

— 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 `*` 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 Number
              ou bool ou NoneType

<type-iter> ::= str ou range ou list[<type>]
              ou set[<type>]
              ou dict[<type>:<type>]

<type-var> ::= alpha ou beta etc.

<type-fonc> ::= -> <type>
              ou <type-args> -> <type>

<type-args> ::= <type>
              ou <type> * <type-args>
              ou <type> * ...
              ou <type>^n
avec          n = 1, 2, 3, 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 (<nom-args>) :
                <prog>
                return <expression>

<nom-args> ::= variable
              ou variable , <nom-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 (<nom-args>) :
    """ <type-fonc>
        HYPOTHESE texte
        texte explicatif """
```

— 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 `Number` et renvoient une valeur de type `Number`.

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

— Fonctions du module math —

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

— Fonctions de conversions —

int(x) : $\alpha \rightarrow \text{int}$
convertit x en un entier

float(x) : $\alpha \rightarrow \text{float}$
convertit x en un flottant

str(x) : $\alpha \rightarrow \text{str}$
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[α]**, les variables **i**, **j** et **k** sont de type **int** et les variables **x** et **y** sont de type α .

[] **list[α]**
rend la liste vide

[x, y, ...] **list[α]**
rend la liste contenant x, y, ...

L.append(x) **NoneType**
ajoute x à la fin de la liste L

L + P **list[α]**
effectue la concaténation de L avec P

L[i] α
rend le i ème élément de L
produit une erreur si l'indice i n'est pas valide

L[i:j] **list[α]**
rend la liste des éléments de L de l'indice i à l'indice j-1

L[i:j:k] **list[α]**
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[α, β, \dots]**, la variable **i** est de type **int** et les variables **x**, **y**, ... sont de type α , β , ...

(x, y, ...) **tuple[α, β, \dots]**
rend le n-uplet contenant x, y, ...

x, y, ... = C **NoneType**
affecte à x la 1ère coordonnée de C,
à y la 2nde coordonnée de C, ...

— Manipulation des ensembles —

Dans ce qui suit, les variables **S** et **T** sont de type **set[α]**, et les variables **x** et **y** sont de type α .

set() : **set[α]**
rend l'ensemble vide

{x, y, ...} **set[α]**
rend l'ensemble contenant les valeurs x, y, ...

S.add(x) **NoneType**
ajoute x à l'ensemble S

S | T **set[α]**
rend l'union de S avec T

S & T **set[α]**
rend l'intersection de S avec T

S - T **set[α]**
rend l'ensemble des éléments de S qui ne sont pas dans T

S ^ T **set[α]**
rend l'ensemble des éléments qui sont soit dans S, soit dans T mais pas dans les deux

S <= T **bool**
teste si S est un sous-ensemble de T

S >= T **bool**
teste si S est un sur-ensemble de T

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[$\alpha : \beta$]**, la variable **k** est de type α et la variable **v** est de type β .

dict() **dict[$\alpha : \beta$]**
rend le dictionnaire vide

{k:v, l:w, ...} **dict[$\alpha : \beta$]**
rend le dictionnaire associant v à la clé k, w à la clé l, ...

D[k] = v **NoneType**
associe la valeur v à la clé k de D

D[k] β
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

— Schémas de compréhension —

Les schémas de compréhension suivants permettent de créer respectivement des valeurs de type **list[α]**, **set[α]** et **dict[$\alpha : \beta$]** :

[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 α

expr2 est une expression de type β

var est une variable

iterable est une expression de type **<type-iter>**

predicat est une expression de type **bool**

— Fonctions diverses —

range(n,p) : **int * int -> range**
rend la séquence 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) : **($\alpha * \beta * \dots \rightarrow \gamma$) -> NoneType**
affiche la documentation de la fonction f

type(x) : $\alpha \rightarrow \text{<type>}$
rend le type de x

print(x) : $\alpha \rightarrow \text{NoneType}$
affiche la valeur de x