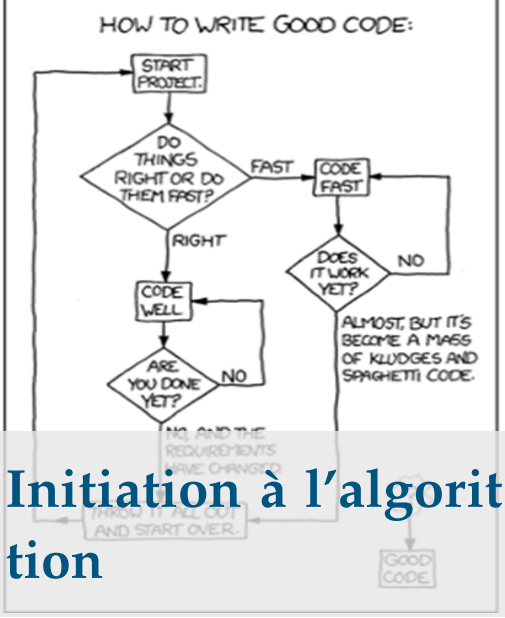


2 Initiation à l'algorithmique et à la programmation

2.1 Avant-Propos

2.1.1 Définitions

Définition – Informatique – Wikipedia

L'informatique est un domaine d'activité scientifique, technique, et industriel concernant le traitement automatique de l'information numérique par l'exécution de programmes informatiques hébergés par des dispositifs électriques-électroniques : des systèmes embarqués, des ordinateurs, des robots, des automates, etc.

Définition – Algorithmique – Wikipedia

Un algorithme est une méthode générale pour résoudre un type de problèmes. Il est dit correct lorsque, pour chaque instance du problème, il se termine en produisant la bonne sortie, c'est-à-dire qu'il résout le problème posé.

L'efficacité d'un algorithme est mesurée notamment par :

- sa durée de calcul ;
- sa consommation de mémoire vive (en partant du principe que chaque instruction a un temps d'exécution constant) ;
- la précision des résultats obtenus (par exemple avec l'utilisation de méthodes probabilistes) ;
- sa scalabilité (son aptitude à être efficacement parallélisé) ;
- etc.

Les ordinateurs sur lesquels s'exécutent ces algorithmes ne sont pas infiniment rapides, car le temps de machine reste une ressource limitée, malgré une augmentation constante des performances des ordinateurs. Un algorithme sera donc dit performant s'il utilise avec parcimonie les ressources dont il dispose, c'est-à-dire le temps CPU, la mémoire vive et (aspect objet de recherches récentes) la consommation électrique. L'analyse de la complexité algorithmique permet de prédire l'évolution en temps calcul nécessaire pour amener un algorithme à son terme, en fonction de la quantité de données à traiter.

2.1 Avant-Propos 1
2.2 Variables, Types 3
2.3 Fonctions 6
2.4 Structures algorithmiques 7
2.5 Tableaux et type `list` en `python` 11
2.6 Chaînes et type `str` en `python` 16

- Choisir un type de variable.
- Concevoir un algorithme utilisant une structure conditionnelle (Si), une structure itérative (while), une structure itérative (for).
- Concevoir une fonction.
- Manipuler des listes.
- Instruction et expression.

Définition – Programmation – Wikipedia

La programmation, appelée aussi codage dans le domaine informatique, désigne l'ensemble des activités qui permettent l'écriture des programmes informatiques. C'est une étape importante du développement de logiciels.
L'écriture d'un programme se fait dans un langage de programmation. Un logiciel est un ensemble de programmes (qui peuvent être écrits dans des langages de programmation différents) dédié à la réalisation de certaines tâches par un (ou plusieurs) utilisateurs du logiciel.

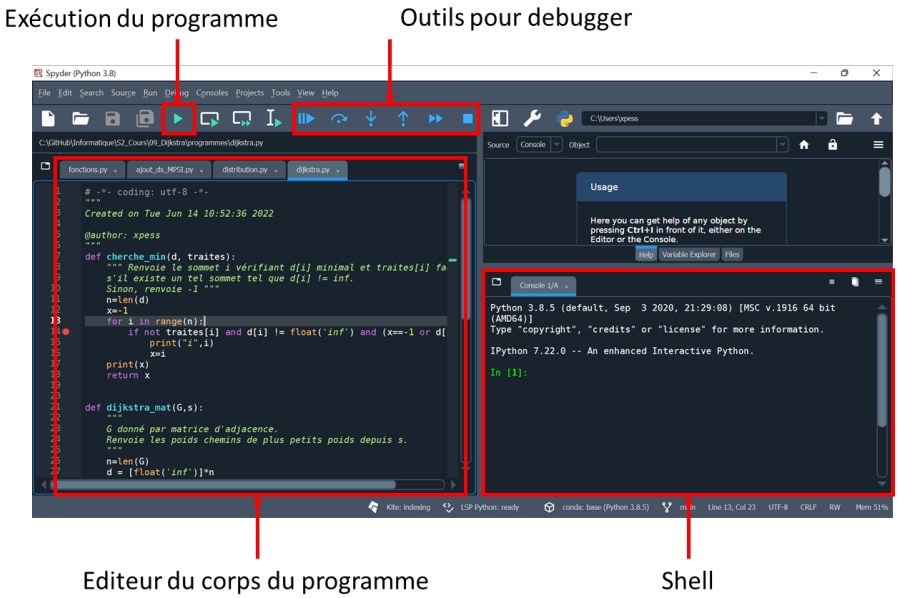
La programmation représente donc ici la rédaction du code source d'un logiciel. On utilise plutôt le terme développement pour dénoter l'ensemble des activités liées à la création d'un logiciel et des programmes qui le composent. Cela inclut la spécification du logiciel, sa conception, puis son implémentation proprement dite au sens de l'écriture des programmes dans un langage de programmation bien défini, ainsi que la vérification de sa correction, etc.

2.1.2 Environnement de développement

Pour répondre à un problème informatique, on pourra réfléchir à l'algorithme permettant de le résoudre. Une des premières approches consiste donc à utiliser un papier et un crayon pour proposer une réponse structurée au problème.

Il faudra ensuite traduire cet algorithme dans un langage de programmation. Nous utiliserons pour cela `Python`.

Pour programmer en `Python` il est nécessaire d'utiliser un éditeur de texte (pour écrire le code) et un interpréteur pour exécuter le code. En règle générale, on utilise pour cela un environnement de développement (IDE). On pourra **Pyzo** ou **Spyder** (ou tout autre environnement vous paraissant satisfaisant).



2.2 Variables, Types

2.2.1 Quelques définitions

Définition – Valeurs

Les valeurs désignent les données manipulées par un algorithme ou une fonction. Une valeur peut ainsi être : un nombre, un caractère, une chaîne de caractères, une valeur de vérité (`Vrai` ou `Faux`) *etc*.
En Python, comme dans la plupart des langages de programmation, ces valeurs sont **typées** selon l'objet qu'elles représentent : il y a ainsi des valeurs de **type** entier, flottant, chaîne de caractères, booléens ... Leur représentation en mémoire varie beaucoup d'un langage à l'autre, mais ce sont souvent les mêmes objets que l'on cherche à traduire.

Définition – Expression

Une expression est une suite de caractères faisant intervenir des valeurs et des opérations, afin d'obtenir une nouvelle valeur. Pour calculer cette nouvelle valeur, la machine doit évaluer l'expression. Voici des exemples d'expressions : `42, 1+4, 1.2 / 3.0, x+3`.

En Python, pour évaluer une expression, il suffit de la saisir dans un interpréteur (console, shell), qui calcule et affiche alors la valeur qu'il a calculée.

```
>>> 42
    42
>>> 1+4
    5
```

Définition – Variable

Une **variable** désigne une zone mémoire de l'ordinateur dans la RAM. Il s'agit d'un endroit où l'on peut **stocker** une valeur, y **accéder** et **changer** cette valeur.
Pour faire référence à une variable, on utilise un nom de variable, en général composé d'une ou plusieurs lettres. Dans la mesure du possible, on choisit un nom explicite, ce qui permet une meilleure lecture du programme.

2.2.2 Types simples

En programmation, associer un type à une valeur permet :

- de classer les valeurs en catégories similaires. Par exemple, tous les entiers (type int), tous les flottants (type float)...
- de contrôler quelles opérations sont faisables sur ces valeurs : par exemple, tout entier (type int) pourra être utilisé dans une soustraction, ..., alors qu'une chaîne ne le pourra pas. Il sera également impossible d'additioner un entier et un booléen.

Une expression n'a a priori pas de type, car le type de la valeur qu'elle renvoie dépend des types des sous-expressions. Ainsi `a+b` sera un entier (resp. un flottant, une chaîne) si `a` et `b` le sont, mais sera un flottant si `a` est un entier et `b` un flottant.

Pour afficher le type d'une valeur ou d'une expression après l'avoir évaluée, on utilise `type`.

```
>>> type(42)
    int
>>> type(1.2/3.0)
    float
```

Définition – Entiers

Ce type est noté `int` en Python .

- **Constantes :** ce sont les entiers relatifs écrits en base 10. Ils ne sont pas bornés en Python .
- **Opérateurs :**
  1. + : addition usuelle ;
  2. − : soustraction usuelle (`15-9` renvoie `6`), mais aussi opposé (`-4`) ;
  3. // : division entière : `a//b` correspond au quotient de la division euclidienne de `a` par **b** si `b` est strictement positif (`256 // 3` renvoie `85` car $256 = 85 * 3 + 1$). Mais si `b` est strictement négatif, alors `a//b` renvoie ce quotient moins 1 ($15 = (-4) \times (-4) - 1 = (-4) \times (-3) + 3$, le quotient de la division euclidienne de 15 par −4 est −3, mais `15//-4` renvoie `-4`). Cette division n'est pas définie si `b` est nul ;
  4. % : modulo : même remarque que dans le point précédent, relativement au reste de la division euclidienne cette fois (`256 % 3` renvoie `1`, `15%-4` renvoie `-1`).
  5. ** : exponentiation (`2**3` renvoie `8`).

Les règles de précédence, autrement dit les règles de priorité entre opérations, sont similaires aux règles mathématiques usuelles.

Définition – Flottants

Ce type est noté `float` en Python .

- **Constantes :** ce sont les nombres à virgule flottante. Nous en donnerons une définition précise dans un chapitre ultérieur : pour simplifier, disons pour l'instant que ce sont des nombres à virgule, avec un nombre borné de chiffres dans leur écriture.
- **Opérateurs :**
  1. + : addition usuelle ;
  2. − : soustraction usuelle, et aussi opposé.
  3. / : division usuelle.
  4. ** : exponentiation.

Définition – Expression – Booléen

Ce type est noté `bool` en Python.

- **Constantes :** il y en a deux : `True` et `False`.
- **Opérateurs :**
  1. `not` : négation usuelle ;
  2. `and` : conjonction usuelle ;
  3. `or` : disjonction usuelle.
- **Opérateurs de comparaison :**
  1. == : test d'égalité : `2==3` renvoie `False`, `4==4` renvoie `True`.
  2. != : `a != b` est une écriture équivalente à `not (a == b)`.

3. `<,>,<=,>=` : ce à quoi on s'attend.

Exemple –

`True or False` renvoie `True`.
`not(False or True) and True` renvoie `False`.

Remarque

Python permet les chaînes de comparaisons : `1<2<3` renvoie `True`, et `(1<2<-2) and (-5<2<6)` renvoie `False`.

2.2.3 Conversions – *Cast*

Il est possible de convertir en entier en flottant avec la fonction `float`. La réciproque est possible dans une certaine mesure : `float(2)` renvoie `2.0`, `int(2.0)` renvoie `2`, mais `int(2.6)` renvoie `2` et `int(-3.6)` renvoie `-3`.

La fonction `int` appliquée à un nombre flottant procède donc à une troncature, ce n'est pas la fonction donnant la partie entière d'un nombre flottant (fonction `floor` du module `math`).

2.2.4 Conversions

On peut convertir des types simples en chaînes avec la fonction `str` : `str(2.3)` renvoie `'2.3'`.
À l'inverse, `int, float` et `bool` permettent de faire l'inverse quand cela est cohérent : **`bool('True')`** renvoie `True`, mais `int('2.3')` renvoie une erreur.

Il est également possible de passer d'un type composé à un autre grâce aux fonctions `str`, `tuple` et `list` : `str((1,'a',[1,2]))` renvoie `"(1,'a',[1,2])"`, `tuple([1,2,3])` renvoie `(1,2,3)` et `list('Coucou ?')` renvoie `['C', 'o', 'u', 'c', 'o', 'u', ' ', '?']`.

2.2.5 Affectation

Définition – Affectation

Quand on stocke une valeur `d` dans une variable `var`, on dit que l'on **affecte** `d` à la variable `var`. La valeur `d` est encore appelée **la donnée**.

En Python , cette affectation est faite avec la commande `=`, comme suit.

```
>>> var  = 1
```

Les variables (et donc l'affectation) sont incontournables : si vous ne stockez pas une donnée (résultat par exemple d'un calcul), vous ne pourrez pas la réutiliser, ni la modifier.

```
>>> x = 42                  # x prend la valeur 42.
>>> x = x + 2               # On additionne à la donnée stockée dans x
                            # le nombre 2 et on stocke le résultat
                            # de nouveau dans x.
```

```
>>> x                    # on accède à la valeur qui est mémorisée dans x.
    44
```

Dans la variable affectée, la nouvelle donnée « écrase » la donnée précédente : on perd son ancienne valeur.

Dans un programme Python, on utilisera l'instruction `print(x)` pour afficher dans la console la valeur de la variable `x`.

Il est également possible d'affecter des valeurs à plusieurs variables en une fois, en utilisant des tuples. Ainsi : `a,b = (1,2)` affecte la valeur 1 à `a` et la valeur 2 à `b`.

Remarque

En Python, le **typage** des variables est **dynamique**. Cela signifie que lorsqu'on affecte une valeur à une variable, Python reconnaît automatiquement le type de la valeur. Ce n'est pas le cas de tous les langages de programmation.

Notion d'adressage

En Python, le passage des variables se fait par **référence**. C'est-à-dire que lorsqu'on manipule une variable (qu'on l'envoie à une fonction par exemple), on ne donne pas comme argument la valeur de la fonction, mais son adresse mémoire. Pour les types non mutables (entiers, flottants, chaîne de caractères, tuples), le passage par référence ne pose pas de problèmes. Pour les types mutables (listes), le passage par référence peut conduire à des résultats « inattendus » notamment lors de la copie de listes.

```
>>> a=2
>>> print(a)
    2
>>> a=3
>>> print(a)
    3
>>> a=[1,2]
```

```
>>> print(a)
    [1,2]
>>> b=a
>>> a[0]=2
>>> print(a,b)
    [2,2] [2,2]
```

Ici l'affectation `b=a` ne crée pas une nouvelle liste `b`. On crée juste une nouvelle variable `b` qui a la même adresse mémoire que `a`. Ainsi en modifiant `a`, on modifie `b`.

2.3 Fonctions

2.3.1 Objectif : modularité des programmes

Pour répondre à un problème donné, il est souvent nécessaire d'enchaîner plusieurs voire un nombre important d'instructions. Pour améliorer la lisibilité d'un programme et aussi pour pouvoir réutiliser ces instructions classiques (dans le même programme ou dans un autre), on privilégie les programmes simples (ou sous-programmes) appelés **fonctions**.

Dans chaque langage, il y a déjà un nombre incalculable de fonctions déjà construites (en Python, par exemple, `print`) mais on s'aperçoit très vite que l'on a besoin de créer ses propres fonctions.

Remarque

- Pour bien se faire comprendre, il est important de choisir un nom de fonction explicite. Il faut aussi mettre des commentaires pour expliquer ce que fait la fonction.
- Il est nécessaire de bien cibler les paramètres dont on a besoin c'est-à-dire les données nécessaires à l'exécution de la fonction.
- Il faut repérer ce que renvoie la fonction : rien (exemple : un simple affichage ou la modification d'un fichier...), un nombre entier, un flottant, une liste...
- Enfin, il faut toujours tester sa fonction sur un ensemble significatif de valeurs pour repérer d'éventuelles erreurs ou manquements.

2.3.2 Écriture en Python

```
def nom_de_la_fonction(parametres):
    """ commentaire expliquant la fonction """
    #bloc d'instructions
    return(resultat)
```

Remarque

En Python , l'indentation est signifiante : après le mot-clé `def`, chaque ligne indentée fait partie de la fonction. La première ligne non indentée rencontrée marque la fin de la fonction : cette ligne ne fait plus partie de la fonction, mais celles qui précéde en font partie. Il est donc impératif d'indenter quand il le faut, et seulement quand il le faut.

Annotations des fonctions

Afin de préciser à l'utilisateur ou au lecteur de votre programmes quels sont les types d'arguments ou le type retourné par la fonction, il est possible d'utiliser des annotations. Celles-ci ont un rôle uniquement documentaire.

```
def foo(par: int) -> float :   # Ces annotations indiquent que la fonction
    prend
                               # comme un argument un entier
                               # et renvoie un float.
    """ commentaire expliquant la fonction """
    #bloc d'instructions
    return(2.*par)
```