Formation Python

Initiation par la pratique

Votre formateur : Quentin BIDOLET

Nous contacter :
client@m2iformation.fr

0.1

Présentations

Quentin BIDOLET

- 5 ans d’expérience en développement Python
- Surtout du web, un profil plutôt Back avec quelques expériences sur du Front

Participants

- Les bases de la programmation.
- Les concepts de la POO.
- Droits admin obligatoires sur tous les postes.
- Quelles sont vos attentes par rapport à cette formation ?
- Quelles sont les technos et langages sur lesquels vous travaillez habituellement ?

0.2

Go to Slack

Accès à Slack

- Vérifiez dans vos email que vous avez bien reçu le lien d'invitation au Slack de la formation.

- Merci de bien vouloir me rejoindre sur Slack.

0.3

GitHub

Le dépôt GitHub de la formation

- Donner le lien du dépôt de la formation sur Slack.
- A chaque notion abordée : un commit.
- Vous pourrez fork mon dépôt après la formation.
- Si vous souhaitez refaire les programmes après la formation, vous aurez accès à tous les commit pour chaque étape.

0.4

Supports de cours et livres

Support et livres

- Adresse gmail des participants dans slack.
- Partage du support de cours en lecture seule.
- Dossier Livres avec les droits éditeur.

0.5

Déroulement de la formation

Horaires, temps de pause et dossier pédagogique

- De 9h à 17h ou de 9h30 à 17h30 (voir convocations).
- Une pause de 15mn le matin et une pause de 15mn l'après midi.
- Midi, une pause déjeuner de 1h15.
- Dernier jour de la formation se termine à 15h00 (voir convocations).
- Obligation du dernier jour à faire avant 15h donc, au retour de la pause déjeuner : vérifier les feuilles d'émargement et faire les évaluations formateur.
- Si certifications à passer, elles se feront en ligne le dernier jour l'après midi, après les évaluations formateur.
- Le dernier jour nous passerons en mode Atelier.

0.6

Déroulé pédagogique

Le processus pédagogique

- Pour chaque notion abordée :
  - Ce que nous voulons faire, quel est l'objectif.
  - La partie théorique et où trouver l'information dans la doc officielle.
  - La pratique en codant.
  - Le code sur Slack pour ceux qui veulent faire un copier-coller.

Atelier du dernier jour

- Un ensemble de TP qui vous obligeront à écrire votre propre code, à chercher dans la doc, sur Google et à revoir le code que nous avons écrit durant la semaine.

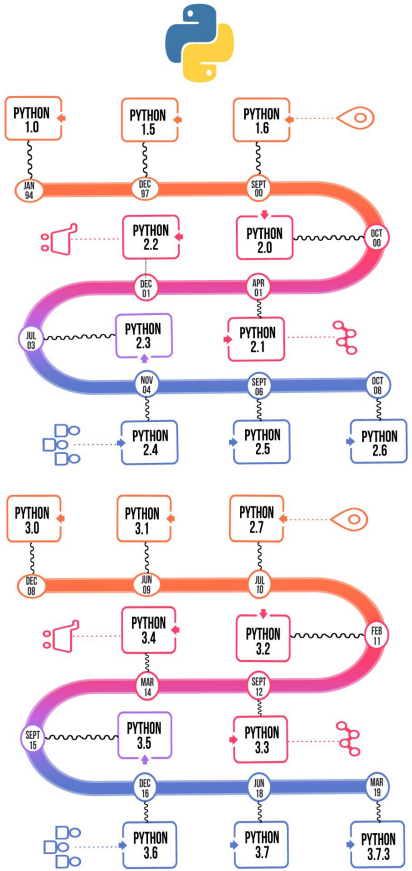
- Le but étant que vous soyez le plus à l'aise possible avec le Python et que vous sachiez structurer et développer vos propres projets Python à la fin de formation.

0.7

Python

Historique

- Python 2 existe depuis 1991, plus vieux que Java !
- Python 3 : compilation de Hard Fixes
- Python 2.7 est la dernière version de Python 2



Champs d’application

- Administration système
- Web
- Machine learning et Intelligence Artificielle
- Calculs scientifiques
- Applications bureau
- Systèmes embarqués

0.8

Installations outils et IDE

Installer Python

- PIP (Package Installer for Python) pour installer les librairies utilisées durant la formation https://pip.pypa.io/en/stable/installation/
- Python 3.10

  https://www.python.org/downloads/release/python-3109/
- Vérifiez dans cmd la version installée.
- https://docs.python.org/3/whatsnew/

Installer PyCharm

- https://www.jetbrains.com/pycharm/download/

0.9

Conventions

PEP 8

- 4 espaces par niveau d’indentation.
- Lignes de 79/120 caractères maximum.
- 2 lignes vides pour séparer les fonctions.
- Méthodes dans une classe doivent être séparées par une ligne vide.
- UTF-8 sans déclaration d’encodage.
- https://peps.python.org/pep-0008/

1

Echange avec la console

Console

- `pip list`
- `python --version`
- `import this`

https://docs.python.org/fr/3.10/using/cmdline.html

Commit

- Commit **console.py**

2

Premier programme

Un langage à typage dynamique

- Les langages de programmation ont des variables. Les langages à typage statique ont besoin que l'on déclare le type de la variable avant de pouvoir les utiliser.

- Contrairement à un typage dynamique qui permet directement de manipuler les valeurs et laisser à l'interpréteur le typage de l'objet.

- C'est possible car Python est un langage **interprété**, c'est-à-dire qu'il lit votre programme ligne par ligne pour le traduire en langage machine.

Langage compilé vs interprété

- Un langage interprété permet le typage dynamique mais il est souvent plus lent qu'un langage compilé.
- Leurs vitesses s'est améliorée au fil des années car leur interpréteur devient de plus en plus optimisé.
- Pendant de longue année, les langages dynamiques étaient surtout utilisés pour faire de court programme, les scripts. Ils préparaient les données à être traitées par de plus long programme, qui eux étaient compilés. On appelait ces programmes des **glue code**.
- Bien que les langages interprétés soient plus adaptés pour les scripts, ils sont maintenant capables de s'attaquer à tout type de programme.

Les extensions .py et .pyc

- Pour écrire votre code Python, vous créez un fichier avec l'extension **.py**

- Bien que Python soit un langage interprété, on trouve la présence de fichiers **.pyc** qui contiennent du bytecode compilé. Ces fichiers sont créés par l'interpréteur quand un fichier .py est importé. C'est la traduction de votre code en langage machine.

Python, quand ne pas l'utiliser

- Python n'est pas le meilleur langage dans toutes les situations.
- Il n'est pas installé partout par défaut et vous n'avez pas toujours la possibilité de l'installer.
- Il est assez rapide pour la grande majorité des situations, mais il peut ne pas être suffisant dans le cas de longs calculs où la performance est la priorité. Si votre programme passe son temps à faire des calculs complexes et que la performance est vitale, il est peut être plus adapté de l'écrire dans un langage compilé comme le C, C++ ou Java qui seront plus rapides.

Python, pourquoi l'utiliser

- Souvent, un bon algorithme en Python est plus performant qu'un inefficace en C. L'excellente vitesse de développement en Python vous donne plus de temps pour fignoler votre programme.
- Dans beaucoup d'applications, la vitesse de programme est la conséquence de temps d'attente d'un serveur ou programme externe et le CPU a peu d'impact sur le résultat final.
- L'interpréteur standard de Python est écrit en C et peut être étendu avec du code C, vous pouvez toujours optimiser votre programme de cette manière.
- L'interpréteur Python est de plus en plus rapide.

Commentaires, blocs et indentations

- Commenter avec #
- Continuer les lignes avec \

```
alphabet = 'abcdefg' + \
'hijklmnop' + \
'qrstuv' + \
'wxyz'
```

Commit

- Commit **comment.py**

Saisie et affichage

- print() et pprint()
- input()

Commit

- Commit **print.py**
- Commit **input.py**

3

Types de base

Booléens, numériques et chaîne de caractères

- Les booléens (**True** ou **False**)
- Les réels (**3.141** ou **1.0e6**)
- Entier, réel, booléen, chaîne de caractères [...] tout est objet.
- Python est fortement typé, c'est-à-dire que le type d'un objet ne peut pas changer.

Nom des variables

- Un nom de variable peut contenir des **lettres minuscules**, **majuscules**, des **chiffres** et des **underscores** (_).
- Elles ne peuvent pas commencer par un nombre. Les noms commençant par un underscore sont traités spécialement, nous y reviendrons plus tard.
- Certains noms sont réservés, vous pouvez les consulter ci-dessous. https://www.w3schools.com/python/python_ref_keywords.asp

Les variables sont des noms

- `a = 12`
- Une variable attache un nom à un objet, un assignement ne copie pas la valeur.
- `b = a`
- `print(b)`
- Cela affiche le même objet.

Commit

- Commit **variables.py**

Les opérateurs mathématiques

- Addition et soustraction : + et -
- Multiplication et exponentielle : * et **
- Division et division euclidienne : / et //
- Modulo : %

Commit

- Commit **maths.py**

Ecrire un nombre

- En base 2 : **0b** ou **0B**
- En base 8 : **0o** ou **0O**
- En hexadecimal : **0x** ou **0X**

Taille maximale d'un entier

- En Python 2 la taille d'un entier était limitée à 32 bits, soit entre -2, 147 483 648 et 2 147 483 647.
- Un entier long était stocké en 64 bits, soit entre -9 223 372 036 854 775 808 et 9 223 372 036 854 775 807.
- En Python 3, plus d'entier long et les entiers n'ont plus de limite !
-
```
>>> googol = 10**100
```
-
```
>>> googol * googol
```

Conversions de types

- ```
  >>> int(True)
  1
  ```
- ```
  >>> float(True)
  1.0
  ```
- ```
  >>> str(True)
  'True'
  ```
  ```

Commit

- Commit **types.py**

Commit

- Commit **conversions.py**

Manipulation de chaîne de caractères

- Guillemets simples ou doubles `' '` ou `" "`
- Multiligne avec trois guillemets simples ou doubles `''''` ou `"""`
- Concaténation avec `+`
- Escape avec `\`
- Extraire avec des caractères avec `[ start : end : step ]`
- Cast avec `str()`
- Longueur avec `len()`
- Concaténation avec `join()`

Manipulation de chaîne de caractères

- Séparer avec **split()**
- Remplacer avec **.replace()**
- Supprimer un caractère avec **strip()**
- **.startwith()** / **.endswith()** / **.find()** / **.rfind()** / **.count()** / **.isalnum()**
- **.capitalize()** / **.title()** / **.upper()** / **.lower()** / **.swapcase()**
- **.center()** / **.ljust()** / **.rjust()**

Commit

- Commit **string.py**

4

Les instructions de base

Les opérateurs de base

- Egalité ==
- Inequalité !=
- Plus petit que <
- Plus petit ou égal <=
- Plus grand que >
- Plus grand ou égal >=
- Appartenance in

Ce qui est **Faux**

- Booléen **False**
- Null **None**
- L'entier zéro **0**
- Le réel zéro **0.0**
- Chaine de caractère vide **""**
- Liste vide **[ ]**
- Tuple vide **()**
- Dictionnaire vide **{ }**
- Set vide **set()**

- Tout le reste est **True**

Structure conditionnelle

```
if b > a:
    print("b is greater than a")
elif a == b:
    print("a and b are equal")
else:
    print("a is greater than b")
```

Boucle while

```
i = 1
while i < 6:
    print(i)
    i += 1
```

Boucle for

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)

for x in "banana":
    print(x)
for x in range(2, 30, 3):
    print(x)

for x in range(6):
    print(x)
else:
    print("Finally finished!")
```

Break

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
    if x == "banana":
        break
```

Continue

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
        print(x)
```

Commit

- Commit **instructions.py**

5

Types agrégés

Types agrégés : List

- Les listes stockent un ou plusieurs objets.
- Création avec **list()** ou **[ ]**

```
thislist = ["apple", "banana", "cherry"]
```

```
list(("apple", "banana", "cherry"))
```

Manipulation de liste

- Sélectionner avec [*indice*] et [*début* : *fin*]
- Ajouter avec **.append()**
- Combiner avec **.extend()** ou **+=**
- Insérer sur un offset avec **.insert()**
- Supprimer sur un offset avec **.del()**
- Supprimer une valeur avec **.remove()**
- Obtenir un objet par offset et le supprimer avec **.pop()**
- Avoir l'offset d'un objet avec **.index()**
- Tester une valeur avec "value" **in** ["value"]
- Compter les occurrences avec **.count()**
- Convertir en string avec **.join()**
- Trier avec **.sort()**
- Longueur avec **len()**
- Assigner avec **Copy()** / **.copy()** ou **=**

Commit

- Commit **list.py**

Types agrégés : Dictionnaires

- Similaire à une liste mais l'ordre n'a pas d'importance.
- On ne les sélectionne pas avec un offset mais une **clé.**
- La **clé** est souvent un string mais cela peut être tout type immuable de Python : booléen, integer, float, tuple, string etc.
- Ils sont mutables donc on peut **ajouter**, **supprimer**, **modifier** les éléments.

Création de dictionnaire

- Création avec { }
- Attention, les clés doivent être **unique**.
- Conversion en utilisant **dict()**

Manipulation de dictionnaire

- Ajout ou suppression d'un élément avec **[** *clé* **]**
- Combinaison avec **.update()**
- Supprimer un élément par Key avec **del**
- Supprimer tous les éléments avec **.clear()**
- Tester la présence avec **in**

Manipulation de dictionnaire

- Obtenir un élément par clé
- Si la clé n’est pas présente une exception est levée
- Toutes les clés avec **.keys()**
- Toutes les valeurs avec **.values()**
- Toutes les clés-valeurs avec **.items()**
- Assigner avec **Copy** / **copy()** ou **=**

Commit

- Commit **dict.py**

Types agrégés : Tuples

- Similaire aux listes, les tuples sont des séquences d'objets qui, contrairement aux listes, sont immuables.

- On ne peut ni ajouter, supprimer ou modifier les objets une fois qu'un tuple est défini.

Création de tuple

- Création avec ()
- On sépare les éléments par des virgules.
- S'il n'y a qu'un seul élément, il faut obligatoirement ajouter une virgule à la fin.
- Avec plusieurs éléments le dernier n'est pas obligatoirement suivi de virgule.

Manipulation de tuples

- Avec les tuples on peut assigner plusieurs variables en une seule fois.
- Cela s'appelle l'*unpacking*.

Pourquoi utiliser des tuples

- Ils utilisent moins d’espace.
- On ne peut pas modifier un tuple par erreur.
- On peut utiliser les tuples comme un dictionnaire.
- Peut être une simple alternative à des objets.
- Les arguments de fonction sont des tuples.

Commit

- Commit **tuples.py**

Types agrégés : Set

- Un **set** est une liste de clé unique.
- Création et conversion avec **set()**
- Tester une valeur avec **in**

Commit

- Commit **set.py**

Résumé des structures de données

```
marx_list = ['Groucho', 'Chico', 'Harpo']

marx_tuple = ('Groucho', 'Chico', 'Harpo')

marx_dict = {'Groucho': 'banjo', 'Chico': 'piano',
'Harpo': 'harp'}

>>> marx_list[2]
'Harpo'

>>> marx_tuple[2]
'Harpo'

>>> marx_dict['Harpo']
'harp'
```

6

Procédures et fonctions

Déclaration de fonction

- En utilisant **def()**

```
def do_nothing():
    pass
```

- Vous pouvez passer vos arguments sous la forme d'un tuple.

Les paramètres par défaut

```
def menu(wine, entree, dessert='pudding'):
    return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

Arguments : *args

- Attention, * n'est pas un pointeur
- Il n'y a pas de pointeur en Python.
- Un astérisque groupe des variables dans un tuple de valeurs paramétrées.
- C'est très utile pour print().

```
def print_more(required1, required2, *args):
    print('Need this one:', required1)
    print('Need this one too:', required2)
    print('All the rest:', args)
```

Arguments : **kwargs

- Vous pouvez utiliser deux astérisques pour regrouper les arguments dans un dictionnaire.

```
>>> print_kwargs(wine='merlot', entree='mutton',
dessert='macaroon')
```

```
Keyword arguments: {'dessert': 'macaroon', 'wine': 'merlot',
'entree': 'mutton'}
```

Arguments : *args et **kwargs

- Si vous utilisez les deux arguments, il faut faire attention à l'ordre.
- Vous n'êtes pas obligé d'utiliser les noms **args** et **kwargs**, il s'agit seulement d'un usage standardisé.

Closure

- Vous pouvez définir une fonction à l'intérieur d'une autre.
- Une fonction interne peut être utilisée comme une **closure**.
- C'est une fonction qui se souvient des valeurs dans le contexte où elle a été créée, même si elle n'est plus actuellement en ce contexte.

```
def outer_function(x):
    def inner_function(y):
        return x + y
    return inner_function

closure = outer_function(10)
print(closure(5))   # affiche 15
```

Fonction lambda

- Une fonction lambda est une fonction anonyme, on l'a définie dans une seule expression.

```
edit_story(stairs, lambda word: word.capitalize() + '!')
```

Les générateurs

- Une fonction génératrice en Python est une fonction qui utilise le mot clé yield pour retourner une valeur à chaque itération, plutôt que de retourner une seule valeur à la fin de son exécution.

- Cela permet de produire une séquence de valeurs à mesure qu'elles sont demandées, ce qui peut être plus efficace en termes de mémoire et de temps d'exécution que de générer une liste complète en mémoire

Commit

- Commit **functions.py**

L'import de module

- La façon la plus simple est d'utiliser **import module**, où module est le nom d'un autre fichier Python, sans préciser l'extension.
- On peut aussi importer en utilisant **from module import fonction**
- A vous de choisir selon votre préférence.
- Dans tous les cas vous pouvez utiliser l'alias **as**
- **dir()** pour lister tous les noms de fonctions ou de variables d'un module.

7

Exception et Débogage

La gestion des exceptions

- Une exception est un événement qui se produit lorsqu'un programme ne peut pas exécuter une instruction.

- Gérer les exceptions permet de contrôler les erreurs qui se produisent lors de l'exécution d'un programme.

- Une gestion appropriée des exceptions peut permettre à un programme de continuer à s'exécuter, même s'il rencontre des erreurs.

- Cela se fait à l'aide de blocs try-except.

  try contient le code qui peut générer une exception.

  except contient le code qui est exécuté si une exception est levée.

Types d’exceptions

- Python possède de nombreux types d'exceptions prédéfinies, comme ValueError, TypeError, IndexError, etc.
- Vous pouvez également définir vos propres exceptions en créant une nouvelle classe dérivée de la classe Exception.
- En Python, il est possible de lever une exception à l'aide du mot-clé raise.
- Lorsqu'une exception est levée, le programme s'arrête et cherche un bloc try-except pour gérer l'exception.

Débogage : pas à pas avec pdb

- Le debugger standard de Python, **pdb**.
- Il faut importer **pdb** puis on écrit **-m pdb** avant le fichier à débugger.
- **python -m pdb programme.py**
- Cela nous place à la première ligne
- **c** (continue) pour continuer jusqu'à la fin
- **s** (step) va itérer sur tout le code Python, le vôtre, une bibliothèque ou autre.
- **n** (next) pour itérer sur la prochaine ligne de votre programme uniquement.

Débogage : Conseils

- Utiliser **s** quand vous n'êtes pas sûr de savoir où est le problème.
- Utiliser **n** quand vous savez que cette fonction n'est pas la cause.
- La plupart du temps le problème vient de votre code, les bibliothèques sont généralement bien testées.

8

Gestion de fichiers

Créer ou ouvrir un fichier

- `fileobj = open( filename, mode )`

- **fileobj** est un objet fichier retourné par **open()**. **filename** est le nom du fichier et **mode** est un string qui définit le type de fichier et ce qu'on veut faire.

- **r** pour lire
- **w** pour écrire. Si le fichier n'existe pas il est créé. Si le fichier existe, on écrit par dessus.
- **x** pour écrire, mais seulement si le fichier n'existe pas.
- **a** pour append (écrire après la fin) si le fichier existe.
- La deuxième lettre du mode est le type de fichier **t** (ou rien) veut dire texte et **b** pour binary.

Ecrire dans un fichier

```
fout = open('relativity', 'wt')
fout.write(poem)
fout.close()
```

- Faut-il écrire **write()** ou **print()** ? Par défaut **print()** ajoute un espace après chaque argument et une nouvelle ligne à la fin. Pour que **write()** fonctionne comme **print()** il faut ajouter les deux arguments suivants :
- **Sep** (séparateur, par défaut un espace, ‘ ‘)
- **End** (fin du string, par défaut une nouvelle ligne, ‘\n)

Lecture de fichier

- On peut appeler **read()** sans argument pour lire tout un fichier d'un coup ou lui donner un nombre maximum de caractères. Attention avec ce paramètre, 1 Go va consommer 1Go de mémoire.
- Une fois qu'on a utilisé **read()** jusqu'à la fin, il va retourner un string vide.
- On peut aussi lire un fichier ligne par ligne avec **readline()**.

```
poem = ''
fin = open('relativity', 'rt' )
while True:
    line = fin.readline()
    if not line:
        break
    poem += line
```

- La façon la plus simple de lire un fichier est d'utiliser un itérateur.

Fermer un fichier

- Vous pouvez fermer un fichier automatiquement en l'ouvrant avec **with**

```
with open('relativity', 'wt') as fout:
    fout.write(poem)
```

- Ou le fermer manuellement avec **close()**.

Rechercher dans un fichier

- Utiliser **.seek(offset, origin)**
- Si **origin** est 0, offset depuis le début.
- Si **origin** est de 1, offset commence à compter la position courante.
- **.tell()** pour avoir l'offset de la position courante
- Lire jusqu'à la fin du fichier:

```
bdata = fin.read()
len(bdata) 1
bdata[0]
```

Commit

- Commit **files.py**

9

Modules intégrés

Le module shutil

- Shutil permet de copier, créer et supprimer des opérations sur un fichier.

- **shutil.copy(***source, destination, *, follow_symlinks = True***)** est utilisé pour copier le contenu d'un fichier source dans un fichier ou répertoire. Cela copie aussi les permissions du fichier. Les autres metadata comme la date de création ou de modification ne sont pas copiées.

- **shutil.copy2(***source, destination, *, follow_symlinks = True***)** est identique à la première mais copie toutes les metadatas.

Le module shutil

- **shutil.copyfile(***source, destination, *, follow_symlinks = True***)** copie un fichier, sans métadata.

- **shutil.copytree(***src, dst, symlinks = False, ignore = None, copy_function = copy2, ignore_dangling_symlinks = False***)** copie récursivement un répertoire entier. Le répertoire de destination ne doit pas exister.

- https://docs.python.org/3/library/shutil.html

Le module os

- Le module **os** fournit des fonctions pour interagir avec le système d'exploitation. **CWD** est le répertoire courant, là où le script est exécuté (Current Working Directory).

- **os.getcwd()** nous donne le répertoire courant.

- **os.chdir(***path***)** permet de changer de répertoire courant.

- **os.mkdir(***path, mode = 0o777, exist_ok = False***)** crée un nouveau répertoire.

- **os.mkdirs(***path, mode = 0o777, exist_ok = False***)**, idem et crée tous les sous-répertoire récursivement.

Le module os

- **os.listdir(*path*)** renvoie la liste de tous les fichiers et répertoires du chemin passé en paramètre.
- **os.remove(*path*)** supprimer un fichier.
- **os.rmdir(*path*)** supprimer un répertoire vide.
- **os.name** nous donne le nom du système d'exploitation.
- https://docs.python.org/fr/3/library/os.html

Le module zlib

- Le module **zlib** apporte des fonctions pour compresser et décompresser des données.

- **zlib.compress(***data, level***)** retourne les données compressées. Level est un entier qui permet de contrôler le niveau de compression.

  1 optimise la rapidité tandis que 9 est la meilleure compression.

- **zlib.compressobj(***level, method, wbits, memLvel, strategy***)** pour une grande quantité de données.

- **zlib.decompress(***data, wbits, bufsize***)** décompresse des données.

- **os.decompressobj(***wbits, [zdict]***)** pour une grande quantité de données.

- https://docs.python.org/3/library/zlib.html

Commit

- Commit **os.py**
- Commit **shutil.py**

10

Ecrire des modules

Définition de module

- Un module est un fichier contenant une collection de fonctions qu'on veut inclure dans une application.
- Un module peut contenir des fonctions et des variables de tous types.
- Aucune règle de nommage.
- L'extension doit être **.py**

Module : Packages

- De quelques lignes de codes à plusieurs fichiers ou même plusieurs répertoires, on peut organiser les modules dans une hiérarchie de fichier appelée packages.
- On crée un répertoire source avec un fichier **__init__.py**
- Il peut être vide mais Python en a besoin pour reconnaître le répertoire en tant que package.

__main__.py

- C’est le fichier qui va contenir le main de votre module, ce nom de fichier doit être impérativement respecté pour que Python puisse le reconnaître en tant que tel.

- Votre programme peut savoir s’il est exécuté ou non depuis un import en vérifiant son propre **__name__**

```
if __name__ == '__main__':
    pass
```

Installation de modules externes

- **easy_install**, sorti en 2004 est maintenant déprécié.
  Historiquement, c'était le gestionnaire de package de référence.

- **Pip** (Package Installer for Python) est la référence depuis 2008

https://packaging.python.org/en/latest/discussions/pip-vs-easy-install/

11

Les classes

Approche objet

- Des nombres aux modules, tout est objet en Python. Un objet contient à la fois une donnée (appelé attribut) et du code (appelé méthode).

- Cela représente une instance unique de quelque chose. Par exemple, les objets **integers** (entiers) représentent une valeur entière à travers un objet qui facilite la manipulation des nombres avec des méthodes additionnelles comme les additions ou les multiplications.

- Les strings sont aussi des objets et ont des méthodes comme on l'a vu comme **.capitalize()** ou **.replace()**

Classes et instances

- Quand vous créez un objet à partir de rien, vous devez créer une classe qui indique ce qu'il contient.

```
class Person():
    pass
```

- C'est la classe la plus simple que nous pouvons créer, une classe vide.

Constructeur : __init__

- On ajoute un constructeur à cette classe :

```
class Person():
    def __init__(self):
        pass
```

- L'argument self spécifie que la fonction réfère à l'objet même. Quand vous définissez **__init__()** dans une définition de classe, le premier paramètre doit toujours être **self**. Le mot **self** est un standard que vous devez respecter.

- On peut ajouter les paramètres que l'on veut après le **self**.

Méthodes membres

- Vous pouvez définir une fonction à l'intérieur d'une autre.

```
def outer(a, b):
    def inner(c, d):
        return c + d
    return inner(a, b)
```

Héritage multiple

- Une classe peut hériter d'une ou plusieurs classes en Python, tout comme le C++. C'est appelé l'héritage multiple.

- En héritage multiple, toutes les fonctionnalités des classes de bases sont héritées dans la classe enfant.

```
class Base1:
    pass
class Base2:
    pass
class HeritageMultiple(Base1, Base2):
    pass
```

12

Interface graphique

Programmation UI avec Tkinter

- Tkinter est un module de base intégré dans Python donc pas d'installation. Il permet de réaliser des interfaces graphiques.

- Commençons par un Hello World!

```
from tkinter import *
fenetre = Tk()
label = Label(fenetre, text="Hello World!")
label.pack()
fenetre.mainloop()
```

Les widgets Tkinter

- Pour créer votre logiciel graphique, vous devez ajouter dans une fenêtre des éléments que l'on nomme **widget**. Un **widget** peut aussi bien être un bouton qu'un simple label.

Création d’une UI simple

https://docs.python.org/3/library/tkinter.html

Accès aux bases de données relationnelles

- Nous allons utiliser SQLite, une base de données légère, portable et open source.

13

Web : CGI

Web : Common Gateway Interface CGI

- Au début d'internet, **the Common Gateway Interface (CGI)** a été conçu pour que les clients fassent fonctionner des programmes externes (de serveur web) et retourne le résultat.

- CGI a également géré les arguments en entrée pour le programme externe. Par contre, le programme reprend de zéro pour chaque accès client, ce qui fait que le programme ne scale pas très bien car tout programme à un temps de démarrage, aussi petit soit-il.

- Pour éviter ce temps de démarrage, on a fusionné le langage interpréteur avec le serveur web. PHP a le sien dans Apache avec **mod_php**, Perl dans **mod_perl** et Python dans **mod_python**.

Web : Web Server Gateway Interface WSGI

- Le développement web en Python a fait un bond en avant avec la définition d'un Web Server Gateway Interface (WSGI), un API universel entre les applications web Python et les serveurs webs. Tout framework web en Python utilise le WSGI. Vous n'avez pas besoin de connaître son fonctionnement en détail, mais c'est important de savoir qu'il existe et quel est son utilité !

Web : Frameworks

- Les serveurs Web gèrent l'HTTP et le WSGI mais vous utilisez un framework pour vraiment écrire du code Python qui va alimenter votre site.

- Si vous voulez faire un site web en Python, il existe plusieurs frameworks, les plus connus sont **Django** et **Flask**.

- Nous utiliserons **Django** dans cette formation.

Frameworks web : Fonctionnalités

- Tous ces frameworks gèrent les fonctionnalités suivantes :

  Routes, interprète les URLS et trouve le code correspondant

  Templates, fusionne les données back-end dans une page HTML

  Authentification, gère les noms d'utilisateurs, mots de passe

  Autorisation, attribut des permissions

  Sessions, maintient un stockage de données temporaires lors de la visite d'un utilisateur sur le site

14

TP - Validation des acquis

Formation Python

Initiation par la pratique

Votre formateur : Quentin BIDOLET

Nous contacter :
client@m2iformation.fr