Chapitre :
Langages et programmation

∽

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. En NSI, nous utiliserons presque exclusivement le langage **Python**, créé par Guido Van Rossum en 1991, dont le nom est un hommage à la troupe d'humoristes anglaise Monty Python.

I. Éléments du langage Python

1. Introduction

Définition Dans un programme, 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.

En Python, comme dans bien d'autres langages de programmation, une variable a un **type**, qui caractérise l'ensemble des valeurs qui peuvent lui être affectées. Nous reviendront plus précisément sur certains types ailleurs, mais on peut déjà évoquer rapidement les types simples suivants :

- `int` pour les entiers (`12`) et `float` pour une partie des nombres décimaux (`3.4`), dits flottants ;
- `bool` (pour booléen) est celui des variables qui valent Vrai (`True`) ou Faux (`False`) ;
- `str` (pour string) est celui des chaînes de caractères, écrites entre guillemets (`"abcd"`).

Pour chaque type de variable, certaines fonctions (ou méthodes) sont définies pour les manipuler.
Il existe également des types composés, que nous verrons plus en détail plus tard, comme les **listes**.

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.
En première il faut connaître celles des types cités plus haut, y compris les listes.

2. Expressions

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

Les expressions numériques, en particulier, peuvent-être construites à l'aide d'opérations dont les plus fréquentes sont les suivantes :

somme	`x+y`
différence	`x-y`
produit	`x*y`
puissance	`x**n`
division (décimale)	`x/y`

Exemples
L'expression `x**n` est le résultat de x^n, donc `2**3` vaut $2^3 = 2 \times 2 \times 2 = 8$.
L'expression `7/2` vaut `3.5`

Il existe deux autres opérations sur les entiers à connaître, liées à la division euclidienne entière. Soit `n` et `d` deux entiers. Alors :

quotient	`n // d`
reste	`n % d`

Exemples
`7//2` vaut `3` car dans 7 il y a 3 fois 2.
`7%2` vaut `1` car le reste de la division de 7 par 2 vaut 1 ($7 = 3 \times 2 + 1$).

Il y a au moins deux opérations simples sur les chaînes de caractères à connaître :

- La **concaténation**, qui consiste à accoler deux chaînes, se fait avec la somme (+) entre deux chaînes.
- La multiplication, qui consiste à concaténer plusieurs fois la même chaîne, se fait avec le produit (∗) d'un entier par une chaîne.

Exemples
L'expression `"abc"+"def"` vaut `"abcdef"`
L'expression `3*"ab"` vaut `"ababab"`

Pour vérifier le type d'une expression, on utilise la fonction `type()`.

Exemples

```
>>> type(5)
<class 'int'>
>>> type(3.5)
<class 'float'>
>>> type("abc")
<class 'str'>
```

Il existe des fonctions de conversion :

- Pour convertir une chaîne de caractère en nombre, que ce soit entier ou flottant, on utilise respectivement les fonctions `int()` et `float()`.

  Exemples

  ```
  >>> int("5")
  5
  >>> float("5.3")
  5.3
  >>> float("5")
  5.0
  ```

- Dans l'autre sens, pour convertir un nombre en chaîne de caractère, on utilise la fonction `str()`.

  Exemples

  ```
  >>> str(35)
  '35'
  >>> str(5-9.4)
  '-4.4'
  ```

3. Instructions

Définition Une **instruction** est une commande que la machine exécute, qui généralement change l'état de la mémoire ou réalise une action, comme un affichage.

Exemples Une instruction affectant une valeur à une variable `x` : `x = 7`
Une instruction permettant d'affecter des valeurs simultanément à deux variables : `a,b=15,3`
Une instruction qui permet d'afficher une valeur : `print(a)`

Voici ci-dessous le tableau des instructions élémentaires et structures algorithmiques avec leur traduction dans la syntaxe du langage Python, à connaître parfaitement :

pseudo algorithmique	Python	Remarques

Saisir X	`X=input("X=?")`
	ou
	`X=int(input("X=?"))`
	ou
	`X=float(input("X=?"))`
		X est une chaîne de caractères
		
		X est un entier
		
		X est un flottant

X←Y	`X=Y`	X prend la valeur Y

Afficher X
 	
Afficher "abcd"
	`print(X)`
	
	`print("abcd")`
		X peut être à peu près
		de n'importe quel type
		Voir la fonction `print` plus loin.

Si condition Alors
| ...
| ...
FinSi
...
	`if condition:`
	`    ...`
	`    ...`
	`...`
		Les instructions à exécuter si la condition est vraie sont indentées
		
		le retour à l'indentation de départ indique la fin

Sinon Si
 	
Sinon
	`elif condition:`
	`    ...`
	`else:`
	`    ...`
		indentation au même niveau que `if`
		
		même chose

Pour I allant de 1 à N Faire
| ...
| ...
Fin Pour
...
	`for I in range(1,N+1):`
	`    ...`
	`    ...`
	`...`
		Les instructions à exécuter dans la boucle sont indentées
		le retour à l'indentation de départ indique la fin
		Voir après les paramètres de `range`

Tant que condition Faire
| ...
Fin Tant que
...
	`while condition:`
	`    ...`
	`...`
		Même principe que pour `if` et `for` pour l'indentation

Quelques précisions sur l'utilisation de la fonction `range` qui permet de générer des suites de nombres régulières et qui est utilisée généralement avec les boucles `for` :
`range(d,f,p)` génère les entiers de `d` (inclus) à `f` (exclu), avec un pas de `p`. Par défaut, `d=0` et `p=1`.

Exemples
`range(10)` signifie `range(0,10,1)` et génère les entiers de `0` à `9`.
`range(2,7)` signifie `range(2,7,1)` et génère les entiers de `2` à `6`.
`range(2,9,3)` génère les entiers `2;5;8`.
`range(5,-1,-1)` génère les entiers `5;4;3;2;1;0`.

À savoir : la fonction `input()` retourne une chaîne de caractère. C'est pour cela que, lorsque l'on demande un entier, ou un flottant, on doit l'utiliser en combinaison avec les fonctions de conversion `int()` ou `float()`.

Revenons d'autre part sur les **conditions** nécessaires dans les instructions conditionnelles `if`, `elif` et la boucle `while`. Ce sont des expressions **booléennes**, dont la valeur est `True` ou `False`.
On peut comparer des nombres :

- `x == y` vaut `True` si `x` est égal à `y`. Noter qu'il s'agit d'un « double égal », car l'égalité simple est l'instruction d'affectation et n'est pas comme ici une expression booléenne ;
- `x != y` vaut `True` si `x` est différent de `y` ;
- `x >= y` vaut `True` si `x` est supérieur ou égal à `y`. De manière similaire, on a `x>y`, `x<=y` et `x<y`.

On peut de plus faire des opérations booléennes, autrement dit composer des tests de la manière suivante :

- `c1 or c2` est `True` si et seulement si `c1` **ou** `c2` est vraie (éventuellement les deux : ou inclusif) ;
- `c1 and c2` est `True` si et seulement si `c1` **et** `c2` sont vraies ;
- `not c1` est la négation (le contraire) de `c1`.

Les tests sont toujours réalisés de la gauche vers la droite et s'arrêtent dès que le résultat peut être connu sans réaliser les suivants.
Dans le cas de tests complexes, il faut mettre des parenthèses pour éviter les erreurs de priorité.

Enfin, précisons l'usage de la fonction `print` qui permet de l'affichage de chaînes de caractères, mais également de valeurs de variables.
Elle peut prendre plusieurs arguments. Dans ce cas, par défaut elle va afficher chacun d'eux séparés par un espace et elle termine par un retour à la ligne.

Exemple

```
>>> x = 5
>>> print("abc",3,"4",x)
abc 3 4 5
>>> for i in range(3):
...     print(i)
...
0
1
2
```

On peut modifier ce comportement par défaut en ajoutant en paramètre la valeur du caractère de séparation (`sep`) et du caractère de fin (`end`), qui sont en fait des chaînes de caractères. Par défaut, on a `sep=" "` (un espace) et `end="\n"` (un retour à la ligne).

Exemple

```
>>> print(3,"4",sep="->")
3->4
>>> for i in range(3):
...     print(i,end=" ")
...
0 1 2
```

Il est également possible de construire des chaînes de caractères spéciales (des `f-string`, que l'on reconnaît avec la lettre `f` écrite juste avant la chaîne de caractère) permettant de faire des phrases contenant des valeurs de variables, en évitant de donner une succession d'arguments à la fonction.

Exemple

```
>>> x = 3
>>> print("La valeur de x est",x)
La valeur de x est 3
>>> print(f"La valeur de x est {x}")
La valeur de x est 3
```

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

▶ **Exercices :** Fiche d'exercices sur Python, Instructions élémentaires 1 et 2

4. 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 les définitions de deux fonctions :

```
def ma_fonction(x):
    if x > 0:
        return x**2+1 # on peut retourner une expression
    else:
        return 0

def une_fonction():
    n = int(input("Donnez un nombre : "))
    print(f'vous avez entré la valeur {n}')
```

La seconde ne prend pas d'argument, et elle ne retourne aucune valeur (même si elle affiche du texte). Pour exécuter une fonction, il suffit d'écrire son nom et, entre parenthèses, les valeurs de ses arguments. Les parenthèses sont nécessaires, même quand la fonction n'a pas besoin d'argument. Pour enregistrer la valeur de retour, on l'affecte à une variable :

```
>>> ma_fonction(5)
26
>>> valeur = ma_fonction(4) # valeur vaut alors 17
>>> valeur-10
7
```

Remarque L'interpréteur affiche la valeur retournée (il affiche toujours les valeurs des expressions qui lui sont fournies), mais ce n'est pas ce qui se passe quand on exécute un fichier contenant la même instruction. Pour l'afficher, on doit utiliser la fonction `print()` :

```
print(ma_fonction(5))
```

 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.
Il est possible cependant de déclarer des variables comme globales avec la syntaxe `global x`.

► **Exercices :** Fiche d'exercices sur Python (Fonctions)

La fiche de cours sur les types contient des éléments de manipulation des tuples (`n`-uplets), listes et dictionnaires.

II. Divers langages de programmation

Cette section fait l'objet d'exposés réalisés par les groupes d'élèves.

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).
Une forme de signature différente peut être donnée dans la définition d'une fonction, où l'on précise le type des arguments et le type de la sortie.

Exemple

```
def egaux(x: int, y: float) -> bool:
    return float(x) == y
```

Cependant il ne s'agit que d'une annotation, et l'interpréteur ne vérifie pas le bon type des éléments en entrée :

```
>>> egaux(5., 5) # mauvais types de nombres d'après la signature
True
```

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
```

Il est possible d'ajouter une chaîne de caractères à la suite de l'insertion, qui s'affiche dans le cas où l'assertion n'est pas satisfaite.

```
>>> assert [] != [], 'Liste vide'
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    assert [] != [], 'Liste vide'
AssertionError: Liste vide
```

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.

► **Exercices :** activité notebook avec `assert` ; fiche sur les tests (sans `assert`)

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 (pillow) : 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).

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

pygame : permet de réaliser des jeux graphiques dans un cadre simplifié.

⊛ **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.
Si beaucoup de possibilités se présentent, il est possible, voire conseillé, de faire plusieurs fichiers, chacun s'exécutant indépendamment des autres et montrant une facette particulière. Chaque membre d'un groupe peut donc prendre en charge un ou plusieurs aspects différents.
On pourra chercher des idées et sources sur Internet, mais le travail devra être compris. En particulier, le code devra être commenté, dans l'objectif de faire comprendre à quelqu'un ne connaissant pas ce module d'en expliquer le fonctionnement et l'intérêt.
D'autre part, les fonctions définies dans le code devront être munies de spécifications.