Variables et types

Une notion importante pour tous les langages est la notion de variable : c'est une zone mémoire dans laquelle une donnée est sauvegardée. Par exemple, lorsque l'on tape la commande `a = 1`, Python va enregistrer la valeur 1 dans une zone mémoire pour pouvoir ensuite la réutiliser (c'est peu intéressant pour 1 mais ça le serait déjà plus pour une approximation précise de π). Par exemple pour l'instruction `2 + a`, la console va remplacer `a` par sa valeur, ici 1, pour le calcul.

Pour qu'un langage puisse ranger convenablement une variable en mémoire, c'est-à-dire prévoir la bonne taille mémoire nécessaire, l'écrire puis la relire lorsque cela sera demandé, il est nécessaire que le langage connaisse le type de la variable. Pour comprendre cette notion (voir https://fr.wikipedia.org/wiki/Type_(informatique (https://fr.wikipedia.org/wiki/Type_(informatique))), on peut faire l'analogie suivante : pour que le langage puisse transformer une valeur donnée par l'utilisateur en une valeur lisible par l'ordinateur (écrite en binaire) il lui faut un dictionnaire. Le langage choisira un dictionnaire différent lorsque la valeur est un entier, une chaine de caractères ou un nombre décimal. Le type d'une variable permet au langage de choisir le dictionnaire.

Il y a différents types classiques : `int`, `float`, `complex`, `str` ... Afin de connaître le type d'une variable, on peut utiliser la fonction `type`. Python utilise un typage dynamique fort : les variables sont typées, mais, sauf précision contraire, le typage est déterminé automatiquement. Le type d'une variable peut être adapté dynamiquement en fonction des instructions.

Exemple :

In [1]:
```
a = 1.        # a est un entier (integer) egale à 1
a = 0.1 * a   # a est converti (cast) en nombre decimal (float) appr
ochant 0.1
```

Les entiers

Les entiers peuvent être représentés par des objets de type `int`. En python, il n'y a pas de plus grand entier, c'est-à-dire que la taille de la boîte qui stocke un entier peut être aussi grande que la place libre dans l'ordinateur...

Voici un petit code pour exemple :

In [13]:
```
a = 1
print(type(a))
b = 1000
b = b ** b  # b vaut 1000 ^ 1000 = 10^3000
print(b)
```

```
<class 'int'>
100000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000
```

In [14]:
```
b = b*1.
```

```
---------------------------------------------------------------------
---------
OverflowError                                Traceback (most recent c
all last)
<ipython-input-14-f5c29e597ca0> in <module>
----> 1 b = b*1.

OverflowError: int too large to convert to float
```

Les réels

On peut représenter les nombres réels par des objets de type `float` . Il y a donc un plus grand réel que peut supporter la machine, ainsi qu'un plus petit réel supérieur à 0 !

Attention : les calculs ne sont pas exacts !

In [15]:
```
a = 1.
print(type(a))
```

```
<class 'float'>
```

Il est possible de définir un entier en utilisant une notation classique de type ingénieur :

In [25]:
```
# erreur en utilisant des floats
a = 1.e15
b = a+1
print(b)
a = 1.e16
b = a+1
print(b)

# exact quand on prend des entiers
a = 10000000000000000
b = a+1
print(b)
```

```
1000000000000001.0
1e+16
10000000000000001
```

Les booléens

La notion de booléens est une notion essentielle en informatique. Cela correspond à une variable qui peut prendre deux états : vrai et faux. En python les booléens sont `True` et `False` .

Ils sont souvent utilisés en combinaison des tests (Cf C02)

In [27]:
```
vrai, faux = True, False
print(type(vrai))
print(type(faux))
```

```
<class 'bool'>
<class 'bool'>
```

Les opérations sur les booléens correspondent aux opérations logiques :

- *et* correspond à l'opérateur `and`
- *ou* correspond à l'opérateur `or`
- *non* correspond à l'opérateur `not`

In [29]:
```
print(vrai and faux)
print(vrai or faux)
print(not vrai)
print(not faux)
```

```
False
True
False
True
```

On rappelle les règles élémentaires

- non (A ou B) = (non A) et (non B)
- non (A et B) = (non A) ou (non B)

In [32]:
```
print(not (vrai or faux))
print(not (vrai and faux))
```

```
False
True
```

Il est possible de créer des booléens à partir des opérateurs de comparaison. Par exemple

In [2]:
```
x = 1
y = 1.0
print(x == y)  # opérateur d'égalité
```

```
True
```

In [4]:
```
x, y = 1, 2
print(x > y)   # opérateur supérieur
print(x != y)  # opérateur de différence
```

```
False
True
```

Exercice

Afin de visualiser un problème de tous les calculs faits par un ordinateur, exécutez la suite d'instructions suivantes :

- affectez 0.1 à une variable appelée `x`
- ajoutez 0.2 à la variable `x`
- affectez 0.3 à une variable appelée `y`
- affichez les variables `x` et `y` puis le booléen `x==y`

Que remarquez-vous ?

In [ ]:
```
```

Les chaînes de caractères

Les chaînes de caractères peuvent être représentées par des objets de la classe `str` . Elles sont vues comme des listes de caractères et toutes les opérations sur les listes s'appliquent aux objets de type `str` .

Il y a un certain nombre de fonctions supplémentaires qui peuvent être bien pratiques...

In [33]:
```
help(str)
```

```
Help on class str in module builtins:
```

```
class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding
or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handle
r.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __eq__(self, value, /)
 |      Return self==value.
 |
 |  __format__(self, format_spec, /)
 |      Return a formatted version of the string as described by f
ormat_spec.
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getitem__(self, key, /)
 |      Return self[key].
 |
 |  __getnewargs__(...)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __iter__(self, /)
 |      Implement iter(self).
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
```

```
 |  __len__(self, /)
 |      Return len(self).
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __repr__(self, /)
 |      Return repr(self).
 |
 |  __rmod__(self, value, /)
 |      Return value%self.
 |
 |  __rmul__(self, value, /)
 |      Return value*self.
 |
 |  __sizeof__(self, /)
 |      Return the size of the string in memory, in bytes.
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  capitalize(self, /)
 |      Return a capitalized version of the string.
 |
 |      More specifically, make the first character have upper cas
e and the rest lower
 |      case.
 |
 |  casefold(self, /)
 |      Return a version of the string suitable for caseless compa
risons.
 |
 |  center(self, width, fillchar=' ', /)
 |      Return a centered string of length width.
 |
 |      Padding is done using the specified fill character (defaul
t is a space).
 |
 |  count(...)
 |      S.count(sub[, start[, end]]) -> int
 |
 |      Return the number of non-overlapping occurrences of substr
ing sub in
 |      string S[start:end].  Optional arguments start and end are
```

```
 |      interpreted as in slice notation.
 |
 |  encode(self, /, encoding='utf-8', errors='strict')
 |      Encode the string using the codec registered for encoding.
 |
 |      encoding
 |        The encoding in which to encode the string.
 |      errors
 |        The error handling scheme to use for encoding errors.
 |        The default is 'strict' meaning that encoding errors rai
se a
 |        UnicodeEncodeError.  Other possible values are 'ignore',
'replace' and
 |        'xmlcharrefreplace' as well as any other name registered
with
 |        codecs.register_error that can handle UnicodeEncodeError
s.
 |
 |  endswith(...)
 |      S.endswith(suffix[, start[, end]]) -> bool
 |
 |      Return True if S ends with the specified suffix, False oth
erwise.
 |      With optional start, test S beginning at that position.
 |      With optional end, stop comparing S at that position.
 |      suffix can also be a tuple of strings to try.
 |
 |  expandtabs(self, /, tabsize=8)
 |      Return a copy where all tab characters are expanded using
spaces.
 |
 |      If tabsize is not given, a tab size of 8 characters is ass
umed.
 |
 |  find(...)
 |      S.find(sub[, start[, end]]) -> int
 |
 |      Return the lowest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notati
on.
 |
 |      Return -1 on failure.
 |
 |  format(...)
 |      S.format(*args, **kwargs) -> str
 |
 |      Return a formatted version of S, using substitutions from
args and kwargs.
 |      The substitutions are identified by braces ('{' and '}').
 |
 |  format_map(...)
 |      S.format_map(mapping) -> str
```

```
 |
 |      Return a formatted version of S, using substitutions from
mapping.
 |      The substitutions are identified by braces ('{' and '}').
 |
 |  index(...)
 |      S.index(sub[, start[, end]]) -> int
 |
 |      Return the lowest index in S where substring sub is found,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notati
on.
 |
 |      Raises ValueError when the substring is not found.
 |
 |  isalnum(self, /)
 |      Return True if the string is an alpha-numeric string, Fals
e otherwise.
 |
 |      A string is alpha-numeric if all characters in the string
are alpha-numeric and
 |      there is at least one character in the string.
 |
 |  isalpha(self, /)
 |      Return True if the string is an alphabetic string, False o
therwise.
 |
 |      A string is alphabetic if all characters in the string are
alphabetic and there
 |      is at least one character in the string.
 |
 |  isascii(self, /)
 |      Return True if all characters in the string are ASCII, Fal
se otherwise.
 |
 |      ASCII characters have code points in the range U+0000-U+00
7F.
 |      Empty string is ASCII too.
 |
 |  isdecimal(self, /)
 |      Return True if the string is a decimal string, False other
wise.
 |
 |      A string is a decimal string if all characters in the stri
ng are decimal and
 |      there is at least one character in the string.
 |
 |  isdigit(self, /)
 |      Return True if the string is a digit string, False otherwi
se.
 |
 |      A string is a digit string if all characters in the string
are digits and there
```

```
 |      is at least one character in the string.
 |
 |  isidentifier(self, /)
 |      Return True if the string is a valid Python identifier, Fa
lse otherwise.
 |
 |      Call keyword.iskeyword(s) to test whether string s is a re
served identifier,
 |      such as "def" or "class".
 |
 |  islower(self, /)
 |      Return True if the string is a lowercase string, False oth
erwise.
 |
 |      A string is lowercase if all cased characters in the strin
g are lowercase and
 |      there is at least one cased character in the string.
 |
 |  isnumeric(self, /)
 |      Return True if the string is a numeric string, False other
wise.
 |
 |      A string is numeric if all characters in the string are nu
meric and there is at
 |      least one character in the string.
 |
 |  isprintable(self, /)
 |      Return True if the string is printable, False otherwise.
 |
 |      A string is printable if all of its characters are conside
red printable in
 |      repr() or if it is empty.
 |
 |  isspace(self, /)
 |      Return True if the string is a whitespace string, False ot
herwise.
 |
 |      A string is whitespace if all characters in the string are
whitespace and there
 |      is at least one character in the string.
 |
 |  istitle(self, /)
 |      Return True if the string is a title-cased string, False o
therwise.
 |
 |      In a title-cased string, upper- and title-case characters
may only
 |      follow uncased characters and lowercase characters only ca
sed ones.
 |
 |  isupper(self, /)
 |      Return True if the string is an uppercase string, False ot
herwise.
```

```
 |
 |      A string is uppercase if all cased characters in the strin
g are uppercase and
 |      there is at least one cased character in the string.
 |
 |  join(self, iterable, /)
 |      Concatenate any number of strings.
 |
 |      The string whose method is called is inserted in between e
ach given string.
 |      The result is returned as a new string.
 |
 |      Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'
 |
 |  ljust(self, width, fillchar=' ', /)
 |      Return a left-justified string of length width.
 |
 |      Padding is done using the specified fill character (defaul
t is a space).
 |
 |  lower(self, /)
 |      Return a copy of the string converted to lowercase.
 |
 |  lstrip(self, chars=None, /)
 |      Return a copy of the string with leading whitespace remove
d.
 |
 |      If chars is given and not None, remove characters in chars
instead.
 |
 |  partition(self, sep, /)
 |      Partition the string into three parts using the given sepa
rator.
 |
 |      This will search for the separator in the string.  If the
separator is found,
 |      returns a 3-tuple containing the part before the separator
, the separator
 |      itself, and the part after it.
 |
 |      If the separator is not found, returns a 3-tuple containin
g the original string
 |      and two empty strings.
 |
 |  replace(self, old, new, count=-1, /)
 |      Return a copy with all occurrences of substring old replac
ed by new.
 |
 |        count
 |          Maximum number of occurrences to replace.
 |          -1 (the default value) means replace all occurrences.
 |
 |      If the optional argument count is given, only the first co
```

```
unt occurrences are
 |      replaced.
 |
 |  rfind(...)
 |      S.rfind(sub[, start[, end]]) -> int
 |
 |      Return the highest index in S where substring sub is found
,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notati
on.
 |
 |      Return -1 on failure.
 |
 |  rindex(...)
 |      S.rindex(sub[, start[, end]]) -> int
 |
 |      Return the highest index in S where substring sub is found
,
 |      such that sub is contained within S[start:end].  Optional
 |      arguments start and end are interpreted as in slice notati
on.
 |
 |      Raises ValueError when the substring is not found.
 |
 |  rjust(self, width, fillchar=' ', /)
 |      Return a right-justified string of length width.
 |
 |      Padding is done using the specified fill character (defaul
t is a space).
 |
 |  rpartition(self, sep, /)
 |      Partition the string into three parts using the given sepa
rator.
 |
 |      This will search for the separator in the string, starting
at the end. If
 |      the separator is found, returns a 3-tuple containing the p
art before the
 |      separator, the separator itself, and the part after it.
 |
 |      If the separator is not found, returns a 3-tuple containin
g two empty strings
 |      and the original string.
 |
 |  rsplit(self, /, sep=None, maxsplit=-1)
 |      Return a list of the words in the string, using sep as the
delimiter string.
 |
 |        sep
 |          The delimiter according which to split the string.
 |          None (the default value) means split according to any
whitespace,
```

```
 |          and discard empty strings from the result.
 |        maxsplit
 |          Maximum number of splits to do.
 |          -1 (the default value) means no limit.
 |
 |      Splits are done starting at the end of the string and work
ing to the front.
 |
 |  rstrip(self, chars=None, /)
 |      Return a copy of the string with trailing whitespace remov
ed.
 |
 |      If chars is given and not None, remove characters in chars
instead.
 |
 |  split(self, /, sep=None, maxsplit=-1)
 |      Return a list of the words in the string, using sep as the
delimiter string.
 |
 |      sep
 |        The delimiter according which to split the string.
 |        None (the default value) means split according to any wh
itespace,
 |        and discard empty strings from the result.
 |      maxsplit
 |        Maximum number of splits to do.
 |        -1 (the default value) means no limit.
 |
 |  splitlines(self, /, keepends=False)
 |      Return a list of the lines in the string, breaking at line
boundaries.
 |
 |      Line breaks are not included in the resulting list unless
keepends is given and
 |      true.
 |
 |  startswith(...)
 |      S.startswith(prefix[, start[, end]]) -> bool
 |
 |      Return True if S starts with the specified prefix, False o
therwise.
 |      With optional start, test S beginning at that position.
 |      With optional end, stop comparing S at that position.
 |      prefix can also be a tuple of strings to try.
 |
 |  strip(self, chars=None, /)
 |      Return a copy of the string with leading and trailing whit
espace removed.
 |
 |      If chars is given and not None, remove characters in chars
instead.
 |
 |  swapcase(self, /)
```

```
 |      Convert uppercase characters to lowercase and lowercase ch
aracters to uppercase.
 |
 |  title(self, /)
 |      Return a version of the string where each word is titlecas
ed.
 |
 |      More specifically, words start with uppercased characters
and all remaining
 |      cased characters have lower case.
 |
 |  translate(self, table, /)
 |      Replace each character in the string using the given trans
lation table.
 |
 |        table
 |          Translation table, which must be a mapping of Unicode
ordinals to
 |          Unicode ordinals, strings, or None.
 |
 |      The table must implement lookup/indexing via __getitem__,
for instance a
 |      dictionary or list.  If this operation raises LookupError,
the character is
 |      left untouched.  Characters mapped to None are deleted.
 |
 |  upper(self, /)
 |      Return a copy of the string converted to uppercase.
 |
 |  zfill(self, width, /)
 |      Pad a numeric string with zeros on the left, to fill a fie
ld of the given width.
 |
 |      The string is never truncated.
 |
 |  ----------------------------------------------------------------
--------
 |  Static methods defined here:
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accura
te signature.
 |
 |  maketrans(...)
 |      Return a translation table usable for str.translate().
 |
 |      If there is only one argument, it must be a dictionary map
ping Unicode
 |      ordinals (integers) or characters to Unicode ordinals, str
ings or None.
 |      Character keys will be then converted to ordinals.
 |      If there are two arguments, they must be strings of equal
length, and
```

```
 |      in the resulting dictionary, each character in x will be m
apped to the
 |      character at the same position in y. If there is a third a
rgument, it
 |      must be a string, whose characters will be mapped to None
in the result.
```

In [57]:
```
message = "voici un message très important à lire"
print(message)
print(message.capitalize())
print(message.center(80))
print(message.find('i'))
print(message.find('z'))
```

```
voici un message très important à lire
Voici un message très important à lire
                    voici un message très important à lire
2
-1
```

In [59]:
```
print(message.isdecimal())
print("1234654".isdecimal())
print("1234654".zfill(80))
print(int("12345654"))
```

```
False
True
00000000000000000000000000000000000000000000000000000000000000000000
00000001234654
12345654
```

Les listes et les tuples

Les listes et les tuples permettent de stocker plusieurs objets (pas forcément du même type). La différence essentielle est que les listes sont *mutables* (il est possible de modifier les objets de la liste) alors que les tuples sont *non mutables* (il est impossible de modifier les objets du tuple).

La fonction `len()` permet de récupérer la taille de la liste et l'opérateur `[]` permet d'avoir accès (de lire et éventuellement d'écrire) aux éléments.

La fonction `append` permet d'ajouter un élément à la liste. Les opérateurs `+`, `*` sont également utilisables sur les listes pour concaténer.

In [68]:
```
t = (0, 1, 2, "toto", "titi", [0, 1, 2])
print(t)
print(t[0])
t[0] = 1
```

```
(0, 1, 2, 'toto', 'titi', [0, 1, 2])
0

---------------------------------------------------------------------
---------
TypeError                                 Traceback (most recent c
all last)
<ipython-input-68-3399665a0db0> in <module>
      2 print(t)
      3 print(t[0])
----> 4 t[0] = 1

TypeError: 'tuple' object does not support item assignment
```

In [69]:
```
# ATTENTION TOUTEFOIS, ON PEUT FAIRE DES TRUCS BIZARRE
print(t[5])
t[5][0] = 'z'
print(t)
```

```
[0, 1, 2]
(0, 1, 2, 'toto', 'titi', ['z', 1, 2])
```

In [70]:
```
l = [0, 1, 2, "toto", "titi"]
print(l)
print(l[0])
l[0] = t
print(l)
```

```
[0, 1, 2, 'toto', 'titi']
0
[(0, 1, 2, 'toto', 'titi', ['z', 1, 2]), 1, 2, 'toto', 'titi']
```

In [76]:
```
t = (0, 1, 2, 3, 4)
l = list(t)
print(l)
ll = l + l
print(ll)
l2 = l * 2
print(l2)
print("-"*80)
```

```
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 0, 1, 2, 3, 4]
---------------------------------------------------------------------
--------------
```

Il est possible d'avoir accès à plusieurs éléments d'un coup à l'aide des `:` .

In [98]:
```
l = [0, 1, 2, 3, 4, 5, 6]
debut_l = l[:3]
fin_l = l[3:]
print(debut_l)
print(fin_l)
print(debut_l + fin_l)

# nous avons fait des copies !!!
debut_l[0] = 10
print(debut_l)
print(l)

# on peut parcourir dans l'autre sens
print(l[::-1])

# on peut parcourir de 2 en 2 et aussi à l'envers
print(l[::2])
print(l[-1::-2])
```

```
[0, 1, 2]
[3, 4, 5, 6]
[0, 1, 2, 3, 4, 5, 6]
[10, 1, 2]
[0, 1, 2, 3, 4, 5, 6]
[6, 5, 4, 3, 2, 1, 0]
[0, 2, 4, 6]
[6, 4, 2, 0]
```

Compréhension de listes

`Python` offre des fonctions très intéressantes et très puissantes pour traiter les listes : la compréhension de liste. La syntaxe est la suivante :

```
nouvelle_liste = [
    fonction(element) for element in liste if condition(element)
]
```

In [13]:
```
l = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In [15]:
```
[lk for lk in l if lk % 2 == 0]
```

Out[15]:
```
[0, 2, 4, 6, 8]
```

In [16]:
```
[lk**2 for lk in l]
```

Out[16]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In [18]:
```
[lk%2 * lk**2 for lk in l]
```

Out[18]: [0, 1, 0, 9, 0, 25, 0, 49, 0, 81]

Exercice

> Créez la liste des puissances de 2 de 2^0 à 2^9 en utilisant la compréhension de liste

In [ ]:
```

```