

BRUXELLES
FORMATION



former pour l'emploi

Introduction à



David Jelgersma



Cofinancé par
l'Union européenne

Objectif pédagogique :

Comprendre les concepts du langage Python pour vous permettre d'écrire un programme simple utilisant les mots clés du langage. On pose les bases pour les modules qui seront donnés ensuite (la "complexité" arrivera à ce moment-là) ; c'est-à-dire dans le contexte de l'analyse de données.

Au programme:

- Petit historique et quelques notions théoriques
- Installation de l'environnement de travail
- Le mode REPL et découverte des types principaux
- Contrôle de flux
- Fonctions
- Modules et (POO si on a le temps)





Petit historique et (vraiment) quelques notions théoriques

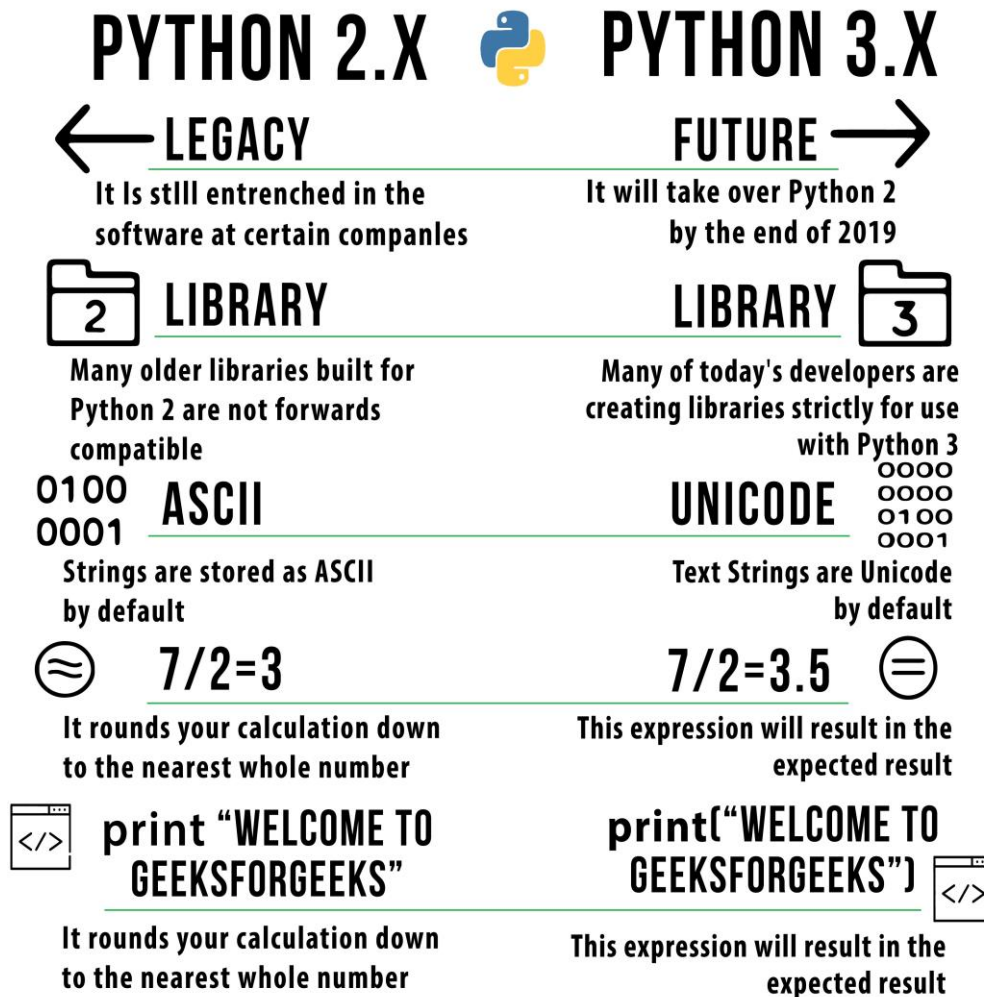


Introduction

- Qu'est-ce qu'un programme informatique (un script)?
- Langage de bas niveau vs haut niveau (scratch vs assembleur)
- Python dans tout ça ?
- Langage compilé vs interprété, Python est entre les deux
- Genèse : Guido van Rossum et les Monthly Pythons
- Pourquoi tant de succès ? (Applications (bureautique, multimédia, web, jeux ...), simplicité et productivité)
- Tout est objet (2 mots sur la POO)



Python 2 vs Python 3



Et beaucoup d'autres choses...

Python 2 : octobre 2000

...

Python 3 : décembre 2008

Python 2.7 : juillet 2010 ! (fin support janvier 2020)

Un petit mot sur les PEP et l'explication de cette situation.

Remarques :

<https://peps.python.org/>

<https://pep8.org/> (best practices)





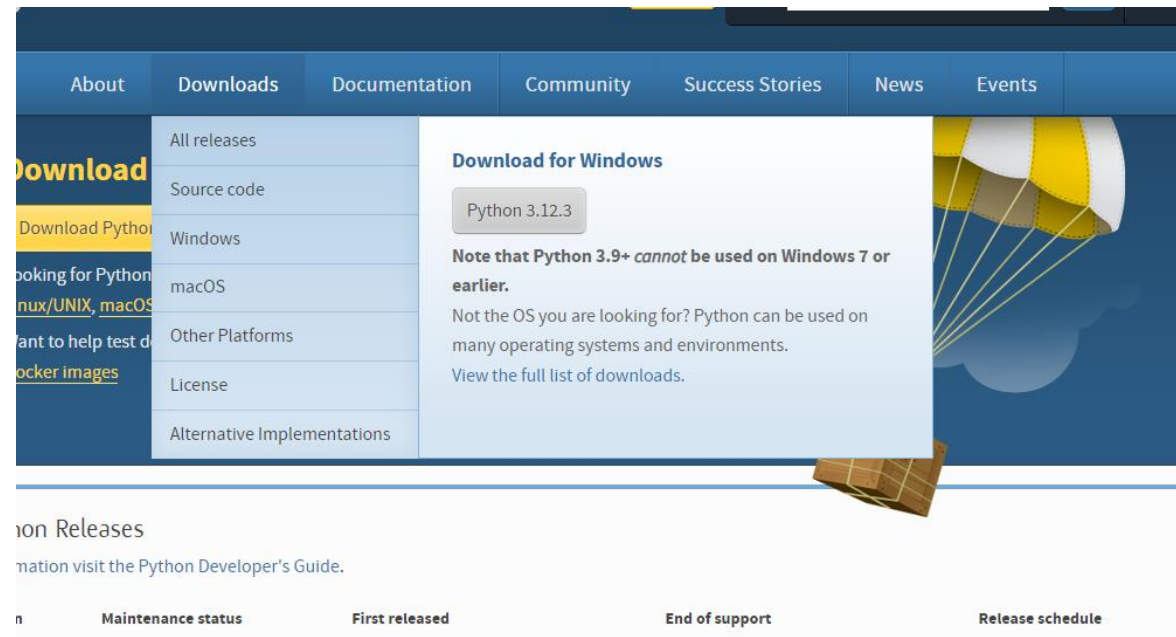
Installation et premiers contacts



Installation

- Modes d'installation (python.org, Anaconda, miniconda, ...)

- Installation de Python :
www.python.org



- Lancer Python, l'interpréteur Python ou REPL pour les intimes :
 - Python.exe



Zen of Python (PEP 20)

>>> import this

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```



Un petit « Hello World ! » pour commencer

- Pourquoi un "Hello World !" ???
- En Python, c'est très simple :

```
print("Hello world!")
```



REPL et découverte des premiers types de données



Trouver l'info

- La future histoire de votre vie en IT !
- Google, Python.org (documentation)
- d'autres ressources viendront en fin de cours.
- Introspection dans l'interpréteur REPL :
 - `help()` --> utilitaire d'aide,
`dir(obj)` et `help(obj)`
 - `Help('modules')` --> liste les modules
 - ex. découverte des fonctions dans une library mal documentée sur le net
- Remarque pour quitter REPL : `exit()`



Variables

- Exemple : $x = 3$
 - Attention c'est une affectation, ça n'a pas le sens mathématique.
 - Précisions sur le fonctionnement de l'affectation. Mais on y reviendra plus tard.
- Identificateurs (*nom de la variable*)
 - Conventions de nommage et mots réservés au langage
 - Notion d'expression (exemple simple $x = y + 3$)
- Toujours initialiser une variable (lui donner une valeur)
- Typage dynamique + typage fort (adaptation sémantique)



Calculatrice Python : Les types numériques

- `int` : type entier (précision infinie, limitée par la mémoire de l'ordinateur)
- `float` : type virgule flottante
- `complex` : type complexe (juste pour info)



Calculatrice Python : les fonctions prédéfinies sur les types numériques

- 4 opérations `+`, `-`, `*`, `/`
 - Division entière `//` et opérateur modulo (reste d'une division euclidienne) `%`
 - Fonction puissance x^y : `x**y` (x^2 : `x**2`)
 - Valeur absolue : `abs()`
 - Arrondi : `round()`
 - Conversion : `int()`, `float()` et Type : `type()`
 - remarque: `>>> help(4)`, par exemple, donne plus de fonctions (pour info)
- > Expérimentez dans REPL avec tout cela :

Conversion de types, différentes fonctions sur les types, opérations...



Les types booléens

- Type bool : True (vrai, 1) or False (faux,0)
- Valeurs considérées comme fausses :
 - `None` (data type `NoneType`)
 - `False`
 - `0` (in every numerical data type)
 - Empty strings, lists and tuples: `''`, `[]`, `()`
 - Empty dictionaries: `{}`
 - Empty sets `set()`
- Le reste est évalué à True



Les types booléens

- Opérateurs de comparaison : qui renvoient un bool

- Comparison of content: `==`, `<`, `>`, `<=`, `>=`, `!=`
- Comparison of object identity: `a is b`, `a is not b`
- And/or operator: `a and b`, `a or b`
- Chained comparison: `a <= x < b`, `a == b == c`, ...
- Negation: `not a`

```
if not (a==b) and (c<3):  
    pass
```

Ne fait rien!

- Pourquoi ? (notion de table de vérité)
- Testez dans REPL
- Ca sera surtout très utile dans le contrôle de flux
 - Les instruction conditionnelles et les répétitions



Tables de vérité

AND Truth Table

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

OR Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

XOR Truth Table

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

NOT Truth Table

A	B
0	1
1	0

Figure 1. Basic logic truth tables

Le type chaîne de caractères

- Type str
- Définition : `s = 'spam'` , `s = "spam"`
Multiline strings: `s = """spam"""`
- Utilisez `dir()` et `help()` pour découvrir les fonctions utilisables sur les chaînes de caractères.



Fonctions sur le type chaîne de caractères

String

```
<str> = <str>.strip()           # Strips all whitespace characters from both ends.  
<str> = <str>.strip('<chars>')   # Strips all passed characters from both ends.
```

```
<list> = <str>.split()           # Splits on one or more whitespace characters.  
<list> = <str>.split(sep=None, maxsplit=-1) # Splits on 'sep' str at most 'maxsplit' times.  
<list> = <str>.splitlines(keepends=False)  # Splits on \n,\r,\r\n. Keeps them if 'keepends'.  
<str> = <str>.join(<coll_of_strings>)      # Joins elements using string as separator.
```

```
<bool> = <sub_str> in <str>       # Checks if string contains a substring.  
<bool> = <str>.startswith(<sub_str>) # Pass tuple of strings for multiple options.  
<bool> = <str>.endswith(<sub_str>)  # Pass tuple of strings for multiple options.  
<int> = <str>.find(<sub_str>)       # Returns start index of first match or -1.  
<int> = <str>.index(<sub_str>)      # Same but raises ValueError if missing.
```

```
<str> = <str>.replace(old, new [, count]) # Replaces 'old' with 'new' at most 'count' times.  
<str> = <str>.translate(<table>)         # Use `str.maketrans(<dict>)` to generate table.
```

```
<str> = chr(<int>)                 # Converts int to Unicode char.  
<int> = ord(<str>)                 # Converts Unicode char to int.
```

- Also: `'lstrip()'`, `'rstrip()'`.
- Also: `'lower()'`, `'upper()'`, `'capitalize()'` and `'title()'`.



Le type chaîne de caractères : les séquences d'échappement

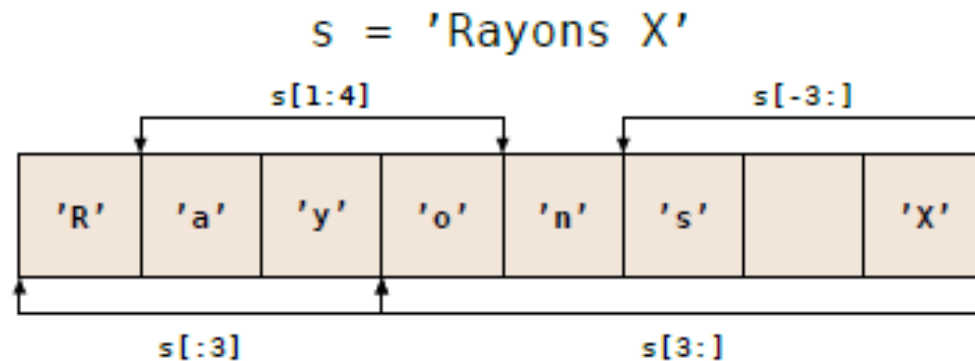
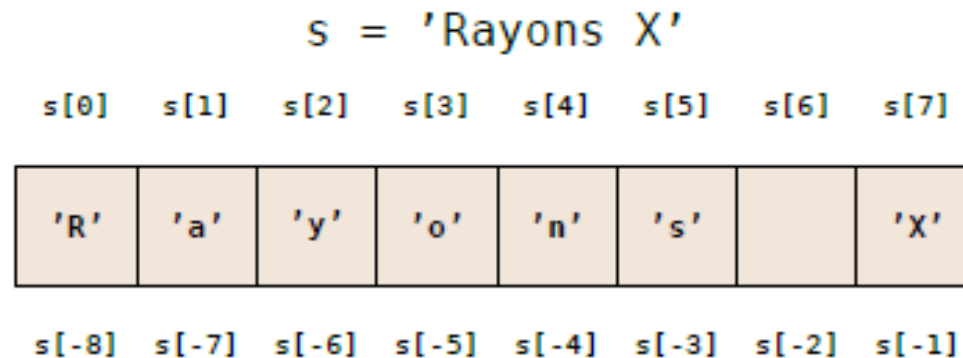
- Ceci permet d'insérer certains caractères spéciaux dans une chaîne :
 - Le cas d'école étant d'insérer un guillemet simple ou double

Séquence	Signification
<code>\saut_ligne</code>	saut de ligne ignoré (en fin de ligne)
<code>\\</code>	affiche un antislash
<code>\'</code>	apostrophe
<code>\"</code>	guillemet
<code>\a</code>	sonnerie (<i>bip</i>)
<code>\b</code>	retour arrière
<code>\f</code>	saut de page
<code>\n</code>	saut de ligne
<code>\r</code>	retour en début de ligne
<code>\t</code>	tabulation horizontale
<code>\v</code>	tabulation verticale
<code>\N{nom}</code>	caractère sous forme de code Unicode nommé
<code>\uhhhh</code>	caractère sous forme de code Unicode 16 bits
<code>\Uhhhhhhhh</code>	caractère sous forme de code Unicode 32 bits
<code>\ooo</code>	caractère sous forme de code octal
<code>\xhh</code>	caractère sous forme de code hexadécimal



Le type chaîne de caractères : indexation

Slicing : `s[début: fin(exclus): pas(optionnel)]`



Faites quelques tests



Entrées et sorties simples

- Les entrées : inputs
- Il s'agit d'affecter à une variable une saisie au clavier :

```
>>> f1 = input("Entrez un flottant : ")
Entrez un flottant : 12.345
>>> type(f1)
<class 'str'>
>>> f2 = float(input("Entrez un autre flottant : "))
Entrez un autre flottant : 12.345
>>> type(f2)
<class 'float'>
```

```
>>> i = input("Entrez un entier : ")
Entrez un entier : 3
>>> i
'3'
>>> iplus = int(input("Entrez un entier : ")) + 1
Entrez un entier : 3
>>> iplus
4
>>> ibug = input("Entrez un entier : ") + 1
Entrez un entier : 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```



Entrées et sorties simples

- Les sorties : outputs
- Il s'agit d'afficher une variable ou un littéral :

```
>>> a, b = 2, 5
>>> print(a, b)
2 5
>>> print("Somme :", a + b)
Somme : 7
>>> print(a - b, "est la différence")
-3 est la différence
>>> print("Le produit de", a, "par", b, "vaut :", a * b)
Le produit de 2 par 5 vaut : 10
>>> print("On a <", 2**32, a*b, "> cas!", sep="~~~")
On a <~~~4294967296~~~10~~~> cas !
>>> # pour afficher autre chose qu'un espace en fin de ligne :
>>> print(a, end="@")
2@>>> print()
```





Installation VS Code



Intro

- Les IDE (*Integrated Development Environment*)

Environnement de développement intégré

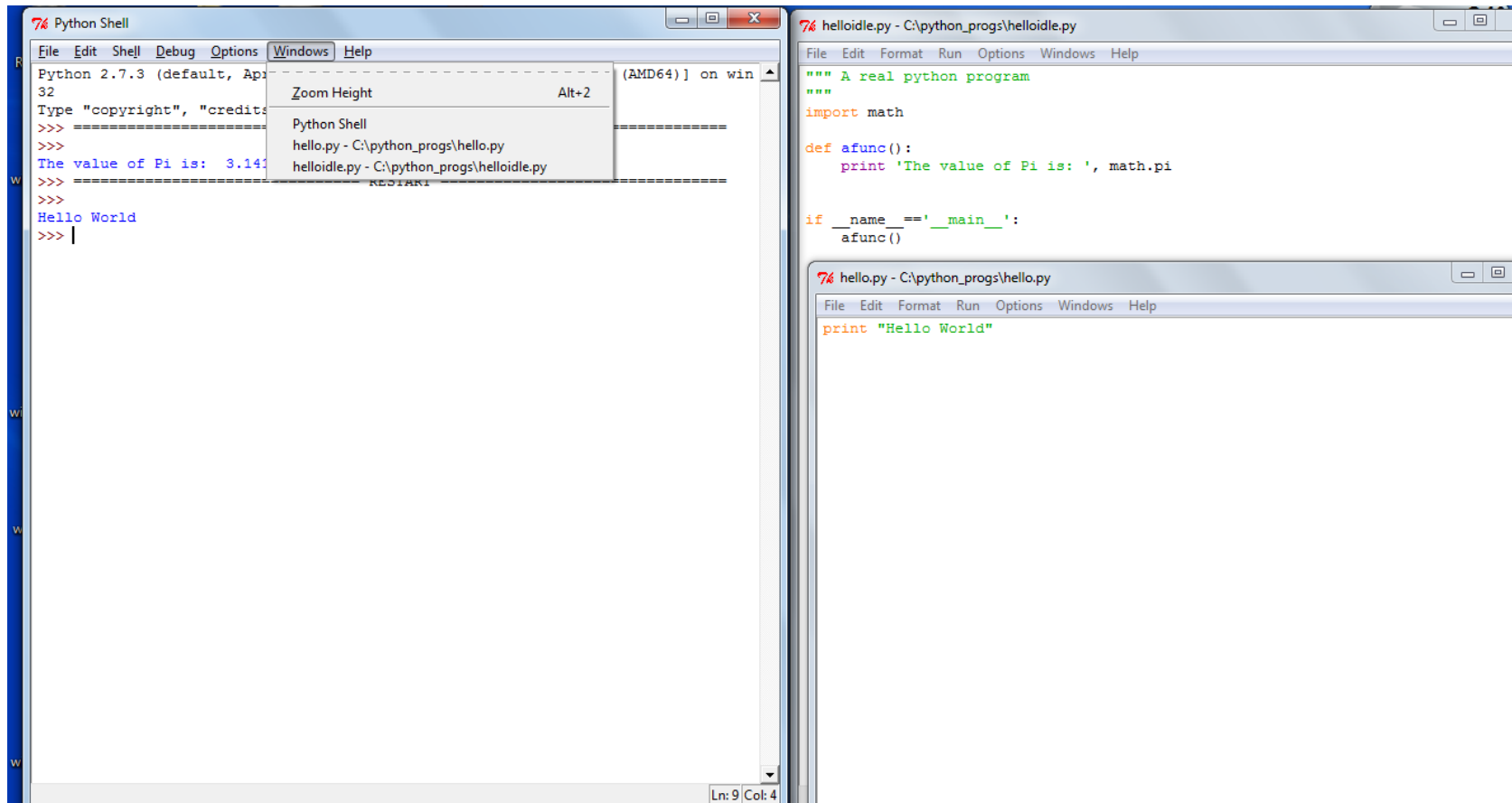
Logiciel qui facilite la conception d'applications qui rassemble tous les outils de développement, à savoir :

- Un éditeur de code
- Un débogueur
- Un gestionnaire de fichiers/architecture du projet
- Un/des compilateur(s)
- L'intégrations d'extensions tierces



Exemple d'un IDE très basique

- IDLE (*Integrated DeveLopment Environment*)

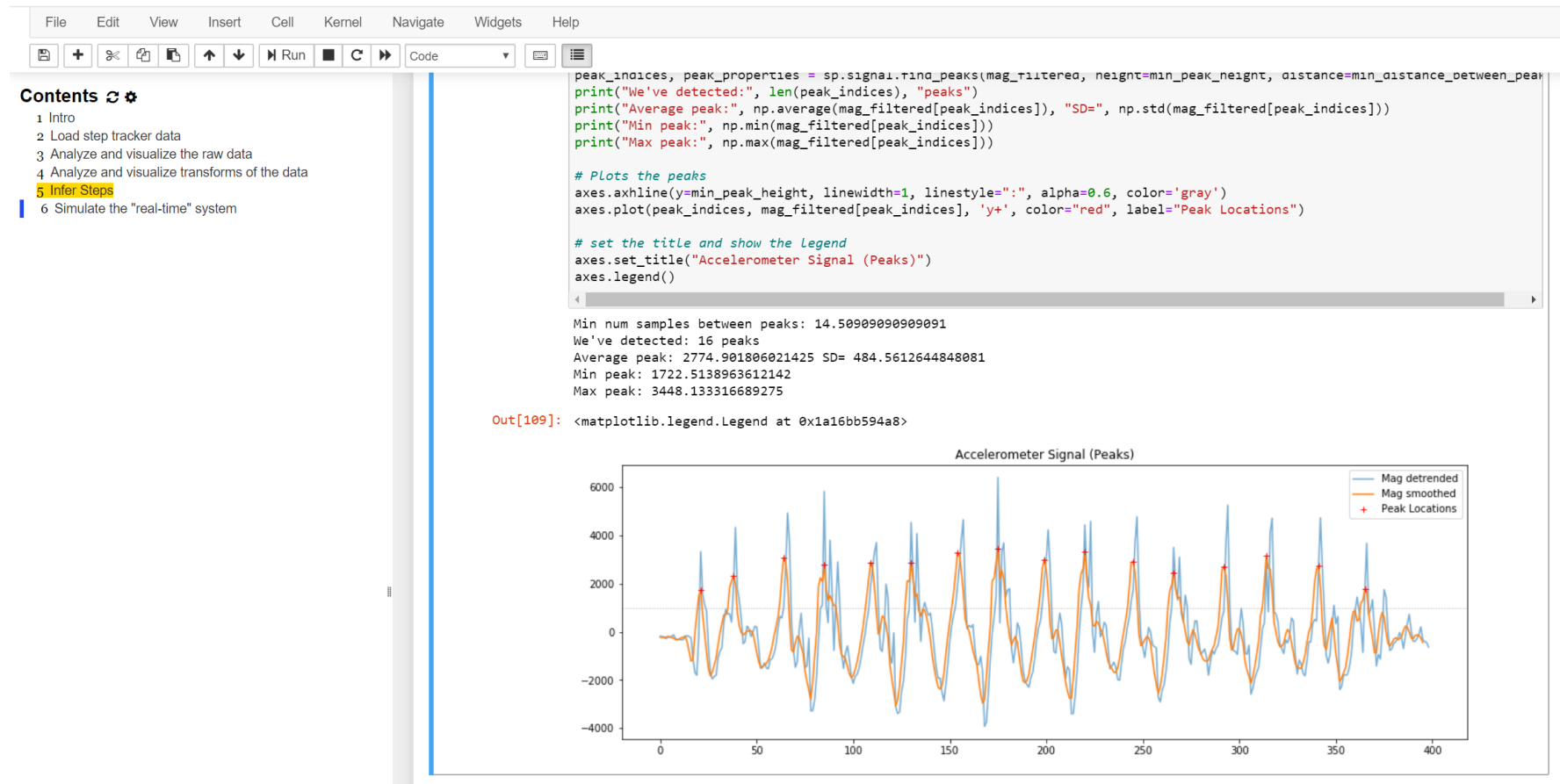


Une autre approche

- Jupyter (un REPL survitaminé!)



Jupyter Step Tracker Analysis Last Checkpoint: a few seconds ago (unsaved changes)



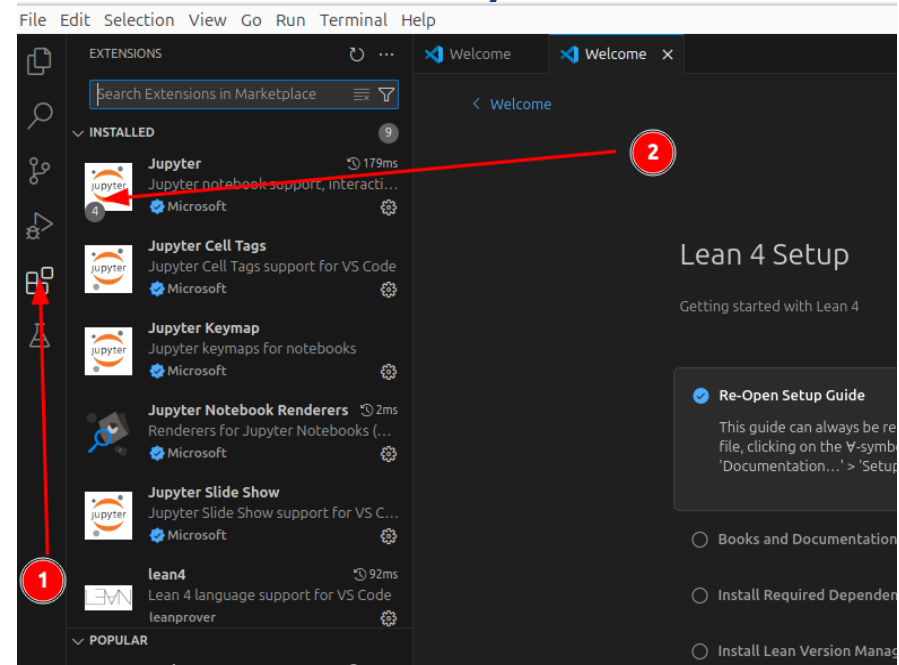
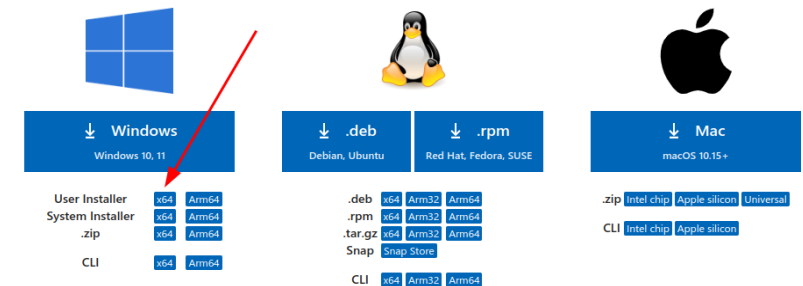
former pour l'emploi

Installation et configuration VS Code

Download Visual Studio Code

Free and built on open source. Integrated Git, debugging and extensions.

- Rendez-vous sur la page : code.visualstudio.com/Download
- Installation VS Code
- Lancement
- Installation des extensions (qu'est-ce donc cela?)
 - Python (Description de l'extension)



former pour l'emploi



Remarques préliminaires

... avant de créer des programmes (scripts).



Structurer son code source

- Avant même de regarder le code source, l'architecture d'une application est organisée en fichiers et répertoires. Ils sont organisés selon un framework (django), design pattern (MVC) et les principes métiers.
- Le code est subdivisé, découpé, en blocs logiques :
Chaque partie de code devrait accomplir une tâche spécifique.
- Le code source est commenté à bon escient : pas de commentaires « évidents » ou trop lacunaires.
- Le code se veut réutilisable : On ne le duplique pas. Et on (évite) de réinventer la roue, en utilisant des modules tiers (attention cependant à faire un choix prudent).
- Le code source inclut des tests unitaires permettant de minimiser les bugs éventuels.

Indentation

- Définition (remarque : VS Code = un tab vaut 4 espaces)
- Permet de créer des blocs composés d'une suite d'instructions.
 - Exemples (tout ceci deviendra plus clair dans les slides suivants et à l'usage) :

```
#_Exemples_d'instruction_composée_simple
#_(les_espaces_sont_indiqués_par_un_caractère_spécifique)_ :

ph_=6.0

if_ph_<7.0:
    print("C'est un acide.")

if_ph_>7.0:
    print("C'est une base.")
```

```
#_Exemple_d'instruction_composée_imbriquée_ :
n_=3

if_n_<=0:
    print('"n" est négatif ou nul')
else :
    print('"n" est positif')
    if_n_>2:
        print('"n" est supérieur à 2')
```



Documenter le code

```
def hello_world():  
    # A simple comment preceding a simple print statement  
    print("Hello World")
```

```
class SimpleClass:  
    """Class docstrings go here."""  
  
    def say_hello(self, name: str):  
        """Class method docstrings go here."""  
  
        print(f'Hello {name}')
```

- <https://realpython.com/documenting-python-code/>



Définir une fonction

```
def add(a, b):  
    """Returns the sum of a and b."""  
  
    mysum = a + b  
    return mysum
```

```
result = add(3, 5)  
print(result)
```

- A tester dans VS Code
- A tester dans REPL et faire un `help(add)`



Types de base - Exercices

- 1) Écrire un programme, qui définit 3 variables : une variable de type texte, une variable de type nombre entier, une variable de type nombre décimal et qui affiche leur type.
- 2) Écrire un programme qui affiche le type du résultat des instructions suivantes :

`a=3`

`a==3`

- 3) Écrire un programme qui inverse une chaîne de caractères en utilisant le slicing.
- 4) Écrire un programme qui demande un nombre supérieur à 100 et puis un nombre inférieur à 10. Puis qui donne combien de fois le petit nombre peut "rentre" dans le plus grand nombre.
- 5) Écrire un programme qui affiche le volume d'une sphère après avoir demandé son rayon.





Contrôle de flux



IF – ELIF – ELSE (choisir)

```
>>> x = 5
>>> if x < 0:
...     print("x est négatif")
... elif x % 2 != 0:
...     print("x est positif et impair")
...     print ("ce qui est bien aussi!")
... else:
...     print("x n'est pas négatif et est pair")
...
x est positif et impair
ce qui est bien aussi!
```

- Décryptage :
 - Initialisation de la variable de condition (ne pas oublier)
 - « if – elif – else » valide une condition vraie (un booléen), une expression conditionnelle, et continue l'exécution de son bloc d'instructions (attention à l'indentation!)
 - elif et else sont facultatifs, il peut y avoir plusieurs elif.



IF – ELIF – ELSE (choisir) – opérateur ternaire

```
>>> plus_petit = x if x < y else y # utilisation de l'opérateur ternaire
>>> print("Plus petit : ", plus_petit)
Plus petit : 3
```

- Décryptage :
 - L'opérateur ternaire est une *expression* qui fournit une valeur que l'on peut utiliser dans une affectation ou un calcul.



IF – ELIF – ELSE - Exercices

- 1) Ecrire un programme qui demande 2 entiers et affiche la valeur du plus petit.
- 2) Ecrire un programme qui convertit les euros en dollars et vice versa.
- 3) Ecrire un programme qui calcule l'aire d'un cercle ou d'un triangle et l'affiche. Le programme demande donc, en fonction de la figure, les infos nécessaires au calcul.



While (boucle/loop/répétition)

```
# Fibonacci series:  
# the sum of two elements defines the next  
a, b = 0, 1  
while a < 10:  
    print(a)  
    a, b = b, a+b
```

- Décryptage :
 - Initialisation de la variable de condition (ne pas oublier)
 - « while » exécute la suite d'instruction tant qu'une condition est vraie (un booléen)
 - Attention à l'indentation



While - Exercices

- Créer un programme qui demande un nombre entier et stoppe dès que sa valeur est entre 8 et 10 et l'affiche. (par exemple)
- A l'aide d'une boucle while créez un programme qui demande un nombre et l'additionne au nombre précédent tant que la somme est inférieure à 25 (au choix). Le résultat de l'addition est indiqué à chaque itération.
- Créez une variable à deviner et fixez sa valeur. A l'aide d'une boucle while demandez à l'utilisateur d'entrer un nombre : tant que la valeur entrée n'est pas égale au nombre à deviner indiquez si la valeur est trop grande ou trop petite et demandez d'entrer un autre nombre. Si la valeur entrée est correcte afficher un message. Option : indiquez le nombre d'essais.



For – parcours d'un itérable

- Itérable : à chaque fois que l'on fait appel à lui il renvoie la valeur suivante (itération)
- range(début, fin, pas) : un exemple d'itérateur

```
for i in range(10):  
    print(i)      # 0, 1, 2, 3, ..., 9  
  
for i in range(3, 10):  
    print(i)      # 3, 4, 5, ..., 9  
  
for i in range(0, 10, 2):  
    print(i)      # 0, 2, 4, 6, 8  
else:  
    print("Loop completed.")
```



For – parcours d'un itérable (suite)

- On rencontrera d'autres itérables à l'usage :
--> tuples, listes, sets, dictionnaires et strings...
- Une liste est un autre exemple (on en parle juste après)

```
for item in ["spam", "eggs", "bacon"]:  
    print(item)
```

- Une chaîne de caractères aussi :

```
for lettre in "ciao":  
    print(lettre)
```

- Testez à votre aise...



Break et Continue (concerne FOR et WHILE)

- Quand « break » s'exécute il interrompt complètement la boucle en cours. Exemple :

```
>>> for x in range(1, 11):  
...     if x == 5:  
...         break  
...     print(x, end=" ")  
...  
1 2 3 4  
>>> print("Boucle interrompue pour x =", x)  
Boucle interrompue pour x = 5
```

- Quand « continue » s'exécute il interrompt l'itération en cours et continue à la suivante. Exemple :

```
>>> for x in range(1, 11):  
...     if x == 5:  
...         continue  
...     print(x, end=" ")  
...  
1 2 3 4 6 7 8 9 10  
>>> # la boucle a sauté la valeur 5
```



For - Exercice

- Créer un programme qui affiche 3 fois votre prénom. Variante : qui demande combien de fois l'afficher.
- Créer un programme qui calcule la table de multiplications d'un nombre saisi par l'utilisateur.



Structures de données



Listes – type de données *list*

- Définition

```
couleurs = ['trèfle', 'carreau', 'coeur', 'pique']  
print(couleurs) # ['trèfle', 'carreau', 'coeur', 'pique']  
couleurs[1] = 14  
print(couleurs) # ['trèfle', 14, 'coeur', 'pique']  
list1 = ['a', 'b']  
list2 = [4, 2.718]  
list3 = [list1, list2] # liste de listes  
print(list3) # [['a', 'b'], [4, 2.718]]
```

- Test d'appartenance :

```
>>> 'trèfle' in couleurs  
True
```



Listes – quelques méthodes et fonctions

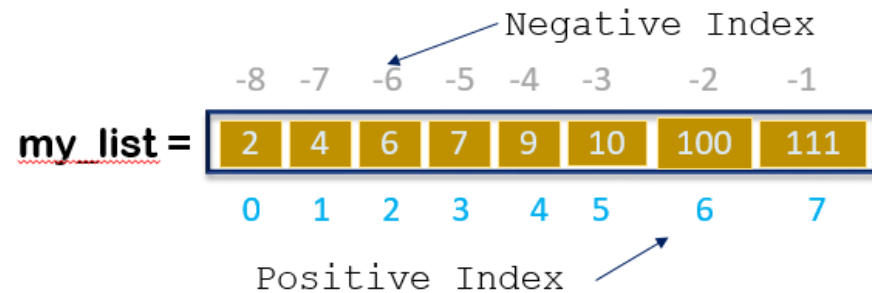
- `s = [1, "spam", 9.0, 42]` , `s = []`
- **Append an element:** `s.append(x)`
- Extend with a second list: `s.extend(s2)`
- Count appearance of an element: `s.count(x)`
- Position of an element: `s.index(x[, min[, max]])`
- Insert element at position: `s.insert(i, x)`
- Remove and return element at position: `s.pop([i])`
- **Delete element:** `s.remove(x)`
- Reverse list: `s.reverse()`
- **Sort:** `s.sort([cmp[, key[, reverse]])`
- Sum of the elements: `sum(s)`



Listes : Manipulation des « tranches » (Slicing) et indexation

- On fonctionne comme pour les chaînes de caractères

Index in Python



```
>>> mots = ['jambon', 'sel', 'miel', 'confiture', 'beurre']
>>> mots[2:4] = [] # effacement par affectation d'une liste vide
>>> mots
['jambon', 'sel', 'beurre']
>>> mots[1:3] = ['salade']
>>> mots
['jambon', 'salade']
>>> mots[1:] = ['mayonnaise', 'poulet', 'tomate']
>>> mots
['jambon', 'mayonnaise', 'poulet', 'tomate']
>>> mots[2:2] = ['miel'] # insertion en 3è position
>>> mots
['jambon', 'mayonnaise', 'miel', 'poulet', 'tomate']
```

- Bien tester (ce doit bien être compris!)



Listes – Définition par compréhension

En première approche pour construire une liste comprenant les carrés des nombres de 0 à 9 :

```
a = []  
for i in range(10):  
    a.append(i**2)
```

Même chose en compréhension :

```
a = [i**2 for i in range(10)]
```

On peut rajouter une condition si nécessaire :

```
a = [i**2 for i in range(10) if i != 4]
```



Listes : exercice

- Dans les deux exercices suivants, la liste suivante sera utilisée :
 - — lapin
 - — chat
 - — chien
 - — chiot
 - — dragon
 - — ornithorynque
- Ecrire un programme qui crée la liste des animaux et l'affiche à l'écran
- Ecrire un programme qui, pour chaque nom d'animal, l'affiche et affiche le nombre de caractères de son nom.
- Ecrire un programme qui renvoie le plus petit élément d'une liste de chiffres.
- Ecrire la définition par compréhension d'une liste qui contient des entiers pairs de 1 à 20.



Tuples – type de données *tuple*

- Un tuple est une collection ordonnée et non modifiable d'éléments éventuellement hétérogènes.
- Définition simple :
 - Mon_tuple = (1,8,7)

- `s = 1, "spam", 9.0, 42`

- `s = (1, "spam", 9.0, 42)`

- Constant list

- Count appearance of an element: `s.count(x)`

- Position of an element: `s.index(x[, min[, max]])`

- Sum of the elements: `sum(s)`



Tuples : exercice

- Créer un tuple qui contient 5 pays et l'affiche. Demander à l'utilisateur de choisir un pays et d'afficher son index, sa position dans le tuple (`tuple.index("pays")`). Enfin demander une position dans la liste et demandez d'afficher le pays correspondant.



Tuples, listes et chaînes de caractères sont des séquences. Quelques remarques :

Strings, lists and tuples have much in common: They are **sequences**.

- Does/doesn't s contain an element?

```
x in s, x not in s
```

- **Concatenate sequences:** `s + t`

- Multiply sequences: `n * s`, `s * n`

- **i-th element:** `s[i]`, i-th to last element: `s[-i]`

- Subsequence (slice): `s[i:j]`, with step size k: `s[i:j:k]`

- Subsequence (slice) from beginning/to end: `s[:-i]`, `s[i:]`, `s[:]`

- **Length** (number of elements): `len(s)`

- **Smallest/largest element:** `min(s)`, `max(s)`

- Assignments: `(a, b, c) = s`

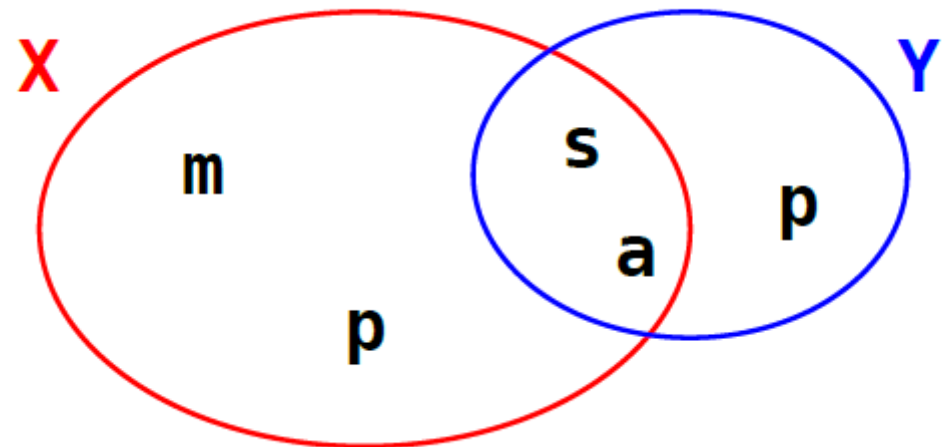
→ `a = s[0]`, `b = s[1]`, `c = s[2]`



Ensembles - type de données *sets*

- Ensemble non ordonné, pas de doubles, itérable

```
>>> X = set('spam')
>>> Y = set('pass')
>>> X
{'s', 'p', 'm', 'a'}
>>> Y # pas de duplication : qu'un seul 's'
{'s', 'p', 'a'}
>>> 'p' in X
True
>>> 'm' in Y
False
>>> X - Y # ensemble des éléments de X qui ne sont pas dans Y
{'m'}
>>> Y - X # ensemble des éléments de Y qui ne sont pas dans X
set()
>>> X ^ Y # ensemble des éléments qui sont soit dans X soit dans Y
{'m'}
>>> X | Y # union
{'s', 'p', 'm', 'a'}
>>> X & Y # intersection
{'s', 'p', 'a'}
```



Ensembles – quelques remarques

- `s = {"a", "b", "c"}`
alternative `s = set([sequence])`, required for empty sets.
- **Constant set:** `s = frozenset([sequence])`
e.g. empty set: `empty = frozenset()`
- **Subset:** `s.issubset(t)`, `s <= t`, strict subset: `s < t`
- **Superset:** `s.issuperset(t)`, `s >= t`, strict superset: `s > t`
- **Union:** `s.union(t)`, `s | t`
- **Intersection:** `s.intersection(t)`, `s & t`
- **Difference:** `s.difference(t)`, `s - t`
- **Symmetric Difference:** `s.symmetric_difference(t)`, `s ^ t`
- **Copy:** `s.copy()`

As with sequences, the following works:

```
x in s, len(s), for x in s, s.add(x), s.remove(x)
```




Ensembles : exercices

- Ecrire un programme qui affiche le contenu (en itérant) d'un ensemble (que vous aurez défini ou que l'utilisateur créera avec des input)
- Faire un programme qui demande un mot et affiche le nombre de voyelles.



Dictionnaires – type de données *dict*

Syntaxe

 Collection de couples *cle* : *valeur* entourée d'accolades.

Les dictionnaires constituent un type composite mais ils n'appartiennent pas aux séquences.

Les dictionnaires sont *modifiables* mais *non-ordonnés* : les couples enregistrés n'occupent pas un ordre immuable, leur emplacement est géré par un algorithme spécifique (algorithme de *hash*). Le caractère non-ordonné des dictionnaire est le prix à payer pour leur rapidité !

Une *clé* pourra être alphabétique, numérique... en fait tout type *hachable* (donc liste et dictionnaire exclus). Les *valeurs* pourront être de tout type sans exclusion.

```
>>> store = { "spam": 1, "eggs": 17}
>>> store["eggs"]
17
>>> store["bacon"] = 42
>>> store
{'eggs': 17, 'bacon': 42, 'spam': 1}
```

On peut itérer sur un dictionnaire, par exemple en affichant les paires *clé : valeur*

```
for key in store:
    print(key, store[key])
```



Dictionnaires – exemples

```
>>> d1 = {}      # dictionnaire vide. Autre notation : d1 = dict()
>>> d1["nom"] = 3
>>> d1["taille"] = 176
>>> d1
{'nom': 3, 'taille': 176}
>>>
>>> d2 = {"nom": 3, "taille": 176}  # définition en extension
>>> d2
{'nom': 3, 'taille': 176}
>>>
>>> d3 = {x: x**2 for x in (2, 4, 6)} # définition en compréhension
>>> d3
{2: 4, 4: 16, 6: 36}
>>>
>>> d4 = dict(nom=3, taille=176)  # utilisation de paramètres nommés
>>> d4
{'taille': 176, 'nom': 3}
>>>
>>> d5 = dict([("nom", 3), ("taille", 176)]) # utilisation d'une liste de couples clés/valeurs
>>> d5
{'nom': 3, 'taille': 176}
```



Dictionnaires – fonctions et méthodes

- **Delete an entry:** `del(store[key])`
- **Delete all entries:** `store.clear()`
- **Copy:** `store.copy()`
- **Does it contain a key?** `key in store`
- **Get an entry:** `store.get(key[, default])`
- **Remove and return entry:** `store.pop(key[, default])`
- **Remove and return arbitrary entry:** `store.popitem()`



Dictionnaires : exercice

- Créez un programme qui demande à l'utilisateur ses 4 plats préférés et les stocke dans un dictionnaire qui commence à l'index 1. Afficher ce dictionnaire. Finalement demander à l'utilisateur le plat qu'il aime le moins parmi les 4 affichés, le supprimer et afficher le nouveau dictionnaire (optionnel : trié!).





Les fonctions



Définir une fonction

```
def add(a, b):  
    """Returns the sum of a and b."""  
  
    mysum = a + b  
    return mysum
```

```
result = add(3, 5)  
print(result)
```

- A tester dans VS Code
- A tester dans REPL et faire un `help(add)`



Fonctions : retour de plusieurs valeurs

```
def foo():  
    a = 17  
    b = 42  
    return (a, b)  
  
ret = foo()  
(x, y) = foo()
```

Les valeurs retournées le sont à l'aide de tuples ou de listes

Fonctions : arguments par défaut

```
def fline(x, m=1, b=0): #  $f(x) = m*x + b$   
    return m*x + b  
  
for i in range(5):  
    print(fline(i), end=" ")  
#force newline  
print()  
for i in range(5):  
    print(fline(i, -1, 1), end=" ")
```

Attention : il n'est pas autorisé de définir les valeur par défaut avant les autres paramètres



Remarque (importante) : la portée des variables (objets)

- La portée d'une variable fait référence à la partie du code où cette variable est accessible ou visible. La portée des variables est déterminée par le bloc dans lequel elles sont définies.
 - **Portée locale** : Une variable définie à l'intérieur d'une fonction est dite locale à cette fonction. Elle n'est accessible que depuis cette fonction et n'existe pas en dehors de celle-ci.
 - **Masquage de variables** : Si une variable locale dans une fonction a le même nom qu'une variable globale, la variable locale masque la variable globale à l'intérieur de cette fonction

```
def ma_fonction():  
    x = 10  
    print("La valeur de x à l'intérieur de la fonction :", x)  
  
ma_fonction()  
# print(x) # Cela générera une erreur car x n'est pas accessible
```

```
a = 5  
  
def ma_fonction():  
    a = 10  
    print("La valeur de a à l'intérieur de la fonction :", a)  
  
ma_fonction()  
print("La valeur de a en dehors de la fonction :", a)
```



Remarque (importante) : la portée des variables (objets)

- **Portée globale** : Une variable définie en dehors de toutes les fonctions est dite globale. Elle est accessible de n'importe où dans le script, y compris à l'intérieur des fonctions.

```
y = 20

def ma_fonction():
    print("La valeur de y à l'intérieur de la fonction :", y)

ma_fonction()
print("La valeur de y en dehors de la fonction :", y)
```

- **Portée de bloc** : En Python, les blocs tels que les boucles for, les boucles while et les blocs if/else n'introduisent pas de nouvelles portées. Les variables définies à l'intérieur de ces blocs sont également accessibles en dehors d'eux.

```
for i in range(5):
    z = i * 2
print("La valeur de z en dehors de la boucle :", z)
```

- **Utilisation de la déclaration 'global'** : Si vous souhaitez modifier une variable globale à l'intérieur d'une fonction, vous devez utiliser la déclaration 'global' pour indiquer à Python que vous faites référence à la variable globale.

```
def increment():
    global count
    count += 1
```



Fonctions : exercices

- Ecrire un programme qui contient 2 fonctions qui testent si un nombre est pair ou impair. Ce programme demande un nombre à l'utilisateur et affiche s'il est pair ou impair.
- Ecrire un programme qui demande une chaîne de caractères à l'utilisateur et affiche s'il s'agit d'un palindrome. Merci d'utiliser une fonction qui **renvoie un bool**. La commande `list(chaine)` permet de transformer une chaîne de caractères en liste de caractères. La commande `"".join(list)` permet de transformer une liste en chaîne de caractères.





La gestion des exceptions



Le traceback

- Lorsqu'une erreur se produit, elle traverse toutes les couches de code comme une bulle d'air remonte à la surface de l'eau. Si elle atteint la surface sans être interceptée par le mécanisme des **exceptions**, le programme s'arrête et l'erreur est affichée. Le *traceback* (message complet d'erreur affiché) précise l'ensemble des couches traversées. (à lire de bas en haut)
- Exemple :

```
exception2.py > [E] ErreurTest
1  def division(a , b) :
2      return a / b
3
4  ErreurTest = division(10,0)
5
```

```
PS D:\Recherches\ExosCoursPython_David> & C:/Users/david/Python312/python.exe d:/Reche
Traceback (most recent call last):
  File "d:\Recherches\ExosCoursPython_David\exception2.py", line 4, in <module>
    ErreurTest = division(10,0)
                  ~~~~~~
  File "d:\Recherches\ExosCoursPython_David\exception2.py", line 2, in division
    return a / b
           ~^~
ZeroDivisionError: division by zero
```

- Gérer une exception permet d'intercepter une erreur pour éviter un arrêt du programme.
- La gestion des exceptions permet de gérer ces erreurs de manière contrôlée, ce qui rend le code plus robuste et facilite le débogage.



La gestion des exceptions

- Voici comment cela fonctionne :

1. **Levée d'exception** : Lorsqu'une erreur se produit dans un bloc de code, une exception est levée. Cela peut être dû à différentes raisons telles que des erreurs de syntaxe, des tentatives d'accès à des variables non définies, des opérations mathématiques invalides, etc.

1. Liste : https://www.w3schools.com/python/python_ref_exceptions.asp

2. Ou : `dir(__builtins__)` et `help(__builtins__)`

2. **Capture d'exception** : Pour gérer ces exceptions, Python utilise des blocs try-except. Dans un bloc try, vous placez le code où vous pensez qu'une exception pourrait se produire. Dans un bloc except, vous spécifiez quelles exceptions vous souhaitez gérer et comment vous souhaitez les gérer.

exception2.py > ...

```
1 def division(a , b) :  
2     return a / b  
3 try:  
4     ErreurTest = division(10,0)  
5 except ZeroDivisionError:  
6     print("Erreur : On ne divise pas par zéro !")  
7
```

```
● PS D:\Recherches\ExosCoursPython_David> & C:/Users/david/Python312/python.exe  
Erreur : On ne divise pas par zéro !
```



La gestion des exceptions

- Syntaxe générale :

```
try:
    ...                # séquence normale d'exécution
except <exception_1> as e1:
    ...                # traitement de l'exception 1
except <exception_2> as e2:
    ...                # traitement de l'exception 2
...
else:
    ...                # clause exécutée en l'absence d'erreur
finally:
    ...                # clause toujours exécutée
```



Exception : exercice

- Créez un programme qui demande à l'utilisateur son année de naissance et renvoie son âge. Gérer le fait que l'utilisateur peut éventuellement entrer autre chose qu'un nombre.





Les modules



Quelques infos sur les modules

- Module : fichier script Python permettant de définir des éléments de programme réutilisables. Ce mécanisme permet d'élaborer efficacement des bibliothèques de fonctions ou de classes.
- Exemples : pandas, numpy, matplotlib, opencv (cv), scipy, etc etc etc !
- Python est cependant « batteries included » : il existe de nombreux modules standards.
 - Liste ici : <https://docs.python.org/3/library/>
- Python Package Index : <https://pypi.org/> (parfois difficile de s'y retrouver !)
 - Google (ou autre) is your friend
- Ordre d'import : modules de la bibliothèque standard, bibliothèque tierces puis enfin les modules persos.



Importation de modules

```
import math  
s = math.sin(math.pi)
```

```
import math as m  
s = m.sin(m.pi)
```

```
from math import pi as PI, sin  
s = sin(PI)
```

```
from math import *  
s = sin(pi)
```

Attention aux conflits d'espace
de noms

Online help: `dir(math)`, `help(math)`



Les modules : exercice

- Importer le module random, utiliser dir() et help() pour l'explorer. Créer un programme qui génère une liste de 10 nombres aléatoires entre 1 et 6.



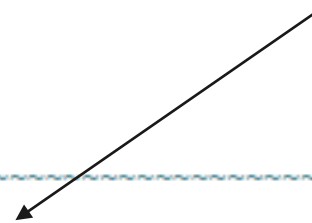
Créer son module

```
# cube.py

def cube(x) :
    """retourne le cube de <x>."""
    return x**3

# auto-test ~~~~~
if __name__ == "__main__" : # vrai car module principal
    if cube(9) == 729:
        print("OK !")
    else :
        print("KO !")
```

J'explique cette
ligne de suite!



L'importer

```
import cube

# programme principal ~~~~~
for i in range(1, 4) :
    print("cube de", i, "=", cube_m.cube(i))

"""
cube de 1 = 1
cube de 2 = 8
cube de 3 = 27
"""
```



Modules - Exercice

- Reprendre un des exercices concernant les fonctions et transformez-le en module. Utilisez ce module dans un programme.





Notions sur la programmation orientée objet (POO)



Quelques infos sur la POO

- C'est un gros morceau !

- la *POO* permet de mieux modéliser la réalité en concevant des modèles d'objets, les *classes*.
- Ces classes permettent de construire des *objets* interactifs entre eux et avec le monde extérieur.
- Les objets sont créés indépendamment les uns des autres, grâce à l'*encapsulation*, mécanisme qui permet d'embarquer leurs propriétés.
- Les classes permettent d'éviter au maximum l'emploi des variables globales.
- Enfin les classes offrent un moyen économique et puissant de construire de nouveaux objets à partir d'objets préexistants.

- Terminologie : une *classe* (nouveau type/structure de données) est *instanciée* pour créer un *objet* qui contient des *attributs* (données) et des *méthodes* (actions/_{fonctions})
 - D'ailleurs attention méthode vs fonction dans le code (notation pointée)



Quelques infos sur la POO

- Exemple de définition et utilisation d'une classe

```
import math

class Cercle():
    def __init__(self, r):
        self.radius = r

    def surface(self):
        return math.pi*self.radius**2

    def perimetre(self):
        return 2*math.pi*self.radius
```

```
unCercle = CCercle.Cercle(3)
print (unCercle.surface())
print (unCercle.perimetre())
```



Quelques infos sur la POO

- Exemple de définition et utilisation avec les fonctions magiques (dunders)

```
>>> class Vecteur2D:
...     def __init__(self, x0, y0):
...         self.x = x0
...         self.y = y0
...     def __add__(self, second): # addition vectorielle
...         return Vecteur2D(self.x + second.x, self.y + second.y)
...     def __str__(self):        # affichage d'un Vecteur2D
...         return "Vecteur({:g}, {:g})".format(self.x, self.y)
...
>>> v1 = Vecteur2D(1.2, 2.3)
>>> v2 = Vecteur2D(3.4, 4.5)
>>>
>>> print(v1 + v2)
Vecteur(4.6, 6.8)
```



POO – Petite applications directe

- Faire une classe chien (ou chat) avec des attributs : nom, taille, poids. Instancier la classe et faire marcher votre animal.



Ressources



former pour l'emploi

Petite liste (loin d'être exhaustive) de ressources sur le net

- Google ! (ou autre)
- python.org
- <https://realpython.com/> ← une mine !
- <https://planetpython.org/>
- Youtube (pour ceux qui aiment...) : Clever programmer, real python, anaconda...
- Awesome python (notamment sur github) pour trouver les bons modules.
- Chaque module (connu) possède son site et est généralement bien documenté.
- ChatGPT (ok je sors)



BRUXELLES FORMATION



former pour l'emploi

Merci !