UNIVERSITÉ JEAN MONNET SAINT-ÉTIENNE

Info. TP1: Rappels

- http://learn.heeere.com/python
- http://learn.heeere.com/2018-infospichi-8a82
- http://learn.heeere.com/2018-info-8c99 (cours de L1)

Rendu : sous forme d'archive

À la fin, vos réponses seront rendues dans une archive zip : cette archive contiendra un fichier `compte-rendu.txt` (avec votre **nom**, **groupe**, et les réponses aux questions précédées d'une apostrophe) et vos **fichiers pythons**.

Pour faire une **archive zip** de votre TP pour le rendu, exécutez, dans le terminal :

```
cd ~/info-spichi/
zip -r tp1.zip tp1/
```

Important : mise en place (5 minutes)

- travaillez dans un dossier dédié au TP, lui même dans `~/info-spichi`,
- créez un fichier `compte-rendu.txt` pour écrire la date, vos noms et les réponses aux questions précédées d'une apostrophe,
- pour chaque exercice, créez un dossier et placez y les fichiers python que vous écrivez.

$\mapsto$ **Aide** $\mapsto$ Il faut lancer un terminal et y taper les commandes (les `# ...` sont des commentaires) :

```
cd                          # Retourner à la racine de notre compte
mkdir info-spichi           # Créez le dossier si besoin
cd info-spichi              # Se déplacer dedans
mkdir tp1                   # Créer le dossier du TP
cd tp1                      # Se déplacer dedans
touch compte-rendu.txt      # Créer un fichier vide
emacs compte-rendu.txt &    # L'ouvrir avec l'éditeur emacs
```

Exercice 1 – Introduction et téléchargement (5 minutes)

Dans ce TP, vous allez utiliser la bibliothèque graphique qtido. Il vous faudra donc télécharger depuis le site du cours le fichier `qtido.py`. Voir le site du cours (ou directement http://learn.heeere.com/python/reference-qtido/) pour une liste des fonctions de la bibliothèque qtido.

Q1) Téléchargez la bibliothèque graphique depuis le site du cours (dans « outils divers »). Vous devrez plus tard copier ce fichier `qtido.py` dans le dossier de votre programme pour pouvoir l'utiliser.

Exercice 2 – Range et boucles `for` (15 minutes)

Rappel : pour cet exercice, travaillez dans un dossier `ex2` lui même dans le dossier `tp1`.

Q2) Écrivez un programme `quatre.py` qui affiche les entiers de 20 (inclus) à 200 (exclus), de quatre en quatre, en utilisant une boucle `for`, la fonction `range()` et la fonction `print()`.

|↦ **Aide** |↦ La fonction `input()` permet de bloquer le programme en attendant que l'utilisateur tape une ligne de texte. Cette fonction prends une chaîne de caractère en paramètre qui s'appelle *l'invite* car elle sert à *inviter* l'utilisateur à taper quelque chose.

La fonction `input()` renvoie une valeur qui est la chaîne de caractères entrée par l'utilisateur. Il existe de fonctions de conversion qui permettent d'interpréter une chaîne de caractère en nombre entier (`int()`) ou en nombre réel (`float()`) mais aussi de convertir des nombres en chaîne de caractères (`str()`). Voici un exemple utilisant `input` et des conversions.

```
1 age_string = input("Quel est votre age ? ")
2 suivant = int(age_string) + 1
3 suivant_str = str(suivant)
4 print("Vous allez avoir " + suivant_str + " ans !")
```

Q3) Écrivez un programme `lepas.py` qui utilise la fonction `input()` pour demander 1 entier (`pas`) à l'utilisateur et ensuite affiche les entiers de 20 (inclus) à 200 (exclus), de `pas` en `pas`.

Q4) Écrivez un programme `tout.py` qui demande 3 entiers (`debut`, `fin`, `pas`) à l'utilisateur avec 3 appels à la fonction `input()` et affiche les nombres de `debut` (inclus) à `fin` (exclus), par pas de `pas`.

|↦ **Aide** |↦ Nous allons utiliser la fonction `split` qui permet de prendre une chaîne de caractères et de la découper en sous-partie, par exemple en utilisant l'espace comme séparateur. Si l'on écrit par exemple `li = str.split("bonjour toto !", " ")`, alors la valeur retournée (`li`) sera une liste contenant 3 éléments : les chaînes de caractères `bonjour`, `toto` et `!`. On peut écrire indifféremment `str.split("a b c", " ")` et `"a b c".split(" ")`. Cela fonctionne aussi bien avec n'importe quelles expression (variables, etc).

Q5) Écrivez un programme `toutenun.py` qui demande 3 entiers (`debut`, `fin`, `pas`) à l'utilisateur grâce à un seul appel à la fonction `input()` et utilise la fonction `split()` puis affiche les nombres de `debut` (inclus) à `fin` (exclus), par pas de `pas`.

Exercice 3 – Technique de l'accumulateur (25 min)

Rappel : pour cet exercice, travaillez dans un dossier `ex3` lui même dans le dossier `tp1`.

Dans cet exercice, chaque programme doit demander un entier « n » à l'utilisateur et afficher le résultat. Supposons que l'on veut par exemple calculer la somme $S_n = 1 + 2 + 3 + \cdots + n$ avec la technique de l'accumulateur. Nous allons créer une variable (l'accumulateur) que l'on mettra à jour progressivement grâce à chaque élément de la somme.

```
1 acc = 0
2 for element in range(1,n+1) :
3     acc = acc + element
4 print(acc)
```

Cette technique fait donc apparaître 3 parties :

- ligne 1 : initialisation avec la valeur « neutre pour la somme » c'est à dire 0,
- ligne 2 : parcours de la liste des entiers des 1 à n (liste retournée par `range(1,n+1)`,
- ligne 3 : mise à jour de l'accumulateur, à chaque fois,
- ligne 4 : exploitation de la valeur calculée (ici, simplement pour afficher sa valeur).

Dans les questions ci-dessous, utilisez la technique de l'accumulateur après avoir demandé la valeur de n à l'utilisateur.

Q6) Écrivez un programme `fractions.py` permettant de calculer $Q_n = \frac{1}{2} + \frac{2}{3} + \cdots + \frac{n}{n+1}$.

Q7) Écrivez un programme `factoriel.py` permettant de calculer $n! = 1 \times 2 \times 3 \times \cdots \times n$.

$\mapsto$ **Aide** $\mapsto$ L'opérateur `%` calcule le reste de la division entière (appelé aussi « modulo »), par exemple `121 % 50` (« 121 modulo 50 ») vaut `21`. On peut aussi remarquer que `350 % 50` vaut `0`, ce qui signifie que 350 est un multiple de 50.

Q8) Écrivez un programme `plusmoins.py` permettant de calculer $A_n = 1 - 2 + 3 - \cdots \pm n$.

Q9) Écrivez un programme `moyenne.py` permettant de calculer $\mu_n = \frac{1+2+3+\cdots+n}{n}$.

Q10) Challenge : `rerere.py` doit calculer $R_n = 1 + \cfrac{n}{1+\cfrac{n-1}{1+\cfrac{\cdots}{1+\cfrac{2}{1+\cfrac{1}{1+0}}}}}$ (avec n divisions).

Q11) Challenge : `escargot.py` doit calculer $F_n = F_{n-1} + F_{n-2}$ (définie par rapport à elle même) avec comme cas particulier que $F_0 = 1$ et $F_1 = 1$.

Exercice 4 – Structure globale (45 min)

Rappel : pour cet exercice, travaillez dans un dossier `ex4` lui même dans le dossier `tp1`.

Structure typique d'un programme avec fonction « principale »

Un programme Python doit être proprement structuré avec 4 sections : 1) les imports des modules utilisés par le programme, 2) la définition des fonctions nécessaires, 3) la définition de la fonction principale, qui contient le code de base du programme, 4) l'appel de la fonction principale.

Voici un exemple qui trace trois carrés (à l'aide d'une fonction qui doit être créée puisqu'elle n'existe pas).

```
1
2  # 1) importation de bibliothèques
3  from qtido import *
4
5  # 2) définition des fonctions
6  def carre(fen, x, y, taille):
7      rectangle(fen, x, y, x+taille, y+taille)
8
9  # 3) définition de la fonction principale
10 def principale():
11     f = creer(600, 500)
12     carre(f, 100, 100, 100)
13     carre(f, 400, 100, 100)
14     carre(f, 225, 300, 150)
15     attendre_pendant(f, 1000) # afficher pendant 1000 millisecondes
16     exporter_image(f, "toto.png") # sauver/écrire un fichier image
17
18 # 4) appel de la fonction principale
19 principale()
```

Q12) En respectant cette structure, écrire un programme `diag.py` qui trace une diagonale à l'aide de carrés (par exemple avec 50 carrés de taille 10) depuis en haut à gauche vers en bas à droite.

Q13) Idem mais dans `taille.py` et en demandant à l'utilisateur de rentrer la taille des carrés (un nombre).

Q14) Idem mais dans `liste.py` et en demandant à l'utilisateur de rentrer la taille de chacun des carrés, sous forme d'une liste de nombres, par exemple `10 9 8 7 6 10 8 6 4 2` (qui tracerait 10 carrés puisqu'il y a 10 valeurs).

Technique de la simulation

Pour programmer une simulation, il faut d'abord réfléchir à ce qu'est l'état du système à simuler et comment il doit être stocké (quelles variables, avec quels noms et quel contenu). Il faut ensuite réfléchir à l'état initial du système (valeur au début de la simulation). Ensuite il faut programmer : 0) l'initialisation des variables auxiliaires, 1) l'initialisation de l'état du système, 2) la boucle de simulation avec une condition d'arrêt (si elle existe), 3) dans la boucle, l'affichage du système, 4) dans la boucle, la gestion des événements (temps, touches clavier, etc.) qui **n'affiche pas** et ne fait que modifier l'état.

Voici un exemple de la partie **principale** d'un programme de simulation d'un disque qui grandit avec le temps et prends une taille au hasard s'il devient trop grand.

```
    # 0) initialisation des variables auxiliaires
    f = creer(400, 400)
    # 1) initialisation de l'état et autre
    rayon = 50
    # 2) boucle de simulation avec condition d'arrêt
    while not est_fermee(f):
        # 3) affichage du système
        effacer(f)
        disque(f, 200, 200, rayon)
        # 4) gestion des événements : ici, modifier l'état
        attendre_evenement(f, 10) # attente de 10 millisecondes
        ev = dernier_evenement(f)
        if ev == None: # temps écoulé
            rayon = rayon + 2
            if rayon > 200:
                rayon = random.random()*100
        elif ev == TOUCHE_ECHAP: # autre événement
            exit()
```

Q15) Écrire un programme de simulation `lineaire.py`, dans lequel une balle se déplace à vitesse constante vers la droite.

Q16) Idem mais dans `gravite.py`, en ajoutant la gravité vers le bas qui change la vitesse verticale, selon le bilan des forces.

Q17) Outils : Si c'est pas déjà fait, essayez les outils mentionnés à la fin des transparents du « cours-02 » (sur le site du cours), en particulier `inter.py` et `pytohtml.py`.