Chapitre 2

Mots réservés et fonctions internes

2.1 Mots réservés

Python 3 possède 33 mots réservés :

and as assert break	class continue def del	elif else except	False Finally for from global	if import in is lambda	None nonlocal not or
pass raise return	True try	while with	yield		

On ne peut pas nommer une variable avec un mot réservé :

```
>>> global = 2300
SyntaxError: invalid syntax
```

2.2 Fonctions internes

Python 3 possède 72 fonctions internes (*built-in functions*) listées ci-après.

```
abs           bin         callable     delattr
all           bool        chr          dict
any           bytearray   classmethod  dir
ascii         bytes       compile      divmod
                          complex
                          copyright
                          credits

enumerate     filter      getattr      hasattr
eval          float       globals      hash
exec          format                   help
exit          frozenset                hex

id            len         map          next
input         license     max
int           list        memoryview
isinstance    locals      min
issubclass
iter

object        pow         quit         range
oct           print                    repr
open          property                 reversed
ord                                    round

set           tuple       vars         zip
setattr       type
slice
sorted
staticmethod
str
sum
super
```

Comme avec la plupart des langages, l'*appel* à une fonction se fait avec des parenthèses (à l'intérieur desquelles on fera passer les arguments attendus) :

```
>>> abs(-56)
56
>>> int(4.0089)
```

```
4
>>> sum([4,6,11])
22
```

Note. Les fonctions `quit`, `exit`, `copyright`, `license` et `credits` ne sont pas utilisées pour programmer. On laisse au lecteur le soin de les tester.

2.3 Module `builtins`

Voici une manipulation (expliquée au chapitre 30) permettant de générer la liste des fonctions internes. On commence par importer le module `builtins` qui contient toutes les fonctions internes :

```
>>> import builtins
```

On pourrait afficher la liste `dir(builtins)` des attributs de `builtins`, mais cette dernière contient 149 items alors que seulement 72 d'entre eux nous intéressent. On va donc filtrer à l'aide de trois fonctions : on va jeter les attributs qui ne sont pas appelables, les noms qui commencent par un underscore, et les noms qui contiennent des lettres capitales :

```
>>> def app(chaine):
        x = getattr(builtins,chaine)
        return callable(x)
>>> def su(chaine):
        return chaine[0]!="_"
>>> capitales = set("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
>>> def sc(chaine):
        return ( capitales & set(chaine) == set() )
>>> L = dir(builtins)
>>> [x for x in L if app(x) and sc(x) and su(x)]
['abs', 'all', 'any', 'ascii', 'bin',
'bool', 'bytearray', 'bytes', 'callable',
'chr', 'classmethod', 'compile', 'complex',
'copyright', 'credits', 'delattr', 'dict',
'dir', 'divmod', 'enumerate', 'eval', 'exec',
'exit', 'filter', 'float', 'format', 'frozenset',
'getattr', 'globals', 'hasattr', 'hash', 'help',
```

```
'hex', 'id', 'input', 'int', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list',
'locals', 'map', 'max', 'memoryview', 'min', 'next',
'object', 'oct', 'open', 'ord', 'pow', 'print',
'property', 'quit', 'range', 'repr', 'reversed',
'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str', 'sum', 'super', 'tuple',
'type', 'vars', 'zip']
```

Nous retrouvons bien les 72 fonctions internes.

2.4 Méthodes internes

Python est un *langage orienté objet* : toutes les données sont des *objets*. Chaque type d'objet (on dit chaque *classe*) possède des *attributs*. Parmi ses attributs, il y a des fonctions. Ces fonctions sont appelées *méthodes*. Les fonctions attribuées à `str`, par exemple, s'appellent les méthodes de `str`. Pour lister les attributs de `str`, on appelle `dir(str)` :

```
>>> L = dir(str)
>>> len(L)
76
```

Ainsi, chaque chaîne de caractères possède 76 attributs. Parmi eux : 1 attribut-donnée et 75 méthodes. Parmi ces 75 méthodes, 31 méthodes spéciales (*wrappers*) et 44 méthodes ordinaires. On peut filtrer la liste `dir(str)` afin de ne garder que les méthodes ordinaires : il suffit (dans ce cas précis) de jeter tous les noms qui commencent par un underscore :

```
>>> [x for x in L if x[0]!="_"]
['capitalize', 'casefold', 'center',
'count', 'encode', 'endswith', 'expandtabs',
'find', 'format', 'format_map', 'index',
'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'maketrans',
'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip',
```

```
'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Toutes ces méthodes (appelées *méthodes internes* car elles sont définies en interne par Python) enrichissent considérablement le langage Python.

Pour programmer, nous n'utilisons que les méthodes internes ordinaires (et éventuellement des attributs ordinaires).

L'appel d'une méthode suit une syntaxe précise : si on veut appeler une méthode `maméthode` appartenant à un objet `monobjet` en faisant passer les paramètres `mesparamètres`, on écrit :

```
>>> monobjet.maméthode(mesparamètres)
```

(on applique `maméthode` à `monobjet`.)

Exemple :

```
>>> "alexandre trijoulet".count("a")
2
```

Ci-dessous, le nombre d'attributs-données (spéciaux ou ordinaires) et méthodes (spéciales ou ordinaires) par type usuel :

Type	Données spé	Données ordin	Méthodes spé	Méthodes ordin
`int`	1	4	60	4
`float`	1	2	48	5
`complex`	1	2	44	1
`range`	1	3	26	2
`bool`	1	4	60	4
`str`	1	0	31	44
`list`	2	0	32	11
`tuple`	1	0	29	2
`set`	2	0	35	17
`frozenset`	1	0	32	8
`dict`	2	0	26	11
`function`	11	0	23	0
`fichier-bin`	2	4	31	19
`fichier-txt`	3	8	30	16
`bytes`	1	0	29	38
`bytearray`	2	0	32	46

Nous verrons qu'une même méthode interne peut exister chez plusieurs types différents.

2.5 Méthodes spéciales

Il y a quatre sortes d'opérateurs en Python : les opérations numériques, les affectations-opérations, les comparaisons et les appels ou accès à items.

opérateur	syntaxe
+	x+y +x
*	x*y
-	x-y -x
/	x/y
//	x//y
%	x%y
**	x**y
&	x&y
|	x|y
^	x^y
~	~x
<<	x<<y
>>	x>>y

opérateur	syntaxe
=	x=y
+=	x+=y
*=	x*=y
-=	x-=y
/=	x/=y
//=	x//=y
%=	x%=y
**=	x**=y
&=	x&=y
|=	x|=y
^=	x^=y
<<=	x<<=y
>>=	x>>=y

opérateur	syntaxe
is	x is y
is not	x is not y
in	x in y
not in	x not in y
==	x==y
>=	x>=y.
>	x>y
<=	x<=y
<	x<y
!=	x!=y

opérateur	syntaxe
.	x.y x.y=v
[]	x[y] x[y]=v
()	x()

Derrière chaque opérateur se cache une méthode (*méthode spéciale*). Chaque opérateur sera décrit ultérieurement. Chaque méthode spéciale, à l'instar des méthodes ordinaires, possède son identificateur, et accepte la syntaxe d'appel habituelle

```
>>> monobjet.maméthode(mesparamètres)
```

même si elles ont plutôt vocation à être appelées avec le symbole d'opération correspondant. À titre culturel (et afin de faciliter l'exploration de Python), nous donnons ci-dessous le nom de chacune de ces méthodes spéciales :

opération	identifiant	utilisation	équivalent
+	__add__	x+y	x.__add__(y)
+	__radd__	y+x	x.__radd__(y)
+	__pos__	+x	x.__pos__()
*	__mul__	x*y	x.__mul__(y)
*	__rmul__	y*x	x.__rmul__(y)
-	__sub__	x-y	x.__sub__(y)
-	__rsub__	y-x	x.__rsub__(y)
-	__neg__	-x	x.__neg__()
/	__truediv__	x/y	x.__truediv__(y)
/	__rtruediv__	y/x	x.__rtruediv__(y)
//	__floordiv__	x//y	x.__floordiv__(y)
//	__rfloordiv__	y//x	x.__rfloordiv__(y)
%	__mod__	x%y	x.__mod__(y)
%	__rmod__	y%x	x.__rmod__(y)
**	__pow__	x**y	x.__pow__(y)
**	__rpow__	y**x	x.__rpow__(y)
&	__and__	x&y	x.__and__(y)
&	__rand__	y&x	x.__rand__(y)
|	__or__	x|y	x.__or__(y)
|	__ror__	y|x	x.__ror__(y)
^	__xor__	x^y	x.__xor__(y)
^	__rxor__	y^x	x.__rxor__(y)
~	__invert__	~x	x.__invert__()
<<	__lshift__	x<<y	x.__lshift__(y)
<<	__rlshift__	y<<x	x.__rlshift__(y)
>>	__rshift__	x>>y	x.__rshift__(y)
>>	__rrshift__	y>>x	x.__rrshift__(y)

opération	identifiant	utilisation	équivalent
=		x=y	
+=	__iadd__	x+=y	x.__iadd__(y)
*=	__imul__	x*=y	x.__imul__(y)
-=	__isub__	x-=y	x.__isub__(y)
/=	__itruediv__	x/=y	x.__itruediv__(y)
//=	__ifloordiv__	x//=y	x.__ifloordiv__(y)
%=	__imod__	x%=y	x.__imod__(y)
**=	__ipow__	x**=y	x.__ipow__(y)
&=	__iand__	x&=y	x.__iand__(y)
|=	__ior__	x|=y	x.__ior__(y)
^=	__ixor__	x^=y	x.__ixor__(y)
<<=	__ilshift__	x<<=y	x.__ilshift__(y)
>>=	__irshift__	x>>=y	x.__irshift__(y)

opération	identifiant	utilisation	équivalent
is		x is y	
is not		x is not y	
in	__contains__	x in y	y__contains__(x)
not in		x not in y	
==	__eq__	x==y	x.__eq__(y)
>=	__ge__	x>=y	x.__ge__(y)
>	__gt__	x<y	x.__gt__(y)
<=	__le__	x<=y	x.__le__(y)
<	__lt__	x<y	x.__lt__(y)
!=	__ne__	x!=y	x.__ne__(y)

opération	identifiant	utilisation	équivalent
.	__getattribute__	x.y	x.__getattribute__("y")
. avec =	__setattr__	x.y=v	x.__setattr__("y",v)
[]	__getitem__	x[y]	x.__getitem__(y)
[] avec =	__setitem__	x[y]=v	x.__setitem__(y,v)
()	__call__	x()	x.__call__()

Exemple :

```
>>> (10).__add__(3)
```