I_{00} Méthode de programmation

PCSI 2024 – 2025

I Exécution d'un programme python

1. Expression

Une expression est la notation d'un calcul (au sens large). Lors de l'exécution du programme, l'expression sera réduite à (remplacé par) une valeur (au sens large : nombre, mais aussi chaine de caractères, liste etc...). Lorsque l'ordinateur fait le calcul de l'expression, on dit qu'il l'évalue.
Par exemple si l'on tape `m.cos(-2 + 2)`, python va d'abord évaluer (-2 +2) , puis remplacer cette expression par son résultat et évaluer `m.cos(0)`, puis exécuter le reste du programme en faisant comme si on avait tapé directement `1.0` (la valeur de l'expression).

2. Instruction

Une instruction est la notation d'un « ordre » pour l'ordinateur qui ne renvoie pas de résultat. Pour autant, une instruction n'est pas sans effet : elle modifie « l'état interne de l'ordinateur ». En terme de vocabulaire, on dit que l'ordinateur exécute les instructions.
On peut citer l'affectation : `a = [1, 2]` associe l'étiquette "a" à une zone dans la mémoire qui contiendra la liste `[1, 2]`. Cette instruction ne renvoie aucune valeur et ne fait que changer l'état de la mémoire.
On peut aussi avoir une modification de l'affichage à l'écran, production d'un signal vers les hauts-parleurs etc...

3. Exécution d'un programme

Un programme est une suite d'instructions faisant passer l'état de l'ordinateur de l'état initial à l'état final .
De façon générale, sauf structures particulières, un programme s'exécute de haut en bas ligne par ligne. À chaque ligne, les expressions sont évaluées (en suivant un ordre de priorité lorsqu'il y a plusieurs opérations), puis remplacé par leur valeur avant l'exécution de l'instruction.
Pour observer l'exécution d'un programme, vous pouvez utiliser l'excellent outil en ligne python tutor :
http://www.pythontutor.com/
Démonstration 1

II Portées des variables

Lorsque le programme devient un petit peu gros, il y a beaucoup de fonctions. Dans chaque fonction, on utilise des noms de variables (i,j, mavariable). Il ne faut pas qu'il y ait de conflit entre ces différentes variables, ainsi les variables n'existent en général qu'à l'intérieur de la fonction et sont « effacées » à la sortie.

1. Principe et exemples

En programmation (et en python), on distingue deux sortes de variables : les variables globales et les variables locales.
Par exemple, dans le programme

```
>>> def f():
...     y = 8
...     return 3*y
>>> f()
24
>>> y
NameError: name 'y' is not defined
```

la variable y est locale, c'est-à-dire définie uniquement à l'intérieur de la fonction.
On dit que la portée de la variable y est limitée au corps de la fonction f.

⚠ Si on veut « forcer » une variable à être définie dans tout le programme, il faut la définir dans la fonction comme variable globale.
C'est en général une mauvaise pratique.

```
def reinitialise_x():
    x = 0
```

ne fait qu'affecter 0 à une variable locale x, cette variable n'a de sens qu'à l'intérieur de la fonction réinitialise. Au contraire

```
def reinitialise_x_qui_marche():
    global x
    x = 0
```

Considère qu'une variable x existe à l'extérieur de la fonction et va modifier sa valeur.

```
x = 5
reinitialise_x()
print(x) #affiche 5, le x de la fonction est local
reinitialise_x_qui_marche()
print(x) #affiche 0 : la fonction vient modifier la variable globale
```

Python tutor

2. Spécificité de python : définition implicite

Lorsque l'on utilise une variable dans une fonction sans lui avoir préalablement affecté une valeur, python la considère implicitement comme globale.
Exemple :

```
def f():
    a = b + 1 #b utilisé sans être défini dans la fonction : variable
       implicitement déclarée comme globale
    return a #a est une variable locale car définie dans la fonction
b = 3
print(f() ) #affiche 4
```

⚠ **Évitez autant que possible les variables globales, et faites leur porter un nom explicite assez long afin de bien les différencier.**

3. Cas particulier des listes

Lorsque l'on écrit $L2 = L1$, $L2$ ne contient pas une copie de la liste $L1$ mais pointe vers la même liste.

```
L1 = [1,2,3,1,2,3]
L2= L1 # L2 pointe vers la même liste que L1
L1[ 3 ] = 5 # on modifie la liste vers laquelle pointe L1
print(L2) # [1,2,3,5,2,3] la liste vers laquelle pointe L2 est aussi
   modifiée puisque c'est la même !
L2 = L1[:] # on crée une nouvelle liste qui est la copie de L1
L1[ 3 ] = 0
print(L2) # non modifié [1,2,3,5,2,3]
```

Lien Python tutor

Lors du passage en argument d'une liste à une fonction, le comportement est le même.
Ainsi, avec une fonction, une liste se comporte un peu comme une variable globale : ATTENTION !
Lien Python tutor

```
L = [1,2,3,1,2,3]
def f(liste):
    liste[0] = -10
    return None
f(L)
print(L) #[-10,2,3,1,2,3]
```

4. Conséquences

Il faut donc être vigilant lors de l'utilisation de liste passée en argument d'une fonction et se poser la question « a-t-on le droit ou non de modifier la liste en dehors de la fonction ? ».

Par exemple, pour tester si une liste est triée, un de vos camarade a d'abord codé une fonction `tri`, puis a proposé la fonction :

```
def est_trie(L):
    if tri(L) == L:
        return True
    return False #équivalent à return tri(L) == L
```

Remarque : c'est une mauvaise idée à de nombreux point de vue, en particulier du point de vue du temps de calcul car trier la liste est plus couteux que de vérifier directement si elle est triée. Mais ici mon but est d'illustrer le fait que la fonction renvoyait toujours `True`.

La fonction `tri` était codée d'une façon qui ressemble à :

```
def tri(L):
     ...
        temp = L[i]
        L[i] = L[i+1]      #notation python simplifiée
        L[i+1] = temp     #L[i], L[i+1] = L[i+1],L[i]
    ... # il faut savoir faire la version non simplifiée
    return L
```

En fait ici, le **return** `L` est facultatif parce que la liste a déjà été modifiée. C'est ce qui explique que le programme de votre camarade renvoyait toujours True. Python évalue `tri(L)` qui renvoie une liste contenant les objets triés, mais a comme effet de bord de modifier la liste `L`. Donc ensuite lorsque l'on évalue `tri(L)== L`, la liste L a été modifiée et on teste l'égalité entre deux fois exactement la même chose, que la liste soit initialement triée ou non.
Cela peut être le comportement désiré (exemple .append sur une liste modifie la liste et renvoie None) ou non. Si on ne souhaite pas modifier la liste de départ, il faut en faire une copie en commençant la fonction tri par `res = L[:]` puis[1] en travaillant sur `res` et non pas sur `L`.

1. Attention, c'est une copie superficielle, ne fonctionne pas si `L` est une liste de liste. Utiliser alors le module deepcopy.

III Commentaires et spécifications des entrées

1. Intérêt

Il est important de commenter son code et de spécifier très clairement ce que doivent être les entrées et les sorties de ses fonctions pour plusieurs raisons, notamment :

- comprendre son propre code lorsque l'on travaille dessus un peu plus tard ;
- permettre le travail en groupe (très important, on ne fait presque rien tout seul, la plupart des programmes que vous utilisez = plusieurs programmeurs pendant plusieurs mois)
- se forcer à justifier oblige à réfléchir à « pourquoi ce que l'on fait est correct » et évite donc des erreurs en voulant aller trop vite. À l'oral, exemple de Baptiste et moi qui nous nous expliquions nos code pour trouver les bugs

Remarque : on apprendra à démontrer que notre code est correct dans un prochain chapitre : les variants et invariants de boucle pourront se présenter en commentaire.

2. Spécification

Lorsque l'on écrit une fonction, il est très important de réfléchir à quel type d'objet on attend et quelles sont les limites sur ces objets[2]. Cette spécification peut être indiquée de plusieurs façons. En python le moyen naturel est dans la « docstring » : chaine de caractères sur plusieurs lignes au début d'une fonction qui explique le comportement de la fonction. C'est à cette docstring que vous avez accès en faisant help(nomDeLaFonction) dans un interpréteur.

```
def f(x, n):
    """docstring = chaine de char accessible via l'aide
    fonction qui prend en argument un nombre x et un entier n
    et qui renvoie x**n via un algorithme d'exponentiation rapide
    """
    ...
```

On peut aussi indiquer `def f(x:float, n:int)->float:` mais ça ne se substitue pas aux commentaires.
remarque à l'oral : dans les problèmes de concours, cela généralement spécifié dans la question, donc docstring un peu inutile, mais si vous introduisez une fonction vous même, très important de le faire.

IV Assertions

Python dispose de différents moyens pour arrêter un programme en cas d'entrée invalide. Nous utiliserons les assertions[3] : `assert expressionBooléenne`
On écrit `assert` suivi d'une expression booléenne qui doit renvoyer `True` dans le cas d'un comportement normal de notre programme.
Par exemple dans le programme précédent, si l'on ajoute `assert type(n)== int and n>=0` juste en dessous de la docstring, cela bloquera l'exécution du programme si le paramètre n n'est pas un entier ou s'il s'agit d'un entier strictement négatif.

2. C'est une cause de problèmes de sécurité fréquent : un utilisateur maladroit ou malintentionné envoie des données non prévues dans un programme

3. Remarque : ce n'est pas tout à fait ainsi que vous ferez pour vérifier la validité des arguments d'une fonction si vous faites du python de façon professionnelle. En effet, le comportement de l'assertion n'est pas « spécifié » lorsque le booléen est False est ce n'est prévu qu'à des fins de débogage. Toutefois le but est de vous sensibiliser à l'importance de la validation des entrées.

V Tests

Une erreur étant vite arrivée malgré tout le soin que l'on peut apporter aux étapes précédentes, il est important[4] de tester ses fonctions avec des valeurs dont on sait ce qu'elles doivent donner en sortie.
Il faut essayer de faire autant que possible des tests « exhaustifs », gérant tous les types de cas possibles.

Par exemple, si on veut tester une fonction qui nous indique si une liste est triée, tester dans le cas de la liste `[1,2,3]` et être content d'avoir True n'est pas suffisant ! En effet, le code pourrait très bien être le suivant :

```
1 def est_trie(L:list)->bool:
2     return True
```

Ainsi la fonction semble correcte tant que l'on ne test qu'avec une liste triée.
Il existe différentes techniques pour tester ses fonctions, nous retiendrons deux idées importantes : le partitionnement des domaines d'entrées et les test des valeurs limites.

Partitionnement des domaines d'entrées : il s'agit de diviser les entrées du programme en un nombre fini de sous-ensembles qui doivent se comporter sensiblement de la même façon. Par exemple dans le cas du test du caractère trié ou non d'une liste on peut imaginer comme sous-ensembles : les listes triées, les listes non triées, les listes d'un seul élément[5], la liste vide.
Si notre fonction risque de recevoir des données d'un utilisateur, il faut penser à tester avec des entrées non valides (i.e. des entrées qui ne vérifie pas les hypothèses spécifiées dans la docstring, que ce soit sur le type ou sur les contraintes) pour tester la sécurité.

Test des valeurs limites : il faut bien entendu tester le cas « standard », c'est-à-dire un cas typique de ceux que l'on doit rencontrer, mais des erreurs se trouvent souvent dans les cas limites (listes vides, $n = 0$ si on a fait l'hypothèse $n >= 0$, le premier terme pour la suite de Fibonacci). Par exemple, si l'on a spécifié que $n \leqslant N$, alors il est judicieux de tester le comportement pour $n = N$, mais aussi pour $n = N - 1$ et $n = N + 1$.

Exemple/exercice : proposer un assert à mettre en début de fonction et des tests (valeurs en entrées et résultats correspondant attendu) pour une fonction qui répondrait au problème[6] ci-dessous.

Une fonction qui renvoie le montant du salaire d'un employé en fonction du nombre d'heures travaillées, sachant que :

- Les 20 premières heures sont sans suppléments et valent 10 euros chacune.
- De la 21^e à la 25^e heure le salaire horaire est majoré de 50% (attention, seules les heures supplémentaires sont majorées, les 20 premières heures sont toujours payées au même tarif).
- Au-delà de 25 heures, le salaire horaire est majoré de 100%.

assert (type(n) == int or type(n) == float) and n>=0 and n<=70 ; prendre la limite légale dans le pays considéré à la place de 70
Test : tester avec :

— 15 heures $\rightarrow$ 150 (cas de base, on peut en prendre un autre) ;
— 30 $\rightarrow$ 365 (cas de base, on peut en prendre un autre) ;
— 23 $\rightarrow$ 335 (cas de base, on peut en prendre un autre) ;
— 0 $\rightarrow$ 0 (cas limite) ;
— 20 $\rightarrow$ 200 (cas limite) ;

4. D'après un document que j'ai pu trouver, la vérification et la validation représente entre 30% - 50% du coût de développement de logiciels.
5. sous-entend que dans les cas précédent on se limitait aux listes de plusieurs éléments
6. Valeurs non-réalistes simplifiées pour que les calculs puissent être fait de tête.

- $21 \to 215$ (cas limite) ;
- $25 \to 265$ (cas limite) ;
- $26 \to 285$ (cas limite).

Si on risque de recevoir des données d'un utilisateur : test de cas non valides

- -10 (hors du domaine prévu)
- "physique" (mauvais type d'objet

Utilisation de bibliothèques : bien que l'on vous apprenne cette année à reprogrammer des algorithmes classiques, il est en général plus pertinent d'utiliser des fonctions déjà présentes dans des bibliothèques. Un des avantages (en plus du gain de temps) est qu'elles sont utilisées par de nombreuses personnes et donc que de nombreuses valeurs ont déjà été testées. Réciproquement, si vous écrivez du code qui risque d'être utilisé par un grand nombre de personne, vous avez une grande responsabilité et vous devez faire en sorte qu'il soit fiable et sûr même dans les cas limites ou non prévus.

Table des matières