par un / qui désigne le root, on part du fichier dans lequel on est actuellement. Ainsi en étant dans le fichier home, si je voulais rejoindre le fichier bernie, j'aurai simplement à taper bernie

En utilisant la commande cd permet de changer de **directory.**

La commande ls permet de voir les **directory** disponibles à ce niveau

On peut créer un directory en utilisant la commande mkdir (=make directory). Les noms de **directory** ne peuvent pas avoir d'espace

Le caractère ~ est un raccourci pour revenir au niveau /home/"user"

La commande suivante permet de revenir d'un niveau cd ..

La commande rmdir(=remove directory) permet de supprimer un **directory.**

Flags

Les **commands** ont un concept qui s'appelle **flags** qui permet de changer le comportement d'une commande. Les flags sont des booléens (True ou False). Par défaut ils sont False. Si tu ajouter un **flag** à une commande, il passe automatiquement en True. Pour passer un **flag** à True il faut mettre un (ou 2) "–" devant le nom du **flag.**

Par exemple, au niveau où l'on se trouve, taper ls – author permet de voir les directory au niveau actuel mais également qui les a créés et quand.

Hidden files

Beaucoup de fichiers sont cachés pour éviter qu'on les modifie  
Les fichiers cachés ont leur nom qui commence par un "."  
Pour voir les fichiers cachés il faut ajouter -a (=all) dans la commande ls.

La commande touch permet de créer un nouveau **file**.

Pipe

(La commande less permet d'avoir une édition texte)

Le caractère "|" est appelé un **pipe**. Ce pipe permet de passer l'output d'une commande en input d'une autre

Environmental variables

Les **environmental variables** sont des variables stockées dans l'os que les programmes peuvent utiliser pour récupérer des informations (nom du pc, programmes lancés, type d'os utilisés…)

Il est possible d'utiliser des **environmental variables** avec la syntax:  
export variable\_name = variable\_value

Pour faire référence à une variable créée il faut mettre le signe "$" devant

pinkrabbit@MSI:~$ export x=100  
pinkrabbit@MSI:~$ echo $x

Ces variables disparaissent une fois la fenêtre fermée. Pour les faire persister il faut les ajouter dans un **hidden file** appelé .profile dans le **home directory**. Saisir export x=100 dans le fichier texte .profile pour que la variable devienne persistante.

Users

Il existe plusieurs utilisateurs sur un ordinateur. Pour savoir quel utilisateur est actuellement en cours d'utilisation il faut taper la commande whoami

L'utilisateur disposant des accès administrateurs est le root user.  
Pour éviter de se logger en tant que root user quand on veut effectuer une commande, il faut faire précéder la commande à exécuter par sudo (=superuser do). S'il y a un mot de passe sur l'ordinateur, il va prompt pour saisir le mot de passe

**Regular Expressions**

Easter egg: taper ça dans Bash sort un poème  
python3 -c "import this"

A simple match

La command grep sert à chercher un mot dans un fichier, il y a donc 2 paramètres : le mot puis le fichier. Exemple: nous avons sorti le poème dans un texte appelé zen.txt  
grep Beautiful zen.txt  
> Beautiful is better than ugly

Les flags -i = ignore la casse et -o = imprimer uniquement le mot qui a matché la recherche.

Dans Python il u a un module appelé re qui contient toutes les **regular expressions.**

Dans le module re il y a une méthode appelée findall avec comme paramètres le terme recherché et la source

import re

l = "Beautiful is better than ugly"

matches = re.findall("Beautiful", l)

print(matches)

>> ['Beautiful']

Il est possible d'ignorer la casse en passant comme troisième paramètre re.IGNORECASE

Match Beginning and End

Dans Bash, il est possible de définir des règles de **match** en utilsant des caractères particuliers

Placer le caractère "^" avant le mot recherché permet de chercher les **match** en début de phrase  
Placer Le caractère "$" à la fin du mot recherché permet de chercher les **match** en fin de phrase

Dans Python, pour obtenir toutes les réponses et pas seulement la première, il faut ajouter en troisième paramètre re.MULTILINE

import re

zen = (le poème en question)

matches = re.findall("idea.$", zen,re.MULTILINE)

print(matches)

>> ['idea.' , 'idea.']

Match Multiple Characters

Pour chercher les **matchs** de plusieurs caractères possibles, il faut mettre ces différents caractères entre crochet, il va ainsi tester les différentes possibilités et sortir les différents **matchs**



Il a bien sorti en rouge les 2 possibilités qu'étaient two et too (o&w étant entre crochet)

Dans Python:

import re  
string = "Two too"  
m = re.findall("t[ow]o", string, re.IGNORECASE)  
print(m)  
>> ['Two', 'too']

Match Digits

Il est possible de faire **matcher** les chiffres via [[:digit:]] dans Bash



et via \d dans Python.

import re  
string = "123?34 hello"  
m = re.findall("\d", string, re.IGNORECASE)  
print(m)  
>> ['1','2', '3' '4','3','4']

Repetition

Le caractère "\*" signifie matcher le caractère précédent 0 ou + de fois.



Dans cet exemple il va matcher tous les mots qui contiennent "tw" et 0 ou + de "o"

Le caractère "." signifie n'importe quel caractère.



la **regular expression** \* est dit **greedy**, dans le sens où il va essayer de faire matcher matcher un maximum de texte possible.  
Afin de rendre le caractère **non-greedy** il faut (dans Python uniquement) rajouter le symbole "?" pour le faire matcher le moins de texte possible.

import re

t = "\_\_one\_\_ \_\_two\_\_"

found = re.findall("\_\_.\*?\_\_", t)

for match in found:

print(match)

>> \_\_one\_\_

>> \_\_two\_\_

Escaping

L'**escape** permet de contourner le sens "**regular expression"** du caractère par son sens textuel.  
Dans bash il faut précéder le caratère par \\



Pareil dans Python

import re

t = "i love $"

found = re.findall("\\$", t, re.IGNORECASE)

print(found)

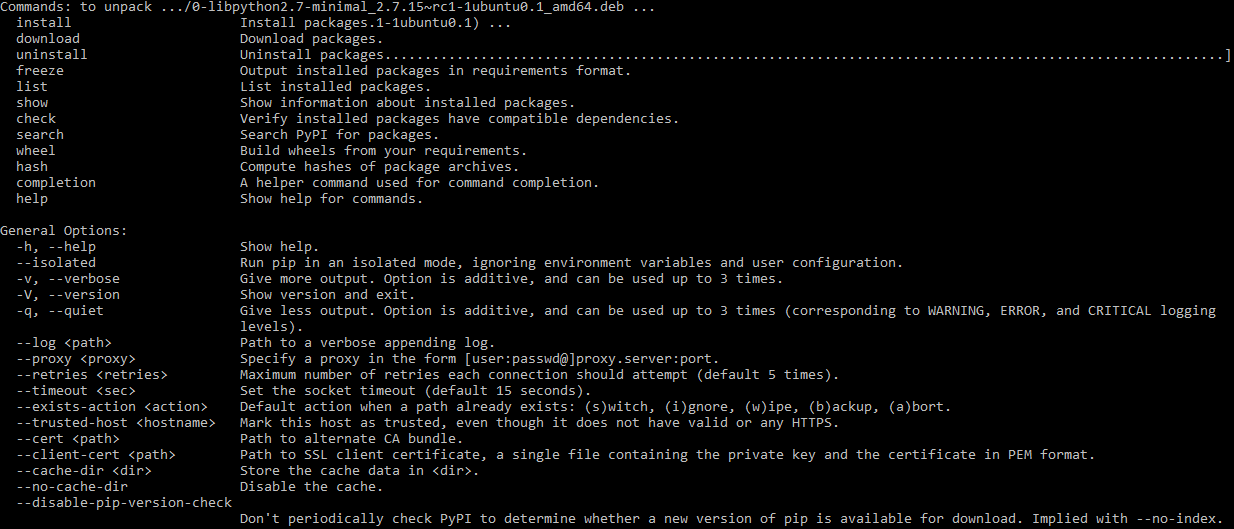
>> ['$']

**Package Manager**

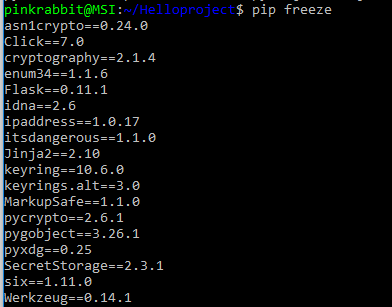
Un **package manager** est un programme qui installe et gère d'autres programmes. Par exemple les développeurs web utilisent souvent un **web framework** un programme qui aide à créer un site web.

NB: utiliser sudo apt update pour obtenir les packages disponibles dans Ubuntu.

Set de commandes sur Ubuntu



pip freeze permet de voir les packages installés



Il est préférable de travailler dans des environnements virtuels plutôt que d'installer tous les packages sur place. Cela permet d'avoir des packages séparés les uns des autres par programme.

Metadata = Data about data  
Dependencies: Les programmes dont un programme a besoin pour tourner correctement  
Apt-get = A package manager (Ubuntu)  
Pip: A Package manager for Python  
$PYTHONPATH = environmental variable where are stored modules where Python will look for

**Version Control**

<https://github.com/JohnBomber/hangman.git>

**Pushing** = Actualiser le **central repository** des changements faits dans le **local repository**

**Pulling** = Actualiser le **local repository** des changements faits dans le **central repository**



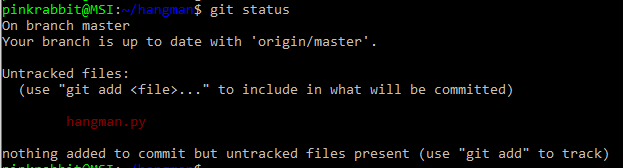
Première ligne = URL du **central repository** d'où le projet va pull des data  
Deuxième ligne = URL du **central repository** où le projet va push les data

Pushing example

On a ajouté dans le local repository, dans le fichier sur mon ordinateur, un document qui n'est pas dans le central repository. Pour le push il faut faire 3 étapes:

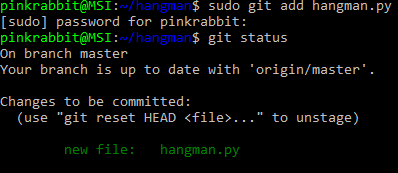
* **stage** your file = Indiquer à Git quel fichier modifié j'ai envie de **push** au **central repository**

La commande git status indique l'état du projet



Les fichiers unstage sont en rouge, les fichiers stage sont en vert

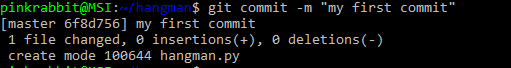
Pour ajouter le fichier taper : git add hangman.py



Pour le unstage taper git reset hangman.py

* **commiting** your files: Indiquer à Git d'enregistrer les changements faits

Pour faire ceci la commande est git commit -m [message]  
le -m est le **flag** qui signifie qu'on va ajouter un message au **commit** (important de bien documenter ce qu'on fait)



Une fois le **commit** fait on passe à l'étape 3:

* **Push** : Envoyer les changements dans le **central repository** avec la commande git push origin master

Pulling example

Pour pull un fichier il suffit d'entrer le code git pull origin master

Reverting Versions

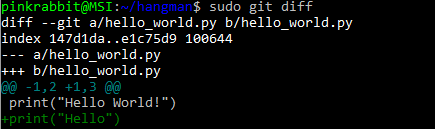
Git sauvegarde les projets à chaque **commit** de fichier. Chaque Chaque **commit** a un **commit number.**

Pour voir l'historique il suffit de taper la commande git log

Ensuite pour passer à la version du commit, il suffit d'entrer la commande git checkout numéro de commit

Diff

la commande git diff permet de voir les différences entre le **local** et le **central repository** dans les fichiers (et pas les fichiers en plus ou en moins entre le local et le central)



Les ajouts par rapport au central sont en vert et précédés du signe "+"

Les suppressions par rapport au central sont en rouge et précédés signe "-"