Chapitre :
Langages et programmation

I. Constructions élémentaires

1. Instructions

Définition Un **programme** est la description d'un **algorithme** dans un **langage** compréhensible par un humain et par une machine qui l'exécute afin de traiter des données.

Les programmes sont donc écrits par des humains, et les machines les interprètent et les exécutent.

Définition Les données traitées par les machines sont stockées en mémoire dans des **variables**. Une variable est donc l'association d'un espace mémoire avec un nom. Le nom d'une variable est composé d'une chaîne de caractères alphanumériques qui ne doit pas être un mot réservé (correspondant à une instruction par exemple), ni commencer par un chiffre.

Une variable, dans certains langages comme Python, peut avoir un **type**, qui définit l'ensemble des valeurs qui peuvent lui être affectées. Nous reviendront plus précisément sur les types plus tard, mais on peut déjà évoquer les type `int` pour les entiers et `float` pour des nombres décimaux.
Le type `bool` (pour booléen) est celui des variables qui valent soit Vrai soit Faux.
Le type `str` (pour string) est celui des chaînes de caractères, écrites entre guillemets.
Pour chaque type de variable, certaines fonctions sont définies pour les manipuler.
Il existe également des types composés, que nous verrons plus en détail plus tard, comme les **listes**.

Définition Une **expression** est une combinaison de variables, opérations, etc. qui donne un résultat d'un certain type.

Exemples Une expression dont le résultat est de type int : `2*3//4`
Une expression dont le résultat est de type str : `"abc"+"def"`

Définition Une **instruction** est une commande que la machine exécute.

Le langage Python a été créé par Guido Van Rossum en 1991, son nom est un hommage à la troupe d'humoristes anglaise Monty Python.
À l'adresse https://docs.python.org/fr/3/library/stdtypes.html se trouve la description des types standards de Python, avec les fonctions et méthodes associées. Il faut connaître celles des types cités plus haut, ainsi que les listes, sur lesquels nous reviendrons.

Voir : fiche des instructions élémentaires avec leur traduction en Python.

▶ **Exercice :** fichier Python notebook de traduction d'algorithmes en langage Python.

2. Fonctions

Définition Une **fonction** est une suite d'instructions dépendant éventuellement de paramètres (ou arguments) qui porte un nom qui permet d'y faire appel dans un programme.
En Python, la syntaxe est la suivante (attention à l'indentation) :

```
def NomDeFonction(argument1,...):
    Instructions
    return Resultat
```

L'instruction `return` permet d'indiquer la variable (ici `Resultat`) ou l'expression qui est renvoyée par la fonction. L'exécution de cette instruction stoppe immédiatement l'exécution de la fonction, y compris si elle se trouve en plein milieu du code, d'une boucle en particulier. Elle n'est pas nécessaire, et si il n'y en a pas, l'appel de la fonction a la valeur `None`.
Une fonction qui ne prend pas d'argument est notée avec des parenthèses vides.

Exemple Voici un exemple de deux fonctions très simples, dont l'une fait appel à l'autre.

```
def surface(r):
    S=3.14*r**2
    return S

def main():
    rayon=float(input("Donner une longueur de rayon de disque :"))
    S=surface(rayon)
    print("La surface du disque vaut environ",S)
```

 La **portée des variables** est une notion importante en programmation. Tout paramètre ou toute variable définie dans une fonction, sans instruction supplémentaire, est **locale**, c'est à dire que sa valeur n'est accessible que dans la fonction elle-même. La valeur d'une variable portant le même nom en dehors de la fonction n'est pas modifiée.

Exemple

```
x=2

def modifie(x):
    x=x+2
    return x

print(modifie(x))
print(x)
```

La première valeur affichée sera 4, alors que la seconde sera 2 (la valeur de `x` en dehors de la fonction). Autrement dit la valeur de `x` dans la fonction n'est pas la même que la valeur de `x` à l'extérieur de la fonction. Si l'on souhaite qu'une variable `x` d'une fonction soit la même qu'une variable définie en dehors de celle-ci, il faut ajouter dans la définition de la fonction la ligne de code suivante :

```
    global x
```

▶ **Exercices :** Fiche TD sur Python

▶ **Exercices :** Fiche d'exercices sur Python

II. Divers langages de programmation

Cette section fait l'objet d'exposés.

III. Spécification, tests et assertions

La spécification est, en Python, un commentaire permettant d'expliquer à l'utilisateur le rôle d'une fonction, de préciser les contraintes d'usage et les propriétés du résultat retourné par la fonction. Elle s'écrit entre triples guillemets (ou triples apostrophes) comme ci-dessous :

Exemple Voici un code Python avec une spécification :

```
def fonction(x):
    '''Cette fonction prend comme argument un nombre (float ou int)
    et renvoie son carré'''
    return x**2 # Retour de la fonction ; commentaire non affiché
```

La spécification est affichée lorsque l'on fait appel à l'aide sur une fonction donnée.

Exemple Voici trois affichages de spécifications :

```
>>> help(fonction)
Help on function fonction in module __main__:

fonction(x)
    Cette fonction prend comme argument un nombre (float ou int)
    et renvoie son carré

>>> help(max)
Help on built-in function max in module builtins:

max(...)
    max(iterable, *[, default=obj, key=func]) -> value
    max(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its biggest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the largest argument.

>>> help(list.append)
Help on method_descriptor:

append(self, object, /)
    Append object to the end of the list.
```

C'est également ce texte qui est affiché dans une bulle une fois que l'on a tapé, dans l'interpréteur, le nom d'une fonction puis une parenthèse.
Ce texte peut également être appelé **prototype** ou **signature**, bien que cette notion puisse être plus précise (voire obligatoire) avec d'autres langages de programmation (où l'on va préciser le type des données d'entrées et le type de la sortie).
Pour pouvoir certifier qu'une fonction fait bien ce que l'on attend d'elle, sans faire de démonstration formelle, il est bien de pouvoir au minimum la tester, en utilisant des **jeux de tests**.

Exemple On crée une fonction prenant en argument une liste :

```
def permute(l):
    '''Cette fonction prend comme argument une liste
    et échange son premier et son dernier élément.
    Exemple : si l=[1,2,3,4], alors après permute(l)
    la liste l est [4,2,3,1]'''
    l[0],l[-1]=l[-1],l[0]
    return l
```

On teste alors cette fonction avec divers arguments :

```
>>> permute([1,2,3,4])
[4, 2, 3, 1]
>>> permute([[1,2],[3,4],[5,6]])
[[5, 6], [3, 4], [1, 2]]
>>> permute(['a'])
['a']
>>> permute([])
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    permute([])
  File "<pyshell#14>", line 6, in permute
    l[0],l[-1]=l[-1],l[0]
IndexError: list index out of range
```

On voit ici une erreur, puisque l'algorithme ne prend pas en compte la possibilité que la liste soit vide. Il y a deux manières de réagir à cela : soit on modifie l'algorithme (en renvoyant par exemple la liste vide si l'argument est la liste vide), soit on ajoute une **assertion**, indiquant qu'il y a une condition (plus précisément ici une **précondition**) pour appliquer la fonction : que la liste soit non vide (sinon ça n'a pas de sens de permuter...).
Cela se fait avec le mot clé assert en Python :

```
def permute(l):
    '''...'''
    assert l!=[]
    l[0],l[-1]=l[-1],l[0]
    return l
```

À l'exécution de la fonction sur une liste vide, on obtient alors là aussi un message d'erreur, indiquant quelle assertion n'est pas vérifiée :

```
>>> permute([])
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    permute([])
  File "<pyshell#23>", line 6, in permute
    assert l!=[]
AssertionError
```

On peut également utiliser des assertions pour décrire les **postconditions**, c'est à dire les conditions à satisfaire par les variables en sortie. Ces notions de précondition et postcondition, ainsi que celles de variants et invariants que nous avons vues en algorithmique, sont le cadre de la programmation par contrat.
En principe les assertions s'utilisent plutôt en phase de test d'un programme. Ensuite, pour éviter qu'un programme se termine sur une erreur prévisible, on cherche à l'éviter, ou à la gérer (en utilisant une structure du type `try ... except`).
Pour les tests à effectuer, on peut soit les créer « à la main », soit les créer automatiquement, selon le type de fonction et leurs arguments que l'on a à tester. On peut prendre des jeux de données déterminés ou des jeux de données aléatoires. L'idée est également de faire des jeux de données dépendant de la structure algorithmique de la fonction testée (faire en sorte que tous les cas de test soient vérifiés, par exemple).
Faire des fonctions qui testent des fonctions est utile pour cela.

 Même si l'on fait beaucoup de tests, on ne peut pas certifier que la fonction ne renverra pas d'erreur sur un jeu de données précis. Seule une preuve formelle peut le faire.

Exemple Pour clore la section, voici un exemple de spécification plus précise que celles données plus haut, détaillant les préconditions et postconditions, ainsi que les types d'entrée et de sortie :

```
def pgcd(a, b):
    """Le pgcd de a et b.
    Préconditions:
        a > 0
        b > 0
    Postconditions
        a % __return__ == 0    # __return__ : la valeur retournée
        b % __return__ == 0
        result le plus grand entier qui divise a et b
    :param a: premier entier
    :type a:  int
    :param b: deuxième entier
    :type b:  int
    :return:  le pgcd de a et b
    :rtype:   int
    """
```

▶ **Exercices :** fiche sur les tests

IV. Modules et bibliothèques

Nous avons déjà vu l'utilisation de modules dans Python, et qu'il faut les importer avec l'instruction `import`.
utiliser la fonction `help` permet d'avoir de la documentation sur les modules (seulement une fois qu'ils ont été importés).
Nous avons pu voir quelques modules, comme `random` permettant d'obtenir des fonctions d'aléa ou `math` permettant l'utilisation de fonctions mathématiques particulières. Il en existe énormément pour Python. Voici une courte liste :

PIL : permet de manipuler des images

tkinter : permet de faire des interfaces graphiques

turtle : permet de créer une interface de dessin, implémentant un moyen utilisé il y a de nombreuses années pour introduire l'algorithmique auprès d'enfants. Voir ici la page Wikipedia (anglaise)

numpy : étend le langage Python pour manipuler des matrices, ou tableaux à plusieurs dimensions, et avec des fonctions permettant de les manipuler.

matplotlib : permet entre autres de faire des tracés de courbes, mais aussi de surfaces.

⊛ **Activité :** Choisir (par groupe) un de ces modules (éventuellement un hors de la liste, pourvu qu'il soit disponible sans installation complexe) et faire un mini-projet l'utilisant. Le but principal est de se familiariser avec le module et d'en montrer quelques possibilités. On pourra chercher des idées et sources sur Internet, mais le travail devra être compris. Le code devra être commenté. D'autre part, les fonctions définies dans le code devront être munies de spécifications.