Informatique PCSI
Écriture d'un programme

1 Instructions

1.1 Expression et affectation

■ *Une expression* est utilisée pour calculer une valeur ou pour appeler une procédure (une fonction qui renvoie la valeur `None`).

Une expression est composée de noms (ou identifiants), de nombres, de chaînes de caractères, d'éléments séparés par des signes de ponctuation, entourés de parenthèses, de crochets , d'accolades et d'opérateurs. Une expression a une valeur.

■ Une *affectation* est utilisée pour lier un nom à une valeur ou pour modifier un élément d'un objet mutable comme une liste. L'instruction *affecter* est représentée par le signe =.

La syntaxe est : `nom = expression` ou `nom[i] = expression`.

On peut aussi écrire : `n1, n2, ..., nk = seq` si `seq` est une séquence à `k` éléments.

En Python, une expression et une affectation sont deux instructions particulières.

1.2 Instructions

Une *instruction* est une morceau de code minimal qui produit un effet. Une instruction est exécutée par une machine.

Une *instruction simple* peut s'écrire sur une seule ligne. On peut en écrire plusieurs séparées par des points-virgules.

Outre l'expression et l'affectation, quelques instructions simples sont :

■ *affirmer* avec `assert`, qui est suivi d'une expression, (les assertions sont précisées plus loin dans ce chapitre) ;

■ *renvoyer* avec `return`, qui est suivi d'une expression et s'emploie uniquement dans une fonction, (une expression peut être un n_uplet écrit sans parenthèse) ;

■ *arrêter* avec `break`, mot isolé qui permet d'arrêter une boucle ;

■ *importer* avec `import`, suivi d'un nom de module, (ou de fonction, de constante, appartenant à un module qui est précisé).

Une *instruction composée* est une instruction sur une ligne terminée par deux-points suivie d'une ou plusieurs instructions indentées : on dispose par exemple de `if` (qui peut être suivi de `elif`, de `else`) pour exécuter des instructions selon une condition, de `for` et de `while` pour exécuter des instructions de manière répétée, de `def` pour définir une fonction.

Exceptée les expressions, une instruction n'a pas de valeur. Certaines instructions nécessitent une expression, comme avec le signe =, les mots `assert`, `if`, `elif`, `while`, qui sont suivis d'une expression, donc d'une valeur. Ces choix de conception du langage Python ont des conséquences à connaître.

En premier lieu, ces choix permettent d'éviter quelques erreurs de programmation qui pourraient passer inaperçues en première lecture. Si par exemple on écrit `assert x = 1`, ou `while x = 0`, ou `if (assert x > 0)`, ou `return x = 0`, cela provoque une erreur de syntaxe et le programme s'arrête. Ce sont des erreurs habituelles chez les novices et qui arrivent aussi simplement lorsqu'on veut écrire trop vite un programme.

Si par contre une instruction comme `x = 1` avait une valeur, et si on écrivait `while x = 1` par erreur à la place de `while x == 1`, on pourrait obtenir une boucle infinie (dans ce cas, on finirait par s'en rendre compte). Mais si on écrivait `if (x = 1)` au lieu de `if (x == 1)`, l'exécution du programme pourrait se poursuivre « normalement ». Et un bug peut-être difficile à déceler surviendrait plus tard.

En deuxième lieu, ces choix donnent de la liberté dans l'écriture et permettent des raccourcis. Les mots `if`, `elif`, `while`, `assert` doivent être suivis d'une expression. Cette expression n'a pas forcément

une valeur booléenne, mais quelle que soit sa valeur, celle-ci est interprétée comme une valeur booléenne. Un exemple complètement artificiel comme `assert not print("!!!")` ne provoque pas d'erreur. En effet, `print("!!!")` est une expression de valeur `None` et l'expression `not None` a pour valeur `True`. Toute nombre nul et tout conteneur vide est interprété comme une valeur `False`. C'est donc le cas pour `0`, `0.0`, `""`, `[]`, `()`, `{}`, et aussi pour `None`. Toute autre valeur est interprétée comme une valeur `True`.

Cela permet d'écrire par exemple `if x % 2` plutôt que `if x % 2 == 1` pour tester si un nombre `x` est impair, ou `while x` plutôt que `while x != 0` pour poursuivre une boucle tant que `x` n'est pas nul. Mais attention, cela peut conduire à d'autres types de difficultés et donc de bugs potentiels.

1.3 Effet de bord

On dit qu'une fonction a un *effet de bord* si son exécution modifie quelques chose en dehors de ce qui est défini dans le corps de la fonction, par exemple un de ses paramètres ou une variable globale définie dans le programme.

Un effet de bord, en anglais *side effect*, est un effet secondaire qui peut être désiré ou pas. Une fonction a un objectif principal. Et la suite des instructions écrites pour atteindre cet objectif peut avoir des effets sur les objets manipulés. C'est en particulier le cas avec les fonctions qui manipulent des objets mutables comme les listes. C'est un effet durable qui apparaît avec une fonction autrement que par la valeur renvoyée.

Dans le code qui suit, la première fonction a un effet de bord, pas la seconde.

```
def ajoute(liste, x):
    liste.append(x)

def ajoute(liste, x):
    liste = liste + [x]
    return liste
```

Avec la première fonction, la liste passée en paramètre est modifiée par la méthode `append`. Avec la seconde fonction, l'affectation crée une nouvelle liste (qui est une variable locale).

On peut utiliser l'une ou l'autre de ces fonctions mais en connaissance de cause pour éviter un éventuel bug.

Remarque : une fonction sans effet de bord peut être plus simple à tester.

Dans l'exemple qui suit, on définit deux fonctions `multiplie` qui prennent en paramètres une liste et un nombre.

```
def multiplie(liste, x):
    res = liste
    for i in range(len(res)):
        res[i] = x * res[i]
    return res

def multiplie(liste, x):
    res = liste
    return [x * res[i] for i in range(len(res))]
```

On définit une liste `liste1` et on écrit `liste2 = multiplie(liste1, 3)`. On obtient le même résultat pour `liste2` avec les deux fonctions mais `liste1` est modifiée avec la première fonction.

Remarque : un effet de bord peut être désiré, par exemple quand on utilise une liste ou un dictionnaire pour mémoriser des données.

2 Types

Le *type* d'une variable est le type de l'objet auquel peut être lié le nom de la variable. Il définit l'ensemble des valeurs autorisées, les opérations et les fonctions utilisables.

2.1 Types de base

Les types de base sont les types numériques `int`, `bool` et `float`.

Le type `int` est utilisé pour représenter les nombres entiers. Leur taille n'est limitée que par la capacité de la machine et le temps nécessaire à leur utilisation. Un calcul avec de très grands entiers est possible en Python mais peut prendre un temps important.

Le type `bool` permet de représenter les valeurs booléennes `True` (vrai) et `False` (faux). Ces valeurs peuvent être considérées comme les entiers 1 pour `True` et 0 pour `False`.

Le type `float` est utilisé pour représenter les nombres réels. Les nombres du type `float`, forcément en nombre fini, sont des décimaux qui servent à approximer les réels. Ils sont stockés dans la machine sous la forme de « nombres en virgule flottante » avec des valeurs comprises entre environ $-1,7 \times 10^{308}$ et $1,7 \times 10^{308}$.

Citons aussi le type `None` qui n'a qu'une seule valeur, la valeur `None`. C'est par exemple la valeur que renvoie une fonction sans instruction `return`.

- **Opérations sur les types numériques**

Nous avons les opérations mathématiques habituelles : addition, soustraction, multiplication, exponentiation et division notées respectivement `a + b`, `a - b`, `a * b`, `a ** b` et `a / b`.

La division entière `//` et l'opération modulo `%` utilisées avec des entiers naturels, donnent respectivement le quotient et le reste, de type `int`, dans la division euclidienne.

- **Opérateurs de comparaison et opérateurs booléens**

Les opérateurs mathématiques classiques de comparaisons $=$, $\neq$, $<$, $\leq$, $>$, $\geq$ s'écrivent en Python respectivement ==, !=, <, <=, > et >=.

`x == y` a pour valeur `True` si `x` et `y` ont la même valeur, sinon a pour valeur `False`.

`x != y` a pour valeur `True` si `x` et `y` n'ont pas la même valeur, sinon a pour valeur `False`.

Les opérateurs logiques sont `and`, `or`, `not` :

`a and b` a pour valeur `True` si `a` et `b` ont la valeur `True` et sinon a pour valeur `False` ;

`a or b` a pour valeur `False` si `a` et `b` ont la valeur `False` et sinon a pour valeur `True` ;

`not a` a pour valeur `True` si `a` a la valeur `False` et sinon a pour valeur `False`.

Ces opérateurs fonctionnent de manière séquentielle, on parle de « caractère paresseux ». Avec `a and b`, `b` n'est évalué que si `a` vaut `True`. Avec `a or b`, `b` n'est évalué que si `a` vaut `False`.

2.2 Le type str

C'est un type simple mais structuré. Le mot str est une abréviation de string. Ce type est utilisé pour représenter des chaînes de caractères, par exemple ce qui est obtenu dans la machine lorsqu'on appuie sur les touches du clavier. On utilise des guillemets ou des apostrophes comme `ch = "bonjour"` ou `ch = 'bonjour'`. Un caractère est une chaîne de longueur 1.

On obtient la longueur d'une chaîne, le nombre de caractères contenus dans la chaîne, avec la fonction `len`. Par exemple, `len("bonjour")` a pour valeur l'entier 7.

Chaque caractère d'une chaîne a un indice qui va de 0 à $n-1$ si n est la longueur de la chaîne et il est possible d'accéder au caractère d'indice `i` d'une chaîne `ch` avec la notation `ch[i]`.

Nous pouvons aussi accéder à une suite de caractères d'une chaîne `ch` avec la notation `ch[i:j]` qui est une chaîne de caractères contenant dans l'ordre les caractères de `ch` d'indices `i` inclus à `j` exclu. On parle aussi d'une *tranche*.

2.3 Types structurés composés

Ce sont par exemple les types `tuple`, `list`, `dict`.

Les dictionnaires de type dict sont décrits plus loin dans une section particulière.

Un objet de type list ou tuple est composé de zéro, un ou plusieurs objets, à priori de n'importe quel type. Ce sont des types *itérables* comme le type str.

Un objet de type list ou tuple représente un ensemble ordonné. Les éléments sont indexés par un entier commençant à 0 et se terminant à $n-1$ si n est le nombre d'éléments.

Ce nombre d'éléments est obtenu avec la fonction `len`.

L'accès à un élément particulier ou à une suite d'éléments s'obtient comme pour les chaînes de caractères. On utilise la syntaxe avec crochets : par exemple `x = liste[i]` permet d'affecter à `x` l'élément d'indice `i`.

Un point très important est qu'un élément d'une liste peut être modifié avec une instruction d'affectation alors que c'est interdit avec le type tuple (il n'est pas mutable). On peut écrire pour une liste `liste[0] = val`, mais avec un tuple ce type d'instruction provoque une erreur.

De nombreuses méthodes permettent de travailler avec des listes. La syntaxe doit être bien connue : le nom de la méthode s'écrit après le nom de l'objet auquel elle s'applique, les deux noms étant séparés par un point sans espace.

La méthode `append` est très souvent employée. L'instruction `liste.append(x)` ajoute, à la fin de la liste `liste`, l'objet `x`. Cette méthode sert en particulier à construire une liste en ajoutant les éléments un par un à la fin de la liste.

■ Opérations communes aux types list, tuple et str

▸ Appartenance

On utilise le mot `in`. L'expression `x in obj` a pour valeur `True` si `x` est un élément de `obj` et sinon a pour valeur `False`.

▸ Concaténation

Si `obj1` et `obj2` sont du même type, `obj1 + obj2` est la concaténation de `obj1` et `obj2`.

`n * obj` est la concaténation de n copies de `obj` si n est un entier naturel.

La fonction `type` permet de déterminer le type d'un objet. Les fonctions `int`, `float`, `str` permettent de convertir si c'est possible un objet d'un autre type respectivement en un objet de type int, float, str.

2.4 Les listes

En langage Python, une liste est un objet de type list qui est un ensemble ordonné d'éléments hétérogènes de types quelconques.

Les éléments d'une liste sont séparés par des virgules et entourés de crochets.

■ Génération

— En extension : `liste = [1, 3, 5, 7, 9]`

— En compréhension : `liste = [2*i+1 for i in range(5)]`

— Par conversion : `liste = list(range(1, 10))`

— Par ajouts successifs :
```
liste = []
for i in range(1, 10):
    liste.append(i)
```
De manière générale, on crée en compréhension une liste avec une instruction comme :
`liste = [f(u) for u in objet]`, où `f` est une fonction et `objet` est de type str ou list.

■ **Manipulation**

La fonction `len` : l'expression `len(liste)` a pour valeur la longueur de la liste, c'est-à-dire le nombre d'éléments de la liste.

On accède à chaque élément par des indices qui prennent les valeurs de `0` à `len(liste)-1`.
Avec `liste = [1, 3, 5]`, `liste[0]` vaut `1`, `liste[1]` vaut `3`, `liste[2]` vaut `5` et une écriture comme `liste[3]` provoque une erreur et l'arrêt d'un programme.

Soit : `liste = [1, 3, 5, 7, 9]`.
`liste[1:4]` vaut `[3, 5, 7]`.
`liste[:4]` vaut `[1, 3, 5, 7]` (par défaut l'indice de début `d` vaut 0).
`liste[1:]` vaut `[3, 5, 7, 9]` (par défaut l'indice de fin `f` vaut `len(liste)`).
Et donc : `liste[:]` vaut `[1, 3, 5, 7, 9]`.

Une liste vide : `liste = list()` ou `liste = []`.
Les opérateurs + et $\star$ concatènent des listes pour créer une nouvelle liste :
l'expression `[1, 2] + [3, 4]` vaut `[1, 2, 3, 4]` ;
l'expression `3 * [1, 2]` vaut `[1, 2, 1, 2, 1, 2]`.

Les opérateurs + et $\star$ n'ont pas d'effet de bord, de nouvelles listes sont créées.
L'expression `x in liste` vaut `True` si `x` est un élément de `liste`, sinon vaut `False`.
Le type list est mutable : on peut modifier la valeur d'un élément par une affectation.
Attention, en écrivant `liste = [1]` puis `liste = [3]`, on ne change pas la valeur du premier objet `[1]` auquel est lié `liste`, mais on crée un nouvel objet `[3]` auquel on lie `liste`. Par contre avec `liste = [2, 4]` puis `liste[1] = 6`, on change la valeur de `[2, 4]` qui devient `[2, 6]`.

Il existe plusieurs méthodes spécifiques aux listes. Les méthodes `append` et `pop`, modifient une liste respectivement en ajoutant un élément en fin de liste ou en supprimant et renvoyant l'élément en fin de liste. Par exemple :
```
liste = [1, 2, 3]
liste.append(4)   (liste vaut [1, 2, 3, 4])
liste.pop()   (a pour valeur 4, l'élément est retiré de la liste, liste vaut [1, 2, 3]).
```

On peut créer une liste de listes : `liste = [[1, 2], [3, 4], [5, 6]]`.
`liste[i][j]` est l'élément d'indice `j` de la liste d'indice `i`. Par exemple, `liste[1][0]` est l'élément d'indice 0 de la liste `[3, 4]`, soit le nombre 3.

■ **Parcours et itération**

Nous avons deux manières de parcourir les éléments d'une liste.
Si nous n'avons pas besoin de la valeur de l'indice :
```
liste = [1, 3, 5, 7, 9]
for u in liste:
    print(u)
```

Si nous avons besoin de la valeur de l'indice :
```
for i in range(len(liste)):
    print(i, liste[i])
```

2.5 Dictionnaires

Dans une liste, un n-uplet ou une chaîne de caractères, les éléments sont ordonnés. On les repère par leur indice. Si n est la longueur, à chaque entier de 0 à $n-1$ correspond un élément.

- **Définition**

Un dictionnaire, objet de type dict, est une association entre des clés et des valeurs. Les clés sont des objets non mutables, les valeurs des objets quelconques. Ces clés ne sont pas ordonnées. On accède à une clé en temps constant selon un processus qu'il n'est pas nécessaire de décrire ici. On accède à une valeur par sa clé.

Pour créer un dictionnaire, on écrit entre des accolades les couples `clé: valeur`, une clé et la valeur sont séparées par le signe deux-points, les couples sont séparés par des virgules.

Exemple : `jours = {"dimanche": 1, "lundi": 2, "mardi": 3, "mercredi": 4, "jeudi": 5, "vendredi": 6, "samedi": 7}`.

- **Manipulation**

La construction par compréhension existe pour les dictionnaires.

```
>>> d = {x: x**2 for x in range(1, 5)}
>>> d
{1: 1, 2: 4, 3: 9, 4: 16}
```

- **Accès aux clés**

La méthodes `keys` permet d'obtenir les différentes clés.

```
>>> jours.keys()
dict_keys(['dimanche', 'lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi',
           'samedi'])
```

La méthode `items` permet d'obtenir l'ensemble des couples (clé, valeur).

```
>>> jours.items()
dict_items([('dimanche', 1), ('lundi', 2), ('mardi', 3), ('mercredi', 4),
('jeudi', 5), ('vendredi', 6), ('samedi', 7)])
```

Le mot `in` permet de tester l'appartenance d'une clé à un dictionnaire, pas d'une valeur.

```
>>> 4 in jours
False
>>> "dimanche" in jours
True
>>>
```

Donc, avec `for elt in dic`, où `dic` est un dictionnaire, la variable d'itération `elt` est une clé.

```
>>> cle = []
>>> for elt in jours:
        cle.append(elt)
```

La variable `cle` a pour valeur `['dimanche', 'lundi', 'mardi', 'mercredi', 'jeudi', 'vendredi', 'samedi']`.

Mais il est aussi possible d'itérer sur les couples clés-valeurs en utilisant la méthode `items`.

```
>>> for cle, val in jours.items():
        print(cle, val)

dimanche 1
lundi 2
mardi 3
mercredi 4
jeudi 5
vendredi 6
samedi 7
```

L'accès à une valeur s'obtient comme avec les listes, les tuples ou les chaînes. Mais il faut préciser la clé à la place de l'indice. Par exemple, `jours['dimanche']` nous donne la valeur 1.

- **Insertion, modification**

Comme pour les listes, il est possible de modifier une valeur par affectation, par exemple avec une instruction comme `jours['dimanche'] = 0`. Pour insérer un élément, on écrit une instruction comme `jours['dimanche2'] = 8`. On ajoute un deuxième dimanche à la semaine.

Si la clé `c` existe, l'instruction `d[c] = val` modifie la valeur associée à la clé `c`. Si la clé `c` n'existe pas, elle est créée et la valeur `val` lui est associée.

- **Nombre d'éléments :** comme avec les listes, les n-uplets et les chaînes, la fonction `len` renvoie la longueur d'un dictionnaire qui est à la fois le nombre de clés et le nombre de couples.