Support de cours Python

Table des matières

Présentation

Ce support vise à introduire des concepts de base en programmation, avec une forte invitation à la pratique et aux bonnes pratiques.

Python-1 intro

Python

Python est un langage inventé par le programmeur Guido van Rossum dans les années 90. C'est un projet open source et multi-plateformes.

Python est un langage interprété, contrairement à d'autres, comme le C qui nécessite une phase de compilation qui produit une version exécutable pour une plateforme cible, le code source d'un programme Python est directement portable (l'analyse et sa traduction en langage machine se fait au cours de son lancement par un interpréteur Python lié à l'OS hôte).

Vérification de votre système

Ouvrir une **console système** (un terminal), et lancer la commande : `python3 --version` (ou `python --version`)

Exemple de résultat attendu (vérifier que vous êtes en 3.x.x) :

Listing 1. Exemple ($ est le prompt, à ne pas recopier)

```
$ python3 --version
Python 3.10.6
```

Python peut être lancé en mode interactif ou en mode script.

Python en mode interactif

Il suffit de lancer la commande `python3`, dans ce cas l'interpréteur attend la saisie de code, qu'il interprétera à partir du moment où il reçoit une donnée de fin de saisie (touche `Enter`)

Pour sortir de ce mode, lancer la commande `CTRL + D` (ou `CTRL + Z` sous windows).

Listing 2. Tester ce code

```
$ python3
>>> print('Hello World')
Hello World
>>>
```

Python en mode script

Dans ce mode (usage courant) le code source est placé dans un fichier text, avec `.py` comme extension.

Tester cette procédure

1. créer un fichier texte nommé `tp1.py`
2. inscrire l'instruction `print('Hello World')` comme seule ligne de ce fichier
3. sauvegarder le fichier `tp1.py`
4. dans un terminal, lancer la commande : `python3 tp1.py`.

   Vous devriez voir afficher `Hello World`

Cette procédure peut être automatisée (la première instruction est équivalente aux étapes 1 à 3 de la procédure présentée). L'opérateur système `'>'` dirige temporairement le flux standard vers un fichier.

```
echo "print('Hello World')" > tp1.py | python3 tp1.py
```

L'opérateur système `'>>'` dirige le flux standard en fin du fichier cible (ajout).

⇒ La redirection du flux standard vers un fichier est une technique bien connue

> des hackers.

Documentation

- Préférer le site officiel docs.python.org et stackoverflow.
- Consulter la documentation technique des objets du langages `help()` (et `dir()`).
  - la fonction help() extrait des chaînes de documentation (modules, fonctions et méthodes). Exemples :
    - `help(print)`
    - `help(sys.exit)` (le module `sys` doit être importé avant cette commande)
- Consulter régulièrement le référentiel de bonnes pratiques de programmation en python :
  - Style Guide for Python Code - par Kenneth Reitz et Tanya Schlusser
  - Le guide de l'auto-stoppeur pour Python - par Kenneth Reitz

Mode interactif

EXERCICE 1

1. Lancez l'interpréteur dans un terminal
2. Tester une à une les expressions suivantes et essayez d'expliquer ce qui se passe dans chacun des cas. N'hésitez pas à échanger entre vous et à chercher des explications en ligne. Le langage python est utilisé par une grande communauté internationale. Voir le tableau de correspondances opérateur-fonction : https://docs.python.org/fr/3/library/operator.html#mapping-operators-to-functions

Code	Résultat	Explications
`2 * 11 + 3 + 4 * 5 - 4 + 1`	`42`	La priorité des opérateurs arithmétiques suit les conventions usuelles. (voir priiorité des opérateurs)
`2 ** 16`		
`pow(2, 16)`		
`2/3`		
`int(2/3)`		
`round(2/3)`		

`import math`
`math.ceil(2/3)`
`math.ceil(2/5)`		

`4%2`		
`12%7`		
`float(40+2)`		
`"année" + "2023"`		Concaténation de 2 chaînes de caractères
`s=samedi`		
`s="samedi"`		
`s * 4`		
`s[0], s[1]`		
`s[6]`		
`'spam'[0]`		
`"spam"[1]`		
`"spam"[0:2]`		
`"spam"[1:]`		
`('pizza' + s[0]) * 3`		

Les opérations associées aux structures de données de type *conteneur* seront étudiées plus loin. Pour les curieux, voir : https://docs.python.org/fr/3/library/stdtypes.html#typesseq

Quel éditeur de code utiliser ?

Vous trouverez ici https://docs.python.org/3/using/editors.html un inventaire de nombreux éditeurs.

Voici une sélection :

1. `vi` : l'éditeur incontournable, présent dans toutes les distributions linux. Il vous sauvera de situation délicate (intervention à distance sur un serveur)
2. `Visual Studio Code` et son plugin python : léger mais suffisament complet pour démarrer avec python (complétion, vérification de cohérence de type, débogueur intégré…), également utilisé pour le dev frontend.
3. pyCharm, l'IDE à destination des professionnels (gain de productivité assuré)

[vscode] | *vscode-python.png*

Figure 1. Visual Studio Code avec le plugin Python, ouvert sur le dossier TPS

Exercices en mode script

Un script python, appelé aussi "module", peut être utilisé directement, comme dans l'exemple `python3 tp1.py` ou intégré dans un autre module (via l'instruction `import`).

Python vient avec de nombreux modules préinstallés. Vous pouvez en consulter la liste avec : `pip3 list -V`. Pour en utiliser d'autres, il faut préalablement les télécharger. Voir ici pour en savoir plus sur la gestion des modules : https://docs.python.org/fr/dev/installing/index.html

Dans un premier temps, placez-vous dans un dossier de votre espace personnel dédié aux exercices en python.

EXERCICE 2

Voici un code source d'un programme python respectant les conventions d'usage :

Listing 3. fichier tp1.py

```
#!/usr/bin/env python3 ①
def exo2() -> None : ②
  """
   exercice 2 du TP1
  """
  nom = input("Entrez votre nom : ")
  print("Bienvenue " + nom + " !")


if __name__ == "__main__": ③
  import sys ④
  exo2() ⑤
  sys.exit(0) ⑥
```

① (optionnel) Shebang. Permet de rendre le script "directement" exécutable. voir https://stackoverflow.com/questions/6908143/should-i-put-shebang-in-python-scripts-and-what-form-should-it-take

② Définition d'une fonction nommée `exo2`, qui déclare ne "rien" retourner (`None`), avec sa chaîne de documentation

③ Si le script est utilisé directement (en argument de l'interpréteur python), alors la valeur de la variable `__main__` est "*__main__*", sinon il est utilisé en import dans un autre script et c'est le nom du script (module), sans son extension. Remarque : les noms de variables encadrés de 2 underscores (`__`) sont des variables système (pré)définies par l'interpréteur.

④ Importation du module `sys` (qui contient des fonctions système, dont `exit` utilisée plus loin)

⑤ Appel de la fonction `exo2`.

⑥ Appel la fonction `exit` afin de demander la sortie du mode interpréteur de python avec transmission du code de retour. Voir https://docs.python.org/fr/3/library/sys.html#sys.exit, ou, en mode interactif, appeler l'aide sur cette fonction via la commande `help(sys.exit)`. **Retourner zéro signifie que le programme se termine avec succès, toute autre valeur signale à l'appelant une anomalie à l'exécution**.

Travail à faire

1. Si ce n'est pas déjà fait, créer un dossier `dev`
2. Créer un sous-dossier `dev/TPS` et **ouvrir ce dossier** avec l'éditeur visual studio code
3. Créer le fichier `tp1.py`
4. Recopiez le code ci-dessus
5. (optionnel) Rendez-le exécutable (par exemple avec la commande `chmod +x tp1.py`)
6. Exécutez-le (dans un terminal), éventuellement corrigez les erreurs de frappe.
7. Modifiez la fonction `exo2()`, afin qu'elle affiche le prénom et le nom. Elle devra pour cela inviter l'utilisateur à entrer son prénom.

EXERCICE 3

On vous présente un programme exprimé en pseudo-langage et une traduction en Python. Après avoir pris connaissance de la version en pseudo-langage, recopier la traduction proposée en Python (code source ci-dessous) comme nouvelle fonction dans le script `tp1.py`.

Listing 4. Version pseudo-langage

```
Afficher("Entrez un nombre entier svp :")

lire un nombre au clavier et placer sa valeur
dans une variable nommée x ①

Si x est pair Alors
  Afficher("Ce nombre est pair")
Sinon
  Afficher("Ce nombre est impair")
```

```
FinSi
```

① ou plus simlement : `x <-- lire un nombre au clavier`

Listing 5. Version python

```
def exo3() -> None :
  x = int(input("Entrez un nombre entier svp : "))
  if x % 2 == 0 :  # le reste de division par 2 est-il zéro ?
    print("Ce nombre est pair")
  else :
    print("Ce nombre est impair")
```

Travail à faire

1. Intégrer la nouvelle fonction `exo3` dans le module `tp1.py`
2. Appeler cette fonction dans le `main` de `tp1.py`.
3. Tester différentes valeurs afin de vérifier la justesse du code. (Si l'utilisateur ne saisit pas un nombre, le programme s'arrête brutalement - c'est normal, la gestion des cas d'erreurs sera abordée utlérieurement)

EXERCICE 4

On souhaite proposer une variante de la fonction `exo2` de sorte que, si l'utilisateur ne fournit pas d'identité, le programme lui attribut d'office le nom "anonymous".

Voici une version en pseudo-code founit par un de vos collègues.

Listing 6. Version pseudo-langage

```
Afficher("Entrez votre nom svp :")
nom <-- lire une chaîne de caractère au clavier
Afficher("Entrez votre prenom svp :")
prenom <-- lire une chaîne de caractère au clavier
Si nom est vide Alors
  Afficher("Bonjour anonymus !")
Sinon
  Afficher("Bonjour " + prenom + " " + nom + " !")
FinSi
```

Travail à faire

1. Étudier la version en pseudo-langage ci-dessus puis proposer une traduction fidèle en Python.
2. Travaillez à partir d'une copie de la fonction `exo2` que vous nommerez **`exo4`**, puis appelez cette fonction dans le main.
3. Tester votre code

EXERCICE 5

L'algorithme proposé par votre collègue dans l'exercice précédent manque de logique. Avez-vous repéré ce qui cloche ?

Si l'utilisateur ne décline pas son identité à la demande de son nom, alors le programme ne devrait pas lui demander son prénom.

Proposez une amélioration de la fonction `exo4`, que vous nommerez `exo5`, qui respecte cette nouvelle logique.

Ce que vous devez faire :

1. Fournir d'abord une version en pseudo-langage
2. Faire valider votre version par un professeur
3. Traduire votre version en Python (une nouvelle fonction nommée `exo5`)
4. Tester et mettre au point votre fonction

EXERCICE 6

Transcrire le programme suivant en une fonction Python (nommée `exo6`).

Listing 7. Version pseudo-langage

```
n <- 60
m <- 7
afficher("Les entiers valent ", m , "et ", n)
afficher("leur somme est ", m+n)
afficher("leur différence est ", m-n)
afficher("leur produit est ", m*n)
afficher("leur quotient est ", m/n)
afficher("le reste de la division entière m/n est ", m modulo n)
```

Puis améliorer la fonction `exo6` de sorte que l'utilisateur puisse lui-même fournir des valeurs pour les zones mémoire référencées par les identificateurs `m` et `n`. Vérifier la justesse des sorties.

EXERCICE 7

Cet exercice introduit la notion de type. En effet, toute variable est associée, à un instant *t*, à un et un seul type. Le type de la variable est déterminé par l'interpréteur au moment de l'affectation et peut être consulté à l'exécution par un appel à la fonction **`type`**.

Listing 8. Comment connaître le type d'une variable

```
>>> x = 42
>>> type(x)
<class 'int'>   # <== le type de x est int
```

Commençons par définir une fonction qui réalise une somme de 2 entiers reçues en argument. Nous appellerons cette fonction `somme`.

```
def somme(arg1: int, arg2: int) -> Int :
  """
    Return la somme des arguments
  """
  # affecte à la var result le résultat de l'opération +
  result = arg1 + arg2
  return result ①
```

① On remarquera que la fonction "n'affiche" rien. C'est très important. **Le fait d'afficher ou non la valeur retournée est de la responsabilité de l'appelant, pas de l'appelé** (voir Glossaire Appelant/Appelé)

Voici un exemple de programme (une fonction) qui appelle la fonction `somme` (ligne 25)

```
def exo7() -> None:
  print("Bonjour, je suis un programme écrit en Python.")

  # invite l'utilisateur à entrer un nombre entier
  # l'information est stockée dans une zone mémoire
  # référencée par 'str_n1'
  str_n1 = input("Entrez un nombre entier : ")

  # affiche une information sur le type de l'objet crée
  print("Le type de l'objet crée est ", type(str_n1))

  # n1 est l'image de str_n1 par la fonction int(). Le rôle de int()
  # est de tenter de traduire son argument en une valeur
  # numérique (un entier).
  n1=int(str_n1)

  # affiche une information sur le type de l'objet crée
  print("Le type du nouvel objet crée est ", type(n1))

  # idem
  n2 = input("Entrez un second nombre entier : ")

  # appel à la fonction somme, définie plus haut,
  # en vue de réaliser une addition (normalement pb de type ici)
  res = somme(arg1 = n1, arg2 = n2) ①

  # affichage du résultat
  print("La somme des deux nombres est : ", res)

  # dernière instruction pour une fin annoncée
  print("bye, je meurs...")
```

```
if __name__ == "__main__":
  import sys
  exo7()
  sys.exit(0)
```

① On remarquera l'usage des valeurs `n1` et `n2` comme valeurs d'arguments de la fonction `somme`. Une autre façon d'appeler la fonction est de passer les valeurs par position, par exemple : `res = somme(n1, n2)`, qui aura même effet.

Travail à faire

1. Adapter le script `tp1.py` (ajout de la fonction `somme` et `exo7`)
2. Tester et **comprendre pourquoi la fonction `exo7` bugue**
3. Corriger la fonction `exo7`
4. Modifiez la fonction `exo7.py` afin qu'elle réalise la somme de 3 nombres.
5. Faire évoluer la fonction `exo7.py` afin qu'elle réalise, en plus de la somme de 3 nombres, le produit de ces 3 nombres. Pour cela vous devrez créer, juste après la déclaration de `somme()`, une nouvelle fonction nommée `produit()`, inspirée de `somme()`.

EXERCICE 8 (FINAL)

A l'issue de cette première séance de travaux pratiques, vous avez appris à **programmer des fonctions** simples en Python, à **les appeler** dans la partie *main* du script/module `tp1.py`.

Votre mission : Au lancement de `tp1.py`, permettre à l'utilisateur de choisir la fonction qu'il souhaite exécuter parmi les fonctions `exo2()`, `exo3()`, ..., `exo7()` du module.

Travail à faire

1. Ajouter une fonction nommée `main`. Son rôle sera de répondre à cette demande.
2. Faire en sorte que le code du *main* de `tp1.py` appelle cette nouvelle fonction.
3. Tester le tout

Contrôler vos connaissances et contribuer aux QCMs

Travail à faire

1. Contrôler vos connaissances sur quizbe.org. (choisir `PYTHON-LDV`, scope `p-1-intro`)
2. Enrichir la base de données QCM. Pour cela, proposer, pour le thème `PYTHON-LDV` (scope `p-1-intro`), 2 questions QCM originales et personnelles, sur des thèmes couverts pas cette séquence d'exercices. **Il est important d'associer un feedback à chacune des réponses proposées, qu'elles soient justes ou fausses**.

Glossaire

variable

symbole dans le code, dont le **nom témoigne de son rôle**, qui désigne selon son utilisation soit un **contenant** (une adresse mémoire) soit une **valeur de contenu** (interprétée selon son **type**)

type

Pour l'interpréteur, désigne comment utiliser et stocker la valeur d'une variable.
Pour l'utilisateur, limite l'exploitation de la variable aux opérations définies sur ce type.

Listing 9. La fonction 'type' retourne le type de son argument

```
>>> type(42)
<class 'int'>
>>> type(42) is int
True
>>> type(42) is str
False
>>> type("42") is str
True
>>>
```

Voir la discussion ici : https://stackoverflow.com/questions/4843173/how-to-check-if-type-of-a-variable-is-string

valeur d'une variable

interprétation du contenu de (l'espace occupé par) la variable, selon son type.

Exemple : `x = "42"` ou `y = 42`;

La valeur de `x` est `"42"` est son type est **string**

La valeur de `y` est `42` est sont type est **int**

Ces deux variables pointent vers des espaces mémoire peu comparables en terme de structure binaire et de taille.

Constante littérale

Valeur inscrite dans le code source

Ces valeurs sont typées. Exemple : `'ANNEE'` `"ANNEE"` `"""ANNEE"""` sont des constantes littérales de type **string**

Convention de nommage

Les conventions usuelles

Table 1. Les conventions courantes

nom	*usage*
UpperCamelCase	nom de classe

nom	*usage*
lowerCamelCase	propriété, méthode, variable, paramètre… en Java, JS, PHP…
lower_snake_case	nom de variable et fonction en python
SCREAMING_SNAKE_CASE	constante
kebab-case	ressource web, identifiant, attribut html

return

Directive signifiant la fin de l'exécution du corps de la fonction où elle est présente. Peut être suivie d'une valeur qui sera retournée à l'appelant.

Appelant

Instruction (un bout de code) qui appelle une fonction en se fiant à son entête. Exemple : `x = len(y)` est une instruction **appelant** l'exécution de la fonction `len`.

Appelé

Un sous-programme identifié par un nom (nom d'une fonction par exemple) Exemple : `x = len(y)`, `len` est la fonction **appelée**. Dans cet exemple, la valeur de retour de la fonction appelée est stockée, par l'appelant, dans une variable nommée `x`.

Entête d'une fonction

Nommée également **interface**, définit le nom de la fonction, ses paramètres éventuels et son type de retour (ou `None`). Exemple (première ligne) :

```
def bidon() -> None :
  pass
```

Corps d'une fonction

Nommée également **implementation**, définit le travail de la fonction, et sa valeur de retour, si attendue. La séquence d'instructions est placée dans le bloc indenté qui suit l'entête de la fonction.

Algorithme déterministe

(sensibilisation) Une fonction, pour une même donnée d'entrée, doit invariablement produire la même sortie.

Muable

Une structure de données **muable** signifie que les variables de ce type peuvent sont susceptibles d'avoir leur valeur évoluée dans le temps. Exemple `list`, contre exemple `str` (ne peut pas être modifiée après création)

Immuable

Une structure de données **immuable** signifie que les variables de ce type ont une valeur qui ne peut être modifiée après leur création. Exemple `str` ou `tuple`, ne permettent pas l'ajout d'éléments, et leurs éléments ne peuvent être ni modifiés ni supprimés. Contre exemple `list`

(accepte l'ajout, et les éléments d'une instance de `list` peuvent être modifier, supprimer)