— **Grammaire des types** —

Les éléments simples du langage (noms, symboles) sont en caractères `machine`, les ensembles d'éléments simples en caractères sans sérif. Les éléments composés sont entre < et >. L'étoile $\star$ signifie la répétition d'un élément et l'étoile $*$ signifie le produit cartésien.

<type> : :=<type-base>
ou <type-iter>
ou <type-var>
ou (<type-fonc>)
ou `Tuple[`<type>*`]`

<type-base> : := `int` *ou* `float`
ou `bool` *ou* `None`

<type-iter> : := `str` *ou* `List[`<type>`]`
ou `Set[`<type>`]`
ou `Dict[`<type>`,`<type>`]`

<type-var> : := `T` *ou* `U` *etc.*

— **Grammaire du langage** —

<prog> : := <definition>
ou <expression>
ou <affectation>
ou <alternative>
ou <boucle-while>
ou <boucle-for>
ou <test>
ou <sequence>

<définition> : := `def` nom-fonc `(`<args>`)->`<type>`:`
<prog>
`return` <expression>

<args> : := variable `:`<type>
ou variable `:`<type> `,` <args>

<expression> : := variable *ou* constante
ou <application>
ou <op-un> <expression>
ou <expression> <op-bin> <expression>

<op-un> : := `- not`

<op-bin> : := `+ - * / == != <= >= % // **`
`and or`

<application> : := nom-fonc`(`<argument>`)`

<argument> : := <expression>
ou <expression> `,` <argument>

<affectation> : := variable `=` <expression>

<alternative> : := `if` <expression> `:`
<prog>
ou `if` <expression> `:`
<prog>
<alternant>

<alternant> : := `else :`
<prog>
ou `elif` <expression> `:`
<prog>
ou `elif` <expression> `:`
<prog>
<alternant>

<boucle-while> : := `while` <expression> `:`
<prog>

<boucle-for> : := `for` variable `in` <expression> `:`
<prog>

<test> : := `assert` <expression>

<sequence> : := <prog>
<prog>

— **Commentaires** —

Ligne commençant par un dièse (`#`) au moins.

— **Vocabulaire** —

Mots-clefs réservés : `# ( ) ’ + - * / = < > % " ! and or not if elif else while for in assert import def return`

constante : *les booléens* `True False`*, les nombres, les chaînes de caractères,* `None`

variable, nom-fonc : *tout ce qui n'est pas constante ni réservé*

— **Spécification et signature de fonction** —

`def` nom-fonc `(`<args>`)->`<type>`:`
`"""` `Précondition:` *texte*
`texte descriptif` `"""`

— **Opérations booléennes** —

Les opérateurs suivants travaillent sur des expressions de type `bool` et renvoient une valeur de type `bool`.

`not b` : *rend la négation de b*
`a and b` : *rend la conjonction de a et b*
`a or b` : *rend la disjonction de a et b*

— **Opérations sur les valeurs** —

Les opérateurs suivants travaillent sur des expressions de type `Valeur` et renvoient une valeur de type `bool`.

`a == b` : *vérifie que a et b sont égaux*
`a != b` : *vérifie que a et b ne sont égaux*
`a >= b` : *vérifie que a est plus grand ou égal à b*
`a > b` : *vérifie que a est strictement plus grand que b*
`a <= b` : *vérifie que a est inférieur ou égal à b*
`a < b` : *vérifie que a est strictement inférieur à b*

— **Opérations sur les nombres** —

Les opérateurs suivants travaillent sur des expressions de type `float` et renvoient une valeur de type `float`.

`a + b` : *effectue l'addition de a et de b*
`a - b` : *effectue la soustraction de a par b*
`a * b` : *effectue la multiplication de a par b*
`a ** b` : *effectue l'exponentiation de a par b*
`a / b` : *effectue la division (réelle) de a par b*
produit une erreur lorsque b est égal à 0

— **Opérations sur les entiers** —

Les opérateurs suivants travaillent sur des expressions de type `int` et renvoient une valeur de type `int`.

`a // b` : *effectue la division euclidienne de a par b*
produit une erreur lorsque b est égal à 0
`a % b` : *rend le reste de la division euclidienne de a par b*
produit une erreur lorsque b est égal à 0

— **Fonctions arithmétiques** —

`min(a : float, b : float,...) -> float :`
rend le plus petit des arguments
`max(a : float, b : float,...) -> float :`
rend le plus grand des arguments
`abs(x : float) -> float :`
rend la valeur absolue de x

— **Fonctions du module** `math` —

`sqrt(x : float) -> float :`
Précondition : x<0
rend la valeur de $\sqrt{x}$
`cos(x : float) -> float`
rend le cosinus de x (en radian)
`sin(x : float) -> float`
rend le sinus de x (en radian)

— Fonctions de conversions —

`int(x : T) -> int`
convertit x en un entier
`float(x : T) -> float`
convertit x en un flottant
`str(x : T) -> str`
convertit x en une chaîne de caractères

— Manipulation des chaînes de caractères —

Dans ce qui suit, les variables `s` et `t` sont de type `str` et les variables `i`, `j` et `k` sont de type `int`. Les expressions suivantes sont toutes de type `str`.

`s + t`	*effectue la concaténation de* `s` *avec* `t`
`s[i]`	*rend le* `i` *ème caractère de* `s`
`s[i:j]`	*rend la chaîne composée des caractères de* `s` *de l'indice* `i` *à l'indice* `j-1`
`s[i:j:k]`	*rend la chaîne composée des caractères de* `s` *de l'indice* `i` *à l'indice* `j-1` *par pas de* `k` *caractères*
`len(s)` :	*rend le nombre de caractères dans* `s`

— Manipulation des listes —

Dans ce qui suit, les variables `L` et `P` sont de type `List[T]`, les variables `i`, `j` et `k` sont de type `int` et les variables `x` et `y` sont de type `T`.

`[ ]`	`List[T]`
	rend la liste vide
`[x, y, ...]`	`List[T]`
	rend la liste contenant `x` *,* `y`*, ...*
`L.append(x)`	`None`
	ajoute `x` *à la fin de la liste* `L`
`L + P`	`List[T]`
	effectue la concaténation de `L` *avec* `P`
`L[i]`	`T`
	rend le `i` *ème élément de* `L`
	produit une erreur si l'indice `i` *n'est pas valide*
`L[i:j]`	`List[T]`
	rend la liste des éléments de `L` *de l'indice* `i` *à l'indice* `j-1`
`L[i:j:k]`	`List[T]`
	rend la liste des éléments de `L` *de l'indice* `i` *à l'indice* `j-1` *par pas de* `k` *éléments*
`len(L)` :	`int`
	rend le nombre d'éléments de `L`

— Manipulation des n-uplets —

Dans ce qui suit, la variable `C` est de type `Tuple[T1, T2, ...]`, la variable `i` est de type `int` et les variables `x`, `y`, ... sont de type `T1`, `T2`, ...

`(x, y, ...)`	`Tuple[T1, T2, ...]`
	rend le n-uplet contenant `x` *,* `y`*, ...*
`x, y, ... = C`	`None`
	affecte à `x` *le 1er élément de* `C`*,* *à* `y` *le second élément de* `C`*, ...*

— Manipulation des ensembles —

Dans ce qui suit, les variables `Si` sont de type `Set[T]`, et les variables `x` et `y` sont de type `T`.

`set()` :	`Set[T]`
	rend l'ensemble vide
`{x, y, ...}`	`Set[T]`
	rend l'ensemble contenant les valeurs `x` *,* `y` *, ...*
`S.add(x)`	`None`
	ajoute `x` *à l'ensemble* `S`
`S1 | S2`	`Set[T]`
	rend l'union de `S1` *avec* `S2`
`S1 & S2`	`Set[T]`
	rend l'intersection de `S1` *avec* `S2`
`S1 - S2`	`Set[T]`
	rend l'ensemble des éléments de `S1` *qui ne sont pas dans* `S2`
`S1 ^ S2`	`Set[T]`
	rend l'ensemble des éléments qui sont soit dans `S1`*, soit dans* `S2` *mais pas dans les deux*
`S1 <= S2`	`bool`
	teste si `S1` *est un sous-ensemble de* `S2`
`S1 >= S2`	`bool`
	teste si `S1` *est un sur-ensemble de* `S2`
`x in S`	`bool`
	teste si `x` *est un élément de* `S`
`len(S)` :	`int`
	rend le nombre d'éléments de `S`

— Manipulation des dictionnaires —

Dans ce qui suit, la variable `D` est de type `Dict[T, U]`, la variable `k` est de type `T` et la variable `v` est de type `U`.

`dict()`	`Dict[T, U]`
	rend le dictionnaire vide
`{k:v,l:w,...}`	`Dict[T, U]`
	rend le dictionnaire associant `v` *à la clé* `k` *,* `w` *à la clé* `l` *, ...*
`D[k] = v`	`None`
	associe la valeur `v` *à la clé* `k` *de* `D`
`D[k]`	`U`
	rend la valeur associée à la clé `k` *de* `D`
	produit une erreur si la clé `k` *n'existe pas*
`k in D`	`bool`
	teste si `k` *est une clé de* `D`
`len(D)` :	`int`
	rend le nombre d'associations dans `D`

Les itérations sur un dictionnaire peuvent se faire
par clefs : `for k in D`,
ou par associations : `for (k,v) in D.items()`

— Schémas de compréhension —

Les schémas de compréhension suivants permettent de créer respectivement des valeurs de type `List[T]`, `Set[T]` et `Dict[T,U]` :

```
[expr for var in iterable if predicat]
{expr for var in iterable if predicat}
{expr1:expr2 for var in iterable if predicat}
```

où

`expr` et `expr1` sont des expressions de type `T`
`expr2` est une expression de type `U`
`var` est une variable
`iterable` est une expression de type <type-iter>
`predicat` est une expression de type `bool`

— Fonctions diverses —

`range(n : int, p : int) -> range`
rend l'intervalle des entiers compris entre n et p
`ord(c : str) -> int`
renvoie l'entier correspondant au code de `c`
`chr(i : int) -> str`
renvoie le caractère correspondant au code `i`
`help(f) -> None`
affiche la documentation de la fonction f
`type(x) -> <type>`
rend le type de x
`print(x) -> None`
**** Procédure ****
affiche la représentation de la valeur x